UNIT IV

Evolutionary Learning:
Genetic Algorithms, Genetic Operators, Genetic Programming
Ensemble learning:
Boosting, Bagging
Dimensionality Reduction:
Linear Discriminant Analysis, Principal Component Analysis

I. Evolutionary Learning

- In this chapter we are going to start by treating evolution the same way that we treated neuroscience earlier by picking a few useful concepts, and then filling in the gaps with computer science in order to make an effective learning method.
- To see why this might be interesting, you need to view evolution as a search problem.
- Evolution works on a population through an imaginary fitness landscape, which has an implicit bias towards animals that are 'fitter', i.e., those animals that live long and generate more and healthier offspring.
- The genetic algorithm models the genetic process that gives rise to evolution.
- where both parents give some genetic information to their offspring.
- Hence, children have similarities with their parents, and there is lots of genetic inheritance. However, there are also random mutations, caused by copying errors which means that some things do change over time.

1. Genetic Algorithms

- The genetic algorithm shows many of the things that are best and worst about machine learning: it is often, but not always, very effective, it has an array of parameters that are crucial, but hard to set, and it is impossible to guarantee that it will find a result that is any good at all.
    - It often works very well, and it has become a very popular algorithm for people to use when they have no idea of any other way to find a reasonable solution.
    - Genetic algorithms perform both exploitation and exploration, so that they can make incremental improvements to current good solutions, but also find radically new solutions, some of which may be better than the current best.
- The Genetic Algorithm is a computational approximation to how evolution performs search, which is by producing modifications of the parent genomes in their offspring and thus producing new individuals with different fitness.
- Like another mathematical model it attempts to abstract away everything except the important parts that we need to understand what evolution does.
- The things that we need to model simple genetics inside a computer and solve problems with it are:
    - a method for representing problems as chromosomes
    - a way to calculate the fitness of a solution
    - a selection method to choose parents
    - a way to generate offspring by breeding the parents

- We are going to use an example to describe the methods, which is an NP-complete problem.
  - known as the **knapsack problem** (a knapsack is a rather old name for a rucksack or bag). The knapsack problem is easy to describe, but difficult to solve in general. Here is the version of it that we will use:
    - Suppose that you are packing for your holidays. You've bought the biggest and best rucksack that was for sale, but there is still no way that you are going to fit in everything you want to take(camera, money, addresses of friends, etc.) and the things that your mum is insisting you take(spare phrasebook, stamps to write home with, etc.).
    - As a good computer scientist you decide to measure how much space it takes up and then write a program to work out how to fill as much of the bag as possible, so that you get the best value for your airfare.
    - This problem, and variations of it, appear in various disguises in cryptography, combinatorics, applied mathematics, logistics, and business, so it is an important problem. Unfortunately, since it is NP-complete, finding the optimal solution for interesting cases (pretty much anything above 10 items) is computationally impossible.

1.1 String Representation

- The first thing that we need is some way to represent the individual solutions, in analogy to the chromosome.
- GAs use a string, with each element of the string (equivalent to the gene) being chosen from some alphabet. The different values in the alphabet, which is often just binary, are analogous to the alleles.
- For the problem we are trying to solve we have to work out a way of encoding the description of a solution as a string. We then create a set of random strings to be our initial population.
- It is possible to modify the GA so that the alphabet it uses runs over the real it is quite popular, because of the number of applications, but it is not as elegant as using a discrete alphabet. It also tends to make the mutation operator that we will see later less useful.
- For the knapsack problem the alphabet is very simple, since we can make it binary, since for each item we just need to say whether or not we want to take it. We make the string L units long, where **L** is the total **number of things** we would like to take with us, and make each unit a binary digit. We then encode a solution using 0 for the things we will not take and 1 for the things we will. So if there were four things we wanted to take, then (0, 1, 1, 0) would mean that we take the middle two, but not the first or last.
- Note that this does not tell us whether or not this string is possible (that is, whether the things that we have said we will take will actually fit into the knapsack), nor whether it is a good string (whether it fills the knapsack). To work these out we need some way to decide how well each string fulfills the problem criteria. This is known as the **fitness of the string.**

1.2 Evaluating Fitness

- The fitness function can be seen as an oracle that takes a string as an argument and returns a value for that string.
- Together with the string encoding the fitness function forms the problem-specific part of the GA.
- It is worth thinking about what we want from our fitness function. Clearly, the best string should have the highest fitness, and the fitness should decrease as the strings do less well on the problem.

- For the knapsack problem, we decided that we wanted to make the bag as full as possible. So we would need to know the volume of each item that we want to put into the knapsack, and then for a given string that says which things should be taken, and which should not, we can compute the total volume. This is then a possible fitness function.
- However, it does not tell us anything about whether they will fit into the bag—with this fitness function the optimal solution is to take everything. So we need to check that they will fit, and if they will not, reduce the fitness of that solution. One option would be to set the fitness to 0 if the things in that string will not all fit. However, suppose that the solution is almost perfect, it is just that there is one thing too many in the knapsack. By setting the fitness to 0 we are reducing the chance of this solution being allowed to evolve and improve during later iterations.

For this reason we will make the fitness function be the sum of the values of the items to be taken if they fit into the knapsack, but if they do not we will subtract twice the amount by which they are too big for the knapsack from the size of the knapsack. This allows solutions that are only just over to be considered for improvement, but tries to ensure that they are not the fittest solutions around.

There is an obvious greedy algorithm that finds solutions to the knapsack problem. At each stage it takes the largest thing that hasn't been packed yet and that will still fit into the bag, and iterates that rule. This will not necessarily return the optimal solution (unless each thing is larger than the sum of all the ones smaller than it, in which case it will), but it is very quick and simple.

So a GA should be getting a much better solution than the greedy rule most of the time to be worth all the effort involved in writing and running it.

1.3 Population

- We can now measure the fitness of any string. The GA works on a population of strings, with the first generation usually being created randomly.
- The fitness of each string is then evaluated, and that first generation is bred together to make a second generation, which is then used to generate a third, and so on.
- After the initial population is chosen randomly, the algorithm evolves to produce each successive generation, with the hope being that there will be progressively fitter individuals in the populations as the number of generations increases.

1.4 Generating Offspring: Parent Selection

- For the current generation we need to select those strings that will be used to generate new offspring.
  - However, it is also good to allow some exploration in there, which means that we have to allow some possibility of weak strings being considered. There are two commonly employed ways to do this, although the last one tends to produce better results:

Truncation Selection

- Pick some fraction f of the best strings and ignore the rest. For example, f = 0.5 is often used, so the best 50% of the strings are put into the pool, each twice so that the pool is the right size. The pool is randomly shuffled to make the pairs. This is obviously very easy to

implement, but it does limit the amount of exploration that is done, biasing the GA towards exploitation.

