CHAPTER 6

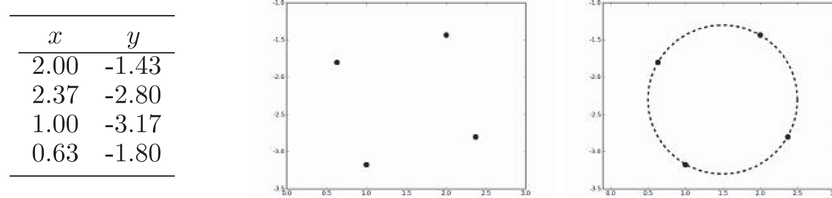
Dimensionality Reduction

When looking at data and plotting results, we can never go beyond three dimensions in our data, and usually find two dimensions easier to interpret. In addition, we have already seen the curse of dimensionality (Section 2.1.2) 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. 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 extreme cases such as the Self-Organising Map that we will see in Section 14.3, where the number of dimensions becomes three or fewer, we can also plot the data, which makes it much easier to understand and interpret.

With this many good things to say about dimensionality reduction, clearly it is something that we need to understand. The importance of the field for machine learning and other forms of data analysis can be seen from the fact that in the year 2000 there were three articles related to dimensionality reduction published together in the prestigious journal *Science*. At the end of the chapter we are going to see two of the algorithms that were described in those papers: Locally Linear Embedding and Isomap.

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. While many people use neural networks precisely because they don't want to 'get their hands dirty' and look at the data themselves, as we have already seen, the results will be better if you check for correlations and other simple things before using the neural network or other learning algorithm. 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 datapoints, 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 $(\pi/6, 4\pi/6, 7\pi/6, 11\pi/6)$) and using this one coordinate, the angle, makes the data a lot easier to understand and analyse.



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 that we will see in this chapter 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 (Chapter 12) 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 $2^d - 1$ possible sets of features to search over, from any individual feature up to the full set. In general, greedy methods (Section 9.4) are employed, although backtracking can also be employed to check whether the search gets stuck.

Many of the algorithms that we will see in this chapter are unsupervised. The disadvantage of this is that we are not then able to use the knowledge of their classes in order to reduce the problem further. However, we will start off by considering 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.

6.1 LINEAR DISCRIMINANT ANALYSIS (LDA)

Figure 6.2 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, $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, the mean of the entire dataset ($\boldsymbol{\mu}$), and the covariance of each class with itself (see Section 2.4.2 for a description of covariance), which is $\sum_j (\mathbf{x}_j - \boldsymbol{\mu})(\mathbf{x}_j - \boldsymbol{\mu})^T$. 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 p_c, the probability of the class (that is, the number of datapoints 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:

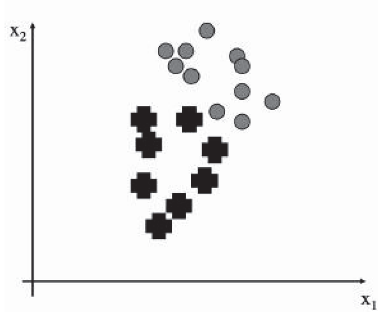


FIGURE 6.2 A set of datapoints in two dimensions, with two classes.

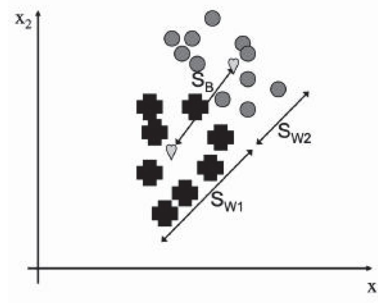


FIGURE 6.3 The meaning of the between-class and within-class scatter. The hearts mark the means of the two classes.

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

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 6.3. 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 S_B/S_W 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 S_B/S_W as large as possible is something that we want to be true for our data when we reduce the number of dimensions. Figure 6.4 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.

Remember from Chapter 3 that any line can be written as a vector $\mathbf{w}$ (which we used as our weight vector in Section 3.4; it is one row of weight matrix $\mathbf{W}$). The projection of the data can be written as $z = \mathbf{w}^T \cdot \mathbf{x}$ for datapoint $\mathbf{x}$. This gives us a scalar that is the distance along the $\mathbf{w}$ vector that we need to go to find the projection of point $\mathbf{x}$. To see this, remember that $\mathbf{w}^T \cdot \mathbf{x}$ is the sum of the vectors multiplied together element-wise, and is equal to the size of $\mathbf{w}$ times the size of $\mathbf{x}$ times the cosine of the angle between them. We can make the size of $\mathbf{w}$ be 1, so that we don't have to worry about it, and all that is then described is the amount of $\mathbf{x}$ that lies along $\mathbf{w}$.

So we can compute the projection of our data along $\mathbf{w}$ 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: $\mu'_c = \mathbf{w}^T \cdot \boldsymbol{\mu}_c$. Now we just need to work out what happens to the within-class and between-class scatters.

