SECTION SECTION 3 Dimension Reduction

Curse of Dimensionality (COD)

Suppose that we have a one variable data set with 1,000,000 cases. If we want a histogram with our variable divided into 10 intervals (bins) we would have, on average, 100,000 cases/bin.
Now suppose the data is in two dimensions. If we want to have each variable divided into 10 intervals we would have 100 bins and, on average, 10,000 cases/bin.

Dim	Bins	Ave. Cases/bin
1	10	100,000
2	100	10,000
3	1,000	1,000
4	10,000	100
5	100,000	10
6	1,000,000	1
7	10,000,000	0.1
$\vdots$	$\vdots$	$\vdots$
10	10,000,000,000	0.0001

We see that the data is sparsely distributed in the bins.

It is also interesting to consider the nature of the hypercubes. If we have a box in 3 dimensions
$[(1,1,1),(1,1,-1),(1,-1,-1),(1,-1,1),(-1,1,1),(-1,1,-1),(-1,-1,-1),(-1,-1,1)]$, the distance from $(0,0,0)$ to any corner is $\sqrt{3}$, while in 4 dimensions it is $\sqrt{4}$, etc. As a result, the distance from the centre to a corner is increasing, while the distance from the centre to an edge is constant (1) so a greater proportion of the volume will be in the corners. Almost every point is closer to an edge than to another point. This sparseness problem is commonly called the ”curse of dimensionality” (COD).

Since we use samples to estimate an unknown function, our estimates may be inaccurate (biased). Meaningful estimation is possible only for sufficiently *smooth* functions but sparseness of high-dimensional space makes it difficult to collect enough samples to attain a high enough density to ensure a sufficiently smooth function. Smoothness constraints describe how individual cases in the training data are combined by the learning method in order to construct the function estimate. Accurate function estimation depends on having enough cases within the neighbourhood specified by the smoothness constraints. As the number of dimensions increases, the number of cases needed to give the same density increases exponentially. This could be offset by increasing the neighbourhood size with dimensionality (increasing the number of cases falling within the neighbourhood) but this is at the expense of imposing stronger (and possibly incorrect) constraints. Low data density requires us to specify stronger, more accurate constraints on the problem solution.

The COD is due to the geometry of high-dimensional spaces. A large radius is needed to enclose a fraction of the data points in a high-dimensional space (see above example). For a given fraction of cases, it is possible to determine the edge length of the hypercube using

$$e_d(p) = p^{1/d}$$

where p is the (prespecified) fraction of cases. In a 10-dimensional space ($d = 10$) to enclose 10% of the cases, the edge length is $e_{10}(0.1)) = 0.80$. Thus very large neighbourhoods are needed to capture even a small portion of the data. For a sample of size n, the expected L_∞ distance between data points is

$$D(d,n) = 1/2(1/n)^{1/d}$$

so for a 10-dimensional space, $D(10, 1000) \approx 0.5$ and $D(10, 10,000) \approx 0.4$.

Thus there are serious problems associated with making local estimates for high-dimensional samples and a lot of extrapolation will be required.

4.1 Reducing Dimensionality

In data that has very high dimensions, it can be important to reduce the effective dimension of the data to enable us to employ some methods that work better in lower dimensions. For example, doing an All Subsets Regression on 12 variables will be far easier than on 125 variables.

An example of dimension reduction

Consider data points at $(1,1),\ (2,2),\ (3,3),\ (4,4),\ (5,5)$.

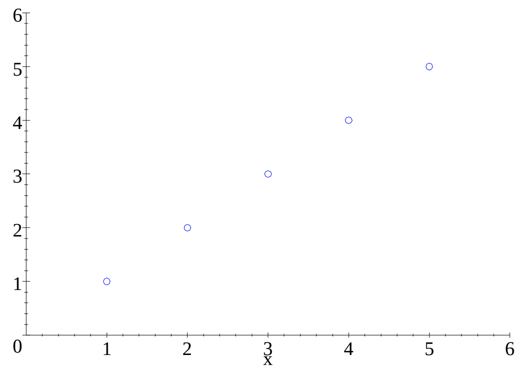

Figure 1.

These points are specified in terms of the two orthonormal vectors $\mathbf{e}_1 = (1,0)$ and $\mathbf{e}_2 = (0,1)$.
What happens if we instead use the two orthonormal vectors $\mathbf{e}_1' = \left(1/\sqrt{2}, 1/\sqrt{2}\right)$ and $\mathbf{e}_2' = \left(-1/\sqrt{2}, 1/\sqrt{2}\right)$? This gives us a new set of axes as shown.

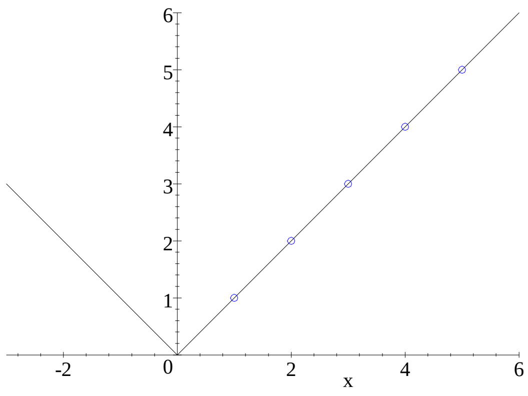

Figure 2.

Because the points all lie on the basis vector $\mathbf{e}_1'$ we can ignore the other basis vector for our new coordinate system. This results in a reduction of the dimension of our dataset.
What happens if some or all of the data points are not exactly on the line? If they are not too far off, we may feel that we can ignore the slight difference and represent the data in terms of just one vector.

4.2 Principal Components Analysis (PCA)

PCA is one of the standard methods for dimension reduction.

When one looks at data with a scatterplot matrix, the only structure that can be seen is that which is visible from the original coordinate axes and is restricted to relationships between two variables. If we look at a scatterplot matrix and find that every scatterplot that involved one of the variables is virtually a horizontal or vertical straight line then we would conclude that we could model the data without using that variable. We might find, if we rotate the data, that a similar behaviour could be seen for ***linear combinations of the variables***. PCA is a method that enables us to see if such structure exists.

Consider our data as random variables $X_{1,}...,X_p$ with n observations for each of these random variables. Principal components are special linear combinations of the p random variables. These linear combinations represent a *new coordinate system* that is obtained by rotating the original system that had $X_{1,}...,X_p$ as the coordinate axes. The new axes represent directions of variability and provide a simpler, more parsimonious description of the covariance structure of the original data. Principal components depend solely on the covariance matrix $\sum$ (or the correlation matrix $\boldsymbol{\rho}$) of $X_{1,}...,X_p$.

Specifically, *PCA* looks at variance in the data and identifies the mutually orthogonal directions of decreasing variance. In *PCA* we form as many new variables as we have original variables. The new variables are linear combinations of the old variables. But they are chosen in such a way that the first linear combination ($PC1$) explains the highest proportion of the variance in the original variables. The second linear combination ($PC2$) is orthogonal to the first and it explains the second largest proportion of variance in the original variables. The third linear combination ($PC3$) is chosen to be orthogonal to the first two and it explains the third largest proportion of the variance in the original variables, etc.

Let the random vector $\mathbf{X}^T = [X_{1,}...,X_p]$ have covariance matrix

$$\sum = \begin{bmatrix} var(X_1) & cov(X_1,X_2) & \cdots & cov(X_1,X_p) \\ cov(X_2,X_1) & var(X_2) & \cdots & cov(X_2,X_p) \\ \vdots & \cdots & \ddots & \vdots \\ cov(X_p,X_1) & \cdots & \cdots & var(X_p) \end{bmatrix}$$

with eigenvalues $\lambda_1 \geq ... \geq \lambda_p \geq 0$ and associated eigenvectors. We shall see that eigenvalues and eigenvectors have specific statistical interpretations. Consider p linear combinations of the original p variables, i.e.

$$Y_i = \boldsymbol{\ell}_i^T\mathbf{X}, i = 1,...,p$$

Then

$$var(Y_i) = \boldsymbol{\ell}_i^T \sum \boldsymbol{\ell}_i$$

and

$$cov(Y_i, Y_k) = \ell_i^T \sum \ell_k \quad \text{for } i, k = 1, ..., p$$

To eliminate indeterminacy, we restrict ourselves to coefficient vectors of length one. Hence $PC1$ (i.e. Y_1) is the linear combination $\ell_1^T\mathbf{X}$ that maximizes $var(\ell_1^T\mathbf{X})$ subject to

$$\ell_1^T \ell_1 = 1;$$

$PC2$ (i.e. Y_2) is the linear combination $\ell_2^T\mathbf{X}$ that maximizes $var(\ell_2^T\mathbf{X})$ subject to

$$\ell_2^T \ell_2 = 1$$

and

$$cov(Y_1, Y_2) = 0$$

etc.

We can show that

$$max_{\ell \neq \mathbf{0}} \frac{\ell^T \sum \ell}{\ell^T \ell} = \lambda_1 = Var(Y_1)$$

and is attained when $\ell = \mathbf{e}_1$ and

$$max_{\ell \perp \mathbf{e}_1, ..., \mathbf{e}_k} \frac{\ell^T \sum \ell}{\ell^T \ell} = \lambda_{k+1} (= Var(Y_{k+1})) \quad \text{for } k = 1, 2, ..., p-1$$

and

$$\sum_{i=1}^{p} var(X_i) = \sum_{i=1}^{p} var(Y_i).$$

We have thus constructed independent components that are conserving the total variance in the dataset but we have not yet reduced the dimension. To reduce dimension we may drop the latter *PC*s which explain less of the variance in the original data.

We can illustrate on the simple example (Figure 1) described above.

```
library(stats)
A <- cbind(1:5,1:5)
(A.pc <- prcomp(A))
Standard deviations:
[1] 2.236068e+00 1.431424e-16
Rotation:
           PC1        PC2
[1,] 0.7071068 -0.7071068
[2,] 0.7071068  0.7071068
summary(A.pc)
Importance of components:
                        PC1 PC2
Standard deviation     2.24  1.43e-16
Proportion of Variance 1.00  0.00e+00
Cumulative Proportion  1.00  1.00e+00
(A.new <- A%*%A.pc$rotation[,1])
         [,1]
[1,] 1.414214
[2,] 2.828427
[3,] 4.242641
[4,] 5.656854
[5,] 7.071068
A.new%*%A.pc$rotation[,1]
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    3
[4,]    4    4
[5,]    5    5
```

We can see that principal components produce the transformation that we expected and allow us to obtain the coordinates in the new coordinate system that corresponds to the eigenvectors (note $PC1$ is $\mathbf{e}_1'$ and $PC2$ is $\mathbf{e}_2'$), as well as the process for returning to the original coordinate system.

Now consider a more realistic problem.

