NUI Chapter 8. Face Recognition

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/jg/??; only important fragments are described here.]

In the last chapter I developed software that could detect and track a face as it moved in front of a webcam. The next step, and the topic of this chapter, is to attach a name to the face, to recognize it using a technique called *eigenfaces*.

The outcome is the GUI application shown in Figure 1.

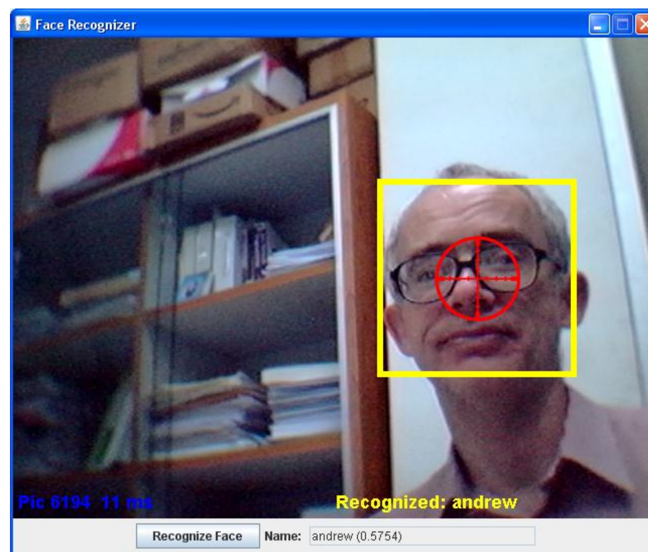


Figure 1. A GUI Face Recognizer Application.

When the user presses the "Recognize Face" button (at the bottom of the window in Figure 1), the currently selected face is compared against a set of images that the recognizer code has been trained against. The name associated with closest matching training image is reported in a textfield (and in yellow text in the image pane), along with a distance measure represented the 'closeness' of the match.

The recognition process relies on a preceding training phase involving multiple facial images, with associated names. Typical training images are shown in Figure 2.

 © Andrew Davison 2013

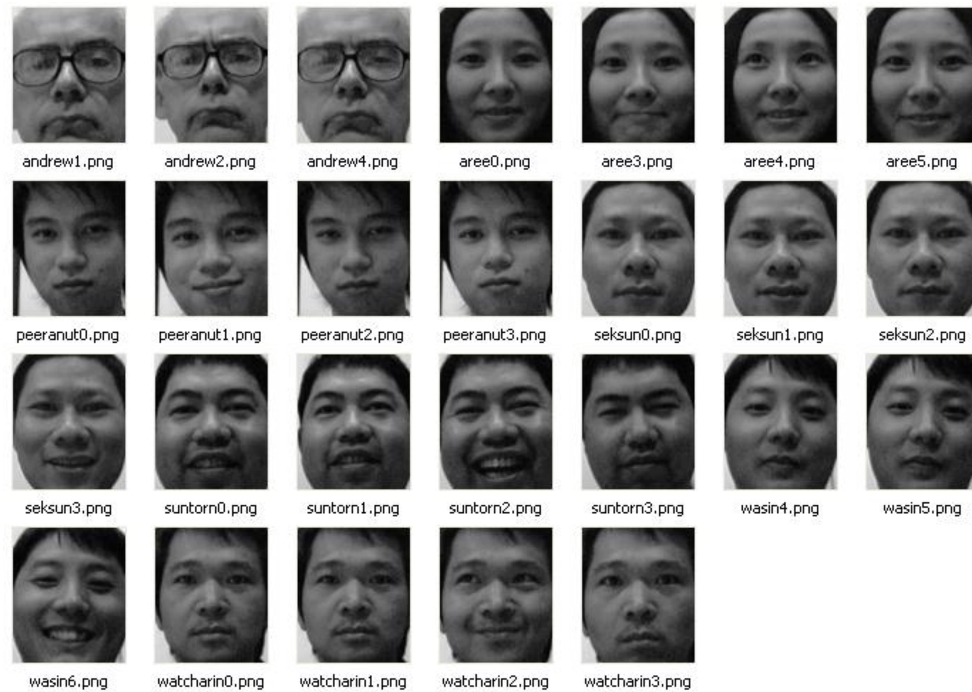


Figure 2. Training Images.

It's important that the training images all be cropped and orientated in a similar way, so that the variations between the images are caused by facial differences rather than differences in the background or facial position. There should be uniformity in the images' size, resolution, and brightness. It's useful to include several pictures of the same face showing different expressions, such as smiling and frowning. The name of each person is encoded in each image's filename. For instance, I'm represented by three image files, called "andrew1.png", "andrew2.png", and "andrew4.png".

The training process creates *eigenfaces* whichare composites of the training images which highlight elements that distinguish between faces. Typical eigenfaces generated by the training session are shown in Figure 3.

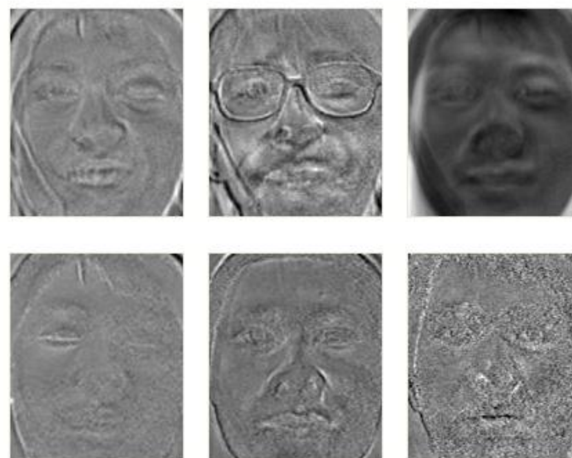


Figure 3. Some Eigenfaces.

Due to their strange appearance, eigenfaces are sometimes called *ghost faces*.

 © Andrew Davison 2013

Each training image can be represented by a weighted sequence of eigenfaces, as depicted in Figure 4.

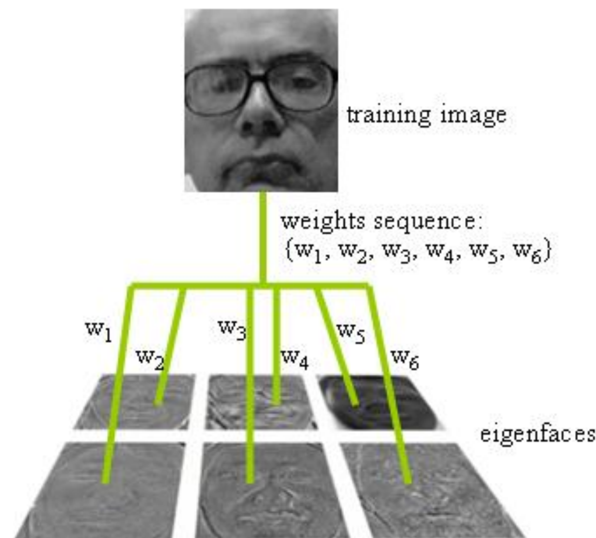


Figure 4. A Training Image as a Weights Sequence of Eigenfaces.

The idea is that a training image can be decomposed into the weighted sum of multiple eigenfaces, and all the weights stored as a sequence.

Not all eigenfaces are equally important – some will contain more important facial elements for distinguishing between images. This means that it is not usually necessary to use all the generated eigenfaces to recognize a face. This allows an image to be represented by a smaller weights sequence (e.g. using only three weights instead of six in Figure 4). The downside is that less weights means that less important facial elements will not be considered during the recognition process.

Another way of understanding the relationship between eigenfaces and images is that each image is positioned in a multi-dimensional *eigenspace*, whose axes are the eigenfaces (see Figure 5)

 © Andrew Davison 2013

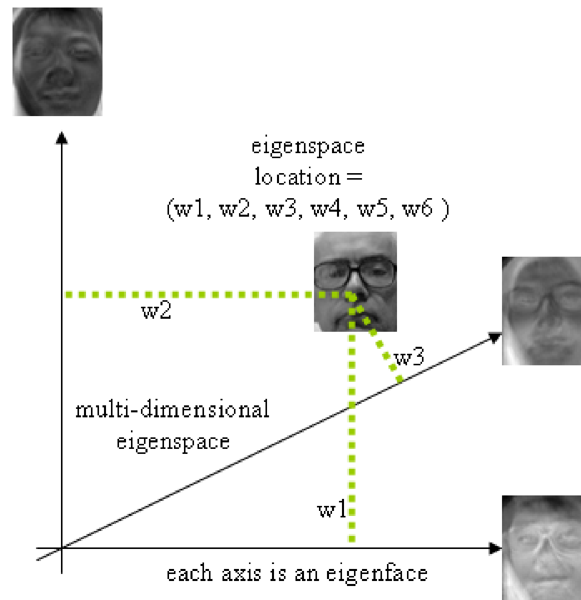


Figure 5. An Image in Eigenspace.

The weights can now be viewed as the image's coordinates in the eigenspace. Due to my poor drawing skill, I've only shown three axes in Figure 5, but if there are six weights, then there should be six orthogonal axes (one for each eigenface).

After the training phase comes face recognition. A picture of a new face is decomposed into eigenfaces, with a weight assigned to each one denoting its importance (in the same way as in Figure 4). The resulting weights sequence is compared with each of the weights sequences for the training images, and the name associated with the 'closest' matching training image is used to identify the new face.

Alternatively, we can explain the recognition stage in terms of eigenspaces: the new image is positioned in the eigenspace, with its own coordinates (weights). Then a distance measure (often Euclidean distance) is used to find the closest training image (see Figure 6).

