Aprendizagem Automática (Machine Learning) Lecture Notes

Ludwig Krippahl, 2018. No rights reserved.

**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:

1 <book\_flight> please book me on </book\_flight> <numflt> flight twenty one </numflt>

2

3 <i\_want\_to\_go> i would like to fly </i\_want\_to\_go>

4 <city\_from> from philadelphia </city\_from> <city\_to> to dallas </city\_to> 5

6 <request1> could you please list the </request1> flights

7 <city\_from> from boston </city\_from> <city\_to> to denver </city\_to> 8 on <date> july twenty eighth </date>

1

2 *CHAPTER 1. INTRODUCTION AND COURSE OVERVIEW*

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 unla belled 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.

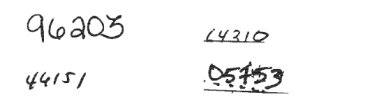
Some tasks involve sequences of decisions, such as when playing a game, and the outcome can only be assessed after all decisions are made (e.g. win or lose). This is an example of a *Reinforcement Learning* problem, where each move is not good or bad by itself but only in the context of the sequence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

With such a diverse range of applications and problems, Machine Learning benefits from con tributions 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

*1.3. FUNDAMENTAL CONCEPTS* 3 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* = *θ*, 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* = *θ* model. In this case, we would have a model with two parameters, (*x − θ*1)2 + (*y − θ*2)2 = 1. We can say that the circle of radius 1 centered at(*−*1*, −*1)is an instance of the (*x−θ*1)2+(*y−θ*2)2 = 1 model, or a hypothesis from this hypothesis class, and the line *y* = 0 is an instance of the *y* = *θ*

4 *CHAPTER 1. INTRODUCTION AND COURSE OVERVIEW *Figure 1.2: Arbitrary data set in two dimensions, divided into two classes (red and blue)

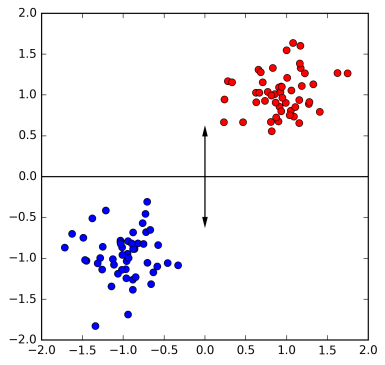
model or an hypothesis from that hypothesis class. The important point here is to distinguish between the universe of possibilities we are considering, which is the *model* or *hypothesis class*, and the specific instantiation of the model, or *hypothesis*, we obtain by setting the parameters and which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

*1.4. OVERVIEW OF MACHINE LEARNING PROBLEMS* 5 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 *unsuper vised 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 [22]. 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 [22] also report a *semi-supervised learning* approach, where the labelled examples

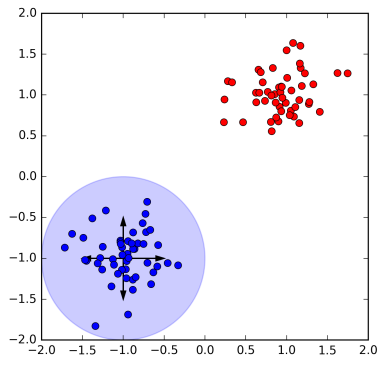
6 *CHAPTER 1. INTRODUCTION AND COURSE OVERVIEW *Figure 1.4: Another hytpothesis class: circles of radius 1 to isolate one class from the other

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.

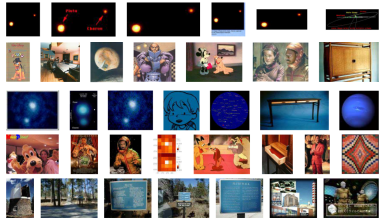
*1.5. GOALS AND COURSE OUTLINE* 7 

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

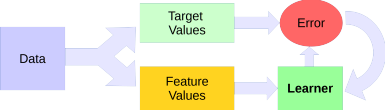
Figure 1.7: Diagram of a supervised learning process

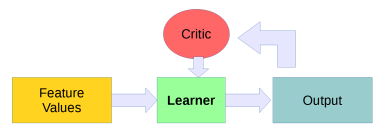


Figure 1.8: Labelled face images used in training (left) and the result of applying the classifier (right). For more details see the original paper [22].

**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

8 *CHAPTER 1. INTRODUCTION AND COURSE OVERVIEW *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 [19], Chapter 1

3. Marsland [18], Chapter 1, sections 1.1 through 1.4.

**Part I**

**Supervised Learning** 9

**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)*, ...,*(*xn, yn*)*}*, and we assume there is some unknown function *F*(*X*) : *X → Y* . Our goal is to find a function *g*(*θ, X*) : *X → Y* , which is a function of some set of parameters *θ*, that approximates the unknown *F*(*X*) : *X → 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*(*θ, 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)*, ...,*(*xn, yn*)*}* set.