Consider the following 2-dimensional ellipse, with 500 points from a random uniform distribution:

```
numb<- 500             # data set with 500 points
a <- 10                # semi-major axis
b <- 5                 # semi-minor axis
x <- runif(numb,-a,a)  # x is random uniform in [-a, a] i.e. U[-10,10]
y <- matrix(0,1,numb)
for (i in (1:numb)) {
   aa <- b*(1 - (x[i]/a)^2)^(1/2)
   y[i] <- runif(1,-aa,aa)        # a random number in U[-aa, aa]
}
y <- as.vector(y)
plot(x, y, pch = 20, main = "Ellipse? - default scaling", cex = 1.5)
   # To make it look like an ellipse.
plot(x,y, pch = 20, asp = 1, main = "Ellipse with correct scaling", cex = 1.5)
```

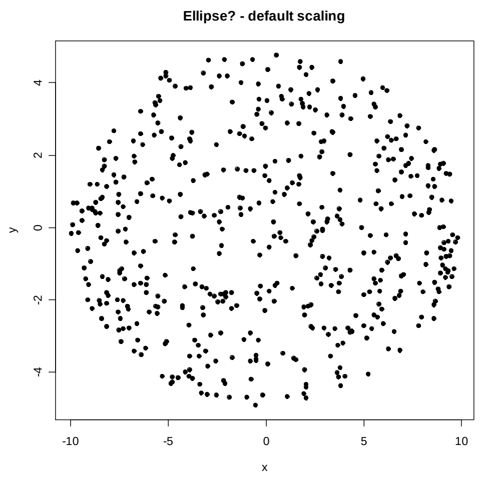

Figure 3.

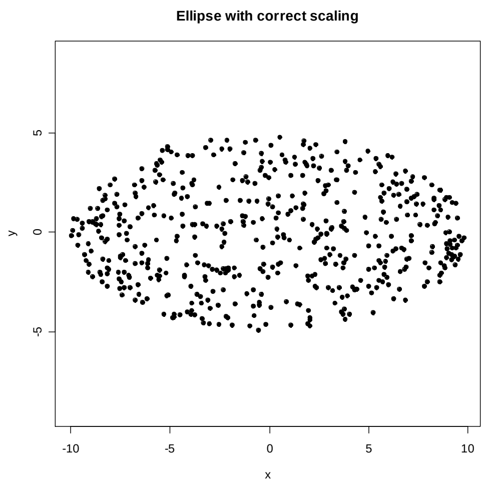

Figure 4.

Note: One concern in plotting data is ensuring that the **scaling** on the axes is correct. The left hand graph is the *default* plotting of an elliptic cloud. It looks like a circle because the usual default on plotting is to fill the graphic window as much as possible. The right hand graph is a better representation because it has the same scaling units on both axes.

It is possible that a figure window may be covered by other windows. When that happens, changes to the figure will be done but not be seen. The use of the command

```
bringToTop(which = dev.cur())
```

can assist by either displaying the figure or flashing it on the taskbar.

Now set up the directories

```
drive <- "D:"
code.dir <- paste(drive, "DataMining/Code", sep = "/")
data.dir <- paste(drive, "DataMining/Data", sep = "/")
```

Suppose we now rotate our elliptic cloud by $\pi/3$ and display it with the original.

```
(R <- cbind(c(cos(pi/3), sin(pi/3)),c(sin(pi/3), -cos(pi/3))))
          [,1]       [,2]
[1,] 0.5000000  0.8660254
[2,] 0.8660254 -0.5000000
Z <- cbind(x,y)
XX <- Z%*%R
```

Save the data the first time and read it in the rest of the time so that we can replicate results.

```
# write.table(XX, row.names = F, col.names = F, quote = F, file = paste(data.dir,
"PC_XX.dat", sep = ""))
XX <- read.table(paste(data.dir, "PC_XX.dat", sep = "/"))
```

We need to convert the data frame to a matrix to allow multiplication

```
Z <- as.matrix(XX)%*%R
plot(Z, pch = 20, asp = 1, main = "Rotated ellipse (red) with original")
points(XX, pch = 20, col = "red")
```

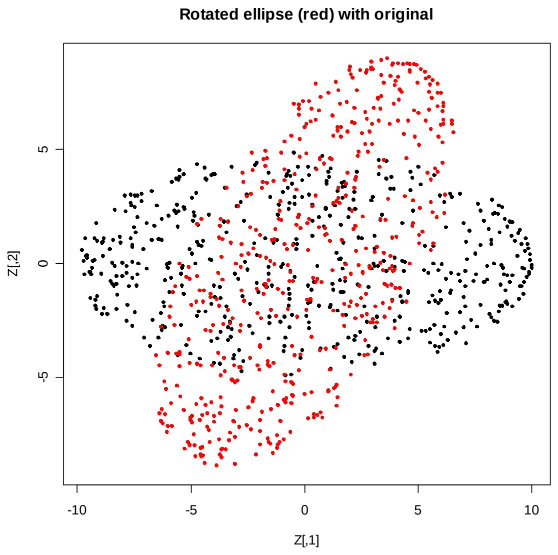

Figure 5.

It is obvious that the larger variance was along the *x*-axis; now it will be along a line at an angle of $\pi/3$.

We can find the principal components of this data using the Singular Value Decomposition (*SVD*) of the covariance matrix $\sum$ of the data.

```
(pc.1 <- svd(cov(XX)))
sqrt(pc.1$d)
```

The Singular Value Decomposition is

```
$d
[1] 33.306878 5.528184
$u
           [ ,1]       [ ,2]
[1,] -0.4774353 -0.8786669
[2,] -0.8786669 0.4774353
$v
           [ ,1]       [ ,2]
[1,] -0.4774353 -0.8786669
[2,] -0.8786669 0.4774353
```

The standard deviations are

```
[1] 5.771211 2.351209
```

Alternatively, using the `eigen` command, we can obtain the principal components of the ‘new’ data set.

```
(pc.2 <- eigen(cov(XX)))
$values
[1] 33.306878 5.528184
$vectors
          [ ,1]        [ ,2]
[1,] -0.4774353 -0.8786669
[2,] -0.8786669  0.4774353
```

(We might get different signs because the direction along the vector is not specified.)

We can also use the principal components method that is given in the library `stats` . It computes other quantities as well.

```
(pc.3 <- prcomp(XX))
Standard deviations:
[1] 5.771211 2.351209
Rotation:
PC1 PC2
V1 -0.4774353 -0.8786669
V2 -0.8786669  0.4774353
plot(pc.3, main = ”Scree plot for ellipse”, col = c(”red”,”yellow”))
```

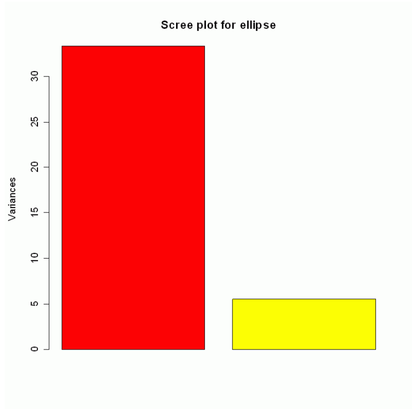

Figure 6.

The ***scree*** plot gives an idea of the relative importance of the principal components since it plots the variance (i.e.the eigenvalues) explained by each successive principal component. Note that they will necessarily be in decreasing order.

```
summary(pc.3)
Importance of components:
                         PC1   PC2
Standard deviation     5.771 2.351
Proportion of Variance 0.858 0.142
Cumulative Proportion  0.858 1.000
```

Now we can plot the principal axes (i.e. the eigenvectors) on the data. (`abline` plots lines using the slope and intercept.)

```
plot(XX, pch = 20, asp = 1, main = "Ellipse with principal axes", cex = 1.5)
A <- diag(c(1, 1))
PCaxes <- A%*%pc.3$rotation
abline(0,pc.3$rotation[2,1]/pc.3$rotation[1,1], col = "red")
abline(0,pc.3$rotation[2,2]/pc.3$rotation[1,2], col = "blue")
m.1 <- pc.3$rotation[2,1]/pc.3$rotation[1,1]
m.2 <- pc.3$rotation[2,2]/pc.3$rotation[1,2]
for (i in sample(1:500,10)) {
   x.1 <- XX[i,1]
   y.1 <- XX[i,2]
   points(x.1, y.1, pch = 15,col = "blue")
   x.2 <- (y.1 - m.2*x.1)/(m.1 - m.2)
   y.2 <- m.1*x.2
   lines(c(x.1,x.2), c(y.1,y.2), col = "blue")
}
for (i in sample(1:500,5)) {
   x.1 <- XX[i,1]
   y.1 <- XX[i,2]
   points(x.1, y.1, pch = 15,col = "red")
   x.2 <- (y.1 - m.1*x.1)/(m.2 - m.1)
   y.2 <- m.2*x.2
   lines(c(x.1,x.2), c(y.1,y.2), col = "red")
}
```

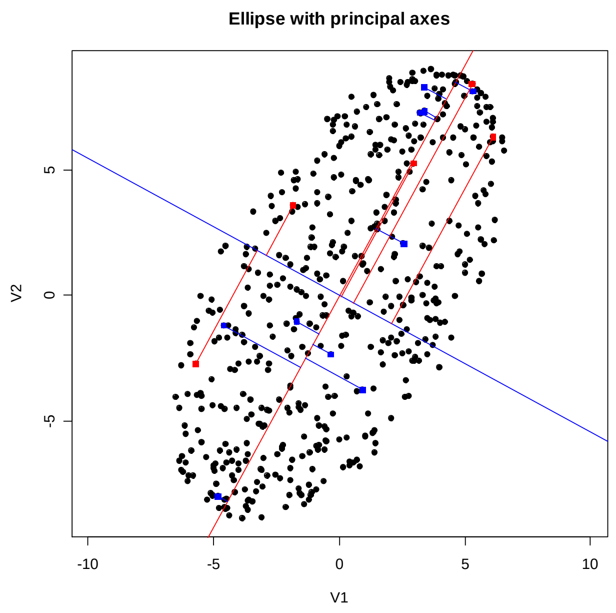

Figure 7. The principal axes are plotted on the cloud (red for the 1st component). The variance is computed using the sum of the squares of the perpendicular distances from the point to the component axes.

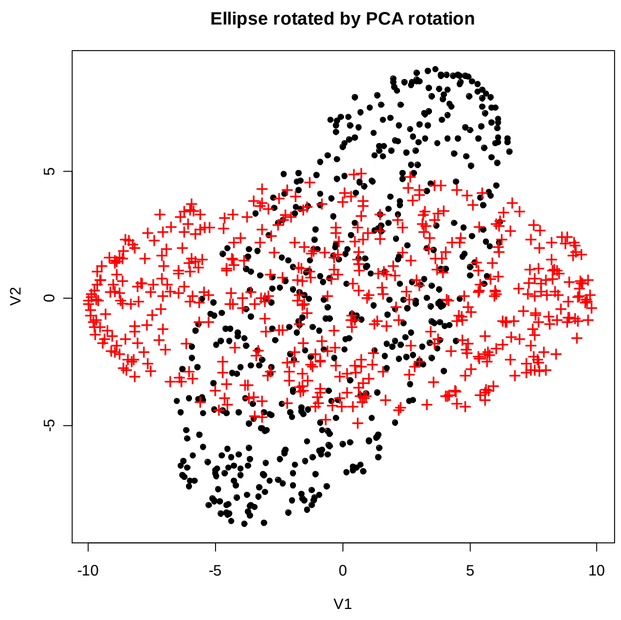

Figure 8. The red cloud is the cloud after we have used the principal component rotation to make the 1st principal component axis horizontal.

```
XXR <- as.matrix(XX)%*%PCaxes
plot(XX, pch = 20, asp = 1, main = "Ellipse rotated by PCA rotation", cex = 1.5)
points(XXR, col = "red", pch = "+", cex = 1.5)
```

We need to use `as.matrix(XX)` to convert the data frame values to a matrix.