Fitness Proportional Selection

- The better option is to select strings probabilistically, with the probability of a string being selected being proportional to its fitness. The function that is generally used is (for string α):

$$p^{\alpha} = \frac{F^{\alpha}}{\sum_{\alpha'} F^{\alpha'}}, \tag{10.1}$$

where F^{α} is the fitness. If the fitness is not positive then F needs to be replaced by $\exp(sF)$ throughout, where s is the selection strength, a parameter, and you might recognise the equation as the soft-max activation from Chapter 4:

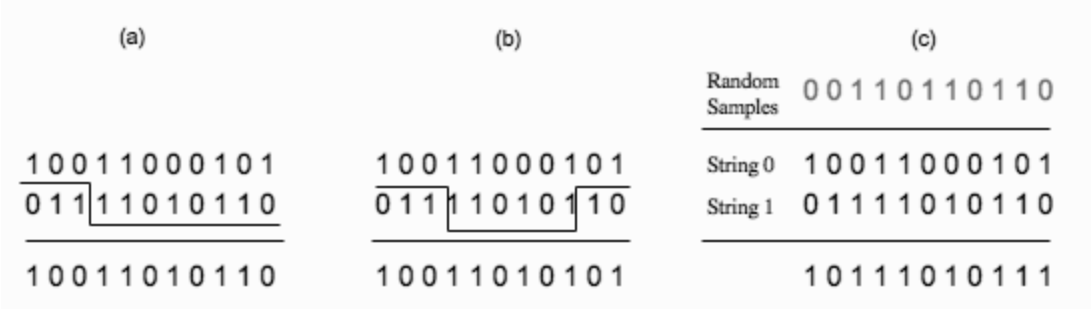
$$p^{\alpha} = \frac{\exp(sF^{\alpha})}{\sum_{\alpha'} \exp(sF^{\alpha'})}. \tag{10.2}$$

- There is an implementation issue here. We want to pick each string with probability proportional to its fitness, but if we only have one copy of each string, then the probability of picking each string is the same.
- One way around this is to add more copies of the fitter strings, so that they are more likely to get chosen. This is sometimes called '**roulette selection**', because if you imagine that each string gets an area on a roulette wheel, then the larger the area associated to one number, the more likely it is that the ball will land there. You can then just randomly pick strings from this larger set.
- Having selected our breeding pairs, we now need to decide how to combine their two strings to generate the offspring, which is the genetics part of the algorithm. There are two genetic operators that are generally used

GENETIC OPERATORS

1.Crossover

- In biology, organisms have two chromosomes, and each parent donates one of them. Members of our GA population only have one chromosome-equivalent, the string.
- Thus, we generate the new string as part of the first parent and part of the second. The most common way of doing this is to pick one point at random in the string, and to use parent 1 for the first part of the string, up to the crossover point and parent 2 for the rest. We actually generate two offspring, with the second one consisting of the first part of parent 2 and the second part of parent 1. This scheme is known as **single point crossover.**
- Extension to multi-point crossover is hopefully obvious. The most 'extreme' version is known as uniform crossover and consists of independently selecting each element of the string at random from the two parents.

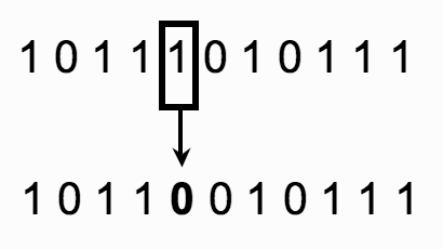


The different forms of the crossover operator.
(a) Single point crossover. A position in the string is chosen at random, and the offspring is made up of the first part of parent 1 and the second part of parent 2.
(b) Multi-point crossover. Multiple points are chosen, with the offspring being made in the same way.
(c) Uniform crossover. Random numbers are used to select which parent to take each element from.

- Crossover is the operator that performs global exploration, since the strings that are produced are radically different to both parents in at least some places. The hope is that sometimes we will take good parts of both solutions and put them together to make an even better solution.
- Crossover is not always useful, depending upon the problem; for example, in the **Travelling Salesman Problem**, the strings that are generated by crossover might not even be valid tours.
- However, when it is useful, it is often the more powerful of the genetic operators, and has led to the building block hypothesis of how GAs work.
- The idea is that GAs work well on problems where the solution comes from putting together lots of little solutions, so that different strings assemble each separate building block, and then crossover puts those substrings together to make the final solution.

2.Mutation

- The other genetic operator is mutation, which effectively performs local random search. The value of any element of the string can be changed, governed by some (usually low) probability p.
- For our binary alphabet in the knapsack problem, mutation causes a bit-flip, as is shown in Figure 10.3.
- For chromosomes with real values, some random number is generally added or subtracted from the current value.
- Often, p 1/L where L is the string length, so that there is approximately one mutation in each string. This might seem quite high, but it is often found to be a good choice given that the mutation rate has to trade off doing lots of local search with the risk of disrupting the good solutions.



The effects of mutation on a string

3.Elitism, Tournaments, and Niching

- At this stage we have taken pairs of parents, and produced pairs of offspring. There is now a choice of what to do with them.
- The simplest option would be **simply replace** the parents by their children to make a completely new population, and carry on from there.
- This seems a bit risky: we could potentially lose a really good string that we find early on in the search, and that we never see again.
- Use **elitism**, which takes some number of the fittest strings from one generation and puts them directly into the next population, replacing strings that are already there either at random, or by choosing the least fit to replace. Note that at every iteration the population stays the **same size** something else that is unlike real evolution.
- Another solution is to implement a **tournament**, where the two parents and their two offsprings compete, with the two fittest out of the four being put into the new population
- While elitism and tournaments both ensure that good solutions aren't lost, they both have the problem that they can **encourage premature convergence**, where the algorithm settles down to a constant population that never changes even though it hasn't found an optimum.
- This happens because the GA favors fitter members of the population, which means that a solution that reaches a local maximum will generally be favored.
- Tournaments and elitism encourage this, because they **reduce** the amount of **diversity** in the population by allowing the same individuals to remain over many generations.
- This means that the **exploration** aspect of the GA stops occurring. **Exploration** will be downplayed, making it hard to escape from the local maximum—most strings will have worse fitness, and will therefore be replaced in the population.
- Eventually, the majority of the strings in the population will be the same, but will **represent a local maximum, not the global maximum**.
- The randomness in the GA is a very large part of why it works, and schemes to reduce that randomness often harm the overall results.
- One way to solve the problem of premature convergence is through **niching** (also known as using **island populations**),
- where the population is separated into several subpopulations, which all evolve independently for some period of time, so that they are likely to have converged to different local maxima, and a few members of one subpopulation are occasionally injected as **'immigrants**' into another subpopulation.
- Another approach is known as **fitness sharing**, where the fitness of a particular string is averaged across the number of times that that string appears in the population.

The Basic Genetic Algorithm

- **Initialisation**
  - generate N random strings of length L with our chosen alphabet
- **Learning**
  - repeat:
    - create an (initially empty) new population
    - repeat:
      - select two strings from current population by fitness
      - recombine them to produce two new strings
      - mutate the offspring
      - either add the two offspring to the population or use elitism or tournaments
      - keep track of the best string in the population
    - until N strings for the new population are generated
    - replace the current population with the new population
  - until stopping criteria met

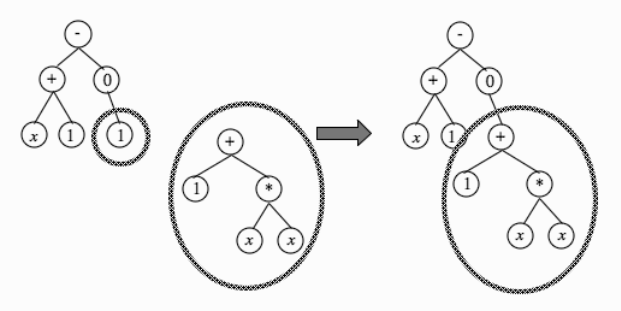
- The algorithm is often run for a fixed number of generations.
- It is a computationally very expensive algorithm, especially if the fitness function is non-trivial to evaluate.