2. The performance measure: training error, using *Y* .

3. The data: the *{*(*x*1*, y*1)*, ...,*(*xn, yn*)*}* 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 → 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.

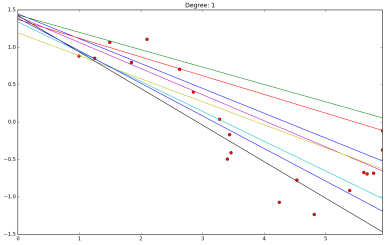
11

12 *CHAPTER 2. INTRODUCTION TO SUPERVISED LEARNING*

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* = *θ*1*x*1 + *θ*2*x*2 + *...* + *θn*+1, where each *xn* 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* = *θ*1*x* + *θ*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*) +

We want to approximate *F*(*x*) with a model *g*(*x, θ*1*, θ*2). Assuming that the error is random and normally distributed:

*∼ N*(0*, σ*2)

then, if *g*(*x, θ*1*, θ*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, θ*1*, θ*2) as:

*p*(*y|x*) *∼ N* (*g*(*x, θ*1*, θ*2)*, σ*2)

*2.2. LINEAR REGRESSION* 13

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 *θ*1*, ..., θn* 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 *X* = *{xt, yt}Nt*=1 and knowing that *p*(*x, y*) = *p*(*y|y*)*p*(*x*), then the likelihood of the set of parameters *θ* is

*l*(*θ|X* ) = Y*n t*=1

*p*(*xt, yt*) = Y*n t*=1

*p*(*yt|xt*) *×*Y*n t*=1

*p*(*xt*)

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, *L*, instead of the

likelihood *l*:

*L*(*θ|X* ) = *log*Y*n t*=1

*p*(*yt|xt*) + *log*Y*n t*=1

*p*(*xt*)

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.

*L*(*θ|X* ) *∝ log*Y*n t*=1

*p*(*yt|xt*)

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*) *∼ N* (*g*(*x, θ*)*, σ*2)

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

*N* (*z, µ, σ*) = 1

*σ~~√~~*2*πe−*(*z−µ*)2*/*2*σ*2

leaving:

*L*(*θ|X* ) *∝ log*Y*n t*=1

1

*σ~~√~~*2*πe−*[*yt−g*(*xt|θ*)]2*/*2*σ*2

which can be simplified to:

*L*(*θ|X* ) *∝ log*Y*n t*=1

*e−*[*yt−g*(*xt|θ*)]2

*L*(*θ|X* ) *∝ −*X*n t*=1

[*yt − g*(*xt|θ*)]2

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

*E*(*θ|X* ) = X*n t*=1

[*yt − g*(*xt|θ*)]2

14 *CHAPTER 2. INTRODUCTION TO SUPERVISED LEARNING*

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:

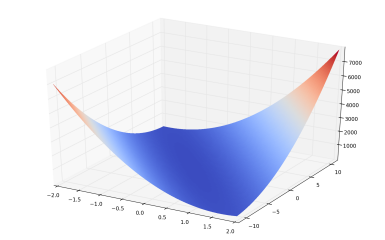
*E*(*θ|X* ) = 12X*n t*=1

[*yt − g*(*xt|θ*)]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.

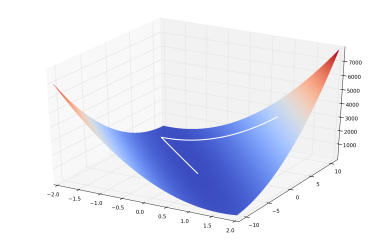
**2.3 Least Mean Squares minimization**

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.

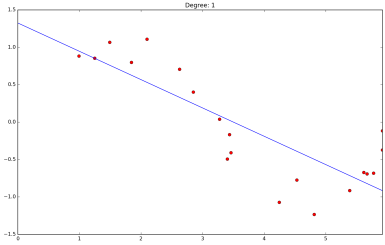
Figure 2.2: Surface of the squared error function for two parameters.

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.

*2.4. BEYOND THE STRAIGHT* 15 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

16 *CHAPTER 2. INTRODUCTION TO SUPERVISED LEARNING*

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 *X∈* = *{xt*1*, xt*2*, yt}* by making *x*1 = *x*2and *x*2 = *x* for each point in the original set. This gives us the data set represented in Figure 2.5

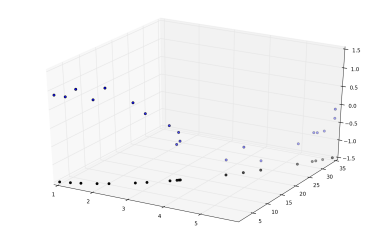


Figure 2.5: Transformed data set. Note that the black dots below are just the “shadows” to indicate the projection of the data in the (*x*2*, x*) plane.