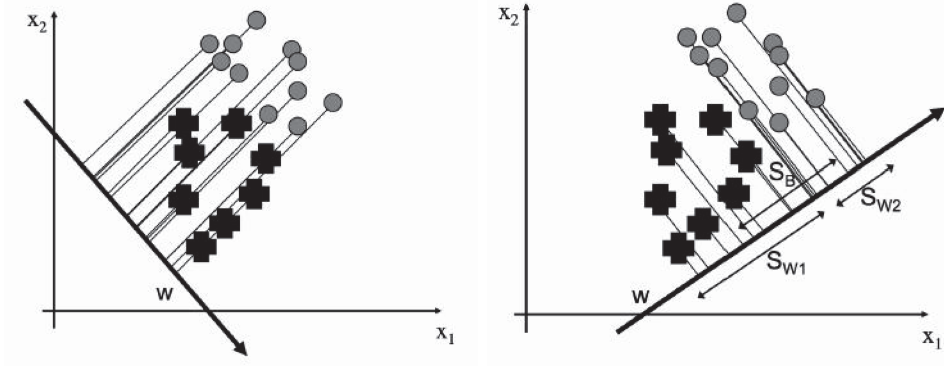


FIGURE 6.4 Two different possible projection lines. The one on the left fails to separate the classes.

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:

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

If there are only two classes in the data, then we can rewrite Equation (6.2) as $S_B = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$. To see this, consider that there are N_1 examples of class 1 and N_2 examples of class 2. Then substitute $(N_1 + N_2)\mu = N_1\mu_1 + N_2\mu_2)$ into Equation (6.2). The rewritten $S_B\mathbf{w}$ is in the direction $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$, and so $\mathbf{w}$ is in the direction of $S_W^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$, as can be seen by recalling that the scalar product is independent of order, and so after substituting in the new expression for S_B, the order of the bracketed terms can be changed. Note that we can ignore the ratio of within-class and between-class scatter, since it is a scalar and therefore does not affect the direction of the vector.

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

Turning this into an algorithm is very simple. You simply have to compute the between-class and within-class scatters, and then find the value of $\mathbf{w}$. In NumPy, the entire algorithm can be written as (where the generalised eigenvectors are computed in SciPy rather than NumPy, which was imported using `from scipy import linalg as la`):

```
C = np.cov(np.transpose(data))

# Loop over classes
classes = np.unique(labels)
for i in range(len(classes)):
    # Find relevant datapoints
    indices = np.squeeze(np.where(labels==classes[i]))
    d = np.squeeze(data[indices,:])
    classcov = np.cov(np.transpose(d))
    Sw += np.float(np.shape(indices)[0])/nData * classcov

Sb = C - Sw
# Now solve for W and compute mapped data
# Compute eigenvalues, eigenvectors and sort into order
evals,evecs = la.eig(Sw,Sb)
indices = np.argsort(evals)
indices = indices[::-1]
evecs = evecs[:,indices]
evals = evals[indices]
w = evecs[:,:redDim]
newData = np.dot(data,w)
```

As an example of using the algorithm, Figure 6.5 shows a plot of the first two dimensions of the iris data (with the classes shown as three different symbols) before and after applying LDA, with the number of dimensions being set to two. While one of the classes (the circles) can already be separated from the others, all three are readily distinguishable after LDA has been applied (and only one dimension, the y one, is required for this).

6.2 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 data, 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 so that the data now runs along the $x-$axis and is centred 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

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

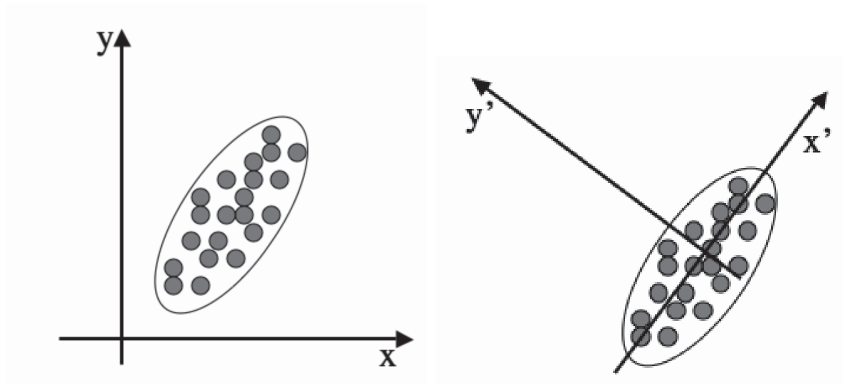


FIGURE 6.6 Two different sets of coordinate axes. The second consists of a rotation and translation of the first and was found using Principal Components Analysis.

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 centres 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 $\mathbf{X}$ 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 $\mathbf{P}^T$) so that $\mathbf{Y} = \mathbf{P}^T\mathbf{X}$, where $\mathbf{P}$ is chosen so that the covariance matrix of $\mathbf{Y}$ is diagonal, i.e.,

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

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

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

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:

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

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 $\text{cov}(\mathbf{Y})$ 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:

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

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

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

At first sight it doesn't look very interesting, but the important thing is to realise that $\boldsymbol{\lambda}$ is a column vector, while $\mathbf{Z}$ is a full matrix, and it can be applied to each of the $\mathbf{p}_i$ vectors that make up $\mathbf{P}$. Since $\boldsymbol{\lambda}$ is only a column vector, all it does is rescale the $\mathbf{p}_i$s; it cannot rotate it or do anything complicated like that. So this tells us that somehow we have found a matrix $\mathbf{P}$ so that for the directions that $\mathbf{P}$ is written in, the matrix $\mathbf{Z}$ does not twist or rotate those directions, but just rescales them. These directions are special enough that they have a name: they are eigenvectors, and the amount that they rescale the axes (the $\boldsymbol{\lambda}$s) by are known as eigenvalues.