To see how well PCA determined the orientation of the ellipse, we will compare our known rotation that we applied to the data with the rotation obtained from PCA.
The original ellipse was rotated using the matrix

```
R
          [ ,1]       [ ,2]
[ 1,] 0.5000000  0.8660254
[ 2,] 0.8660254 -0.5000000
```

```
PCaxes
```

The Principal Components rotation is

```
             PC1        PC2
  [ 1,] -0.4774353 -0.8786669
  [ 2,] -0.8786669  0.4774353
```

It is possible that the first principal component (*PC*1) could be an adequate summary of the total variance in the data provided we feel that the deviations from that line are within our acceptable level of error. (i.e. Does the first Principal Component *PC*1 explain a sufficiently high proportion of the total variance?) In that case we would have reduced the dimension from two to one.

One difficulty is that by computing principal component variables *PC*1, *PC*2, etc., we are computing linear combinations of the original variables so we have probably moved away from physical variables (which have an interpretation) to linear combinations of variables which may have no interpretation. Note that we have not reduced the dimension of our data - we have as many new principal component variables as we had original variables- but if we are willing to **discard** some of the new principal component variables because they account for a very small proportion of the total variance in the dataset, we can reduce the dimension of our problem. We may then be able to use methods that apply to lower dimensional data. Keep in mind that we will then be using the first few principal components (which are linear combinations of the original variables) so we may lose some ability to interpret results.

Consider the flea beetle example.

Read in files required for a flea beetle example.

```
source(paste(code.dir, ”ReadFleas.r”, sep = ”/”))
```

PCA gives-

```
(pc.flea <- prcomp(d.flea))
Standard deviations:
[ 1] 31.108528 15.053393  5.976129  5.079179  1.973860  1.095499
Rotation:
              PC1         PC2         PC3         PC4         PC5          PC6
tars1 -0.93222707 -0.32027041  0.14174096  0.06777484  0.02191107 -0.056705703
tars2  0.02339181 -0.43942897 -0.87177026  0.12101445 -0.17802317 -0.005905547
head   0.01753944 -0.12027844 -0.13020947 -0.02687343  0.90584022  0.383408753
aede1  0.15580879 -0.53893004  0.17396789 -0.80548829 -0.04659334 -0.063489191
aede2 -0.05351583 -0.01036395  0.06991407 -0.03878271 -0.38068526  0.919627805
aede3  0.32087002 -0.63190912  0.40965694  0.57421634 -0.01374570 -0.001067120
```

```
plot(pc.flea, col = heat.colors(6))
```

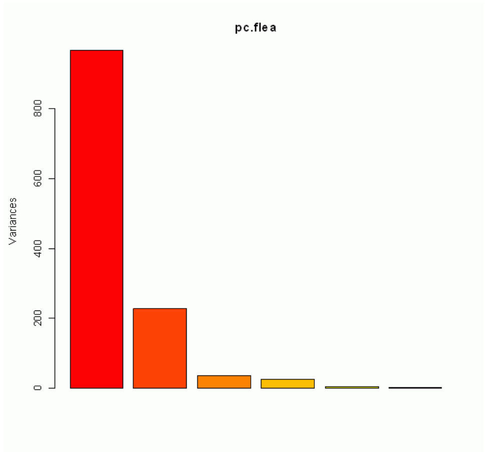

Figure 9.

We see that the first principal component is quite dominant and is

$$PC1 = -0.93\,tars1 + 0.023\,tars2 + 0.018\,head + 0.16\,aede1 - 0.054\,aede2 + 0.32\,aede3$$

(Recall that the importance of *tars*1 and *aede*3 was seen in Ggobi.)

Consider an example that has 256 variables.

For automated mail sorting, the U. S. Postal Service needs to be able to convert Zip Codes (machine-produced or handwritten) into the corresponding digits. [http://www-stat.stanford.edu/~tibs/ElemStatLearn/datasets/zip.info]. It does so by doing a scan that converts the digit (in the form of an image) to grayscale values on a grid. For example using a 16x16 grid, each image would be represented by 256 variables (the intensity of each pixel). For machine-produced digits, this intensity would be quite uniform for each image and the pattern for each digit would be distinct, enabling accurate automatic reading of the digit. Handwritten digits tend to be quite variable (see below). In order to get automatic recognition of handwritten digits, the handwritten digits were scanned and converted to grayscale values on a 16×16 grid. The goal is to determine characteristics associated with each digit in order to identify the handwritten digit correctly.

We read the data for all 10 digits into a set of *number of cases* x 256 matrices in such a way that the greyscale values for the first digit are placed in the first row, the second digit in the second row, etc.

```
d.file <- {}
d.digits <- c({})
for (i in 0:9) {
   d.file[i+1] <- paste(data.dir, "/train_", i, ".dat", sep = "")
   d.digits[[i+1]] <- matrix(scan(d.file[i+1], sep = ","), ncol = 256,
                                        byrow = T)
}
```

The number of cases varies

```
(num.cases <- unlist(lapply(d.digits, dim))[seq(1,20,2)])
```

```
[1] 1194 1005  731  658  652  556  664  645  542  644
```

Consider the first 144 cases of a few handwritten digits in the dataset.
The `layout` command allows us to create a matrix of images (in this case 12 x 12) and the `byrow = TRUE` indicates that the first 12 images go in the first row, the second 12 in the second row, etc.

The `par(mar = c(0,0,0,0))` command specifies that there will be no margins around the images.

The `matrix(digits[i,],16,16)[,16:1]` command takes each row of `digits`, places them in a 16 x 16 matrix, and re-orders the columns with the `[,16:1]` expression.

The `image` command plots a matrix of values. `col = gray((255:0)/255)` determines the ‘blackness’ - try different values. See `?image`.)

```
plot.digits <- function(digits) {
   x11(width = 6, height = 6)    # Open a graphics window of given size
        # Create a plot matrix with 144 subplots - plot in row-wise order
   layout(matrix(1:144, 12, 12, byrow = TRUE))
        # No margin (see ?par)
   oldpar <- par(mar = c(0,0,0,0))
   for (i in 1:144) {
          # xaxt = "n", yaxt = "n" - no axes
      image(matrix(digits[i,],16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
   }
   par(oldpar)
}
```

We now plot the representations of the digits ‘2’, ‘3’, ‘5’, ‘8’ (the `+1` appears because the digit 0 is in d.digits[1]).

```
plot.digits(d.digits[[2+1]])
plot.digits(d.digits[[3+1]])
plot.digits(d.digits[[5+1]])
plot.digits(d.digits[[8+1]])
```

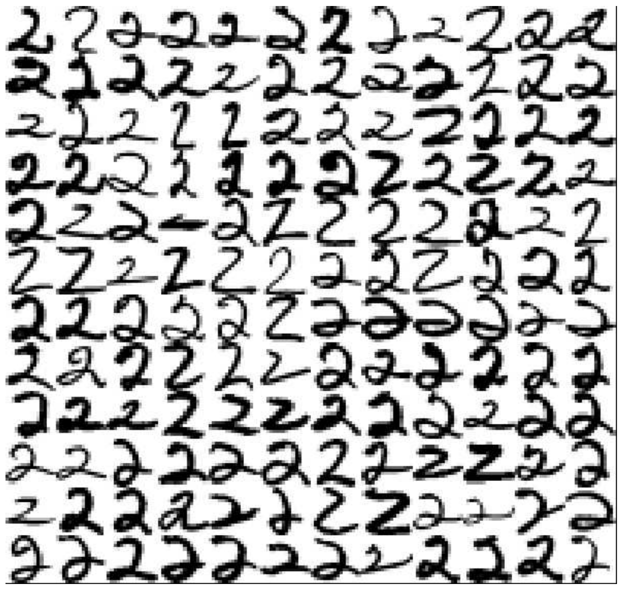

Figure 11.

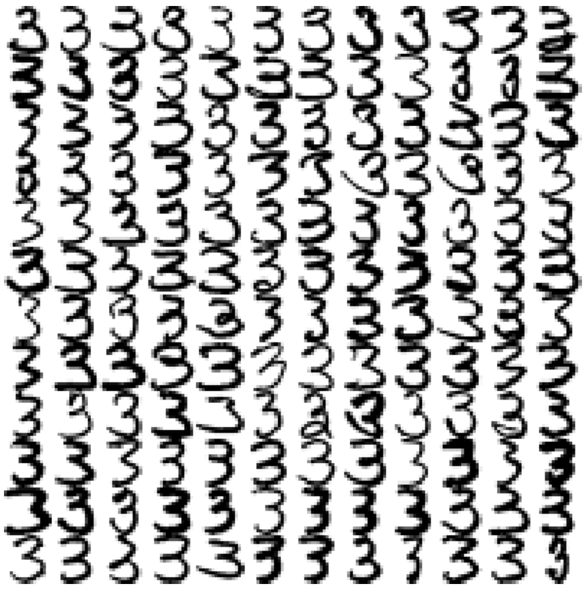

Figure 12.

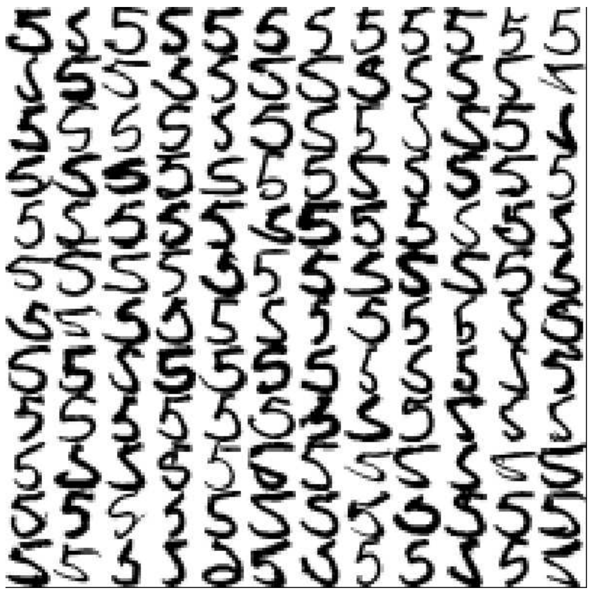

Figure 13.