Now our model is the equation for the plane, which we can write as *y* = *θ*1*x*1 + *θ*2*x*2 + *θ*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.

*2.4. BEYOND THE STRAIGHT* 17

Now we can convert back into our original problem. We know that our data always falls in the line where *x*1 = *x*22, 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* = *θ*1*x* + *θ*2) to that of a quadratic curve, *y* = *θ*1*x*2 + *θ*2*x* + *θ*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,

18 *CHAPTER 2. INTRODUCTION TO SUPERVISED LEARNING*

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

*2.7. FURTHER READING* 19 3. Marsland [18], 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 el ements of some universe *U* from their observable features *X*. The data is a labelled subset of *U*, *{*(*x*1*, y*1)*, ...,*(*xn, yn*)*}*, 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 *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 *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* gives us an estimate of the *generalization error*, which is the difference between the *true error* and the *training error*. This 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

21

22 *CHAPTER 3. OVERFITTING IN LINEAR REGRESSION *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.

while the *test error* is the error measured on the sample of points we assigned to the test set. The *test error* is an unbiased estimator for the *true error* because, assuming the data set is a random sample of *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

*3.2. UNDERFITTING AND OVERFITTING* 23

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 *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

24 *CHAPTER 3. OVERFITTING IN LINEAR REGRESSION *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.

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.

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

*3.4. REGULARIZATION* 25

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

26 *CHAPTER 3. OVERFITTING IN LINEAR REGRESSION*

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*(*θ*) = X*n t*=1

*yt − g*(*xt|θ*) 2+ *λ*X*m j*=1

*θ*2*j*

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 20031. 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:

1 **def** random\_split(data,test\_points):

2 *"""return two matrices splitting the data at random*

3 *"""*

4 ranks = np.arange(data.shape[0])

5 np.random.shuffle(ranks)

6 train = data[ranks>=test\_points,:]

7 test = data[ranks<test\_points,:]

8 **return** train,test

9

10 data = np.loadtxt(’life\_exp.csv’,delimiter=’\t’)

11 scale=np.**max**(data,axis=0)

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

*3.5. APPLICATION EXAMPLE* 27 Figure 3.8: Life expectancy versus per capita GDP.

12 data=data/scale

13 train, temp = random\_split(data, 90)

14 valid, test = random\_split(temp, 45)

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.

1 **def** mean\_square\_error(data,coefs):

2 *"""Return mean squared error*

3 *X on first column, Y on second column*

4 *"""*

5 pred = np.polyval(coefs,data[:,0])

6 error = np.mean((data[:,1]-pred)\*\*2)

7 **return** error

8

9 best\_err = 10000000 *# very large number*

10 **for** degree **in range**(1,9):

11 coefs = np.polyfit(train[:,0],train[:,1],degree)

12 valid\_error = mean\_square\_error(valid,coefs)

13 **if** valid\_error < best\_err:

14 best\_err = valid\_error

15 best\_coef = coefs

16 best\_degree = degree

17

18 test\_error = mean\_square\_error(test,best\_coef)

19 **print** best\_degree,test\_error

28 *CHAPTER 3. OVERFITTING IN LINEAR REGRESSION*

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



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.

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 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 polyno mial 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’)

*3.5. APPLICATION EXAMPLE* 29

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 documentation2.

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

2 http://scikit-learn.org/stable/modules/linear\_model.html

30 *CHAPTER 3. OVERFITTING IN LINEAR REGRESSION *

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.

**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, 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 *~w* perpendicular to the desired plane and a constant *w*0, the points belonging to the plane can be found by this equation:

*~wT~x* + *w*0 = 0

31

32 *CHAPTER 4. LOGISTIC REGRESSION*

Figure 4.2 shows a plane separating two sets of points and the corresponding *~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*(*~x*) = *~wT~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 *~w* and the constant *w*0.

**4.2 The wrong way: least mean squares**

Since the function *y*(*~x*) = *~wT ~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* =X*N j*=1

(*y*( *~xj* ) *− tj* )2

*4.2. THE WRONG WAY: LEAST MEAN SQUARES* 33

where *~xj* are the points in our training set and *tj*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*(*~x*) = *~wT ~x* + *w*0, we can include *w*0 in *~w* and simply add a 1 to each *~x*:

*w*e = (*w*0*, ~w*)*, x*e = (1*, ~x*)*, y*(*~x*) = *w*e*T x*e

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]:

*xnew* =*x − min*(*X*)

*max*(*X*) *− min*(*X*)

34 *CHAPTER 4. LOGISTIC REGRESSION*

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:

*xnew* =*x − µ*(*X*)

*σ*(*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.

1 **import** numpy as np

2

3 mat = np.loadtxt(input\_data,delimiter=’\t’)

4 Ys = mat[:,[-1]]

5 Xs = mat[:,:-1]

6 means = np.mean(Xs,0)

7 stdevs = np.std(Xs,0)

8 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.