All eigenvectors of a square symmetric matrix $\mathbf{A}$ are orthogonal to each other. This tells us that the eigenvectors define a space. If we make a matrix $\mathbf{E}$ that contains the (normalised) eigenvectors of a matrix $\mathbf{A}$ as columns, then this matrix will take any vector and rotate it into what is known as the eigenspace. Since $\mathbf{E}$ is a rotation matrix, $\mathbf{E}^{-1} = \mathbf{E}^T$, so that rotating the resultant vector back out of the eigenspace requires multiplying it by $\mathbf{E}^T$, where by 'normalised', I mean that the eigenvectors are made unit length. So what should we do between rotating the vector into the eigenspace, 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

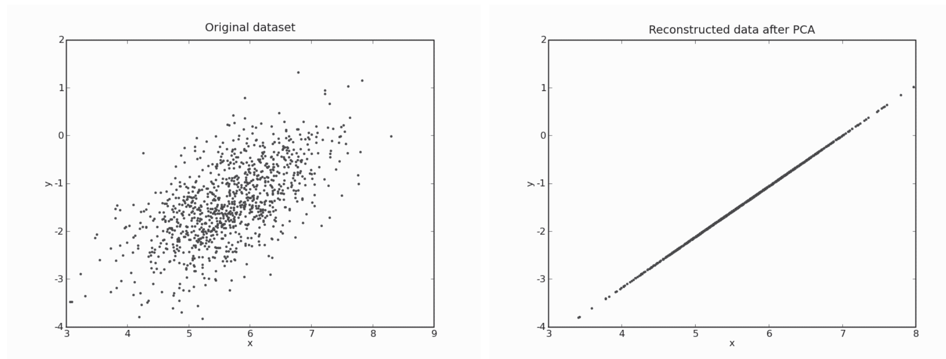


FIGURE 6.7 Computing the principal components of the 2D dataset on the left and using only the first one to reconstruct it produces the line of data shown on the right, which is along the principal axis of the ellipse that the data was sampled from.

diagonal matrix that has the eigenvalues along its diagonal, $\mathbf{D}$. So we can decompose any square symmetric matrix $\mathbf{A}$ into the following set of matrices: $\mathbf{A} = \mathbf{EDE}^T$, 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 eigenvalues 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 eigenvalue would be close to 1), and so the dimensions with large eigenvalues have lots of variation and are therefore useful dimensions, while for those with small eigenvalues, all the datapoints are very tightly bunched together, and there is not much variation in that direction. This means that we can throw away dimensions where the eigenvalues 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 $\mathbf{V}^{-1}\mathbf{CV} = \mathbf{D}$, 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

NumPy can compute the eigenvalues and eigenvectors for us. They are both returned in `evals,evecs = np.linalg.eig(x)`. This makes the entire algorithm fairly easy to implement:

```
def pca(data,nRedDim=0,normalise=1):

    # Centre data
    m = np.mean(data,axis=0)
    data -= m

    # Covariance matrix
    C = np.cov(np.transpose(data))

    # Compute eigenvalues and sort into descending order
    evals,evecs = np.linalg.eig(C)
    indices = np.argsort(evals)
    indices = indices[::-1]
    evecs = evecs[:,indices]
    evals = evals[indices]

    if nRedDim>0:
        evecs = evecs[:,:nRedDim]

    if normalise:
        for i in range(np.shape(evecs)[1]):
            evecs[:,i] / np.linalg.norm(evecs[:,i]) * np.sqrt(evals[i])

    # Produce the new data matrix
    x = np.dot(np.transpose(evecs),np.transpose(data))
    # Compute the original data again
    y=np.transpose(np.dot(evecs,x))+m
    return x,y,evals,evecs
```

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.

6.2.1 Relation with the Multi-layer Perceptron

We will see (in Section 14.3.2) that PCA can be used in the SOM algorithm to initialise 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. We already mentioned it when we talked about the auto-associative MLP in Section 4.4.5. 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

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

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.

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. However, to understand it better we will look at it from a different viewpoint, thinking of the actions of the first layer as kernels, which are described in Section 8.2.

6.2.2 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 $\Phi(\cdot)$ to each datapoint $\mathbf{x}$ 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:

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

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

---

The only tricky part of the implementation is in the diagonalisation of $\mathbf{K}$, which is generally done using some well-known linear algebra identities, leading to:

```
K = kernelmatrix(data,kernel)

# Compute the transformed data
D = np.sum(K,axis=0)/nData
E = np.sum(D)/nData
J = np.ones((nData,1))*D
K = K - J - np.transpose(J) + E*np.ones((nData,nData))

# Perform the dimensionality reduction
evals,evecs = np.linalg.eig(K)
indices = np.argsort(evals)
indices = indices[::-1]
evecs = evecs[:,indices[:redDim]]
evals = evals[indices[:redDim]]

sqrtE = np.zeros((len(evals),len(evals)))
for i in range(len(evals)):
    sqrtE[i,i] = np.sqrt(evals[i])

newData = np.transpose(np.dot(sqrtE,np.transpose(evecs)))
```

This is a computationally expensive algorithm, since it requires computing the kernel

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.