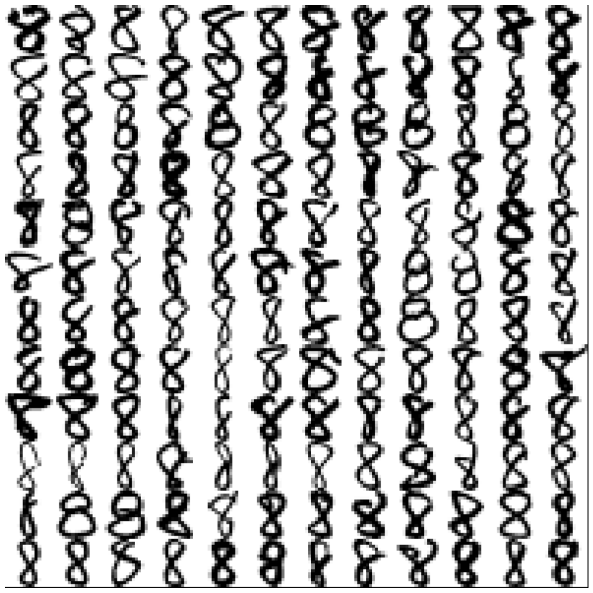

Figure 14.

It is obvious that there is a lot of variation in the shape and thickness.

```
library(stats)
graphics.off()
pc.digits <- {}
for (i in 0:9) {
   pc.digits[[i+1]] <- prcomp(d.digits[[i+1]])
   plot(pc.digits[[i+1]], col = heat.colors(10), main = i)
   print(summary(pc.digits[[i+1]]))
   readline("Press return..")
}
```

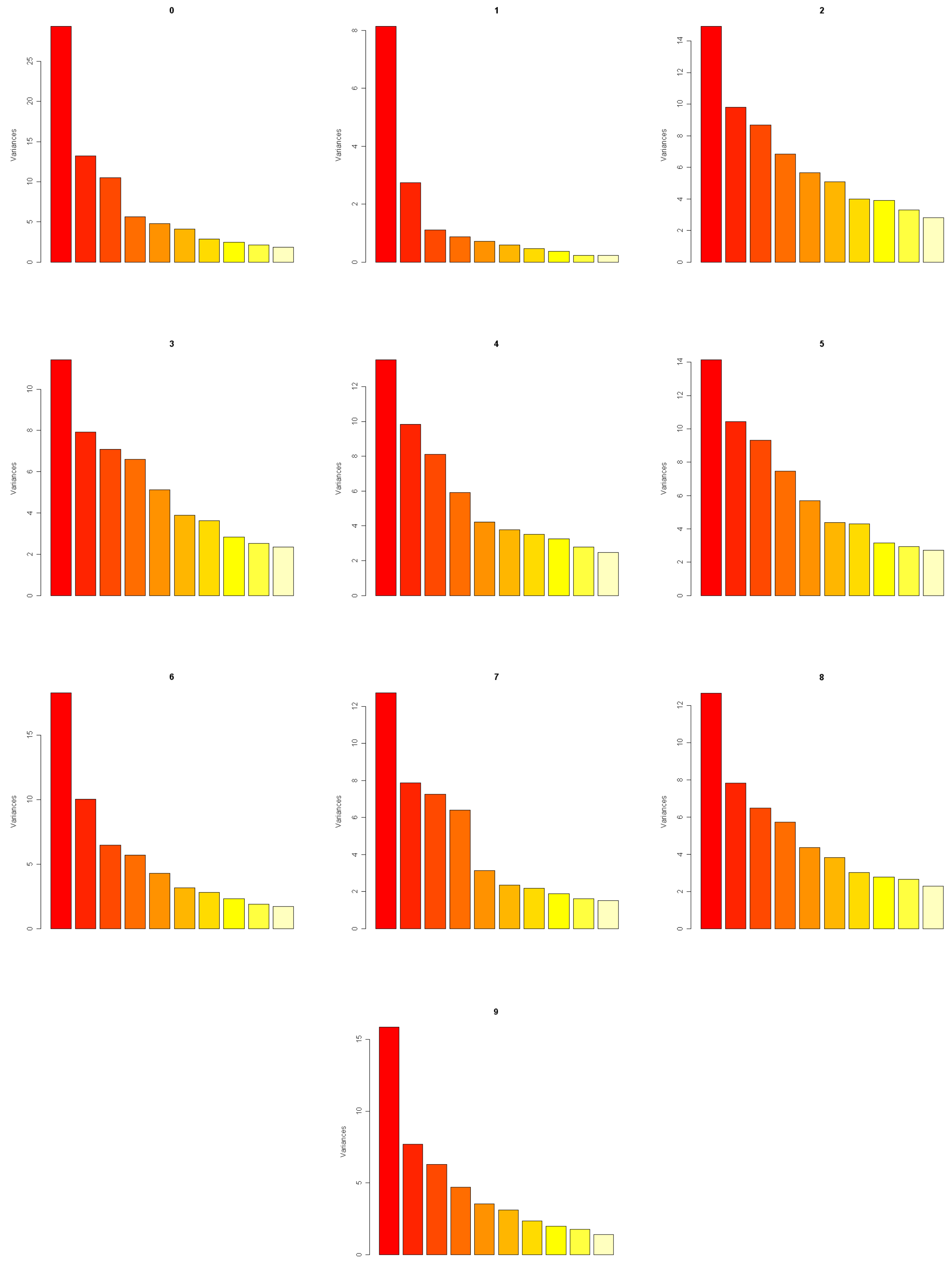

Figure 16. Principal Components for 0 - 9

This shows the first 10 principal components. Let us look at the principal components as displayed for the digit 3 by `summary(pc.digits[[i+1]])`.

```
Importance of components:
                         PC1   PC2    PC3    PC4    PC5    PC6    PC7    PC8
Standard deviation     3.379 2.816 2.6599 2.5680 2.2613 1.9724 1.9066 1.6835
Proportion of Variance 0.127 0.088 0.0785 0.0732 0.0567 0.0432 0.0403 0.0314
Cumulative Proportion  0.127 0.215 0.2931 0.3663 0.4230 0.4662 0.5065 0.5379
                          PC9   PC10   PC11   PC12   PC13   PC14   PC15   PC16
Standard deviation     1.5851 1.5287 1.4235 1.3358 1.3052 1.2407 1.1724 1.0906
Proportion of Variance 0.0279 0.0259 0.0225 0.0198 0.0189 0.0171 0.0153 0.0132
Cumulative Proportion  0.5658 0.5917 0.6142 0.6340 0.6529 0.6700 0.6852 0.6984
                         PC17   PC18   PC19    PC20    PC21    PC22    PC23
Standard deviation     1.0706 1.0067 0.9864 0.94142 0.89733 0.89487 0.86294
Proportion of Variance 0.0127 0.0112 0.0108 0.00983 0.00893 0.00888 0.00826
Cumulative Proportion  0.7111 0.7224 0.7331 0.74297 0.75190 0.76078 0.76904
                         PC24    PC25    PC26    PC27    PC28    PC29    PC30
Standard deviation     0.8388 0.82166 0.80241 0.77612 0.75564 0.73613 0.73355
Proportion of Variance 0.0078 0.00749 0.00714 0.00668 0.00633 0.00601 0.00597
Cumulative Proportion  0.7769 0.78434 0.79148 0.79816 0.80449 0.81051 0.81647
                          PC31    PC32    PC33    PC34    PC35    PC36   PC37
Standard deviation     0.72223 0.69970 0.69593 0.67441 0.65674 0.64538 0.6370
Proportion of Variance 0.00579 0.00543 0.00537 0.00505 0.00478 0.00462 0.0045
Cumulative Proportion  0.82226 0.82769 0.83306 0.83811 0.84289 0.84751 0.8520
                         PC38   PC39    PC40    PC41    PC42    PC43    PC44
Standard deviation     0.6225 0.6151 0.59972 0.57805 0.57474 0.57171 0.55860
Proportion of Variance 0.0043 0.0042 0.00399 0.00371 0.00366 0.00363 0.00346
Cumulative Proportion  0.8563 0.8605 0.86450 0.86821 0.87187 0.87550 0.87896
                          PC45    PC46    PC47    PC48    PC49   PC50    PC51
Standard deviation     0.54688 0.53869 0.53151 0.52598 0.51026 0.4936 0.49021
Proportion of Variance 0.00332 0.00322 0.00313 0.00307 0.00289 0.0027 0.00267
Cumulative Proportion  0.88228 0.88549 0.88863 0.89170 0.89458 0.8973 0.89995
                          PC52    PC53    PC54    PC55    PC56    PC57    PC58
Standard deviation     0.47660 0.46972 0.45966 0.45415 0.45177 0.44200 0.43255
Proportion of Variance 0.00252 0.00245 0.00234 0.00229 0.00226 0.00217 0.00208
Cumulative Proportion  0.90247 0.90492 0.90726 0.90955 0.91182 0.91398 0.91606
                          PC59    PC60   PC61    PC62    PC63    PC64    PC65
Standard deviation     0.42818 0.42049 0.4141 0.40813 0.40554 0.40185 0.39848
Proportion of Variance 0.00203 0.00196 0.0019 0.00185 0.00182 0.00179 0.00176
Cumulative Proportion  0.91809 0.92005 0.9220 0.92380 0.92563 0.92742 0.92918
                         PC66    PC67    PC68    PC69    PC70    PC71    PC72
Standard deviation     0.3914 0.38427 0.37629 0.37553 0.36578 0.36246 0.35829
Proportion of Variance 0.0017 0.00164 0.00157 0.00156 0.00148 0.00146 0.00142
Cumulative Proportion  0.9309 0.93252 0.93409 0.93565 0.93714 0.93859 0.94002
                         PC73    PC74    PC75    PC76    PC77    PC78    PC79
Standard deviation     0.3547 0.34802 0.34547 0.34009 0.33361 0.33034 0.32361
Proportion of Variance 0.0014 0.00134 0.00132 0.00128 0.00123 0.00121 0.00116
Cumulative Proportion  0.9414 0.94276 0.94408 0.94536 0.94660 0.94781 0.94897
                          PC80    PC81    PC82    PC83    PC84    PC85    PC86
Standard deviation     0.31983 0.31583 0.31248 0.30989 0.30383 0.30114 0.29579
Proportion of Variance 0.00113 0.00111 0.00108 0.00107 0.00102 0.00101 0.00097
Cumulative Proportion  0.95010 0.95121 0.95229 0.95336 0.95438 0.95539 0.95636
                          PC87    PC88    PC89    PC90    PC91    PC92    PC93
Standard deviation     0.29536 0.29124 0.28636 0.28608 0.28391 0.28096 0.27791
Proportion of Variance 0.00097 0.00094 0.00091 0.00091 0.00089 0.00088 0.00086
Cumulative Proportion  0.95733 0.95827 0.95918 0.96009 0.96098 0.96186 0.96271
                          PC94    PC95   PC96    PC97    PC98    PC99   PC100
Standard deviation     0.27364 0.27051 0.2679 0.26156 0.26002 0.25703 0.25466
Proportion of Variance 0.00083 0.00081 0.0008 0.00076 0.00075 0.00073 0.00072
Cumulative Proportion  0.96354 0.96435 0.9651 0.96591 0.96666 0.96739 0.96811
                        PC101   PC102   PC103   PC104   PC105   PC106   PC107
Standard deviation     0.2515 0.24831 0.24498 0.24344 0.24249 0.23823 0.23705
Proportion of Variance 0.0007 0.00068 0.00067 0.00066 0.00065 0.00063 0.00062
Cumulative Proportion  0.9688 0.96950 0.97016 0.97082 0.97147 0.97210 0.97272
                         PC108   PC109   PC110   PC111   PC112   PC113   PC114
```

```
Standard deviation     0.23416 0.23096 0.23025 0.22562 0.22385 0.22264 0.22033
Proportion of Variance 0.00061 0.00059 0.00059 0.00056 0.00056 0.00055 0.00054
Cumulative Proportion  0.97333 0.97392 0.97451 0.97508 0.97563 0.97618 0.97672
                         PC115   PC116   PC117  PC118   PC119   PC120   PC121
Standard deviation     0.21758 0.21686 0.21599 0.2115 0.21029 0.20830 0.20638
Proportion of Variance 0.00053 0.00052 0.00052 0.0005 0.00049 0.00048 0.00047
Cumulative Proportion  0.97725 0.97777 0.97829 0.9788 0.97927 0.97975 0.98023
                         PC122   PC123   PC124   PC125   PC126   PC127   PC128
Standard deviation     0.20357 0.20118 0.20011 0.19753 0.19597 0.19338 0.19273
Proportion of Variance 0.00046 0.00045 0.00044 0.00043 0.00043 0.00041 0.00041
Cumulative Proportion  0.98069 0.98114 0.98158 0.98201 0.98244 0.98285 0.98327
                         PC129  PC130   PC131   PC132   PC133   PC134   PC135
Standard deviation     0.19204 0.1903 0.18540 0.18348 0.18224 0.17902 0.17751
Proportion of Variance 0.00041 0.0004 0.00038 0.00037 0.00037 0.00036 0.00035
Cumulative Proportion  0.98367 0.9841 0.98446 0.98483 0.98520 0.98555 0.98590
```