Limitations of the GA

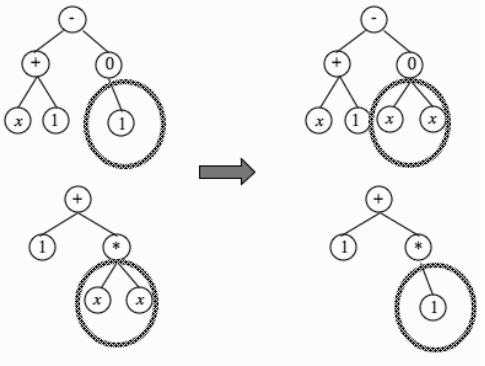
- There are lots of good things about genetic algorithms, and they work amazingly well a lot of the time. However, they are not without problems, a significant one of which is they can be **very slow.**
- The main problem is that once a **local maximum** has been reached, it can often be a **long time** before a string is produced that escapes from the local maximum and finds another, higher, maximum.
- In addition, because we generally do not know anything about the **fitness landscape**, we can't see how well the GA is doing.
- A more basic criticism of genetic algorithms is that it is **very hard** (read basically impossible) to **analyse the behaviour of the GA** which means that we cannot guarantee that the algorithm will converge at all, and certainly not to the optimal solution.
- That said, genetic algorithms are widely used when **other methods do not** work, and they are usually treated as a **black box**—strings are pushed in one end, and eventually an answer emerges.
  - This is risky, because without knowledge of how the algorithm works it is not possible to improve it, nor do you know how cautiously you should treat the results.

3.Genetic Programming

- One extension of genetic algorithms that has had a lot of attention is the idea of genetic programming.
- This was introduced by John Koza, and the basic idea is to represent a computer program as a tree (imagine a flow chart of the code).
- For certain programming languages, notably LISP, this is actually a very natural way to represent a program, but it doesn't work very well in Python.
- Tree-based variants on mutation and crossover are defined (replace subtrees by other subtrees, either randomly generated (mutation, Figure 10.13) or swapped from another tree (crossover, Figure 10.14)), and then the genetic program runs just like a normal genetic algorithm, but acting on these program trees rather than strings.
- Figure 10.15 shows a set of simple trees that perform arithmetic operations, and some possible developments of them, made using these operators.

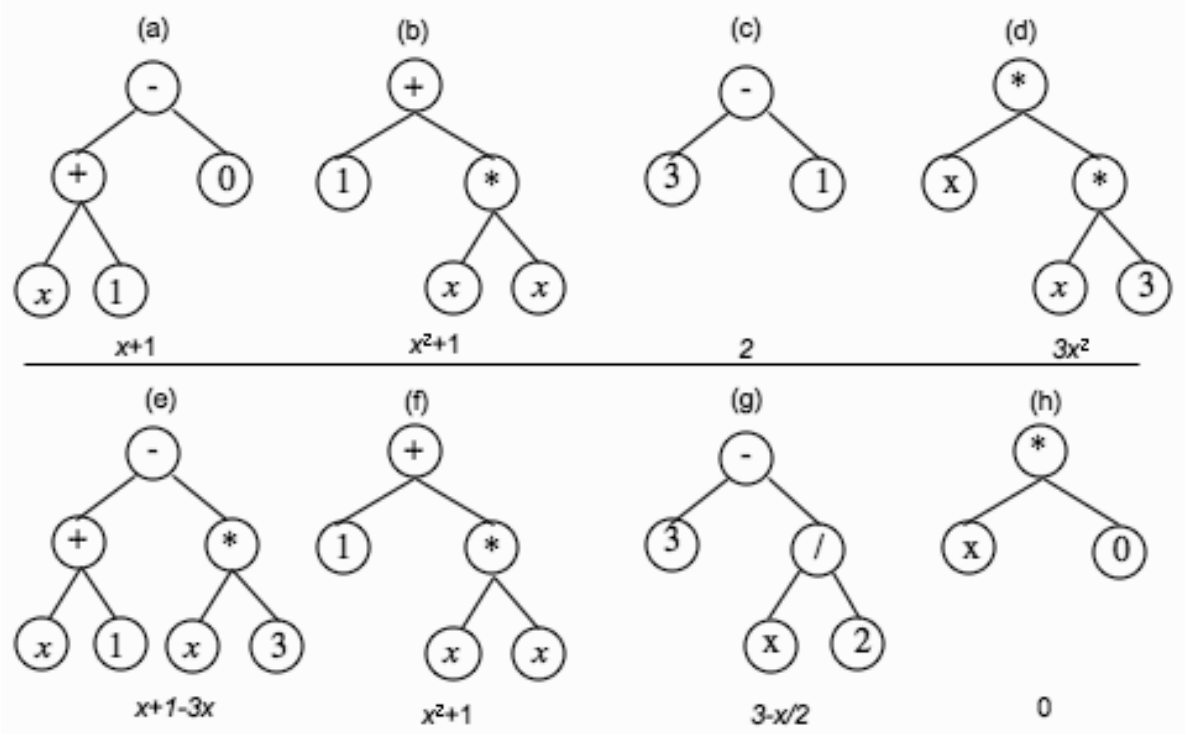


Example of a mutation in genetic programming



Example of a crossover in genetic programming

- Top: Four arithmetical trees. Bottom: Example developments of the four trees: (e) and (h) are a possible crossover of (a) and (d), (f) is a copy of (b), and (g) is a mutation of (c).



II. Ensemble Learning

- The basic idea is that by having **lots of learners** that each get slightly different results on a dataset—some learning certain things well and some learning others—and putting them together, the results that are generated will be **significantly better** than any one of them on its own.
- Figure shows the basic idea of ensemble learning, as these methods are collectively called. Given a relatively simple binary classification problem and some learner that puts an ellipse around a subset of the data, combining the ellipses can provide a considerably more complex decision boundary.

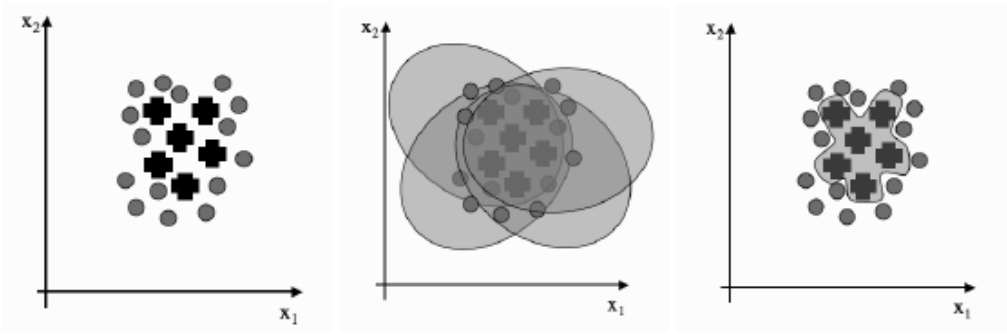


FIGURE 13.1 By combining lots of simple classifiers (here that simply put an elliptical decision boundary onto the data), the decision boundary can be made much more complicated, enabling the difficult separation of the pluses from the circles.