 © Andrew Davison 2013

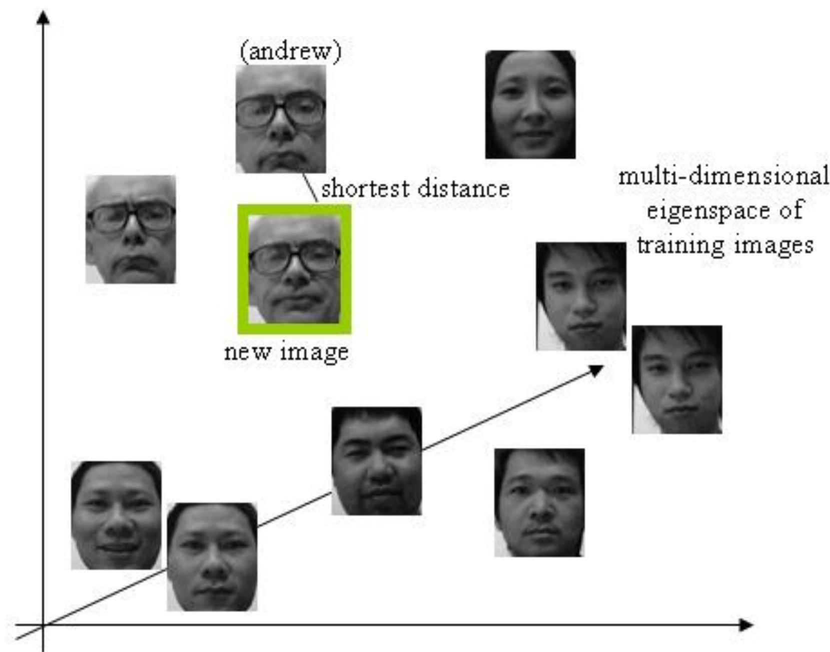


Figure 6. Recognizing an Image in Eigenspace.

Although I've been using faces in my explanation, there's nothing preventing this technique being applied to other kinds of pictures, in which case it's called *eigenimage* recognition. Eigenimaging is best applied to objects that have a regular structure made up of varying subcomponents. Faces are a good choice because they all have a very similar composition (two eyes, a nose and mouth), but with variations between the components (e.g. in size and shape).

1. How Eigenfaces are Generated

Eigenfaces (and eigenimages) are created using a mathematical technique called Principal Components Analysis (PCA) which highlights the similarities and differences in data. I won't be explaining much of the maths behind PCA, relying instead on visual intuitions about how it works. I've used examples taken from "A Tutorial on Principle Components Analysis" by Lindsay Smith, available online at http://www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf

If you want a more rigorous treatment of PCA, then browse through a textbook on linear algebra, such as *Elementary Linear Algebra* by Howard Anton, from publisher John Wiley.

Rather than start with images, I'll explain PCA using two sets of numerical data – a set of x values and set of y values presented in Table 1.

x	y
2.5	2.4
0.5	0.7

 © Andrew Davison 2013

2.2	2.9
1.9	2.2
3.1	3.0
2.3	2.7
2	1.6
1	1.1
1.5	1.6
1.1	0.9

Table 1. x and y data.

The x and y data could be collection of student heights and weights, or the prices for pork bellies and leeks.

My objective is to highlight the similarities and differences between the two data sets in a numerical manner. I'll assume you're familiar with statistical measures like the mean and standard deviation (a measure of the spread of data around the mean). Variance is another measure of spread, which is just the standard deviation squared:

$$\mathrm{var}(x) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(x_i - \bar{x})}{(n-1)}$$

n is the number of data items (10 in Table 1), $\bar{x}$ is the mean of the data set.

Unfortunately mean and variance only give us information about the shape of the data in a single set (i.e. the mean and variance of the x data set); I want to quantify the differences *between* data sets (i.e. between the x and y sets). Covariance is such a measure – it's a generalization of the variance equation which compares two sets of data:

$$\mathrm{cov}(x,y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{(n-1)}$$

If I fire up my calculator, and apply the cov() function to the x and y data given in Table 1, the result is 0.6154. The important part is the sign: a negative value means that as one data set increases, the other decreases. If the value is positive (as here), then both data sets increase together. A simple visual confirmation of this is to plot corresponding x and y data as points on a graph, as in Figure 7.

 © Andrew Davison 2013

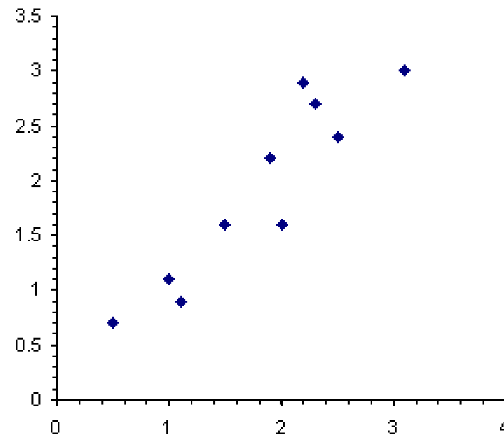


Figure 7. A Plot of x and y Data.

If the data has more than two data sets (e.g. x, y, and z data sets) then the covariance is calculated between all data set pairs, resulting in cov(x, y), cov(x, z), and cov(y, z) values. There's no need to calculate cov(y, x), cov(z, x), and cov(z, y) since the equation's definition means that cov(A, B) is equal to cov(B, A).

The standard way of storing covariance values between multiple data sets is in a matrix, where the data sets become the column and row indices. For example, the covariances for x, y, and z data would become a 3x3 matrix:

$$\begin{pmatrix} \operatorname{cov}(x,x) & \operatorname{cov}(x,y) & \operatorname{cov}(x,z) \\ \operatorname{cov}(y,x) & \operatorname{cov}(y,y) & \operatorname{cov}(y,z) \\ \operatorname{cov}(z,x) & \operatorname{cov}(z,y) & \operatorname{cov}(z,z) \end{pmatrix}$$

Down the main diagonal, the covariance is between a data set and itself, which is equivalent to the variance in that data (you can see that by comparing the var(x) and cov(x, x) equations). The matrix is also symmetrical around the main diagonal.

My x and y data would be represented by a 2x2 covariance matrix:

$$\begin{pmatrix} 0.6166 & 0.6154 \\ 0.6154 & 0.7166 \end{pmatrix}$$

1.1. From Covariance to Eigenvectors

A covariance matrix is a useful way of showing the relationships between data sets, but I can obtain more information by using the matrix to calculate *eigenvectors* and *eigenvalues*.

An eigenvector is an ordinary vector that when multiplied by a given matrix only changes its magnitude; eigenvalue is the fancy name for that magnitude. As an example, I'll use the matrix:

$$\begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix}$$

An eigenvector for this matrix is $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$ because when multiplied by that matrix, the same vector is returned multiplied by 4:

 © Andrew Davison 2013

$$\begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 12 \\ 8 \end{pmatrix} = 4 \times \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

4 is the eigenvalue for the $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$ eigenvector.

Eigenvectors can only be found for square matrices, and not even for every one of those. For a given n x n matrix that does have eigenvectors, there will be n of them. For example, my 2x2 example above has two eigenvectors (and corresponding eigenvalues).

The eigenvector relationship can be written mathematically as:

$$Av = \lambda v$$

A is the square matrix, *v* an eigenvector for A and the scalar λ is its eigenvalue.

For their use in PCA, eigenvectors should be normalized so they are all have the same unit length. For instance, the eigenvector from above, $\begin{pmatrix} 3 \\ 2 \end{pmatrix}$, has the non-unit length $\sqrt{(3^2 + 2^2)} = \sqrt{13}$. I divide the vector by this value to make it unit length:

$$\begin{pmatrix} 3 \\ 2 \end{pmatrix} \div \sqrt{13} = \begin{pmatrix} 3/\sqrt{13} \\ 2/\sqrt{13} \end{pmatrix}$$

You may be wondering how eigenvectors (and their eigenvalues) are calculated for a given matrix? It's fairly easy for 2x2 or 3x3 matrices, and the details are explained in linear algebra books (such as *Elementary Linear Algebra* by Howard Anton). The calculation is more time-consuming for larger matrices, but thankfully I'll be using the Colt math library (http://acs.lbl.gov/software/colt/) to do the work in the next section.

If you recall, I generated a covariance matrix at the end of the last section:

$$\begin{pmatrix} 0.6166 & 0.6154 \\ 0.6154 & 0.7166 \end{pmatrix}$$

Since it's a 2x2 matrix, it has two eigenvectors and eigenvalues:

$$\begin{pmatrix} -0.7352 \\ 0.6779 \end{pmatrix} \text{ and } 0.049$$

$$\begin{pmatrix} 0.6779 \\ 0.7352 \end{pmatrix} \text{ and } 1.284$$

Both eigenvectors have unit length.

I started this section by saying that eigenvectors and eigenvariables reveal more information about the relationship between data sets; in my case the relationship between the x and y data from Table 1. So what exactly is revealed?

1.2. Using Eigenvectors for PCA

Let's return to the plot of the x and y data shown in Figure 7. In Figure 8, their means ($\bar{x}$ and $\bar{y}$) have been subtracted from the data sets, so they are shifted to be centered around the origin (the data has been *normalized*).