There do not seem to be any dominant components in this case. We see that when using 16 out of 256 components, about 70% of the variance is accounted for; using 51 out of 256 components, about 90% of the variance is accounted for. It takes another 29 components to account for 95% of the variance. We have to determine what our tolerance is (i.e. what proportion of the variance are we wishing to account for?). Note that this produces a lot of output.

```
pc.digits[[3+1]]

$sdev
  [1] 3.379209e+00 2.816344e+00 2.659944e+00 2.567952e+00 2.261306e+00 1.972429e+00
  [7] 1.906591e+00 1.683494e+00 1.585108e+00 1.528654e+00 1.423500e+00 1.335809e+00
 [13] 1.305222e+00 1.240664e+00 1.172391e+00 1.090591e+00 1.070565e+00 1.006657e+00
 [19] 9.864085e-01 9.414192e-01 8.973307e-01 8.948743e-01 8.629366e-01 8.387566e-01
.
.
.
[229] 4.935517e-02 4.835835e-02 4.590472e-02 4.424380e-02 4.238942e-02 4.195997e-02
[235] 3.960521e-02 3.839665e-02 3.744329e-02 3.493125e-02 3.469602e-02 3.187392e-02
[241] 2.977113e-02 2.900486e-02 2.816116e-02 2.732529e-02 2.584797e-02 2.445102e-02
[247] 2.096828e-02 1.811004e-02 1.641156e-02 1.448598e-02 1.326148e-02 1.206861e-02
[253] 1.118102e-02 8.358757e-03 1.419367e-03 4.476719e-05

$rotation
                PC1           PC2           PC3           PC4           PC5
  [1,]  1.099612e-03 -2.819665e-04  7.083808e-04 -6.411722e-04  2.205328e-03
  [2,]  7.604992e-03 -1.695856e-03  2.534928e-03 -3.823239e-03  1.083701e-02
  [3,]  1.790017e-02 -2.676772e-03  1.286969e-02  6.994898e-03  2.765253e-02
  [4,]  2.299110e-02 -1.416062e-02  4.563322e-02  2.643396e-02  5.581772e-02
  [5,]  1.034658e-02 -2.710035e-02  7.873074e-02  5.901414e-02  7.673893e-02
  [6,] -5.753018e-03 -2.931406e-02  9.973891e-02  8.669514e-02  6.975713e-02
  [7,] -1.776670e-02 -2.315438e-02  1.035462e-01  1.071173e-01  4.110756e-02
  [8,] -1.759542e-02 -6.651151e-03  9.411738e-02  1.235880e-01  7.910551e-03
  [9,] -5.205410e-03 -5.212135e-03  7.991397e-02  1.219229e-01  4.933417e-03
 [10,]  2.101444e-02 -7.289502e-03  6.769933e-02  1.007456e-01  2.155735e-02
 [11,]  5.211035e-02 -1.133868e-02  5.741599e-02  6.751857e-02  5.483082e-02
 [12,]  4.344731e-02 -4.299763e-03  3.059531e-02  2.396153e-02  5.736097e-02
 [13,]  2.458593e-02  1.260481e-03  1.308472e-02 -4.973076e-04  3.145985e-02
 [14,]  5.425093e-03 -6.667111e-04  3.317569e-03 -3.907695e-03  9.428703e-03
 [15,]  3.360513e-04 -3.769016e-04  9.255188e-04 -1.014723e-04  6.460084e-04
 [16,]  1.308072e-06 -2.441618e-06  9.819137e-07  6.563653e-07 -1.445957e-06
 [17,]  6.610023e-03 -2.103299e-03  3.189284e-03 -3.323865e-03  1.209708e-02
 [18,]  2.211880e-02 -7.013420e-03  1.557243e-02 -1.093116e-02  3.541177e-02
 [19,]  4.971525e-02 -2.054937e-02  5.344012e-02  1.591546e-02  1.005163e-01
 [20,]  6.694901e-02 -6.641992e-02  1.026674e-01  5.569076e-02  1.585556e-01
 [21,]  4.391158e-02 -1.014963e-01  1.027717e-01  8.177299e-02  1.499565e-01
 [22,]  2.620040e-02 -9.880501e-02  7.411042e-02  7.926029e-02  7.626994e-02
 [23,]  2.015773e-02 -8.254495e-02  3.866674e-02  6.348251e-02  2.034777e-02
 [24,]  2.065485e-02 -7.698070e-02  1.547307e-02  5.430896e-02 -6.722872e-03
 [25,]  2.293019e-02 -6.868419e-02  1.094210e-02  5.219952e-02 -1.548793e-02
 [26,]  3.210304e-02 -4.605961e-02  2.857134e-02  6.772131e-02 -9.114666e-03
 [27,]  5.619332e-02 -3.161787e-02  5.490477e-02  9.915164e-02  2.923214e-02
 [28,]  9.646655e-02 -2.222523e-02  6.449774e-02  9.504049e-02  9.393785e-02
 [29,]  8.776529e-02 -4.057568e-03  3.916974e-02  3.550419e-02  9.051134e-02
 [30,]  4.756880e-02  3.595471e-03  1.725299e-02 -7.793875e-03  5.752924e-02
.
.
.
[250,]  9.107206e-02  4.892861e-02 -4.105314e-02  1.400713e-01 -1.583591e-01
[251,]  1.123120e-01  5.401507e-02 -4.764714e-02  8.534363e-02 -1.357300e-01
[252,]  9.436383e-02  3.232665e-02 -4.275065e-02  2.787946e-02 -7.425245e-02
[253,]  5.269707e-02  2.202144e-02 -2.660328e-02 -5.700130e-03 -2.265225e-02
[254,]  2.196619e-02  1.303770e-02 -1.066433e-02 -8.679210e-03 -4.212086e-03
[255,]  4.266127e-03  5.026219e-03 -3.766860e-03 -3.290478e-03 -8.020112e-04
[256,]  5.289045e-05  6.962248e-05 -9.195723e-05 -4.593701e-05  5.537029e-06
```

It may be difficult to understand what these linear combinations mean. So let us look at the first four principal components for each number.

```
pc <- array(dim = c(4 , 256, 10), dimnames = list(c(1:4),1:256,c(0:9)))
for (i in 0:9) {
   pc[1,,i+1] <- pc.digits[[i+1]]$rotation[,1]
   pc[2,,i+1] <- pc.digits[[i+1]]$rotation[,2]
   pc[3,,i+1] <- pc.digits[[i+1]]$rotation[,3]
   pc[4,,i+1] <- pc.digits[[i+1]]$rotation[,4]
}
```

We can take a look at the ***average*** of all the data along with the ***first four principal components*** (in this case, the principal component vectors are themselves ‘characters’ of a sort).

```
graphics.off()
x11(width = 4, height = 2.4)
layout(matrix(1:50, 5, 10, byrow = TRUE))
oldpar <- par(mar = c(0,0,0,0))
for (i in 0:9) {
   mean <- apply(d.digits[[i+1]], 2, mean)
   image(matrix(mean,16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
   image(matrix(pc[1,,i+1],16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
   image(matrix(pc[2,,i+1],16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
   image(matrix(pc[3,,i+1],16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
   image(matrix(pc[4,,i+1],16,16)[,16:1], xaxt = "n", yaxt = "n",
            col = gray((255:0)/255))
}
par(oldpar)
```

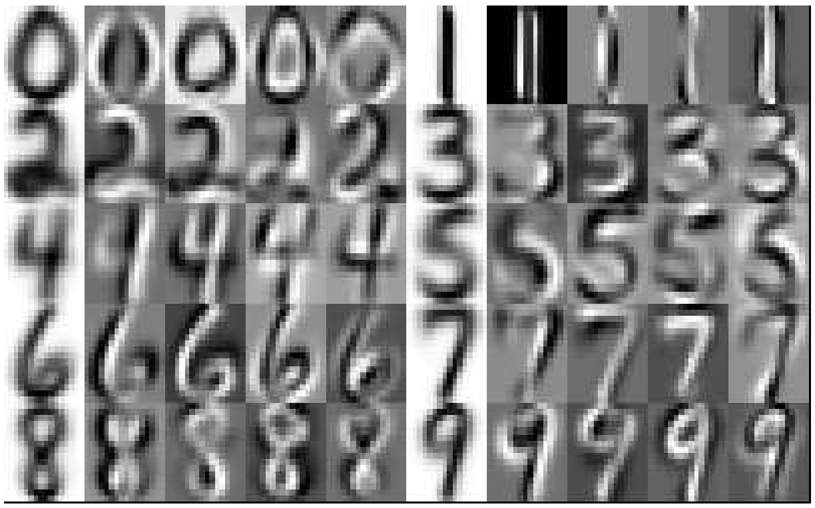

Figure 17. Mean and first 4 principal components

The first cell is the mean, the second is *PC*1, the third is *PC*2, the fourth is *PC*3 and the fifth is *PC*4.

We can look at what happens if we evaluate `mean+λ*PC1` ($-7 \leq \lambda \leq 7$).