- There are then only a couple of questions to ask:
- which learners should we **use**, how should we ensure that they **learn different things**, and how should we **combine** their results?
- Ensuring that the learners see **different things** can be performed in different ways, and it is the primary difference between the algorithms that we shall see.
- Suppose that you have **lots and lots of data**. In that case you could simply **randomly partition** the data and give different sets of data to different classifiers. Even here there are choices: do you make the partitions **separate**, or include **overlaps**? If there is no overlap, then it could be difficult to work out how to combine the classifiers, or it might be very simple:
- Ensemble methods do very well when there is very little data as well as when there is too much. To see why, think **cross-validation** .
    - We used cross validation when there **was not enough data** to go around, and trained lots of neural networks on different subsets of the data. Then we threw away most of them.
    - **With an ensemble method we keep them all**, and combine their results in some way. One very simple way to combine the results is to use **majority voting** —, it's good enough for machine learning.
    - Majority voting has the interesting property that for binary classification, the combined classifier will only get the answer wrong if **more than half** of the classifiers were wrong.

1.Boosting

- If we take a collection of **very poor learners**, each performing only just better than chance, then by putting them together it is possible to make an ensemble learner that can perform arbitrarily **well.** So we just need lots of low-quality learners, and a way to put them together usefully, and we can make a learner that will do very well.
- 1990 boosting algorithm, which was rather data hungry. In that algorithm, the **training set** was split into **three**.
- A classifier was **trained** on the **first third**, and then **tested** on the **second third**.
- All of the data that was **misclassified** during that testing was used to form a **new dataset**, along with an **equally sized random selection** of the data that was **correctly classified**.
- A **second classifier** was trained on this new dataset, and then **both** of the classifiers were tested on the final **third** of the dataset. If they both produced the **same output**, then that **data point** was **ignored**, **otherwise** the data point was added to yet another **new dataset**, which formed the **training** set for a third classifer.

AdaBoost(adaptive boosting)

- The innovation that AdaBoost (which stands for adaptive boosting) uses is to give **weights** to each data point according to **how difficult** previous classifiers have found to get it correct. These weights are given to the classifier as part of the input when it is trained.
- The AdaBoost algorithm is conceptually very simple.
- At each iteration a new classifier is trained on the training set, with the **weights** that are applied to the training set for each data point being **modified** at **each iteration** according to how successfully that data point has been classified in the past.
- The weights are initially all set to the same value, **1/N**, where N is the number of data points in the training set.

- Then, at each iteration, the error (ε) is computed as the sum of the weights of the **misclassified points**, and the **weights** for **incorrect** examples are updated by being **multiplied by = (1 − ε)/ ε.**
- **Weights** for **correct** examples are left alone, and then the whole set is **normalized** so that it **sums to 1** (which is effectively a reduction in the importance of the correctly classified data points).
- Training terminates after a set number of iterations, or when either all of the data points are classified correctly, or one point contains more than half of the available weight.

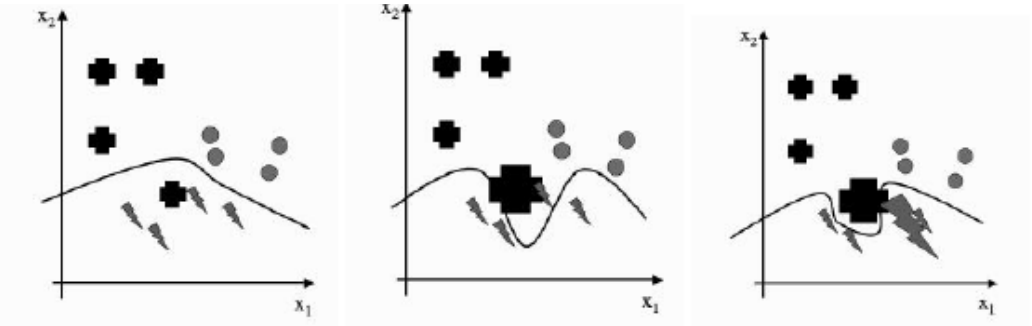


FIGURE 13.2 As points are misclassified, so their weights increase in boosting (shown by the datapoint getting larger), which makes the importance of those datapoints increase, making the classifiers pay more attention to them.

AdaBoost Algorithm

- Initialise all weights to $1/N$, where N is the number of datapoints
- While $0 < \epsilon_t < \frac{1}{2}$ (and $t < T$, some maximum number of iterations):
  - train classifier on $\{S, w^{(t)}\}$, getting hypotheses $h_t(x_n)$ for datapoints x_n
  - compute training error $\epsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n))$
  - set $\alpha_t = \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$
  - update weights using:

$$w_n^{(t+1)} = w_n^{(t)} \exp(\alpha_t I(y_n \neq h_t(x_n))/Z_t, \qquad (13.1)$$

    where Z_t is a normalisation constant
- Output $f(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$

- Most of the work of the algorithm is done by the classification algorithm, which is given new weights at each iteration. In this respect, boosting is not quite a stand-alone algorithm: the classifiers need to consider the weights when they perform their classifications.

- As a very simple example showing how boosting works, a very simple classifier was created that can only separate data by fitting one either horizontal or vertical line, with it choosing which to fit at the current iteration at random. A two-dimensional dataset was created with data in the top

right-hand corner being in one class, and the rest in another, plus a couple of the data points were randomly mislabeled to simulate noise. Clearly, this dataset cannot be separated by a single horizontal or vertical decision boundary.

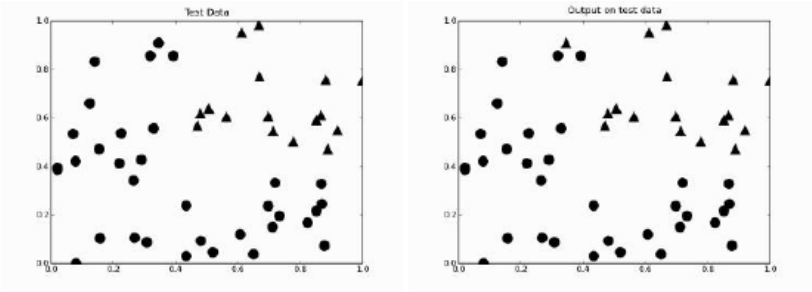


FIGURE 13.3 Boosting learns this simple dataset very successfully, producing an ensemble classifier that is rather more complicated than the simple horizontal or vertical line classifier that the algorithm boosts. On the independent test set shown here, the algorithm gets only 1 datapoint wrong, and that is one that is coincidentally close to one that was misclassified to simulate noise in the training data.

$$G_t(\alpha) = \sum_{n=1}^{N} \exp\left(-y_n(\alpha h_t(x_n) + f_{t-1}(x_n))\right), \tag{13.2}$$

where $f_{t-1}(x_n)$ is the sum of the hypotheses of that datapoint from the previous iterations:

$$f_{t-1}(x_n) = \sum_{\tau=0}^{t-1} \alpha_\tau h_\tau(x_n). \tag{13.3}$$

in the algorithm are nothing more than the second term in Equation (13.2), which can therefore be rewritten as:

$$G_t(\alpha) = \sum_{n=1}^{N} w^{(t)} \exp\left(-y_n \alpha h_t(x_n)\right). \tag{13.4}$$

- Figure 13.4 shows the training data, the error curve on both the training and testing sets, and the first few iterations of the classifier, which can only put in one horizontal or linear classification line.
- Clearly, such impressive results require some explanation and understanding. The key to this understanding is to compute the loss function, which is simply the measure of the error that is applied. The loss function for AdaBoost has the form

