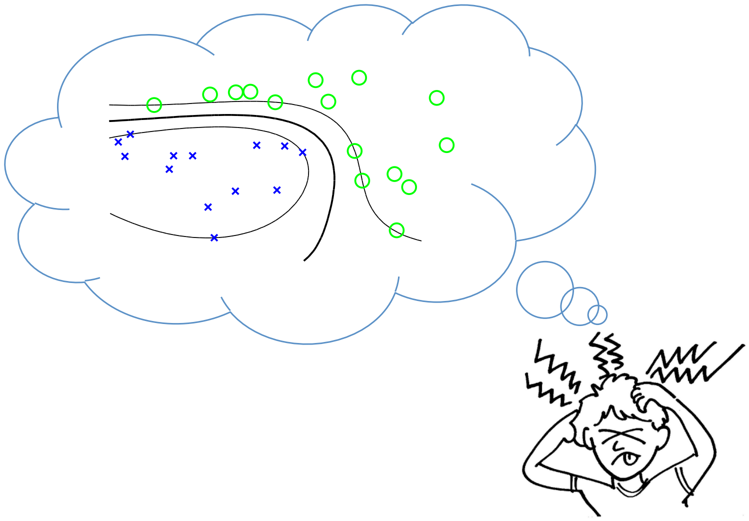
The COMP61011 Not-Very-Scary Guide to ...

Machine Learning



Professor Gavin Brown, University of Manchester (Sept 2018)

http://studentnet.cs.manchester.ac.uk/pgt/COMP61011/

Contents

Page

1 Introduction **1**
1.1 The Role of Machine Learning within Artificial Intelligence . . . 1
1.2 Machine Learning needs Data 2
1.3 Machine Learning Algorithms build a 'model' of the Data 4
1.4 Machine Learning Algorithms make mistakes 5
1.5 Mathematics is Fundamental to ML 5
1.6 Summary and Overview of the Course 6

2 Geometric Models I: Linear Models **7**
2.1 Building & Evaluating a 'model' of the data 8
2.1.1 The Problem to Solve 8
2.1.2 The Simplest Linear Model: The Decision Stump 9
2.1.3 Evaluating Models: Training vs. Testing 12
2.2 More Linear Models 14
2.2.1 The Perceptron 14
2.2.2 Logistic Regression 18
2.3 What you should know by now 23
2.4 Optional: The Geometry of Linear Models 24
2.5 Optional: Neural Networks 27
2.5.1 What is a neural network? 27

3 Experimental Methods I **29**
3.1 Reminder of Supervised Learning Protocol 30
3.2 Model Selection via Cross-Validation 31
3.3 Plotting your Results 33
3.4 Useful Trick: Rescaling Your Data 34
3.5 What you should know by know 35

4 Geometric Models II: Support Vector Machines **37**
4.1 What you need to know about SVMs 38
4.2 Decision Boundaries with Large Margins 39
4.3 SVM loss function and Learning Algorithm 40
4.4 SVMs with "Soft" Margins 42

4.5 Non-Linear SVMs : The “Kernel Trick” 45
4.5.1 The Kernel Trick 47
4.5.2 The Polynomial kernel 48
4.5.3 The RBF kernel 49
4.5.4 Model Selection with SVMs 50
4.6 What you should know by now 52
4.7 Optional: The Original SVM research paper 52

5 Distance Based Methods 63
5.1 The Nearest Neighbour Classifier 64
5.2 Handling Categorical Data 67
5.3 Comparison of k-NN to other Models 67
5.4 What you should know by now 67

6 Tree Models 69
6.1 From Decision *Stumps...* to Decision *Trees* 70
6.2 Big Trees Overfit, Little Trees Underfit 73
6.3 Dealing with Categorical Data 74
6.4 Measuring Information Gain 76
6.5 What you should know by now 81

7 Experimental Methods II: ROC Analysis 83
7.1 Evaluating Model Performance 84
7.2 What you should know by now 99

8 Probabilistic Models 101
8.1 Probability Basics 102
8.1.1 Random Variables 102
8.1.2 Estimated Probabilities vs. True Probabilities 103
8.1.3 Joint Probabilities 104
8.1.4 Conditional Probabilities 105
8.2 Bayes’ Theorem 108
8.2.1 Deriving the Theorem 108
8.2.2 Diagnosing Diabetes 109
8.2.3 Thinking about it another way 112
8.2.4 One more Bayes example: Catching Bad Guys 113
8.3 Naive Bayes Models 114
8.4 Bayesian Networks 116
8.5 What you should know by now 118

9 Committees of Models 119
9.1 Background 120
9.2 Bagging and Random Forests 121
9.3 Adaboost 123
9.4 What you should know by now 125

10 Feature Selection 127
10.1 Background . . . 128
10.2 The ‘Wrapper’ Method . . . 129
10.3 Filter Methods . . . 131
10.4 Optional: Information Theory . . . 135
10.5 What you should know by now . . . 135

11 Conclusions 137

A What is true understanding? 139

B Visualising Bayes Theorem 141

How to use these Notes

These notes are meant to provide full explanations of all material necessary to do well in this course. They are intended to be a self-contained pack — but for maximum benefit should be read alongside the slides, and alongside the suggested extra papers, from the COMP61011 website.

Please take note that to achieve the highest grades on this module, you should be thoroughly understanding everything in these notes, reading all the suggested papers, and doing some additional 'googling' of your own to find out other details. Although your January exam will only be on the material explicitly pointed to by the website/notes/slides, your mini-project in weeks 4-6 accounts for a significant amount of the overall grade (see website) so has scope for reading outside the supplied syllabus.

The website is: `http://studentnet.cs.manchester.ac.uk/pgt/COMP61011/`

Enjoy the course!

Gavin

Quotes from previous students

> *Computer Science is a hard subject.*
> *Machine Learning is one of the harder bits of CS.*

> *Machine Learning is one of those topics in CS where it's very easy to convince yourself you understand what the algorithm is doing, but in fact you don't.*

Anon

Chapter 1

Introduction

Welcome to the class! When you opted for “Foundations of Machine Learning”, what did you think it was about? Did it conjure up scenes from sci-fi movies, robots, and conscious computers? Well... I’m sorry to disappoint you, but that’s not this class. Let’s make it clear what we are studying, and why it’s just as interesting, if not more so, than those sci-fi visions of the future.

1.1 The Role of ML within AI

You’re a human being (I assume), so you can do many things. You can perceive the world around you by **seeing**, and **listening**. You can communicate by **speaking** and **gesturing**. You can **move around**, navigating your environment. You can **understand** and **reason** about abstract concepts such as jokes, politics, and prime numbers. And, you can **learn** from your experiences. Each one of these things is an ability that we associate with *intelligence.* Reproducing these abilities on a computer is known as *artificial intelligence.* For each of these abilities, there is an entire research field, with thousands of scientists working on the problem of getting computers to do it. Current popular sub-fields of artificial intelligence can be seen in the context of these human abilities:

ARTIFICIAL INTELLIGENCE

Human ability	*Name of A.I. Research Field*
Seeing	Computer vision
Talking	Speech synthesis
Listening	Speech recognition
Understanding language	Natural Language Processing
Reasoning	Automated Reasoning
Consciousness	Philosophy / Cognitive Science
Walking / moving around	Robotics
Learning	**Machine Learning**

This is not an exhaustive taxonomy — there are many other fields of A.I. which do not map so easily one-to-one with human abilities. In any case, the point to remember is that, each of these problems is so hard individually, that it has spawned careers for thousands upon thousands of scientists. In fact, behind each of these sub-fields are hundreds of sub-topics, such that most scientists can only learn maybe 10-20% of them in his/her whole career.

The abilities in the table are listed in no particular order, but notice that the last of these is **machine learning**. This means programming a computer so that it can **learn** from experience. When I say 'learn', please don't read too much into the word – it means something very specific. Learning, as we will use the word, is not something innate to biological intelligence, or even biological life. Though biology has inspired some artificial intelligence researchers, most algorithms in Machine Learning are purely mathematical with *no specific derivation from biological intelligence.* As such, the 'learning' algorithms we will meet are not magic, they are just computer programs, computational processes, like any other you have seen in the past[1]. So, when I say 'learn' what I really mean is 'adaptation in response to observed data'. The adaptation is entirely formalised, written as equations, to 'update' the state of the system from one timestep to the next, in response to data that we observe.

LEARNING

We could spend weeks (or years) debating this issue, including meaning of the word 'intelligence' — in fact some researchers spend their entire careers pursuing this line of philosophical work. Machine Learning, and many other fields in the table above, take more of an *engineering* point of view, in that we want to build computer systems to exhibit these behaviours, whether or not they are true components of human intelligence. So, let's get down to the details, and look at the 'fuel' which all machine learning algorithms need: *data*.

1.2 Machine Learning needs Data

All machine learning algorithms require one thing above all: *data*. Without data, there is nothing to learn from. In the same way, humans cannot learn without data — when you learn a new skill, your 'data' comes from observing the world around you. Luckily (for machine learning enthusiasts) the modern world is drowning in data. Amazon processes over 250,000 book sales per day. There are over 50 million credit card transactions per day in the USA alone. The UPS delivery company routes over 1,000,000 packages every night, generating hundreds of thousands of statistics on delivery times/delays etc.

Machine Learning is the study of algorithms and data structures that enable us to extract *meaning* from this mass of data. First let us consider exactly what we mean by 'data'. It is convenient to regard data as having two aspects: one is the *objects we wish to study*, and the other is the *characteristics of those objects*. For example, we might study historical credit card transactions: the 'objects' might be the individual transactions, and the characteristics are measurements

[1] Beware the person who says "I want to use machine learning to solve problem X". Very often, that just means they haven't thought hard enough about the problem. Try substituting any other field of computer science in there - "I want to use databases to solve X", doesn't sound so magic now, but it's an equivalent misnomer.

we recorded, such as the time/place/amount of the transaction, or whether it turned out to be a fraudulent transaction. In technical terms, the objects are known as *examples*, and the characteristics are known as *features*. The features could be continuous variables, such as time/amounts, or binary/categorical values, or a mixture of both. Let's take an example of some data that a doctor may encounter in a healthcare situation: the examples would be patients that the doctor had seen over a period of time, and the features would be various measurements taken on those patients. The set of measurements might be:

EXAMPLES
FEATURES

Height (cm)
Weight (kg)
Systolic blood pressure (mmHg)
Diastolic blood pressure (mmHg)
Blood sugar after meal (mM)
Diagnosed with diabetes? (1/0)

These would be taken from different patients, and arranged in a table:

x_1	x_2	x_3	x_4	x_5	y
187	80	120	30	4.5	0
160	70	119	36	5.6	0
150	80	185	60	8.8	1
192	92	140	50	6.8	1
168	110	155	45	7.8	1

Here each row is a person (one of our examples), and each column is a measurement (our features). Notice that we've given them shorthand names, so x_1 is the height, x_2 is the weight, x_3 is the systolic blood pressure, etc. Notice in particular that we've given the final measurement, the diabetes diagnosis, a different notation, y — this is what we call the *label* for each example. In this case it is binary, but in general it could be continuous or categorical. Now, given our data, we will consider two things we might want to do with it.

LABEL

Prediction : We may wish to use data to predict something about some future as yet unseen data. For example, we might use this healthcare data to predict the labels (i.e. diagnosis) for *new* patients, advising them if they are at risk of developing diabetes. The key point here is the new patients have never been seen before — so the learning system needs to be able to pick up on *general* patterns in the x data we provide, that are predictive of the disease y. These are known as *supervised* learning algorithms, since the "ground truth" labels are provided for each patient.

SUPERVISED LEARNING

Description : Some data is too complex to understand, in which case we may wish to see it *described* or visualised in a different way. For example, by visualising health data we might find that there are in fact sub-types of a disease – e.g. type-I and type-II diabetes — which need to be treated in different ways. These are known as *unsupervised* learning algorithms, as they are generally applied when we do not know the ground truth label y for each patient, and just wish to analyse patterns in the x data.

UNSUPERVISED LEARNING

The vast majority of this module will be spent on *prediction*, that is, *supervised* algorithms. There also exist so-called *unsupervised* algorithms — though we will not cover them in this module. This is not because supervised algorithms are more prevalent, or 'better', it is simply a matter of time – in this unit we have 5 weeks of taught material, and will cover the supervised methods in a lot of depth. For those of you taking COMP61021 "Modelling and Visualization of High Dimensional Data", you will see unsupervised methods there, or you may choose to study them for the project part of this module.

1.3 ML Algorithms build a 'model' of the Data

So given all that data, what do we do with it? Supervised Learning is concerned with creating and using mathematical data structures. The structures MODEL we use are known as *models*, and are expressed in various forms, e.g. trees, graphs, algebraic equations, and probability distributions. The emphasis is on constructing these models automatically from the data—where this automatic construction is referred to as *learning*. The way in which a model is used is illustrated in the figure below. Remembering the x (data) and y (labels) from the previous section, we call this the 'training' data, which is used to construct a model. Once the model is constructed, it is evaluated by testing its predictions on unseen data, which we call 'testing' data.

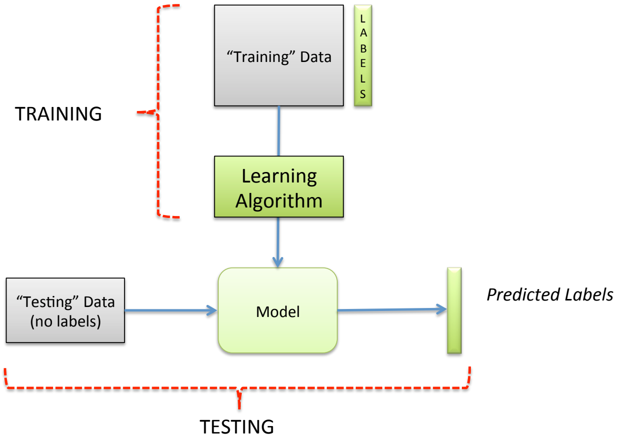


Figure 1.1: *The basic supervised learning pipeline — the* **"training" stage***: use data to build a model, then the* **"testing" stage***: evaluating the model on unseen testing data.*

This brings us to a very important aspect of machine learning, one that we will see again and again through the course. We do *not* necessarily want the model to predict the labels on the training data perfectly. This may seem non-intuitive, but what we ultimately want is for the model to make good predictions on the *testing* data. If the model learns too well on the training data, such that it can predict very well there, but cannot predict well on testing data, then it is said to have *over-fitted*. You can think about this like a student (the model) revising for an exam (the testing data). You might get past papers to practice on, but if you just memorise the papers, and memorise lecture notes, you will not necessarily have learnt the general *skills* you need. We'll see this in more detail in the coming chapters.

OVERFITTING

1.4 Machine Learning Algorithms make mistakes

An important difference between machine learning (ML) and other types of algorithms in computer science is that ML algorithms can make *mistakes*. Unlike for example, sorting algorithms, which guarantee to return the list of values you provide in order, an ML algorithm, which tries to *predict* the answer to questions you pose to it, does not *always* get it correct. Possibly even more distressing, you may feel, is the fact that some ML algorithms are *stochastic*, in that they rely on a *random* initialisation of some parameters, and *randomly* make mistakes on different data points depending on this initialisation.

You may think of this as a major disadvantage. However, consider the complexity of the task you are asking of the computer — even further, consider if a *human* could even distinguish all cases correctly. Could a doctor perfectly diagnose **everyone** who came into the surgery? Probably not, as there are always exceptions and outliers. Due to this, we need formalised ways of *measuring performance* of our different algorithms, to determine which is the 'best'. I put *best* in quotes there as it is a ambiguous word — 'best' can mean lots of things, for example an algorithm can be more accurate, or faster at learning, or takes up less memory, etc. We'll see that there are several measurements to formally evaluate the performance of our algorithms in these different ways.

1.5 Mathematics is Fundamental to ML

The language in which machine learning algorithms are phrased is *mathematics*, specifically, we will draw on many aspects of statistics, a bit of calculus, and lots of linear algebra, and probability theory. **There is no escaping this fact.** You may well take the course and try to avoid the mathematical aspects, but I promise you that you'll likely emerge with a less than satisfactory grade, likely in the bottom quarter of this class. That is the reality. If you don't know that $\log(ab) = \log a + \log b$, or the expression $\sum_i w_i x_i > t$ is alien and scares you, then maybe you should seriously reconsider taking this course. I'm not trying to scare you, or to put you off if you are a determined person. But you need to be aware of what this subject is about. Remember:

> *Computer Science is a hard subject.*
> *Machine Learning is one of the harder bits of CS.*

Be sure to look through the maths primer on the COMP61011 website, to see the type of maths we will be making use of. You can also obviously start reading ahead in these notes, to see what it's like.

1.6 Summary and Overview of the Course

So, we've established what this module is *not* about. It is not robotics, neuroscience, consciousness studies, or computer vision. However, that is not to say that Machine Learning (ML) does not slip into other fields in some way. The various sub-fields of A.I. do overlap quite a bit. For example, much of modern computer vision draws heavily on ideas that originally developed in the ML field, and there are fields such as "Text Mining" that do not correspond to a single human ability, yet draw extensively on many aspects of ML.

TEXT MINING

DATA MINING

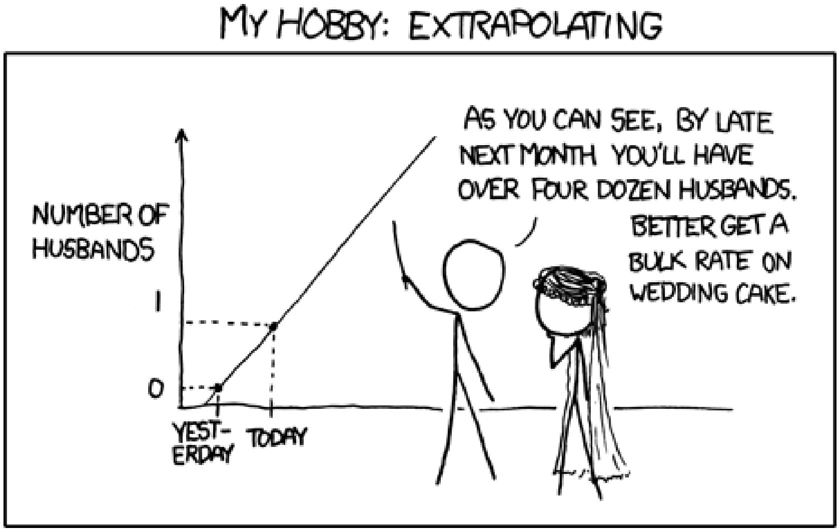
ML is also sometimes equated with the field of *data mining.* The distinction between these fields is subtle, and some would argue there is no difference at all, or that Data Mining is a subfield of ML. It is true that Data Mining is mostly concerned with analysis of real-world industry data, whereas Machine Learning has many other aspects, incorporating tools from such diverse areas as theoretical computer science, control theory, and neuroscience. Data Mining arguably uses the tools produced by machine learning, and possibly extends them to issues one would only encounter in particular industries. For the mostpart, it's just terminology, and not really worth worrying about.

In this module we will focus on the core ideas of ML, while later modules you may encounter will specialise them so you can study vision, text, speech processing, constraints in industrial settings, etc, etc.

In week 1, we will cover the basics of the *supervised learning pipeline.* The setup of this is common across all methods we will cover in the coming weeks, and you should know it well. In week 2, we will delve further into experimental methods, and geometric models. Weeks 3/4 will focus on models derived from *probabilities*, and finally in week 5 we will wrap up with some guest lectures and advanced topics that you may be able to use in your mini-projects.

Chapter 2

Geometric Models I: Linear Models


MY HOBBY: EXTRAPOLATING
NUMBER OF HUSBANDS
1
0
YEST-ERDAY
TODAY
AS YOU CAN SEE, BY LATE NEXT MONTH YOU'LL HAVE OVER FOUR DOZEN HUSBANDS.
BETTER GET A BULK RATE ON WEDDING CAKE.

2.1 Building & Evaluating a 'model' of the data

LINEAR MODEL

Machine Learning is concerned with creating and using **mathematical data structures** that allow us to make predictions about the data. The data structures we use (known as "models") are expressed in various forms, e.g. **trees, graphs, algebraic equations,** and **probability distributions**. The emphasis is on constructing these models automatically from the data—this automatic construction is referred to as *learning*. We will now meet the first of these, which are all known as **linear models**. This, like most of machine learning, will involve some mathematics, specifically this section will require some geometry and a little calculus, so you may like to revise some old textbooks. Before we get into that, we need a problem to solve with our learning algorithms.

2.1.1 The Problem to Solve

Imagine you are working for a local rugby club. Your task is to make a computer system that can distinguish between rugby players, and the ballet dancers who work next door, occasionally trying to sneak into the rugby club changing rooms. One option here is to meticulously code an algorithm that could in some way distinguish between these two types of people. However, this might not be stable to small changes in the people, such as if a rugby player grows a beard, or the ballet dancer gains some weight. Instead, we will have a machine learning algorithm *learn* the general patterns that distinguish these people, from a set of examples that we will provide to it. Your employer provides a datafile, a sample of which is shown in below.

x_1,	x_2,	y (label)
98.79,	157.59,	1
93.64,	138.79,	1
42.89,	171.89,	0
...		
...		
87.91,	142.65,	1
97.92,	162.12,	1
47.63,	182.26,	0
92.72,	154.50,	1

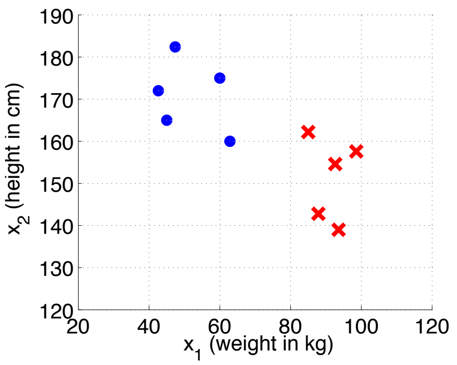


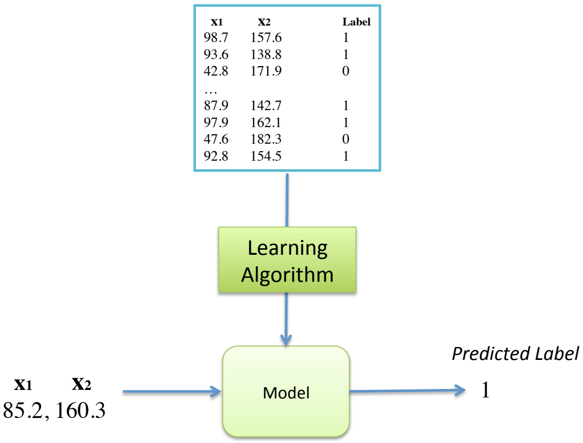
Figure 2.1: *The rugby-ballet problem. The raw data is on the left, showing the two features x_1: the person's weight in kilograms, and x_2: the person's height in centimetres. It also shows the label y, which here is either 1 or 0. The scatterplot on the right shows us a slightly nicer way to visualise the data. The rugby players ($y = 1$) are red crosses. The ballet dancers ($y = 0$) are blue dots.*

The data on the left is clearly in a 'spreadsheet' form, and a little difficult to understand, but we've scatter plotted it on the right, to see it in a different way.

Clearly, we can scatterplot data like this only if we have just two features. All of the algorithms we cover will work in general for any number of features, but visualising the behaviour in 2-d is useful to learn the basics.

MULTIPLE FEATURES

We must always keep in mind that this data is just a small fraction of what our finished model may see when it is eventually "deployed in the field" and tested for real. The supervised learning pipeline is shown in figure 2.2, showing how we make use of this data. We have access to this, and only this, and then we must make a prediction on a new datapoint, which here we will pretend is $\mathbf{x} = \{85.2, 160.3\}$.



x1	x2	Label
98.7	157.6	1
93.6	138.8	1
42.8	171.9	0
…		
87.9	142.7	1
97.9	162.1	1
47.6	182.3	0
92.8	154.5	1

Figure 2.2: *Reminder of the basic supervised learning pipeline — using training data to build a model, then evaluating it on unseen testing data. On this course you will learn several different model types, and the appropriate learning algorithm for each.*

2.1.2 The Simplest Linear Model: The Decision Stump

Given a visualization of the rugby-ballet data in figure 2.1, your own well-engineered learning algorithm (i.e. that thing between your ears) can spot a pattern. We can write a very simple program that will solve the problem,

$$\texttt{if } x_1 > 70 \texttt{ then } \hat{y} = 1 \texttt{ else } \hat{y} = 0 \tag{2.1}$$

where we use the notation $\hat{y}$ to indicate a prediction of the variable y. If we imagine a function $f(\mathbf{x}) = (x_1 - t)$, with $t = 70$, then an equivalent rule is:

$$\texttt{if } f(\mathbf{x}) > 0 \texttt{ then } \hat{y} = 1 \texttt{ else } \hat{y} = 0 \tag{2.2}$$

The point of writing this in the second way is that the function $f(\mathbf{x})$ is now self-contained, and we can now call it a **model**, in that it has a parameter, t.

PARAMETERS

Other models will have many more parameters. This model is called a *decision stump*, where the threshold required to switch the decision from 0 to 1 is the parameter t – the point at which it switches is called the **decision boundary**. This model has a **linear decision boundary**.

DECISION BOUNDARY

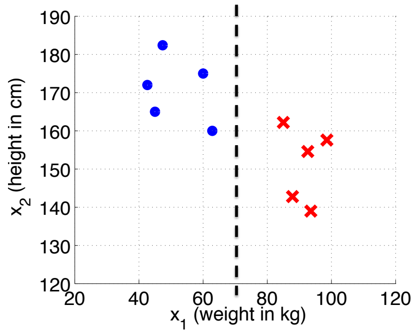


Figure 2.3: *The rugby-ballet data, with the decision boundary for the "decision stump" model (where the threshold parameter* $t = 70$*).*

Note that a different parameter setting, e.g. $t = 90$ would have yielded a different decision boundary, and hence would have classified our testing datapoint $\mathbf{x} = \{85.2, 160.3\}$ as a 0 instead of a 1.

DECISION RULE

So, our model is just the function $f(\mathbf{x}) = (x_1 - t)$, and the *decision rule* for the model is a simple "if-then" rule. When the parameter is set correctly, the model $f(\mathbf{x})$ should give good predictions. The only parameter to set is the t, so we do a simple line-search to find the optimal value, measuring the number of errors made on the data for each possible threshold. Finding the best parameter setting is what we refer to as "learning".

Learning Algorithm for Decision Stump: *Line search*

```
$stepsize \leftarrow 1$, $minErr \leftarrow 99999$
for $t = \min(x)$ to $\max(x)$ by $stepsize$ do
    $numErrs = numberOfErrors(t)$
    if $numErrs \leq minErr$ then
        $minErr \leftarrow numErrs$
        $t_{best} \leftarrow t$
    end if
end for
return $t_{best}$
```

Here, note that we have assumed a function $numberOfErrors(t)$ which evaluates the decision rule eq(2.2) with threshold t on the data, and informs us how many errors were made. Notice also that we've had to assume a *stepsize*, the steps by which we search in the space of possible thresholds. This algorithm with *stepsize* = 1 will perfectly solve the rugby-ballet problem, but on a complex dataset, such as that shown below, the wrong choice of step size could easily overshoot the best parameter value.

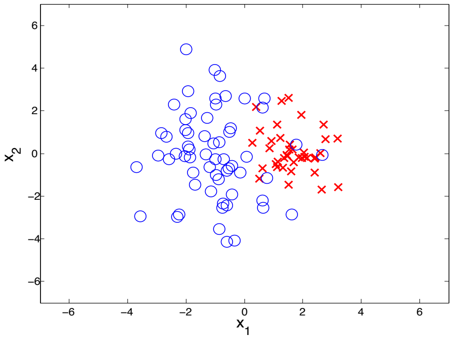


Figure 2.4: *A non-linearly separable problem.*

This is called a *non-linearly separable* problem. The original rugby-ballet data in figure 2.1 allows us to fit a linear model (i.e. draw a linear decision boundary) and perfectly separate the classes, thus it is called a *linearly separable* problem. Whereas we cannot fit a linear decision boundary perfectly between the two classes in Figure 2.4. This data also helps us illustrate the concept of an *error landscape*. Below we have a plot (as we vary t) of the number of errors that the decision stump, eq.(2.2) makes.

NONLINEARLY SEPARABLE

LINEARLY SEPARABLE

ERROR LANDSCAPE

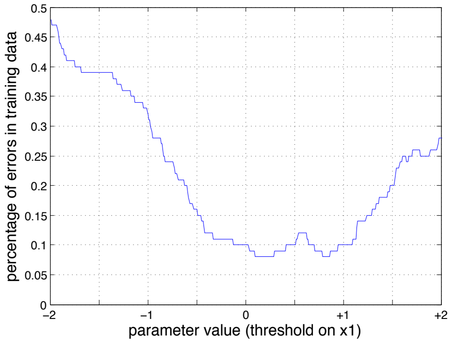


Figure 2.5: The *error landscape* for the non-linearly separable data, varying t.

At different possible threshold values, the stump makes different numbers of errors. When $t = -2$, most points are predicted to be $y = 1$, therefore lots of errors are made. The *minimum* error is at about $t = 0.2$, or $t = 0.8$. These both cause about 8% error, so in this case there is no reason to pick one over the other, so we could just pick one arbitrarily – though more sophisticated solutions will emerge in later chapters. The learning algorithm returns a setting for the parameter, t_{best}, which we then use in eq(2.2), and this is our final model.

Notice also that if we had chosen a step size of 1, we may well have overshot the minimum, so the more fine grained the search, the better, but of course this makes it more computationally intensive. You'll see that most learning algorithms have these sort of trade-offs — lower error comes at the price of more computation.

SELF-TEST
Using what you know from the landscape, draw an optimal stump decision boundary for x_1 into Fig 2.4.

2.1.3 Evaluating Models: Training vs. Testing

This is possibly the most important topic in the entire course. We have so far not considered what will happen to our model when it is deployed out in the real world. We will have "trained" the model, on the data that we have, but then it will encounter new situations, some that it has never seen, like perhaps a 120kg ballet dancer. You can think of this somewhat like your **driving lessons**, versus your **driving test**. Your model is effectively in lessons while you are training it—hence the name "supervised learning". Then, you deploy it to the real world and it is *tested*.

if $f(\mathbf{x}) > 0$ then $\hat{y} = 1$ else $\hat{y} = 0$

Your driving test would not be on the exact same roads, in the exact same car, in the exact same weather conditions, that you had your lessons. In your driving test, you would have to show you can **generalise** the skills you have learnt. In machine learning, equivalently, models have to be able to show good **generalisation**. GENERALISATION

The critical point is, we have only **one** set of data! So, how can we possibly know what will come when the model is deployed? The answer is that we **split** the data up, into *training* and *testing* data. By this I mean we take a random half of the examples as training data, and the other half as testing data. When we learn the parameters of our model, we only allow it to see the training data. Then, we fix the parameters, and see how well it does on the testing data.

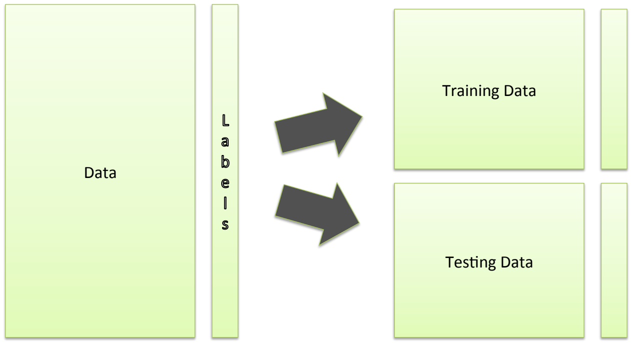


Figure 2.6: *Splitting data into training and testing sets.*

If we were to just learn the model (train) on the entire dataset, we might just learn the characteristics of this dataset, effectively *fine-tuning* our model so it would just work well here, as opposed to working well out in the real world as well. This *fine-tuning* is known in technical terms as *overfitting*, and is obviously something we wish to avoid. OVERFITTING

With this in mind, we can present pseudo-code for a good learning protocol:

```
1. Split the data randomly in half, into training and testing sets.

2. Train a model on the training set.

3. Test this model on the testing set and record the error rate.

4. Repeat this procedure many times, and calculate the average error rate.
```

Notice that we repeated this many times, splitting the data randomly. On each split, the model was trained, and the test set it was evaluated on was kind of like a 'mock' exam. The final 'score' that we assign to the model is the *average test error rate* over the several repeats. We could also report the standard deviation over those repeats, which would tell us more about how our model is sensitive to the random changes in the data.

This is the simplest protocol for supervised learning, and it in fact does have limitations, that we have not yet made clear. We will cover this in much more detail in the Experimental Methods chapter. For now, we will move on to look at a couple more interesting models, still linear, but able to cope with slightly more complex data.

2.2 More Linear Models

The decision stump model has a clear limitation – it works only on a single feature. If the data in Figure 2.4 was rotated 45 degrees, the model would fail miserably to separate the classes. The next model we see will deal with this limitation.

2.2.1 The Perceptron

At the moment, the boundary is only taking into account one of the features. To represent a decision boundary for an arbitrary number of features, we use the *discriminant function.* In n dimensions, the **discriminant function** is:

$$f(\mathbf{x}) = \sum_{j=1}^{d} w_j x_j - t \qquad (2.3)$$

MATRIX NOTATION $\sum_{j=1}^{d} w_j x_j = \mathbf{w}^T\mathbf{x}$

We are also going to start using a different notation for compactness. The expression $\sum_{j=1}^{d} w_j x_j$ can also be written in matrix notation, as $\mathbf{w}^T\mathbf{x}$. If you find this confusing, please refer to a textbook on linear algebra, or some online tutorials on vector mathematics.

The discriminant function describes the *equation of a plane.* A plane is the mathematical generalisation of a line, i.e. a line is a 1-dimensional object, whereas a plane can be of arbitrary dimension, in this case we will say d-dimensional. The discriminant function draws a boundary of dimension $d-1$, that is, if $d = 2$, it draws a 1-dimensional boundary, i.e. a line. If $d = 3$, it draws a 2-dimensional boundary, i.e. a plane.

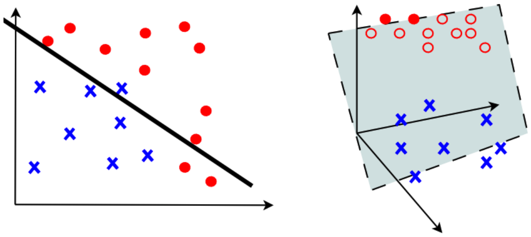


Figure 2.7: Boundaries drawn by the linear model in eq(2.3). **LEFT**: Data where $d = 2$, and since the discriminant is $d-1$ dimensional, the boundary is a 1-d line. **RIGHT**: Data where $d = 3$, so the discriminant is a 2-d plane.

Let's take a brief diversion to examine the relationship between this "discriminant function" and the equation of a line which, as you will remember from school[1] takes the form $y = mx + c$. For any point $\mathbf{x}$ that sits *on the plane* in

[1] If you are from the UK school system. Otherwise you may know it as a different notation, for example in Brazil: $y = ax + b$, in Italy: $y = mx + q$, or in Greek education: $\psi = \alpha\chi + \beta$.

d-dimensional space, we have

$$f(\mathbf{x}) \quad = \quad \mathbf{w}^T\mathbf{x} - t \quad = \quad \sum_{j=1}^{d} w_j x_j - t \quad = \quad 0 \tag{2.4}$$

In 2-dimensions:

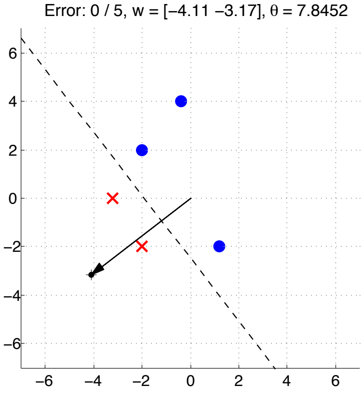
$$\begin{align} f(\mathbf{x}) = w_1x_1 + w_2x_2 - t &= 0 \tag{2.5}\\ \frac{w_1}{w_2}x_1 + x_2 &= \frac{t}{w_2} \tag{2.6}\\ x_2 &= -\frac{w_1}{w_2}x_1 + \frac{t}{w_2} \tag{2.7} \end{align}$$

This now follows the geometry of the line, $y = mx + c$, with

$$m = (-\frac{w_1}{w_2}), \qquad c = \frac{t}{w_2} \tag{2.8}$$

where m is the *gradient* of the line (decision boundary), and c is its intercept on the vertical axis **where** $x_1 = 0$. You can now see that the values of the parameters $\mathbf{w}$ and t have the effect of *moving* the decision boundary around in the space.

The decision boundary is always oriented *orthogonally* (90°) to the parameter vector. The boundary in the image below shows this, the arrow represents the parameter vector, which here is $\mathbf{w} = \{-4.11, -3.17\}$, and $t = 7.8452$.



SELF-TEST
Using eq(2.8), draw onto the image above where the decision boundary would be if $\mathbf{w} = \{4, 4\}$, and $t = 8$.

In 2-d, the discriminant function is a line, in higher dimensions, it becomes a hyper-plane. Remember, for a point $\{x_1, ..., x_d\}$ **on** the plane, $f(\mathbf{x}) = 0$. On one side of the plane, $f(\mathbf{x}) > 0$, on the other, $f(\mathbf{x}) < 0$. The weight vector always points toward the side where $f(\mathbf{x}) > 0$.

Once again, this gives us a very easy way to write a simple computer program that will solve our problem:

$$\texttt{if } f(\mathbf{x}) > 0 \texttt{ then } \hat{y} = 1 \texttt{ else } \hat{y} = 0 \tag{2.9}$$

PERCEPTRON DECISION RULE

This is the *Perceptron decision rule.* Notice here that this is identical to the rule for the decision stump, but the model is different. For the decision stump, the model was $f(\mathbf{x}) = (x_1 - t)$, whereas here, the model is $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t$.

All we have to do now is find the right values of $\mathbf{w}$ and t, that position the boundary in the right place so as to *separate* the two clusters of points as well as possible — when this is the case, it will be classifying more points correctly.

EPOCHS

The way to do this is the **Perceptron Learning Algorithm**. We know that the position of the decision boundary is controlled by the parameters $\mathbf{w}$ and t, so we update these, trying to find the configuration that will classify our data well. The algorithm is iterative, in the sense that it makes multiple *passes* through the data, which we call *epochs*, but in other texts you might also hear this referred to as 'iterations'. The algorithm proceeds as follows.

Perceptron Learning Algorithm

```
$t \leftarrow rand()$, $\mathbf{w} \leftarrow randvector()$
$\alpha \leftarrow 0.1$                                   // alpha is our stepsize
repeat
    for each training example $(\mathbf{x}, y)$ do
        for each parameter $j$ do
            $w_j = w_j - \alpha \times (\hat{y} - y) \times x_j$
        end for
        $t = t + \alpha \times (\hat{y} - y)$
    end for
until changes to parameters are zero
```

The algorithm proceeds by updating the parameters by a small step size α, examining each training datapoint in turn, and updating the parameters for each. The parameter update rule is $\mathbf{w}_j = \mathbf{w}_j - \alpha \times (\hat{y} - y) \times x_j$, where $\hat{y}$ is the output of the decision rule eq(2.9). Since both y and $\hat{y}$ are either 0/1, we can examine what the possible updates to the parameters for each datapoint, in the table below.