Because we wish to do this several times, we will create a function to display the mean + pcs of one number.

```
display.mean.pc <- function(pc, digits) {
   mean <- apply(digits, 2, mean)
   for (i in 1:15) {
      image(matrix(mean+(i-8)*pc, 16,16)[,16:1], xaxt = "n",
             yaxt = "n", col = gray((255:0)/255))
   }
}
```

We will also use a function that will put all the numbers using one principal component in the same plot

```
display.pcs <- function (pcnum) {
   x11(width = 7, height = 5)
   oldpar <- par(mar = c(1,0,0,0))
   layout(matrix(1:150, 10, 15, byrow = TRUE))
   for (i in 0:9) {
      display.mean.pc(pc[pcnum,,i+1], d.digits[[i+1]])
   }
   bringToTop(which = dev.cur())
   par(oldpar)
}
display.pcs(1)
```

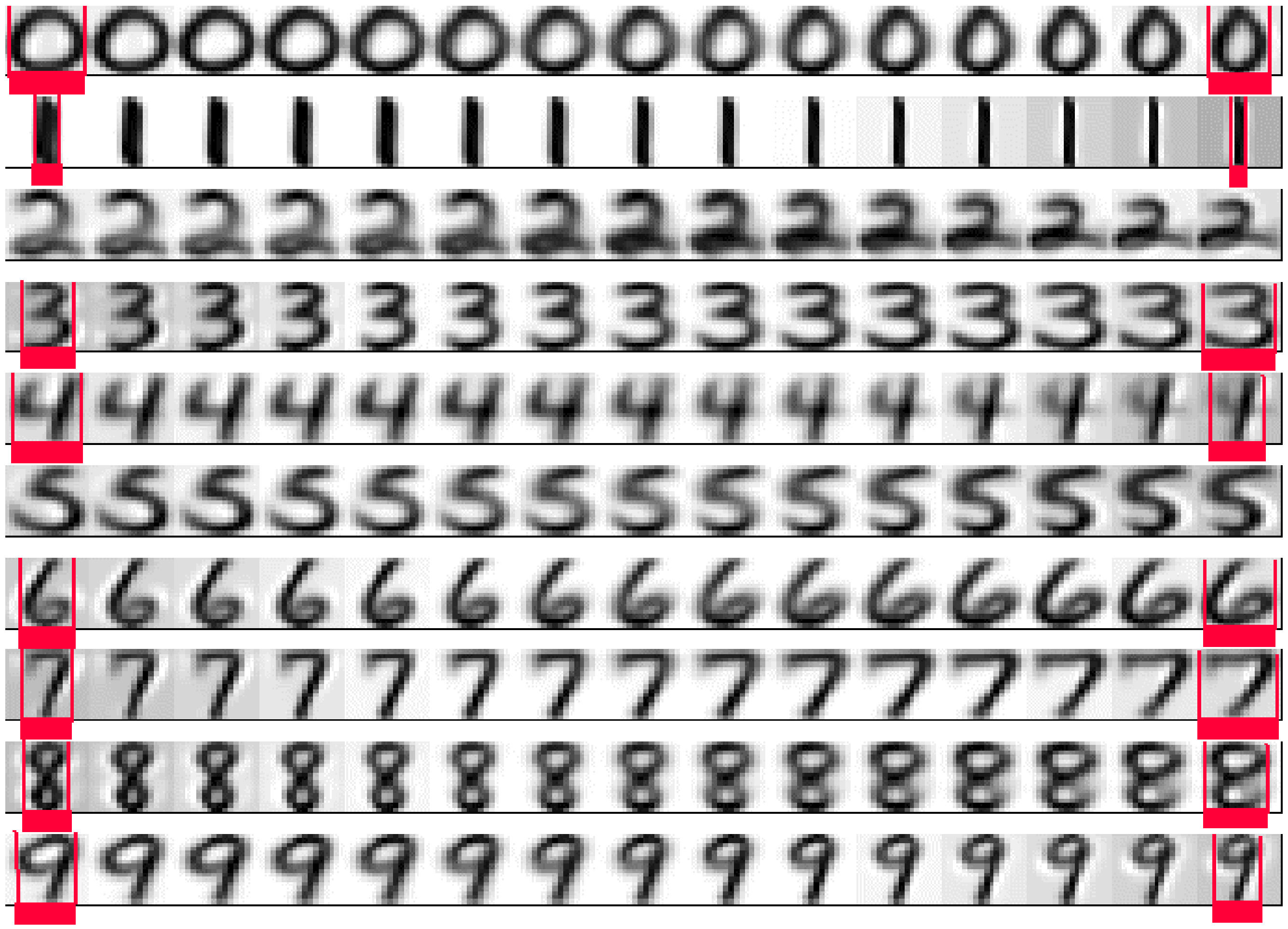

Figure 18.

It appears that *PC*1 is generally associated with the ***width of the character***. although for ‘2’ it appears to be the height and for ‘5’ the relative widths of the upper and lower halves.

And what happens if we evaluate `mean+`μ`*`$PC2$ ($-7 \le \mu \le 7$)?

```
display.pcs(2)
```

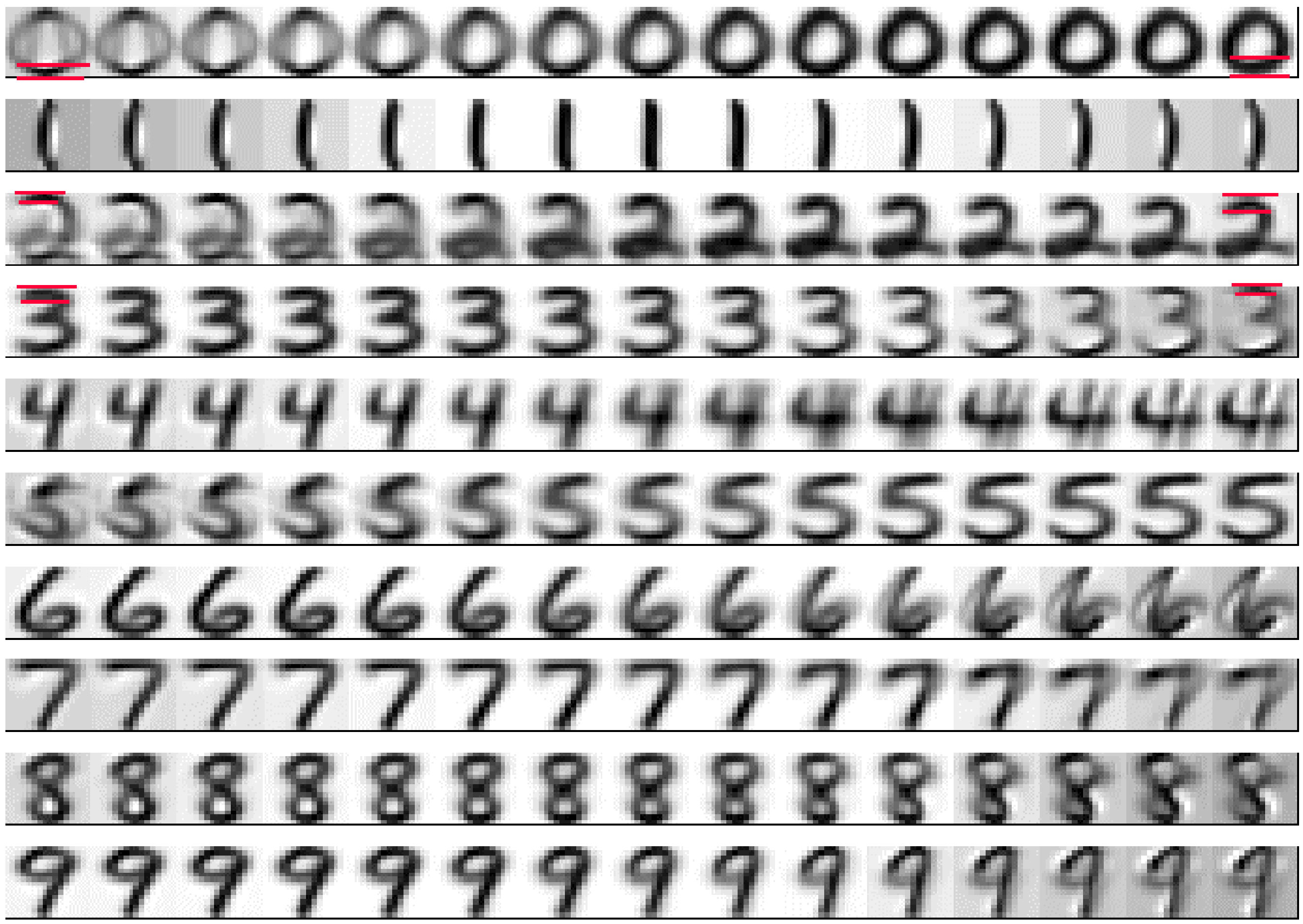

Figure 19.

In several cases, it appears that $PC2$ is associated with the ***thickness of the character*** although in ‘1’, it seems to be the direction of the curve; in ‘5’, the width; in ‘9’, the slope. For several of the digits, it appears to be associated with variability (‘2’, ‘4’, ‘5’, ‘6’).

Now what happens if we evaluate `mean`+μ`*`$PC3$ ($-7 \le \mu \le 7$)?

```
display.pcs(3)
```