1 **import** numpy as np

2

3 **def** expand\_features(X):

4 *"""append a columns of 1*

5 *"""*

6 X\_exp = np.ones((X.shape[0],X.shape[1]+1))

7 X\_exp[:,:-1] = X

8 **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.

1 **def** quad\_cost(theta,X,y):

2 *"""return error value comparing signed distance with y*

3 *theta is a column of coefficients*

4 *X is a matrix with one example per row, and a 1 in the last column* 5 *y is a vector of classes 0 or 1*

6 *"""*

7 coefs = np.zeros((**len**(theta),1))

8 coefs[:,0] = theta

9 vals = np.dot(X,coefs)

10 **return** np.mean((vals-(2\*y-1))\*\*2)

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

1 **from** scipy.optimize **import** minimize

*4.3. CLASSIFICATION BY LOGISTIC REGRESSION* 35

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 *w*e1and the features *~x* that can tell us the 1Remember that *w*e = (*w*0*, ~w*).

36 *CHAPTER 4. LOGISTIC REGRESSION*

probability of an example with features *~x* belonging to class *C*1:

*g*(*~x,w*e) = *P*(*C*1*|~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*|~x*) = *P*(*C*0*|~x*) = 1 *− P*(*C*1*|~x*)

which is equivalent to:

*ln P*(*C*1*|~x*)

1 *− P*(*C*1*|~x*)= 0

Thus, we can write

1 *− P*(*C*1*|~x*)= *ln g*(*~x,w*e)

*ln P*(*C*1*|~x*)

Rearranging, we obtain

1 *− g*(*~x,w*e)= *~wT~x* + *w*0

The function

*g*(*~x,w*e) = 1

1 + *e−*( *~wT ~x*+*w*0)

*f*(*x*) = 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 *~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*|~x*) = *P*(*C*0*|~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.

*4.4. LINEAR SEPARABILITY, REVISITED* 37

Now, given that:

*g*(*~x,w*e) = *P*(*C*1*|~x*)

the likelihood of our parameters *w*e, which is the product of the probabilities of the classes of our examples given our hypothesis, will be

*L*(*w*e*|X*) = Y*N*

*n*=1

and the logarithm of the likelihood is

*gtn*

*n*(1 *− gn*)1*−tn*

*l*(*w*e*|X*) = X*N n*=1

[*tn* ln *gn* + (1 *− tn*) ln(1 *− gn*)]

where *gn* is the result of *g*( *~xn,w*e) and *tn* 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*(*w*e) = *−*X*N n*=1

[*tn* ln *gn* + (1 *− tn*) ln(1 *− gn*)]

with

*gn* =1

1 + *e−*( *~wT ~xn*+*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,

38 *CHAPTER 4. LOGISTIC REGRESSION *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

*4.4. LINEAR SEPARABILITY, REVISITED* 39

6 X\_exp[:,-1] = X[:,0]\*X[:,1]

7 **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.

1 mat = np.loadtxt(’gene\_data\_2.txt’,delimiter=’\t’)

2 Ys = mat[:,[-1]]

3 Xs = mat[:,:-1]

4 means = np.mean(Xs,0)

5 stdevs = np.std(Xs,0)

6 Xs = (Xs-means)/stdevs

7 X\_exp = poly\_3features(Xs)

8 reg = LogisticRegression(C=1e12, tol=1e-10)

9 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.

1 **def** poly\_7features(X):

2 *"""append a five columns with the product,*

3 *square and cube of the first two features*

4 *"""*

5 X\_exp = np.zeros((X.shape[0],X.shape[1]+5))

40 *CHAPTER 4. LOGISTIC REGRESSION *

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.

*4.5. SUMMARY* 41

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 on 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*(*θ|X* ) = X*n t*=1

[*yt*˘*g*(*xt|θ*)]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:

[*tn* ln *gn* + (1 *− tn*) ln(1 *− gn*)]

with

*E*(*w*e) = *−*X*N n*=1

*gn* =1

1 + *e−*( *~wT ~xn*+*w*0)

Since *gn* 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:

43

44 *CHAPTER 5. OVERFITTING LOGISTIC REGRESSION*

*L*(*w*e) = 1*N*X*N n*=1

*H*(*pn, qn*) = *−*1*N*X[*tn* ln *gn* + (1 *− tn*) ln(1 *− gn*)]

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 *gn* and the class *t ∈* 0*,* 1. This is called the *Brier score*:

*E*(*w*e) = 1*N*X*N n*=1

[*tn − gn*]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 *gn ≥* 0*.*5 is predicted to be in class *t* = 1 and any point with *gn <* 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*:

*5.1. SCORING BINARY CLASSIFIERS* 45

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

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

*accuracy* =*true positives* + *true negatives*

*N*

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

*true positives* + *f alse positives recall* =*true positives*

*true positives* + *f alse 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*:

*F*1 =2 *× true positives*

2 *× true positives* + *f alse positives* + *f alse negatives*

*F*1 = 2*precision × recall*

*precision* + *recall*

Although the usual approach is to consider that *gn ≥* 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 *gn ≥ α, α ∈* [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.