Predicted label $\hat{y}$	True label y	Parameter update Δw
0	0	0
0	1	αx
1	0	$-\alpha x$
1	1	0

From this table we can see that if the predicted label is equal to the true label, i.e. the perceptron rule gets the datapoint correct, then there is no update. If $\hat{y} = 1$ but in fact the true label is $y = 0$, then w is decreased by $-\alpha x$. This has the effect of reducing the sum $\sum_i w_i x_i$, bringing it potentially under the threshold t, so in future the prediction might be corrected to $\hat{y} = 0$. The reverse is true if $\hat{y} = 0$ and $y = 1$, where the parameters are increased. Notice that the threshold parameter t is also being updated in a similar manner.

We can also see that with a larger α stepsize, the algorithm will make bigger updates. This step size is called the *learning rate*. Don't be duped into thinking bigger is better! In fact in the vast majority of cases, 0.1 does the job just fine.

LEARNING RATE

We are looking at this algorithm more for historical interest, and as a stepping-stone, than for practical use. In practice, it is severely limited, and can only solve *linearly separable* problems — remind yourself of the definition of this on page 11.

However, it is *guaranteed* to solve the problem if it is linearly separable. The Perceptron Convergence Theorem states this formally:

Perceptron Convergence Theorem
If a dataset is linearly separable, the perceptron learning algorithm will converge to perfect classification within a finite number of training steps. If the data is linearly inseparable, the algorithm will diverge and oscillate forever.

In the next section we will see an extension of this model, which solves some of its problems, in particular, the next model will not oscillate if the problem is inseparable, but instead will find a reasonable parameter setting and terminate.

2.2.2 Logistic Regression

Remember we have a linear model $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t$, and that the value of $f(\mathbf{x})$ for any particular $\mathbf{x}$ can vary in $(-\infty, +\infty)$. This is in fact a bit strange. We know that our model is just trying to make a decision between two possible classes, so it doesn't really need to have the full range to plus/minus infinity. Furthermore, consider that for a particular $\mathbf{x}$, we might have $f(\mathbf{x}) > 0$, indicating that the model thinks the datapoint is class 1, but how confident can we be in this decision? What is the *probability* of class 1 versus class 0? This question is answered by a *Logistic Regression* model.

We redefine our model to have the output,

$$f(\mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x} - t)}} \tag{2.10}$$

SIGMOID

This is called the *sigmoid* function. Notice that all this does is take our original model, and pass it through the sigmoid. The output is a number in the range $[0, 1]$, and the function looks like an 'S' , shown in figure 2.8.

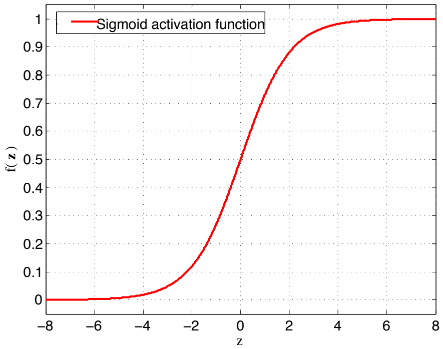


Figure 2.8: *Sigmoid function, also known as logistic curve. The output (vertical axis) is a value in the range* $[0, 1]$*, hence we can interpret it as a probability. The input to the function (horizontal axis) is our original linear model* $\mathbf{w}^T\mathbf{x} - t$.

CONDITIONAL PROBABILITY

Since this is a value between 0 and 1, we treat it as a *probability*, and therefore our model is an estimate of the quantity $p(y = 1|\mathbf{x})$, which is read "the conditional probability of $y = 1$, given the value of $\mathbf{x}$". As a result of this, *our decision rule must change.* The previous decision rule, eq.(2.9), used an inequality about 0, but this new model is always greater than zero. Since it's a probability, the new rule just checks whether it is larger than 0.5:

$$\texttt{if } f(\mathbf{x}) > 0.5 \texttt{ then } \hat{y} = 1 \texttt{ else } \hat{y} = 0 \tag{2.11}$$

SELF-TEST
Imagine $\mathbf{w} = \{2, 3\}$, $\mathbf{x} = \{3, 1\}$, and $t = 7$. What is the value of the logistic regression model, i.e. what is $f(\mathbf{x})$? What about if $w_2 = -3$? What is the predicted label $\hat{y}$ in each case?

We will now derive a learning algorithm for this new model. Remember, the algorithms we will study in this course are, at their heart, just computer programs. They are not *intelligent* in the general sense of the word, and they cannot simply learn as a human can. If we are to have an algorithm that can 'learn', then we need to set a precise objective for the algorithm to pursue. These are called *loss functions*. The loss is the 'cost' incurred by a model for any given prediction that it makes — the loss will be high if it makes many mistakes, and low if it makes few mistakes. Ideally of course, we would like the model to have no cost at all, i.e. minimum possible loss.

LOSS FUNCTION

The first one we will meet is called the *log loss*. The log loss, also known as the *cross-entropy* loss function is:

LOG LOSS
CROSS-ENTROPY

$$L(f(\mathbf{x}), y) = -\Big\{ y \log f(\mathbf{x}) + (1-y) \log(1 - f(\mathbf{x})) \Big\} \tag{2.12}$$

To understand this, remember that for any datapoint y is either 1 or 0. So, only one of the two terms inside the brackets applies for any given datapoint. When $y = 1$, the loss is $-\log f(\mathbf{x})$, which when plotted looks like this:

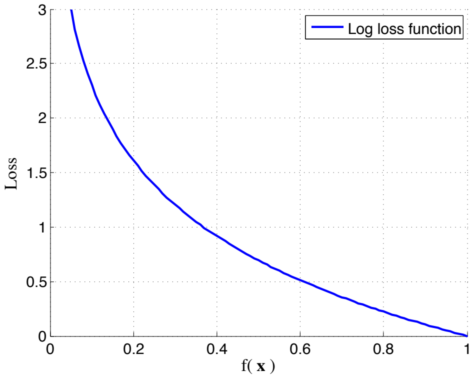


Figure 2.9: *The log loss function, for $y = 1$. The loss (i.e. cost) is highest when $f(\mathbf{x}) \approx 0$, in fact at exactly 0, the loss is infinite. Therefore, the algorithm is encouraged to push the output toward the correct response at $f(\mathbf{x}) = 1$.*

On the other hand, when the true label $y = 0$, we get the term $-\log(1 - f(\mathbf{x}))$, the loss is the opposite, highest at $f(\mathbf{x}) = 1$, lowest when $f(\mathbf{x}) = y = 0$.

When a loss function is summed or averaged over all data points, it is called an *error function.* The summed log loss gives us an overall error function of:

ERROR FUNCTION

$$E = -\sum_{i=1}^{N} \Big\{ y_i \log f(\mathbf{x}_i) + (1 - y_i) \log(1 - f(\mathbf{x}_i)) \Big\} \tag{2.13}$$

This error function is known by two names: the cross entropy error, or the negative log-likelihood. I'm letting you know this just in case you see it in other places referred to by a different name.

NEGATIVE LOG LIKELIHOOD

We now have to find the right parameter settings for $\mathbf{w}, t$, that minimise the error function. To do this we will use a technique called **gradient descent**. Figure 2.10 shows the summed cross entropy error on the simple dataset from the previous section. Here we've set the logistic regression parameters in a particular way (see caption) and varied one of them so you can see again the **error landscape** for this parameter, just as we saw a landscape earlier in fig 2.4 for the decision stump parameter.

GRADIENT DESCENT

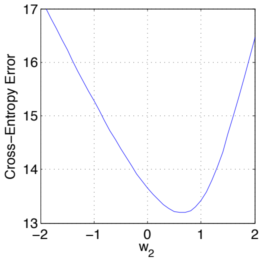


Figure 2.10: *Another error landscape: the summed cross-entropy error, eq.(2.13), for the simple dataset from the previous section, using a logistic regression with $w_1 = 3$, $t = 3$, and w_2 varied in the range* $[-2, 2]$.

The value of this error function is high when (for example) $w_2 = -1.2$, and decreases toward its *global minimum* at $w_2 = 0.6$. Imagine we currently had $w_2 = -1.2$, and wanted to decrease our error. From the diagram, you can clearly see we want to *increase* the w_2 parameter. We note that the *gradient* (i.e. slope) of the error function at $w_2 = -1.2$ is approximately $\Delta E / \Delta w_2 \approx -1.7$. **The rule of gradient descent says: in order to decrease error, we should update parameters in the direction of the *negative gradient*.** In this case, the sign of the gradient is negative, so we must make a positive change to w_2. The reverse is true if $w_2 = +1.2$. There the gradient is about $+2.5$, positive, so we should make a negative change to w_2, moving it toward the minimum.

The gradient of a function for a parameter is given by its *partial derivative* with respect to that parameter. For the cross entropy error function, eq(2.13), with the logistic regression model, eq(2.10), the partial derivative with respect to a parameter w_j is:

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial f(\mathbf{x})} \frac{\partial f(\mathbf{x})}{\partial(\mathbf{w}^T\mathbf{x} - t)} \frac{\partial(\mathbf{w}^T\mathbf{x} - t)}{\partial w_j} = \sum_i (f(\mathbf{x}_i) - y_i) \cdot x_{ij} \tag{2.14}$$

SELF-TEST
If you are ok with calculus, try proving eq(2.14) for yourself. Hint: $\partial f(\mathbf{x})/\partial(\mathbf{w}^T\mathbf{x} - t) = f(\mathbf{x})(1 - f(\mathbf{x}))$.

You may well now be very confused. A slightly more intuitive way to think about gradient descent is simply changing the parameters such that you move "downhill" in the error landscape. Figure 2.11 shows an example of this for a more complex error function, and show two parameters instead of just one.

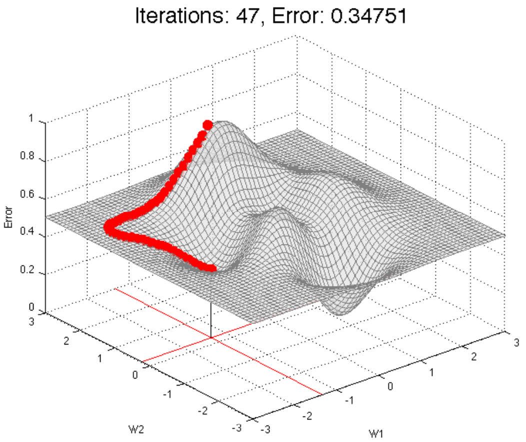


Figure 2.11: Illustration of gradient descent on a more complex error function. The algorithm was started from a random location ($w_1 = -0.2, w_2 = 1.6$), and followed the steepest gradient at each step. The algorithm converged in 47 steps, and found a parameter configuration with error 0.34751. Notice that on this occasion it converged to a *local* minimum at ($w_1 = -1.3, w_2 = 0.2$), instead of the *global* minimum at ($w_1 = 1.2, w_2 = -1.6$).

Algorithm: *Gradient Descent*

```
$\mathbf{w} \leftarrow randvector()$, $\alpha \leftarrow 0.1$
repeat
    for each parameter $j$ do
        $\mathbf{w}_j = \mathbf{w}_j - \alpha \frac{\partial E}{\partial w_j}$        // move $w_j$ in direction of negative gradient
    end for
until termination criterion met
```

In practice, we often use an approximation to this algorithm, called *stochastic gradient descent.* With this modification we compute the gradient for each example one by one, and modify the parameters for each, in an iterative fashion as we did with the Perceptron learning algorithm. Stochastic Gradient Descent is often applied as it works well more effectively in very large datasets, such as many found in industrial applications.

The final learning algorithm takes a similar form to the Perceptron learning algorithm. In fact you'll notice that this is virtually identical, the key difference is in the parameter update equation, where it uses $f(\mathbf{x})$ instead of $\hat{y}$. The threshold parameter is again updated — in a similar way to before, though now we know that these new updates *guarantee* to minimise the error function, as far as it can be done so.

Learning Algo. for Logistic Regression: *Stochastic Gradient Descent*

```
$t \leftarrow rand()$
$\mathbf{w} \leftarrow randvector()$
$maxepochs \leftarrow (5 \times numFeatures)$            //usually ok for most datasets
$\alpha \leftarrow 0.1$                                   // alpha is our stepsize
for $epoch = 1$ to $maxepochs$ do
    for each training example $(\mathbf{x}, y)$ do
        for each parameter $j$ do
            $\mathbf{w}_j = \mathbf{w}_j - \alpha(f(\mathbf{x}) - y)x_j$     //step in negative gradient direction
        end for
        $t = t + \alpha(f(\mathbf{x}) - y)$                 //step in negative gradient direction
    end for
end for
```

Another difference is the Perceptron will eventually stop updating if all data points are correctly classified, whereas the logistic regression keeps iterating to improve its loss function, and thus a specification *maxepochs* is necessary to say how many iterations it should perform. In fact it turns out that the logistic regression has a unique minimum, so converges there (if slowly) as more epochs are run.

Perhaps most interesting about this algorithm, is that it shows the perceptron algorithm is an approximation to gradient descent. This explains at least partially *why* the perceptron algorithm works.

2.3 What you should know by now

By now you should know (and be able to define) the following :

Decision Stump

Linearly Separable versus Non-Linearly Separable (or linearly inseparable)

Linear model / Discriminant function

Overfitting

Perceptron Learning algorithm

Logistic Regression

Cross-entropy / log loss function

Error function / Error Landscape

Gradient Descent

2.4 Optional: The Geometry of Linear Models

This section will not be covered in lectures, but is intended as extra reading, to give you a deeper understanding of the geometry of the linear classifier. You will not be asked to reproduce any of these proofs in an exam, though attempting to understand these will provide a very solid basis for a deeper understanding of the next topic. To begin, we will show that:

1. The angle of the separating plane is controlled by the vector $\mathbf{w}$.
2. The shift of the separating plane from the origin is given by $\frac{t}{||\mathbf{w}||}$.

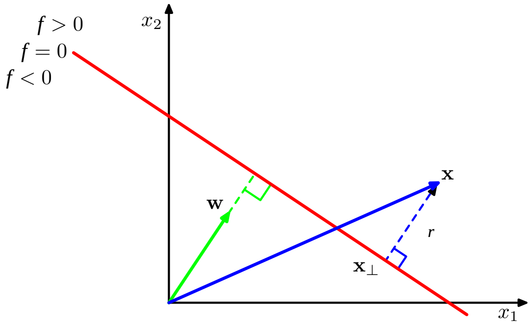
We start from the definition of our linear classifier:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t \tag{2.15}$$

For two points $\mathbf{x}_A$ and $\mathbf{x}_B$ that sit *on* the plane, we have,

$$\begin{aligned} \mathbf{w}^T\mathbf{x}_A - t &= \mathbf{w}^T\mathbf{x}_B - t = 0 && (2.16)\\ \mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) &= 0 && (2.17) \end{aligned}$$

The vector $(\mathbf{x}_A - \mathbf{x}_B)$ points along the decision boundary. Now, since the product of two vectors is proportional to the cosine of the angle between them, and $cos(90) = 0$, then we have that $\mathbf{w}^T$ is **orthogonal** (90 degrees) to the vector pointing along the boundary. Wherever the vector $\mathbf{w}$ points, the plane sits at 90 degrees to it.



What about the t term? What role does that play? Remembering that the unit vector[2] of $\mathbf{w}$ is $\frac{\mathbf{w}}{||\mathbf{w}||}$, then we can write an expression for an arbitrary point $\mathbf{x}$ as,

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{||\mathbf{w}||} \tag{2.18}$$

that is, a point $\mathbf{x}_\perp$ that is the orthogonal projection of $\mathbf{x}$ onto the plane, plus the unit vector scaled by a quantity r, the (signed) distance from the plane to

[2] http://www.ies.co.jp/math/java/vector/unit_vector/unit_vector.html

point $\mathbf{x}$. The unit vector $\frac{\mathbf{w}}{||\mathbf{w}||}$ tells you the direction, and r "stretches" the point $\mathbf{x}_\perp$ out towards $\mathbf{x}$. If $\mathbf{x}$ sat behind the plane, r would be negative. To find an expression for r, the first step is to take the above, multiply by $\mathbf{w}^T$ and subtract t.

$$\mathbf{x} = \mathbf{x}_\perp + r\frac{\mathbf{w}}{||\mathbf{w}||} \tag{2.19}$$

$$\mathbf{w}^T\mathbf{x} - t = \mathbf{w}^T\mathbf{x}_\perp - t + r\frac{\mathbf{w}^T\mathbf{w}}{||\mathbf{w}||} \tag{2.20}$$

Then remembering that $||\mathbf{w}|| = \sqrt{\mathbf{w}^T\mathbf{w}}$ and using eq(2.15), this is

$$f(\mathbf{x}) = f(\mathbf{x}_\perp) + r\frac{\mathbf{w}^T\mathbf{w}}{||\mathbf{w}||} \tag{2.21}$$

$$f(\mathbf{x}) = r||\mathbf{w}|| \tag{2.22}$$

So the length of the vector from the plane to a point $\mathbf{x}$ orthogonal to it, is

$$r = \frac{f(\mathbf{x})}{||\mathbf{w}||} \tag{2.23}$$

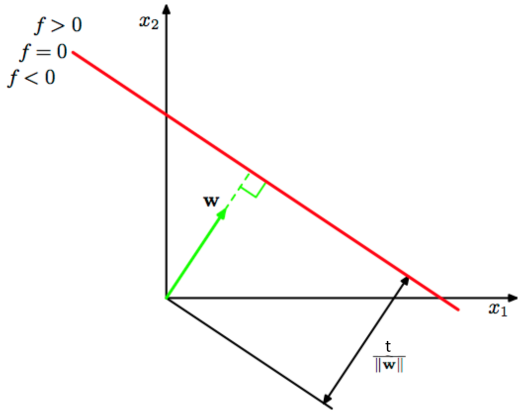
If we choose our 'arbitrary point' to be the origin, $\mathbf{x} = \{0, ..., 0\}$, we have

$$r = \frac{(\mathbf{w}^T\mathbf{x} - t)}{||\mathbf{w}||} = \frac{-t}{||\mathbf{w}||} \tag{2.24}$$

This is the vector from the plane the origin, so reversing it, the *vector from the origin to the plane* is

$$s = \frac{t}{||\mathbf{w}||} \tag{2.25}$$

So, the distance of the plane from the origin is controlled by t.



An interesting observation to note is that the *same hyper-plane can be defined by several different parameter settings*. Imagine multiplying the weight vector $\mathbf{w}$ by a constant scalar k. This stretches the weight vector in the above figures, but does not change the *angle* of the plane, since only the magnitude, not the direction, of the vector is changed. This does however, shift the plane somewhat, since $\mathbf{w}$ is involved in the $s = \frac{t}{||\mathbf{w}||}$ term. This can be compensated for, by scaling the t by the same value k. Proof follows. First define $\mathbf{w}' = k\mathbf{w}$.

$$\begin{aligned} ||\mathbf{w}'|| &= \sqrt{k\mathbf{w}^T k\mathbf{w}} \\ &= \sqrt{k^2}\sqrt{\mathbf{w}^T\mathbf{w}} \\ &= k||\mathbf{w}||. \end{aligned} \tag{2.26}$$

Then define $t' = kt$, and the shift of the plane from the origin is still given by,

$$s = \frac{t'}{||\mathbf{w}'||} = \frac{kt}{k||\mathbf{w}||} = \frac{t}{||\mathbf{w}||}. \tag{2.27}$$

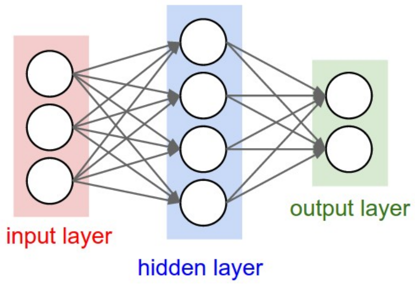
So, multiplying all the parameters by a constant term does not change the position of the plane. This result will be key to a deep understanding the next major topic, of *Support Vector Machines*.

2.5 Optional: Neural Networks

This section will not be covered in lectures, but is intended as optional extra reading. You will not be asked to reproduce any of this material in an exam, though you may find it interesting for your projects.

2.5.1 What is a neural network?

Put simply, a neural net is just some logistic regression units plugged together:



Each logistic regression unit is referred to as a *neuron*, and they are usually arranged in layers as shown in the image. The image shown is a simple example known as a fully-connected *multi-layer perceptron* – however, there can be more complex connection patterns and even loops.

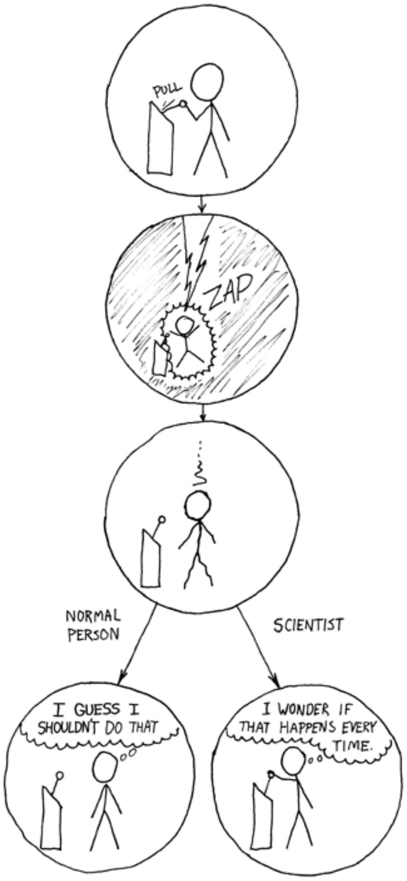
The **input layer** is where we feed in the input vector, $\mathbf{x}$, as we have for the single logistic regression – the example shown in the image is for 3 inputs, so the $\mathbf{x}$ vector has only two elements. Each layer feeds the output of the neurons into the next layer, finally emerging from the output layer – which are again just logistic regression units.

There are several valid learning algorithms for a neural network. The most well known one is called *back-propagation*, which is a simple extension of the gradient descent principle we covered in this chapter. BACKPROP

If you wish to read more on this topic, there are many resources on the web, and in the textbooks recommended on the course unit website.

Chapter 3

Experimental Methods I


PULL
ZAP
NORMAL PERSON
SCIENTIST
I GUESS I SHOULDN'T DO THAT
I WONDER IF THAT HAPPENS EVERY TIME.

3.1 Reminder of Supervised Learning Protocol

So, you've just learnt a whole bunch of stuff. You should think about this material as a set of tools, and remember that it is possible to use tools effectively, but also possible to use them very naively. We will now look at how to *evaluate* models thoroughly and fairly, i.e. to avoid using them in a naive manner.

You will no doubt remember the diagram below from earlier chapters. This outlines the basic protocol we follow when applying machine learning. We have some data, which we use to construct a model, after which, the model is evaluated on some other data, that it has never seen before. The data to build the model is called *training data*, and the other is called *testing data*. Just to remind you, the *training error* is the number of mistakes that the model makes on the training dataset after it has been trained. The *testing error* is the number of mistakes that the model makes on the *testing* dataset after it has been trained.

TRAINING DATA
TESTING DATA

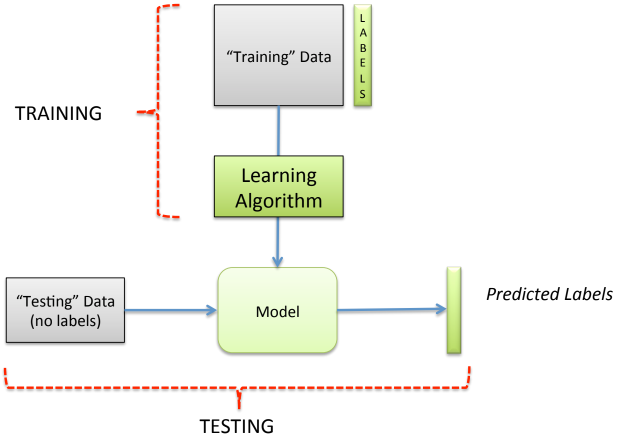


Figure 3.1: *The basic supervised learning pipeline — the* **"training" stage***: use data to build a model, then the* **"testing" stage***: evaluating the model on unseen testing data.*

The trouble is, we have only one set of data. We've already seen part of the solution back in section 2.1.3, with the idea of splitting the dataset into *training* and *testing* data. We take a random half of the data, train the model, then test it on the other half. This protocol ensures that when we test the model, it can't "cheat", by having seen the answers before, kind of like a student having to sit an exam. The random splitting is one of several ways to evaluate a model — we will now discuss another one.

3.2 Model Selection via Cross-Validation

Note: *there is a nice online video which explains the concepts in this section in a slightly different way - you may like to watch it.*

`http://vimeo.com/29569892`

Imagine you have two possible models which you could use on a problem. You have a data with 1000 examples. We would like to predict as well as possible on future, unseen, testing data. You have a logistic regression, and a decision stump. Which one is "best"? Which model should you select?

The problem of picking the 'best' from a pool of possible models is called *model selection*. You *could* train each model on the 1000 examples that you have, see how many mistakes each one makes, then select the model that makes the fewest mistakes. This would be a terrible idea.

Instead, we will do the following with the data. We will split the data into several parts, called *folds*.

DATA FOLD

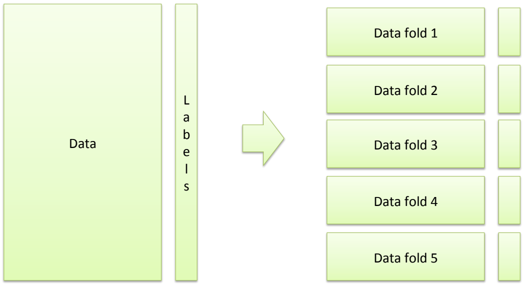


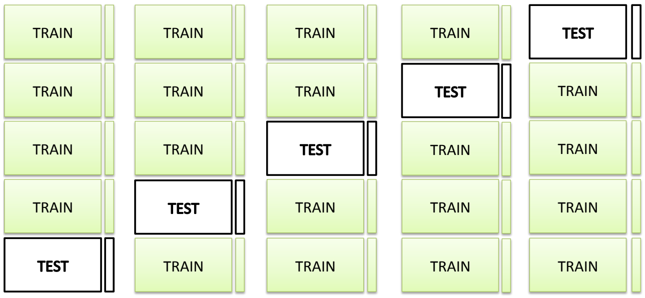
Figure 3.2: *Splitting the data into folds for cross validation.*

In the first step, we train the model on folds 1-4, and test on fold 5. The next step trains the model on folds 1,2,3 and 5, but tests on 4. The next trains on 1,2,4,5, then tests on 3, and so on. In each step, the model has *never seen* the folds that were left out, so it's kind of like being deployed out in the real world, when it encounters some unseen data. Finally, we average the errors that happened in each fold. This final average is called the *cross-validation error*. The results of this procedure might be, for example with a logistic regression:

CROSS-VALIDATION ERROR

Train on folds 1,2,3,4... then **test** on fold 5, where the error rate is: **16%**
Train on folds 1,2,3,5... then **test** on fold 4, where the error rate is: **12.5%**
Train on folds 1,2,4,5... then **test** on fold 3, where the error rate is: **17%**
Train on folds 1,3,4,5... then **test** on fold 2, where the error rate is: **14.1%**
Train on folds 2,3,4,5... then **test** on fold 1, where the error rate is: **10%**

Logistic Regression - Average : $(16 + 12.5 + 17 + 14.1 + 10)/5 = 13.92\%$



TRAIN	TRAIN	TRAIN	TRAIN	TEST
TRAIN	TRAIN	TRAIN	TEST	TRAIN
TRAIN	TRAIN	TEST	TRAIN	TRAIN
TRAIN	TEST	TRAIN	TRAIN	TRAIN
TEST	TRAIN	TRAIN	TRAIN	TRAIN

Figure 3.3: *Splitting the data into folds for cross validation.*

Then, say we repeated this whole process for a decision stump, and got:

Train on folds 1,2,3,4... then **test** on fold 5, where the error rate is: **16%**

Train on folds 1,2,3,5... then **test** on fold 4, where the error rate is: **16.5%**

Train on folds 1,2,4,5... then **test** on fold 3, where the error rate is: **17%**

Train on folds 1,3,4,5... then **test** on fold 2, where the error rate is: **16.2%**

Train on folds 2,3,4,5... then **test** on fold 1, where the error rate is: **17.1%**

Stump - Average : $(16 + 16.5 + 17 + 16.2 + 17.1)/5 = 16.5\%$

MODEL STABILITY

What can we tell about these models from the results? One might consider that the logistic regression is the best classifier to go with, as it has a lower average error rate. But, notice that the variation in the performances across the five folds is larger with the logistic regression. Quantifying this, the *standard deviation* is higher at $\sigma = 2.794$ for the logistic regression, than the stump which has $\sigma = 0.4827$. The stump is said to be a *more stable* model. I am not saying that now you should prefer the stump model, but simply pointing out that there are *multiple ways to quantify the utility of a model.*

LOOCV

Leave One Out Cross Validation (LOOCV) is the extension of this principle when we have very small datasets. In this situation where we might have N data points, we make N folds where each fold is just a single example. So, we train on $N - 1$ points, then test on the one that we left out, then repeat for every example. This is clearly quite computationally expensive, so we normally only do it for very small datasets.

You might think this is the end of the chapter - that we now know how to evaluate a model. In fact, we have overlooked an important question — **is this cross validation error a good estimate of the future generalisation error?** The answer is no, in fact it will be an *optimistic* estimate, i.e. it will make you think your model is better than it actually is. The reason for this is that you have implicitly fitted some of the parameters to the training data, by iterating over it with the different folds.

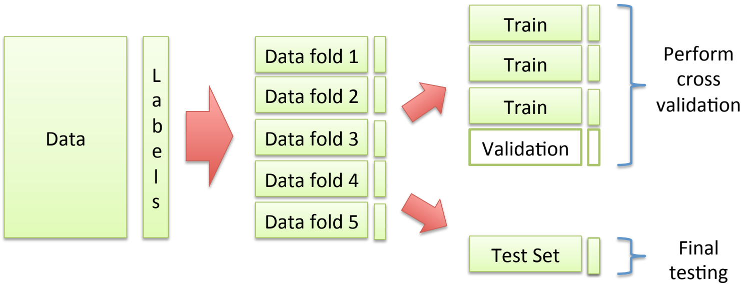


Figure 3.4: *Full cross-validation procedure, with train, validation and test sets.*

The solution is to separate your data into 3 parts : a training set, a validation set, and a test set. So, the final procedure is:

1. Split the data into (for example) $K = 5$ folds.
2. Keep the 5th fold as a hold-out test set.
3. Perform cross validation with the remaining 4 folds.
4. Select the model that performs best on average over those 4 folds.
5. Evaluate the model on the hold-out test set.

This final figure, the error on the hold-out test set, is a good estimate of future generalisation performance. However, it has a **variance**. To reduce the variance, this entire procedure can be repeated many times, by shuffling the data examples. The average test error (and standard deviations) can then be reported. VARIANCE

3.3 Plotting your Results

In the previous sections you will have noticed *variation* in the performance of the models. The *standard deviation* of error is an important property of a model. Imagine telling a customer that your model will likely have generalisation performance on average of 20% error, plus or minus 10%, versus being able to tell them you can get 22% error, plus or minus 0.1%. The more stable model is potentially more *reliable* even if on average it does slightly worse.

In this case we should be plotting *confidence intervals* around our performances, as shown in the figure below.

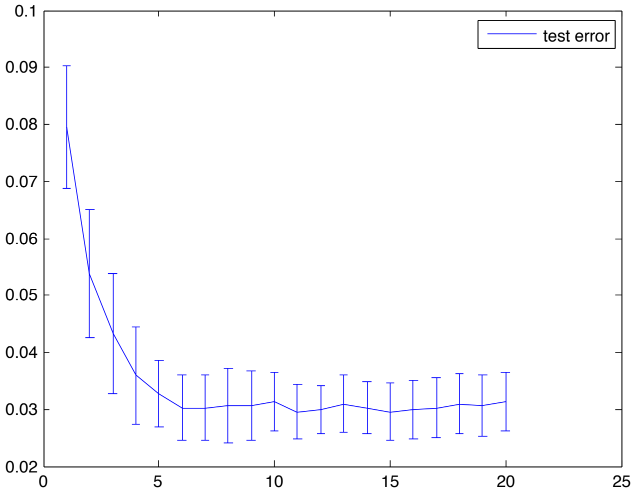


Figure 3.5: *Error bars — the line is the mean performance, while the upper/lower bars represent the 95% confidence intervals.*

If you are not aware of the concept of confidence intervals, please consult a statistics tutorial — some are mentioned in the conclusion section of this chapter and on the course website.

3.4 Useful Trick: Rescaling Your Data

Sometimes, you will get data with varying *ranges*. By this I mean you might have measured peoples' salaries in pounds, and their height in centimetres. Let's say the most wealthy person in your dataset earns £70,000, and the least wealthy earns £5,000. The heights are then in the range 160cm to 200cm. If we supply these features to a logistic regression, the learning will take a long time to converge. *Feature scaling* speeds up gradient descent by avoiding many extra iterations that are required when one or more features take on much larger values than the rest. The usual way of doing this is called *normalisation*, which works as follows.

1. Given a feature, calculate the mean and standard deviation

2. Subtract this mean value from every example.

3. Divide the result by the standard deviation.

The result of this is a new feature, which we replace the old one with.

3.5 What you should know by know

You should know what cross validation means, and why it is needed. You should know what confidence intervals are, and why there are needed. You should ideally be able to perform one or two statistical tests, which you can make use of in your projects.

For more information on confidence intervals and statistical tests, consult the book chapter on the course website:

Statistical estimation using confidence intervals, by David Jones.
`http://studentnet.cs.manchester.ac.uk/pgt/COMP61011/materials/PharmStats.pdf`

Chapter 4

Geometric Models II: Support Vector Machines

HEY, MISS LENHART! I FORGOT EVERYTHING ABOUT ALGEBRA THE MOMENT I GRADUATED, AND IN 20 YEARS NO ONE HAS NEEDED ME TO SOLVE ANYTHING FOR X!
I TOLD YOU I'D NEVER USE IT!
IN YOUR FACE!
IT'S WEIRD HOW PROUD PEOPLE ARE OF NOT LEARNING MATH WHEN THE SAME ARGUMENTS APPLY TO LEARNING TO PLAY MUSIC, COOK, OR SPEAK A FOREIGN LANGUAGE.

4.1 What you need to know about SVMs

We're about to embark on learning a new model, the *Support Vector Machine*. The models we've seen so far (decision stump, perceptron, logistic regression) are all quite simple and effective classifiers, though quite limited. The SVM on the other hand is an absolutely state of the art modern classifier, that became popular in the early 2000s. To summarise what is in this chapter, there are several points you should know:

1. **Support Vector Machines are state of the art classifiers.**
   If I had to trust my life to a machine learning algorithm, there are only two algorithms I would go with — SVMs are one of them. You'll find out what the other is in a chapter to come.

2. **They are VERY maths heavy.**
   If you thought this course has been heavy on the maths so far, then you're in for a surprise — we've barely begun. The SVM was originally derived in 1963 by a Russian mathematician called Vladmir Vapnik, and understanding them to the deepest of levels requires knowledge of topics such as as "reproducing kernel Hilbert spaces". Fortunately, you can learn to *use* SVMs without too much of this depth, but you can't avoid it completely.

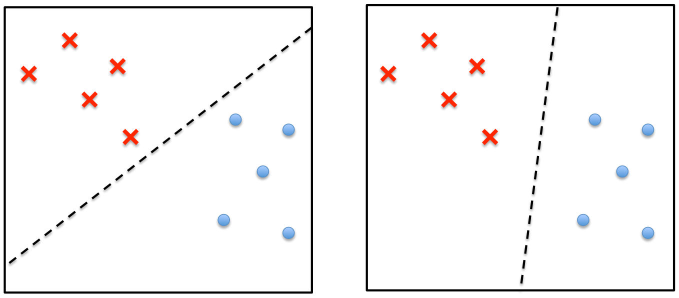
3. **They maximise the margin**.
   The SVM learns an optimal linear boundary in the sense that they place the boundary exactly halfway between the two classes. This is called the boundary with maximum "margin".

4. **They use a 'hinge' loss function.**
   The hinge loss gives nice properties, but it requires a new type of learning algorithm called "quadratic programming", instead of the gradient descent we've used til now. We will not cover QP in detail but you should know it exists.

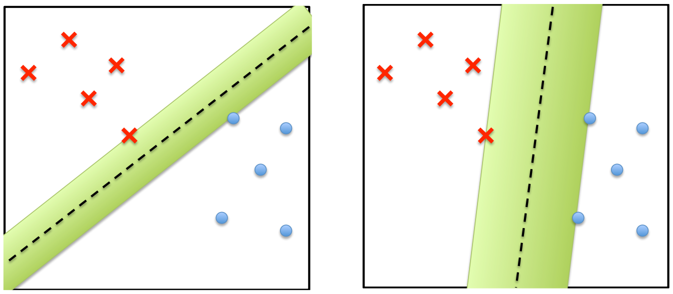
5. **They can solve non-linearly separable problems.**
   All the models we've seen so far draw linear decision boundaries — the decision stump, the perceptron and the logistic regression cannot perfectly solve problems where the data is 'muddled up' with overlapping classes. The SVM is also a linear model, but using the "kernel trick", can solve nonlinearly separable problems.

4.2 Decision Boundaries with Large Margins

Below we have two identical problems (on the left and right) with two alternative decision boundaries, each of which perfectly separate the data. Given the choice of the two, which one would you choose to deploy out in the field as a model?



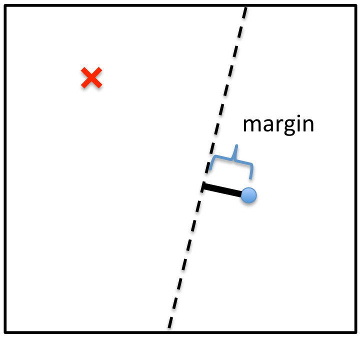
I hope you just said "the one on the right". With both solutions, the parameter vectors were positioned so as to correctly classify all the training data points. However there is a difference between them — the boundary on the left has 'only just' separated the points, whereas the one on the right has 'breathing space', where the distance from the boundary to the closest data point is much larger. This is illustrated below for each boundary by imagining the decision boundary getting fatter and fatter, until it touches some data points.



The diagram on the right is said to have *large margins*, i.e. a lot of space on either side of the decision boundary. In this case, if a test point arrived that was quite similar to the training data, we would have the best possible chance of correctly classifying it.

The topic of this chapter is the *Support Vector Machine*, which is a *linear model*, meaning it is of the form $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t$. The SVM uses a special loss function and learning algorithm which allow it to find a linear decision boundary with *large margins*. So, **another way of thinking about an SVM is that it *finds the optimal linear decision boundary.*** We will now get a little more formal in describing these concepts, to see how the SVM achieves this.

MARGIN The **margin of a decision boundary** is the perpendicular distance to the closest training datapoint.



MAXIMUM MARGIN The natural choice of decision boundary would be the one that has *maximum margin*. The Support Vector Machine finds this decision boundary.