 © Andrew Davison 2013

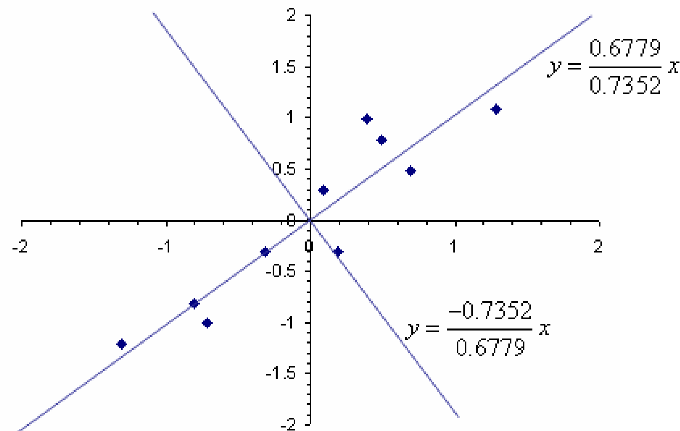


Figure 8. Normalized Plot of x and y Data with Eigenvectors.

The two eigenvectors can be added to the graph as lines by converted the vectors into equations. $\begin{pmatrix}-0.7352\\0.6779\end{pmatrix}$ becomes $y = \frac{-0.7352}{0.6779} x$ and $\begin{pmatrix}0.6779\\0.7352\end{pmatrix}$ becomes $y = \frac{0.6779}{0.7352} x$. The normalized data and two equations are shown in Figure 8 as blue lines.

Figure 8 shows the way that eigenvectors highlight the relationships between data sets – the vectors indicate how the data sets are spread out within the coordinate space. This 'spread' information allows us to more easily distinguish between data points.

Of the two eigenvectors (marked in blue), the $y = \frac{0.6779}{0.7352} x$ line is most useful since the data is most spread out along that line. This can be confirmed numerically by looking at the eigenvalues associated with the eigenvectors. The $y = \frac{0.6779}{0.7352} x$ eigenvector is a better spread indicator since its corresponding eigenvalue (1.284) is the larger of the two eigenvalues.

The other line, $y = \frac{-0.7352}{0.6779} x$, contributes useful 'spread' information by showing how the data is spaced out to the left or right of the main eigenvector (the $y = \frac{0.6779}{0.7352} x$ line). However, it's a less important component of the data spread, as shown by its smaller eigenvalue (0.049).

The eigenvector with the highest eigenvalue (i.e. the $y = \frac{0.6779}{0.7352} x$ line, or the $\begin{pmatrix}0.6779\\0.7352\end{pmatrix}$ vector) is called the *principle component* of the data set. I've just used the eigenvector and eigenvariable information to carry out PCA on the data sets.

All the eigenvectors extracted from a matrix are perpendicular. This means that its possible to rotate (and perhaps reflect) the data so that the eigenvectors become aligned with the axes. If I choose to rotate and reflect the principle component $\begin{pmatrix}0.6779\\0.7352\end{pmatrix}$ to line up with the y-axis, the result is shown in Figure 9.

 © Andrew Davison 2013

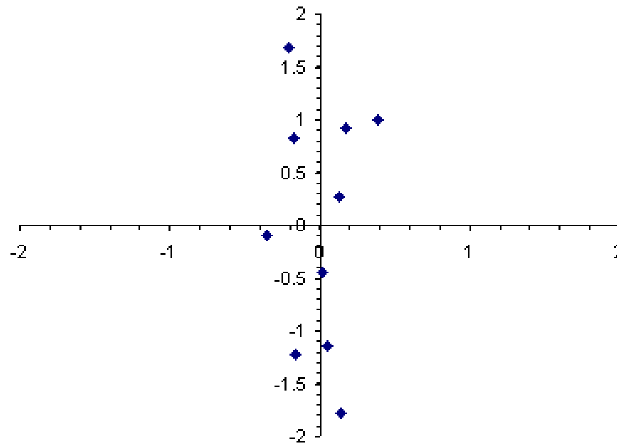


Figure 9. Rotated and Reflected Version of Figure 8.

Strictly speaking, there's no need to reflect the data as we move from Figure 8 to Figure 9. However, these data points are the output of the Colt eigenvector function (employed later), so I've decided to use them unedited.

In Figure 9, a data coordinate consists of the point's distances from the eigenvector axes. For example the point nearest the origin is (0.13, 0.27). Alternatively, I can talk about each point having a weight relative to each eigenvector, which forms its weight sequence. The point nearest the origin has the weights sequence {0.13, 0.27}.

Either of the two notations (coordinates or weights) can be used with eigenfaces – in Figure 4 the image is a weighted sequence of six eigenfaces. Alternatively, it's a data point in six-dimensional eigenspace in Figure 5, with each axis defined by an eigenface (or eigenvector).

It helps to remember how this data will be used later. I'll obtain new data (a new face image), and want to decide which of the existing data (training images) is most similar to it. There are a few possible distance measures I could use, but a simple one is the Euclidean distance, which is the straight line distance between two points in space.

At this recognition stage, I could compare the new data with the data shown in Figure 9. However, in real-world examples, we usually want to reduce the data size first. The aim is to speed up the recognition task, while retaining sufficient data to distinguish between the points when we compare them with the new data.

The trick is to retain all the data points, but reduce the dimensionality of the eigenspace. This reduces the data size by removing some of the coordinates axes, which is equivalent to removing some of the eigenvectors. But which eigenvectors should I remove? I should delete the ones that have least influence on the spread of the data, which I know by looking at the eigenvalues associated with the eigenvectors.

The data shown in Figure 9 have the following eigenvectors and eigenvalues:

$$\begin{pmatrix} -0.7352 \\ 0.6779 \end{pmatrix} \text{ and } 0.049$$

$$\begin{pmatrix} 0.6779 \\ 0.7352 \end{pmatrix} \text{ and } 1.284$$

 © Andrew Davison 2013

The first eigenvector has been rotated onto the x-axis, and the second eigenvector onto the y-axis.

The first eigenvector contributes least to the spread information, due to its small eigenvalue (and also by looking at Figure 9). If I discarded it, the data points in Figure 9 are projected onto the y-axis, as the x-axis disappears; the result is shown in Figure 10.

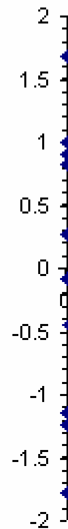


Figure 10. Data from Figure 9 Projected onto the y-axis.

This approach means that I've halved the amount of data stored for each data point. However, the data is still spread out enough so that a new data point can be compared with it using a Euclidian distance measure. I'll go through an example of this in the next section.

Unfortunately, the outcome of eigenvector removal may not always be so positive. The removal means that spread information is lost, and this may be critical if the original points were close together. For example, Figure 9 contains three points near y == 1 which are spread out along the x-axis. After that axis' removal, the points are much closer together in Figure 10, and so harder to differentiate.

2. Programming PCA

This is a book about programming, and so it's time to implement a siimplified version of the PCA algorithm. Java doesn't directly support the necessary statistical and linear algebra operations, but there are many third party libraries that do. I'll use the Colt package, available from http://acs.lbl.gov/software/colt/.

Colt supports multi-dimensional arrays, linear algebra operations, including eigenvectors, and a wide range of maths and statistics (including covariance). On the downside, the package is quite old (dating from 2004), and comes with hardly any information aside from its API documentation. In truth, I chose it because the Javafaces package (http://code.google.com/p/javafaces/) I'll be using later is built on top of Colt.

 © Andrew Davison 2013

A good starting point for finding alternatives to Colt is the JavaNumerics website (http://math.nist.gov/javanumerics/), which has a lengthy libraries section containing links to over 40 tools. There's a recent benchmark for Java matrix libraries at http://code.google.com/p/java-matrix-benchmark/wiki/RuntimeQ9400_2010_02, with EJML (http://code.google.com/p/efficient-java-matrix-library/) appearing to be the fastest for Eigenvector calculation for smaller matrices, and JBlas (http://jblas.org/) better for larger ones. There's an interesting blog entry discussing these benchmarks at http://measuringmeasures.com/blog/2010/3/28/matrix-benchmarks-fast-linear-algebra-on-the-jvm.html

The main() function of my Pca.java code employs the data that I introduced back in Table 1 to build a covariance matrix, which is used to calculate eigenvectors and eigenvalues. A new piece of data is then compared to the transformed training data using a Euclidian distance measure.

```
public static void main(String args[])
{
  double[][] trainingData = new double[][]{
   {2.5, 0.5, 2.2, 1.9, 3.1, 2.3, 2.0, 1.0, 1.5, 1.1},    // x data
   {2.4, 0.7, 2.9, 2.2, 3.0, 2.7, 1.6, 1.1, 1.6, 0.9},    // y
        };    // same as in Table 1

  // compute covariance matrix
  DoubleMatrix2D matCov = calcCovarMat(trainingData);

  // get eigenvectors and eigenvalues
  EigenvalueDecomposition matEig =
                  new EigenvalueDecomposition(matCov);

  DoubleMatrix2D eigenVecs = matEig.getV();
  System.out.println("\nEigenvectors: \n" + eigenVecs);

  DoubleMatrix1D realEigVals = matEig.getRealEigenvalues();
  System.out.println("\nEigenvalues: \n" + realEigVals);

  reportBiggestEigen(realEigVals);
                  // report column pos of largest eigenvalue

  // calculate array of means for each row of training data
  double[] means = calcMeans(trainingData);

  // recognition task for new data  (x, y)
  double[] newData = new double[]{ 2.511, 2.411 };

  // transform all data
  DoubleMatrix2D matTransAllData =
        transformAllData(newData, trainingData, means, eigenVecs);

  /* report minimal euclidean distances between new data
     and training data */
  minEuclid(matTransAllData);
}  // end of main()
```

The complexities of eigenvector and eigenvalue calculation are hidden inside the Colt EigenvalueDecomposition class, which is initialized with a covariance matrix, and then has get() methods for accessing the eigenvectors and scalars (getV() and getRealEigenvalues()).