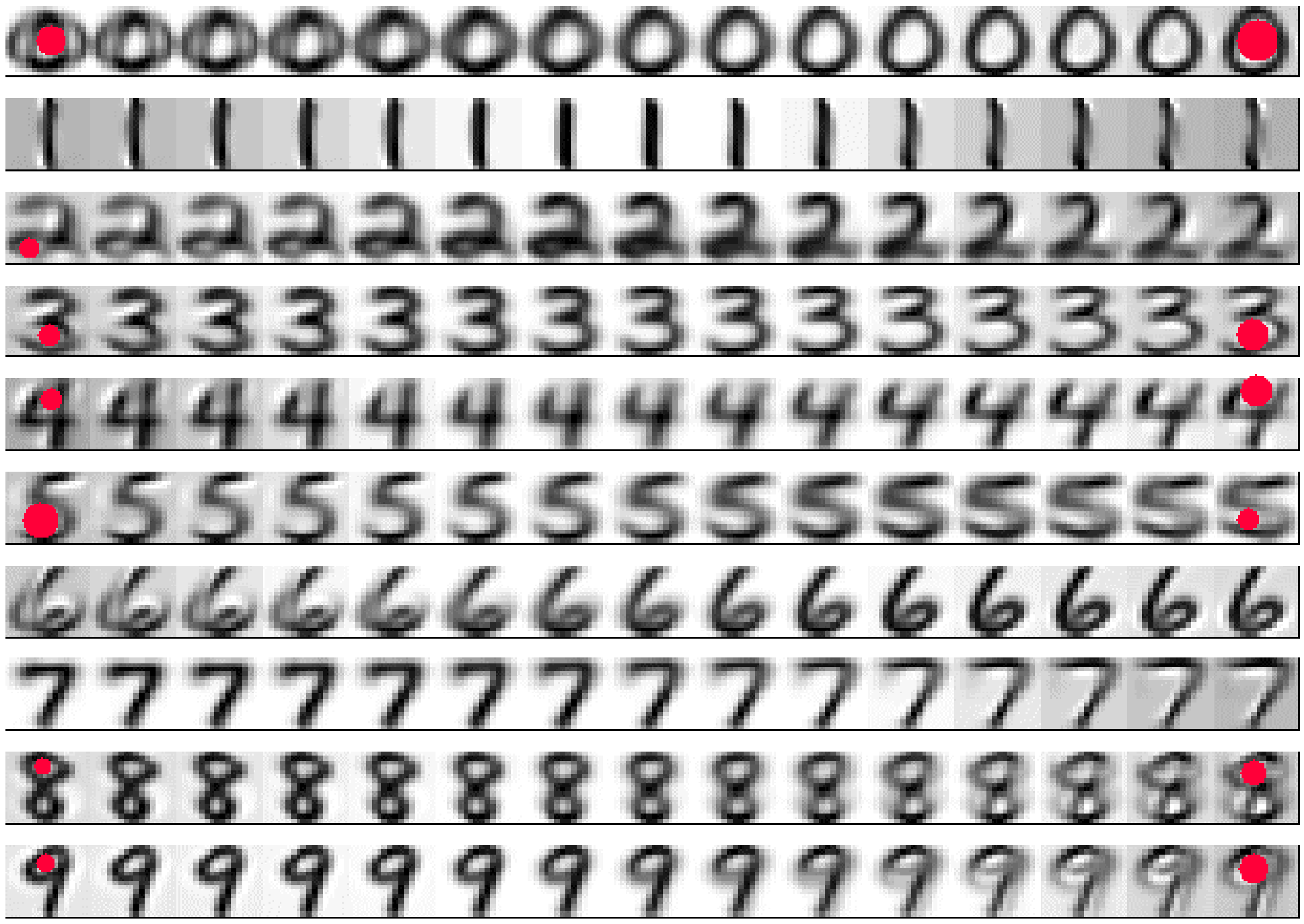

Figure 20.

It appears that $PC3$ is associated with the ***size of a 'loop' of the character***.

And what happens if we evaluate `mean+`μ`*`$PC4$ ($-7 \leq \mu \leq 7$)?

```
display.pcs(4)
```

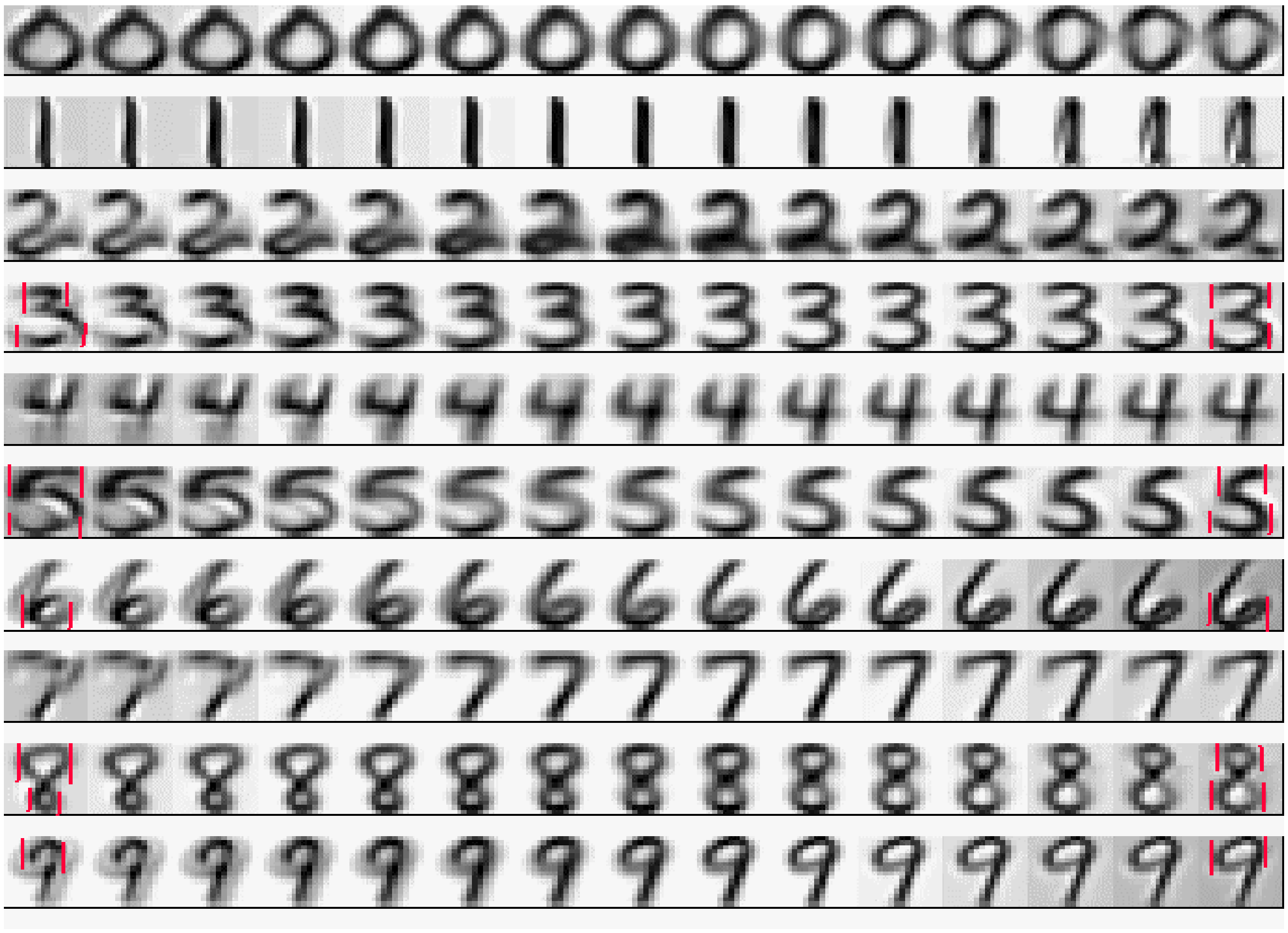

Figure 21.

It appears that for ‘3’, ‘5’, ‘8’ *PC*4 is associated with the ***the relative widths of the upper and lower halves***. For ‘6’ and ‘9’ it looks like total width but it might be relative width.

We can reconstruct our original data (as we noted earlier) using all the principal components, but instead of a full reconstruction, suppose we use a subset of the principal component vectors - for example the first 20. Our first step will be to represent all cases in terms of the new coordinate system (we did this earlier with `pc.1` etc.).

```
d.digits.pc <- {}
for (i in 0:9) {
   d.digits.pc[[i+1]] <- d.digits[[i+1]]%*%pc.digits[[i+1]]$rotation
}
```

For purposes of comparison, we plot the original first 144 images in our dataset (Figure 18).

```
plot.digits(d.digits)
```

Next we find the reconstruction of these images in terms of the first 20 principal components. We use a function to simplify the reconstruction.

We create a new array `tmp` to hold the data. The `cbind(d.digits.pc[[digit+1]][,j])` creates a *number of cases* × 1 **matrix** from the **vector** representing the j^{th} principal component of our data. This is necessary in order to be able to do the matrix multiplication.) Each component is rotated by multiplying by the 1 × 256 PC vector `pc.digits[[digit+1]]$rotation[,j]` to give a *number of cases* × 256 array, representing the full dataset in the original space. The result for each PC is added to the accumulated results for the previous PCs in `tmp`.

```
recreate <- function(pc.range, digit) {
tmp <- matrix(0, num.cases[digit+1], 256)
tmp <-
d.digits.pc[[digit+1]][,pc.range]%*%t(pc.digits[[digit+1]]$rotation[,pc.range])
tmp <- tmp/max(abs(range(tmp))) # We want to scale the data to lie in [-1, 1]
tmp
}
```

The following will use the previous function to reconstruct the images and plot the first 144 of the reconstructed images for the requested digit

- in original form;
- recreated from using only the first 20 principal components;
- recreated from using only the first 100 principal components;
- difference between a 100 PC and a 20 PC reconstruction;
- difference between a 256 PC and a 100 PC reconstruction

```
display.recreate <- function(digit) {
   plot.digits(d.digits[[digit+1]])
   pc.1.20 <- recreate(1:20, digit)
   plot.digits(pc.1.20)
   pc.1.100 <- recreate(1:100, digit)
   plot.digits(pc.1.100)
   pc.21.100 <- recreate(21:100, digit)
   plot.digits(pc.21.100)
   pc.101.256 <- recreate(101:256, digit)
   plot.digits(pc.101.256)
}
```

```
display.recreate(1)
```

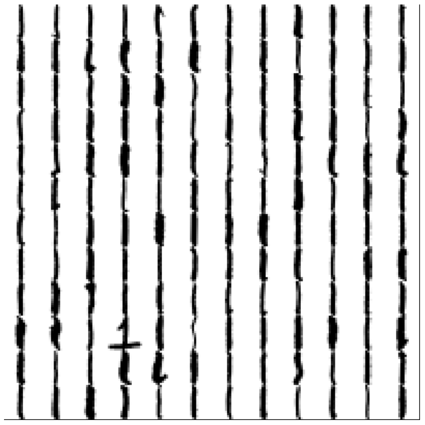

Figure 22. Original data for 1

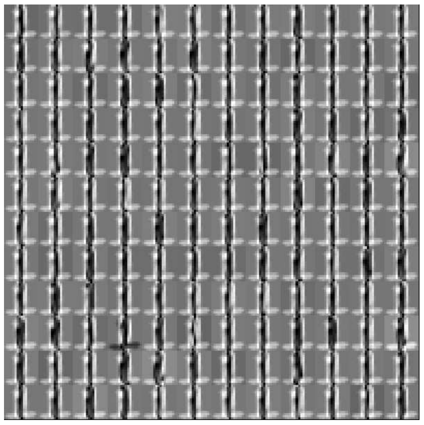

Figure 23. Reconstruction - 20 PC for 1

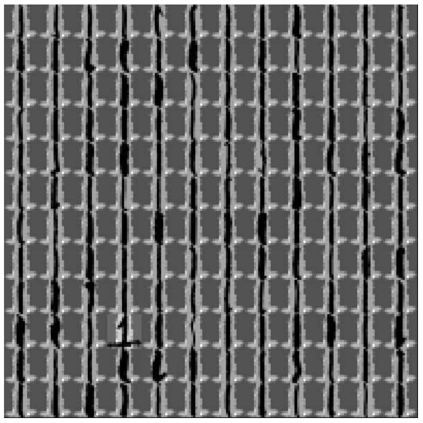

Figure 24. Reconstruction - 100 PC for 1

Because ‘1’ is such a simple digit, it is difficult to draw meaningful conclusions from these reconstructions.