Using a bit of quite complex geometry (see the original SVM paper, at the end of this chapter), it was proven that the margin for a decision boundary is inversely proportional to the length of its parameter vector $\mathbf{w}$. More formally,

$$margin \propto \frac{1}{||\mathbf{w}||} \tag{4.1}$$

where the length of $\mathbf{w}$ is given by $||\mathbf{w}|| = \sqrt{\sum_{j=1}^{d} w_j^2}$. So, in order to maximise the margin, we can equivalently *minimise* $||\mathbf{w}||$. However, the obvious solution is just to set $w_j = 0$ for all j, which would clearly be an awful model, just setting all parameters to zero. So, we need a constraint, to ensure that the parameters do not shrink too much, but are balanced against the need to correctly classify the data points. This constraint is encoded by the *hinge loss function.*

4.3 SVM loss function and Learning Algorithm

The convention in SVM terminology is to have class labels that are $-1/+1$, or written more formally $y \in \{-1, +1\}$. We remember of course that the linear model $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t$ has value 0 for an $\mathbf{x}$ that sits exactly on the decision boundary, and is positive on one side, negative on the other. So, combining these two facts, we know that if,

$$y_i f(\mathbf{x}_i) > 0 \tag{4.2}$$

then the data point was correctly classified by the model. *Take a minute to think about this, as this simple notational trick will be used again and again through the following sections.*

The "hinge" loss function makes use of this, and can be stated as,

$$L_{hinge} = \max\Big\{0, 1 - y_i f(\mathbf{x}_i)\Big\} \tag{4.3}$$

Or graphically,

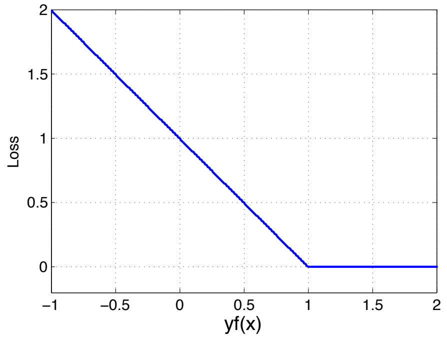


Figure 4.1: *The "hinge" loss function. Notice that the loss decreases linearly until $yf(\mathbf{x}) > 1$, at which point the "hinge" turns the loss to zero. This ensures that once the model is correct with certain confidence, it is not penalised any further.*

The hinge point means that the loss is not continuously differentiable, hence we cannot use gradient descent as our learning algorithm here. Instead, we'll formalize our SVM error function such that we can use a different learning algorithm.

If we combine the need to minimise $||\mathbf{w}||$ with the need to minimise the hinge loss, and sum over all training data points, we get our SVM error function,

$$E = \sum_{i=1}^{N} \max\Big\{0, 1 - y_i f(\mathbf{x}_i)\Big\} + \frac{1}{2}\sum_{j=1}^{d} w_j^2 \tag{4.4}$$

Notice that instead of minimising $||\mathbf{w}||$, we have chosen to minimise the square of it, with a constant $\frac{1}{2}$ placed in front of it. This is important as it makes our SVM error function into a *quadratic programming* (QP) problem with *linear constraints*. The $\frac{1}{2}\sum_j w_j^2$ is the quadratic part, and the hinge loss is the linear constraint.

QUADRATIC PROGRAMMING

The learning algorithm for SVMs is therefore any *QP solver*, of which there are many, highly efficient, implementations. While for logistic regression, we played with the details of gradient descent, and maybe you even implemented it, the details of QP solvers are quite complex, and it is not really recommended that you try to implement them yourself.

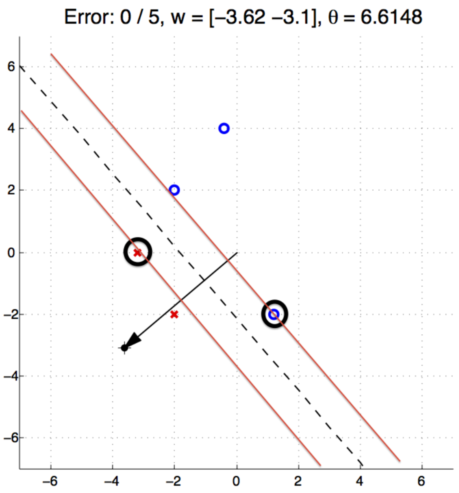


Figure 4.2: An optimal linear boundary, in the sense that it maximises the margins, as placed by an SVM. The data points that are circled are known as the *support vectors*.

SUPPORT VECTORS

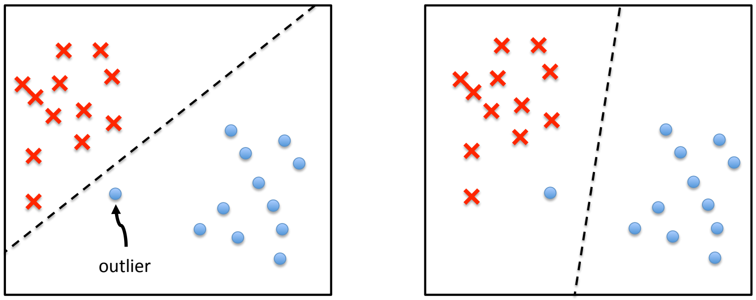
The figure above illustrates the result of applying an SVM to a simple linearly separable dataset. The boundary is shown as a dashed line, and the margin is illustrated by the solid lines, called the *supporting hyperplanes*. The data points that touch the supporting hyperplanes are called the *support vectors*, which reveals to you why this is called a *Support Vector Machine*.

LINEAR-IN-THE-PARAMETERS

However, notice that this is still a *linear decision boundary*. Yes, that's right, **the SVM is still a linear model**, or more technically, it is *linear-in-the-parameters*. But, it has two tricks up its sleeve which make it a much more powerful learning paradigm.

4.4 SVMs with "Soft" Margins

Imagine we had a dataset, with two possible decision boundaries to choose from, illustrated below.

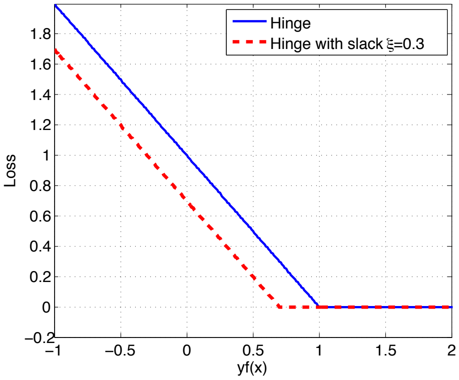


The one on the left is correctly classifying all the training points, whereas the one on the right makes one classification error. However, it's quite clear from looking at the scatterplot, that there is an 'outlier' datapoint, one that was probably mis-labeled when our dataset was constructed. Wouldn't it be nice if we had a way of modifying our SVM loss function such that it could *automatically* take account of things like this, without us having to scatterplot and look for outliers? Well fortunately there is, and it's called the *soft margin* SVM. SOFT MARGIN SVM

We now modify our hinge loss function, to include *slack*[1] *variables.* The new loss function is, SLACK VARIABLES

$$L_{hinge} = \max\Big\{0, 1 - y_i f(\mathbf{x}_i) - \xi_i\Big\}, \qquad (4.5)$$

or, graphically,



This gives allowance for data points that might be mislabelled—the slack variable ξ_i is the amount of slack given to datapoint $\mathbf{x}_i$. If the slack for a point is greater than 1, it means the datapoint is "allowed" to be incorrectly classified, so as to allow the overall dataset to be reasonably well classified. If it is greater than 0, then it is allowed to violate the margins, hence why this is called the 'soft' margin' SVM. These concepts are illustrated in figure 4.3.

The slack however does not come without a price. For each bit of slack, the SVM must pay a penalty. The error function for the soft margin SVM turns out to be a simple modification of the original:

$$E = \sum_{i=1}^{N} \max\Big\{0, 1 - y_i f(\mathbf{x}_i) - \xi_i\Big\} + \frac{1}{2}\sum_{j=1}^{d} w_j^2 + C\sum_{i=1}^{N} \xi_i \qquad (4.6)$$

where we have used the hinge loss with slack, and included a penalty $C\sum_i \xi_i$, which penalises the SVM for allowing slack variables. The problem therefore becomes a tradeoff between allowing slack and optimising the loss function, all while maximising the margin. The trade-off can be influenced by the user via the coefficient C. The value of C roughly translates as how "soft" the margins

[1] Dictionary definition of "slack": *to loosen, to make allowance for, to be less strict*

will be. A higher value means more the penalty for margin violations is very severe, hence a stricter SVM solution. A very small value says no penalty for slack variables, hence we are allowed to make mistakes. The default value is usually $C = 1$. The effect is illustrated graphically in figure 4.4.

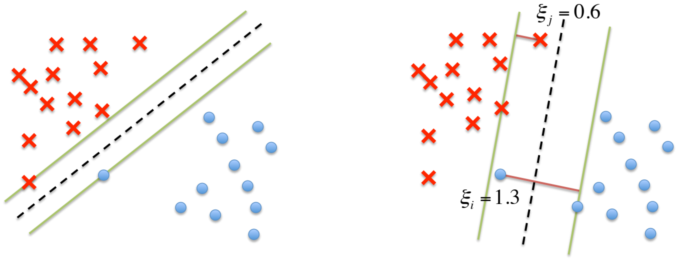


Figure 4.3: *Soft margin SVM. The ξ values are optimized by the QP algorithm, allowing the margin to "soften" at certain points along its length, essentially allowing the SVM to ignore points that cross over its boundary.*

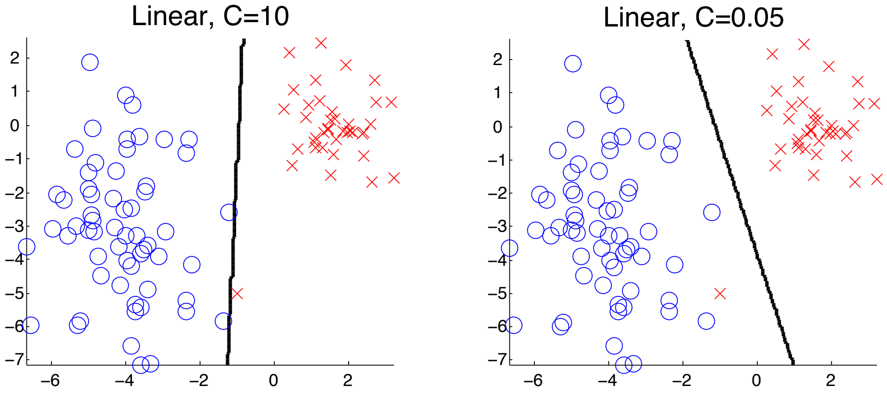
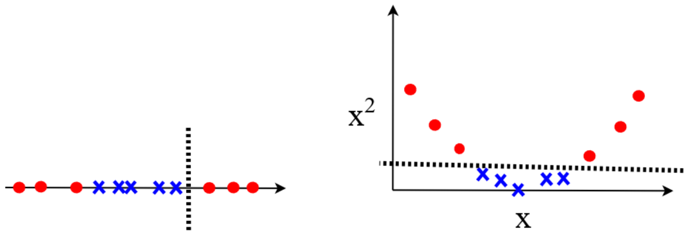


Figure 4.4: *The effect of C, the slack variable penalty. A large value (left) means a very strict penalty, so a very strict SVM solution will be found, even if the margin is small. A smaller value (right) means some data points are allowed to violate the margins, hence an approximate SVM solution is found, but it has a larger margin.*

4.5 Non-Linear SVMs : The “Kernel Trick”

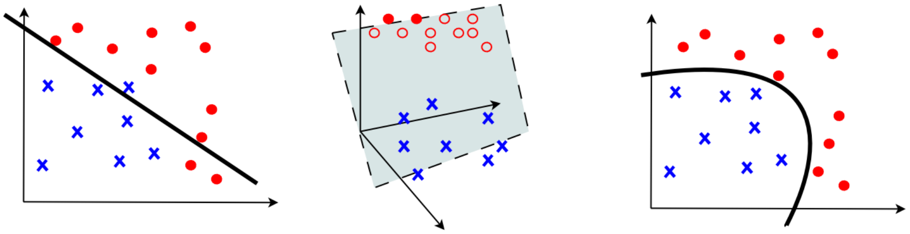
SVMs have a further trick up their sleeve, allowing us to solve non-linearly separable problems. The general idea will first be illustrated, then we’ll go into the maths of exactly how it works. Imagine we had to solve the linearly inseparable problem on the left below.



Linearly inseparable in 1D...

...becomes separable in 2D

We started with a one-dimensional problem. The crosses and circles are intermixed, so cannot be separated with a straight line. On the right, we have added an extra dimension to our plot, x^2, and all of a sudden the problem becomes linearly separable. The principle is further explored, where the original data is in 2d, in the diagrams below, and we illustrate how an SVM can help us.



Here, a dataset (LEFT) is non-linearly separable in the original 2d space. If we project the data upward into a 3d space (MIDDLE), and find that we can separate it with a linear boundary (which is a plane in 3d). When we return to the original 2d space (RIGHT) the data points are “warped” back to their original positions, but the decision plane is also warped, giving us a *nonlinear* boundary.

Our challenge in this section therefore, is to understand *how we can define an appropriate high dimensional space*, into which we will project our data.

Just as a reminder, the *linear SVM* is of the form,

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - t. \tag{4.7}$$

As a first step before we go into this, we use a simple rearrangement of the linear model. Instead of having the threshold t as a separate parameter, we can incorporate it as an extra w_0 parameter:

$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} - t && (4.8)\\ &= \sum_{j=1}^{d} w_j x_j - t && (4.9)\\ &= \sum_{j=1}^{d} w_j x_j + (-1 \times w_0) && (4.10)\\ &= \sum_{j=0}^{d} w_j x_j && (4.11) \end{aligned}$$

where $t = w_0$ and $x_0 = -1$ is a new constant input feature. This has built the threshold into our model in a visually different way, though mathematically equivalent. If you derive the update equations for gradient descent as we did earlier, you should find them to be identical.

Now, returning to our problem, we have to think up some new set of features, projecting the $\mathbf{x}$ into a higher dimensional space so we can separate the classes with a linear boundary. We can denote the hypothetical high dimensional space by the function $\phi(\cdot)$ which acts upon a vector $\mathbf{x}$ to produce a new $\mathbf{x}$. So, we can rewrite our model as,

$$f(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) \tag{4.12}$$

But what ϕ should we choose? Fortunately the *Kernel trick* shows us how we can avoid this problem. The Kernel trick relies on a piece of mathematics called the *Representer theorem* (Kimeldorf & Wahba, 1971).

4.5.1 The Kernel Trick

The **Representer theorem** (Kimeldorf & Wahba, 1971) proved that an optimal linear decision boundary was always of the form $\mathbf{w}^* = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i$, for some set of parameters α_i, where the $\mathbf{x}_i$ are the training data vectors, and remember $y_i \in \{-1, +1\}$. Put another way, this shows that, the optimal parameters for a linear model are actually a simple linear combination of the training data points[2]. This result can be plugged back into our original equation for the linear model, which when operating to classify a hypothetical test point $\mathbf{x}'$ is of the form,

$$\begin{aligned} f(\mathbf{x}') = \mathbf{w}^T\mathbf{x}' &= \left\{\sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i\right\}^T \mathbf{x}' && (4.13) \\ &= \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}' && (4.14) \end{aligned}$$

or if we used our hypothetical ϕ,

$$f(\mathbf{x}') = \sum_{i=1}^{N} \alpha_i y_i \phi(\mathbf{x}_i)^T \phi(\mathbf{x}') \tag{4.15}$$

Notice that the training data points are expressed only as *dot products* with the test point, i.e. the result of $\phi(\mathbf{x}_i)^T\phi(\mathbf{x}')$ is a *scalar quantity.* Wouldn't it be nice if there was a class of mathematical functions, that would give us the scalar result that we needed, but *without computing the ϕ high dimensional space*? That would seem quite strange, and sounds like wishful thinking, but it's true, they are called 'kernel functions', and their existence is referred to as the **kernel trick**.

KERNEL TRICK

$$K(\mathbf{x}_i, \mathbf{x}') = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}') \tag{4.16}$$

where ϕ is some high dimensional function. So, we can use a kernel function instead of the dot product, as so:

$$f(\mathbf{x}') = \sum_{i=1}^{N} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}'). \tag{4.17}$$

The α_i parameters that are left will still be learnt via quadratic programming, so all we have to do is choose an appropriate kernel. **Mercer's Theorem** states that any continuous, symmetric, positive semi-definite function $K(\mathbf{x}_i, \mathbf{x}')$ is a valid kernel. We will now meet two examples of this.

MERCER'S THEOREM

[2] This can also be seen by examining the updates in the Perceptron learning algorithm.

4.5.2 The Polynomial kernel

POLYNOMIAL KERNEL

The *polynomial kernel* is expressed,

$$K(\mathbf{x}_i, \mathbf{x}') = (1 + \mathbf{x}_i^T \mathbf{x}')^d \tag{4.18}$$

where d is the degree of the polynomial. Note that I have reused notation here, d was earlier the dimension of the original feature space - here I used it again simply because d is the usual notation for the polynomial degree in the literature.

If you use a 2nd order polynomial, and multiply out the kernel above, you should get to the conclusion that it uses the implicit feature space: $\phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$. So it has projected from a 2-d space into a 6-d space. When we project back into the original 2-d space, we get a non-linear boundary. The clever thing is that although we technically will be working with high dimensional functions ϕ, we *never explicitly compute them*, we just work with the kernel function instead.

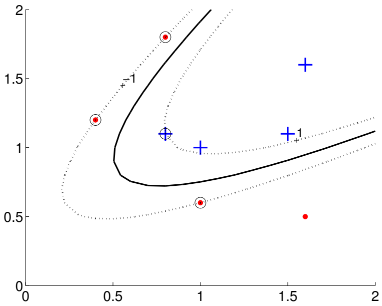


Figure 4.5: *A nonlinearly separable dataset being solved by an SVM with a polynomial kernel of degree 2. The* **support vectors** *are circled.*

The effect of varying the degree of the polynomial can be seen below. The higher degree, the more high order terms are introduced to the implicit feature space ϕ, hence the final decision boundary becomes more complex.

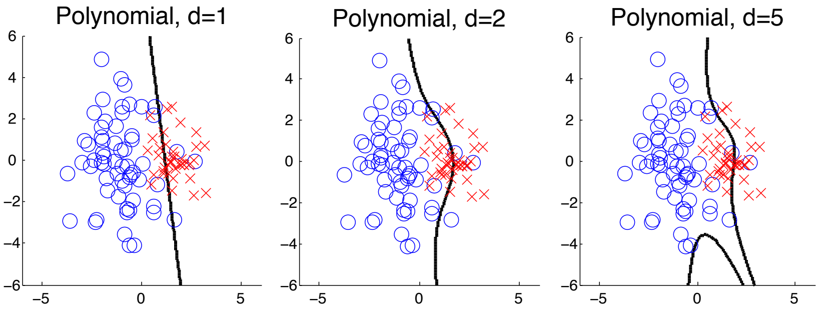


Figure 4.6: The effect of the degree parameter when using a polynomial kernel.

This also illustrates that the *linear* kernel is a special case of the polynomial kernel, when $d = 1$.

4.5.3 The RBF kernel

A very popular kernel is the Radial Basis Function (RBF), also known as the Gaussian kernel:

$$K(\mathbf{x_i}, \mathbf{x}') = e^{-\gamma(\mathbf{x}_i - \mathbf{x}')^2} \tag{4.19}$$

Where $\gamma = \frac{1}{2\sigma^2}$ controls the width of the Gaussian, and σ is the standard deviation. So, a large γ means a small standard deviation, hence a very narrow RBF kernel. This gives rise to the name 'inverse width' parameter for γ.

A small γ implies a very smooth fit, with a large region of influence for a given training data point. Similarly, narrow RBF width (large γ) means the influence of training points is is much more localised. In the limit ($\gamma \to \infty$), a data point is not connected to any other data point.

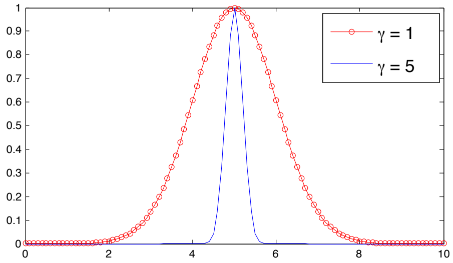


Figure 4.7: Varying the width of the RBF kernel.

The larger γ is, the more overfitting we might expect.
A smaller γ will produce smooth boundaries, possibly underfitting.

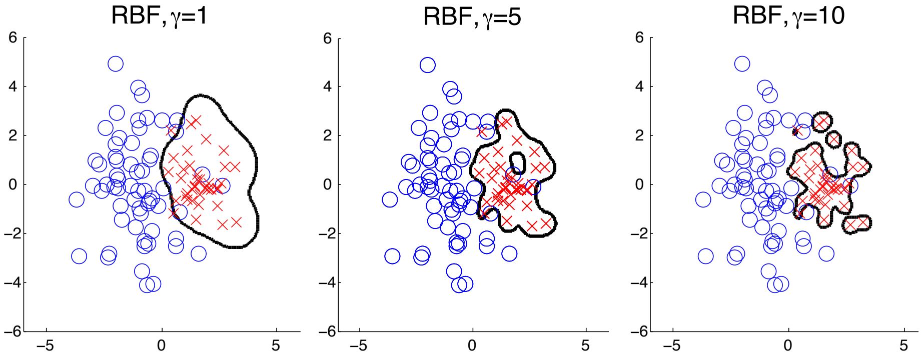


Figure 4.8: A non-linear problem being solved by an SVM with an RBF kernel. The effect of γ, the inverse-width of the Gaussian kernel, is illustrated.

4.5.4 Model Selection with SVMs

We have just introduced some new parameters, that need to be set, via model selection, i.e. we pick the SVM configuration with the best performance. Techniques to find a good setting are fully described in the suggested reading on the course website "A Practical Guide to SVM Classification". It should be noted that **the parameters are not independent**, so for example, below is what happens when we try to set the C (slack variable penalty) and γ parameters, at the same time.

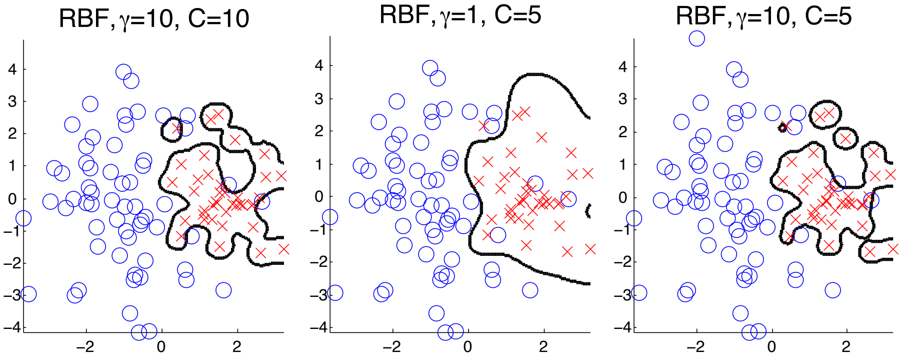


Figure 4.9: The effect of varying two things at once!

SELF-TEST

A learning algorithm constructs a kind of “data structure”, which is known in Machine Learning terminology as a __________. Perceptrons and Logistic Regressions are examples of this, which can only solve ___________ separable problems.

The SVM is a __________ model, just like the Perceptron. The first difference is that the SVM can identify the boundary with ____________ ________, which decreases its generalisation error. The datapoints lying on the ________ are known as ________ vectors. The second difference is that the SVM uses the ______________ trick and projects into a ______________ dimensional space, then solves the linear classification problem in that space instead of the original. All of this assumes that the data will be _________ separable in the new space. The third difference is the use of ___________ variables. This means it can solve ______________ problems by allowing some points to not be constrained to be correct.

Although the SVM fits a linear boundary exactly, without need for an iterative algorithm, it still has parameters. The slack variables are controlled by the ____ parameter. If we use a Gaussian kernel, the ______ parameter controls the width of the Gaussian.

4.6 What you should know by now

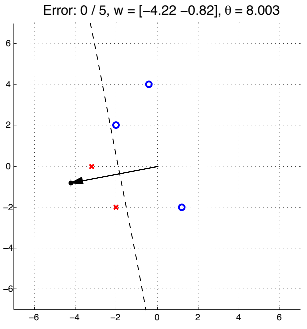
By now you should know (and be able to define) the following :

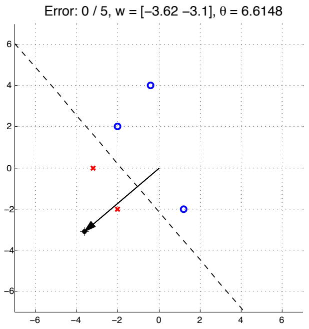
Margin
Hinge loss
The SVM objective function
Slack variables
Kernels and the Kernel trick
Support Vector

A handy table to summarise what we've done so far....

Model name	**Loss Function**	**Learning Algorithm**
Decision Stump	Zero-One	Line Search
Perceptron	-	Perceptron Algorithm
Logistic Regression	Log	Gradient Descent
Support Vector Machine	Hinge (+margin)	Quadratic Programming

SELF-TEST
Which one of these two solutions is likely to be better for testing data? Why?





4.7 Optional: The Original SVM Paper

On the following pages is the research article where SVMs were first invented, published in the Computational Learning Theory conference, 1992. Sometimes going back to the source is the best way to understand something. Enjoy.

A Training Algorithm for Optimal Margin Classifiers

Bernhard E. Boser*
EECS Department
University of California
Berkeley, CA 94720
boser@eecs.berkeley.edu

Isabelle M. Guyon
AT&T Bell Laboratories
50 Fremont Street, 6th Floor
San Francisco, CA 94105
isabelle@neural.att.com

Vladimir N. Vapnik
AT&T Bell Laboratories
Crawford Corner Road
Holmdel, NJ 07733
vlad@neural.att.com

Abstract

A training algorithm that maximizes the margin between the training patterns and the decision boundary is presented. The technique is applicable to a wide variety of classifiaction functions, including Perceptrons, polynomials, and Radial Basis Functions. The effective number of parameters is adjusted automatically to match the complexity of the problem. The solution is expressed as a linear combination of supporting patterns. These are the subset of training patterns that are closest to the decision boundary. Bounds on the generalization performance based on the leave-one-out method and the VC-dimension are given. Experimental results on optical character recognition problems demonstrate the good generalization obtained when compared with other learning algorithms.

1 INTRODUCTION

Good generalization performance of pattern classifiers is achieved when the capacity of the classification function is matched to the size of the training set. Classifiers with a large number of adjustable parameters and therefore large capacity likely learn the training set without error, but exhibit poor generalization. Conversely, a classifier with insufficient capacity might not be able to learn the task at all. In between, there is an optimal capacity of the classifier which minimizes the expected generalization error for a given amount of training data. Both experimental evidence and theoretical studies [GBD92, Moo92, GVB$^+$92, Vap82, BH89, TLS89, Mac92] link the generalization of a classifier to the error on the training examples and the complexity of the classifier. Methods such as structural risk minimization [Vap82] vary the complexity of the classification function in order to optimize the generalization.

In this paper we describe a training algorithm that automatically tunes the capacity of the classification function by maximizing the margin between training examples and class boundary [KM87], optionally after removing some atypical or meaningless examples from the training data. The resulting classification function depends only on so-called supporting patterns [Vap82]. These are those training examples that are closest to the decision boundary and are usually a small subset of the training data.

It will be demonstrated that maximizing the margin amounts to minimizing the maximum loss, as opposed to some average quantity such as the mean squared error. This has several desirable consequences. The resulting classification rule achieves an errorless separation of the training data if possible. Outliers or meaningless patterns are identified by the algorithm and can therefore be eliminated easily with or without supervision. This contrasts classifiers based on minimizing the mean squared error, which quietly ignore atypical patterns. Another advantage of maximum margin classifiers is that the sensitivity of the classifier to limited computational accuracy is minimal compared to other separations with smaller margin. In analogy to [Vap82, HLW88] a bound on the generalization performance is obtained with the "leave-one-out" method. For the maximum margin classifier it is the ratio of the number of linearly independent supporting patterns to the number of training examples. This bound is tighter than a bound based on the capacity of the classifier family.

The proposed algorithm operates with a large class of decision functions that are linear in their parameters but not restricted to linear dependences in the input components. Perceptrons [Ros62], polynomial classifiers, neural networks with one hidden layer, and Radial Basis Function (RBF) or potential function classifiers [ABR64, BL88, MD89] fall into this class. As pointed out by several authors [ABR64, DH73, PG90], Percep-

*Part of this work was performed while B. Boser was with AT&T Bell Laboratories. He is now at the University of California, Berkeley.

COLT'92-7/92/PA,USA
© 1992 ACM 0-89791-498-8/92/0007/0144...$1.50

trons have a dual kernel representation implementing the same decision function. The optimal margin algorithm exploits this duality both for improved efficiency and flexibility. In the dual space the decision function is expressed as a linear combination of basis functions parametrized by the supporting patterns. The supporting patterns correspond to the class centers of RBF classifiers and are chosen automatically by the maximum margin training procedure. In the case of polynomial classifiers, the Perceptron representation involves an untractable number of parameters. This problem is overcome in the dual space representation, where the classification rule is a weighted sum of a kernel function [Pog75] for each supporting pattern. High order polynomial classifiers with very large training sets can therefore be handled efficiently with the proposed algorithm.

The training algorithm is described in Section 2. Section 3 summarizes important properties of optimal margin classifiers. Experimental results are reported in Section 4.

2 MAXIMUM MARGIN TRAINING ALGORITHM

The maximum margin training algorithm finds a decision function for pattern vectors $\mathbf{x}$ of dimension n belonging to either of two classes A and B. The input to the training algorithm is a set of p examples $\mathbf{x}_i$ with labels y_i:

$$(\mathbf{x}_1, y_1),\ (\mathbf{x}_2, y_2),\ (\mathbf{x}_3, y_3),\ \ldots,\ (\mathbf{x}_p, y_p) \qquad (1)$$

$$\text{where } \begin{cases} y_k = 1 & \text{if } \mathbf{x}_k \in \text{class A} \\ y_k = -1 & \text{if } \mathbf{x}_k \in \text{class B.} \end{cases}$$

From these training examples the algorithm finds the parameters of the decision function $D(\mathbf{x})$ during a learning phase. After training, the classification of unknown patterns is predicted according to the following rule:

$$\begin{array}{ll} \mathbf{x} \in \text{A} & \text{if } D(\mathbf{x}) > 0 \\ \mathbf{x} \in \text{B} & \text{otherwise.} \end{array} \qquad (2)$$

The decision functions must be linear in their parameters but are not restricted to linear dependences of $\mathbf{x}$. These functions can be expressed either in direct, or in dual space. The direct space notation is identical to the Perceptron decision function [Ros62]:

$$D(\mathbf{x}) = \sum_{i=1}^{N} w_i \varphi_i(\mathbf{x}) + b. \qquad (3)$$

In this equation the φ_i are predefined functions of $\mathbf{x}$, and the w_i and b are the adjustable parameters of the decision function. Polynomial classifiers are a special case of Perceptrons for which $\varphi_i(\mathbf{x})$ are products of components of $\mathbf{x}$.

In the dual space, the decision functions are of the form

$$D(\mathbf{x}) = \sum_{k=1}^{p} \alpha_k K(\mathbf{x}_k, \mathbf{x}) + b, \qquad (4)$$

The coefficients α_k are the parameters to be adjusted and the $\mathbf{x}_k$ are the training patterns. The function K is a predefined kernel, for example a potential function [ABR64] or any Radial Basis Function [BL88, MD89]. Under certain conditions [CH53], symmetric kernels possess finite or infinite series expansions of the form

$$K(\mathbf{x}, \mathbf{x}') = \sum_i \varphi_i(\mathbf{x})\, \varphi_i(\mathbf{x}'). \qquad (5)$$

In particular, the kernel $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + 1)^q$ corresponds to a polynomial expansion $\varphi(\mathbf{x})$ of order q [Pog75].

Provided that the expansion stated in equation 5 exists, equations 3 and 4 are dual representations of the same decision function and

$$w_i = \sum_{k=1}^{p} \alpha_k \varphi_i(\mathbf{x}_k). \qquad (6)$$

The parameters w_i are called direct parameters, and the α_k are referred to as dual parameters.

The proposed training algorithm is based on the "generalized portrait" method described in [Vap82] that constructs separating hyperplanes with maximum margin. Here this algorithm is extended to train classifiers linear in their parameters. First, the margin between the class boundary and the training patterns is formulated in the direct space. This problem description is then transformed into the dual space by means of the Lagrangian. The resulting problem is that of maximizing a quadratic form with constraints and is amenable to efficient numeric optimization algorithms [Lue84].

2.1 MAXIMIZING THE MARGIN IN THE DIRECT SPACE

In the direct space the decision function is

$$D(\mathbf{x}) = \mathbf{w} \cdot \varphi(\mathbf{x}) + b, \qquad (7)$$

where $\mathbf{w}$ and $\varphi(\mathbf{x})$ are N dimensional vectors and b is a bias. It defines a separating hyperplane in φ-space. The distance between this hyperplane and pattern $\mathbf{x}$ is $D(\mathbf{x})/||\mathbf{w}||$ (Figure 1). Assuming that a separation of the training set with margin M between the class boundary and the training patterns exists, all training patterns fulfill the following inequality:

$$\frac{y_k D(\mathbf{x}_k)}{||\mathbf{w}||} \geq M. \qquad (8)$$

The objective of the training algorithm is to find the parameter vector $\mathbf{w}$ that maximizes M:

$$M^* = \max_{\mathbf{w}, ||\mathbf{w}||=1} M \qquad (9)$$

$$\text{subject to } \ y_k D(\mathbf{x}_k) \geq M, \qquad k = 1, 2, \ldots, p.$$

The bound M^* is attained for those patterns satisfying

$$\min_k y_k D(\mathbf{x}_k) = M^*. \qquad (10)$$

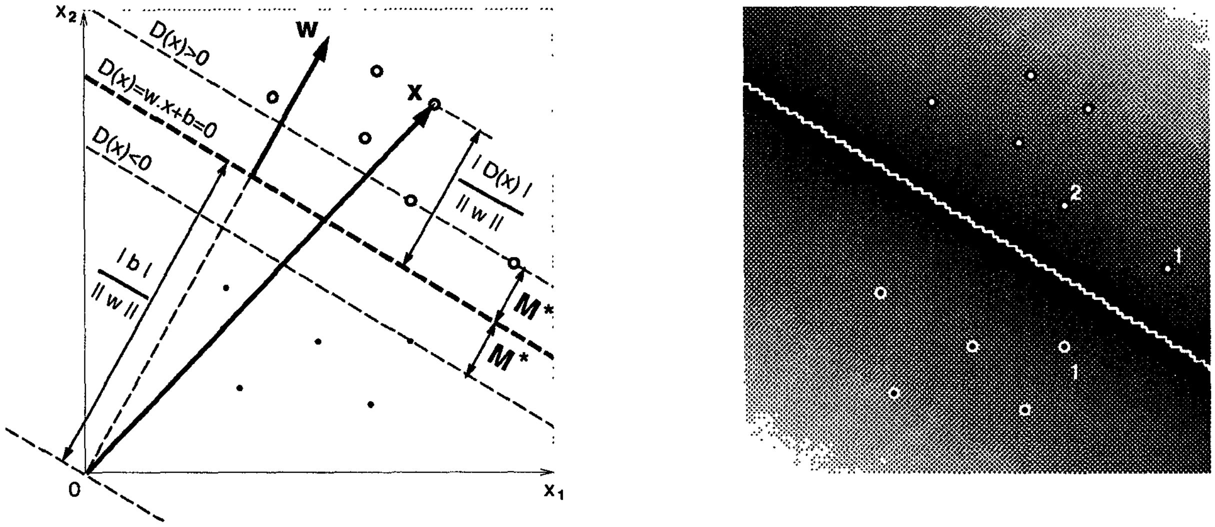


Figure 1: Maximum margin linear decision function $D(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$ ($\varphi = \mathbf{x}$). The gray levels encode the absolute value of the decision function (solid black corresponds to $D(\mathbf{x}) = 0$). The numbers indicate the supporting patterns.

These patterns are called the supporting patterns of the decision boundary.

A decision function with maximum margin is illustrated in figure 1. The problem of finding a hyperplane in φ-space with maximum margin is therefore a minimax problem:

$$\max_{\mathbf{w}, ||\mathbf{w}||=1} \min_k y_k D(\mathbf{x}_k). \qquad (11)$$

The norm of the parameter vector in equations 9 and 11 is fixed to pick one of an infinite number of possible solutions that differ only in scaling. Instead of fixing the norm of $\mathbf{w}$ to take care of the scaling problem, the product of the margin M and the norm of a weight vector $\mathbf{w}$ can be fixed.

$$M||\mathbf{w}|| = 1. \qquad (12)$$

Thus, maximizing the margin M is equivalent to minimizing the norm $||\mathbf{w}||$.[1] Then the problem of finding a maximum margin separating hyperplane $\mathbf{w}^*$ stated in 9 reduces to solving the following quadratic problem:

$$\min_{\mathbf{w}} ||\mathbf{w}||^2 \qquad (13)$$
$$\text{under conditions} \quad y_k D(\mathbf{x}_k) \geq 1, \qquad k = 1, 2, \ldots, p.$$

The maximum margin is $M^* = 1/||\mathbf{w}^*||$.

In principle the problem stated in 13 can be solved directly with numerical techniques. However, this approach is impractical when the dimensionality of the φ-space is large or infinite. Moreover, no information is gained about the supporting patterns.

2.2 MAXIMIZING THE MARGIN IN THE DUAL SPACE

Problem 13 can be transformed into the dual space by means of the Lagrangian [Lue84]

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{k=1}^{p} \alpha_k \left[y_k D(\mathbf{x}_k) - 1 \right] (14)$$
$$\text{subject to} \quad \alpha_k \geq 0, \qquad k = 1, 2, \ldots, p.$$

The factors α_k are called Lagrange multipliers or Kühn-Tucker coefficients and satisfy the conditions

$$\alpha_k \left(y_k D(\mathbf{x}_k) - 1 \right) = 0, \qquad k = 1, 2, \ldots, p. \qquad (15)$$

The factor one half has been included for cosmetic reasons; it does not change the solution.

The optimization problem 13 is equivalent to searching a saddle point of the function $L(\mathbf{w}, b, \boldsymbol{\alpha})$. This saddle point is a the minimum of $L(\mathbf{w}, b, \boldsymbol{\alpha})$ with respect to $\mathbf{w}$, and a maximum with respect to $\boldsymbol{\alpha}$ ($\alpha_k \geq 0$). At the solution, the following necessary condition is met:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w}^* - \sum_{k=1}^{p} \alpha_k^* y_k \varphi_k = 0,$$

hence

$$\mathbf{w}^* = \sum_{k=1}^{P} \alpha_k^* y_k \varphi_k. \qquad (16)$$

The patterns which satisfy $y_k D(\mathbf{x}_k) = 1$ are the supporting patterns. According to equation 16, the vector $\mathbf{w}^*$ that specifies the hyperplane with maximum margin is a linear combination of only the supporting patterns, which are those patterns for which $\alpha_k^* \neq 0$. Usually the number of supporting patterns is much smaller than the number p of patterns in the training set.

[1] If the training data is not linearly separable the maximum margin may be negative. In this case, $M||\mathbf{w}|| = -1$ is imposed. Maximizing the margin is then equivalent to maximizing $||\mathbf{w}||$.