 © Andrew Davison 2013

The training data and new data are transformed inside transformAllData(), which mean-normalizes the data and rotates it so the principal component (most significant eigenvector) coincides with the y-axis. In other words, the data is changed to look like that shown in Figure 9 (although no graph is displayed by the program).

The smallest Euclidian distance between the new data and the training data is calculated in minEuclid(), which reports the index position of the nearest data in the training set.

The new data is hardwired to be the point (2.511, 2.411), which is closest to the training data point (2.5, 2.4), which is at index position 0 in trainingData[][].

transformAllData() uses all the eigenvectors, but it would be quite easy to use only the principal vector, $\begin{pmatrix} 0.6779 \\ 0.7352 \end{pmatrix}$, so the data would be projected onto the y-axis as in Figure 10.

2.1. The Covariance Matrix

A covariance matrix for x, y, and z data would be the 3x3 matrix:

$$\begin{pmatrix} \text{cov}(x,x) & \text{cov}(x,y) & \text{cov}(x,z) \\ \text{cov}(y,x) & \text{cov}(y,y) & \text{cov}(y,z) \\ \text{cov}(z,x) & \text{cov}(z,y) & \text{cov}(z,z) \end{pmatrix}$$

cov() is a built-in function of the Colt library, in Descriptive.covariance(), and so my calcCovarMat() spends most of its time building the matrix:

```
private static DoubleMatrix2D calcCovarMat(double[][] trainingData)
{
  int numRows = trainingData.length;

  DoubleMatrix2D matCov = new DenseDoubleMatrix2D(numRows, numRows);
  for (int i = 0; i < numRows; i++) {
    DoubleArrayList iRow = new DoubleArrayList(trainingData[i]);
    double variance = Descriptive.covariance(iRow, iRow);
    matCov.setQuick(i, i, variance);   // main diagonal value

    // fill values symmetrically around main diagonal
    for (int j = i+1; j < numRows; j++) {
      double cov = Descriptive.covariance(iRow,
            new DoubleArrayList(trainingData[j]));
      matCov.setQuick(i, j, cov);    // fill to the right
      matCov.setQuick(j, i, cov);    // fill below
    }
  }
  return matCov;
}  // end of calcCovarMat()
```

The covariance matrix for my training data is: $\begin{pmatrix} 0.6166 & 0.6154 \\ 0.6154 & 0.7166 \end{pmatrix}$

 © Andrew Davison 2013

2.2. Using the Eigenvectors and Eigenvariables

The eigenvectors and eigenvariables obtained from the covariance matrix are returned in Colt DoubleMatrix2D and DoubleMatrix1D data structures, which are easy to print and manipulate. Their values are printed as:

```
Eigenvectors:
2 x 2 matrix
-0.735179 0.677873
 0.677873 0.735179

Eigenvalues:
1 x 2 matrix
0.049083 1.284028
```

The vectors and scalars are listed in column order, so the principal component is the second column $\begin{pmatrix}0.677873\\0.735179\end{pmatrix}$ because its eigenvalue is 1.284028.

My reportBiggestEigen() method iterates through the eigenvalues matrix looking for the biggest value, and then reports its column position (i.e. column 1). This could be used to select the principal component from the eigenvectors.

2.3. Recognizing New Data

The new data is the coordinate (2.511, 2.411), and "recognition" means finding the training data point that is closest to that coordinate.

The new coordinate is added to the existing training data inside transformAllData(), which transforms all the data in the ways shown in Figures 8 and 9 – first the data is mean normalized, then rotated and reflected so the principle component is aligned with the y-axis. This alignment is achieved by using the normalized eigenvectors to transform the data points via matrix multiplication:

transformed data = transposed_eigenvectors × data_matrix

The eigenvectors are stored in column order, which must be changed to row order by transposition before the matrix multiplication can be carried out.

The code:

```
private static DoubleMatrix2D transformAllData(double[] newData,
                        double[][] trainingData, double[] means,
                        DoubleMatrix2D eigenVecs)
{
  int numRows = trainingData.length;
  int numCols = trainingData[0].length;

  // create new matrix with new data in first column,
  // training data in the rest
  DoubleMatrix2D matAllData =
           new DenseDoubleMatrix2D(numRows, numCols+1);

  DoubleMatrix1D matNewData = DoubleFactory1D.dense.make(newData);
  matAllData.viewColumn(0).assign(matNewData);  //new data in 1st col

  DoubleMatrix2D matData = DoubleFactory2D.dense.make(trainingData);
```

```
  matAllData.viewPart(0, 1, numRows, matData.columns()).
                                          assign(matData);
     // training data in the other columns

  // subtract mean from all data matrix
  for (int i=0; i < numRows; i++)
    matAllData.viewRow(i).assign(Functions.minus(means[i]));

  // transpose the eigenvectors
  DoubleMatrix2D eigenVecsTr = Algebra.DEFAULT.transpose(eigenVecs);

  // return the transformed data
  return Algebra.DEFAULT.mult(eigenVecsTr, matAllData);
}  // end of transformAllData()
```

The new data is added to the first column of a new matrix called matNewData, and the training data to the other columns. The matrix is mean-normalized, and the eigenvectors are transposed. The transformed data is created by multiplying the transposed eigenvectors to the data.

The resulting matrix is:

$$\begin{pmatrix} -0.1758 & -0.1751 & 0.1429 & 0.3844 & 0.1304 & -0.2095 & 0.1753 & -0.3498 & 0.0464 & 0.0178 & -0.1627 \\ 0.8435 & 0.8280 & -1.7776 & 0.9922 & 0.2742 & 1.6758 & 0.9130 & -0.0991 & -1.1446 & -0.4380 & -1.2238 \end{pmatrix}$$

The first column is the transformed new data coordinate, while the other columns are the training data. A plot of these points, with the new data plotted as a red slightly larger dot, is shown in Figure 11.

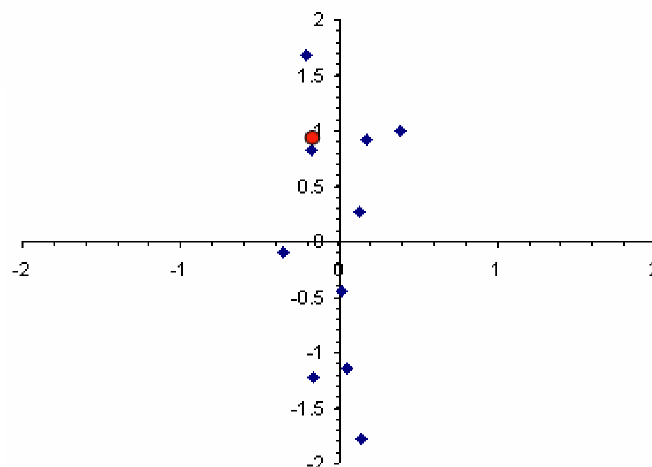


Figure 11. Plot of Transformed New Data and Training Data.

Figure 11 is the same graph as Figure 9, with the addition of the new data point, which is located just above the point (-0.1751, 0.8280), which began as the training point (2.5, 2.4). In other words, the new data point (2.511, 2.411) is closest to the training data point (2.5, 2.4).

This may seem like a lot of work just to compare a few points, but the real advantage of eigenvectors is when the dimensionality of the points becomes must larger (here the dimensionality is two, but when we look at images it will increase to 40,000).

 © Andrew Davison 2013

With larger dimensionality comes the need to speed up the comparison process. The dimensionality of the transformed data can be easily reduced by using less eigenvectors. For example, the smaller eigenvector could be removed from the eigenVecs DoubleMatrix2D before it is passed into transformAllData(). The method would then return a single row of data:

$$(0.8435 \quad 0.8280 \quad -1.7776 \quad 0.9922 \quad 0.2742 \quad 1.6758 \quad 0.9130 \quad -0.0991 \quad -1.1446 \quad -0.4380 \quad -1.2238)$$

This corresponds to a mapping of the data onto the principal component axis, which is represented by the graph in Figure 12.

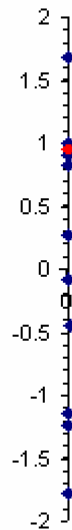


Figure 12. Plot of Transformed New Data and Training Data
Using only the Principal Component.

Figure 12 is similar to Figure 10, except that the new data point is included as a red dot. The figure shows the potential problem with reducing data dimensionality: it may make it harder to decide which training point is closest to the new data.

Another factor is the Euclidean distance equation. It utilizes a threshold value to define when two points are close together. If the value is set too large, then several points may be within an acceptable distance of the new data point.

2.4. Calculating the Distance Measure

Colt contains a number of distance measuring functions. My minEuclid() method calls the Colt Statistic.distance() function on the 2x11 matrix plotted in Figure 11. It calculates the distance between all the points in the matrix, returning a 11x11 symmetric square matrix. I only need the first row (or column), which are the distance measures between the first point (i.e. the new data) and the other points in the matrix (i.e. the training data). The relevant data row is:

$$(0 \quad 0.0156 \quad 2.6404 \quad 0.5795 \quad 0.6464 \quad 0.8330 \quad 0.3578 \quad 0.9586 \quad 2.005 \quad 1.2961 \quad 2.0674)$$

The first element in the row can be disregarded since it's the distance between the new data and itself. The rest of the row must be scanned to find the smallest value, which just happens to be the second value in the row. This is the distance to the first value in the training data, (2.5, 2.4).

