FIGURE 10.5: Plot of the iris data showing the three classes *left:* before and *right:* after LDA has been applied.

```
    d = squeeze(data[indices,:])
    classcov = cov(transpose(d))
    Sw += float(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 = argsort(evals)
indices = indices[::-1]
evecs = evecs[:,indices]
evals = evals[indices]
w = evecs[:,:redDim]
newData = transpose(dot(data,w))
```

As an example of using the algorithm, Figure 10.5 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 triangles) 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).

10.2 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

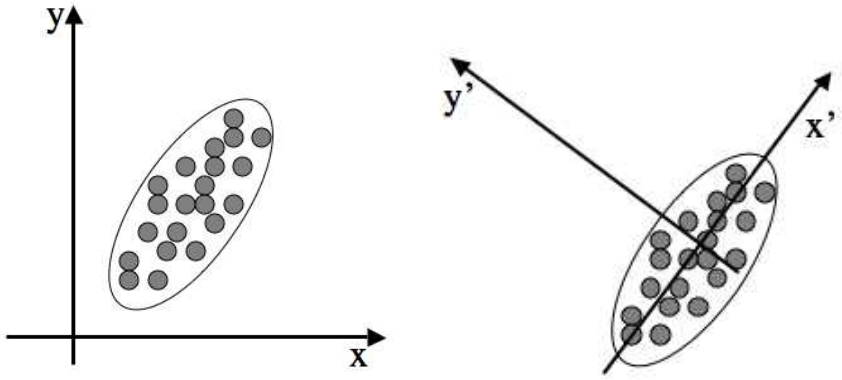

FIGURE 10.6: Two different sets of coordinate axes. The second consists of a rotation and translation of the first and was found using Principal Components Analysis.

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 10.6, 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 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 it 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.,

$$\text{cov}(\mathbf{Y}) = \text{cov}(\mathbf{P}^T\mathbf{X}) = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \lambda_N \end{pmatrix}. \tag{10.7}$$

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

$$\begin{aligned} \text{cov}(\mathbf{Y}) &= E[\mathbf{Y}\mathbf{Y}^T] && (10.8)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{P}^T\mathbf{X})^T] && (10.9)\\ &= E[(\mathbf{P}^T\mathbf{X})(\mathbf{X}^T\mathbf{P})] && (10.10)\\ &= \mathbf{P}^T E(\mathbf{X}\mathbf{X}^T)\mathbf{P} && (10.11)\\ &= \mathbf{P}^T \text{cov}(\mathbf{X})\mathbf{P}. && (10.12) \end{aligned}$$

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:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = \mathbf{P}\mathbf{P}^T\text{cov}(\mathbf{X})\mathbf{P} = \text{cov}(\mathbf{X})\mathbf{P}, \tag{10.13}$$

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:

$$\mathbf{P}\text{cov}(\mathbf{Y}) = [\lambda_1\mathbf{p}_1, \lambda_2\mathbf{p}_2, \dots, \lambda_N\mathbf{p}_N], \tag{10.14}$$

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:

$$\boldsymbol{\lambda}\mathbf{P} = \mathbf{Z}\mathbf{P}. \tag{10.15}$$

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. Since $\boldsymbol{\lambda}$ is only a column vector, all it does is rescale the components of $\mathbf{P}$; 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 unit length and are orthogonal to each other. This tells us that the eigenvectors define a space. If we make a matrix $\mathbf{E}$ that contains the 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 $\mathbf{E}^T$. 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 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

---

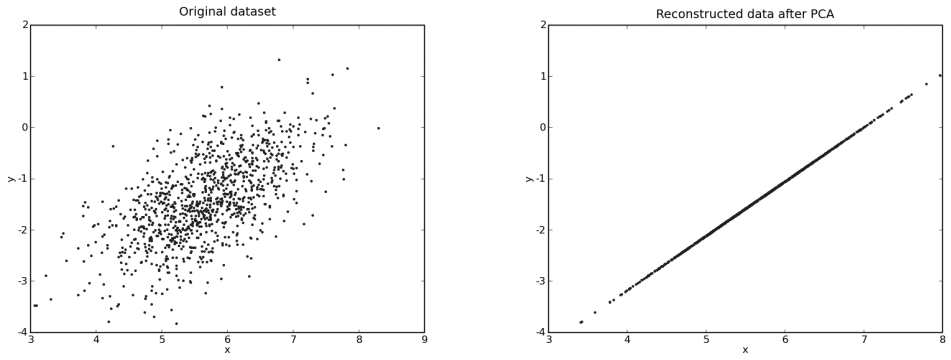

FIGURE 10.7: Computing the principal components of the 2D dataset on the left and using only the first one to reconstruct it produces the line of data shown on the right, which is along the principal axis of the ellipse that the data was sampled from.

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

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

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

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

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

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

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

    # Produce the new data matrix
```

FIGURE 10.8: Plot of the first two principal components of the iris data, showing that the three classes are clearly distinguishable.

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

Two different examples of using PCA are shown in Figures 10.7 and 10.8. 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 10.8 shows the first two dimensions of the iris data, and shows that the three classes are clearly distinguishable after PCA has been applied.

10.2.1 Relation with the Multi-Layer Perceptron

We've seen (in Section 9.3.2) 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 3.4.5. 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 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