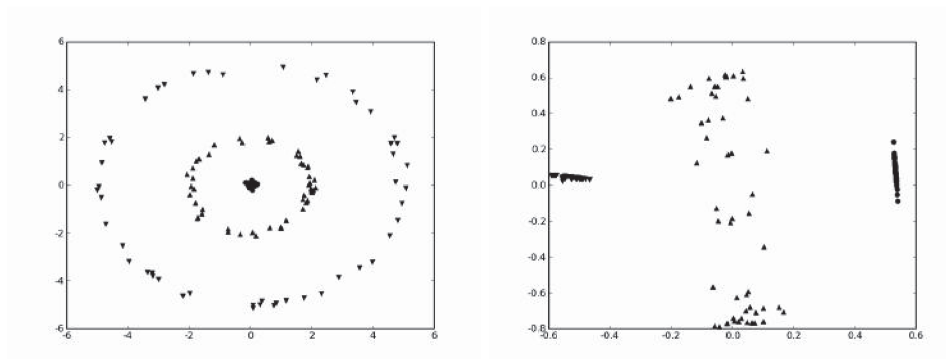


FIGURE 6.10 A very definitely non-linear dataset consisting of three concentric circles, and the (Gaussian) kernel PCA mapping of the iris data, which requires only one component to separate the data.

matrix and then the eigenvalues and eigenvectors of that matrix. The naïve implementation on the algorithm on the website is $\mathcal{O}(n^3)$, but with care it is possible to take advantage of the fact that not all of the eigenvalues are needed, which can lead to an $\mathcal{O}(n^2)$ algorithm.

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.

6.3 FACTOR ANALYSIS

The idea of factor analysis is to ask whether the data that is observed can be explained by a smaller number of uncorrelated factors or latent variables. The assumption is that the data comes from some underlying data source (or set of data sources) that are not directly known. The problem of factor analysis is to find those independent factors, and the noise that is inherent in the measurements of each factor. Factor analysis is commonly used in psychology and other social sciences, and the factors are generally chosen to have some particular meanings: in psychology, they can be related to IQ and other tests.

Suppose that we have a dataset in the usual $N \times M$ matrix $\mathbf{X}$, i.e., each row of $\mathbf{X}$ is an M-dimensional datapoint, and $\mathbf{X}$ has covariance matrix $\mathbf{\Sigma}$. As, with PCA, we centre the data by subtracting off the mean of each variable (i.e., each column): $\mathbf{b}_j = \mathbf{x}_j - \boldsymbol{\mu}_j$, $j = 1 \dots M$, so that the mean $E[\mathbf{b}_i] = 0$. Which we've done before, for example for the MLP and many times since.

We can write the model that we are assuming as:

$$\mathbf{X} = \mathbf{WY} + \boldsymbol{\epsilon}, \tag{6.20}$$

where $\mathbf{X}$ are the observations and $\boldsymbol{\epsilon}$ is the noise. Since the factors $\mathbf{b}_i$ that we want to find should be independent, so $\text{cov}(\mathbf{b}_i, \mathbf{b}_j) = 0$ if $i \neq j$. Factor analysis takes explicit notice of the noise in the data, using the variable $\boldsymbol{\epsilon}$. In fact, it assumes that the noise is Gaussian with zero mean and some known variance: Ψ, with the variance of each element being $\Psi_i = \text{var}(\epsilon_i)$. It also assumes that these noise measurements are independent of each other, which is equivalent to the assumption that the data come from a set of separate (independent) physical processes, and seems reasonable if we don't know otherwise.

The covariance matrix of the original data, $\mathbf{\Sigma}$, can now be broken down into $\text{cov}(\mathbf{Wb} + \boldsymbol{\epsilon}) = \mathbf{WW}^T + \Psi$, where Ψ is the matrix of noise variances and we have used the fact that $\text{cov}(\mathbf{b}) = \mathbf{I}$ since the factors are uncorrelated.

With all of that set up, the aim of factor analysis is to try to find a set of factor loadings $\mathbf{W}_{ij}$ and values for the variance of the noise parameters Ψ, so that the data in $\mathbf{X}$ can be reconstructed from the parameters, or so that we can perform dimensionality reduction.

Since we are looking at adding additional variables, the natural formulation is an EM algorithm (as described in Section 7.1.1) and this is how the computations are usually performed to produce the maximum likelihood estimate. Getting to the EM algorithm takes some effort. We first define the log likelihood (where $\boldsymbol{\theta}$ is the data we are trying to fit) as:

$$Q(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \int p(\mathbf{x}|\mathbf{y}, \boldsymbol{\theta}_{t-1}) \log(p(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_t)p(\mathbf{x}))d\mathbf{x}. \tag{6.21}$$

We can replace several of the terms in here with values, and we can also ignore any terms that do not depend on $\boldsymbol{\theta}$. The end result of this is a new version of Q, which forms the basis of the E-step:

$$Q(\boldsymbol{\theta}_t|\boldsymbol{\theta}_{t-1}) = \frac{1}{2} \int p(\mathbf{x}|\mathbf{y}, \boldsymbol{\theta}_{t-1}) \log(\det(\Psi^{-1})) - \left(\mathbf{y} - W\mathbf{x}\right)^T \Psi^{-1} (\mathbf{y} - W\mathbf{x})\big) \, d\mathbf{x}. \tag{6.22}$$

For the EM algorithm we now have to differentiate this with respect to $\mathbf{W}$ and the individual elements of Ψ, and apply some linear algebra, to get update rules:

$$\mathbf{W}_{new} = \left(\mathbf{y}E(\mathbf{x}|\mathbf{y})^T\right)\left(E(\mathbf{x}\mathbf{x}^T|\mathbf{y})\right)^{-1}, \tag{6.23}$$

$$\Psi_{new} = \frac{1}{N}\text{diagonal}\left(\mathbf{x}\mathbf{x}^T - WE(\mathbf{x}|\mathbf{y})\mathbf{y}^T\right), \tag{6.24}$$

where diagonal() ensures that the matrix retains values only on the diagonal and the expectations are:

$$E(\mathbf{x}|\mathbf{y}) = \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}\mathbf{b} \tag{6.25}$$

$$E(\mathbf{x}\mathbf{x}^T|\mathbf{x}) - E(\mathbf{x}|\mathbf{y})E(\mathbf{x}|\mathbf{y})^T = \mathbf{I} - \mathbf{W}^T(\mathbf{W}\mathbf{W}^T + \Psi)^{-1}W. \tag{6.26}$$