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* curve1. 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.

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 reg = LogisticRegression()

3 reg.fit(X\_r,Y\_r)

4 test\_error = 1-reg.score(X\_t,Y\_t)

1The 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

46 *CHAPTER 5. OVERFITTING LOGISTIC REGRESSION *

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*.



Figure 5.3: A ROC curve.

**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

*5.2. CROSS-VALIDATION AND MODEL SELECTION* 47

evaluate specific hypotheses and not the models themselves. Cross-validation solves these problems. 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 **from** sklearn.utils **import** shuffle

3 mat = np.loadtxt(’dataset\_90.txt’,delimiter=’,’)

4 data = shuffle(mat)

5 Ys = data[:,0]

6 Xs = data[:,1:]

7 means = np.mean(Xs,axis=0)

8 stdevs = np.std(Xs,axis=0)

9 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 *× x*2*}, {x*1*, x*2*, x*1 *× x*2*, x*21*}, ...*, 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

48 *CHAPTER 5. OVERFITTING LOGISTIC REGRESSION*

algorithms for this that we will see later. However, the purpose of this exercise is to help explain this idea of a non-linear transformation making the classes linearly separable in the new representation, an important technique in different machine learning algorithms.

1 **def** poly\_16features(X):

2 *"""Expand data polynomially"""*

3 X\_exp = np.zeros((X.shape[0],X.shape[1]+14))

4 X\_exp[:,:2] = X

5 X\_exp[:,2] = X[:,0]\*X[:,1]

6 X\_exp[:,3] = X[:,0]\*\*2

7 X\_exp[:,4] = X[:,1]\*\*2

8 *#... rest of the expansion here*

9 **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.

1 **from** sklearn.linear\_model **import** LogisticRegression

2 **def** calc\_fold(feats, X,Y, train\_ix,test\_ix,C=1e12):

3 *"""return classification error for train and test sets"""*

4 reg = LogisticRegression(penalty=’l2’,C=C, tol=1e-10)

5 reg.fit(X[train\_ix,:feats],Y[train\_ix,0])

6 prob = reg.predict\_proba(X[:,:feats])[:,1]

7 squares = (prob-Y[:,0])\*\*2

8 **return** (np.mean(squares[train\_ix]),

9 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:

1 **from** sklearn.model\_selection **import** KFold

2 x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

3 kf = KFold(n\_splits=4)

4 **for** train, valid **in** kf.split(x):

5 **print** (train, valid)

6

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

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

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

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

*5.3. CROSS-VALIDATION AND REGULARIZATION* 49

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:

1 **from** sklearn.model\_selection **import** train\_test\_split, StratifiedKFold 2 Xs=poly\_16features(Xs)

3 X\_r,X\_t,Y\_r,Y\_t = train\_test\_split(Xs, Ys, test\_size=0.33, stratify = Ys) 4 folds = 10

5 kf = StratifiedKFold(n\_splits=folds)

6 **for** feats **in range**(2,16):

7 tr\_err = va\_err = 0

8 **for** tr\_ix,va\_ix **in** kf.split(Y\_r,Y\_r):

9 r,v = calc\_fold(feats,X\_r,Y\_r,tr\_ix,va\_ix)

10 tr\_err += r

11 va\_err += v

12 **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 regularization, as we saw before with *ridge regression*, corresponds to 1*λ*. So our regularization term in this case will be

50 *CHAPTER 5. OVERFITTING LOGISTIC REGRESSION *

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).

1

*C*

X*m j*=1

*w*2*j*

where *wj* are the coefficients for the hyperplane separating the classes.

Penalizing the hyperplane will force the coefficients of *w*e to be smaller. This affects the slope of the logistic function:

*g*(*~x,w*e) = 1

1 + *e−w*e*T ~x*

Without regularization, *w*e 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 *w*e 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:

*5.3. CROSS-VALIDATION AND REGULARIZATION* 51 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*−*5to 1015, 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 105to all the points in the training set and then testing with the test set we left out in the beginning.

52 *CHAPTER 5. OVERFITTING LOGISTIC REGRESSION *

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* = 105trained 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

s~~X~~

*Dx,x0* = *p*

*d*

53

*|xd − x0d|p*

54 *CHAPTER 6. LAZY LEARNING *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*:

*Dx,x0* =X *d*

(1*, xd 6*= *x0d* 0*, xd* = *x0d*

*6.3. EXAMPLE OF K-NN CLASSIFICATION* 55

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.

1 **import** numpy as np

2 **def** mink\_dist(x, X, p = 2):

3 *"""return p-norm values of point x distance to vector X"""*

4 sq\_diff = np.power(np.**abs**(X - x),p)

5 dists = np.power(np.**sum**(sq\_diff,1),1.0/p)

6 **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.