The dependence of the Lagrangian $L(\mathbf{w}, b, \boldsymbol{\alpha})$ on the weight vector $\mathbf{w}$ is removed by substituting the expansion of $\mathbf{w}^*$ given by equation 16 for $\mathbf{w}$. Further transformations using 3 and 5 result in a Lagrangian which is a function of the parameters $\boldsymbol{\alpha}$ and the bias b only:

$$J(\boldsymbol{\alpha}, b) = \sum_{k=1}^{p} \alpha_k (1 - b y_k) - \frac{1}{2} \boldsymbol{\alpha} \cdot \boldsymbol{H} \cdot \boldsymbol{\alpha}, \quad (17)$$
$$\text{subject to } \alpha_k \geq 0, \qquad k = 1, 2, \ldots, p.$$

Here $\boldsymbol{H}$ is a square matrix of size $p \times p$ with elements

$$H_{kl} = y_k y_l K(\mathbf{x}_k, \mathbf{x}_l).$$

In order for a unique solution to exist, $\boldsymbol{H}$ must be positive definite. For fixed bias b, the solution $\boldsymbol{\alpha}^*$ is obtained by maximizing $J(\boldsymbol{\alpha}, b)$ under the conditions $\alpha_k \geq 0$. Based on equations 7 and 16, the resulting decision function is of the form

$$\begin{aligned} D(\mathbf{x}) &= \mathbf{w}^* \cdot \boldsymbol{\varphi}(\mathbf{x}) + b & (18) \\ &= \sum_k y_k \alpha_k^* K(\mathbf{x}_k, \mathbf{x}) + b, \qquad \alpha_k^* \geq 0, \end{aligned}$$

where only the supporting patterns appear in the sum with nonzero weight.

The choice of the bias b gives rise to several variants of the algorithm. The two considered here are

1. The bias can be fixed a priori and not subjected to training. This corresponds to the "Generalized Portrait Technique" described in [Vap82].
2. The cost function 17 can be optimized with respect to $\mathbf{w}$ and b. This approach gives the largest possible margin M^* in φ-space [VC74].

In both cases the solution is found with standard nonlinear optimization algorithms for quadratic forms with linear constraints [Lue84, Loo72]. The second approach gives the largest possible margin. There is no guarantee, however, that this solution exhibits also the best generalization performance.

A strategy to optimize the margin with respect to both $\mathbf{w}$ and b is described in [Vap82]. It solves problem 17 for differences of pattern vectors to obtain $\boldsymbol{\alpha}^*$ independent of the bias, which is computed subsequently. The margin in φ-space is maximized when the decision boundary is halfway between the two classes. Hence the bias b^* is obtained by applying 18 to two arbitrary supporting patterns $\mathbf{x}_A \in$ class A and $\mathbf{x}_B \in$ class B and taking into account that $D(\mathbf{x}_A) = 1$ and $D(\mathbf{x}_B) = -1$.

$$\begin{aligned} b^* &= -\frac{1}{2} (\mathbf{w}^* \cdot \boldsymbol{\varphi}(\mathbf{x}_A) + \mathbf{w}^* \cdot \boldsymbol{\varphi}(\mathbf{x}_B)) & (19) \\ &= -\frac{1}{2} \sum_{k=1}^{p} y_k \alpha_k^* \left[K(\mathbf{x}_A, \mathbf{x}_k) + K(\mathbf{x}_B, \mathbf{x}_k) \right]. \end{aligned}$$

The dimension of problem 17 equals the size of the training set, p. To avoid the need to solve a dual problem of exceedingly large dimensionality, the training data is divided into chunks that are processed iteratively [Vap82]. The maximum margin hypersurface is constructed for the first chunk and a new training set is formed consisting of the supporting patterns from the solution and those patterns $\mathbf{x}_k$ in the second chunk of the training set for which $y_k D(\mathbf{x}_k) < 1 - \epsilon$. A new classifier is trained and used to construct a training set consisting of supporting patterns and examples from the first three chunks which satisfy $y_k D(\mathbf{x}_k) < 1 - \epsilon$. This process is repeated until the entire training set is separated.

3 PROPERTIES OF THE ALGORITHM

In this Section, we highlight some important aspects of the optimal margin training algorithm. The description is split into a discussion of the qualities of the resulting classifier, and computational considerations. Classification performance advantages over other techniques will be illustrated in the Section on experimental results.

3.1 PROPERTIES OF THE SOLUTION

Since maximizing the margin between the decision boundary and the training patterns is equivalent to maximizing a quadratic form in the positive quadrant, there are no local minima and the solution is always unique if $\boldsymbol{H}$ has full rank. At the optimum

$$J(\boldsymbol{\alpha}^*) = \frac{1}{2} \|\mathbf{w}^*\|^2 = \frac{1}{2\,(M^*)^2} = \frac{1}{2} \sum_{k=1}^{p} \alpha_k^*. \qquad (20)$$

The uniqueness of the solution is a consequence of the maximum margin cost function and represents an important advantage over other algorithms for which the solution depends on the initial conditions or other parameters that are difficult to control.

Another benefit of the maximum margin objective is its insensitivity to small changes of the parameters $\mathbf{w}$ or $\boldsymbol{\alpha}$. Since the decision function $D(\mathbf{x})$ is a linear function of $\mathbf{w}$ in the direct, and of $\boldsymbol{\alpha}$ in the dual space, the probability of misclassifications due to parameter variations of the components of these vectors is minimized for maximum margin. The robustness of the solution—and potentially its generalization performance—can be increased further by omitting some supporting patterns from the solution. Equation 20 indicates that the largest increase in the maximum margin M^* occurs when the supporting patterns with largest α_k are eliminated. The elimination can be performed automatically or with assistance from a supervisor. This feature gives rise to other important uses of the optimum margin algorithm in database cleaning applications [MGB+92].

Figure 2 compares the decision boundary for a maximum margin and mean squared error (MSE) cost functions. Unlike the MSE based decision function which simply ignores the outlier, optimal margin classifiers are very sensitive to atypical patterns that are close to the

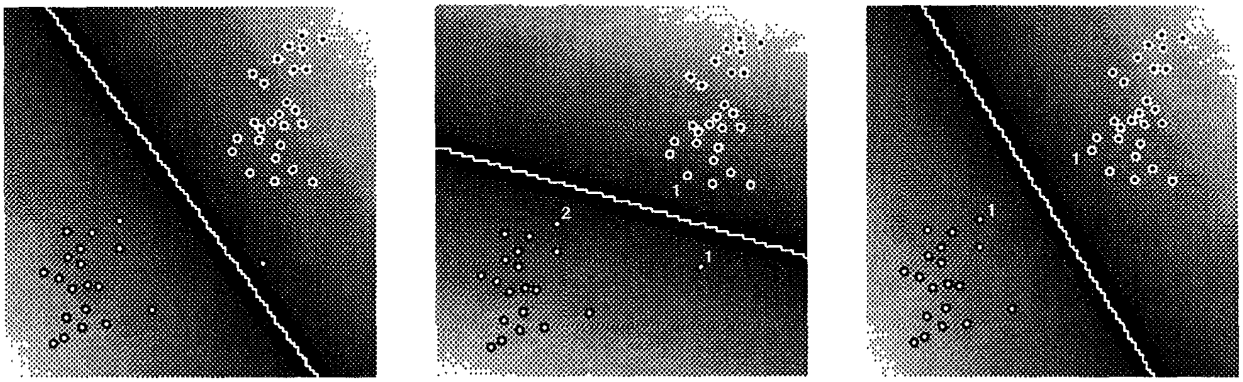


Figure 2: Linear decision boundary for MSE (left) and maximum margin cost functions (middle, right) in the presence of an outlier. In the rightmost picture the outlier has been removed. The numbers reflect the ranking of supporting patterns according to the magnitude of their Lagrange coefficient α_k for each class individually.

decision boundary. These examples are readily identified as those with the largest α_k and can be eliminated either automatically or with supervision. Hence, optimal margin classifiers give complete control over the handling of outliers, as opposed to quietly ignoring them.

The optimum margin algorithm performs automatic capacity tuning of the decision function to achieve good generalization. An estimate for an upper bound of the generalization error is obtained with the "leave-one-out" method: A pattern $\mathbf{x}_k$ is removed from the training set. A classifier is then trained on the remaining patterns and tested on $\mathbf{x}_k$. This process is repeated for all p training patterns. The generalization error is estimated by the ratio of misclassified patterns over p. For a maximum margin classifier, two cases arise: If $\mathbf{x}_k$ is not a supporting pattern, the decision boundary is unchanged and $\mathbf{x}_k$ will be classified correctly. If $\mathbf{x}_k$ is a supporting pattern, two cases are possible:

1. The pattern $\mathbf{x}_k$ is linearly dependent on the other supporting patterns. In this case it will be classified correctly.
2. $\mathbf{x}_k$ is linearly independent from the other supporting patterns. In this case the outcome is uncertain. In the worst case m' linearly independent supporting patterns are misclassified when they are omitted from the training data.

Hence the frequency of errors obtained by this method is at most m'/p, and has no direct relationship with the number of adjustable parameters. The number of linearly independent supporting patterns m' itself is bounded by $\min(N, p)$. This suggests that the number of supporting patterns is related to an effective capacity of the classifier that is usually much smaller than the VC-dimension, $N + 1$ [Vap82, HLW88].

In polynomial classifiers, for example, $N \approx n^q$, where n is the dimension of $\mathbf{x}$-space and q is the order of the polynomial. In practice, $m \leq p \ll N$, i.e. the number of supporting patterns is much smaller than the dimension of the φ-space. The capacity tuning realized by the maximum margin algorithm is essential to get generalization with high-order polynomial classifiers.

3.2 COMPUTATIONAL CONSIDERATIONS

Speed and convergence are important practical considerations of classification algorithms. The benefit of the dual space representation to reduce the number of computations required for example for polynomial classifiers has been pointed out already. In the dual space, each evaluation of the decision function $D(\mathbf{x})$ requires m evaluations of the kernel function $K(\mathbf{x}_k, \mathbf{x})$ and forming the weighted sum of the results. This number can be further reduced through the use of appropriate search techniques which omit evaluations of K that yield negligible contributions to $D(\mathbf{x})$ [Omo91].

Typically, the training time for a separating surface from a database with several thousand examples is a few minutes on a workstation, when an efficient optimization algorithm is used. All experiments reported in the next section on a database with 7300 training examples took less than five minutes of CPU time per separating surface. The optimization was performed with an algorithm due to Powell that is described in [Lue84] and available from public numerical libraries.

Quadratic optimization problems of the form stated in 17 can be solved in polynomial time with the Ellipsoid method [NY83]. This technique finds first a hyperspace that is guaranteed to contain the optimum; then the volume of this space is reduced iteratively by a constant fraction. The algorithm is polynomial in the number of free parameters p and the encoding size (i. e. the accuracy of the problem and solution). In practice, however, algorithms without guaranteed polynomial convergence are more efficient.

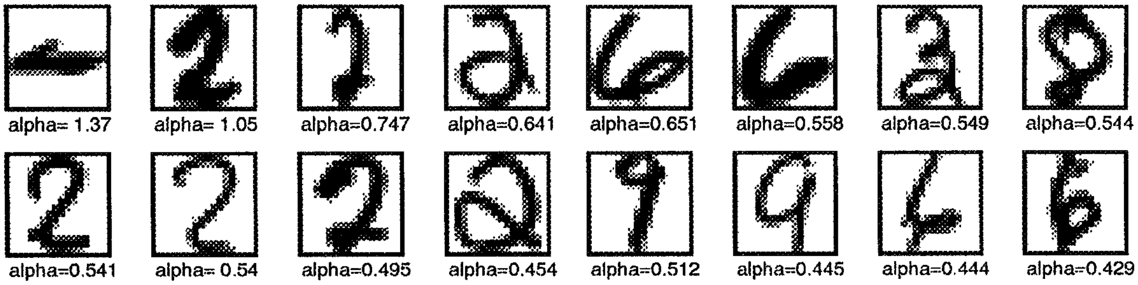


Figure 3: Supporting patterns from database DB2 for class 2 before cleaning. The patterns are ranked according to α_k.

4 EXPERIMENTAL RESULTS

The maximum margin training algorithm has been tested on two databases with images of handwritten digits. The first database (DB1) consists of 1200 clean images recorded from ten subjects. Half of this data is used for training, and the other half is used to evaluate the generalization performance. A comparative analysis of the performance of various classification methods on DB1 can be found in [GVB+92, GPP+89, GBD92]. The other database (DB2) used in the experiment consists of 7300 images for training and 2000 for testing and has been recorded from actual mail pieces. Results for this data have been reported in several publications, see e.g. [CBD+90]. The resolution of the images in both databases is 16 by 16 pixels.

In all experiments, the margin is maximized with respect to $\mathbf{w}$ and b. Ten hypersurfaces, one per class, are used to separate the digits. Regardless of the difficulty of the problem—measured for example by the number of supporting patterns found by the algorithm—the same similarity function $K(\mathbf{x}, \mathbf{x}')$ and preprocessing is used for all hypersurfaces of one experiment. The results obtained with different choices of K corresponding to linear hyperplanes, polynomial classifiers, and basis functions are summarized below. The effect of smoothing is investigated as a simple form of preprocessing.

For linear hyperplane classifiers, corresponding to the similarity function $K(\mathbf{x}, \mathbf{x}') = \mathbf{x} \cdot \mathbf{x}'$, the algorithm finds an errorless separation for database DB1. The percentage of errors on the test set is 3.2 %. This result compares favorably to hyperplane classifiers which minimize the mean squared error (backpropagation or pseudo-inverse), for which the error on the test set is 12.7 %.

Database DB2 is also linearly separable but contains several meaningless patterns. Figure 3 shows the supporting patterns with large Lagrange multipliers α_k for the hyperplane for class 2. The percentage of misclassifications on the test set of DB2 drops from 15.2 % without cleaning to 10.5 % after removing meaningless and ambiguous patterns.

Better performance has been achieved with both databases using multilayer neural networks or other classification functions with higher capacity than linear subdividing planes. Tests with polynomial classifiers of order q, for which $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + 1)^q$, give the following error rates and average number of supporting patterns per hypersurface, $<m>$. This average is computed as the total number of supporting patterns divided by the number of decision functions. Patterns that support more than one hypersurface are counted only once in the total. For comparison, the dimension N of φ-space is also listed.

	DB1		DB2		
q	error	$<m>$	error	$<m>$	N
1 (linear)	3.2 %	36	10.5 %	97	256
2	1.5 %	44	5.8 %	89	$3 \cdot 10^4$
3	1.7 %	50	5.2 %	79	$8 \cdot 10^7$
4			4.9 %	72	$4 \cdot 10^9$
5			5.2 %	69	$1 \cdot 10^{12}$

The results obtained for DB2 show a strong decrease of the number of supporting patterns from a linear to a third order polynomial classification function and an equivalently significant decrease of the error rate. Further increase of the order of the polynomial has little effect on either the number of supporting patterns or the performance, unlike the dimension of φ-space, N, which increases exponentially. The lowest error rate, 4.9 % is obtained with a forth order polynomial and is slightly better than the 5.1 % reported for a five layer neural network with a sophisticated architecture [CBD+90], which has been trained and tested on the same data.

In the above experiment, the performance changes drastically between first and second order polynomials. This may be a consequence of the fact that maximum VC-dimension of an q-th order polynomial classifier is equal to the dimension n of the patterns to the q-th power and thus much larger than n. A more gradual change of the VC-dimension is possible when the function K is chosen to be a power series, for example

$$K(\mathbf{x}, \mathbf{x}') = \exp(\gamma \mathbf{x} \cdot \mathbf{x}') - 1. \tag{21}$$

In this equation the parameter γ is used to vary the VC-dimension gradually. For small values of γ, equation 21 approaches a linear classifier with VC-dimension at

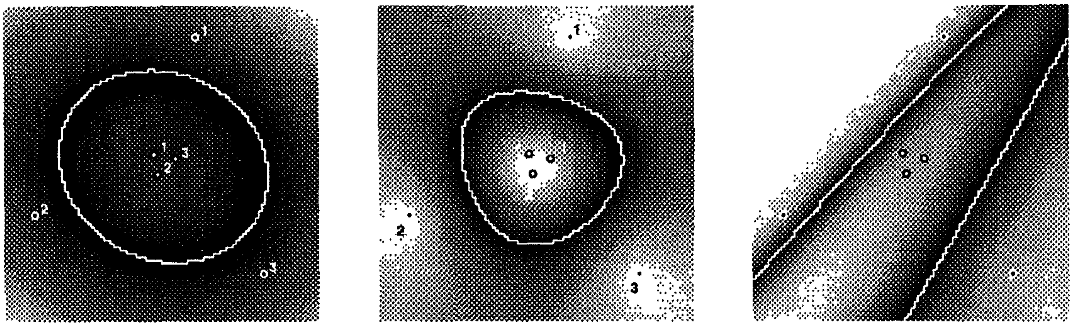


Figure 4: Decision boundaries for maximum margin classifiers with second order polynomial decision rule $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x} \cdot \mathbf{x}' + 1)^2$ (left) and an exponential RBF $K(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|/2)$ (middle). The rightmost picture shows the decision boundary of a two layer neural network with two hidden units trained with backpropagation.

most equal to the dimension n of the patterns plus one. Experiments with database DB1 lead to a slightly better performance than the 1.5 % obtained with a second order polynomial classifier:

γ	DB1
0.25	2.3 %
0.50	2.2 %
0.75	1.3 %
1.00	1.5 %

When $K(\mathbf{x}, \mathbf{x}')$ is chosen to be the hyperbolic tangent, the resulting classifier can be interpreted as a neural network with one hidden layer with m hidden units. The supporting patterns are the weights in the first layer, and the coefficients α_k the weights of the second, linear layer. The number of hidden units is chosen by the training algorithm to maximize the margin between the classes A and B. Substituting the hyperbolic tangent for the exponential function did not lead to better results in our experiments.

The importance of a suitable preprocessing to incorporate knowledge about the task at hand has been pointed out by many researchers. In optical character recognition, preprocessings that introduce some invariance to scaling, rotation, and other distortions are particularly important [SLD92]. As in [GVB+92], smoothing is used to achieve insensitivity to small distortions. The table below lists the error on the test set for different amounts of smoothing. A second order polynomial classifier was used for database DB1, and a forth order polynomial for DB2. The smoothing kernel is Gaussian with standard deviation σ.

	DB1		DB2	
σ	error	$<m>$	error	$<m>$
no smoothing	1.5 %	44	4.9 %	72
0.5	1.3 %	41	4.6 %	73
0.8	0.8 %	36	5.0 %	79
1.0	0.3 %	31	6.0 %	83
1.2	0.8 %	31		

The performance improved considerably for DB1. For DB2 the improvement is less significant and the optimum was obtained for less smoothing than for DB1. This is expected since the number of training patterns in DB2 is much larger than in DB1 (7000 versus 600). A higher performance gain can be expected for more selective hints than smoothing, such as invariance to small rotations or scaling of the digits [SLD92].

Better performance might be achieved with other similarity functions $K(\mathbf{x}, \mathbf{x}')$. Figure 4 shows the decision boundary obtained with a second order polynomial and a radial basis function (RBF) maximum margin classifier with $K(\mathbf{x}, \mathbf{x}') = \exp(-\|\mathbf{x} - \mathbf{x}'\|/2)$. The decision boundary of the polynomial classifier is much closer to one of the two classes. This is a consequence of the nonlinear transform from φ-space to $\mathbf{x}$-space of polynomials which realizes a position dependent scaling of distance. Radial Basis Functions do not exhibit this problem. The decision boundary of a two layer neural network trained with backpropagation is shown for comparison.

5 CONCLUSIONS

Maximizing the margin between the class boundary and training patterns is an alternative to other training methods optimizing cost functions such as the mean squared error. This principle is equivalent to minimizing the maximum loss and has a number of important features. These include automatic capacity tuning of the classification function, extraction of a small number of supporting patterns from the training data that are relevant for the classification, and uniqueness of the solution. They are exploited in an efficient learning algorithm for classifiers linear in their parameters with very large capacity, such as high order polynomial or RBF classifiers. Key is the representation of the decision function in a dual space which is of much lower dimensionality than the feature space.

The efficiency and performance of the algorithm have been demonstrated on handwritten digit recognition problems. The achieved performance matches that of sophisticated classifiers, even though no task specific knowledge has been used. The training algorithm is polynomial in the number of training patterns, even in cases when the dimension of the solution space (φ-space) is exponential or infinite. The training time in all experiments was less than an hour on a workstation.

Acknowledgements

We wish to thank our colleagues at UC Berkeley and AT&T Bell Laboratories for many suggestions and stimulating discussions. Comments by L. Bottou, C. Cortes, S. Sanders, S. Solla, A. Zakhor, and the reviewers are gratefully acknowledged. We are especially indebted to R. Baldick and D. Hochbaum for investigating the polynomial convergence property, S. Hein for providing the code for constrained nonlinear optimization, and D. Haussler and M. Warmuth for help and advice regarding performance bounds.

References

[ABR64] M.A. Aizerman, E.M. Braverman, and L.I. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[BH89] E. B. Baum and D. Haussler. What size net gives valid generalization? *Neural Computation*, 1(1):151–160, 1989.

[BL88] D. S. Broomhead and D. Lowe. Multivariate functional interpolation and adaptive networks. *Complex Systems*, 2:321 – 355, 1988.

[CBD+90] Yann Le Cun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Larry D. Jackel. Handwritten digit recognition with a back-propagation network. In David S. Touretzky, editor, *Neural Information Processing Systems*, volume 2, pages 396–404. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[CH53] R. Courant and D. Hilbert. *Methods of mathematical physics.* Interscience, New York, 1953.

[DH73] R.O. Duda and P.E. Hart. *Pattern Classification And Scene Analysis.* Wiley and Son, 1973.

[GBD92] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4 (1):1 – 58, 1992.

[GPP+89] I. Guyon, I. Poujaud, L. Personnaz, G. Dreyfus, J. Denker, and Y. LeCun. Comparing different neural network architectures for classifying handwritten digits. In *Proc. Int. Joint Conf. Neural Networks.* Int. Joint Conference on Neural Networks, 1989.

[GVB+92] Isabelle Guyon, Vladimir Vapnik, Bernhard Boser, Leon Bottou, and Sara Solla. Structural risk minimization for character recognition. In David S. Touretzky, editor, *Neural Information Processing Systems*, volume 4. Morgan Kaufmann Publishers, San Mateo, CA, 1992. To appear.

[HLW88] David Haussler, Nick Littlestone, and Manfred Warmuth. Predicting 0,1-functions on randomly drawn points. In *Proceedings of the 29th Annual Symposium on the Foundations of Computer Science*, pages 100–109. IEEE, 1988.

[KM87] W. Krauth and M. Mezard. Learning algorithms with optimal stability in neural networks. *J. Phys. A: Math. gen.*, 20:L745, 1987.

[Loo72] F. A. Lootsma, editor. *Numerical Methods for Non-linear Optimization.* Academic Press, London, 1972.

[Lue84] David Luenberger. *Linear and Nonlinear Programming.* Addison-Wesley, 1984.

[Mac92] D. MacKay. A practical bayesian framework for backprop networks. In David S. Touretzky, editor, *Neural Information Processing Systems*, volume 4. Morgan Kaufmann Publishers, San Mateo, CA, 1992. To appear.

[MD89] J. Moody and C. Darken. Fast learning in networks of locally tuned processing units. *Neural Computation*, 1 (2):281 – 294, 1989.

[MGB+92] N. Matic, I. Guyon, L. Bottou, J. Denker, and V. Vapnik. Computer-aided cleaning of large databases for character recognition. In *Digest ICPR.* ICPR, Amsterdam, August 1992.

[Moo92] J. Moody. Generalization, weight decay, and architecture selection for nonlinear learning systems. In David S. Touretzky, editor, *Neural Information Processing Systems*, volume 4. Morgan Kaufmann Publishers, San Mateo, CA, 1992. To appear.

[NY83] A.S. Nemirovsky and D. D. Yudin. *Problem Complexity and Method Efficiency in Optimization.* Wiley, New York, 1983.

[Omo91] S.M. Omohundro. Bumptrees for efficient function, constraint and classification learning. In R.P. Lippmann and et al., editors, *NIPS-90*, San Mateo CA, 1991. IEEE, Morgan Kaufmann.

[PG90] T. Poggio and F. Girosi. Regularization algorithms for learning that are equivalent to multilayer networks. *Science*, 247:978 – 982, February 1990.

[Pog75] T. Poggio. On optimal nonlinear associative recall. *Biol. Cybernetics*, Vol. 19:201–209, 1975.

[Ros62] F. Rosenblatt. *Principles of neurodynamics.* Spartan Books, New York, 1962.

[SLD92] P. Simard, Y. LeCun, and J. Denker. Tangent prop—a formalism for specifying selected invariances in an adaptive network. In David S. Touretzky, editor, *Neural Information Processing Systems*, volume 4. Morgan Kaufmann Publishers, San Mateo, CA, 1992. To appear.

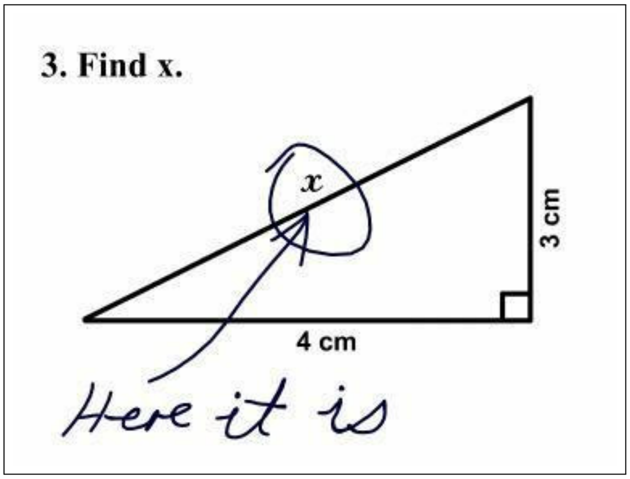
[TLS89] N. Tishby, E. Levin, and S. A. Solla. Consistent inference of probabilities in layered networks: Predictions and generalization. In *Proceedings of the International Joint Conference on Neural Networks*, Washington DC, 1989.

[Vap82] Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data.* Springer Verlag, New York, 1982.

[VC74] V.N. Vapnik and A.Ya. Chervonenkis. *The theory of pattern recognition.* Nauka, Moscow, 1974.

Chapter 5

Distance Based Methods

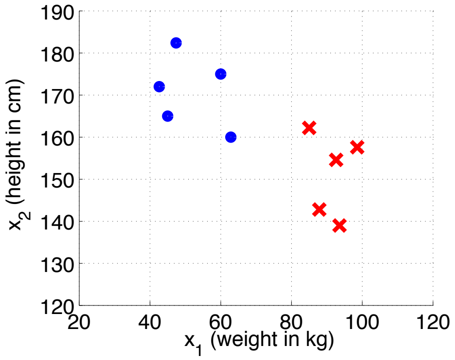


5.1 The Nearest Neighbour Classifier

Notice that I did not call this chapter “Distance Based *Models*”. In this chapter we’re going to quickly deal with a very simple method for performing good classification, that actually *avoids* building any model at all. In fact, it has no learning algorithm either. The advantage of this, as we will see, is that it can be a *very accurate* way of classifying things, though it can be very computationally intensive.

Let’s take a simple dataset :

x_1	x_2	y
42.8	171.9	0
47.6	182.3	0
45.0	165.0	0
60.0	175.0	0
63.0	160.0	0
85.0	162.1	1
98.7	157.6	1
93.6	138.8	1
87.9	142.7	1
92.8	154.5	1



And we get a testing example to classify, of $\mathbf{x} = \{70, 160\}$. The algorithm we will use is called the ‘nearest neighbour’ approach, and is simply this:

Nearest Neighbour Classification

```
for each testing example $\mathbf{x}$, do
    Find the most similar training example
    Predict the $\mathbf{x}$ to have the same label as that example.
end for
```

If you locate $\{70, 160\}$ in the right hand scatterplot, you’ll notice its closest neighbour is a blue dot, so our predicted label will be $\hat{y} = 0$. It seems too simple, doesn’t it? The classification method is just: find the most similar looking thing you’ve seen before, and assume the new thing has the same label as that! The part I’ve left unspecified is how to quantify the “similarity” of two examples. *One* way (we’ll see an alternative later) to do this is by measuring the *Euclidean* distance between the points. If you are unfamiliar with this, you should revise an old textbook on trigonometry, perhaps Pythagoras’ Theorem, which is illustrated in figure 5.1.

EUCLIDEAN DISTANCE

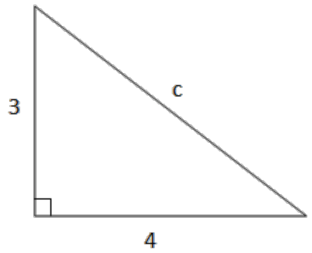


Figure 5.1: Pythagoras' Theorem shows us that here, $c^2 = a^2 + b^2$, so therefore the length of the hypotenuse here is $c = \sqrt{4^2 + 3^2}$. 'Euclidean' distance is the generalisation of this to d dimensions where the diagram shows the case of $d = 2$.

For two points with d features, $\mathbf{x} = \{x_1, ..., x_d\}$, and $\mathbf{x}' = \{x'_1, ..., x'_d\}$, the Euclidean distance between them is,

$$distance(\mathbf{x}, \mathbf{x}') = \sqrt{\sum_{j=1}^{d} (x_j - x'_j)^2}. \tag{5.1}$$

If we had a testing datapoint $\mathbf{x}' = \{70, 160\}$, we can measure the distance from this to each of our training data points:

x_1	x_2	y	$distance(\mathbf{x}, \mathbf{x}')$
42.8	171.9	0	29.69
47.6	182.3	0	31.61
45.0	165.0	0	25.49
60.0	175.0	0	18.03
63.0	160.0	0	7.00
85.0	162.1	1	10.21
98.7	157.6	1	28.8
93.6	138.8	1	31.7
87.9	142.7	1	24.89
92.8	154.5	1	23.45

The closest datapoint is the fifth one, $\mathbf{x} = \{63, 160\}$, which happens to be class $y = 0$, therefore we will predict $\hat{y} = 0$ as well. A simple extension of nearest neighbour classification is the k-Nearest Neighbour classifier, which is:

k-Nearest Neighbour Classification

```
for each testing example $\mathbf{x}$, do
    Find the $k$ most similar training examples
    Predict the $\mathbf{x}$ to be whatever the most common label is, among those $k$.
end for
```

SELF-TEST
What will the prediction, $\hat{y}$, be with the k-nn when $k = 3$? What about when $k = 5$? ... Or, what about the case of $k = 4$? Does this last one tell you anything about sensible choices for k?

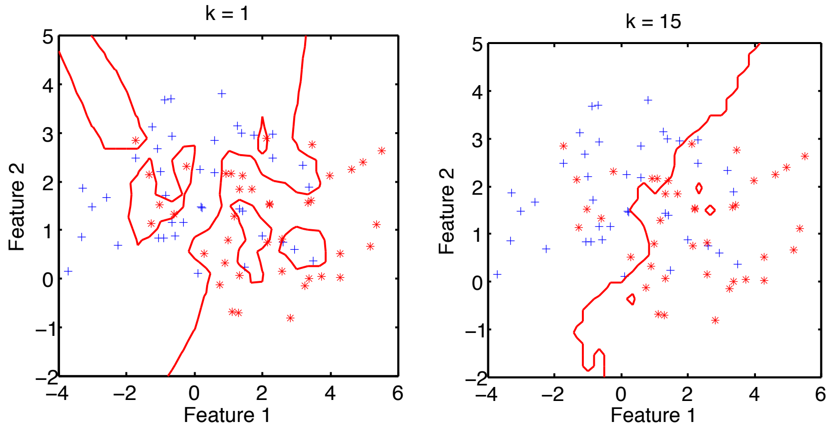


Figure 5.2: k-nn boundaries drawn for $k = 1$ (left) and for $k = 15$ (right). A very small value of k tends to over fit the training data, making very complex decision boundaries. A very large value tends to draw smoother boundaries, but tend to *underfit* the data.

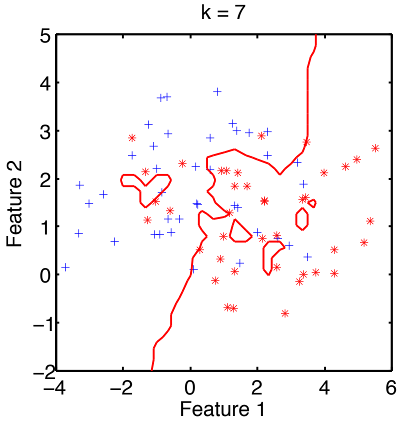


Figure 5.3: The same problem, but boundary drawn for $k = 7$. This medium value of $k = 7$ (for this problem) turned out to give a good trade-off and fitted the data well.

5.2 Handling Categorical Data

The k-NN is a very powerful, albeit computationally intensive, method of classification. The distance measure we discussed, Euclidean, is only one of many you could possibly use within the same framework. For example, what if our data was not real valued quantities such as height in centimetres, but *categorical* data—we might have a variable that takes on values from a set $\{England, Ireland, Scotland\}$ as the place where someone lives. In this case, a more appropriate measure is the *Hamming* distance:

$$hamming(\mathbf{x}, \mathbf{x}') = \sum_{j=1}^{d} \delta(x_j \neq x'_j). \tag{5.2}$$

where δ is the *indicator* function, returning 1 if its argument is true, or false otherwise. This effectively counts the number of non-equal entries in the two examples. In the more complex cases, our data may potentially be a mix of continuous and categorical data, in which case we may have to hybridise the two, having *mixed* distance of some Euclidean and some Hamming.

5.3 Comparison of k-NN to other Models

k-NN in it's simplest form can be very computationally intensive. Just think about what had to happen to calculate what the nearest neighbour to a point was. In the distance measurement, there are $2d$ additions, d multiplies, and a square root, and that's just for distance to a single training point. This would be repeated for *every point*, then the closest k would have to be found, before a prediction can be made. If we compare this to the perceptron, or logistic regression (just d multiplies and d additions to get a prediction) it is far more computationally intensive.

Modifications, like the *condensed* k-nn, and good data structures like *oct-trees*, can improve performance, but are not covered in this course. Something to remember is that k-NN can be severely affected by the scaling of the features, and the presence of irrelevant features.

5.4 What you should know by now

You should be able to say what effect the value of k has on the complexity of the decision boundary. You should also be able to state at least 1 advantage and at least 1 disadvantage of the k-nn classification method, in comparison to other models we have met so far.

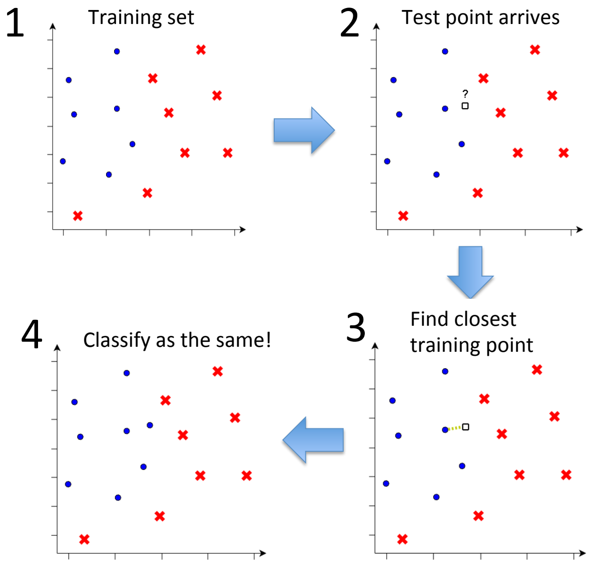
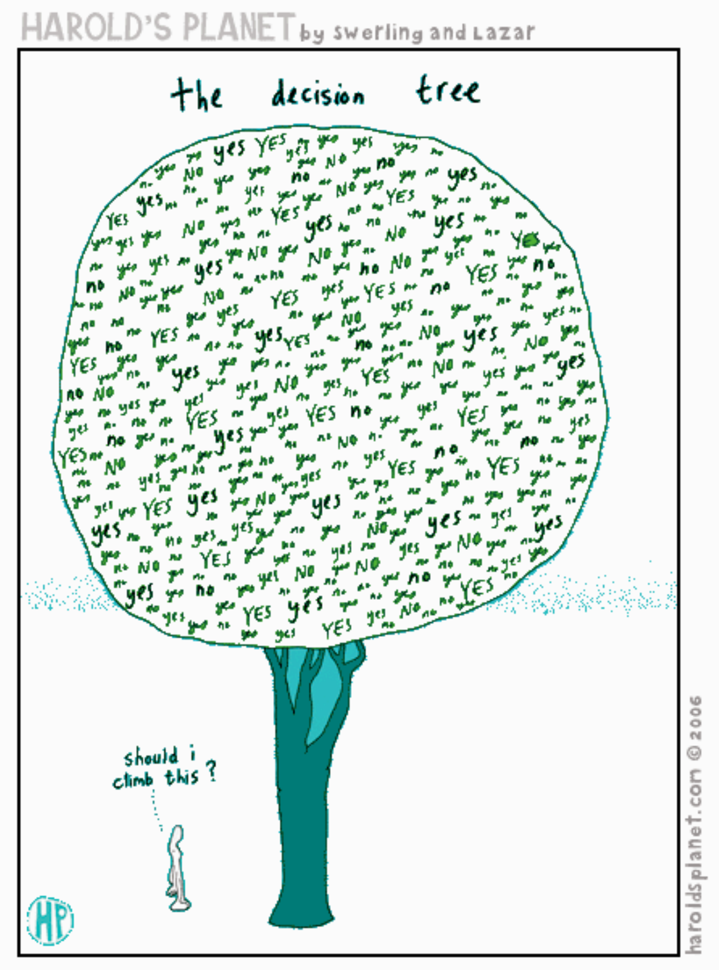


Figure 5.4: Stages of nearest neighbour classification. It's that simple!

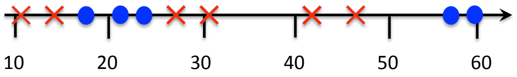
Chapter 6

Tree Models



6.1 From Decision *Stumps*... to Decision *Trees*

Recall the definition of a decision stump back in subsection 2.1.2, and imagine we have the following 1-dimensional problem.



The red crosses are label $y = 1$, and blue dots $y = 0$. With this *non-linearly separable* data, we decide to fit a decision stump. Our model has the form:

$$\texttt{if } x > t \texttt{ then } \hat{y} = 1 \texttt{ else } \hat{y} = 0 \tag{6.1}$$

SELF-TEST
Where is an optimal decision stump threshold for the data and model above? Draw it into the diagram above. Hint: you should find that it has a classification error rate of about 0.364.

SPLITTING THE DATA

If you locate the optimal threshold for this model, you will notice that it commits some errors, and as we know this is because it is a non-linearly separable problem. A way of visualising the errors is to remember the stump *splits* the data in two, as shown below. On the left 'branch', we predict $\hat{y} = 0$, and on the right, $\hat{y} = 1$.

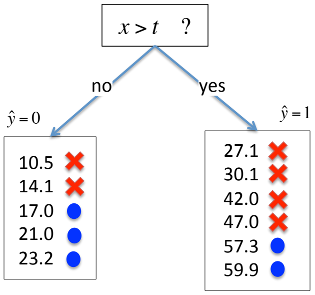


Figure 6.1: The decision stump *splits* the data.

However, if we choose a different threshold of $t = 50$, and use a stump that makes the decision the opposite way round, i.e. `if` $x > t$ `then` $\hat{y} = 0$ `else` $\hat{y} = 1$, then this stump will have a better minimum error, that is 0.273. This, combined with fig 6.1, shows a way for us to make an improved stump model.