The only other things that we need to add to the algorithm is some way to decide when to stop, which involves computing the log likelihood and stopping the algorithm when it stops descending. This leads to an algorithm where the basic steps in the loop are:

```
# E-step
A = np.dot(W,np.transpose(W)) + np.diag(Psi)
logA = np.log(np.abs(np.linalg.det(A)))
A = np.linalg.inv(A)

WA = np.dot(np.transpose(W),A)
WAC = np.dot(WA,C)
Exx = np.eye(nRedDim) - np.dot(WA,W) + np.dot(WAC,np.transpose(WA))

# M-step
W = np.dot(np.transpose(WAC),np.linalg.inv(Exx))
Psi = Cd - (np.dot(W,WAC)).diagonal()

tAC = (A*np.transpose(C)).sum()

L = -N/2*np.log(2.*np.pi) -0.5*logA - 0.5*tAC
if (L-oldL)<(1e-4):
    print "Stop",i
    break
```

The output of using factor analysis on the iris dataset are shown in Figure 6.11.

6.4 INDEPENDENT COMPONENTS ANALYSIS (ICA)

There is a related approach to factor analysis that is known as Independent Components Analysis. When we looked at PCA above, the components were chosen so that they were orthogonal and uncorrelated (so that the covariance matrix was diagonal, i.e., so $\text{cov}(\mathbf{b}_i, \mathbf{b}_j) = 0$ if $i \neq j$). If, instead, we require that the components are statistically independent (so that for $E[\mathbf{b}_i, \mathbf{b}_j] = E[\mathbf{b}_i]E[\mathbf{b}_j]$ as well as the $\mathbf{b}_i$ being uncorrelated), then we get ICA.

The common motivation for ICA is the problem of blind source separation. As with factor analysis, the assumption is that the data we see are actually created by a set of underlying

FIGURE 6.11 Plot of the first two factor analysis components of the iris data, showing that the three classes are clearly distinguishable.

physical processes that are independent. The reason why the data we see are correlated is because of the way the outputs from different processes have been mixed together. So given some data, we want to find a transformation that turns it into a mixture of independent sources or components.

The most popular way to describe blind source separation is known as the cocktail party problem. If you are at a party, then your ears hear lots of different sounds coming from lots of different locations (different people talking, the clink of glasses, background music, etc.) but you are somehow able to focus on the voice of the people you are talking to, and can in fact separate out the sounds from all of the different sources even though they are mixed together. The cocktail party problem is the challenge of separating out these sources, although there is one wrinkle: for the algorithm to work, you need as many ears as there are sources. This is because the algorithm does not have the information we have about what things sound like.

Suppose that we have two sources making noise (s_1^t, s_2^t) where the top index covers the fact that there are lots of datapoints appearing over time, and two microphones that hear things, giving inputs (x_1^t, x_2^t). The sounds that are heard come from the sources as:

$$x_1 = as_1 + bs_2, \tag{6.27}$$
$$x_2 = cs_1 + ds_2, \tag{6.28}$$

which can be written in matrix form as:

$$\mathbf{x} = \mathbf{A}\mathbf{s}, \tag{6.29}$$

where $\mathbf{A}$ is known as the mixing matrix. Reconstructing $\mathbf{s}$ looks easy now: we just compute $\mathbf{s} = \mathbf{A}^{-1}\mathbf{x}$. Except that, unfortunately, we don't know $\mathbf{A}$. The approximation to $\mathbf{A}^{-1}$ that we work out is generally labelled as $\mathbf{W}$, and it is a square matrix since we have the same number of microphones as we do sources.

At this point we need to work out what we actually know about the sources and the signals. There are three things:

- the mixtures are not independent, even though the sources are

- the mixtures will look like normal distributions even if the sources are not (this is because of the Central Limit Theorem, something that we won't look at further here)
- the mixtures will look more complicated than the sources

We can use the first fact to say that if we find factors that are independent of each other then they are probably sources, and the second to say that if we find factors that are not Gaussian then they are probably sources. We can measure the amount of independence between two variables by using the mutual information, which we will see in Section 12.2.1 when we look at entropy. In fact, the most common approach is to use what is rather uglily known as negentropy: $J(y) = H(z) - H(y)$, which maximises the deviations from Gaussianness (where $H(\cdot)$ is the entropy):

$$H(y) = -\int g(y) \log g(y) dy. \tag{6.30}$$

One common approximation is $J(y) = (E[G(y)] - E[G(z)])^2$, where $g(u) = \frac{1}{a} \log \cosh(au)$, so $g'(u) = \tanh(au)$ $1 \leq a \leq 2$. Implementing ICA is actually quite tricky because of some numerical issues, so we won't do it ourselves. There are a few well-used ICA implementations out there, of which the most popular is known as FastICA, which is available in Python as part of the MDP package.

6.5 LOCALLY LINEAR EMBEDDING

Two relatively recent methods of computing dimensionality reduction were mentioned in the introduction because they were published in the journal *Science*. Both are non-linear, and both attempt to preserve the neighbourhood relations in the data (as will be discussed for the SOM in Section 14.3) but they use different approaches. The first tries to approximate the data by sticking together sets of locally flat patches that cover the dataset, while the second uses the shortest distances (geodesics) on the non-linear space to find a globally optimal solution.