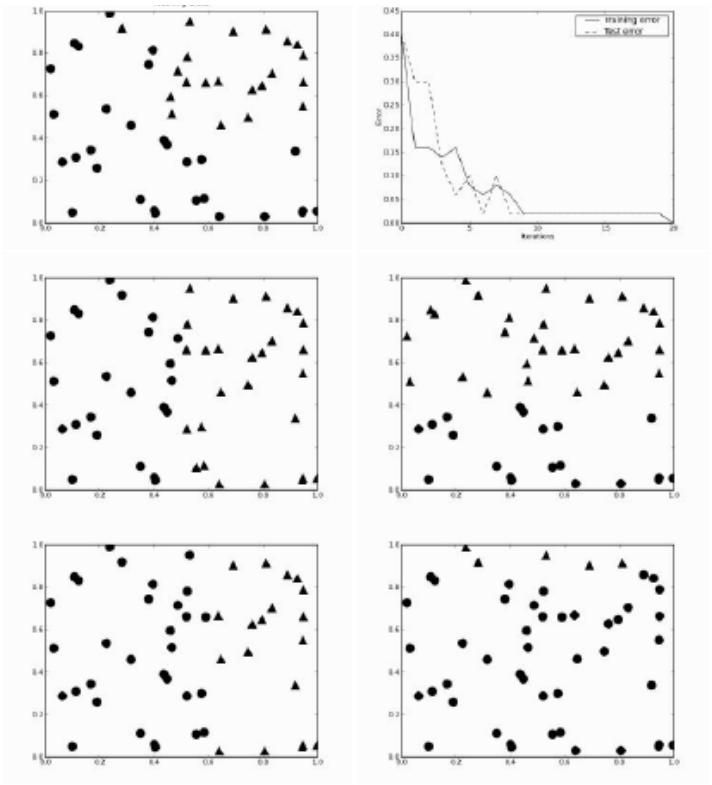


FIGURE 13.4 *Top:* the training data and the error curve. *Middle and bottom:* The first few iterations of the classifier; each plot shows the output of one of the weak classifiers that are boosted by the algorithm.

- Deriving the rest of the algorithm from here requires substituting in for the hypotheses h and then solving for , which produces the full algorithm.
- It is possible to choose other loss functions, and providing that they are differentiable they will provide useful boosting-like algorithms, which are collectively known as arcing algorithms (for **adaptive reweighting and combining).**
- AdaBoost can be modified to perform regression rather than classification (known as real adaboost, or sometimes adaboost.R).
- There is another variant on boosting (also called AdaBoost, confusingly) that uses the weights to sample from the full dataset, training on a sample of the data rather than the full weighted set, with more difficult examples more likely to be in the training sample. This is more in line with the original boosting algorithm, and is obviously faster, since each training run has fewer data to learn about.

Stumping

- There is a very extreme form of boosting that is applied to trees.

- It goes by the descriptive name of stumping. The stump of a tree is the tiny piece that is left over when you chop off the rest, and the same is true here:
- stumping consists of simply taking the root of the tree and using that as the decision maker. So for each classifier you use the very first question that makes up the root of the tree, and that is it.
- [illegible]

2.Bagging

- The simplest method of combining classifiers is known as **bagging**, which stands for bootstrap aggregating, the statistical description of the method.
- A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all.
- The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: B of them, where B is at least 50, and could even be in the thousands.
- The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem.
- It is sufficiently common to have inspired the comment that “statistics is defined as the discipline where those that think don’t count and those that count don’t think.”

Subagging

- However, the method of subagging wins the prize for the oddest sounding word.
- It is a combination of ‘subsample’ and ‘bagging,’ and it is the fairly obvious idea that you don’t need to produce samples that are the same size as the original data.
- If you make smaller datasets, then it makes sense to sample without replacement, but otherwise the implementation is only very slightly different from the bagging one,
- It is common to use a dataset size that is half that of the original data, and the results of this can often be comparable to a full bagging simulation.

III. Dimensionality Reduction:

- These are some of the reasons **why dimensionality reduction is useful ?**.
- However, it can also **remove noise**, significantly **improve** the results of the learning algorithm, make the dataset **easier to work** with, and make the results **easier to understand**.
- In addition, we have already seen the curse of dimensionality means that the **higher** the number of dimensions we have, **the more training data we need**. Further, the dimensionality is an explicit factor for **the computational cost** of many algorithms.
- In extreme cases such as the **Self-Organising Map** where the number of dimensions becomes three or fewer, we can also plot the data, which makes it much easier to understand and interpret.
- There are three different ways to do **dimensionality reduction.**
  - The **first** is **feature selection**, which typically means looking through the features that are available and seeing whether or not they are actually useful, i.e., correlated to the output variables.
  - The **second** method is **feature derivation**, which means deriving new features from the old ones, generally by applying transforms to the dataset that simply change the axes (coordinate system) of the graph by moving and rotating them, which can be written simply as a matrix that we apply to the data. The reason this performs dimensionality reduction is that it enables us to combine features, and to identify which are useful and which are not.
  - The **third** method is simply to **use clustering** in order to group together similar data points, and to see whether this allows fewer features to be used
  - To see how choosing the right features can make a problem significantly simpler, have a look at the table on the left of Figure 6.1.
  - It shows the x and y coordinates of 4 points. Looking at the numbers it is hard to see any correlation between the points, and even when they are plotted it simply looks like they might form corners of a rotated rectangle.
  - However, the plot on the right of the figure shows that they are simply a set of four points from a circle, (in fact, the points at (/6, 4/6, 7/6, 11/6)) and using this one coordinate, the angle, makes the data a lot easier to understand and analyses.

x	y
2.00	-1.43
2.37	-2.80
1.00	-3.17
0.63	-1.80

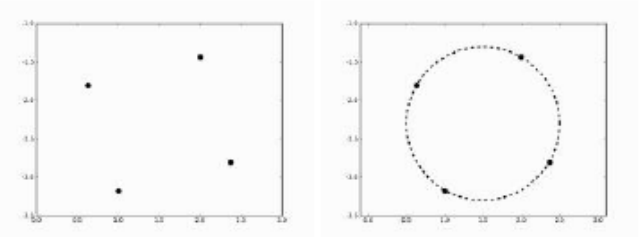


FIGURE 6.1 Three views of the same four points. *Left:* As numbers, where the links are unclear. *Centre:* As four plotted points. *Right:* As four points that lie on a circle.

  - Once we have worked out how to represent the data, **we can suppress dimensions** that aren't useful to the algorithm.

- Even before we get into any form of analysis at all, we can try to perform feature selection, looking at the possible inputs that we have for the problem, and deciding which are useful.
- Many of the **methods merge this idea with transformations of the data**, so that combinations of the different inputs, rather than the inputs themselves, are used. However, even before using any of the algorithms identified here, input features can be ignored if they do not seem to be useful.
- We will see another method of doing **feature selection** later, since it is inherent to the way that the decision tree works:
- at each stage of the algorithm it decides which feature to add next. This is the **constructive way** to decide on the features: start with none, and then iteratively add more, testing the error at each stage to see whether or not it changed at all when that feature was added.
- The **destructive method** is to prune the decision tree, lopping off branches and checking whether the error changed at all.
- In general, **selecting the features is a search problem**. We take the best system so far, and then search over the set of possible next features to add. This can be computationally very expensive, since **for d features** there are $\mathbf{2^d - 1}$ possible sets of features to search over, from any individual feature up to the full set.
- A method of **dimensionality reduction that is aimed at supervised learning, Linear Discriminant Analysis.** This method is credited to one of the best-known statisticians of the 20th century, R.A. Fisher, and dates from 1936.