Our improved decision stump model, for a given threshold t, is:

```
Set $y_{right}$ to the most common label in the ($> t$) subsample.
Set $y_{left}$ to the most common label in the ($< t$) subsample.

if $x > t$ then
    predict $\hat{y} = y_{right}$
else
    predict $\hat{y} = y_{left}$
endif
```

The learning algorithm would be the same, simple line-search to find the optimum threshold that minimises the number of mistakes. So, our improved stump model works by thresholding on the training data, and predicting a test datapoint label as the most common label observed in the training data subsample.

Even with this improved stump, though, we are still making some errors. There is in principle no reason why we can't fit *another* decision stump (or indeed any other model) to these data sub-samples. On the left branch data sub-sample, we could easily pick an optimal threshold for a stump, and the same for the right. Notice that the sub-samples are both *linearly separable*, therefore we can perfectly classify them with the decision stump. The result[1] of doing this is the following model, which is an example of a *decision tree*:

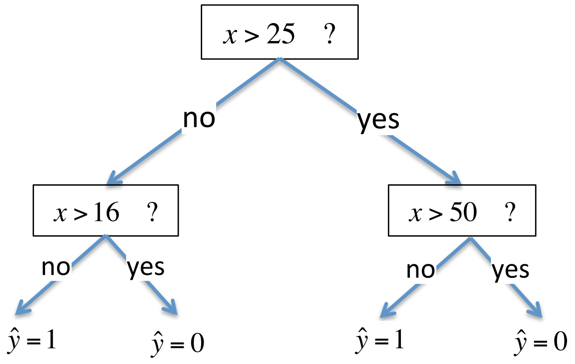


Figure 6.2: A decision tree for the toy 1d problem.

[1] By this point I hope you've figured out that the optimal threshold for the toy problem was about $x = 25$. Several other thresholds (in fact an infinity of them between 23.2 and 27.1) would have got the same error rate of 4/11, but we chose one arbitrarily.

Just as a decision stump is a rule, a decision tree is a *nested set of rules.* The one above can be written as:

```
if $x > 25$ then
   if $x > 50$ then $\hat{y} = 0$; else $\hat{y} = 1$; endif
else
   if $x > 16$ then $\hat{y} = 0$; else $\hat{y} = 1$; endif
endif
```

As you might expect at this point, a learning algorithm to construct this tree automatically from the data will be a *recursive* algorithm. If you are out of practice with the concept of recursion, I suggest you revisit your old computer science books, as it will not be covered on this module.

TREE DEPTH

The algorithm, below, is called with buildTree(*subsample*, *maxdepth*), where *subsample* is the dataset (feature and labels), and *maxdepth* is the maximum allowable depth that you want the tree to grow to. The base case terminates the algorithm when this depth is reached. The algorithm therefore returns a tree with maximum depth less than or equal to this.

Decision Tree Learning Algorithm (sometimes called "ID3")

```
function buildTree( subsample, depth )

   //BASE CASE:
   if ($depth == 0$) OR (all examples have same label) then
       return most common label in the subsample
   end if

   //RECURSIVE CASE:
   for each feature do
       Try splitting the data (i.e. build a decision stump)
       Calculate the cost for this stump
   end for
   Pick feature with minimum cost

   Find left/right subsamples
   Add left branch ← buildTree( $leftSubSample$, $depth - 1$ )
   Add right branch ← buildTree( $rightSubSample$, $depth - 1$ )

   return tree

end function
```

The reason for restricting the depth is the following. As we split the data more times (i.e. build a deeper tree) the data subsamples become smaller and smaller. Eventually, we will be trying to split the data when just 2 or 3 examples will be in the subsample. If we split at this point we will be making new nested rules for these few examples, which may be outliers, just noise in the data, and not really important. This causes *overfitting* to those few examples. We can therefore also implement similar depth restrictions by altering the base case to:

OVERFITTING

```
//BASE CASE:
if (size of subsample $< k$) OR (all examples have same label) then
        return most common label in the subsample
end if
```

where we have set a minimum of k examples in the subsample to generate a new split. Different software packages use different implementations—some restrict depth manually, and some restrict by a minimum subsample size.

6.2 Big Trees Overfit, Little Trees Underfit

We now know how to learn a decision tree. The only user-supplied parameter is the maximum allowable *depth* of the tree. The reason for this is that larger trees tend to be *overfitted* to the training data. A typical scenario is shown in figure 6.2, where the testing error begins to rise, as the depth of the tree grows too large.

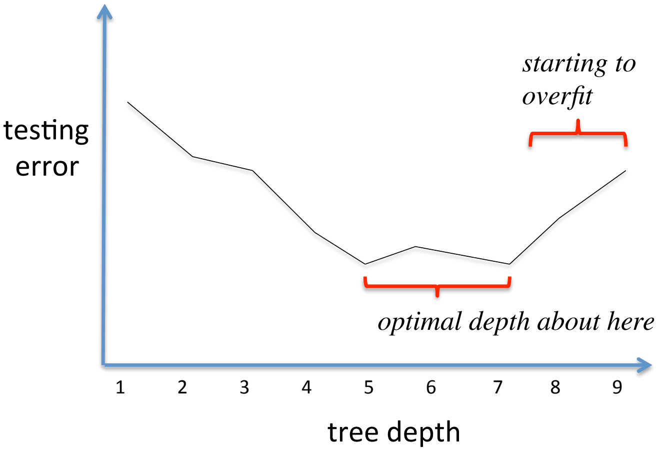


Figure 6.3: *A typical scenario when choosing the right depth for a decision tree. Too deep makes it overfit, and not deep enough makes it underfit. The optimal depth is somewhere in between.*

6.3 Dealing with Categorical Data

So far, we have seen how problems can be solved by models based on geometric principles, i.e. we were able to scatterplot the data and see what it looked like, then apply models based on the geometry of the line $y = mx + c$. What happens when geometry cannot be used? For example, how would you scatterplot the data shown below?

Outlook	**Temperature**	**Humidity**	**Wind**	**Play Tennis?**
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No

The task here is to give advice on whether it is a good idea to play tennis, given the current weather conditions. This data does not correspond to our intuition of plots on a 2d axis, separating it with a plane. We know that the number '1' comes before '2', but does the value 'Rain' come before or after 'Sunny'? We have so far been dealing with *continuous* data, for which SVMs and other geometric classifiers are appropriate, but they cannot deal with the *categorical data* above. Quite nicely, it turns out that decision trees can easily deal with this sort of problem.

CATEGORICAL DATA

A tree that perfectly classifies all the training data is this one:

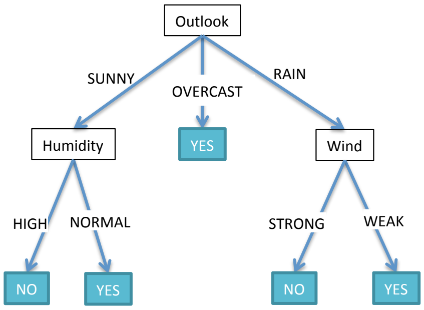


Figure 6.4: A Decision Tree: notice that the different paths down the tree encode a set of if-then logical rules.

Once again, an answer for any given example is found by following a path down the tree, answering questions as you go. Each path down the tree encodes an if-then rule. The full ruleset for this tree is:

```
if ( Outlook==sunny AND Humidity==high )      then NO
if ( Outlook==sunny AND Humidity==normal )    then YES
if ( Outlook==overcast )                      then YES
if ( Outlook==rain AND Wind==strong )         then NO
if ( Outlook==rain AND Wind==weak )           then YES
```

Notice that this tree (or equivalently, the ruleset) can correctly classify every example in the training data. *This tree is our model, or, seen in another light, the set of rules is our model.* These viewpoints are equivalent. The tree was constructed automatically (learnt) from the data, just as the parameters of our linear models in previous chapters were learnt from algorithms working on the data. Notice also that the model deals with scenarios that were *not present in the training data* – for example the model will also give a correct response if we had a completely never-before-seen situation like this:

Outlook	**Temperature**	**Humidity**	**Wind**	**Play Tennis?**
Overcast	Mild	High	Weak	Yes

The tree can therefore deal with data points that were *not in the training data*. If you remember from earlier chapters, this means the tree has good *generalisation* accuracy, or equivalently, it has not *overfitted*.

Let's consider another possible tree, shown in figure 6.5. If you check, this tree *also correctly classifies* every example in the training data. However, the testing datapoint "overcast/mild/high/weak", receives a classification of 'NO'. Whereas, in fact, as we just saw, the correct answer is YES. This decision tree made an incorrect prediction because it was *overfitted* to the training data.

OVERFITTING

As you will remember, we can never tell for sure whether a model is overfitted until it is evaluated on some testing data. However, with decision trees, a strong indicator that they are overfitted is that they are *very deep*, that is the rules are very fine-tuned to conditions and sub-conditions that may just be irrelevant facts. The smaller tree just made a simple check that the outlook was overcast, whereas the deeper tree expanded far beyond this simple rule.

SELF-TEST

What prediction will the tree in figure 6.5 give for this testing datapoint?

Outlook	**Temperature**	**Humidity**	**Wind**
Sunny	Mild	High	Strong

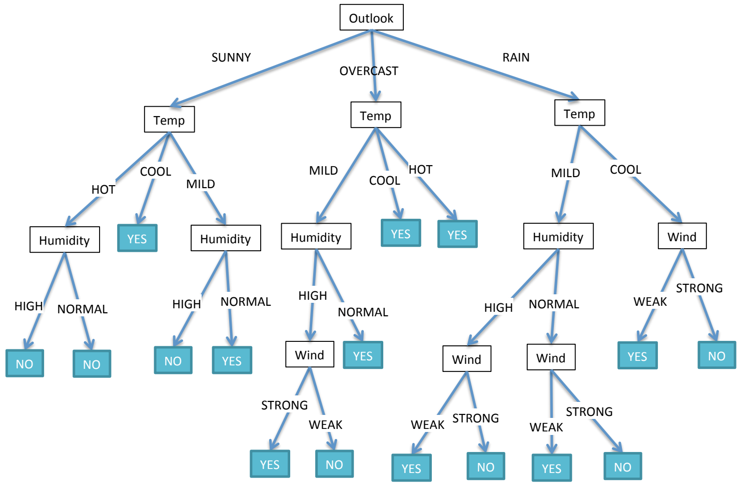


Figure 6.5: An overfitted decision tree.

6.4 Measuring Information Gain

Now, remember to build this tree, we need to recursively split the data, and measure the cost. In this section we'll meet a new cost measure, called 'information gain', which is sometimes preferable. If we had a dataset of examples:

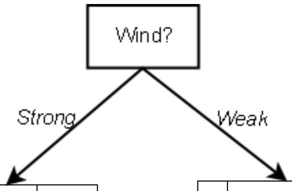
x_1	x_2	y
1	1	1
1	0	0
1	1	1
1	0	1
0	0	0
0	0	0
0	0	0
0	0	1

SPLIT CRITERION

INFORMATION GAIN
MUTUAL INFORMATION

If we split the data based on x_1, we get a classification error rate of 0.25. If we split the data based on x_2, we get the same error rate, 0.25, so according to this *splitting criterion*, there is no reason to prefer one over the other. However, the second feature, x_2, produces a 'pure' node, i.e. it has a classification error on its dataset of 0, and has completely solved the sub-problem it was assigned by the recursive call, so is arguably preferable in practical situations. There is an alternative split criterion we can use, called the *information gain*, also known as the *mutual information*. The information gain for x_1 is $I(X_1; Y) = 0.1887$, whereas $I(X_2; Y) = 0.3113$, telling us that we should prefer x_2, which is the split that generated the pure node.

We will now see how to calculate this for the tennis data we saw earlier. Let's see what it looks like to split the data into subsamples, based on the "wind" feature.



	Outlook	Temp	Humid	Wind	Play?
2	Sunny	Hot	High	Strong	No
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Strong	Yes
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
14	Rain	Mild	High	Strong	No

3 examples say yes, 3 say no.

	Outlook	Temp	Humid	Wind	Play?
1	Sunny	Hot	High	Weak	No
3	Overcast	Hot	High	Weak	Yes
4	Rain	Mild	High	Weak	Yes
5	Rain	Cool	Normal	Weak	Yes
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Weak	Yes
13	Overcast	Hot	Normal	Weak	Yes

6 examples say yes, 2 say no.

From the original 14 examples, 9 have the answer YES, and 5 have the answer NO. We regard this distribution of yes/no as the 'default' information, that is, if we are given no further data on what the weather is like, then the best anyone could do is predict "yes", as this is the majority situation among the data.

However, by splitting our training data up based on what the "wind" feature said, it has changed the distribution of "yes" and "no" answers. Now, if the wind is weak, we should be more confident (see probabilities below) of the decision to play tennis. If the wind is strong, we see there is a 50 : 50 split, so we have become more confident that 'no' is the correct answer.

Before the split : 9 'yes', 5 'no', ... $p('yes') = \frac{9}{14} \approx 0.64$

On the left branch : 3 'yes', 3 'no', ... $p('yes') = \frac{3}{6} = 0.5$

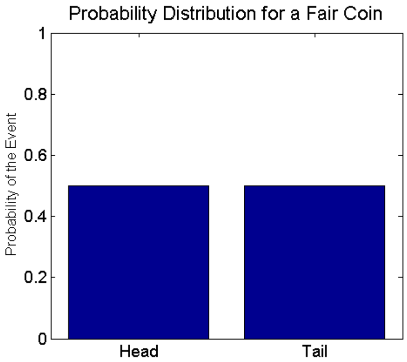
On the right branch : 6 'yes', 2 'no', ... $p('yes') = \frac{6}{8} = 0.75$

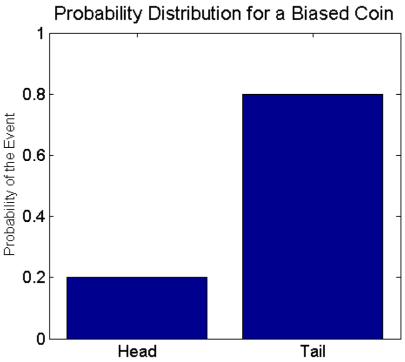
Remember... $p('no') = 1 - p('yes')$

"Splitting" the data on the wind feature has provided us with some *information.* We can make a slightly more well informed decision if we use this information. Which one of the features would give us the **most information?**. To answer this, we must have a way of *measuring the information* we would gain as a score for each feature, then we can pick the feature with the highest score.

Entropy - a measure of information

Think of a coin flip. How “random” is a coin flip? One might say a fair coin, equally balanced for heads/tails, would be “completely” random. On the other hand, a *biased coin*, that is one that would come up more often as tails than as heads, is “less random”.





One might imagine, that if we learnt that the distribution was biased, we have more information, that is *with less randomness, we got more information.* And conversely, when we know nothing at all about the coin, we must presume it is completely random, so *with more randomness, we have less information.* The *entropy* is a mathematical expression for quantifying this randomness. The entropy of a variable X is given by.

$$H(X) = -\sum_{x \in X} p(x) \log p(x) \tag{6.2}$$

where the sum is over all possible states that the variable X could possibly take. The log is base 2, giving us units of measurement ’bits’. In the coin example, we have:

$$\begin{aligned} H(X) &= -\Big(p(head)\log p(head) + p(tail)\log p(tail)\Big) \\ &= -\Big(0.5\log 0.5 + 0.5\log 0.5\Big) \\ &= -\Big((-0.5) + (-0.5)\Big) = 1 \end{aligned}$$

So the entropy, or the “amount of randomness”, is 1 bit. Try to work it out below, for the biased coin.

SELF-TEST
Calculate the entropy of a two-valued feature, i.e. $x \in \{0, 1\}$, with $p(x = 1) = 0.6$.

Measuring the Information Gain of a Feature

Now we know how to measure the amount of randomness (or conversely the amount of information) in a probability distribution, we can use this to measure the gain from a particular feature. What are we measuring the randomness of? The variable telling us whether we should be playing tennis or not (YES/NO). We will call this variable T (for t̲ennis), which can take on two values $T = yes$ and $T = no$. Remembering the figures from earlier, using the wind feature:

Before the split : 9 'yes', 5 'no', giving us probabilities: $p(yes) = \frac{9}{14}$, and $p(no) = \frac{5}{14}$. And so the entropy is,

$$H(T) = -\Big(\frac{9}{14}\log\frac{9}{14} + \frac{5}{14}\log\frac{5}{14}\Big) \approx 0.94029 \tag{6.3}$$

After the split on the left branch (when the wind is strong), there were 3 examples with *yes*, and 3 examples of *no*. The entropy is,

$$H(T|W = strong) = -\Big(\frac{3}{6}\log\frac{3}{6} + \frac{3}{6}\log\frac{3}{6}\Big) = 1 \tag{6.4}$$

After the split on the right branch (when the wind is weak), there were 6 examples with *yes*, and 2 examples of *no*. Now it's your turn to work this one out in the space below.

SELF-TEST

Calculate $H(T|W = weak)$.

Once we have calculated the entropy down each branch, $H(T|W = strong)$ and $H(T|W = weak)$, we take a weighted average of the two, dependent on how many examples totally made it down into each split. In this case, there were $3 + 3 = 6$ examples totally in the *strong* split (left), and $6 + 2 = 8$ examples in the *weak* split (right). The average of the two is:

$$\begin{aligned} H(T|W) &= \frac{6}{6+8}H(T|W = strong) + \frac{8}{6+8}H(T|W = weak) \\ &= 0.42857 + 0.57142 \times H(T|W = weak) \\ &= 0.42857 + 0.57142 \times \underline{\hspace{3cm}} \\ &\approx 0.89215 \end{aligned} \tag{6.5}$$

Make sure you work out the value of $H(T|W = weak)$, and plug it into the above to verify it. Note I have rounded the figures above, so as long as you are accurate to within 3 decimal places, that's fine. Now, we know the entropy **before the split**, and the average entropy **after the split**. To calculate the information gain we simple take the difference of the two:

$$\begin{aligned} I(T;W) &= H(T) - H(T|W) \\ &= 0.94029 - 0.89215 = 0.04814 \end{aligned}$$

This tells us the information gain (in bits) of using "wind" to split on is 0.04814. That's not very much! The maximum possible gain is when $H(T|W) = 0$, so here the maximum possible gain would be 0.94029. This is the case when the split has removed all uncertainty. This will be different for every problem - it depends what $H(T)$ was in the first place. If $H(T)$ was maximal before the split, then it means there was complete uncertainty – a uniform distribution over all the possible values of T. It turns out that the maximal possible value of $H(T)$ is $\log(|T|)$, the logarithm of the number of possible values.

UPPER BOUND OF ENTROPY VALUE

In summary, we calculate the gain by the following simple steps:

1. **Work out the entropy *before* the data split.**
2. **Work out the weighted average entropy after the split.**
3. **Take the difference of these.**

For the original 14 examples, the gain *Outloook* and *Temperature* is:

Gain(outlook) = $0.94029 - 0.6935 = 0.2467$
Gain(temperature) = $0.94029 - 0.911 = 0.029$

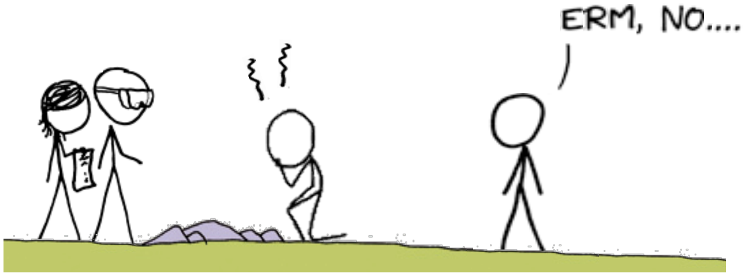
So far, it looks like *Outlook* has the maximum gain, at ≈ 0.2467. I will leave the *humidity* feature for you to work out yourself. Remember again that above I have rounded the figures in the calculations, so 3 decimal places is sufficient.

6.5 What you should know by now

You should be able to calculate the entropy for a binary feature, and also be able to calculate the information gain for a feature, given a class label. You should also know why and how the Id3 algorithm works, including the termination condition (base cases). You should know how overfitting manifests in trees, and how to control it.

Chapter 7

Experimental Methods II: ROC Analysis



7.1 Evaluating Model Performance

So far, we have been evaluating our models by simply a count of how many errors they make, and maybe averaging this via cross-validation. In fact, this is quite naive, and there are many more sophisticated ways to evaluate the performance of a model. In this chapter you will engage in your own reading of a published article on *Receiver Operator Characteristics*, also known as ROC analysis. Take note, you may well encounter machine learning terminology that is slightly different to that which we have used so far—you should get used to this, as different fields (e.g. statistics, data mining, bioinformatics) use different terms, but in fact all mean the same thing. If you get lost, chat to your colleagues or do some Googling to see what the terms mean.

Sections 1-3 are required reading and *will be examinable.* Sections 4 onward are optional, but you may well like to read up on them for your own curiosity, or to make use of in your mini-projects.

Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Pattern Recognition Letters 27 (2006) 861–874

Pattern Recognition Letters

www.elsevier.com/locate/patrec

An introduction to ROC analysis

Tom Fawcett

Institute for the Study of Learning and Expertise, 2164 Staunton Court, Palo Alto, CA 94306, USA

Available online 19 December 2005

Abstract

Receiver operating characteristics (ROC) graphs are useful for organizing classifiers and visualizing their performance. ROC graphs are commonly used in medical decision making, and in recent years have been used increasingly in machine learning and data mining research. Although ROC graphs are apparently simple, there are some common misconceptions and pitfalls when using them in practice. The purpose of this article is to serve as an introduction to ROC graphs and as a guide for using them in research.

Keywords: ROC analysis; Classifier evaluation; Evaluation metrics

1. Introduction

A receiver operating characteristics (ROC) graph is a technique for visualizing, organizing and selecting classifiers based on their performance. ROC graphs have long been used in signal detection theory to depict the tradeoff between hit rates and false alarm rates of classifiers (Egan, 1975; Swets et al., 2000). ROC analysis has been extended for use in visualizing and analyzing the behavior of diagnostic systems (Swets, 1988). The medical decision making community has an extensive literature on the use of ROC graphs for diagnostic testing (Zou, 2002). Swets et al. (2000) brought ROC curves to the attention of the wider public with their *Scientific American* article.

One of the earliest adopters of ROC graphs in machine learning was Spackman (1989), who demonstrated the value of ROC curves in evaluating and comparing algorithms. Recent years have seen an increase in the use of ROC graphs in the machine learning community, due in part to the realization that simple classification accuracy is often a poor metric for measuring performance (Provost and Fawcett, 1997; Provost et al., 1998). In addition to being a generally useful performance graphing method, they have properties that make them especially useful for domains with skewed class distribution and unequal classification error costs. These characteristics have become increasingly important as research continues into the areas of cost-sensitive learning and learning in the presence of unbalanced classes.

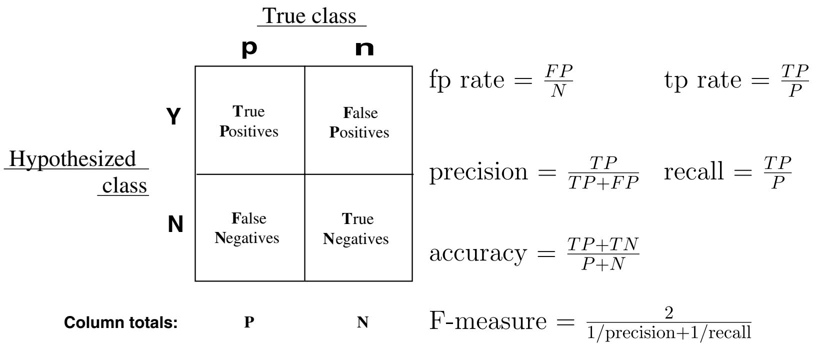
ROC graphs are conceptually simple, but there are some non-obvious complexities that arise when they are used in research. There are also common misconceptions and pitfalls when using them in practice. This article attempts to serve as a basic introduction to ROC graphs and as a guide for using them in research. The goal of this article is to advance general knowledge about ROC graphs so as to promote better evaluation practices in the field.

2. Classifier performance

We begin by considering classification problems using only two classes. Formally, each instance I is mapped to one element of the set $\{\boldsymbol{p}, \boldsymbol{n}\}$ of positive and negative class labels. A *classification model* (or *classifier*) is a mapping from instances to predicted classes. Some classification models produce a continuous output (e.g., an estimate of an instance's class membership probability) to which different thresholds may be applied to predict class membership. Other models produce a discrete class label indicating only the predicted class of the instance. To distinguish between

E-mail addresses: tfawcett@acm.org, tom.fawcett@gmail.com

0167-8655/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.patrec.2005.10.010



		True class p	True class n
Hypothesized class	Y	True Positives	False Positives
	N	False Negatives	True Negatives
Column totals:		P	N

$$\text{fp rate} = \frac{FP}{N} \qquad \text{tp rate} = \frac{TP}{P}$$

$$\text{precision} = \frac{TP}{TP+FP} \qquad \text{recall} = \frac{TP}{P}$$

$$\text{accuracy} = \frac{TP+TN}{P+N}$$

$$\text{F-measure} = \frac{2}{1/\text{precision}+1/\text{recall}}$$

Fig. 1. Confusion matrix and common performance metrics calculated from it.

the actual class and the predicted class we use the labels $\{\boldsymbol{Y},\boldsymbol{N}\}$ for the class predictions produced by a model.

Given a classifier and an instance, there are four possible outcomes. If the instance is positive and it is classified as positive, it is counted as a *true positive*; if it is classified as negative, it is counted as a *false negative*. If the instance is negative and it is classified as negative, it is counted as a *true negative*; if it is classified as positive, it is counted as a *false positive*. Given a classifier and a set of instances (the test set), a two-by-two *confusion matrix* (also called a contingency table) can be constructed representing the dispositions of the set of instances. This matrix forms the basis for many common metrics.

Fig. 1 shows a confusion matrix and equations of several common metrics that can be calculated from it. The numbers along the major diagonal represent the correct decisions made, and the numbers of this diagonal represent the errors—the confusion—between the various classes. The **true positive rate**[1] (also called *hit rate* and *recall*) of a classifier is estimated as

$$\textit{tp rate} \approx \frac{\text{Positives correctly classified}}{\text{Total positives}}$$

The **false positive rate** (also called *false alarm rate*) of the classifier is

$$\textit{fp rate} \approx \frac{\text{Negatives incorrectly classified}}{\text{Total negatives}}$$

Additional terms associated with ROC curves are

$$\begin{aligned}
\text{sensitivity} &= \text{recall}\\
\text{specificity} &= \frac{\text{True negatives}}{\text{False positives} + \text{True negatives}}\\
&= 1 - \textit{fp rate}\\
\text{positive predictive value} &= \text{precision}
\end{aligned}$$

3. ROC space

ROC graphs are two-dimensional graphs in which *tp rate* is plotted on the Y axis and *fp rate* is plotted on the X axis. An ROC graph depicts relative tradeoffs between benefits (true positives) and costs (false positives). Fig. 2 shows an ROC graph with five classifiers labeled A through E.

A *discrete* classifier is one that outputs only a class label. Each discrete classifier produces an (*fp rate*, *tp rate*) pair corresponding to a single point in ROC space. The classifiers in Fig. 2 are all discrete classifiers.

Several points in ROC space are important to note. The lower left point $(0,0)$ represents the strategy of never issuing a positive classification; such a classifier commits no false positive errors but also gains no true positives. The opposite strategy, of unconditionally issuing positive classifications, is represented by the upper right point $(1,1)$.

The point $(0,1)$ represents perfect classification. D's performance is perfect as shown.

Informally, one point in ROC space is better than another if it is to the northwest (*tp rate* is higher, *fp rate* is lower, or both) of the first. Classifiers appearing on the left-hand side of an ROC graph, near the X axis, may be

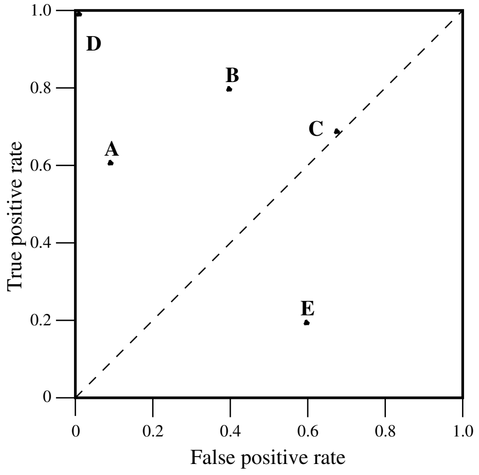


Fig. 2. A basic ROC graph showing five discrete classifiers.

[1] For clarity, counts such as TP and FP will be denoted with upper-case letters and rates such as *tp rate* will be denoted with lower-case.

thought of as “conservative”: they make positive classifications only with strong evidence so they make few false positive errors, but they often have low true positive rates as well. Classifiers on the upper right-hand side of an ROC graph may be thought of as “liberal”: they make positive classifications with weak evidence so they classify nearly all positives correctly, but they often have high false positive rates. In Fig. 2, A is more conservative than B. Many real world domains are dominated by large numbers of negative instances, so performance in the far left-hand side of the ROC graph becomes more interesting.

3.1. Random performance

The diagonal line $y = x$ represents the strategy of randomly guessing a class. For example, if a classifier randomly guesses the positive class half the time, it can be expected to get half the positives and half the negatives correct; this yields the point (0.5, 0.5) in ROC space. If it guesses the positive class 90% of the time, it can be expected to get 90% of the positives correct but its false positive rate will increase to 90% as well, yielding (0.9, 0.9) in ROC space. Thus a random classifier will produce a ROC point that “slides” back and forth on the diagonal based on the frequency with which it guesses the positive class. In order to get away from this diagonal into the upper triangular region, the classifier must exploit some information in the data. In Fig. 2, C’s performance is virtually random. At (0.7, 0.7), C may be said to be guessing the positive class 70% of the time.

Any classifier that appears in the lower right triangle performs worse than random guessing. This triangle is therefore usually empty in ROC graphs. If we negate a classifier—that is, reverse its classification decisions on every instance—its true positive classifications become false negative mistakes, and its false positives become true negatives. Therefore, any classifier that produces a point in the lower right triangle can be negated to produce a point in the upper left triangle. In Fig. 2, E performs much worse than random, and is in fact the negation of B. Any classifier on the diagonal may be said to have no information about the class. A classifier below the diagonal may be said to have useful information, but it is applying the information incorrectly (Flach and Wu, 2003).

Given an ROC graph in which a classifier’s performance appears to be slightly better than random, it is natural to ask: “is this classifier’s performance truly significant or is it only better than random by chance?” There is no conclusive test for this, but Forman (2002) has shown a methodology that addresses this question with ROC curves.

4. Curves in ROC space

Many classifiers, such as decision trees or rule sets, are designed to produce only a class decision, i.e., a ***Y*** or ***N*** on each instance. When such a discrete classifier is applied to a test set, it yields a single confusion matrix, which in turn corresponds to one ROC point. Thus, a discrete classifier produces only a single point in ROC space.

Some classifiers, such as a Naive Bayes classifier or a neural network, naturally yield an instance *probability* or *score*, a numeric value that represents the degree to which an instance is a member of a class. These values can be strict probabilities, in which case they adhere to standard theorems of probability; or they can be general, uncalibrated scores, in which case the only property that holds is that a higher score indicates a higher probability. We shall call both a *probabilistic* classifier, in spite of the fact that the output may not be a proper probability.[2]

Such a *ranking* or *scoring* classifier can be used with a threshold to produce a discrete (binary) classifier: if the classifier output is above the threshold, the classifier produces a ***Y***, else a ***N***. Each threshold value produces a different point in ROC space. Conceptually, we may imagine varying a threshold from $-\infty$ to $+\infty$ and tracing a curve through ROC space. Computationally, this is a poor way of generating an ROC curve, and the next section describes a more efficient and careful method.

Fig. 3 shows an example of an ROC “curve” on a test set of 20 instances. The instances, 10 positive and 10 negative, are shown in the table beside the graph. Any ROC curve generated from a finite set of instances is actually a step function, which approaches a true curve as the number of instances approaches infinity. The step function in Fig. 3 is taken from a very small instance set so that each point’s derivation can be understood. In the table of Fig. 3, the instances are sorted by their scores, and each point in the ROC graph is labeled by the score threshold that produces it. A threshold of $+\infty$ produces the point (0, 0). As we lower the threshold to 0.9 the first positive instance is classified positive, yielding (0, 0.1). As the threshold is further reduced, the curve climbs up and to the right, ending up at (1, 1) with a threshold of 0.1. Note that lowering this threshold corresponds to moving from the “conservative” to the “liberal” areas of the graph.

Although the test set is very small, we can make some tentative observations about the classifier. It appears to perform better in the more conservative region of the graph; the ROC point at (0.1, 0.5) produces its highest accuracy (70%). This is equivalent to saying that the classifier is better at identifying likely positives than at identifying likely negatives. Note also that the classifier’s best accuracy occurs at a threshold of $\geqslant 0.54$, rather than at $\geqslant 0.5$ as we might expect with a balanced distribution. The next section discusses this phenomenon.

4.1. Relative versus absolute scores

An important point about ROC graphs is that they measure the ability of a classifier to produce good *relative*

[2] Techniques exist for converting an uncalibrated score into a proper probability but this conversion is unnecessary for ROC curves.

Inst#	Class	Score	Inst#	Class	Score
1	**p**	.9	11	**p**	.4
2	**p**	.8	12	**n**	.39
3	**n**	.7	13	**p**	.38
4	**p**	.6	14	**n**	.37
5	**p**	.55	15	**n**	.36
6	**p**	.54	16	**n**	.35
7	**n**	.53	17	**p**	.34
8	**n**	.52	18	**n**	.33
9	**p**	.51	19	**p**	.30
10	**n**	.505	20	**n**	.1

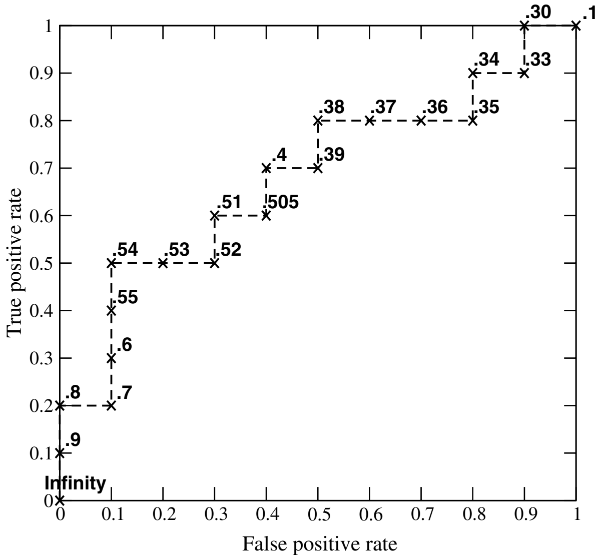


Fig. 3. The ROC "curve" created by thresholding a test set. The table shows 20 data and the score assigned to each by a scoring classifier. The graph shows the corresponding ROC curve with each point labeled by the threshold that produces it.

instance scores. A classifier need not produce accurate, calibrated probability estimates; it need only produce relative accurate scores that serve to discriminate positive and negative instances.

Consider the simple instance scores shown in Fig. 4, which came from a Naive Bayes classifier. Comparing the hypothesized class (which is ***Y*** if score > 0.5, else ***N***) against the true classes, we can see that the classifier gets instances 7 and 8 wrong, yielding 80% accuracy. However, consider the ROC curve on the left side of the figure. The curve rises vertically from (0,0) to (0,1), then horizontally to (1,1). This indicates perfect classification performance on this test set. Why is there a discrepancy?

The explanation lies in what each is measuring. The ROC curve shows the ability of the classifier to rank the positive instances relative to the negative instances, and it is indeed perfect in this ability. The accuracy metric imposes a threshold (score > 0.5) and measures the resulting classifications with respect to the scores. The accuracy measure would be appropriate if the scores were proper probabilities, but they are not. Another way of saying this is that the scores are not *properly calibrated*, as true probabilities are. In ROC space, the imposition of a 0.5 threshold results in the performance designated by the circled "accuracy point" in Fig. 4. This operating point is suboptimal. We could use the training set to estimate a prior for $p(\boldsymbol{p}) = 6/10 = 0.6$ and use this as a threshold, but it would still produce suboptimal performance (90% accuracy).

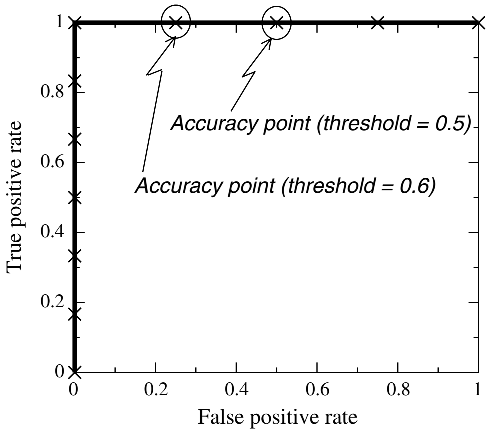
One way to eliminate this phenomenon is to calibrate the classifier scores. There are some methods for doing this (Zadrozny and Elkan, 2001). Another approach is to use an ROC method that chooses operating points based on their relative performance, and there are methods for doing this as well (Provost and Fawcett, 1998, 2001). These latter methods are discussed briefly in Section 6.

A consequence of relative scoring is that classifier scores should not be compared across model classes. One model class may be designed to produce scores in the range [0,1] while another produces scores in [−1,+1] or [1,100]. Comparing model performance at a common threshold will be meaningless.

4.2. Class skew

ROC curves have an attractive property: they are insensitive to changes in class distribution. If the proportion of positive to negative instances changes in a test set, the ROC curves will not change. To see why this is so, consider the confusion matrix in Fig. 1. Note that the class distribution—the proportion of positive to negative instances—is the relationship of the left (+) column to the right (−) column. Any performance metric that uses values from both columns will be inherently sensitive to class skews. Metrics such as accuracy, precision, lift and *F* score use values from both columns of the confusion matrix. As a class distribution changes these measures will change as well, even if the fundamental classifier performance does not. ROC graphs are based upon *tp rate* and *fp rate*, in which each dimension is a strict columnar ratio, so do not depend on class distributions.

To some researchers, large class skews and large changes in class distributions may seem contrived and unrealistic. However, class skews of 10^1 and 10^2 are very common in real world domains, and skews up to 10^6 have been observed in some domains (Clearwater and Stern, 1991; Fawcett and Provost, 1996; Kubat et al., 1998; Saitta and Neri, 1998). Substantial changes in class distributions are not unrealistic either. For example, in medical decision making epidemics may cause the incidence of a disease to increase over time. In fraud detection, proportions of fraud varied significantly from month to month and place to place (Fawcett and Provost, 1997). Changes in a manufacturing practice may cause the proportion of defective units



Inst	Class		Score
no.	True	Hyp	
1	**p**	**Y**	0.99999
2	**p**	**Y**	0.99999
3	**p**	**Y**	0.99993
4	**p**	**Y**	0.99986
5	**p**	**Y**	0.99964
6	**p**	**Y**	0.99955
7	**n**	**Y**	0.68139
8	**n**	**Y**	0.50961
9	**n**	**N**	0.48880
10	**n**	**N**	0.44951

Fig. 4. Scores and classifications of 10 instances, and the resulting ROC curve.

produced by a manufacturing line to increase or decrease. In each of these examples the prevalence of a class may change drastically without altering the fundamental characteristic of the class, i.e., the target concept.