1 **from** scipy.stats **import** mode

2 **def** k\_nearest\_ixs(x, X,k):

3 *"""return indexes of k nearest neighbours*

4 *"""*

5 ixs = np.argsort(mink\_dist(x,X))

6 **return** ixs[:k]

7 **def** knn\_classify(x,X,Y,k):

8 *"""return class of x"""*

9 ix = k\_nearest\_ixs(x,X,k)

10 **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.

**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

56 *CHAPTER 6. LAZY LEARNING *

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.

proportion of an N-dimensional sphere occupied by another sphere whose diameter is 95% that of the first one. For high dimensions, most of the volume is at this frontier region.



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 different points in the training set according to distance is a *kernel function*. A *kernel function K*(*u*) is

*6.5. INSTANCE BASED REGRESSION* 57 Figure 6.6: K-Nearest Neighbours regression with different values of *k*.

a function that satisfies these three conditions:

Z *∞ −∞*

*K*(*u*) *≥* 0 *∀u* (6.1) *K*(*u*)*du* = 1 (6.2) *K*(*−u*) = *K*(*u*) *∀u* (6.3)

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

*K*(*u*) = 1

*~~√~~*2*πe− u*22

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:

*y*ˆ(*x*) =

P*N t*=1

*K*

*x−xt h*

*yt*

P*N*

*t*=1

or the *Priestley-Chao* estimator

*Kx−xt h*

*y*ˆ(*x*) = 1*h*X*N t*=2

(*xt − xt−*1)*K*

*x − xt h*

*yt*

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 **def** nad\_wat(K, h, X, Y, x):

5 num = 0

6 den = 0

7 **for** ix **in range**(**len**(X)):

8 yf = Y[ix]

9 u = (x-X[ix])/h

1A list of common kernel functions can be found on Wikipedia: https://en.wikipedia.org/wiki/Kernel\_ (statistics)

58 *CHAPTER 6. LAZY LEARNING*

10 k = K(u)

11 den = den + k

12 num = num + yf \* k

13 **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, leads to a much smoother estimate.

Figure 6.8: Kernel density estimation.

*6.7. SUMMARY* 59

**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 [19], Sections 8.1 and 8.2

3. Marsland [18], 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* = *xi* and *Y* = *yj*is called the *joint probability* and is represented as:

*p*(*X* = *xi, Y* = *yj* )

The probability of *X* = *xi*is the sum of the joint probabilities of all *Y* values and *X* = *xi*:

*p*(*X* = *xi*) = X*N j*=1

*p*(*X* = *xi, Y* = *yj* )

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* = *xi*) 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* = *yj* given that *X* = *xi*, written *p*(*Y* = *yj|X* = *xi*), is the proportion of *p*(*X* = *xi, Y* = *yj* ) to *p*(*X* = *xi*):

*p*(*Y* = *yj|X* = *xi*) = *p*(*X* = *xi, Y* = *yj* )

*p*(*X* = *xi*)

This means that

*p*(*X* = *xi, Y* = *yj* ) = *p*(*Y* = *yj|X* = *xi*)*p*(*X* = *xi*)

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:

61

62 *CHAPTER 7. NAÏVE BAYES*

Table 7.1: Joint and Marginal probabilities

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

*sum rule p*(*X*) = X*N j*=1

*p*(*X, Yj* )

*product rule p*(*X, Y* ) = *p*(*Y |X*)*p*(*X*)

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* ) *⇔ p*(*Y |X*)*p*(*X*) = *p*(*X|Y* )*p*(*Y* ) *⇔ p*(*Y |X*) = *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*) = *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.

*7.3. NAÏVE BAYES CLASSIFIER* 63

Since the probability of drawing example *x* does not depend on our classifier, we can simplify this expression to:

*p*(*C* = *c|X* = *x*) *∝ 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*:

*CBayes* = argmax

*p*(*C* = *c, X* = *x*)

*c∈{*0*,*1*,...,N}*

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*, ..., xn* to be the components of the feature vector and *Ck* the probability of the example being in class *k*:

*p*(*Ck, x*1*, x*2*, ..., xn*) = *p*(*Ck*)*p*(*x*1*|Ck*)*p*(*x*2*|Ck, x*1)*...p*(*xn|Ck, x*1*, x*2*, ..., xn−*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 neigh bourhood 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*, ..., xn* are *conditionally independent* given the class, it follows that:

64 *CHAPTER 7. NAÏVE BAYES*

*p*(*xn|Ck, x*1*, x*2*, ..., xn−*1) = *p*(*xn|Ck*)

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

*p*(*Ck, x*1*, x*2*, ..., xn*) = *p*(*Ck*)Y*N j*=1

*p*(*xj|Ck*)