 © Andrew Davison 2013

The minEuclid() code:

```
private static void minEuclid(DoubleMatrix2D matTransAllData)
{
  // calculate euclidean distances between all points
  DoubleMatrix2D matDist =
         Statistic.distance(matTransAllData, Statistic.EUCLID);

  // get first row of dist measures, which is for the new data
  DoubleMatrix1D matNewDist = matDist.viewRow(0);
  System.out.println("\nEuclid dists for new data: \n" + matNewDist);

  // get first row of dist measures again, but sorted
  DoubleMatrix1D matSortNewDist = matDist.viewRow(0).viewSorted();

  // retrieve second value in sorted dist data (first will be 0)
  double smallestDist = matSortNewDist.get(1);
  System.out.printf("\nSmallest distance to new data: %.4f\n",
                                                    smallestDist);

  // find its index pos in unsorted distance data
  int pos = -1;
  for (int i=1; i < matNewDist.size(); i++)    // start in 2nd column
    if (smallestDist == matNewDist.get(i)) {
      pos = i;
      break;
    }
  if (pos != -1)
    System.out.println("Closest pt index in training data:"+(pos-1));
  else
    System.out.println("Closest point not found");
}  // end of minEuclid()
```

minEuclid() finds the *second* smallest distance in the first row of the distance matrix, matDist, by sorting it into ascending order. It uses that distance value to search through the unsorted distance matrix and print out the value's index position.

3. From Eigenvectors to Eigenfaces

I've been explaining PCA using a small collection of x and y data up to now. The good news is that it’s a surprisingly simple step from this kind of information to data consisting of face images.

Essentially all that's required is to convert each image into a data point, and then the same PCA algorithm can be applied.

The translation of an image to a data point is a matter of treating an image as a 2D array, and then mapping it into a 1D vector, as Figure 13 suggests.

 © Andrew Davison 2013

$$\begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1N} \\ p_{21} & p_{22} & & p_{2N} \\ \vdots & & \ddots & \vdots \\ p_{N1} & p_{N2} & \cdots & p_{NN} \end{pmatrix} \rightarrow \begin{pmatrix} p_{11} \\ \vdots \\ p_{1N} \\ p_{21} \\ \vdots \\ p_{2N} \\ \vdots \\ p_{NN} \end{pmatrix}$$

Figure 13. From Image to Data Point (p means pixel).

If each image is a 200 x 200 grayscale, then the resulting data point will reside in 40,000-dimensional space. This isn't any problem for the theory, but will lead to very large matrices in the library method calls. For instance, the covariance matrix must compare every dimension, and so will be 40,000 x 40,000 (1.6×10^9) large. This will lead to the generation of 40,000 eigenvectors (and their eigenvalues). The great majority of these eigenvectors will have very small eigenvalues, which suggests that they're not needed in order to differentiate between the image data points.

In general, if a grayscale image is N pixels square then N^2 eigenvectors will be generated. In the following explanation, I'll also assume there are M images.

A clever way of reducing the number of eigenvectors and the size of the matrices being manipulated was introduced by Turk and Pentland in their paper "Face Recognition using Eigenfaces". An expanded journal version, called "Eigenfaces for Recognition", can be found online at http://www.face-rec.org/algorithms/PCA/jcn.pdf

Their idea is to use matrix manipulation to drastically reduce the number of eigenvector calculations. The trick is to 'pull apart' the covariance matrix (which is (N^2 x N^2)) into two matrices of (N^2 x M) and (M x N^2), then switch and multiply them to create a greatly reduced size (M x M) matrix. Applying eigenvector decomposition to this new matrix will be considerably less work since only M eigenvectors will be generated.

Each of these M eigenvectors will have dimensionality M, so will need to be converted back into M eigenvectors with the original data's dimensionality N^2. This is done using fast matrix multiplication.

The end result are M eigenvectors compared to a much large N^2 number obtained with the original approach. In practice, even M eigenvectors aren't usually necessary for accurately recognizing a new image; often $3 \times M/4$ or $M/2$ is sufficient.

These M eigenvectors can be viewed as axes in eigenspace. It's also possible to convert each one from a (N^2 x 1) vector into a (N x N) image, which is why they are also called eigenfaces (or ghost faces) as shown in Figure 3.

4. Implementing Eigenface Recognition

Eigenface recognition is so useful that there already exists an excellent Java application for carrying it out – Javafaces, developed by Sajan Joseph, and available at http://code.google.com/p/javafaces/.

 © Andrew Davison 2013

The current version comes with a nice GUI interface, but I utilized an earlier command-line version. I also refactored the code quite considerably, splitting it into two major classes: BuildEigenFaces and FaceRecognition. As the names suggest, the BuildEigenFaces class generates eigenvectors and eigenvalues from training data, while FaceRecognition accepts new data, and finds the closest matching training image.

In what follows, I'll give an overview of the modified Javafaces code, but I won't explain every method. I've commented the code, which can be downloaded from http://fivedots.coe.psu.ac.th/~ad/jg/?? with the other examples in this chapter. The explanation that follows concentrates on the PCA-related code, not on more standard stuff such as reading and writing to files.

5. The Build Phase

Each training image is stored in a file in a subdirectory called trainingImages/ (examples are shown in Figure 2). There are several outcomes from running BuildEigenFaces, the most crucial being a binary file called eigen.cache. The cache is a serialized FaceBundle object which stores the calculated eigenvectors and eigenvalues, along with various information about the training images.

Two other outputs are two new directories: eigenfaces/ and reconstructed/, neither of which are needed for the subsequent recognition phase because everything relevant is stored inside eigen.cache. However, these directories are useful for checking the build process. eigenfaces/ contains all the eigenfaces stored as images (see Figure 3 for examples), while reconstructed/ is created by BuildEigenFaces employing the eigenvectors and weights to recreate the training images, as a way of double checking the accuracy of the eigenvector calculations.

The classes involved in the build phase are shown in the UML classes diagram in Figure 14; only class names are shown.

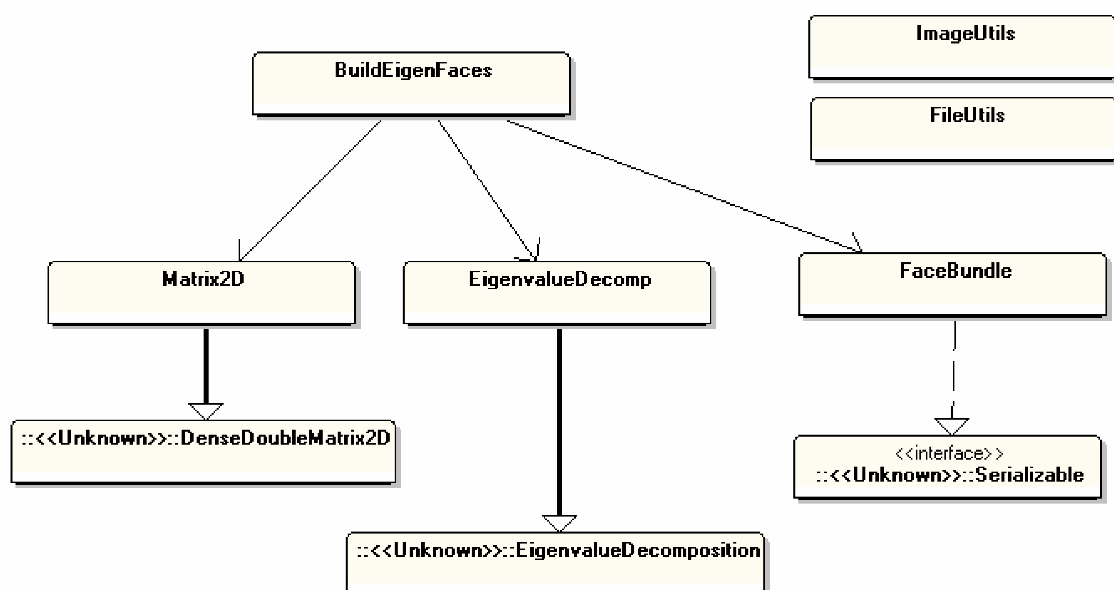


Figure 14. Classes Involved in EigenFace Building.

 © Andrew Davison 2013

Commonly used file and image manipulation methods are stored in the FileUtils and ImageUtils classes. The top-level build class is BuildEigenFaces, which uses simplified versions of the Colt DenseDoubleMatrix2D and EigenvalueDecomposition classes, by subclassing them as Matrix2D and EigenvalueDecomp. The FaceBundle class is used to create a serializable object that is stored in eigen.cache at the end of the build.

Building the Bundle

Most of the important PCA code is called from the makeBundle() method in the BuildEigenFaces class. It creates an eigenvectors/eigenvalues FaceBundle object for the specified training image files, and also saves each eigenvector as an image in eigenfaces/.

```
private static FaceBundle makeBundle(ArrayList<String> fnms)
{
  BufferedImage[] ims = FileUtils.loadTrainingIms(fnms);

  Matrix2D imsMat = convertToNormMat(ims);
                      // each row is a normalized image
  double[] avgImage = imsMat.getAverageOfEachColumn();
  imsMat.subtractMean();
    // subtract mean image from each image (row);
    // each row now contains only distinguishing features
    // from a training image

  // calculate covariance matrix
  Matrix2D imsDataTr = imsMat.transpose();
  Matrix2D covarMat = imsMat.multiply(imsDataTr);

  // calculate Eigenvalues and Eigenvectors for covariance matrix
  EigenvalueDecomp egValDecomp = covarMat.getEigenvalueDecomp();
  double[] egVals = egValDecomp.getEigenValues();
  double[][] egVecs = egValDecomp.getEigenVectors();

  sortEigenInfo(egVals, egVecs);
      // sort Eigenvectors and Eigenvars into descending order

  Matrix2D egFaces = getNormEgFaces(imsMat, new Matrix2D(egVecs));
  System.out.println("\nSaving Eigenfaces as images...");
  FileUtils.saveEFIms(egFaces, ims[0].getWidth());
  System.out.println("Saving done\n");

  return new FaceBundle(fnms, imsMat.toArray(), avgImage,
                egFaces.toArray(), egVals,
                ims[0].getWidth(), ims[0].getHeight());
}  // end of makeBundle()
```

All the images are loaded into a single array with FileUtils.loadTrainingIms(), and then converted to a 2D matrix by convertToNormMat(). It helps to have a visual idea of what this imsMat matrix contains. Assuming that each image is a N x N grayscale image, and there are M training images, then the matrix looks something like Figure 15.