1. Linear Discriminant Analysis (LDA)

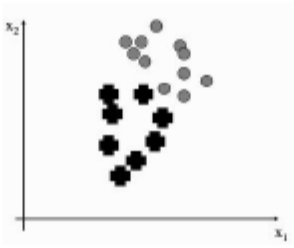


FIGURE 6.2 A set of datapoints in two dimensions, with two classes.

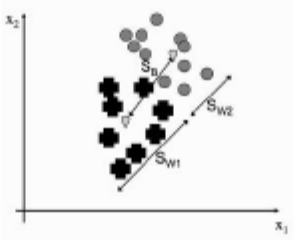


FIGURE 6.3 The meaning of the between-class and within-class scatter. The hearts mark the means of the two classes.

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} p_c(\mathbf{x}_j - \boldsymbol{\mu}_c)(\mathbf{x}_j - \boldsymbol{\mu}_c)^T. \qquad (6.1)$$

- figure shows a simple two-dimensional dataset consisting of two classes. We can compute various statistics about the data, but we will settle for the means of the two classes in the data, μ1 and μ2, the mean of the entire dataset (μ), and the covariance of each class with itself, which is $\sum_j (x_j - \mu)(x_j - \mu)^T$.
- The question is what we can do with these pieces of data.
- The principal insight of LDA is that the covariance matrix can tell us about the scatter within a dataset, which is the amount of spread that there is within the data.

- The way to find this scatter is to multiply the covariance by the pc, the probability of the class (that is, the number of data points there are in that class divided by the total number). Adding the values of this for all of the classes gives us a measure of the within-class scatter of the dataset:

FIGURE 6.2 A set of datapoints in two dimensions, with two classes.

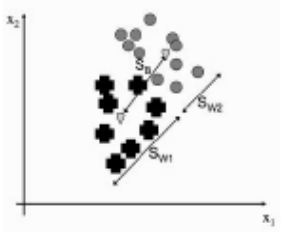


FIGURE 6.3 The meaning of the between-class and within-class scatter. The hearts mark the means of the two classes.

$$S_W = \sum_{\text{classes } c} \sum_{j \in c} p_c (\mathbf{x}_j - \boldsymbol{\mu}_c)(\mathbf{x}_j - \boldsymbol{\mu}_c)^T. \tag{6.1}$$

If our dataset is easy to separate into classes, then this within-class scatter should be small, so that each class is tightly clustered together. However, to be able to separate the data, we also want the distance *between* the classes to be large. This is known as the between-classes scatter and is a significantly simpler computation, simply looking at the difference in the means:

$$S_B = \sum_{\text{classes } c} (\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T. \tag{6.2}$$

- The meanings of these two measurements is shown in Figure 10.3.
- The argument about good separation suggests that datasets that are easy to separate into the different classes (i.e., the classes are discriminable) should have SB/SW as large as possible.
- This seems perfectly reasonable, but it hasn't told us anything about dimensionality reduction.
- However, we can say that the rule about making SB/SW as large as possible is something that we want to be true for our data when we reduce the number of dimensions.
- Figure 10.4 shows two projections of the dataset onto a straight line. For the projection on the left it is clear that we can't separate out the two classes, while for the one on the right we can. So we just need to find a way to compute a suitable projection.
- So we can compute the projection of our data along w for every point, and we will have projected our data onto a straight line, as is shown in the two examples in Figure 6.4. Since the mean can be treated as a datapoint, we can project that as well: $\mu`_c = w^T \cdot \mu_c$. Now we just need to work out what happens to the within-class and between-class scatters.

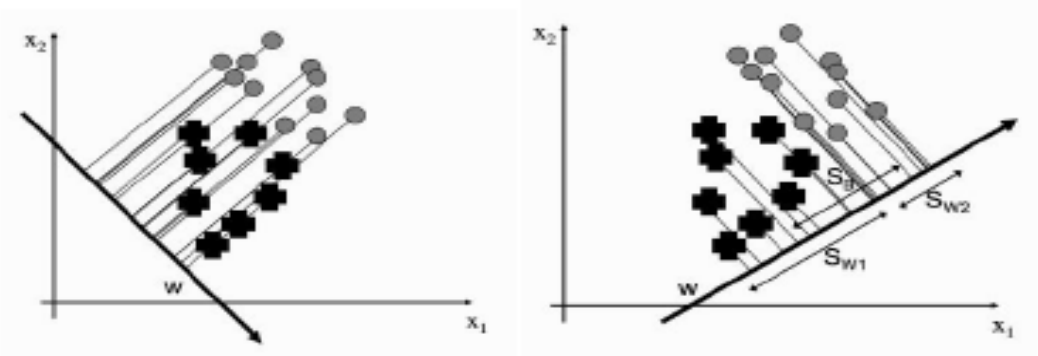


FIGURE 6.4 Two different possible projection lines. The one on the left fails to separate the classes.

Replacing $\mathbf{x}_j$ with $\mathbf{w}^T \cdot \mathbf{x}_j$ in Equations (6.1) and (6.2) we can use some linear algebra (principally the fact that $(\mathbf{A}^T\mathbf{B})^T = \mathbf{B}^T\mathbf{A}^{T^T} = \mathbf{B}^T\mathbf{A}$) to get:

$$\sum_{\text{classes } c} \sum_{j \in c} p_c(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))(\mathbf{w}^T \cdot (\mathbf{x}_j - \boldsymbol{\mu}_c))^T = \mathbf{w}^T S_W \mathbf{w} \quad (6.3)$$

$$\sum_{\text{classes } c} \mathbf{w}^T(\boldsymbol{\mu}_c - \boldsymbol{\mu})(\boldsymbol{\mu}_c - \boldsymbol{\mu})^T\mathbf{w} = \mathbf{w}^T S_B \mathbf{w}. \quad (6.4)$$

So our ratio of within-class and between-class scatter looks like $\frac{\mathbf{w}^T S_W \mathbf{w}}{\mathbf{w}^T S_B \mathbf{w}}$.

In order to find the maximum value of this with respect to $\mathbf{w}$, we differentiate it and set the derivative equal to 0. This tells us that:

$$\frac{S_B\mathbf{w}(\mathbf{w}^T S_W \mathbf{w}) - S_W\mathbf{w}(\mathbf{w}^T S_B \mathbf{w})}{(\mathbf{w}^T S_W \mathbf{w})^2} = 0. \quad (6.5)$$

So we just need to solve this equation for $\mathbf{w}$ and we are done. We start with a little bit of rearranging to get:

$$S_W\mathbf{w} = \frac{\mathbf{w}^T S_W \mathbf{w}}{\mathbf{w}^T S_B \mathbf{w}} S_B \mathbf{w}. \quad (6.6)$$

- Unfortunately, this does not work for the general case. There, finding the minimum is not simple, and requires computing the generalised eigenvectors of $S^{-1}{}_W S_B$, assuming that $S^{-1}{}_W$ exists. We will be discussing eigenvectors in the next section if you are not sure what they are.

2. Principal Components Analysis (PCA)

- The next few methods that we are going to look at are also involved in **computing transformations** of the data in order to identify a lower-dimensional set of axes.
- However, unlike LDA, they are designed for **unlabelled data**.
- This does not stop them being used **for labelled dat**a, since the learning that takes place in the lower dimensional space can still use the target data, although it does mean that they miss out on any information that is contained in the targets.
- The idea is that by **finding particular sets of coordinate axes**, it will become clear that some of the dimensions are not required.