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

ln *p*(*Ck, x*1*, x*2*, ..., xn*) = ln *p*(*Ck*) +X*N*

*j*=1

This means that our classifier can be:

ln *p*(*xj|Ck*)

*C*Naïve Bayes = argmax *k∈{*0*,*1*,...,K}*

ln *p*(*Ck*) +X*N j*=1

ln *p*(*xj|Ck*)

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:

*σk~~√~~*2*πe−*(*x−µk*)2

*p*(*xj|Ck*) = 1

2*σ*~~2~~*k*

where *µk* and *σk* are, respectively, the mean and standard deviation of the values of feature *xj* for all points in class *Ck*. 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*

*7.4. NAÏVE BAYES, EXAMPLE 1: CONTINUOUS FEATURES* 65

*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.



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.

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.

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

*C*Naïve Bayes = argmax *k∈{*0*,*1*,...,K}*

ln *p*(*Ck*) +X*N j*=1

ln *p*(*xj|Ck*)

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*.

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.

66 *CHAPTER 7. NAÏVE BAYES *

Figure 7.2: Kernel density estimation of the four distributions and the estimated joint distributions con ditioned 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.



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.

**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 poisonous1. 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. 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 1From the UCI machine learning repository: http://archive.ics.uci.edu/ml/datasets/Mushroom

*7.5. NAÏVE BAYES, EXAMPLE 2: CATEGORICAL FEATUES* 67 Figure 7.4: Cross-validation results and the final Naïve Bayes classifier for *h* = 1*.*8.

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):

68 *CHAPTER 7. NAÏVE BAYES*

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*:

*p*ˆ(*xj* = *k*) = *count*(*k*) + *α*

*N* + *αd*

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.

1 **def** make\_hists(data,features):

2 hists = []

3 **for** feat **in** features:

4 hists.append(np.ones(**len**(feat)))

5 **for** row **in range**(data.shape[0]):

6 **for** column **in range**(data.shape[1]):

7 hists[column][data[row,column]] +=1

8 **for** ix **in range**(**len**(hists)):

9 hists[ix] = np.log(hists[ix]/**float**(data.shape[0]+**len**(features[ix]))) 10 **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.

1 **def** split\_data(features,test\_fraction):

2 feat\_vals,classes = load\_data(features,’ep’)

3 edible = feat\_vals[classes==0,:]

4 poison = feat\_vals[classes==1,:]

5 e\_test\_points = **int**(test\_fraction\*edible.shape[0])

6 e\_train = edible[e\_test\_points:,:]

7 e\_test = edible[:e\_test\_points,:]

*7.5. NAÏVE BAYES, EXAMPLE 2: CATEGORICAL FEATUES* 69

8 p\_test\_points = **int**(test\_fraction\*poison.shape[0])

9 p\_train = poison[p\_test\_points:,:]

10 p\_test = poison[:p\_test\_points,:]

11 **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*(*Ck*) . This is simply the logarithm of the fraction of each class in the data. This function sums all the terms in this equation:

*C*Naïve Bayes = argmax *k∈{*0*,*1*,...,K}*

ln *p*(*Ck*) +X*N j*=1

ln *p*(*xj|Ck*)

and determines the class according to the maximum value found.

1 **def** classify(e\_class,e\_log,p\_class,p\_log,feat\_mat):

2 classes = np.zeros(feat\_mat.shape[0])

3 **for** row **in range**(feat\_mat.shape[0]):

4 e\_sum = e\_log

5 p\_sum = p\_log

6 **for** column **in range**(feat\_mat.shape[1]):

7 e\_sum = e\_sum + e\_class[column][**int**(feat\_mat[row,column])] 8 p\_sum = p\_sum + p\_class[column][**int**(feat\_mat[row,column])] 9 **if** e\_sum<p\_sum:

10 classes[row]=1

11 **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.

1 **def** do\_bayes():

2 features = get\_features()

3 e\_train,p\_train,e\_test,p\_test = split\_data(features,0.5)

4 e\_hists = make\_hists(e\_train,features)

5 p\_hists = make\_hists(p\_train,features)

6 tot\_len = e\_train.shape[0]+p\_train.shape[0]

7 e\_log = np.log(**float**(e\_train.shape[0])/tot\_len)

8 p\_log = np.log(**float**(p\_train.shape[0])/tot\_len)

9 c\_e = classify(e\_hists,e\_log,p\_hists,p\_log,e\_test)

10 c\_p = classify(e\_hists,e\_log,p\_hists,p\_log,p\_test)

11 errors = **sum**(c\_e)+**sum**(1-c\_p)

12 error\_perc = **float**(errors)/(**len**(c\_e)+**len**(c\_p))\*100

13 **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 Poisonous | 2089 221  15 1737 |

70 *CHAPTER 7. NAÏVE BAYES*

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 p*Np*0(1 *− p*0):

*7.8. PROCESSING DATA* 71

~~p~~*X − Np*0

*Np*0(1 *− p*0)*≈ 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*σ < Np*0 *< X* + 1*.*96*σ*

with *σ* =p*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:

*XLogReg* = 10 *±* 5*.*4 *XkNN* = 6 *±* 3*.*5 *XNB* = 1 *±* 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:

(*|e*01 *− e*10*| −* 1)2

*e*01 + *e*10*≈ χ*21

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 *kNN vs NB* = 2*.*3 *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.