 © Andrew Davison 2013

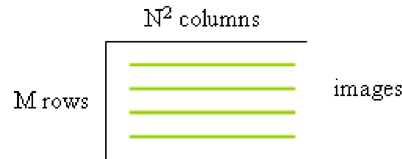


Figure 15. Visualization of the imsMat matrix in makeBundle().

One image is stored per row (which is N^2 long), and there are M rows. convertToNormMat() normalizes the matrix which simplifies the covariance and eigenvalue calculations carried out later.

The Javafaces code does not use Colt's Descriptive.covariance() method; instead it subtracts the average training image from each image, and then calculates the covariance by multiplying the image matrix to its transpose. The code employs the Turk and Pentland trick of multiplying a M x N^2 matrix to a N^2 x M matrix to produce a much smaller covariance matrix of size M x M, as illustrated in Figure 16.

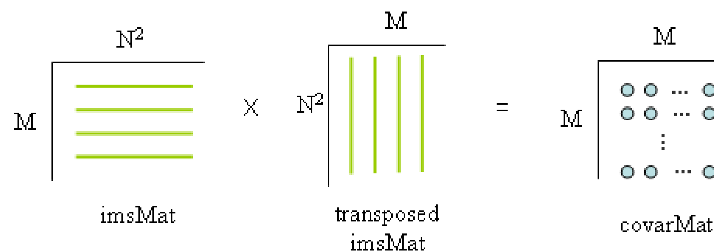


Figure 16. Calculating a M x M Covariance Matrix.

The eigenvectors and eigenvalues are calculated by calling the EigenvalueDecomp class, which is a fairly simple subclass of Colt's EigenvalueDecomposition class. The eigenvariables and eigenvectors are sorted into descending order by sortEigenInfo(), which modifies the two arrays so they look something like those in Figure 17.

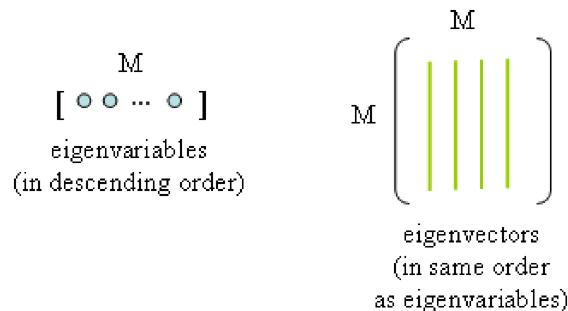


Figure 17. Eigenvariables and Eigenvectors Arrays.

 © Andrew Davison 2013

A subtle point shown by Figure 17 is that the eigenvector matrix is M x M, which means that it does **not** yet contain eigenvectors for the images (which I'll call eigenfaces from here on). Since each image is an N x N vector, then an eigenface for an image must have the dimensionality N^2. Therefore, I have to convert the M x M eigenvector matrix of Figure 17 into a M x N^2 eigenfaces matrix for the images, as illustrated by Figure 18.

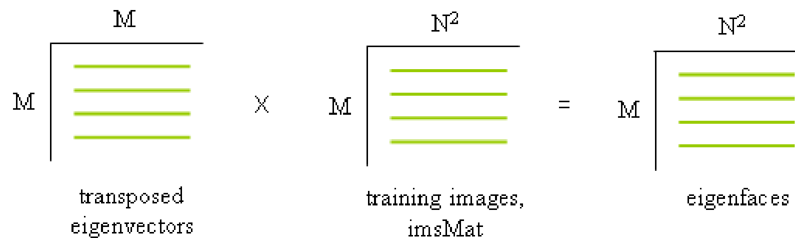


Figure 18. Converting Eigenvectors to Eigenfaces.

This transformation is done by getNormEgFaces(), which multiplies the transpose of the eigenvectors to the images matrix to get the eigenfaces.

At the end of makeBundle(), the eigenfaces are written out, each one being stored as an image in the eigenfaces/ directory (some are shown in Figure 3). More importantly, a FaceBundle object is created to hold the calculated eigenfaces, eigenvariables, and the training images.

6. Recognizing a New Image

The other major part of my modified version of Javafaces is the code to reading in a new image (new data), and deciding which of the training images it most closely resembles. The top-level class is FaceRecognizer, and uses many of the same support classes as BuildEigenFaces. The classes are shown in Figure 19.

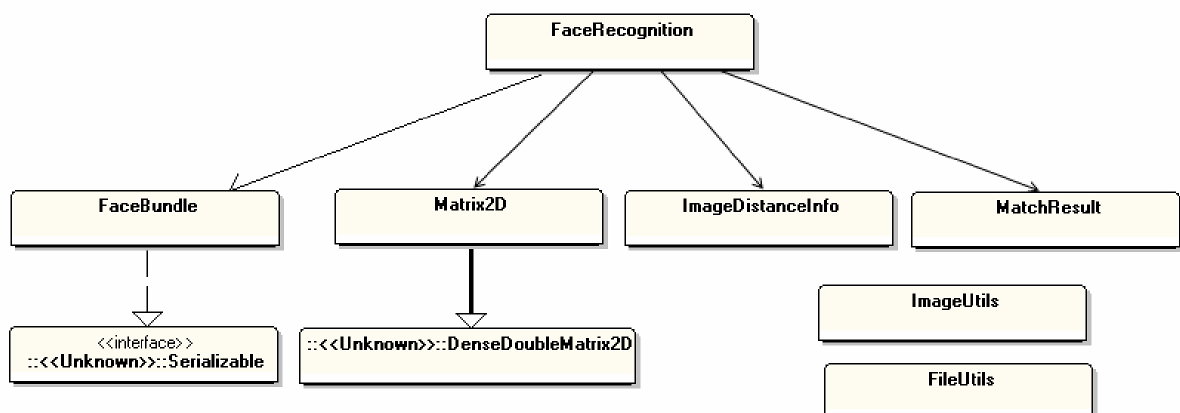


Figure 19. Classes Used by FaceRecognizer.

 © Andrew Davison 2013

FaceRecognizer is quite separate from BuildEigenFaces, but they share information via the eigen.cache file. FaceRecognizer starts by loading the cache, and converts it back into a FaceBundle object. It loads the new image using methods from the FileUtils and ImageUtils classes.

The FaceRecognizer constructor uses the calcWeights() method in FaceBundle to create weights for the training images. This is done now rather than at build-time because its only during face recognition that the user supplies the *number* of eigenfaces that will be used during the recognition process. FaceBundle.calcWeights() is listed below:

```
// in the FaceBundle class
public double[][] calcWeights(int numEFs)
{
  Matrix2D imsMat = new Matrix2D(imageRows);  // training images

  Matrix2D facesMat = new Matrix2D(eigenFaces);
  Matrix2D facesSubMatTr =
     facesMat.getSubMatrix(numEFs).transpose();

  Matrix2D weights = imsMat.multiply(facesSubMatTr);
  return weights.toArray();
}  // end of calcWeights()
```

The required number of eigenfaces is supplied in the numEFs variable, and the resulting weights array is calculated by multiplying the images matrix to a submatrix of the eigenfaces. The multiplication is illustrated in Figure 20 (recall that we are assuming that an image is N x N pixels big, and there are M training images).

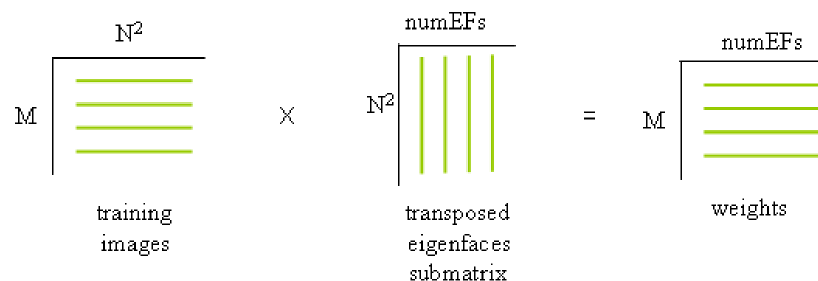


Figure 20. Creating the Training Image Weights.

One way of understanding these weights are as the new coordinates of the M training images after they have been rotated so numEFs eigenvectors align with the axes.