We will look first at the locally linear algorithm, which is called Locally Linear Embedding (LLE). It was introduced by Roweis and Saul in 2000. The idea is to say that by making linear approximations we will make some errors, so we should make these errors as small as possible by making the patches small where there is lots of non-linearity in the data. The error is known as the reconstruction error and is simply the sum-of-squares of the distance between the original point and its reconstruction:

$$\epsilon = \sum_{i=1}^{N} \left(\mathbf{x}_i - \sum_{j=1}^{N} \mathbf{W}_{ij} \mathbf{x}_j \right)^2 . \tag{6.31}$$

The weights $\mathbf{W}_{ij}$ say how much effect the jth datapoint has on the reconstruction of the ith one. The question is which points can be usefully used to reconstruct a particular datapoint. If another point is a long way off, then it probably isn't very useful: only those points that are close to the current datapoint (that are in its neighbourhood) are used. There are two common ways to create neighbourhoods:

- Points that are less than some predefined distance d to the current point are neighbours (so we don't know how many neighbours there are, but they are all close)
- The k nearest points are neighbours (so we know how many there are, but some could be far away)

Solving for the weights $\mathbf{W}_{ij}$ is a least-squares problem, which we can simplify by enforcing the constraints that for any point $\mathbf{x}_j$ that is a long way from the current point $\mathbf{x}_i$, $\mathbf{W}_{ij} = 0$, and that $\sum_j \mathbf{W}_{ij} = 1$. This produces a reconstruction of the data, but it does not reduce the dimensionality at all. For this we have to reapply the same basic cost function, but minimise it according to the positions $\mathbf{y}_i$ of the points in some lower dimensional space (dimension L):

$$\mathbf{y}_i = \sum_{i=1}^{N} \left(\mathbf{y}_i - \sum_{j=1}^{L} \mathbf{W}_{ij}\mathbf{y}_j \right)^2 . \tag{6.32}$$

Solving this is rather more complicated, so we won't go into details, but it turns out that the solution is the eigenvalues of the quadratic form matrix $\mathbf{M}_{ij} = \delta_{ij} - \mathbf{W}_{ij} - \mathbf{W}_{ji} + \sum_k \mathbf{W}_{ji}\mathbf{W}_{kj}$, where δ_{ij} is the Kronecker delta function, so $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. This leads to the following algorithm:

The Locally Linear Embedding Algorithm

- Decide on the neighbours of each point (e.g., K nearest neighbours):
  - compute distances between every pair of points
  - find the k smallest distances
  - set $\mathbf{W}_{ij} = 0$ for other points
  - for each point $\mathbf{x}_i$:
    - create a list of its neighbours' locations $\mathbf{z}_i$
    - compute $\mathbf{z}_i = \mathbf{z}_i - \mathbf{x}_i$
- Compute the weights matrix $\mathbf{W}$ that minimises Equation (6.31) according to the constraints:
  - compute local covariance $\mathbf{C} = \mathbf{Z}\mathbf{Z}^T$, where $\mathbf{Z}$ is the matrix of $\mathbf{z}_i$s
  - solve $\mathbf{CW} = \mathbf{I}$ for $\mathbf{W}$, where $\mathbf{I}$ is the $N \times N$ identity matrix
  - set $\mathbf{W}_{ij} = 0$ for non-neighbours
  - set other elements to $\mathbf{W}/\sum(\mathbf{W})$
- Compute the lower dimensional vectors $\mathbf{y}_i$ that minimise Equation (6.32):
  - create $\mathbf{M} = (\mathbf{I} - \mathbf{W})^T(\mathbf{I} - \mathbf{W})$
  - compute the eigenvalues and eigenvectors of $\mathbf{M}$
  - sort the eigenvectors into order by size of eigenvalue
  - set the qth row of $\mathbf{y}$ to be the $q+1$ eigenvector corresponding to the qth smallest eigenvalue (ignore the first eigenvector, which has eigenvalue 0)

There are a couple of things in there that are a bit tricky to implement, and there is a function that we haven't used before, `np.kron()`, which takes two matrices and multiplies each element of the first one by all the elements of the second, putting all of the results together into one multi-dimensional output array. It is used to construct the set of neighbourhood locations for each point.

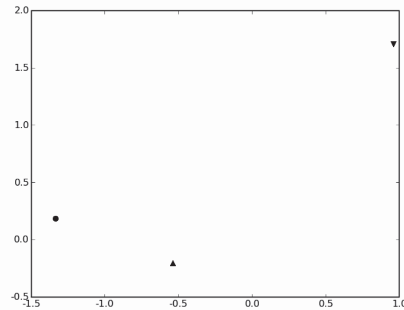


FIGURE 6.12 The Locally Linear Embedding algorithm with $k = 12$ neighbours transforms the iris dataset into three points, separating the data perfectly.

```
for i in range(ndata):
    Z  = data[neighbours[i,:],:] - np.kron(np.ones((K,1)),data[i,:])
    C = np.dot(Z,np.transpose(Z))
    C = C+np.identity(K)*1e-3*np.trace(C)
    W[:,i] = np.transpose(np.linalg.solve(C,np.ones((K,1))))
    W[:,i] = W[:,i]/np.sum(W[:,i])

M = np.eye(ndata,dtype=float)
for i in range(ndata):
    w = np.transpose(np.ones((1,np.shape(W)[0]))*np.transpose(W[:,i]))
    j = neighbours[i,:]
    #print shape(w), np.shape(np.dot(w,np.transpose(w))), np.shape(M[i,j])
    ww = np.dot(w,np.transpose(w))
    for k in range(K):
        M[i,j[k]] -= w[k]
        M[j[k],i] -= w[k]
        for l in range(K):
             M[j[k],j[l]] += ww[k,l]

evals,evecs = np.linalg.eig(M)
ind = np.argsort(evals)
y = evecs[:,ind[1:nRedDim+1]]*np.sqrt(ndata)
```