- This is demonstrated in Figure 6.6, which shows two versions of the same dataset. In the first the data are arranged in an ellipse that runs at 45 to the axes, while in the second the axes have been moved

FIGURE 6.5 Plot of the iris data showing the three classes *left:* before and *right:* after LDA has been applied.

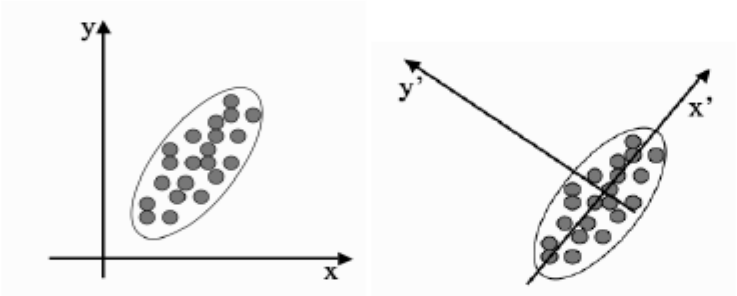


FIGURE 6.6 Two different sets of coordinate axes. The second consists of a rotation and translation of the first and was found using Principal Components Analysis.

So that the data now runs along the x−axis and is centered on the origin.

The potential for dimensionality reduction is in the fact that the y dimension does not now demonstrate much variability, and so it might be possible to ignore it

and use the x axis values alone without compromising the results of a learning algorithm.

In fact, it can make the results better, since we are often removing some of the noise in the data.

- The question is how to choose the axes. The first method we are going to look at is Principal Components Analysis (PCA).
- The idea of a principal component is that it is a **direction in the data with the largest variation.**
- The algorithm first centers the data by **subtracting off the mean**, and then chooses the **direction** with the **largest variation** and places an **axis in that direction**, and then looks at the **variation that remains** and finds **another axis that is orthogonal** to the first and **covers as much of the remaining variation** as possible.
- It then **iterates** this until it has run out of possible axes.
- The **end result** is that all the variation is along the axes of the coordinate set, and so the **covariance matrix is diagonal**—

- Each new variable is uncorrelated with every variable except itself. Some of the **axes that are found last have very little variation**, and so they **can be removed without affecting the variability in the data.**
  Putting this in more formal terms, we have **a data matrix X and we want to rotate it so that the data lies along the directions of maximum variation.** This means that we multiply our data matrix by a **rotation matrix (often written as P**T) so that **Y = PTX**, where **P** is chosen so that the **covariance matrix of Y is diagonal**, i.e.,

$$\mathrm{cov}(\mathbf{Y}) = \mathrm{cov}(\mathbf{P}^T\mathbf{X}) = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_N \end{pmatrix}. \quad (6.7)$$

We can get a different handle on this by using some linear algebra and the definition of covariance to see that:

$$\begin{aligned} \mathrm{cov}(\mathbf{Y}) &= E[\mathbf{Y}\mathbf{Y}^T] && (6.8)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{P}^T\mathbf{X})^T] && (6.9)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{X}^T\mathbf{P})] && (6.10)\\ &= \mathbf{P}^T E(\mathbf{X}\mathbf{X}^T)\mathbf{P} && (6.11)\\ &= \mathbf{P}^T \mathrm{cov}(\mathbf{X})\mathbf{P}. && (6.12) \end{aligned}$$

The two extra things that we needed to know were that $(\mathbf{P}^T\mathbf{X})^T = \mathbf{X}^T\mathbf{P}^{T^T} = \mathbf{X}^T\mathbf{P}$ and that $E[\mathbf{P}] = \mathbf{P}$ (and obviously the same for $\mathbf{P}^T$) since it is not a data-dependent matrix. This then tells us that:

$$\mathbf{P}\mathrm{cov}(\mathbf{Y}) = \mathbf{P}\mathbf{P}^T\mathrm{cov}(\mathbf{X})\mathbf{P} = \mathrm{cov}(\mathbf{X})\mathbf{P}, \quad (6.13)$$

where there is one tricky fact, namely that for a rotation matrix $\mathbf{P}^T = \mathbf{P}^{-1}$. This just says that to invert a rotation we rotate in the opposite direction by the same amount that we rotated forwards.

As $\mathrm{cov}(\mathbf{Y})$ is diagonal, if we write $\mathbf{P}$ as a set of column vectors $\mathbf{P} = [\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_N]$ then:

$$\mathbf{P}\mathrm{cov}(\mathbf{Y}) = [\lambda_1\mathbf{p}_1, \lambda_2\mathbf{p}_2, \dots, \lambda_N\mathbf{p}_N], \quad (6.14)$$

which (by writing the λ variables in a matrix as $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \dots, \lambda_N)^T$ and $\mathbf{Z} = \mathrm{cov}(\mathbf{X})$) leads to a very interesting equation:

$$\boldsymbol{\lambda}\mathbf{p}_i = \mathbf{Z}\mathbf{p}_i \text{ for each } \mathbf{p}_i. \quad (6.15)$$

- At first sight it doesn't look very interesting, but the important thing is to realise that **λ is a column vector,** while Z is a full matrix, and it can be applied to each of the p_i vectors that make up P.
- Since is only a column vector, all it does is rescale the p_{is}; it cannot rotate it or do anything complicated like that. So this tells us that somehow we have found a matrix P so that for the directions that P is written in, the matrix Z does not twist or rotate those directions, but just rescales them.

- These directions are special enough that they have a name: they are **eigen vectors**, and the amount that they rescale the axes (the λ s) by are known as eigen values.
- All **eigen vectors** of a square symmetric matrix A are orthogonal to each other. This tells us that the eigenvectors define a space.
- If we make a matrix E that contains the (normalised) eigen vectors of a matrix A as columns, then this matrix will take any **vector** and **rotate it** into what is known as the **eigen space.**
- Since **E is a rotation matrix**, $\mathbf{E^{-1} = E^T}$, so that rotating the resultant vector back out of the eigen space requires multiplying it by E^T , where by 'normalised', I mean that the eigenvectors are made unit length.
- So what should we do between rotating the vector into the eigen space, and rotating it back out? The answer is that we can stretch the vectors along the axes. This is done by multiplying the vector by a diagonal matrix that has the eigen values along its diagonal, D.
  So we can decompose any square symmetric matrix A into the following set of matrices: $A = EDE^T$, and this is what we have done to our covariance matrix above. This is called the spectral decomposition.
- Before we get on to the algorithm, there is one other useful thing to note.
  - The eigen values tell us how much stretching we need to do along their corresponding eigenvector dimensions.
  - The more of this rescaling is needed, the larger the variation along that dimension (since if the data was already spread out equally then the eigen value would be close to 1), and so the dimensions with large eigen values have lots of variation and are therefore useful dimensions, while for those with small eigen values, all the data points are very tightly bunched together, and there is not much variation in that direction.
  - This means that we can throw away dimensions where the eigen values are very small (usually smaller than some chosen parameter) It is time to see the algorithm that we need.