6.1. Finding a Match

The critical method in FaceRecognizer is findMatch() which carries out the comparison of the new image with the training images. It employs eigenfaces and eigenvectors retrieved from the FaceBundle object:

```
private MatchResult findMatch(BufferedImage im)
{
```

 © Andrew Davison 2013

```
  double[] imArr = ImageUtils.createArrFromIm(im);
         //new image → one-dimensional array

  // convert array to normalized 1D matrix
  Matrix2D imMat = new Matrix2D(imArr, 1);
  imMat.normalise();

  imMat.subtract(new Matrix2D(bundle.getAvgImage(), 1));
                    // subtract mean image
  Matrix2D imWeights = getImageWeights(numEFs, imMat);
     // map image into eigenspace, returning its coords (weights);
     // limit mapping to use only numEFs eigenfaces

  double[] dists = getDists(imWeights);
  ImageDistanceInfo distInfo = getMinDistInfo(dists);
      // find smallest Euclidian dist between image and training imgs

  ArrayList<String> imageFNms = bundle.getImageFnms();
  String matchingFNm = imageFNms.get( distInfo.getIndex() );
              // get the training image filename that is closest

  double minDist = Math.sqrt( distInfo.getValue() );

  return new MatchResult(matchingFNm, minDist);
} // end of findMatch()
```

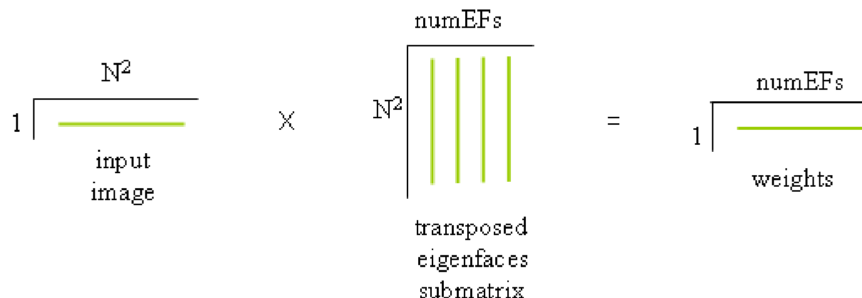
The new image is converted into a one-dimensional array of pixels of length N^2, then converted into a normalized matrix with the average face image subtracted from it (imMat). This is essentially the same transformation as that applied to the training images at the start of BuildEigenFaces.makeBundle().

The image is mapped into eigenspace by getImageWeights(), which returns its resulting coordinates (or weights) as the imWeights matrix

```
private Matrix2D getImageWeights(int numEFs, Matrix2D imMat)
{
  Matrix2D egFacesMat = new Matrix2D( bundle.getEigenFaces() );
  Matrix2D egFacesMatPart = egFacesMat.getSubMatrix(numEFs);
  Matrix2D egFacesMatPartTr = egFacesMatPart.transpose();

  return imMat.multiply(egFacesMatPartTr);
}  // end of getImageWeights()
```

Only the specified numEFs eigenfacesare used as axes. The matrix multiplication is shown in Figure 21.



 © Andrew Davison 2013

Figure 21. Creating the Input Image Weights.

The multiplication is similar to the one in FaceBundle.calcWeights(). The difference is that only a single image is being mapped in Figure 21, whereas in Figure 20 all the training images are being transformed.

findMatch() calculates the Euclidian distance between the new image and the training images by comparing the new image's weights (in imWeights) with the weights for the training images. getDists() returns an array of the sum of the squared Euclidian distances between the input image weights and all the training image weights; the function doesn't use Colt's Statistic.distance() method.

The shortest distance in the array is found with getMinDistInfo(), which also returns the index position of the corresponding training image, wrapped up inside an ImageDistanceInfo object. This index is used to lookup the name of the training image filename. Together the filename and distance are returned to the top-level as a MatchResult object.

6.2. Performing Face Recognition

The following code fragment shows how the FaceRecognizer class can be used:

```
FaceRecognition fr = new FaceRecognition(15);  // use 15 eigenfaces

MatchResult result = fr.match("andrew0.png");  // match new data
        // find a training image closest to "andrew0.png"

if (result == null)
  System.out.println("No match found");
else {  // report matching fnm, distance, name
  System.out.println();
  System.out.print("Matches image in " + result.getMatchFileName());
  System.out.printf("; distance = %.4f\n",
                                         result.getMatchDistance());
  System.out.println("Matched name: " + result.getName() );
}
```

The new data in "andrew0.png" is shown in Figure 22.

Figure 22. The "andrew0.png" Image.

The result of running the code is:

```
Using cache: eigen.cache
Number of eigenfaces: 15
```

 © Andrew Davison 2013

```
Reading image andrew0.png

Matches image in trainingImages\andrew2.png; distance = 0.4840
Matched name: andrew
```

The image most closely matched "andrew2.png" in the training data, which is shown in Figure 23.

Figure 23. The (Rather Grumpy-looking) "andrew2.png" Image.

The name, "andrew", is extracted from the filename (all filenames are made up of a name and a number), and the distance measure is 0.4840.

One problem with this approach is interpreting the meaning of the distance value. Just how close is 0.4840? Unfortunately, this will depend on the particular training image data set, and so the resulting eigenfaces will need to be tested with various images to decide whether 0.484 means "very similar" or "just barely alike".

This problem is highlighted if we run the code again, but with a face which is not in the training set. The "jim0.png" image is shown in Figure 24.

Figure 24. The "jim0.png" Image.

The output of the match code is:

```
Number of eigenfaces: 15
Matching jim0.png
Reading image jim0.png

Matches image in trainingImages\peeranut1.png; distance = 0.6684
Matched name: peeranut
```

The image has been matched with peeranut1.png, shown in Figure 25.

 © Andrew Davison 2013

Figure 25. The "peeranut1.png" Image.

Of course, this is an incorrect match, but how can we recognize that in code? All we have to go on is the distance measure, which is 0.6684. Clearly, if the distance reaches a value like this, we should report "no match". However, the exact threshold value will depend on the training set and the type of images passed for recognition.

7. Generating Training Images

The simplest way of obtaining training images is by adding "Save Face" functionality to the face tracking code of the last chapter. In fact, if you've looked at the code for NUI Chapter 7, you'll find that the necessary code is already there, but I didn't explain it in the text. Figure 26 shows the FaceTracker application in action.

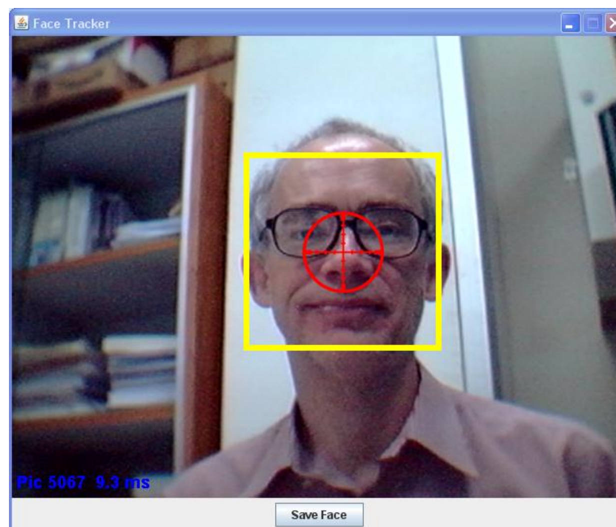


Figure 26. The FaceTracker Application of NUI Chapter 7.

The button at the bottom of the window reads "Save Face". When pressed, it sets a boolean called saveface to true by calling saveFace() in the FacePanel class:

```
// in FacePanel.java in FaceTracker (Chapter 7)
// global
private volatile boolean saveFace = false;

public void saveFace()
{  saveFace = true;  }
```

 © Andrew Davison 2013

Since saveFace will be manipulated in two different threads, it's declared as volatile.

Nothing more happens until trackFace() executes a task in the detection processing thread.

```
// in FacePanel.java in FaceTracker (Chapter 7)

private void trackFace(final IplImage img)
{
  grayIm = scaleGray(img);
  numTasks.getAndIncrement();
  executor.execute(new Runnable() {
    public void run()
    {
      detectStartTime = System.currentTimeMillis();
      CvRect rect = findFace(grayIm);
      if (rect != null) {
        setRectangle(rect);
        if (saveFace) {
          clipSaveFace(img);
          saveFace = false;
        }
      }
      long detectDuration =
              System.currentTimeMillis() - detectStartTime;
      System.out.println(" duration: " + detectDuration + "ms");
      numTasks.getAndDecrement();
    }
  });
}  // end of trackFace()
```

If a face is found in the current image, and saveFace is true, then clipSaveFace() is called to save the face to a file.

clipSaveFile() is passed the complete webcam image so it can do its own clipping, resizing, and grayscale transformations. It employs the global faceRect Rectangle for clipping.

```
// globals
private Rectangle faceRect;    // holds coords of highlighted face


private void clipSaveFace(IplImage img)
{
  BufferedImage clipIm = null;
  synchronized(faceRect) {
    if (faceRect.width == 0) {
      System.out.println("No face selected");
      return;
    }
    BufferedImage im = img.getBufferedImage();
    try {
      clipIm = im.getSubimage(faceRect.x, faceRect.y,
                              faceRect.width, faceRect.height);
    }
    catch(RasterFormatException e) {
      System.out.println("Could not clip the image");
    }
  }
```

 © Andrew Davison 2013

```
  if (clipIm != null)
    saveClip(clipIm);
}  // end of clipSaveFace()
```

faceRect is employed inside a synchronized block because it may be updated at the same time in other threads.

saveClip() resizes the clip so it's at least a standard predefined size suitable for face recognition, converts it to grayscale, and then clips it again to ensure that it is exactly a standard size. Finally the image is stored in savedFaces/.

```
// globals
// for saving a detected face image
private static final String FACE_DIR = "savedFaces";
private static final String FACE_FNM = "face";

private int fileCount = 0;
     // used for constructing a filename for saving a face


private void saveClip(BufferedImage clipIm)
{
  System.out.println("Saving clip...");
  BufferedImage grayIm = resizeImage(clipIm);
  BufferedImage faceIm = clipToFace(grayIm);
  saveImage(faceIm, FACE_DIR + "/" + FACE_FNM + fileCount + ".png");
  fileCount++;
}  // end of saveClip()
```

resizeImage(), clipToFace(), and saveImage() employ standard Java2D functionality; no use is made of JavaCV.