The LLE algorithm produces a very interesting result on the iris dataset: it separates the three groups into three points (Figure 6.12). This shows that the algorithm works very well on this type of data, but doesn't give us any hints as to what else it can do. Figure 6.13 shows a common demonstration dataset for these algorithms. Known as the swissroll for obvious reasons, it is tricky to find a 2D representation of the 3D data because it is rolled up. The right of Figure 6.13 shows that LLE can successfully unroll it.

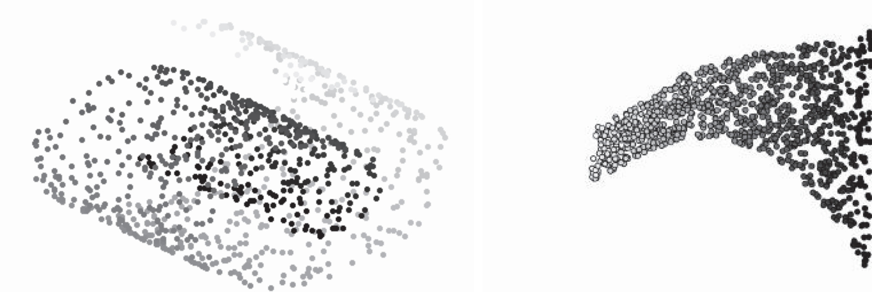


FIGURE 6.13 A common example used to demonstrate LLE is the swissroll dataset shown on the left. To produce a useful 2D representation of this data requires unrolling the data, which the LLE does successfully, as is shown on the right. The shades are used to identify neighbouring points, and do not have any other purpose.

6.6 ISOMAP

The other algorithm was proposed by Tenenbaum et al., also in 2000. It tries to minimise the global error by looking at all of the pairwise distances and computing global geodesics. It is a variant of the standard multi-dimensional scaling (MDS) algorithm, so we'll talk about that first.

6.6.1 Multi-Dimensional Scaling (MDS)

Like PCA, MDS tries to find a linear approximation to the full dataspace that embeds the data into a lower dimensionality. In the case of MDS the embedding tries to preserve the distances between all pairs of points (however these distances are measured). It turns out that if the space is Euclidean, then the two methods are identical. We use the same notational setup as previously, starting with datapoints $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N \in \mathbb{R}^M$. We choose a new dimensionality $L < M$ and compute the embedding so that the datapoints are $\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N \in \mathbb{R}^L$. As usual, we need a cost function to minimise. There are lots of choices for MDS cost functions, but the more common ones are:

Kruskal–Shephard scaling (also known as least-squares) $S_{KS}(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N) = \sum_{i \neq i'} (d_{ii'} - \|\mathbf{z}_i - \mathbf{z}'_i\|)^2$

Sammon mapping $S_{SM}(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N) = \sum_{i \neq i'} \frac{(d_{ii'} - \|\mathbf{z}_i - \mathbf{z}'_i\|)^2}{d_{ii'}}$. This puts more weight onto short distances, so that neighbouring points stay the correct distance apart.

In either case, gradient descent can be used to minimise the distances. There is another version of MDS called classical MDS that uses similarities between datapoints rather than distances. These can be constructed from a set of distances by using the centred inner product $s_{ii'} = (\mathbf{x}_i - \bar{\mathbf{x}}), (\mathbf{x}'_i - \bar{\mathbf{x}})^T$. By doing this it is possible to construct a direct algorithm that does not have to use gradient descent. The function that needs to be minimised is $\sum_{i \neq i'} \left(s_{ii'} - (\mathbf{z}_i - \bar{\mathbf{z}}), (\mathbf{z}'_i - \bar{\mathbf{z}})^T\right)^2$. The computations that are needed are:

The Multi-Dimensional Scaling (MDS) Algorithm

- Compute the matrix of squared pairwise similarities $\mathbf{D}$, $\mathbf{D}_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|^2$
- Compute $\mathbf{J} = \mathbf{I}_N - 1/N$ (where $\mathbf{I}_N$ is the $N \times N$ identity function and N is the number of datapoints)
- Compute $\mathbf{B} = -\frac{1}{2}\mathbf{J}\mathbf{D}\mathbf{J}^T$
- Find the L largest eigenvalues λ_i of $\mathbf{B}$, together with the corresponding eigenvectors $\mathbf{e}_i$
- Put the eigenvalues into a diagonal matrix $\mathbf{V}$ and set the eigenvectors to be columns of matrix $\mathbf{P}$
- Compute the embedding as $\mathbf{X} = \mathbf{P}\mathbf{V}^{1/2}$

This classical MDS algorithm works fine on flat manifolds (dataspaces). However, we are interested in manifolds that are not flat, and this is where Isomap comes in. The algorithm has to construct the distance matrix for all pairs of datapoints on the manifold, but there is no information about the manifold, and so the distances can't be computed exactly. Isomap approximates them by assuming that the distances between pairs of points that are close together are good, since over a small distance the non-linearity of the manifold won't matter. It builds up the distances between points that are far away by finding paths that run through points that are close together, i.e., that are neighbours, and then uses normal MDS on this distance matrix:

The Isomap Algorithm

- Construct the pairwise distances between all pairs of points
- Identify the neighbours of each point to make a weighted graph G
- Estimate the geodesic distances d_G by finding shortest paths
- Apply classical MDS to the set of d_G

Floyd's and Dijkstra's algorithms are well-known algorithms for finding shortest paths on graphs. They are of $\mathcal{O}(N^3)$ and $\mathcal{O}(N^2)$ time complexity, respectively. Any good algorithms textbook provides the details if you don't know them.