**7.8 Processing Data**

In previous chapters we saw some data processing steps that are taken before feeding the data to machine learning algorithms: random shuffling to eliminate correlations between the position of examples and some features or target values; rescaling, either by standardization or normalization; and stratified sampling when generating test, validation or cross-validation subsets. In detail, processing the data for machine learning tasks can be a complex problem on its own and an in depth discussion of these issues would not fit the context of this course. However, there are some basic considerations that you should understand when dealing with data.

First, note that rescaling parameters are also parameters that we fit the data. In principle, if we compute these parameters, such as mean or standard deviation, on the whole data set, the error estimated

72 *CHAPTER 7. NAÏVE BAYES *

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.

on the test set may be biased because we used those examples to help rescale all data. The reason why this is not usually a problem in practice is that these parameters converge sufficiently with the number of examples and it makes little difference whether we use only the training set or the whole data set. For example, with 2000 points, the difference between the mean of the whole set or the mean of a random sample of 1000 is generally less than 2% of the standard deviation of the values. Nevertheless, you should bear in mind that rescaling parameters are also inferred from the data and, in some cases, it may be wise to compute them only with the training set if the difference is significant.

Let us consider a practical example to illustrate this. Suppose we have data on the cost of public transportation for people from different cities. The plot on Figure 7.7 shows the overall distribution of values and the breakdown per city. If we want to use these values to predict something about any random person from these cities, then we should split the data for training, cross-validation and testing using stratified sampling to ensure the same proportion of examples from each city is represented in each subset. This is illustrated in the first coloured strip in Figure 7.7. In this case, given a large enough dataset, we would not need to worry about rescaling the whole data set in one go since the rescaling parameters for each subset would be very similar to those computed on the whole data set.

Now suppose we wanted to use data from these cities to predict something about people from a different city. Now we would have to worry about how we can generalize from data about some cities to other cities. In this case, we would need to create subsets for training, cross-validation and testing that ensure that cities in the test or validation subsets are not also present in the training set, for that would lead us to underestimate the true error. This is illustrated on the second colour strip below the plot on Figure 7.7. And in this case we would have to recompute and apply the scaling parameters for each training step in cross-validation using only the training data for that step, because in this case we could not assume that the rescaling parameters would be the same across cities.

**7.9 Further Reading**

1. Bishop [4], Section 1.2

2. Alpaydin [2], Section 14.6

3. Mitchell [19], Section 6.9

4. Marsland [18], Section 8.1.2

*7.9. FURTHER READING* 73





Figure 7.7: Hypothetical distribution of transportation expenses for people in different cities. The plot shows the global distribution and the distribution per city. The coloured strips illustrate two different ways of splitting the data depending on the problem: using stratified sampling to ensure the same proportion of data from each city in each subset or ensuring that cities do not get split into training and test sets.

**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* =X*d*

*j*=1

*wjxj* + *w*0 *s*(*y*) =

(1*, y >* 0 0*, y ≤* 0

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

75

76 *CHAPTER 8. MULTI-LAYER PERCEPTRON*

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:

*wi* = *wi* + ∆*wi* ∆*wi* = *η*(*t − o*)*xi*

where *t* is the target label of the example, *o* the output of the *perceptron* for that example, *xi*the input value for feature *i* and *wi*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:

1 + *e−y*=1

*s*(*y*) = 1

1 + *e− ~wT ~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* =12X*N j*=1

(*tj − sj*)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*, *Et* =12(*tj − sj*)2, where *tj*is the class of example *j* and *sj*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* (*sj*), the activation is a function of the weighted sum of the inputs (*netj*) 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:

*−δEj*

*δw* = *−δEj δsj*

*δsj*

*δnetj*

*δnetj δw*

where

1 + *e−netj netj* = *w*0 +X

*st* =1

*M*

*i*=1

*wixi*

*8.2. A SINGLE NEURON* 77

Since*δnetj*

*δw* = *x*

*δsj*

*δnetj*= *sj*(1 *− sj*)

*δEj*

*δsj*= *−*(*tj − sj*)

We obtain the following update rule for the weight *i* of the neuron given example *j*: ∆*wji* = *−ηδEj*

*δwi*= *η*(*tj − sj*)*sj*(1 *− sj*)*xji*

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 ∆*wji*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 **??**.

78 *CHAPTER 8. MULTI-LAYER PERCEPTRON*

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.

Figure˜ref8-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.