Once several faces have been saved, their files need to be copied over to the BuildEigenFaces application, which can utilize them as training images. It's necessary to rename the files so they consist of the person's name followed by a unique number.

8. A GUI For Recognizing a New Image

The GUI for face recognition is shown in Figure 27.

© Andrew Davison 2013

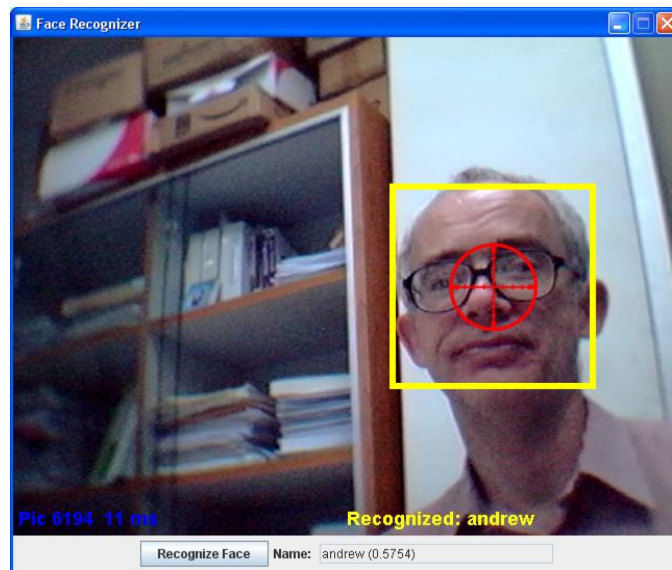


Figure 27. GUI Face Recognizer.

The code is closely based on the FaceTracker application of NUI Chapter 7, but with a different set of GUI controls at the bottom of the window. When the user presses the "Recognize Face" button, the currently selected face is passed to the FaceRecognition class, and the matching training image name and distance measure are reported. It's hard to see in Figure 27, but my name was returned with a distance value of 0.5754. This is quite high, which suggests that the match wasn't exact.

The program is a 'gluing' together of code that I've already discussed – the GUI comes from FaceTracker, while the recognition processing uses the FaceRecognition class described earlier in this chapter. As a consequence, I'm only going to explain the 'glue' where the GUI and recognizer meet.

Figure 28 contains the UML class diagrams for the application.

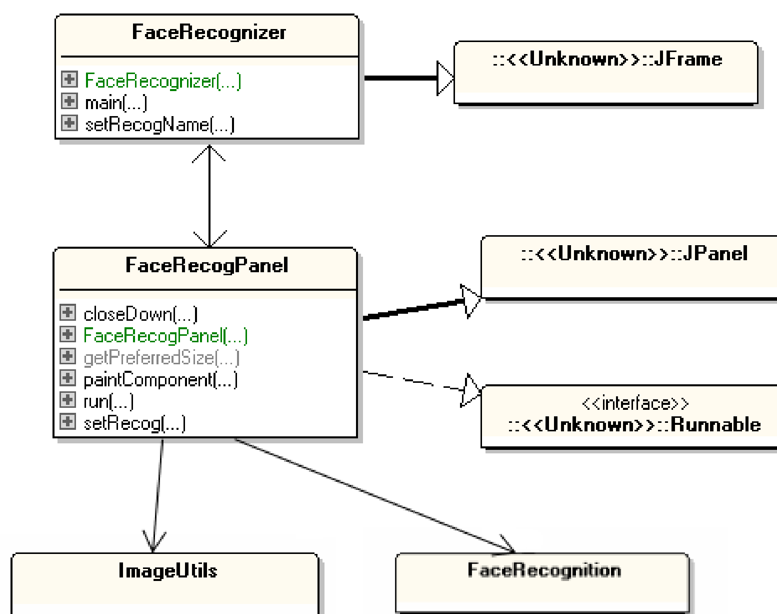


Figure 28. Class Diagrams for FaceRecognizer.

© Andrew Davison 2013

The FaceRecognizer and FaceRecogPanel classes are mostly the same as the FaceTracker and FacePanel classes from the previous chapter, but with new names.

The FaceRecognizer class adds a button and textfield to its window, while FaceRecogPanel includes code for communicating with those GUI elements, and for calling the FaceRecognition class.

The ImageUtils class contains support methods for BufferedImage manipulation, such as clipping and resizing.

Starting Recognition

FaceRecogPanel is threaded in the same way as FacePanel from the last chapter. Its run() method displays the current webcam image every 100 milliseconds, and calls trackFace() to find the face (and recognize it) in a separate thread:

```
// globals for face detection/recognition thread
private ExecutorService executor;
private AtomicInteger numTasks; // to record number of tasks
private IplImage grayIm;


private void trackFace(final IplImage img)
{
  grayIm = scaleGray(img);
  numTasks.getAndIncrement();
        // increment no. of tasks before entering queue
  executor.execute(new Runnable() {
    public void run()
    {
      detectStartTime = System.currentTimeMillis();
      CvRect rect = findFace(grayIm);
      if (rect != null) {
        setRectangle(rect);
        if (recognizeFace) {
          recogFace(img);
          recognizeFace = false;
        }
      }
      numTasks.getAndDecrement();
           // decrement no. of tasks since finished
      long detectDuration =
            System.currentTimeMillis() - detectStartTime;
      System.out.println(" duration: " + detectDuration + "ms");
    }
  });
}  // end of trackFace()
```

The important difference from the tracker code is the if-test after the call to setRectangle() which passes the entire webcam image to recogFace().

The recognizeFace boolean is set to true when the user presses the “Recognize face” button in the GUI (see Figure 27).

recogFace()'s main task is to use the clipping rectangle (previously set by setRectangle()) to cut out the face from the webcam image:

 © Andrew Davison 2013

```
// global
private Rectangle faceRect;   // holds coords of highlighted face


private void recogFace(IplImage img)
{
  BufferedImage clipIm = null;
  synchronized(faceRect) {
    if (faceRect.width == 0) {
      System.out.println("No face selected");
      return;
    }
    clipIm = ImageUtils.clipToRectangle(img.getBufferedImage(),
                 faceRect.x, faceRect.y,
                 faceRect.width, faceRect.height);
  }
  if (clipIm != null)
    matchClip(clipIm);
}  // end of recogFace()
```

faceRect is employed inside a synchronized block because it may be updated at the same time in other threads.

recogFace() may remind you of clipSaveFace() in the previous section, but instead of saving the clip, the clip is passed for recognition to matchClip().

```
// globals
private FaceRecognizer top;
private FaceRecognition faceRecog;
         // this class comes from the JavaFaces example
private String faceName = null;
           // name associated with last recognized face


private void matchClip(BufferedImage clipIm)
 {
  long startTime = System.currentTimeMillis();

  System.out.println("Matching clip...");
  BufferedImage faceIm = clipToFace( resizeImage(clipIm) );
  MatchResult result = faceRecog.match(faceIm);
  if (result == null)
    System.out.println("No match found");
  else {
    faceName = result.getName();
    String distStr = String.format("%.4f",
                          result.getMatchDistance());
    System.out.println("  Matches " + result.getMatchFileName() +
                                 "; distance = " + distStr);
    System.out.println("  Matched name: " + faceName);
    top.setRecogName(faceName, distStr);
  }
  System.out.println("Match time: " +
       (System.currentTimeMillis() - startTime) + " ms");
}  // end of matchClip()
```

matchClip() starts in a similar way to saveClip() from the previous section, by resizing and clipping the image. However, instead of calling saveImage() it passes the

 © Andrew Davison 2013

picture over to the FaceRecognition object, faceRecog. This FaceRecognition class is unchanged from my earlier description, and its object is created in FaceRecogPanel's constructor:

```
// part of FaceRecogPanel()
faceRecog = new FaceRecognition();
    // no numerical argument means a default number
    // of eigenfaces will be used
```

FaceRecognition.match() returns a MatchResult object which contains a name and distance measure. These are used in three ways: a global string, faceName, is assigned the returned name, the name and distance are printed to stdout, and are also passed to the top-level GUI via a call to FaceRecognizer.setRecogName(). setRecogName() writes the two values into the GUI's textfield (see Figure 27).

The global faceName string is utilized by the panel at rendering time to draw the name in large yellow text at the bottom of the image (see Figure 27). The relevant function is writeName() which is called by the panel's paintComponent():

```
private void writeName(Graphics2D g2)
// draw the currently recognized face name onto the panel
{
  g2.setColor(Color.YELLOW);
  g2.setFont(msgFont);
  if (faceName != null)  // draw at bottom middle
    g2.drawString("Recognized: " + faceName, WIDTH/2, HEIGHT-10);
}  // end of writeName()
```

The matchClip() function includes time-measurement code for printed the recognition time to stdout. On my slow test machine this averages about 300 ms. As we saw in the previous chapter, face detection takes about 100 ms, and so there is little point trying to perform detection and recognition more frequently than every 400 ms or so.

The frequency of face processing is controlled by a DETECT_DELAY constant used in FaceRecogPanel's run():

```
// in FaceRecogPanel.run()

if (((System.currentTimeMillis()-detectStartTime) > DETECT_DELAY) &&
    (numTasks.get() < MAX_TASKS))
  trackFace(im);
```

The current setting for DETECT_DELAY is 1000 ms (1 second), which means that a recognition task is initiated roughly every second. This easily exceeds the worst processing time, and so the number of pending recognition tasks should be very small or even 0. Even if too many face tracking tasks are initiated, they will patiently wait their turn in the queue for the ExecutorService without impacting the rendering speed of the image pane (as explained in the previous chapter).

© Andrew Davison 2013