There is one practical aspect of Isomap, which is that getting the number of neighbours right can be important, otherwise the graph splits into separate components (that is, segments of the graph that are not linked to each other), which have infinite distance between them. You then have to be careful to deal only with the largest component, which means that you end up with less data than you started with. Otherwise the implementation is fairly simple.

Figure 6.14 shows the results of applying Isomap to the iris dataset. Here, the default neighbourhood size of 12 produced a largest component that held only one of the three classes, and the other two were deleted. By increasing the neighbourhood size over 50, so that each point had more neighbours than were in its class, the results shown in the figure were produced. On the swissroll dataset shown on the left of Figure 6.13, Isomap produces qualitatively similar results to LLE, as can be seen in Figure 6.15.

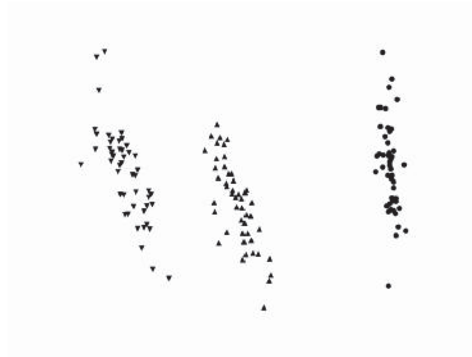


FIGURE 6.14 Isomap transforms the iris data in a similar way to factor analysis, provided that the neighbourhood size is large enough to avoid points becoming disconnected.

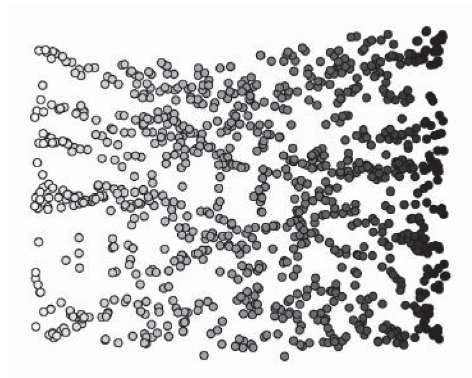


FIGURE 6.15 Isomap also produces a good remapping of the swissroll dataset.

Although the two algorithms produce similar mappings of the swissroll dataset, they are based on different principles. Isomap attempts to find a mapping that preserves the distances between pairs of points within the manifold, no matter how far apart they are, while LLE focuses only on local regions of the manifold. This means that the computational cost of LLE is significantly less, but it can make errors by putting points close together that should be far apart. The choice of which algorithm to use often depends upon the dataset, and trying both of them out for your particular dataset is often a good idea.

FURTHER READING

Surveys of the area of dimensionality reduction include:

- L.J.P. van der Maaten. An introduction to dimensionality reduction using MATLAB. Technical Report MICC 07-07, Maastricht University, Maastricht, the Netherlands, 2007.
- F. Camastra. Data dimensionality estimation methods: a survey. *Pattern Recognition*, 36:2945–2954, 2003.

For more information about many of the methods desribed here, there are books or papers that contain a lot of information. Notable references include:

- (for LDA) Section 4.3 of T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, 2nd edition, Springer, Berlin, Germany, 2008.
- (for PCA) I.T. Jolliffe. *Principal Components Analysis*. Springer, Berlin, Germany, 1986.
- (for kernel PCA) J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 2004.
- (for ICA) J.V. Stone. *Independent Components Analysis: A Tutorial Introduction*. MIT Press, Cambridge, MA, USA, 2004.
- (for ICA) A. Hyvrinen and E. Oja. Independent components analysis: Algorithms and applications. *Neural Networks*, 13(4–5):411–430, 2000.
- (for LLE) S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326, 2000.
- (for MDS) T.F. Cox and M.A.A. Cox. *Multidimensional Scaling*. Chapman & Hall, London, UK, 1994.
- (for Isomap) J.B. Tenenbaum, V. de Silva, and J.C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.
- Chapter 12 of C.M. Bishop. *Pattern Recognition and Machine Learning*. Springer, Berlin, Germany, 2006.

PRACTICE QUESTIONS

Problem 6.1 Use LDA on the iris dataset (which is what Fisher originally tested LDA on).

Problem 6.2 Compare the results with using PCA, which is not supervised and will not therefore be able to find the same space.

Problem 6.3 Compute the eigenvalues and eigenvectors of:

$$\begin{pmatrix} 5 & 7 \\ -2 & -4 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 & 1 \\ 6 & -1 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{6.33}$$

Problem 6.4 Compare the algorithms described in this chapter on a variety of different datasets, including the `yeast` dataset and the `wine` dataset. Input the results of the data reduction method to the MLP and SOM. Are the results better than before this preprocessing?

Problem 6.5 Modify the Isomap code to use Dijkstra's algorithm rather than Floyd's algorithm.

Problem 6.6 Another dataset that the Isomap and LLE algorithms are commonly demonstrated on is the 'S' shape that is available on the website. Download it and test various algorithms, not just Isomap and LLE on it. For Isomap and LLE, try different numbers of neighbours to see the effect that this has.