Precision and recall are common in information retrieval for evaluating retrieval (classification) performance (Lewis, 1990, 1991). Precision-recall graphs are commonly used where static document sets can sometimes be assumed; however, they are also used in dynamic environments such as web page retrieval, where the number of pages irrelevant to a query (N) is many orders of magnitude greater than P and probably increases steadily over time as web pages are created.

To see the effect of class skew, consider the curves in Fig. 5, which show two classifiers evaluated using ROC curves and precision-recall curves. In Fig. 5a and b, the test

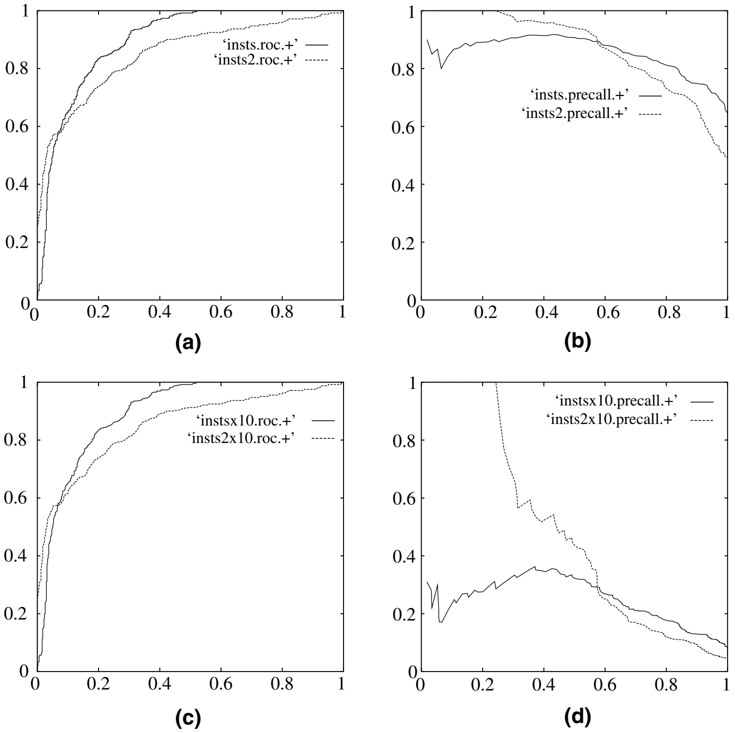


Fig. 5. ROC and precision-recall curves under class skew. (a) ROC curves, 1:1; (b) precision-recall curves, 1:1; (c) ROC curves, 1:10 and (d) precision-recall curves, 1:10.

set has a balanced 1:1 class distribution. Graph 5c and d shows the same two classifiers on the same domain, but the number of negative instances has been increased 10-fold. Note that the classifiers and the underlying concept has not changed; only the class distribution is different. Observe that the ROC graphs in Fig. 5a and c are identical, while the precision-recall graphs in Fig. 5b and d differ substantially. In some cases, the conclusion of which classifier has superior performance can change with a shifted distribution.

4.3. Creating scoring classifiers

Many classifier models are discrete: they are designed to produce only a class label from each test instance. However, we often want to generate a full ROC curve from a classifier instead of just a single point. To this end we want to generate scores from a classifier rather than just a class label. There are several ways of producing such scores.

Many discrete classifier models may easily be converted to scoring classifiers by "looking inside" them at the instance statistics they keep. For example, a decision tree determines a class label of a leaf node from the proportion of instances at the node; the class decision is simply the most prevalent class. These class proportions may serve as a score (Provost and Domingos, 2001). A rule learner keeps similar statistics on rule confidence, and the confidence of a rule matching an instance can be used as a score (Fawcett, 2001).

Even if a classifier only produces a class label, an aggregation of them may be used to generate a score. MetaCost (Domingos, 1999) employs bagging to generate an ensemble of discrete classifiers, each of which produces a vote. The set of votes could be used to generate a score.[3]

Finally, some combination of scoring and voting can be employed. For example, rules can provide basic probability estimates, which may then be used in weighted voting (Fawcett, 2001).

5. Efficient generation of ROC curves

Given a test set, we often want to generate an ROC curve efficiently from it. We can exploit the monotonicity of thresholded classifications: any instance that is classified positive with respect to a given threshold will be classified positive for all lower thresholds as well. Therefore, we can simply sort the test instances decreasing by f scores and move down the list, processing one instance at a time and updating TP and FP as we go. In this way an ROC graph can be created from a linear scan.

The algorithm is shown in Algorithm 1. TP and FP both start at zero. For each positive instance we increment TP and for every negative instance we increment FP. We maintain a stack R of ROC points, pushing a new point onto R after each instance is processed. The final output is the stack R, which will contain points on the ROC curve.

Let n be the number of points in the test set. This algorithm requires an $O(n \log n)$ sort followed by an $O(n)$ scan down the list, resulting in $O(n \log n)$ total complexity.

Statements 7–10 need some explanation. These are necessary in order to correctly handle sequences of equally scored instances. Consider the ROC curve shown in Fig. 6. Assume we have a test set in which there is a sequence of instances, four negatives and six positives, all scored equally by f. The sort in line 1 of Algorithm 1 does not impose any specific ordering on these instances since their f scores are equal. What happens when we create an ROC curve? In one extreme case, all the positives end up at the beginning of the sequence and we generate the "optimistic" upper L segment shown in Fig. 6. In the opposite

Algorithm 1. Efficient method for generating ROC points
Inputs: L, the set of test examples; $f(i)$, the probabilistic classifier's estimate that example i is positive; P and N, the number of positive and negative examples.
Outputs: R, a list of ROC points increasing by *fp rate*.
Require: $P > 0$ and $N > 0$

```
 1: $L_{\text{sorted}} \leftarrow L$ sorted decreasing by $f$ scores
 2: $FP \leftarrow TP \leftarrow 0$
 3: $R \leftarrow \langle\rangle$
 4: $f_{\text{prev}} \leftarrow -\infty$
 5: $i \leftarrow 1$
 6: while $i \leqslant |L_{\text{sorted}}|$ do
 7:     if $f(i) \neq f_{\text{prev}}$ then
 8:         push $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ onto $R$
 9:         $f_{\text{prev}} \leftarrow f(i)$
10:     end if
11:     if $L_{\text{sorted}}[i]$ is a positive example then
12:         $TP \leftarrow TP + 1$
13:     else        /* $i$ is a negative example */
14:         $FP \leftarrow FP + 1$
15:     end if
16:     $i \leftarrow i + 1$
17: end while
18: push $\left(\frac{FP}{N}, \frac{TP}{P}\right)$ onto $R$   /* This is (1,1) */
19: end
```

[3] MetaCost actually works in the opposite direction because its goal is to generate a discrete classifier. It first creates a probabilistic classifier, then applies knowledge of the error costs and class skews to relabel the instances so as to "optimize" their classifications. Finally, it learns a specific discrete classifier from this new instance set. Thus, MetaCost is not a good method for creating a scoring classifier, though its bagging method may be.

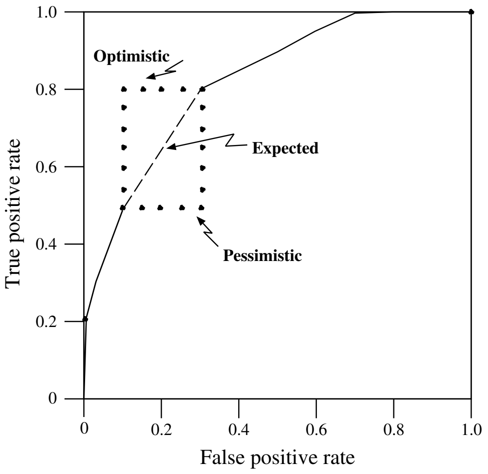


Fig. 6. The optimistic, pessimistic and expected ROC segments resulting from a sequence of 10 equally scored instances.

extreme, all the negatives end up at the beginning of the sequence and we get the "pessimistic" lower L shown in Fig. 6. Any mixed ordering of the instances will give a different set of step segments within the rectangle formed by these two extremes. However, the ROC curve should represent the *expected* performance of the classifier, which, lacking any other information, is the average of the pessimistic and optimistic segments. This average is the diagonal of the rectangle, and can be created in the ROC curve algorithm by not emitting an ROC point until all instances of equal f values have been processed. This is what the f_{prev} variable and the **if** statement of line 7 accomplish.

Instances that are scored equally may seem unusual but with some classifier models they are common. For example, if we use instance counts at nodes in a decision tree to score instances, a large, high-entropy leaf node may produce many equally scored instances of both classes. If such instances are not averaged, the resulting ROC curves will be sensitive to the test set ordering, and different orderings can yield very misleading curves. This can be especially critical in calculating the area under an ROC curve, discussed in Section 7. Consider a decision tree containing a leaf node accounting for n positives and m negatives. Every instance that is classified to this leaf node will be assigned the same score. The rectangle of Fig. 6 will be of size $\frac{nm}{PN}$, and if these instances are not averaged this one leaf may account for errors in ROC curve area as high as $\frac{nm}{2PN}$.

6. The ROC convex hull

One advantage of ROC graphs is that they enable visualizing and organizing classifier performance without regard to class distributions or error costs. This ability becomes very important when investigating learning with skewed distributions or cost-sensitive learning. A researcher can graph the performance of a set of classifiers, and that graph will remain invariant with respect to the operating conditions (class skew and error costs). As these conditions change, the region of interest may change, but the graph itself will not.

Provost and Fawcett (1998, 2001) show that a set of operating conditions may be transformed easily into a so-called *iso-performance line* in ROC space. Two points in ROC space, (FP_1, TP_1) and (FP_2, TP_2), have the same performance if

$$\frac{TP_2 - TP_1}{FP_2 - FP_1} = \frac{c(\boldsymbol{Y}, \boldsymbol{n})p(\boldsymbol{n})}{c(\boldsymbol{N}, \boldsymbol{p})p(\boldsymbol{p})} = m \tag{1}$$

This equation defines the slope of an iso-performance line. All classifiers corresponding to points on a line of slope m have the same expected cost. Each set of class and cost distributions defines a family of iso-performance lines. Lines "more northwest" (having a larger TP-intercept) are better because they correspond to classifiers with lower expected cost. More generally, a classifier is potentially optimal if and only if it lies on the convex hull of the set of points in ROC space. The convex hull of the set of points in ROC space is called the *ROC convex hull* (ROCCH) of the corresponding set of classifiers.

Fig. 7a shows four ROC curves (A through D) and their convex hull (labeled CH). D is not on the convex hull and is clearly sub-optimal. B is also not optimal for any conditions because it is not on the convex hull either. The convex hull is bounded only by points from curves A and C. Thus, if we are seeking optimal classification performance, classifiers B and D may be removed entirely from consideration. In addition, we may remove any discrete points from A and C that are not on the convex hull.

Fig. 7b shows the A and C curves again with two explicit iso-performance lines, α and β. Consider a scenario in which negatives outnumber positives by 10 to 1, but false positives and false negatives have equal cost. By Eq. (1) $m = 10$, and the most northwest line of slope $m = 10$ is α, tangent to classifier A, which would be the best performing classifier for these conditions.

Consider another scenario in which the positive and negative example populations are evenly balanced but a false negative is 10 times as expensive as a false positive. By Eq. (1) $m = 1/10$. The most northwest line of slope 1/10 would be line β, tangent to classifier C. C is the optimal classifier for these conditions.

If we wanted to generate a classifier somewhere on the convex hull between A and C, we could interpolate between the two. Section 10 explains how to generate such a classifier.

This ROCCH formulation has a number of useful implications. Since only the classifiers on the convex hull are potentially optimal, no others need be retained. The operating conditions of the classifier may be translated into an iso-performance line, which in turn may be used to identify a portion of the ROCCH. As conditions change, the hull itself does not change; only the portion of interest will.

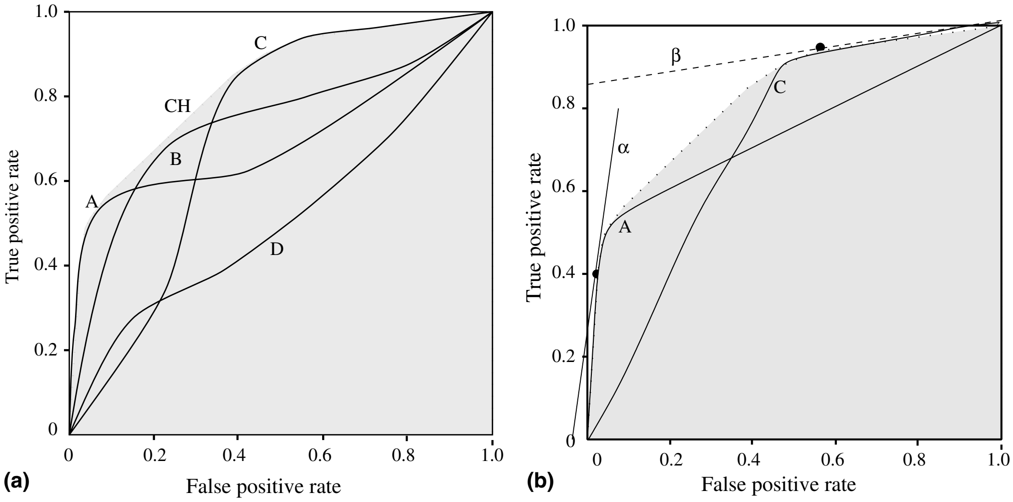


Fig. 7. (a) The ROC convex hull identifies potentially optimal classifiers. (b) Lines α and β show the optimal classifier under different sets of conditions.

7. Area under an ROC curve (AUC)

An ROC curve is a two-dimensional depiction of classifier performance. To compare classifiers we may want to reduce ROC performance to a single scalar value representing expected performance. A common method is to calculate the area under the ROC curve, abbreviated **AUC** (Bradley, 1997; Hanley and McNeil, 1982). Since the AUC is a portion of the area of the unit square, its value will always be between 0 and 1.0. However, because random guessing produces the diagonal line between (0, 0) and (1, 1), which has an area of 0.5, no realistic classifier should have an AUC less than 0.5.

The AUC has an important statistical property: the AUC of a classifier is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. This is equivalent to the Wilcoxon test of ranks (Hanley and McNeil, 1982). The AUC is also closely related to the Gini coefficient (Breiman et al., 1984), which is twice the area between the diagonal and the ROC curve. Hand and Till (2001) point out that Gini + 1 = 2 × AUC.

Fig. 8a shows the areas under two ROC curves, A and B. Classifier B has greater area and therefore better average performance. Fig. 8b shows the area under the curve of a binary classifier A and a scoring classifier B. Classifier A represents the performance of B when B is used with a single, fixed threshold. Though the performance of the two is equal at the fixed point (A's threshold), A's performance becomes inferior to B further from this point.

It is possible for a high-AUC classifier to perform worse in a specific region of ROC space than a low-AUC classifier. Fig. 8a shows an example of this: classifier B is generally better than A except at FPrate > 0.6 where A has a

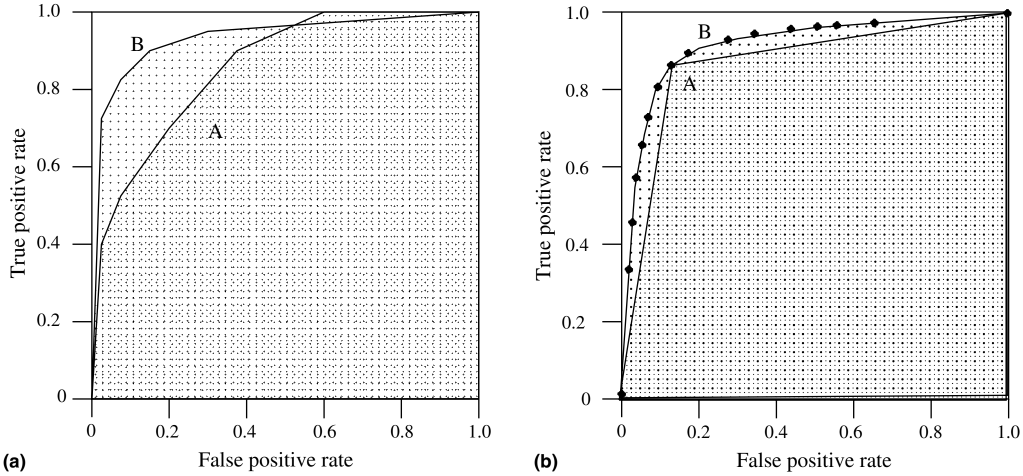


Fig. 8. Two ROC graphs. The graph on the left shows the area under two ROC curves. The graph on the right shows the area under the curves of a discrete classifier (A) and a probabilistic classifier (B).

Algorithm 2. Calculating the area under an ROC curve
Inputs: L, the set of test examples; $f(i)$, the probabilistic classifier's estimate that example i is positive; P and N, the number of positive and negative examples.
Outputs: A, the area under the ROC curve.
Require: $P > 0$ and $N > 0$

```
$L_{\text{sorted}} \leftarrow L$ sorted decreasing by $f$ scores
$FP \leftarrow TP \leftarrow 0$
$FP_{\text{prev}} \leftarrow TP_{\text{prev}} \leftarrow 0$
$A \leftarrow 0$
$f_{\text{prev}} \leftarrow -\infty$
$i \leftarrow 1$
while $i \leqslant |L_{\text{sorted}}|$ do
  if $f(i) \neq f_{\text{prev}}$ then
    $A \leftarrow A + \text{TRAPEZOID\_AREA}(FP, FP_{\text{prev}}, TP, TP_{\text{prev}})$
    $f_{\text{prev}} \leftarrow f(i)$
    $FP_{\text{prev}} \leftarrow FP$
    $TP_{\text{prev}} \leftarrow TP$
  end if
  if $i$ is a positive example then
    $TP \leftarrow TP + 1$
  else      /* $i$ is a negative example */
    $FP \leftarrow FP + 1$
  end if
  $i \leftarrow i + 1$
end while
$A \leftarrow A + \text{TRAPEZOID\_AREA}(N, FP_{\text{prev}}, N, TP_{\text{prev}})$
$A \leftarrow A/(P \times N)$   /* scale from $P \times N$ onto the unit square */
end
function TRAPEZOID_AREA($X1, X2, Y1, Y2$)
  $Base \leftarrow |X1 - X2|$
  $Height_{\text{avg}} \leftarrow (Y1 + Y2)/2$
  return $Base \times Height_{\text{avg}}$
end function
```

slight advantage. But in practice the AUC performs very well and is often used when a general measure of predictiveness is desired.

The AUC may be computed easily using a small modification of algorithm 1, shown in Algorithm 2. Instead of collecting ROC points, the algorithm adds successive areas of trapezoids to A. Trapezoids are used rather than rectangles in order to average the effect between points, as illustrated in Fig. 6. Finally, the algorithm divides A by the total possible area to scale the value to the unit square.

8. Averaging ROC curves

Although ROC curves may be used to evaluate classifiers, care should be taken when using them to make conclusions about classifier superiority. Some researchers have assumed that an ROC graph may be used to select the best classifiers simply by graphing them in ROC space and seeing which ones dominate. This is misleading; it is analogous to taking the maximum of a set of accuracy figures from a single test set. Without a measure of variance we cannot compare the classifiers.

Averaging ROC curves is easy if the original instances are available. Given test sets $T_1, T_2, \ldots, T_n$, generated from cross-validation or the bootstrap method, we can simply merge sort the instances together by their assigned scores into one large test set T_M. We then run an ROC curve generation algorithm such as algorithm 1 on T_M and plot the result. However, the primary reason for using multiple test sets is to derive a measure of variance, which this simple merging does not provide. We need a more sophisticated method that samples individual curves at different points and averages the samples.

ROC space is two-dimensional, and any average is necessarily one-dimensional. ROC curves can be projected onto a single dimension and averaged conventionally, but this leads to the question of whether the projection is appropriate, or more precisely, whether it preserves characteristics of interest. The answer depends upon the reason for averaging the curves. This section presents two methods for averaging ROC curves: vertical and threshold averaging.

Fig. 9a shows five ROC curves to be averaged. Each contains a thousand points and has some concavities. Fig. 9b shows the curve formed by merging the five test sets and computing their combined ROC curve. Fig. 9c and d shows average curves formed by sampling the five individual ROC curves. The error bars are 95% confidence intervals.

8.1. Vertical averaging

Vertical averaging takes vertical samples of the ROC curves for fixed FP rates and averages the corresponding TP rates. Such averaging is appropriate when the FP rate can indeed be fixed by the researcher, or when a single-dimensional measure of variation is desired. Provost et al. (1998) used this method in their work of averaging ROC curves of a classifier for k-fold cross-validation.

In this method each ROC curve is treated as a function, R_i, such that $tp\ rate = R_i(fp\ rate)$. This is done by choosing the maximum *tp rate* for each *fp rate* and interpolating between points when necessary. The averaged ROC curve is the function $\hat{R}(fp\ rate) = \text{mean}[R_i(fp\ rate)]$. To plot an average ROC curve we can sample from $\hat{R}$ at points regularly spaced along the *fp rate*-axis. Confidence intervals of the mean of *tp rate* are computed using the common assumption of a binomial distribution.

Algorithm 3 computes this vertical average of a set of ROC points. It leaves the means in the array *TPavg*.

Several extensions have been left out of this algorithm for clarity. The algorithm may easily be extended to

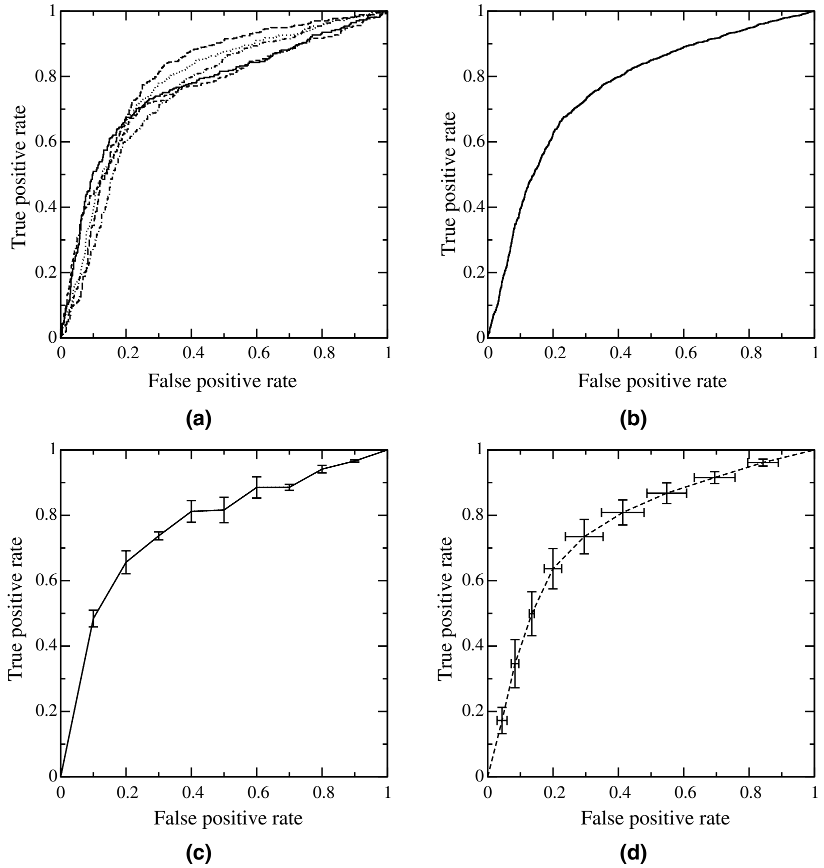


Fig. 9. ROC curve averaging. (a) ROC curves of five instance samples, (b) ROC curve formed by merging the five samples, (c) the curves of a averaged vertically and (d) the curves of a averaged by threshold.

compute standard deviations of the samples in order to draw confidence bars. Also, the function TP_FOR_FP may be optimized somewhat. Because it is only called on monotonically increasing values of *FP*, it need not scan each ROC array from the beginning every time; it could keep a record of the last point seen and initialize *i* from this array.

Fig. 9c shows the vertical average of the five curves in Fig. 9a. The vertical bars on the curve show the 95% confidence region of the ROC mean. For this average curve, the curves were sampled at FP rates from 0 through 1 by 0.1. It is possible to sample curves much more finely but the confidence bars may become difficult to read.

8.2. Threshold averaging

Vertical averaging has the advantage that averages are made of a single dependent variable, the true positive rate, which simplifies computing confidence intervals. However, Holte (2002) has pointed out that the independent variable, false positive rate, is often not under the direct control of the researcher. It may be preferable to average ROC points using an independent variable whose value can be controlled directly, such as the threshold on the classifier scores.

Threshold averaging accomplishes this. Instead of sampling points based on their positions in ROC space, as vertical averaging does, it samples based on the thresholds that produced these points. The method must generate a set of thresholds to sample, then for each threshold it finds the corresponding point of each ROC curve and averages them.

Algorithm 4 shows the basic method for doing this. It generates an array *T* of classifier scores which are sorted from largest to smallest and used as the set of thresholds. These thresholds are sampled at fixed intervals determined by *samples*, the number of samples desired. For a given threshold, the algorithm selects from each ROC curve the point of greatest score less than or equal to the threshold.[4] These points are then averaged separately along their *X* and *Y* axes, with the center point returned in the *Avg* array.

Fig. 9d shows the result of averaging the five curves of Fig. 9a by thresholds. The resulting curve has average points and confidence bars in the *X* and *Y* directions. The bars shown are at the 95% confidence level.

There are some minor limitations of threshold averaging with respect to vertical averaging. To perform threshold averaging we need the classifier score assigned to each point. Also, Section 4.1 pointed out that classifier scores

[4] We assume the ROC points have been generated by an algorithm like 1 that deals correctly with equally scored instances.

Algorithm 3. Vertical averaging of ROC curves
Inputs: *samples*, the number of FP samples; *nrocs*, the number of ROC curves to be sampled, *ROCS*[*nrocs*], an array of *nrocs* ROC curves; *npts*[*m*], the number of points in ROC curve *m*. Each ROC point is a structure of two members, the rates *fpr* and *tpr*.
Output: Array *tpravg*[*samples* + 1], containing the vertical averages.

```
$s \leftarrow 1$
for $fpr_{\text{sample}} = 0$ to $1$ by $1/samples$ do
    $tprsum \leftarrow 0$
    for $i = 1$ to nrocs do
        $tprsum \leftarrow tprsum + \text{TPR\_FOR\_FPR}(fpr_{\text{sample}},$
                $ROCS[i], npts[i])$
    end for
    $tpravg[s] \leftarrow tprsum/nrocs$
    $s \leftarrow s + 1$
end for
end
function $\text{TPR\_FOR\_FPR}(fpr_{\text{sample}}, ROC, npts)$
$i \leftarrow 1$
while $i < npts$ and $ROC\ [i+1].fpr \leqslant fpr_{\text{sample}}$ do
    $i \leftarrow i + 1$
end while
if $ROC[i].fpr = fpr_{\text{sample}}$ then
    return $ROC[i].tpr$
else
    return $\text{INTERPOLATE}(ROC[i], ROC\ [i+1], fpr_{\text{sample}})$
end if
end function
function $\text{INTERPOLATE}(ROCP1, ROCP2, X)$
slope $= (ROCP2.tpr - ROCP1.tpr)/(ROCP2.fpr -$
        $ROCP1.fpr)$
return $ROCP1.tpr + \text{slope} \cdot (X - ROCP1.fpr)$
end function
```

Algorithm 4. Threshold averaging of ROC curves
Inputs: *samples*, the number of threshold samples; *nrocs*, the number of ROC curves to be sampled; *ROCS*[*nrocs*], an array of *nrocs* ROC curves sorted by score; *npts*[*m*], the number of points in ROC curve *m*. Each ROC point is a structure of three members, *fpr*, *tpr* and score.
Output: *Avg*[*samples* + 1], an array of (*X*, *Y*) points constituting the average ROC curve.
Require: *samples* > 1

```
initialize array $T$ to contain all scores of all ROC
    points
sort $T$ in descending order
$s \leftarrow 1$
for $tidx = 1$ to $length(T)$ by $int(length(T)/samples)$ do
    $fprsum \leftarrow 0$
    $tprsum \leftarrow 0$
    for $i = 1$ to nrocs do
        $p \leftarrow \text{ROC\_POINT\_AT\_THRESHOLD}(ROCS[i], npts[i],$
            $T[tidx])$
        $fprsum \leftarrow fprsum + p.fpr$
        $tprsum \leftarrow tprsum + p.tpr$
    end for
    $Avg[s] \leftarrow (fprsum/nrocs, tprsum/nrocs)$
    $s \leftarrow s + 1$
end for
end
function $\text{ROC\_POINT\_AT\_THRESHOLD}(ROC, npts, thresh)$
$i \leftarrow 1$
while $i \leqslant npts$ and $ROC[i].\ score > thresh$ do
    $i \leftarrow i + 1$
end while
return $ROC[i]$
end function
```

should not be compared across model classes. Because of this, ROC curves averaged from different model classes may be misleading because the scores may be incommensurate.

Finally, Macskassy and Provost (2004) have investigated different techniques for generating confidence bands for ROC curves. They investigate confidence intervals from vertical and threshold averaging, as well as three methods from the medical field for generating bands (simultaneous join confidence regions, Working-Hotelling based bands, and fixed-width confidence bands). The reader is referred to their paper for a much more detailed discussion of the techniques, their assumptions, and empirical studies.

9. Decision problems with more than two classes

Discussions up to this point have dealt with only two classes, and much of the ROC literature maintains this assumption. ROC analysis is commonly employed in medical decision making in which two-class diagnostic problems—presence or absence of an abnormal condition—are common. The two axes represent tradeoffs between errors (false positives) and benefits (true positives) that a classifier makes between two classes. Much of the analysis is straightforward because of the symmetry that exists in the two-class problem. The resulting performance can be graphed in two dimensions, which is easy to visualize.

9.1. Multi-class ROC graphs

With more than two classes the situation becomes much more complex if the entire space is to be managed. With n classes the confusion matrix becomes an $n \times n$ matrix containing the n correct classifications (the major diagonal entries) and $n^2 - n$ possible errors (the off-diagonal entries). Instead of managing trade-offs between *TP* and *FP*, we have n benefits and $n^2 - n$ errors. With only three classes, the surface becomes a $3^2 - 3 = 6$-dimensional polytope. Lane (2000) has outlined the issues involved and the prospects for addressing them. Srinivasan (1999) has shown that the analysis behind the ROC convex hull extends to multiple classes and multi-dimensional convex hulls.

One method for handling n classes is to produce n different ROC graphs, one for each class. Call this the *class reference* formulation. Specifically, if C is the set of all classes, ROC graph i plots the classification performance using class c_i as the positive class and all other classes as the negative class, i.e.

$$P_i = c_i \tag{2}$$

$$N_i = \bigcup_{j \neq i} c_j \in C \tag{3}$$

While this is a convenient formulation, it compromises one of the attractions of ROC graphs, namely that they are insensitive to class skew (see Section 4.2). Because each N_i comprises the union of $n-1$ classes, changes in prevalence within these classes may alter the c_i's ROC graph. For example, assume that some class $c_k \in N$ is particularly easy to identify. A classifier for class c_i, $i \neq k$ may exploit some characteristic of c_k in order to produce low scores for c_k instances. Increasing the prevalence of c_k might alter the performance of the classifier, and would be tantamount to changing the target concept by increasing the prevalence of one of its disjuncts. This in turn would alter the ROC curve. However, with this caveat, this method can work well in practice and provide reasonable flexibility in evaluation.

9.2. Multi-class AUC

The AUC is a measure of the discriminability of a pair of classes. In a two-class problem, the AUC is a single scalar value, but a multi-class problem introduces the issue of combining multiple pairwise discriminability values. The reader is referred to Hand and Till's (2001) article for an excellent discussion of these issues.

One approach to calculating multi-class AUCs was taken by Provost and Domingos (2001) in their work on probability estimation trees. They calculated AUCs for multi-class problems by generating each class reference ROC curve in turn, measuring the area under the curve, then summing the AUCs weighted by the reference class's prevalence in the data. More precisely, they define

$$\mathrm{AUC}_{\mathrm{total}} = \sum_{c_i \in C} \mathrm{AUC}(c_i) \cdot p(c_i)$$

where $\mathrm{AUC}(c_i)$ is the area under the class reference ROC curve for c_i, as in Eq. (3). This definition requires only $|C|$ AUC calculations, so its overall complexity is $\mathrm{O}(|C| n \log n)$.

The advantage of Provost and Domingos's AUC formulation is that $\mathrm{AUC}_{\mathrm{total}}$ is generated directly from class reference ROC curves, and these curves can be generated and visualized easily. The disadvantage is that the class reference ROC is sensitive to class distributions and error costs, so this formulation of $\mathrm{AUC}_{\mathrm{total}}$ is as well.

Hand and Till (2001) take a different approach in their derivation of a multi-class generalization of the AUC. They desired a measure that is insensitive to class distribution and error costs. The derivation is too detailed to summarize here, but it is based upon the fact that the AUC is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. From this probabilistic form, they derive a formulation that measures the unweighted *pairwise* discriminability of classes. Their measure, which they call M, is equivalent to:

$$\mathrm{AUC}_{\mathrm{total}} = \frac{2}{|C|(|C|-1)} \sum_{\{c_i, c_j\} \in C} \mathrm{AUC}(c_i, c_j)$$

where n is the number of classes and $\mathrm{AUC}(c_i, c_j)$ is the area under the two-class ROC curve involving classes c_i and c_j. The summation is calculated over all pairs of distinct classes, irrespective of order. There are $|C|(|C|-1)/2$ such pairs, so the time complexity of their measure is $\mathrm{O}(|C|^2 n \log n)$. While Hand and Till's formulation is well justified and is insensitive to changes in class distribution, there is no easy way to visualize the surface whose area is being calculated.

10. Interpolating classifiers

Sometimes the performance desired of a classifier is not exactly produced by any available classifier, but lies between two available classifiers. The desired performance can be obtained by sampling the decisions of each classifier. The sampling ratio will determine where the resulting classification performance lies.

For a concrete example, consider the decision problem of the CoIL Challenge 2000 (van der Putten and Someren, 2000). In this challenge there is a set of 4000 clients to whom we wish to market a new insurance policy. Our budget dictates that we can afford to market to only 800 of them, so we want to select the 800 who are most likely to respond to the offer. The expected class prior of responders is 6%, so within the population of 4000 we expect to have 240 responders (positives) and 3760 non-responders (negatives).

Assume we have generated two classifiers, A and B, which score clients by the probability they will buy the policy. In ROC space A lies at (0.1, 0.2) and B lies at (0.25, 0.6), as shown in Fig. 10. We want to market to exactly 800 people so our solution constraint is $\textit{fp rate} \times 3760 + \textit{tp rate} \times 240 = 800$. If we use A we expect $0.1 \times 3760 + 0.2 \times 240 = 424$ candidates, which is too few. If we use B we expect $0.25 \times 3760 + 0.6 \times 240 = 1084$ candidates, which is too many. We want a classifier between A and B.

The solution constraint is shown as a dashed line in Fig. 10. It intersects the line between A and B at C, approximately (0.18, 0.42). A classifier at point C would give the performance we desire and we can achieve it using linear interpolation. Calculate k as the proportional distance that C lies on the line between A and B:

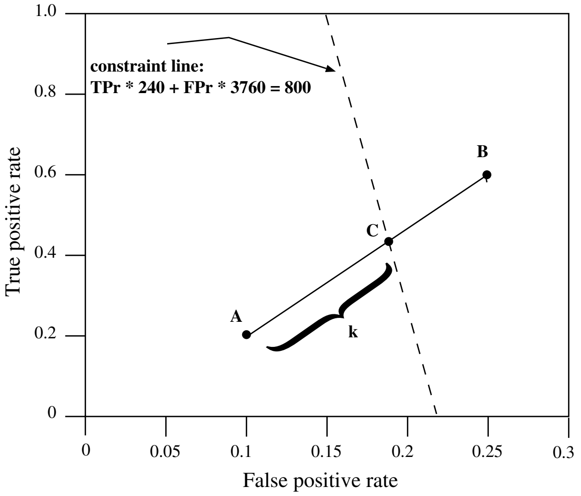


Fig. 10. Interpolating classifiers.

$$k = \frac{0.18 - 0.1}{0.25 - 0.1} \approx 0.53$$

Therefore, if we sample B's decisions at a rate of 0.53 and A's decisions at a rate of $1 - 0.53 = 0.47$ we should attain C's performance. In practice this fractional sampling can be done by randomly sampling decisions from each: for each instance, generate a random number between zero and one. If the random number is greater than k, apply classifier A to the instance and report its decision, else pass the instance to B.

11. Conclusion

ROC graphs are a very useful tool for visualizing and evaluating classifiers. They are able to provide a richer measure of classification performance than scalar measures such as accuracy, error rate or error cost. Because they decouple classifier performance from class skew and error costs, they have advantages over other evaluation measures such as precision-recall graphs and lift curves. However, as with any evaluation metric, using them wisely requires knowing their characteristics and limitations. It is hoped that this article advances the general knowledge about ROC graphs and helps to promote better evaluation practices in the pattern recognition community.

References

Bradley, A.P., 1997. The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern Recogn. 30 (7), 1145–1159.

Breiman, L., Friedman, J., Olshen, R., Stone, C., 1984. Classification and Regression Trees. Wadsworth International Group, Belmont, CA.

Clearwater, S., Stern, E., 1991. A rule-learning program in high energy physics event classification. Comput. Phys. Commun. 67, 159–182.

Domingos, P., 1999. MetaCost: A general method for making classifiers cost-sensitive. In: Proc. Fifth ACM SIGKDD Internat. Conf. on Knowledge Discovery and Data Mining, pp. 155–164.

Egan, J.P., 1975. Signal detection theory and ROC analysis, Series in Cognition and Perception. Academic Press, New York.

Fawcett, T., 2001. Using rule sets to maximize ROC performance. In: Proc. IEEE Internat. Conf. on Data Mining (ICDM-2001), pp. 131–138.

Fawcett, T., Provost, F., 1996. Combining data mining and machine learning for effective user profiling. In: Simoudis, E., Han, J., Fayyad, U. (Eds.), Proc. Second Internat. Conf. on Knowledge Discovery and Data Mining. AAAI Press, Menlo Park, CA, pp. 8–13.

Fawcett, T., Provost, F., 1997. Adaptive fraud detection. Data Mining and Knowledge Discovery 1 (3), 291–316.

Flach, P., Wu, S., 2003. Repairing concavities in ROC curves. In: Proc. 2003 UK Workshop on Computational Intelligence. University of Bristol, pp. 38–44.

Forman, G., 2002. A method for discovering the insignificance of one's best classifier and the unlearnability of a classification task. In: Lavrac, N., Motoda, H., Fawcett, T. (Eds.), Proc. First Internat. Workshop on Data Mining Lessons Learned (DMLL-2002). Available from: http://www.purl.org/NET/tfawcett/DMLL-2002/Forman.pdf.

Hand, D.J., Till, R.J., 2001. A simple generalization of the area under the ROC curve to multiple class classification problems. Mach. Learning 45 (2), 171–186.

Hanley, J.A., McNeil, B.J., 1982. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology 143, 29–36.

Holte, R., 2002. Personal communication.