The Principal Components Analysis Algorithm

- write N datapoints $\mathbf{x}_i = (\mathbf{x}_{1i}, \mathbf{x}_{2i}, \ldots, \mathbf{x}_{Mi})$ as row vectors
- put these vectors into a matrix $\mathbf{X}$ (which will have size $N \times M$)
- centre the data by subtracting off the mean of each column, putting it into matrix $\mathbf{B}$
- compute the covariance matrix $\mathbf{C} = \frac{1}{N}\mathbf{B}^T\mathbf{B}$
- compute the eigenvalues and eigenvectors of $\mathbf{C}$, so $\mathbf{V}^{-1}\mathbf{C}\mathbf{V} = \mathbf{D}$, where $\mathbf{V}$ holds the eigenvectors of $\mathbf{C}$ and $\mathbf{D}$ is the $M \times M$ diagonal eigenvalue matrix
- sort the columns of $\mathbf{D}$ into order of decreasing eigenvalues, and apply the same order to the columns of V
- reject those with eigenvalue less than some η, leaving L dimensions in the data

- Two different examples of using PCA are shown in Figures 6.7 and 6.8. The former shows two-dimensional data from an ellipse being mapped into one principal component, which lies along the principal axis of the ellipse. Figure 6.8 shows the first two dimensions of the iris data, and shows that the three classes are clearly distinguishable after PCA has been applied.

Relation with the Multi-layer Perceptron

- PCA can be used in the SOM algorithm to initialize the weights, thus reducing the amount of learning that is required, and that it is very useful for dimensionality reduction.
- However, there is another reason why people who are interested in neural networks are interested in PCA the auto-associative MLP. The auto-associative MLP actually computes something very similar to the principal components of the data in the hidden nodes, and this is one of the ways that we can understand what the network is doing.
- Of course, computing the principal components with a neural network isn't necessarily a good idea. PCA is linear (it just rotates and translates the axes, it can't do anything more complicated). This is clear if we think about the network, since it is the hidden nodes that are computing PCA, and they are effectively a bit like a Perceptron—they can only perform linear tasks. It is the extra layers of neurons that allow us to do more.

FIGURE 6.8 Plot of the first two principal components of the iris data, showing that the three classes are clearly distinguishable.

- So suppose we do just that and use a more complicated MLP network with four layers of neurons instead of two.
- We still use it as an auto-associator, so that the targets are the same as the inputs. What will the middle hidden layer look like then?
- A full answer is complicated, but we can speculate that the first layer is computing some non-linear transformation of the data, while the second (bottleneck) layer is computing the PCA of those non-linear functions. Then the third layer reconstructs the data, which appears again in the fourth layer. So the network is still doing PCA, just on a non-linear version of the inputs.
- This might be useful, since now we are not assuming that the data are linearly separable.

Kernel PCA

- One **problem** with PCA is that **it assumes that the directions of variation are all straight lines**. This is often **not true**.
- We can use the auto-associator with multiple hidden layers as just discussed, but there is a very nice extension to PCA that uses the **kernel trick** (which is described in Section 8.2) to get around this problem, just as the SVM got around it for the Perceptron.
- Just as is done there, we apply a (**possibly non-linear) function (·) to each data point x** that **transforms** the data into the **kernel space**, and then perform **normal linear PCA** in that space. The covariance matrix is defined in the kernel space and is:

$$\mathbf{C} = \frac{1}{N}\sum_{n=1}^{N}\Phi(\mathbf{x}_n)\Phi(\mathbf{x}_n)^T, \tag{6.16}$$

which produces the eigenvector equation:

$$\lambda\left(\Phi(\mathbf{x}_i)\mathbf{V}\right) = \left(\Phi(\mathbf{x}_i)\mathbf{C}\mathbf{V}\right) \;\; i = 1\ldots N, \tag{6.17}$$

where $\mathbf{V} = \sum_{j=1}^{N}\boldsymbol{\alpha}_j\Phi(\mathbf{x}_j)$ are the eigenvectors of the original problem and the $\boldsymbol{\alpha}_j$ will turn out to be the eigenvectors of the 'kernelized' problem. It is at this point that we can apply the kernel trick and produce an $N \times N$ matrix $\mathbf{K}$, where:

$$\mathbf{K}_{(i,j)} = \left(\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)\right). \tag{6.18}$$

Putting these together we get the equation $N\lambda\mathbf{K}\boldsymbol{\alpha} = \mathbf{K}^2\boldsymbol{\alpha}$, and we left-multiply by $\mathbf{K}^{-1}$ to reduce it to $N\lambda\boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha}$. Computing the projection of a new point $\mathbf{x}$ into the kernel PCA space requires:

$$\left(\mathbf{V}^k\cdot\Phi(\mathbf{x})\right) = \sum_{i=1}^{N}\boldsymbol{\alpha}_i^k\left(\Phi(\mathbf{x}_i)\cdot\Phi(\mathbf{x}_j)\right). \tag{6.19}$$

This is all there is to the algorithm.

The Kernel PCA Algorithm

- Choose a kernel and apply it to all pairs of points to get matrix $\mathbf{K}$ of distances between the pairs of points in the transformed space
- Compute the eigenvalues and eigenvectors of $\mathbf{K}$
- Normalise the eigenvectors by the square root of the eigenvalues
- Retain the eigenvectors corresponding to the largest eigenvalues

---

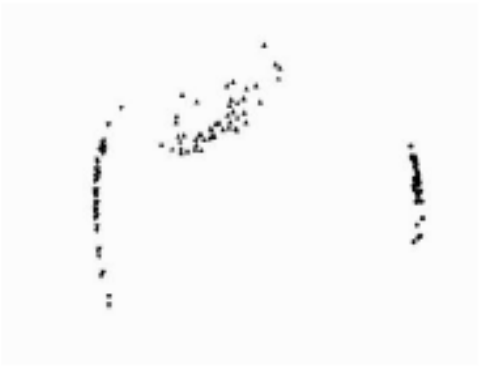


FIGURE 6.9 Plot of the first two non-linear principal components of the iris data, (using the Gaussian kernel) showing that the three classes are clearly distinguishable.

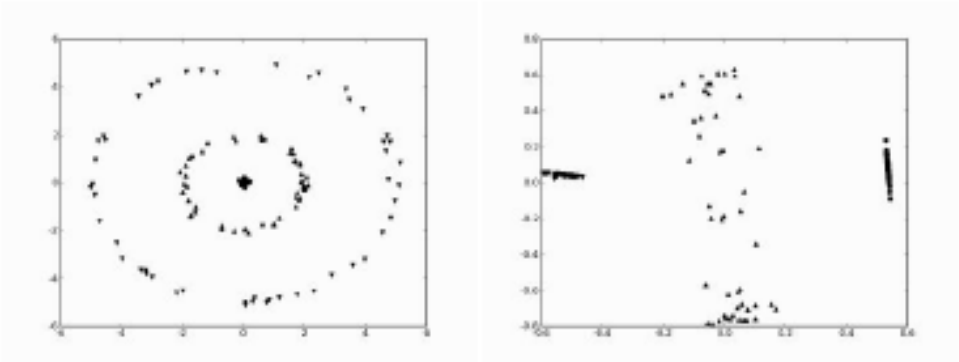


FIGURE 6.10 A very definitely non-linear dataset consisting of three concentric circles, and the (Gaussian) kernel PCA mapping of the iris data, which requires only one component to separate the data.

- Figure 6.9 shows the **output of kernel** PCA when applied to the **iris dataset.**
- The fact that it can separate this data well is not very surprising since the linear methods that we have already seen can do it, but it is a useful check of the method.
- A rather more difficult example is shown in Figure 6.10.
- Data are sampled from three concentric circles.
- Clearly, **linear PCA would not be able to separate this data**, but **applying kernel PCA** to this example separates the data using **only one component.**