Kubat, M., Holte, R.C., Matwin, S., 1998. Machine learning for the detection of oil spills in satellite radar images. Machine Learning 30 (2–3), 195–215.

Lane, T., 2000. Extensions of ROC analysis to multi-class domains. In: Dietterich, T., Margineantu, D., Provost, F., Turney, P. (Eds.), ICML-2000 Workshop on Cost-Sensitive Learning.

Lewis, D., 1990. Representation quality in text classification: An introduction and experiment. In: Proc. Workshop on Speech and Natural Language. Morgan Kaufmann, Hidden Valley, PA, pp. 288–295.

Lewis, D., 1991. Evaluating text categorization. In: Proc. Speech and Natural Language Workshop. Morgan Kaufmann, pp. 312–318.

Macskassy, S., Provost, F., 2004. Confidence bands for ROC curves: Methods and an empirical study. In: Proc. First Workshop on ROC Analysis in AI (ROCAI-04).

Provost, F., Domingos, P., 2001. Well-trained PETs: Improving probability estimation trees, CeDER Working Paper #IS-00-04, Stern School of Business, New York University, NY, NY 10012.

Provost, F., Fawcett, T., 1997. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions. In: Proc. Third Internat. Conf. on Knowledge Discovery and Data Mining (KDD-97). AAAI Press, Menlo Park, CA, pp. 43–48.

Provost, F., Fawcett, T., 1998. Robust classification systems for imprecise environments. In: Proc. AAAI-98. AAAI Press, Menlo Park, CA, pp. 706–713. Available from: <http://www.purl.org/NET/tfawcett/papers/aaai98-dist.ps.gz>.

Provost, F., Fawcett, T., 2001. Robust classification for imprecise environments. Mach. Learning 42 (3), 203–231.

Provost, F., Fawcett, T., Kohavi, R., 1998. The case against accuracy estimation for comparing induction algorithms. In: Shavlik, J. (Ed.), Proc. ICML-98. Morgan Kaufmann, San Francisco, CA, pp. 445–453. Available from: <http://www.purl.org/NET/tfawcett/papers/ICML98-final.ps.gz>.

Saitta, L., Neri, F., 1998. Learning in the "real world". Mach. Learning 30, 133–163.

Spackman, K.A., 1989. Signal detection theory: Valuable tools for evaluating inductive learning. In: Proc. Sixth Internat. Workshop on Machine Learning. Morgan Kaufman, San Mateo, CA, pp. 160–163.

Srinivasan, A., 1999. Note on the location of optimal classifiers in n-dimensional ROC space. Technical Report PRG-TR-2-99, Oxford University Computing Laboratory, Oxford, England. Available from: <http://citeseer.nj.nec.com/srinivasan99note.html>.

Swets, J., 1988. Measuring the accuracy of diagnostic systems. Science 240, 1285–1293.

Swets, J.A., Dawes, R.M., Monahan, J., 2000. Better decisions through science. Scientific American 283, 82–87.

van der Putten, P., van Someren, M., 2000. CoIL challenge 2000: The insurance company case. Technical Report 2000–09, Leiden Institute of Advanced Computer Science, Universiteit van Leiden. Available from: <http://www.liacs.nl/putten/library/cc2000>.

Zadrozny, B., Elkan, C., 2001. Obtaining calibrated probability estimates from decision trees and naive Bayesian classiers. In: Proc. Eighteenth Internat. Conf. on Machine Learning, pp. 609–616.

Zou, K.H., 2002. Receiver operating characteristic (ROC) literature research. On-line bibliography available from: <http://splweb.bwh.harvard.edu:8000/pages/ppl/zou/roc.html>.

7.2 What you should know by now

You should be able to *mathematically* define and calculate the sensitivity and specificity of a model from a confusion matrix. You should be able to do the same for F-measure, and why it is useful. You should know *why* ROC analysis via these quantities is necessary and under which problem scenarios it is most useful. You should know how to evaluate a ROC curve by eye, judging the performance of one model against another.

Chapter 8

Probabilistic Models

THE BEST WAY TO MAKE THIS DECISION IS BY CALCULATING THE EXPECTED VALUE OF EACH POSSIBLE OUTCOME.
scottadams@aol.com
www.dilbert.com
YOU MULTIPLY THE. . .
MUST PRETEND TO BE DEAD.
7-23-05 ©2005 Scott Adams, Inc./Dist. by UFS, Inc.
I SENSE THAT WE'RE DONE HERE.
I HOPE THE DEAD SOME-TIMES COVER THEIR EARS.

The models we've seen so far have been geometric (perceptrons/SVMs), distance-based (k-NN) or trees (decision trees). In this chapter we're going to see a new class of models, those based on *probability theory.* This is in fact one of the hottest areas of Machine Learning right now, being a major focus for research labs like Microsoft and Google. To understand these models will require knowledge of probability theory, specifically *conditional probabilities*, and *joint probabilities.* We will review these in this first section – feel free to skip it if you're already comfortable with these ideas – before moving onto *Bayes' Theorem*, and the probabilistic models that result from using it.

CONDITIONAL / JOINT PROBABILITY

If you find the style of this chapter not to your liking, try reading Appendix B, which provides virtually the same material, but written in a different way.

8.1 Probability Basics

If I flip a coin, what is the *probability* that it will land heads up? The answer to this question is quite intuitive for most people. But, people might answer slightly differently: *"the probability is a half"*, or *"the probability is 50:50"*, or "the probability is 50%". To be mathematically rigorous,we should in fact say *"the probability is 0.5"*. This is because *probabilities* are always between 0 and 1 — different from percentages, which are between 0 and 100. Though the mapping is easy between these two, it is good practice to be strict with yourself[1].

PROBABILITIES

8.1.1 Random Variables

If you wrote a computer program to store a variable called **coinFaceUp**, you would say something like **coinFaceUp = "HEAD"**. The value of this variable is deterministic – it's either set to something, or it's not. In reality, the side of the coin facing upward is a *random* event. As humans, we face events with uncertainty all the time, for example what the weather will be tomorrow. We therefore need a notation to deal with variables that have randomness — where the value assigned to a variable can be "HEAD" with probability 0.5, or "TAIL" with probability 0.5. These are called *random variables.*

RANDOM VARIABLE

We denote the *alphabet* of the random variable as $\{head, tail\}$, and the probability of a particular event as $p(coinFaceUp = head) = 0.5$. We could also take a more abstract naming convention, and refer to a variable X, for which the possible values (alphabet) are $\mathcal{X} = \{0, 1\}$, and $p(X = 1) = 0.5$. So, using this notation, let's consider rolling a dice. In this case, $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$ and for example $p(X = 3) = \frac{1}{6}$.

[1] Imagine if you were modifying a computer program that navigated a spacecraft to Mars – you find a variable called **chanceOfSolarFlare**, set equal to 0.8. You interpret it incorrectly as being in the range $[0, 100]$, a percentage, whereas in reality the programmer set it in the range $[0, 1]$. You would end up believing there was a very small chance of a flare, whereas in fact there is a very large chance!

The *first rule of probability theory is* that probabilities always add up to 1. What I mean by this is simply that if I tell you there is a variable X, with alphabet $\{0, 1, 2\}$, then $p(X = 1) + p(X = 2) + p(X = 3) = 1$. In more general terms:

$$\sum_{x\in\mathcal{X}} p(X = x) = 1$$

A useful consequence of this is that for a variable X, with alphabet $\{0, 1\}$, and $p(X = 0) = 0.2$, we immediately know that:

$$p(X = 1) = \ 1 - p(X = 0) \ = 0.8$$

Before we move onto the next section, a quick note — the notation we've been using is nicely explicit, though can be a little laborious sometimes. As a shorthand notation, to represent the probability of an arbitrary event from the random variable X, we will just use $p(x) = p(X = x)$, where the lower case x is any possible event from the corresponding upper case X.

8.1.2 Estimated Probabilities vs. True Probabilities

We have considered random variables which are genuine random events, like dice and coin flips. We can also consider probabilities observed from data. Imagine a census of the UK population. Let's say there are 64 million people. What is the probability of a male in this population? If there are 31 million men, that makes $p(male) = \frac{31}{64} = 0.484375$.

If I didn't have access to all the census data, and I wanted to *estimate* the male population in the UK, I could do it by finding a reasonable sized *sample* of people, and counting the males/females.

Here, I found 27 people, 11 of whom were male. I estimate the UK male population to be $\frac{11}{27} \approx 0.41$. We denote the estimate by a 'hat', as so: $\hat{p}(male)$.

True probability .... $p(male) = 0.484375$
Estimated probability ...... $\hat{p}(male) = 0.41$.

The true probability $p(male)$ is also known as a *population* parameter, while $\hat{p}(male)$ is known as a *sample estimate* of that parameter. Intuitively, you can imagine that if I found a larger sample of people (more than 27) then I would get closer and closer to the population value. Unfortunately of course, data is usually limited, so we rely on estimation a lot in this field. The frequency counting method above is called *maximum likelihood* estimation, but there are other ways to estimate than just this method. Look up the details online or in another book if you want to know more – the method of "Laplace correction" is a good starting point.

MAXIMUM LIKELIHOOD

8.1.3 Joint Probabilities

The probability of an event (such as finding a male person in the UK) can be simply extended to the concept of *joint probabilities.* For example the probability of a male and being over 180cm tall. In formal mathematical notation this is written in different ways depending on which textbook you read:

$$p(male \text{ AND } tall) \qquad p(male \wedge tall)$$

$$p(male, tall) \qquad p(male \cap tall)$$

...but they all mean the same thing. We will use the $p(male, tall)$ form. Take a look at the following dataset, a sample of 10 examples from a larger population.

person id	gender	tall?	bloodtype
1	M	Y	A
2	M	Y	B
3	M	N	O
4	M	N	B
5	F	Y	A
6	F	Y	O
7	F	N	B
8	F	N	B
9	F	N	O
10	F	N	A

Table 8.1: Some data, sampled from a larger population of people.

You should be able to work out from this table that $\hat{p}(male, tall) = 0.2$. This is because we find the number of examples where we have a tall male (persons 1+2) and divide that by the total number of people, so $\frac{2}{10}$. Notice again that we have *estimated* the population probability, with a maximum likelihood estimate. Similarly, you should be able to confirm for a female with bloodtype A, that the estimated probability is $\hat{p}(female, A) = \frac{2}{10}$.

SELF-TEST
1. What is the estimated probability of being tall?
2. What is $\hat{p}(bloodtype = O)$?
3. Do we know the value of $p(bloodtype = A)$?

Top tip: Read Q3 carefully ;-)

8.1.4 Conditional Probabilities

This is the final bit of tutorial material before we move onto the focus of this chapter. Conditional probabilities are simply the probability of one thing being true, given that we know a certain other thing is true. For example, the probability of having a blood type A, *given that you are male.* In our formal notation, we will denote this as:

$$p(bloodtype = A \mid gender = male)$$

The little bar '|' notation, should be read "*given that*". This can also be interpreted as the prevalence of blood type A in the male population. From our data, we could estimate this as:

$$\hat{p}(bloodtype = A \mid gender = male) \;\; = \;\; \frac{1}{4}$$

This is estimated by simply finding all the people that are male (persons #1-4) and the fraction of those that have bloodtype A, which is just person #1, thus the estimated probability is 0.25.

person id	gender	tall?	bloodtype
1	**M**	**Y**	**A**
2	M	Y	B
3	M	N	O
4	M	N	B
5	F	Y	A
6	F	Y	O
7	F	N	B
8	F	N	B
9	F	N	O
10	F	N	A

Table 8.2: The probability of having bloodtype A in the population of males is denoted $p(bloodtype = A \mid gender = male)$, and is estimated to be 0.25.

A very important thing to note is that $p(x|y) \neq p(y|x)$. If we estimate the probability of being male, given that your blood type is A, then is it:

$$\hat{p}(gender = male \mid bloodtype = A) \;=\; \tfrac{1}{3}$$

Again, you should be able to calculate these simple conditional probabilities just by finding the examples that satisfy the condition *after* the vertical bar '|', and finding the fraction of those that satisfy the condition *before* it.

SELF-TEST
Work out the maximum likelihood estimates of the following probabilities. Note that on some questions I have used the notation $p(gender = male)$, and on others just $p(male)$. Different people use different notations, so you should get used to understanding what is meant in different situations.

$p(female)$	$p(tall = Y)$
$p(bloodtype = O)$	$p(female, tall)$
$p(tall \mid female)$	$p(female \mid tall)$

We can in fact relate these two concepts of conditional and joint probabilities. The rule for arbitrary variables X and Y is:

$$p(x, y) = p(x|y)p(y)$$

We can check this with our dataset, using a shorthand notation $p(typeA)$ for the probability of someone's blood type being A, we estimate the probabilities:

$$\begin{array}{ccccc} \hat{p}(male, typeA) & = & \hat{p}(male \mid typeA) & \hat{p}(typeA) & \\ \frac{1}{10} & = & \frac{1}{3} & \times & \frac{3}{10} \end{array}$$

Interestingly, this rule holds *the other way round too.* What I mean is:

$$p(x, y) = p(y|x)p(x)$$

This is simply because the order of the arguments in $p(X, Y)$ do not matter, i.e. $p(X, Y) = p(Y, X)$. Checking it with the same example:

$$\begin{array}{ccccccc} \hat{p}(male, typeA) & = & \hat{p}(typeA, male) & = & \hat{p}(typeA \mid male)\hat{p}(male) & & \\ \frac{1}{10} & = & \frac{1}{10} & = & \frac{1}{4} & \times & \frac{4}{10} \end{array}$$

Check this yourself for other combinations of variables in the dataset. You should find it holds in all cases.

The final little rule of probability theory we will cover concerns *independent events*. Imagine we've gone to a Las Vegas casino — X is a variable representing a dice roll, and Y is a variable representing another dice roll. What is the probability of rolling two sixes?

We know each die has an alphabet $\{1, 2, 3, 4, 5, 6\}$, and they are completely random, so $p(X = x) = \frac{1}{6}$, for any value of x. If I roll the two dice at once, then we are talking about the *joint probability*, $p(X, Y)$. So, what is the probability $p(X = 6, Y = 6)$? Using the rule to relate conditional and joint probabilities, we know from earlier:

$$p(X = 6, Y = 6) \;=\; p(X = 6|Y = 6)p(Y = 6)$$

...we know that $p(y)$ is $\frac{1}{6}$, but what is $p(x|y)$? Well, translating the maths into plain english, $p(x|y)$ reads, "what is the probability of a 6 from the X dice, *given that* the Y dice is a 6?". We know from simple logic that these events are in fact *independent*, so $p(x|y)$ is just equal to $p(x)$, which is again $\frac{1}{6}$. **The rule for independent events is therefore:**

$$p(x, y) \;=\; p(x)p(y).$$

Check it for yourself with the concept of a coin flip, or maybe a rolling three dice, or any other events you know to be independent of one another.

This has been an *exceptionally* short introduction to probability theory, designed to just give you a feel for the maths we will be doing. There are things *not mentioned* in here, that may come up in the more advanced aspects of this course. To brush up on more detailed aspects, I suggest you consult a textbook (the *Schaum's Outlines* series is generally good) or Google for something like "tutorial probability basics". We do have all the machinery necessary to understand a very important tool for modern Machine Learning: *Bayes' Theorem*.

SCHAUM'S OUTLINES

8.2 Bayes' Theorem

In the mid-18th century, an Englishman named Thomas Bayes discovered a small but very significant piece of mathematics, wrote it in his diaries, and promptly died. It was published after his death by a friend, and forgotten. Many years later, the famous mathematician Pierre-Simon de Laplace, uncovered[2] just what he had done. Laplace developed the ideas significantly further, though the main theorem made use of today is named after Thomas Bayes, and this one idea has revolutionised modern machine learning in the past 15 years.

Figure 8.1: Reverend Thomas Bayes, 1701-1761.

8.2.1 Deriving the Theorem

Imagine we have two random variables, X and Y, both of which have possible values $\{0, 1\}$. **Bayes' theorem applies for more than just binary variables, but this is the simplest scenario to start with.** Let's apply some of rules we know for joint and conditional probabilities. For any values in the X, Y alphabets:

$$p(x, y) = p(x|y)p(y),$$

...but remember it works the other way round:

$$p(x, y) = p(y, x) = p(y|x)p(x).$$

Combining these two results, we have:

$$p(y|x)p(x) = p(x|y)p(y).$$

and finally, dividing both sides by $p(x)$, we get...

$$p(y|x) \;=\; \frac{p(x|y)p(y)}{p(x)}, \tag{8.1}$$

... which is **Bayes' Theorem**. Just an equation, but let's now see what it really *means*, and what it enables us to do.

[2] http://lesswrong.com/lw/774/a_history_of_bayes_theorem/

First we'll apply the theorem to our data on people from Table 8.1. Using Bayes' theorem, and with $X = \{male, female\}$, and $Y = \{tall, not\text{-}tall\}$.

$$p(Y = tall \mid X = male) = \frac{p(X = male \mid Y = tall)p(Y = tall)}{p(X = male)} \tag{8.2}$$

and estimating the various probabilities...

$$\begin{aligned}
\hat{p}(Y = tall \mid X = male) &= 0.5 \\
\hat{p}(X = male \mid Y = tall) &= 0.5 \\
\hat{p}(Y = tall) &= 0.4 \\
\hat{p}(X = male) &= 0.4
\end{aligned}$$

We find it does indeed hold, since $\hat{p}(Y = tall \mid X = male) = \frac{0.5 \times 0.4}{0.4} = 0.5$.

This is correct, but seems kind of strange because we could just work out the probability $p(y|x)$ by doing frequency counts from the data. The power of Bayes' Theorem comes when you can't do this - and have to make *predictions*. We will see the power of Bayes' theorem, by applying it in a real world situation to make a prediction.

8.2.2 Diagnosing Diabetes

According to Wikipedia, there is a world average incidence of diabetes of 2.8%, or in probability notation, $p(Y = diabetes) = 0.028$. There is a medical test, the *Oral Glucose Tolerance Test*, which measures the glucose levels in a person's blood — if above a certain threshold, the test predicts that you have diabetes. We will denote the outcome of this test by the random variable X: DIABETES

$$X = \{positive,\ negative\}, \qquad Y = \{diabetes,\ no\text{-}diabetes\}$$

The OGTT test is worldwide accepted, and very sensitive — it will detect diabetes correctly 95% of the time if it is present. In formal probability notation, this says $p(X = positive|Y = diabetes) = 0.95$. However, occasionally, very rarely, OGTT gives a false positive. This false positive (also known as false alarm) rate is just 1%, so $p(X = positive|Y = no\text{-}diabetes) = 0.01$.

We see that the test is not perfect, hence may make a mistake. Thus, we would like to know the value of $p(Y = diabetes \mid X = positive)$, that is, the chances of someone having diabetes, given that the test says they have it. Bayes' theorem tells us that,

$$p(Y = diabetes \mid X = positive) = \frac{p(X = positive \mid Y = diabetes)p(Y = diabetes)}{p(X = positive)}$$

We don't have a dataset to estimate the probabilities from, but we have our own domain knowledge. The value of $p(Y = \textit{diabetes})$ is known to be 0.028, and $p(X = \textit{positive} \mid Y = \textit{diabetes})$ is known also. The only part missing from our equation is $p(X = \textit{positive})$. This is the overall probability of the test returning a positive result. If we don't have data, how could we possibly know that? Luckily, there is a way.

We use the following rule from probability theory, called *marginalisation*:

$$p(X = x) = \sum_{y \in Y} p(X = x|Y = y)p(Y = y) \tag{8.3}$$

In words, this says is that *the probability of any given X value is equal to the probability of the value, given every possible situation for the Y variable, weighted by the probability of that Y variable.* This concept may take a while to sink into your brain – don't worry. In practical terms for our diabetes problem, it means this:

$$\begin{aligned} p(X = \textit{positive}) \quad = \quad & p(X = \textit{positive} \mid Y = \textit{diabetes})p(Y = \textit{diabetes}) \\ & + p(X = \textit{positive} \mid Y = \textit{no-diabetes})p(Y = \textit{no-diabetes}) \end{aligned} \tag{8.4}$$

Notice that we need to know $p(X = \textit{positive} \mid Y = \textit{no-diabetes})$. This is the false alarm rate, which we know to be $p(X = \textit{positive}|Y = \textit{no-diabetes}) = 0.01$. Remembering that $p(Y = \textit{no-diabetes}) = 1 - p(Y = \textit{diabetes})$, and plugging in the values for these probabilities, we get:

$$\begin{aligned} p(Y = \textit{diabetes} \mid X = \textit{positive}) \quad & = \quad \frac{0.95 \times 0.028}{(0.95 \times 0.028) + (0.01 \times 0.972)} \\ & = \quad \frac{0.0266}{0.0266 + 0.00972} \\ & \approx \quad 0.73237... \end{aligned}$$

So there's a 73% chance that you really do have diabetes, when the test says you do — it's not a perfect test, but maybe you thought it was more reliable than it actually is.

Take your time to go over this again if you don't get it immediately, before moving onto the next sections.

So, we've just worked out the probability that a person really has diabetes, *given that the test says they do,* or mathematically, $p(Y = diabetes \mid X = positive)$.

$$p(Y = diabetes \mid X = positive) = \frac{p(X{=}positive \mid Y{=}diabetes)p(Y{=}diabetes)}{p(X{=}positive)}$$

Similarly, we can work out the probability of *NOT* having diabetes given that the test says they do:

$$p(Y = no\text{-}diabetes \mid X = positive) = \frac{p(X{=}positive|Y{=}no\text{-}diabetes)p(Y{=}no\text{-}diabetes)}{p(X{=}positive)}$$

...which is just $(1-0.73237) = 0.2676$. Now, we could have worked this out just from knowing that probabilities always sum to one, $\sum_{y\in Y} p(Y = y|X = x) = 1$, but notice something interesting by writing it out explicitly... the *denominator is the same* in both cases. And, from Eq. (8.4), we know that this denominator is the sum of all the numerators for every possible value of Y.

If we give the numerators shorthand names, a, and b:

$$\begin{aligned} a &= p(X = positive \mid Y = no\text{-}diabetes)p(Y = no\text{-}diabetes) \\ b &= p(X = positive \mid Y = diabetes)p(Y = diabetes) \end{aligned}$$

Then we can see simply that:

$$p(Y = diabetes \mid X = positive) = \frac{b}{a+b}$$

$$p(Y = no\text{-}diabetes \mid X = positive) = \frac{a}{a+b}$$

Now imagine we had a more advanced test, that could distinguish between different types of diabetes, i.e. we use the notation: $Y = 0$ means no diabetes, $Y = 1$ means type-1 diabetes, $Y = 2$ means type-2 diabetes...

$$\begin{aligned} a &= p(X = positive \mid Y = 0)p(Y = 0) \\ b &= p(X = positive \mid Y = 1)p(Y = 1) \\ c &= p(X = positive \mid Y = 2)p(Y = 2) \end{aligned}$$

and applying the same principle as above...

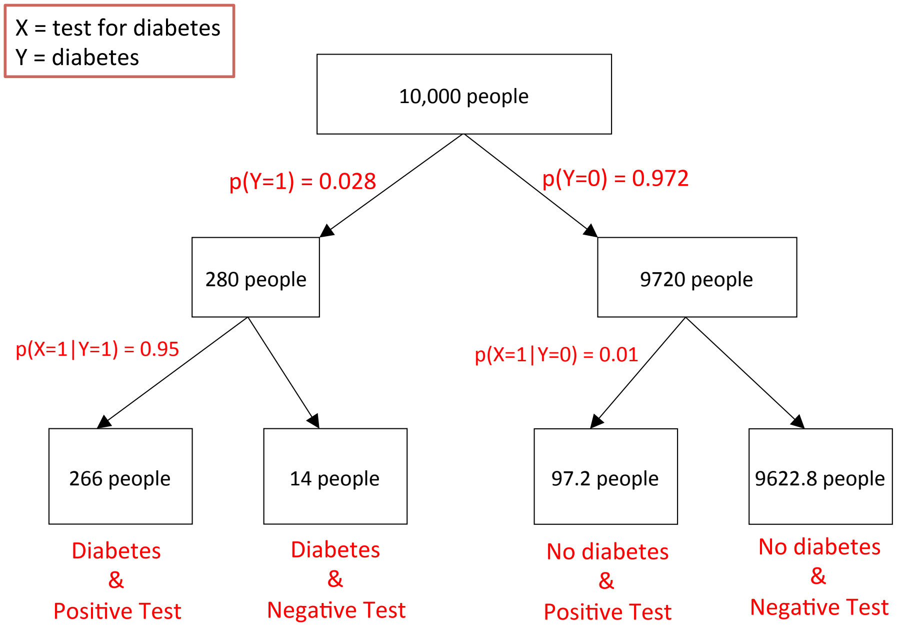
$$p(Y = 1 \mid X = positive) = \frac{b}{a+b+c}$$

...and so on. Again, take your time to go over this – it's tough and quite non-intuitive the first time you meet it.

8.2.3 Thinking about it another way

This section contains a more intuitive, less mathematical way of thinking about Bayes' Theorem. You should use this section as a way to clarify your understanding of the previous section. You should *not* just rely on the approach in this section to do the calculations.

Let's imagine 10,000 randomly chosen people worldwide to test for diabetes. Some of those (2.8%) will have diabetes. When we administer the test, some of each group (those with diabetes, and some of those without) will get a positive result. This is illustrated below.



So if I ask you again, what is $p(Y = 1|X = 1)$, or in other words, what are the chances of diabetes *really* being present, given that the test said it was? Logically from the diagram, you should be able to see that we just take the number of cases where the test was positive $(266 + 97.2)$, and see what fraction of those *really* had diabetes:

$$\frac{266}{266{+}97.2} = \frac{266}{363.2} = 0.73237...$$

which is 73%, exactly the same answer we got by going through Bayes' Theorem.

8.2.4 One more Bayes example: Catching Bad Guys

The US government does a lot of electronic surveillance, trying to catch people threatening terrorist action. The general incidence of terrorism in the population is thankfully quite rare — let's say 1 in a million. So, the NSA (National Security Agency) has a test, to process someone's email automatically, and say if they are suspicious or not.

Let's assume the people at the NSA are quite smart (if a bit scary) so their email analysis somehow correctly identifies terrorist emails 99% of the time. Sometimes, the system makes a mistake and thinks an innocent person is a terrorist – let's say this is 2% of the time. If X is the output of the system, and Y is the true identity of a person (terrorist or not) then we want to calculate the value of $p(Y=1|X=1)$, the chances of them really being a terrorist if the system says they are.

We know that $p(Y=1)=\frac{1}{1{,}000{,}000}$. We also know that $p(X=1|Y=1)=0.99$, and that $p(X=1|Y=0)=0.02$. So, applying Bayes' Theorem we get:

$$\begin{aligned} p(Y=1|X=1) &= \frac{0.99\times\frac{1}{1{,}000{,}000}}{(0.99\times\frac{1}{1{,}000{,}000})+(0.02\times\frac{999{,}999}{1{,}000{,}000})} \\ &= \frac{0.0000009}{0.0000009+0.01999998}\approx 0.0000495... \end{aligned}$$

This is the probability that the NSA has found a REAL bad guy, when their system says they have found one. In every day language, this is a chance of about 1 in $20,000$, that is, for every $20,000$ suspects that the system says "that's a bad guy", only 1 of them will actually be one. You may have thought their systems were quite good with 99% accuracy in finding bad guys, but here you see the true difficulty of the problem they face.

If you want to verify these numbers yourself, you may want to write some code (or just work in fractions) since these numbers are quite small – a standard calculator may lose some precision when you do the divisions.

8.3 Naive Bayes Models

You can use Bayes' Theorem to make predictions as usual on training/testing data. Imagine we had this data:

X_1	X_2	X_3	Y
1	1	0	1
1	0	0	1
0	0	1	1
0	0	0	1
1	0	1	0
0	1	0	0

Table 8.3: Some sample data.

Here we have 3 arbitrarily named features, X_1, X_2 and X_3. I hope you can work out from this data that $\hat{p}(Y = 1|X_1 = 1) = \frac{2}{3}$. **If you cannot do this, go back to the previous section as it's only going to get tougher.**

SELF-TEST
For extra practice – calculate $\hat{p}(Y = 1|X_2 = 0)$, from the data above. Then calculate $\hat{p}(Y = 1, X_2 = 0)$, and $\hat{p}(X_2 = 0)$, and state how these three probabilities relate to each other.

But what if we get a new *testing* example that we've never seen: $\mathbf{x} = \{1, 1, 1\}$? That is, where $X_1 = 1$, $X_2 = 1$ and $X_3 = 1$. We could apply Bayes' Theorem, but to use it, we need to calculate the numerator for when $Y = 1$:

$$p(Y = 1|X_1 = 1, X_2 = 1, X_3 = 1) \propto p(X_1 = 1, X_2 = 1, X_3 = 1|Y = 1)p(Y = 1)$$

where the symbol $\propto$ means "*proportional to*". We know that $p(Y = 1) = \frac{4}{6}$, but what about the right hand side term, $p(X_1 = 1, X_2 = 1, X_3 = 1|Y = 1)$?

To get around this, we make the *conditional independence assumption*:

$$p(x_1, x_2, x_3|y) = p(x_1|y)p(x_2|y)p(x_3|y)$$

or in more general terms:

$$p(\mathbf{x}|y) = \prod_i p(x_i|y)$$

Mathematically, this means *the features are conditionally independent of each other, given the class value.* An intuitive explanation is much harder, but it can be visualised, as in Figure 8.2. This is called the "Naive Bayes" model.

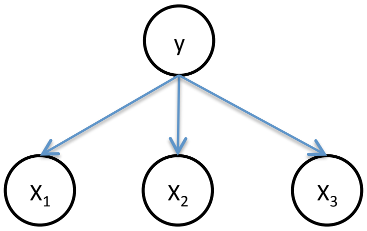


Figure 8.2: *The* graphical representation *of Naive Bayes. The fundamental assumption of Naive Bayes is conditional independence, or* $p(\mathbf{x}|y) = \prod_i p(x_i|y)$. *You should hopefully be able to see how the graph corresponds to this.*

You should not interpret the word "Naive" as to indicate that Naive Bayes is a 'stupid' model. Naive Bayes often performs extremely well, and there is a lot of theoretical analysis[3] trying to figure out why it can work so well, in spite of such a seemingly naive assumption.

So, given the training data in the Table 8.3, what probability will a Naive Bayes classifier predict for $p(Y = 1|\mathbf{x})$, given input $\mathbf{x} = \{1, 1, 1\}^T$?

First estimate necessary probabilities for $Y = 1$,

$$\hat{p}(X_1 = 1|Y = 1) = \tfrac{2}{4}$$

$$\hat{p}(X_2 = 1|Y = 1) = \hat{p}(X_3 = 1|Y = 1) = \tfrac{1}{4}$$

and for $Y = 0$,

$$\hat{p}(X_1 = 1|Y = 0) = \hat{p}(X_2 = 1|Y = 0) = \hat{p}(X_3 = 1|Y = 0) = \tfrac{1}{2}$$

The prior is $\hat{p}(Y = 1) = \frac{4}{6}$, and we know $p(Y = 0) = 1 - p(Y = 1)$. So finally, we have :

$$\hat{p}(Y = 1|\mathbf{x}) \propto \textstyle\prod_i \hat{p}(X_i = 1|Y = 1) \times \hat{p}(Y = 1) = \tfrac{2}{4} \times \tfrac{1}{4} \times \tfrac{1}{4} \times \tfrac{4}{6} = \tfrac{4}{192}$$

$$\hat{p}(Y = 0|\mathbf{x}) \propto \textstyle\prod_i \hat{p}(X_i = 1|Y = 0) \times \hat{p}(Y = 0) = \tfrac{1}{2} \times \tfrac{1}{2} \times \tfrac{1}{2} \times \tfrac{2}{6} = \tfrac{2}{48}$$

And finally, we see that using Naive Bayes, when $\mathbf{x} = \{1, 1, 1\}$, we predict

$$\hat{p}(Y = 1|\mathbf{x}) = \frac{\frac{4}{192}}{\frac{4}{192} + \frac{2}{48}} = \frac{1}{3}$$

which is something we could not have done if we had just used Bayes' Theorem without the conditional independence assumption.

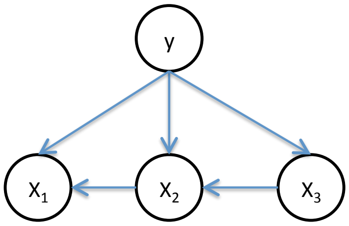
[3] Domingos and Pazzani *"On the Optimality of the Simple Bayesian Classifier under Zero-One Loss", Machine Learning Journal, vol 29, pages 103–130 (1997)*

8.4 Bayesian Networks

You saw in Figure 8.2 the Naive Bayes model, visualised as a graph. This is in fact a special case of a wider class of models, called “Bayesian Networks”. A Bayesian Network is a *directed acyclic graph*[4], that represents a probability distribution. The distribution corresponding to the network shown in Figure 8.2 is:

$$p(x_1, x_2, x_3, y) \;=\; p(y)p(x_1|y)p(x_2|y)p(x_3|y)$$

You should be able to see how the two relate. Another example is this:



The distribution corresponding to this network is:

$$p(x_1, x_2, x_3, y) \;=\; p(y)p(x_1|y, x_2)p(x_2|y, x_3)p(x_3|y)$$

The probabilities can be learnt in exactly the same way, by counting frequencies of occurrence in the data. However, this particular example highlights a problem... for an input $\mathbf{x} = \{1, 1, 1\}$, we estimate probabilities from the data in Table 8.3:

$$\hat{p}(Y = 1) = \tfrac{4}{6}$$
$$\hat{p}(X_1 = 1|Y = 1, X_2 = 1) = 1$$
$$\hat{p}(X_2 = 1|Y = 1, X_3 = 1) = 0$$
$$\hat{p}(X_3 = 1|Y = 1) = \tfrac{1}{4}$$

If we apply Bayes’ theorem (I will leave it as an exercise for you to do this) we find that $\hat{p}(Y = 1|\mathbf{x}) = 0$, and also $\hat{p}(Y = 0|\mathbf{x}) = 0$, that is, neither class label has any probability of happening! This can obviously not be true, and this problem is a symptom of not having enough data — the dataset in the table is too small to allow us to estimate the probabilities properly.

The more links in a Bayesian Network, the more complicated the probability distribution, and hence the more data is needed. The links correspond to the over/underfitting of the model — like a decision tree being

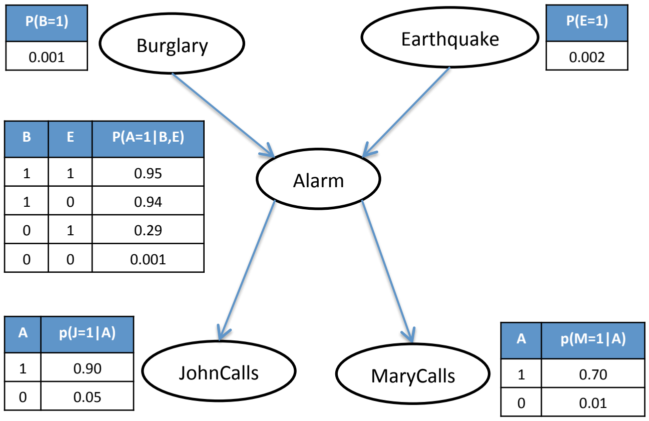
[4] i.e. there cannot be loops in the network.

too deep, this model is too complex for our small data. As a result, it overfits, believing the zero probabilities that it observes in the very small training dataset.

One of the great advantages of Bayesian Networks is that you can very naturally encode human knowledge about a given problem.

Imagine you travel a lot for work, and that your house has a burglar alarm. When it goes off, one of your two neighbours (John and Mary) might phone you to tell you, or maybe both will phone. So, when you see John, or Mary, come up on your phone, you could *infer* (predict) that the alarm has gone off[5]. However, the *cause* of the alarm sounding could be one of two things — either a burglar is in your house, or possibly... an earthquake!

This story is encoded by the following network:



P(B=1)
0.001

P(E=1)
0.002

B	E	P(A=1|B,E)
1	1	0.95
1	0	0.94
0	1	0.29
0	0	0.001

A	p(J=1|A)
1	0.90
0	0.05

A	p(M=1|A)
1	0.70
0	0.01

Here, you can see clearly that in a Bayesian Network, there is a probability distribution stored at each network node. The probability of an earthquake is $p(E = 1) = 0.002$, and the probability of a burglar is $p(B = 1) = 0.001$. The probability of the alarm, when there is a burglar but no earthquake, is $p(A = 1|B = 1, E = 0) = 0.94$. You can read the other probabilities off yourself.

Remember, every network encodes a probability distribution – this one is:

$$p(B, E, A, J, M) \quad = \quad p(J|A)p(M|A)p(A|B, E)p(B)p(E)$$

The clever thing about Bayesian Network models, is that we can use it to make predictions about *any* of the variables. We don't have to designate one of them as a special class variable Y. We can get a prediction for $p(E = 1|J = 1)$,

[5] Let's assume they won't call for any other reason as they know you're busy.

i.e. the probability of an earthquake given that John calls. Or we can get a prediction for $p(B=1|M=1,J=1,E=1)$, i.e. the probability of a burglar being in your house, given that both friends call, and an earthquake occurs!

This last one may seem strange, but we can in fact make a prediction for *any* variable, given the states of any other (sub)set of other variables. These predictions are performed using Bayes' Theorem, combined with the *marginalisation* trick from Eq. (8.3), in a more sophisticated way. If we want to predict the probability of there being a burglar in your house, given that Mary calls, we need to calculate:

$$p(B=1|M=1) \;\propto\; \sum_{j\in J}\sum_{e\in E}\sum_{a\in A} p(j|a)p(M=1|a)p(a|B=1,e)p(B=1)p(e)$$

However, to calculate this, we need three nested for-loops, one for each of the J, E, and A variables. As such this becomes quite computationally intensive, and even more so with bigger networks. There are efficient algorithms, outside the scope of this course unit, to tackle this – these are the *Variable Elimination* algorithm, and the *Belief Propagation* algorithm. Given the complexity of these algorithms, these are more suited to a full course on probabilistic models, rather than a short course like this one.

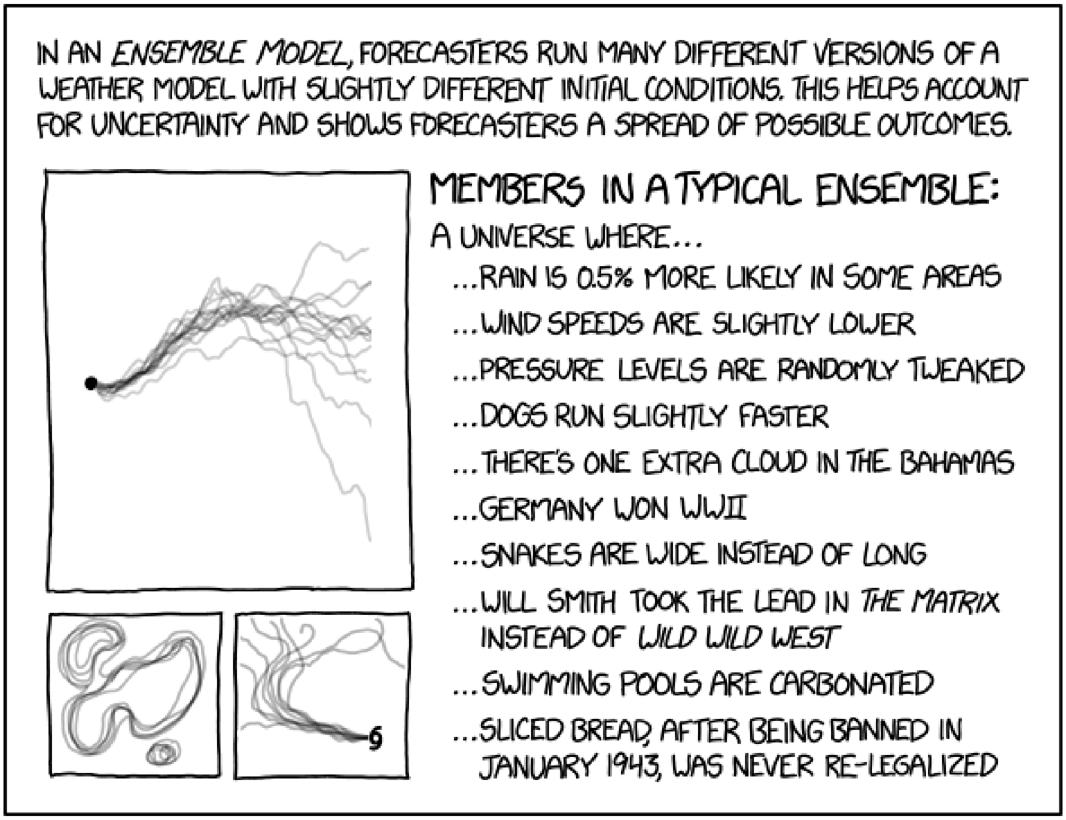
8.5 What you should know by now

You should be able to:

- calculate simple conditional and joint probabilities from a dataset;
- answer questions about a dataset by applying Bayes' Theorem;
- manually calculate what a Naive Bayes would do for a given small dataset;
- draw the network corresponding to a given distribution, and vice versa;
- understand why small datasets can cause problems for complex networks.

Chapter 9

Committees of Models
(also known as 'Ensemble' Methods)


IN AN *ENSEMBLE MODEL*, FORECASTERS RUN MANY DIFFERENT VERSIONS OF A WEATHER MODEL WITH SLIGHTLY DIFFERENT INITIAL CONDITIONS. THIS HELPS ACCOUNT FOR UNCERTAINTY AND SHOWS FORECASTERS A SPREAD OF POSSIBLE OUTCOMES.
MEMBERS IN A TYPICAL ENSEMBLE:
A UNIVERSE WHERE...
...RAIN IS 0.5% MORE LIKELY IN SOME AREAS
...WIND SPEEDS ARE SLIGHTLY LOWER
...PRESSURE LEVELS ARE RANDOMLY TWEAKED
...DOGS RUN SLIGHTLY FASTER
...THERE'S ONE EXTRA CLOUD IN THE BAHAMAS
...GERMANY WON WWII
...SNAKES ARE WIDE INSTEAD OF LONG
...WILL SMITH TOOK THE LEAD IN *THE MATRIX* INSTEAD OF *WILD WILD WEST*
...SWIMMING POOLS ARE CARBONATED
...SLICED BREAD, AFTER BEING BANNED IN JANUARY 1943, WAS NEVER RE-LEGALIZED

9.1 Background

Ensemble Learning refers to the procedures employed to train multiple models and combine their outputs, treating them as a "committee" of decision makers. The principle is that the committee decision, with individual predictions combined appropriately, should have better overall accuracy, on average, than any individual committee member. Numerous empirical and theoretical studies have demonstrated that ensemble models very often attain higher accuracy than single models. The members of the ensemble might be predicting class labels, or posterior probabilities. Therefore, their decisions can be combined by many methods, including averaging, voting, and probabilistic methods. The majority of ensemble learning methods are generic, applicable across broad classes of model types and learning tasks.

The trick that ensemble algorithms exploit is the following: when a single model cannot properly fit the data, the ensemble can make multiple versions that each make errors in different ways—then vote or average their predictions, cancelling out the errors of the individuals by the committee decision.

9.2 Bagging and Random Forests

The Bagging algorithm generates differences between the models by feeding them slightly different training sets. It does this with a data sampling technique called a *bootstrap*. Given a dataset T, of size N, we randomly select N examples, with replacement. The *with replacement* part is very important. It means that we randomly pick one example to be in our training set, then put it back, ensuring that it could be picked *again*. An example dataset and two bootstrap samples are shown below.

BOOTSTRAP

x_1	x_2	x_3	x_4	x_5	y
187	80	120	30	4.5	0
160	70	119	36	5.6	0
150	80	185	60	8.8	1
192	92	140	50	6.8	1
168	110	155	45	7.8	1

187	80	120	30	4.5	0
150	80	185	60	8.8	1
150	80	185	60	8.8	1
168	110	155	45	7.8	1
168	110	155	45	7.8	1

160	70	119	36	5.6	0
160	70	119	36	5.6	0
150	80	185	60	8.8	1
192	92	140	50	6.8	1
168	110	155	45	7.8	1

This shows a dataset (left) and two bootstrap samples taken from it (right). Notice that the first bootstrap (top right) contains **2** copies of the third and fifth examples, but none of the second or fourth. The second bootstrap (bottom right) is generated by following the same randomised procedure, but results in a different training set—with **2** copies of the second example, and one copy each of the third, fourth and fifth, but no copies of the first example. These random differences between training sets are exploited to build different models, in the **Bagging** algorithm:

Bagging (input training data+labels T, number of models M)

```
for $j = 1$ to $M$ do
    Take a bootstrap sample $T'$ from $T$
    Build a model using $T'$.
    Add the model to the set.
end for
return set of models
```

For a test point $\mathbf{x}$, get a response from each model, and take a majority vote.

Remember I said there were only two ML algorithms that I'd be willing to bet my life on? SVMs were one. The other is called "Random Forests", and if I really had to just go with **one**... Random Forests are it. RF is an extension of Bagging, but only for *decision trees*, hence the name—a forest of randomized trees. The randomisation is generated via *two* mechanisms: a **bootstrap**, (as in Bagging), and a **random selection of features** at each split point. The algorithm is:

Random Forests (input training data+labels T, number of trees M)

```
for $j = 1$ to $M$ do
    Take a bootstrap sample $T'$ from $T$
    Build a decision tree using $T'$, but, at every split point:
        - Choose a random fraction $K$ of the remaining features,
        - Pick the best feature (minimising cost) from that subset.
    Add the tree to the set, without pruning
end for
return set of trees
```

For a test point $\mathbf{x}$, get a response from each tree, and take a majority vote.

We see that the trees are effectively forced to not choose the best feature at every split, but in a random way. The result is that the trees will all be a little bit different, but still quite accurate. The majority vote ensures that the little drop in accuracy for each tree doesn't matter too much, and the differences between them ensures that they don't make simultaneous mistakes. The lack of pruning ensures that we don't force them to be too simple, otherwise, they might become too similar to each other.

MICROSOFT KINECT

The final result is a *very* strong classifier. Random Forests are in fact the basis of how the popular Microsoft Kinect controller for Xbox works, when it identifies where your body parts are for tracking.

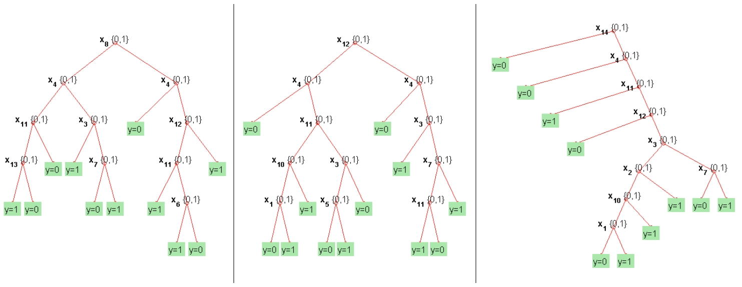


Figure 9.1: Typical result of RF: the trees are diverse and accurate.

9.3 Adaboost

Adaboost is the most well known of the *Boosting* family of algorithms. The algorithm trains models sequentially, with a new model trained at each round. At the end of each round, mis-classified examples are identified and have their emphasis increased in a new training set which is then fed back into the start of the next round, and a new model is trained. The idea is that subsequent models should be able to compensate for errors made by earlier models.

Adaboost occupies somewhat of a special place in the history of ensemble methods. Though the procedure seems heuristic, the algorithm is in fact grounded in a rich learning-theoretic body of literature. Robert Schapire addressed a question on the nature of two complexity classes of learning problems. The two classes are *strongly learnable* and *weakly learnable* problems. Schapire showed that these classes were equivalent; this had the corollary that a weak model, performing only slightly better than random guessing, could be "boosted" into an arbitrarily accurate *strong* model. The original Boosting algorithm was a proof by construction of this equivalence, though had a number of impractical assumptions built-in. The Adaboost algorithm was the first practical Boosting method. The procedure is shown below. Some similarities with Bagging are evident; a key difference is that at each round t, Bagging has a uniform distribution D_t, while Adaboost adapts a non-uniform distribution, and while Bagging effectively trains models in parallel, Adaboost builds models sequentially.

```
**Adaboost** (input training data+labels $T$, number of models $M$)

**Input:** Training set $T = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_N, y_N)\}$, where $y_i \in \{-1, +1\}$

Define a uniform distribution $D_1(i)$ over elements of $T$.

for $j = 1$ to M do
    Train a model $h_j$ using a dataset sampled from $T$ using distribution $D_j$.
    Calculate $\epsilon_j = \sum_{i \in T} \delta(h_j(\mathbf{x}_i) \neq y_i)$
    If $\epsilon_j \geq 0.5$ break
    Set $\alpha_j = \frac{1}{2} \ln\left(\frac{1-\epsilon_j}{\epsilon_j}\right)$
    Update $D_{j+1}(i) = \frac{D_j(i) \exp(-\alpha_j y_i h_j(\mathbf{x}_i))}{Z_j}$
    where $Z_j$ is a normalization factor so that $D_{j+1}$ is a valid distribution.
end for

For a new testing point $(\mathbf{x}', y')$, we take a weighted majority vote, which can
be implemented as, $H(x') = sign\Big(\sum_{j=1}^{M} \alpha_j h_j(\mathbf{x}')\Big)$
```

The ensemble is constructed by iteratively adding models. Each time a model is learnt, it is checked to ensure it has at least $\epsilon_j < 0.5$, that is, it has performance *better than random guessing* on the data it was supplied with. If it does not, either an alternative model is constructed, or the loop is terminated. After each round, the distribution D_j is updated to emphasize incorrectly classified examples. The update causes half the distribution mass of D_{j+1} to be over the examples incorrectly classified by the previous model. More precisely, $\sum_{h_j(\mathbf{x}_i)\neq y_i} D_{j+1}(i) = 0.5$. Thus, if h_j has an error rate of 10%, then examples from that small 10% will be allocated 50% of the next model's training 'effort', while the remaining examples (those correctly classified) are underemphasized. An equivalent (and simpler) writing of the distribution update scheme is to multiply $D_j(i)$ by $\frac{1}{2(1-\epsilon_j)}$ if $h_j(x_i)$ is correct, and by $\frac{1}{2\epsilon_j}$ otherwise.

The updates cause the models to sequentially minimize an exponential bound on the error rate. The training error rate on a data sample T drawn from the true distribution $\mathcal{T}$ obeys the bound,

$$P_{\mathbf{x},y\sim\mathcal{T}}(yH(\mathbf{x}) < 0) \leq \prod_{j=1}^{M} 2\sqrt{\epsilon_j(1-\epsilon_j)}. \tag{9.1}$$

This *upper bound* on the training error (though not the *actual* training error) is guaranteed to decrease monotonically with M, given $\epsilon_j < 0.5$.

In an attempt to further explain the performance of Boosting algorithms, Schapire also developed bounds on the *generalization* error of voting systems, in terms of the voting margin. Note that this is not the same as the *geometric margin*, optimized by Support Vector Machines. The difference is that the voting margin is defined using the one-norm $||\mathbf{w}||_1$ in the denominator, while the geometric margin uses the *two*-norm $||\mathbf{w}||_2$. While this is a subtle difference, it is an important one, forming links between SVMs and Boosting algorithms.The following bound holds with probability $1-\delta$,

$$P_{\mathbf{x},y\sim\mathcal{T}}(H(\mathbf{x}) \neq y) \leq P_{\mathbf{x},y\sim\mathcal{T}}(yH(\mathbf{x}) < \theta) + \tilde{O}\Big(\sqrt{\frac{d}{N\theta^2} - \ln\delta}\Big), \tag{9.2}$$

where the $\tilde{O}$ notation hides constants and logarithmic terms, and d is the *VC-dimension* of the model used. Roughly speaking, this states that the generalization error is less than or equal to the training error plus a term dependent on the voting margin. The larger the minimum margin in the training data, the lower the testing error.

The margin-based theory is only one explanation of the success of Boosting algorithms. Mease & Wyner present a discussion of several questions on why and how Adaboost succeeds. The subsequent 70 pages of discussion demonstrate that the story is by no means simple. The conclusion is, while no single theory can fully explain Boosting, each provides a different part of the still unfolding story.

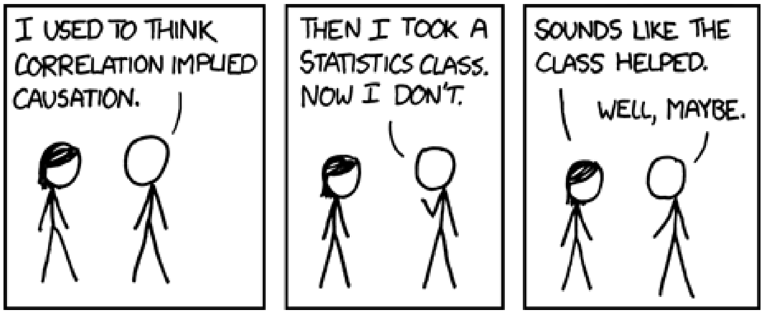
9.4 What you should know by now

You should know why ensemble methods might be useful. You should know key differences between the three algorithms described in this chapter, and be able to fully explain the algorithm, including technical terms like "bootstrap". Ideally, you should have at least attempted to implement these – they may provide good possibilities for your mini-projects.

To fully understand this chapter, it is highly likely that you will need to do some extra reading, outside of just these notes. I can recommend work by Robi Polikar, Tom Dietterich, and Ludmila Kuncheva, as great researchers in the ensemble methods community.

Chapter 10

Feature Selection


I USED TO THINK CORRELATION IMPLIED CAUSATION.
THEN I TOOK A STATISTICS CLASS. NOW I DON'T.
SOUNDS LIKE THE CLASS HELPED.
WELL, MAYBE.

10.1 Background

Imagine you're trying to build a model to predict the weather tomorrow. I think you'd agree that it would be pretty silly to provide it with a feature based on the colour of my shoes. You know this because you know what is meant by the concepts of colour and weather. But what does the HOXa9 gene in the human genome mean? Is it useful in predicting a genetic predisposition to lung cancer? Or does it control hair colour? We don't know. This section is all about how to figure out when a feature, such as the HOXa9, is *useful* for a learning algorithm, and when it is *irrelevant*. More precisely, this section is about how to find the subset of features that will help you, and how to avoid those that will only hinder your learning algorithm.

Why?

Datasets with a large number of features are a significant challenge for Machine Learning. Some of the most practically relevant and high-impact applications, such as *gene expression* data, may easily have more than $10,000$ features. Many of these features may be completely *irrelevant* to the task at hand, or *redundant* in the context of others. Learning in this situation raises important issues, e.g. overfitting to irrelevant aspects of the data, and the computational burden of processing many similar features that provide redundant information. It is therefore an important research direction to automatically identify meaningful smaller subsets of these variables, i.e. *feature selection.*

In the paragraph above, I pointed out some fairly obvious issues, for example—in some problems, certain data will be completely irrelevant. Something not so obvious is the following. The best set of features to give to a Support Vector Machine is **not necessarily** the best set to give to a K-nn classifier. Why? Simply because they draw different types of boundaries, and allow different degrees of flexibility when you change their parameters. The same applies to all learning algorithms. A set of features should *ideally* be chosen specifically for the model; I say ideally because this will not always be possible, due to the computational burden of finding the right subset of size 10 (for example) from a $1,000,000$ feature dataset! We must remember that even with the tennis data, when we had just $d = 4$ features, there are $2^d = 2^4 = 16$ possible feature sets! With a few thousand features, the number of possible combinations is more than the age of the universe!

10.2 The ‘Wrapper’ Method

With $d = 4$ features, there are $2^d = 16$ possible feature sets that we could give to our model to learn from. This is visualised below, showing the 16 possible feature set choices laid out as a search space.

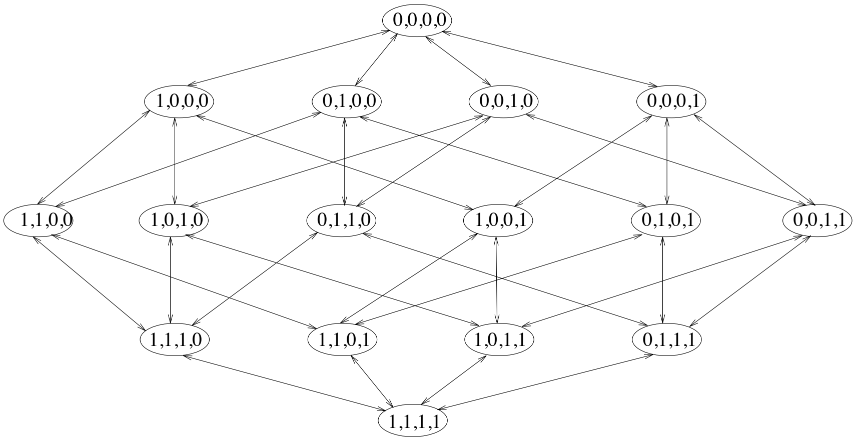


Figure 10.1: Visualising the search space when we have just 4 features.

The task is to find the right set of features that will give lowest test error with the model we have chosen, which we could say for example is an SVM, or a logistic regression, whatever you like. The challenge is therefore a *combinatorial optimisation*, finding the right point in a binary search space. The *wrapper* method for feature selection is the most obvious, and is simply as follows.

Wrapper Method for Feature Selection

1. Start with an initial guess for a good set of features
2. Train and test a model (maybe via cross validation)
3. If your test error is deemed good enough, STOP
4. otherwise, choose a new set of features, and go to line 2.

This is called a ‘wrapper’ for obvious reasons — the search procedure is simply ‘wrapped’ around a model that is trained/tested at step 2. The search procedure itself can take on any form we wish. We could use a *greedy search algorithm*, or just a *genetic algorithm* search, *simulated annealing*, or *branch and bound algorithms*, or many, many others.

The simplest is called a Wrapper **Forward Selection**, as we add features greedily and sequentially, i.e. “moving forward”. At each step, we see how evaluate our model (via cross validation) with the current features, plus each

FORWARD SELECTION

of the remaining ones. We then find which of the remaining ones improves our model the most, and add it permanently to our set. At the next step, we repeat this, looking at whatever features remain.

An alternative would be the reverse process, starting with all features, sequentially discarding them. There, we evaluate our model with the current set, but with one feature removed. Then we sequentially evaluate it with each feature in turn removed, and discard the one that 'damages' performance the least. This is called **Backward Elimination**.

BACKWARD ELIMINATION

Clearly, both of these algorithms are suboptimal—they are not guaranteed to find the optimal subset. Remember of course that finding the optimal subset is a HUGE search problem, and in practice, these two algorithms do perform very well. An alternative, that has been found to find even better subsets, is **Stepwise (or "Floating") Selection**, where we move two steps forward, then one backward, then two more forward, etc. The number of steps clearly is another parameter of the algorithm that needs to be tuned.

Despite these search techniques performing very well in practice, they are HUGELY computationally expensive. At every single step, we need to run the entire learning algorithm over again with a new feature subset. Another disadvantage is that, as mentioned before, a subset chosen for an SVM will not necessarily work well for a KNN or any other algorithm. A Wrapper selection is clearly *specific* to a learning algorithm (whichever one it is wrapped around), so feature subsets cannot be *reused* again later.

A question is therefore, how can we select good subsets, (1) while avoiding running our learning algorithm at every step?, and (2) that will be generically useful across several learning algorithms? The answer to this is *filter* methods.

10.3 Filter Methods

Let's say we were trying to predict someone's Biology exam grade from various possible indicators. I will provide you with their *Chemistry grade, History grade, Biology Mock exam grade,* and *Height.* These are shown in the figures below, plotted against their *actual Biology exam grade.* Which indicator (feature) would you pick?

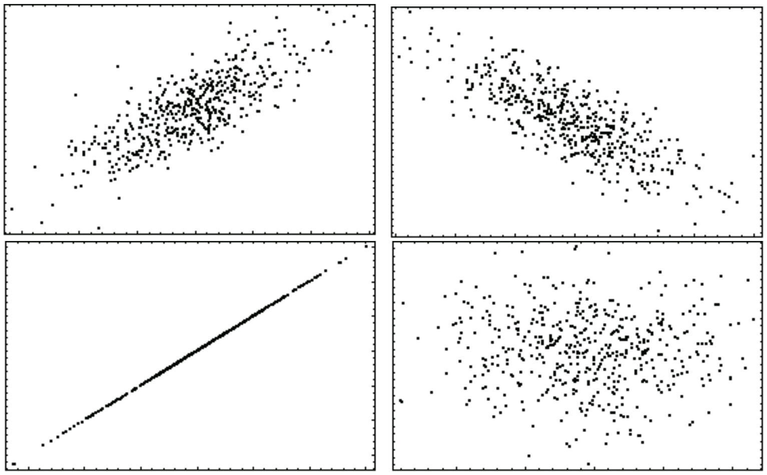


Figure 10.2: *For a class full of 200 students, these plots show the performance on the Biology exam (y-axis), plotted against various indicators of exam performance. Top left: Chemistry exam. Top right: History exam. Bottom left: Mock Biology exam. Bottom right: Height in centimetres*

.

This is again an example of where a human judgement can *easily* pick out the information we need, you would pick the bottom left feature, the mock exam. However, remember we are trying to get a computer program that will automate all of this. How can we *measure* the usefulness of the indicators above? We can measure Pearson's *correlation coefficient* of each plot.

$$r = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{N}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{N}(y_i - \bar{y})^2}} = \frac{cov(x,y)}{std(x)std(y)} \qquad (10.1)$$

Here, the x_i are all the values of the chemistry grades (for example), and y_i is the biology grades. The correlation coefficient is the covariance of the two variables, divided by the product of their standard deviations. The denominator serves to normalise the value between -1 and +1, and overall the r value summarises the degree of *linear correlation* between two variables. An important point to note is that **both** strong positive and strong negative correlation are useful. This may seem quite strange, but it is common to have negative correlations in nature—for example if we plotted the biology grade against the number of hours of TV watched each week, the more hours of TV, the lower your grade!

> *In Python you can get this from NumPy with the function* `r = np.corrcoef(x,y)`*. Calling this returns a 2x2 matrix. Look at some online help to understand more what the entries mean.*

Ranking Features

From the examinations example above, if you had to *rank* the indicators in terms of how useful they would be, you might say the order was: biology mock exam, then chemistry, then history, then height. The height is the least useful feature. The same principle can be applied by our learning algorithms to select features. We *rank the features* in order of the absolute[1] value of the correlation coefficient.

Below we see an example of this. We are using the `heart` dataset. Here for simplicity, we apply a K nearest neighbour with $k = 1$. We use the features in the order they appear in the file, at each step evaluating a knn classifier. The first knn uses feature 1, the second uses features 1 and 2, the third knn uses features 1, 2, and 3, and so on. This is on the left of the figure below. In contrast to this we can *rank the features*, by their $|r|$ value:. We get the ranking: $13, 12, 8, 3, 11, 2, 10, 9, 7, 4, 5$.

The best performance on the left is with features 10 features. The best performance on the right is with 7 features (and 5 is very close too). We have achieved a lower classification error, with less features (so less computational cost). So we improve on both performance and time/space complexity!

The interesting point here is that **we did not use the KNN to select the features**. We selected them purely on their *intrinsic* relatedness to the target label. These features could now be equally used with a SVM, or any other algorithm.

Disadvantage of Ranking by Correlation Coefficient

Pearson's Correlation Coefficient measures *linear correlation.* As such it cannot detect *nonlinear patterns.* In the example scatterplots below, there are clearly observable patterns and relationships between the variables (to the human eye),

[1] Self-test: why do we use the absolute value, not the raw value?

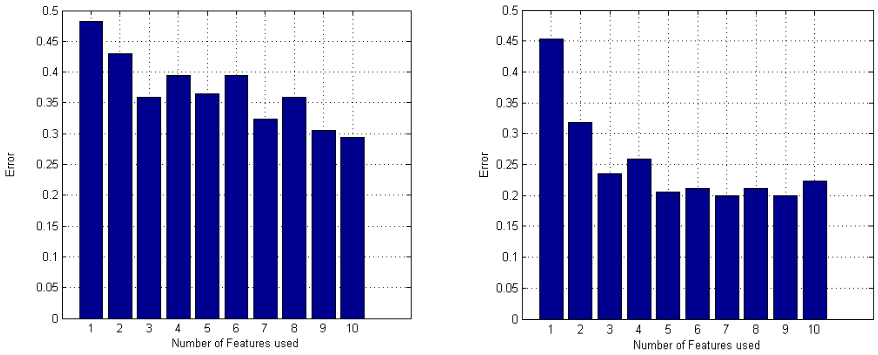


Figure 10.3: *LEFT: Using features in the order they appear in the file. At each step evaluated a KNN with $k = 1$. RIGHT: Using the top features ranked in order of how strongly correlated they are with the label.*

but the r value comes out as 0.0. Therefore features that could be relevant might be ranked very low.

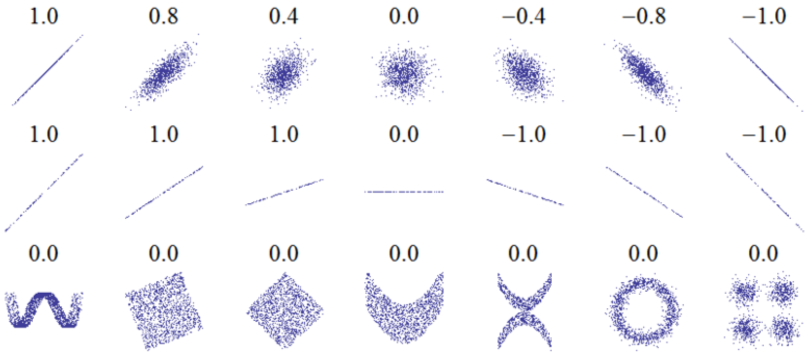


Figure 10.4: Examples of r values for different correlation patterns

Ranking Features by how much "Information" they contain

In the material on Decision Trees lecture, we saw how to quantify what we could gain in terms of predicting the target label, by examining each feature. That is, what is the reduction in uncertainty (entropy) of the label Y, when we examine feature X? The *information gain* quantifies this for us.

$$I(X;Y) \quad = \quad H(Y) - H(Y|X) \qquad (10.2)$$

Another name for the information gain is the *Mutual Information* between X and Y.

When we were measuring the information gain of each feature to decide which we should split the data on, we were really measuring the *mutual information* between the feature and the target label. This principle can be applied in general—we can measure the mutual information of each feature with the label, and *rank* the features by this, instead of the linear correlation coefficient.

In Python you can get this from the skLearn toolkit
`sklearn.metrics.mutual_info_score(x,y)`.
Calling this returns a real valued number, the mutual information.

Major advantage : detects nonlinearities. The MI does not assume any underlying distribution of the data, it can detect arbitrary patterns in the data.

Major disadvantage: May choose redundant features.
Including very similar features (those that provide almost identical information) can be detrimental to a learning algorithm.

We should ideally choose *complementary* features. **This disadvantage is also shared by Pearson's correlation coefficient**. In fact it is a problem for **most** feature selection techniques. Advanced techniques do exist to tackle this—see Fleuret's paper in the conclusion, but this is outside the scope of this course.

10.4 Optional: Information Theory

Mutual Information comes from the field of *Information Theory*, which has its own set of rules for manipulating expresssions. This allows us to rewrite the MI in several ways:

$$\begin{aligned} I(X;Y) &= H(Y) - H(Y|X) \\ &= H(X) - H(X|Y) \\ &= H(X) + H(Y) - H(XY) \\ &= \sum_{x \in X} \sum_{y \in Y} p(x,y) log\Big(\frac{p(x,y)}{p(x)p(y)}\Big) \end{aligned} \tag{10.3}$$

Note that the first two demonstrate the MI measure is symmetric. The final way is the Kullback-Leibler divergence between the joint distribution of the variables, and the product of the marginal distributions. The more similar X and Y, the larger this value. It takes on the value zero when $p(x,y) = p(x)p(y)$, i.e. X and Y are independent. It is maximal when $X = Y$. In general, $0 \leq I(X;Y) \leq min(H(X), H(Y))$.

For further reading, consult the links provided on the course website, in particular take note of the following:

- *An Introduction to Variable and Feature Selection*,
  Guyon & Elisseeff, Journal of Machine Learning Research, vol 3, 2003.
- *Fast Binary Feature Selection with Conditional Mutual Information*,
  F. Fleuret, Journal of Machine Learning Research, vol 5, 2004.
- **Many videos on www.Videolectures.net**

10.5 What you should know by now

You should know why feature selection might be necessary - stating some potential advantages. You should know the difference between wrappers and filters, be able to give at least one example of each, advantages/disadvantages, and understand the search space complexity of using each approach. You should know how mutual information plays a role in feature selection, and it's potential advantages/disadvantages.

Chapter 11

Conclusions

Well, that's it. You are now armed with the powerful basic tools and techniques to apply machine learning algorithms to problems you might encounter. But remember, with great power comes great responsibility – many naive people you encounter out in the world will think ML is basically black magic. It's now your responsibility to educate them.

We just covered the basics of *supervised learning*, There are many many variants, just two examples are:

Semi-supervised Learning : Imagine being provided with a training dataset of 1000 examples, but 500 of the labels are missing. So you have 500 *labelled* examples, and 500 *unlabelled*. The challenge here is how to make use of both resources to trained a good model, such that when it is deployed for real in the world, and tested on new data, it will outperform a model that was trained just using the labeled data.

Online learning : Instead of having the entire training set in memory, what if you were fed a single example at each step, and had to make a prediction? After you predict, you find out the correct label, but not until then. Once you have the label, you can of course update your model, but then you are forced to discard the example and label. The challenge here is that you have to learn continuously, and can never see the previous data. This sort of scenario comes up when there is large amount of data that cannot possibly be stored in memory — the 'big data' scenario.

You may like to do further reading on advanced topics such as these in order to do your projects, or even your dissertations next summer. In the subject of Machine Learning right now, there is significant scope for exploring new ideas, and you are encouraged to do so.

Appendix A

What is true understanding?

You will encounter lots of difficult topics during the next year. It's easy to convince yourself that you understand something, when really you don't. Here are some tips to figure out whether you REALLY understand something. These were originally printed in the book "How to think like a Mathematician", but apply equally for Computer Scientists.

You truly understand the definition of a concept if you...

- can state it precisely in at least 2 different ways,
- can state it in your own words, not from the textbook,
- can give concrete examples of it, including trivial and non-trivial examples,
- can give non-examples of the definition,
- can recognize it in different and unfamiliar situations,
- know situations in which it can be used,
- know precisely why it can be used in those situations,
- know why this particular definition is necessary,
- know similar definitions of the word, and the differences between them

You truly understand a major topic if you...

- can see how it all fits together,
- can encapsulate it in a single sentence,
- can give a concrete example which exhibits many features of the theory,
- can see connections/similarities/differences between this topic and others,
- can move effortlessly between intuition and technical details in an argument
- know why it is interesting and useful,
- know what is the bare minimum needed to make the theory work,
- know which ideas get used again and again in the theory,
- can explain it without notes,
- can explain it to non-specialists (my favourite test subject is my mum)

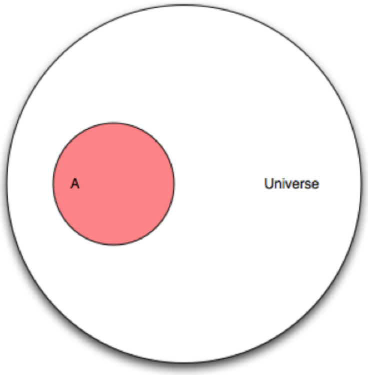
So, next time you think you truly understand something, see how many of the above you can do....

Appendix B

Visualising Bayes Theorem

In this appendix we're going to see a completely different explanation of probabilities and Bayes' Theorem. If you understand Bayes already, you can skip this section. I have just included it to provide another way of thinking, since everyone is different. *The text in this section is credited to* **Oscar Bonilla**.

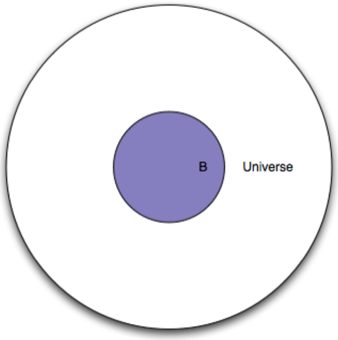
One of the easiest ways to understand probabilities is to think of them in terms of Venn Diagrams. You basically have a Universe with all the possible outcomes (of an experiment for instance), and you are interested in some subset of them, namely some event. Say we are studying cancer, so we observe people and see whether they have cancer or not. If we take as our Universe all people participating in our study, then there are two possible outcomes for any particular individual, either he has cancer or not. We can then split our universe in two events: the event ”people with cancer" (designated as A), and “people with no cancer" (or $\neg A$). We could build a diagram like this:



So what is the probability that a randomly chosen person has cancer? It is just the number of elements in A divided by the number of elements of U (the Universe). We denote the number of elements of A as $|A|$, and call it the *cardinality* of A. Furthermore, define the probability of A, $P(A)$, as

$$P(A) = \frac{|A|}{|U|} \tag{B.1}$$

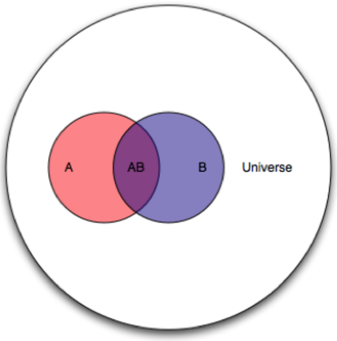
Since A can have at most the same number of elements as U, the probability P(A) can be at most 1.0. Good so far? Okay, let's add another event. Let's say there is a new screening test that is supposed to measure something. That test will be ”positive” for some people, and ”negative” for some other people. If we take the event B to mean ”people for which the test is positive”. We can create another diagram:



So what is the probability that the test will be ”positive” for a randomly selected person? It would be the number of elements of B (the cardinality of B, or $|B|$) divided by the number of elements of U, we call this P(B), the probability of event B occurring.

$$P(B) = \frac{|B|}{|U|} \tag{B.2}$$

Note that so far, we have treated the two events in isolation. What happens if we put them together?



We can compute the probability of both events occurring (AB is shorthand for $A \cap B$) in the same way.

$$P(AB) = \frac{|AB|}{|U|} \tag{B.3}$$

But this is where it starts to get interesting. What can we read from the diagram above?

We are dealing with an entire Universe (all people), the event A (people with cancer), and the event B (people for whom the test is positive). There is also an overlap now, namely the event AB which we can read as "people with cancer and with a positive test result". There is also the event $B - AB$ or "people without cancer and with a positive test result", and the event $A - AB$ or "people with cancer and with a negative test result".

Now, the question we'd like answered is "given that the test is positive for a randomly selected individual, what is the probability that said individual has cancer?". In terms of our Venn diagram, that translates to "given that we are in region B, what is the probability that we are in region AB?" or stated another way "if we make region B our new Universe, what is the probability of A?". The notation for this is $P(A|B)$ and it is read "the probability of A given B".

So what is it? Well, it should be

$$P(A|B) = \frac{|AB|}{|B|} \tag{B.4}$$

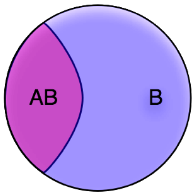
And if we divide both the numerator and the denominator by $|U|$

$$P(A|B) = \frac{\frac{|AB|}{|U|}}{\frac{|B|}{|U|}} \tag{B.5}$$

we can rewrite it using the previously derived equations as

$$P(A|B) = \frac{P(AB)}{P(B)} \tag{B.6}$$

What we've effectively done is change the Universe from U (all people), to B (people for whom the test is positive), but we are still dealing with probabilities defined in U.



Now let's ask the converse question "given that a randomly selected individual has cancer (event A), what is the probability that the test is positive for that individual (event AB)?". Well, that it is just:

$$P(B|A) = \frac{P(AB)}{P(A)} \tag{B.7}$$

Now we have everything we need to derive Bayes' theorem, putting those two equations together we get

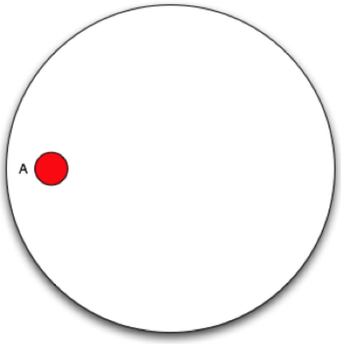
$$P(A|B)P(B) = P(B|A)P(A) \tag{B.8}$$

which is to say P(AB) is the same whether you're looking at it from the point of view of A orB, and finally, we have Bayes rule:

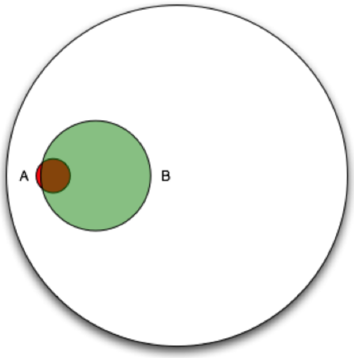
$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{B.9}$$

Which is Bayes' theorem.

Example
Take the following example : 1% of women at age forty who participate in routine screening have breast cancer. 80% of women with breast cancer will get positive mammograms. 9.6% of women without breast cancer will also get positive mammograms. A woman in this age group had a positive mammography in a routine screening. What is the probability that she actually has breast cancer? First of all, let's consider the women with cancer:



Now add the women with positive mammograms, note that we need to cover 80% of the area of event A and 9.6% of the area outside of event A.



It is clear from the diagram that if we restrict our universe to B (women with positive mammograms), only a small percentage actually have cancer. According to the article, most doctors guessed that the answer to the question was around 80%, which is clearly impossible looking at the diagram!

Note that the efficacy of the test is given from the context of A, ”80% of women with breast cancer will get positive mamograms”. This can be interpreted as ”restricting the universe to just A, what is the probability of B?” or in other words $P(B|A)$.

Even without an exact Venn diagram, visualizing the diagram can help us apply Bayes' theorem:

- 1% of women in the group have breast cancer ... $P(A) = 0.01$
- 80% of those women get a positive mammogram, and 9.6% of the women without breast cancer get a positive mammogram too ... $P(B) = 0.8P(A) + 0.096(1 - P(A)) = 0.008 + 0.09504 = 0.10304$
- we can get P(B—A) straight from the problem statement, remember 80% of women with breast cancer get a positive mammogram ... $P(B|A) = 0.8$

Now let's plug those values into Bayes' theorem

$$P(A|B) = \frac{0.8 \times 0.01}{0.10304} \tag{B.10}$$

which is 0.0776 or about a 7.8% chance of actually having breast cancer given a positive mammogram.