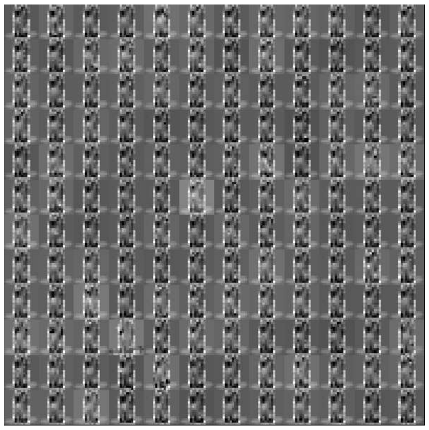

Figure 25. Difference between a 100 PC and a 20 PC reconstruction for ‘1’.

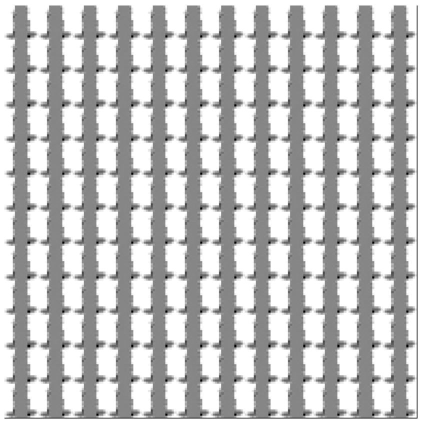

Figure 26. Difference between a 256 PC and a 100 PC reconstruction for ‘1’.

```
display.recreate(3)
```

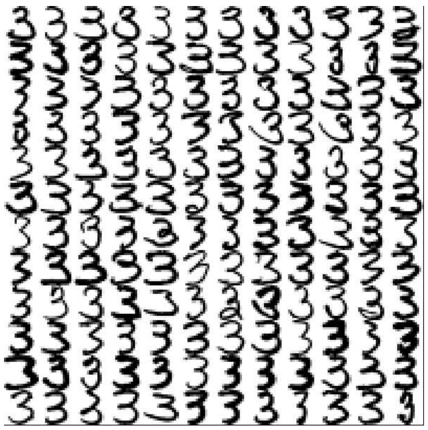

Figure 27. Original data for ‘3’

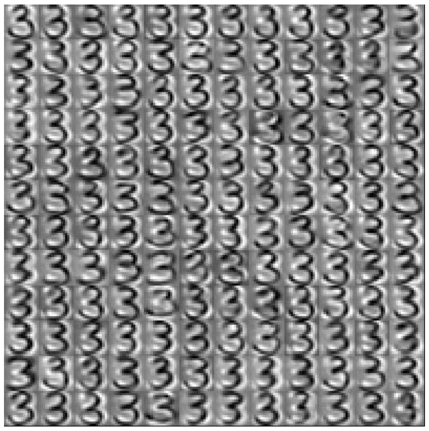

Figure 28. Reconstruction - 20 PC for ‘3’

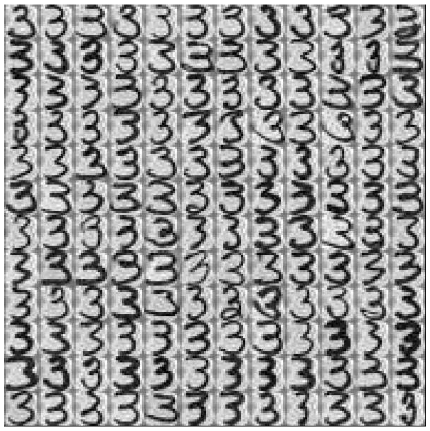

Figure 29.Reconstruction - 100 PC for ‘3’

We can see that much of the shape of the characters has been captured by using the first 20 principal components.

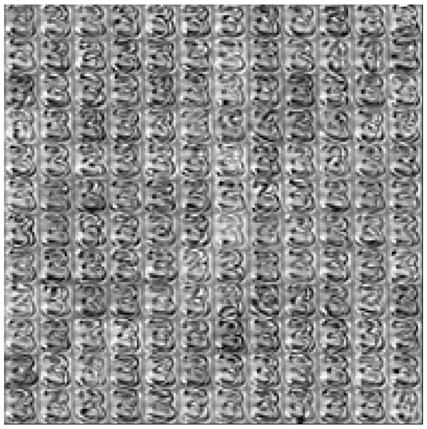

Figure 30. Difference between a 100 PC and a 20 PC reconstruction for ‘3’.

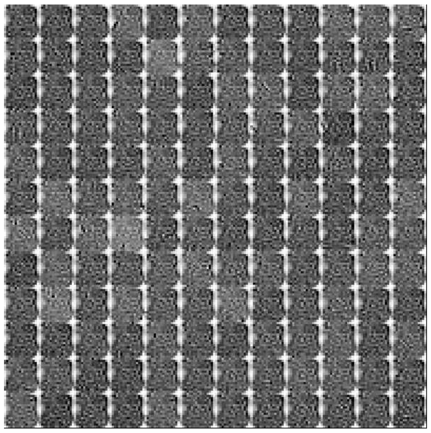

Figure 31. Difference between a 256 PC and a 100 PC reconstruction for ‘3’.

There seems to be little difference between the 100 PC reconstruction and the 256 PC (i.e. complete) reconstruction. Keep in mind that the images have been scaled to enhance the detail as much as possible.

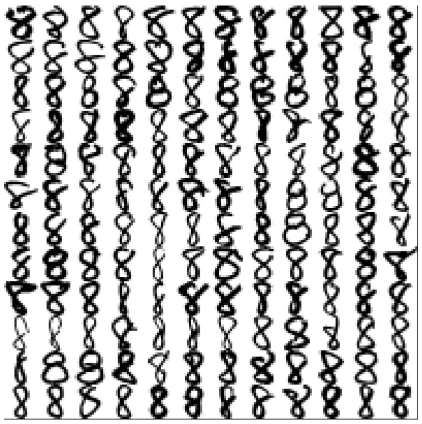

Figure 32. Original data for ‘8’

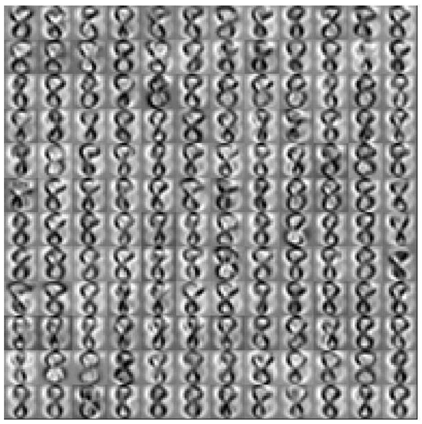

Figure 33. Reconstruction - 20 PC for ‘8’

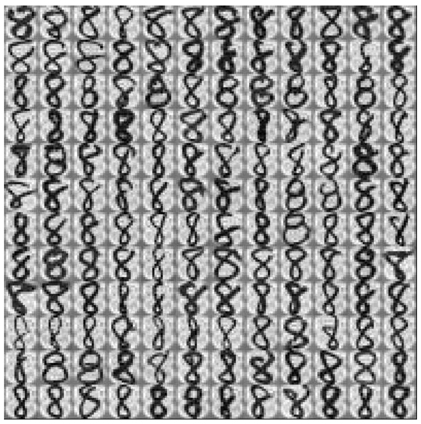

Figure 34. Reconstruction - 100 PC for ‘8’

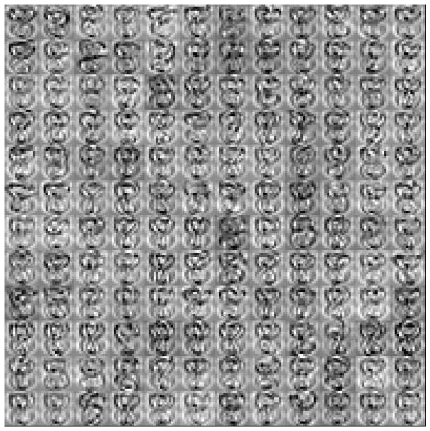

Figure 35. Difference between a 100 PC and a 20 PC reconstruction for ‘8’.

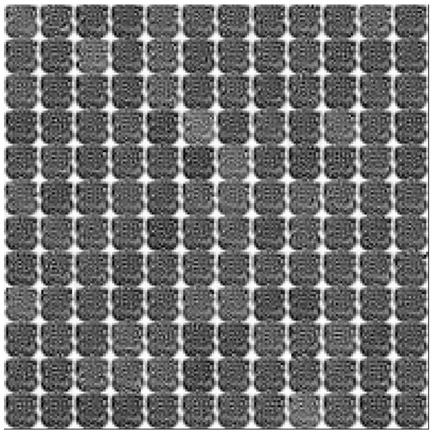

Figure 36. Difference between a 256 PC and a 100 PC reconstruction for ‘8’.

4.3 Multidimensional Scaling

Multidimensional scaling (*MDS*) maps data points in R^p to a lower-dimensional manifold. Consider observations $x_i, \ldots, x_n \; \varepsilon \; R^p$; let d_{ij} be the distance between observations i and j (e.g. Euclidean distance $d_{ij} = \|x_i - x_j\|$). Actually *MDS* needs only some **dissimilarity measure** d_{ij} *between* x_i and x_j; it does not need the x_i and x_j.
(Other methods such as self-organizing maps (*SOM*), which is related to neural networks, and principal curves and surfaces, which are an extension of principal components, need the actual data points.)

Kruskal-Shepard Scaling

We seek $z_1, \ldots, z_N \; \varepsilon \; R^k \; (k < p)$ to minimize *stress* function

$$S_D(z_1, \ldots, z_N) = \left[\sum_{i \neq j} (d_{ij} - \|z_i - z_j\|)^2 \right]^{1/2}.$$

This is ***least squares*** or ***Kruskal-Shepard*** scaling. We try to find a lower-dimensional approximation of the data that *preserves the pairwise distances* as much as possible. Note that the approximation is in terms of the distances rather than the squared distances. A gradient descent algorithm is used to minimize S_D.

Classical Scaling

In ***classical scaling***, (the `cmdscale` in `stats`) we use **similarities** s_{ij}

```
require(stats)
require(MASS)
```

To get an idea of the concept, consider what happens with *projection of a pyramid* versus *MDS on the pyramid*:

- with a projection, the apex would project to the centre and the base corners would remain fixed.
- with MDS, the apex would still project to the centre but the corners may move in order to *try to preserve the relationship of the slant distance to the apex to the base distances* (the higher the apex, the more the corners need to move).

```
(test <- matrix(c(1,1,0, 1,-1,0, -1,1,0, -1,-1,0, 0,0,1),ncol = 3,byrow = T))
     [,1] [,2] [,3]
[1,]    1    1    0
[2,]    1   -1    0
[3,]   -1    1    0
[4,]   -1   -1    0
```

```
[5,]    0    0    1
```

We now find the distances between all pairs of points (d_{ij}).

```
(test.dist <- dist(test))
         1        2        3        4
2 2.000000
3 2.000000 2.828427
4 2.828427 2.000000 2.000000
5 1.732051 1.732051 1.732051 1.732051
```

The function dist, which is in the stats library, produces a lower triangular matrix (with no diagonal elements) which gives the Euclidean distance between every case in the data set. We see that the distance between adjacent corners is 2, between opposite corners is 2.828427 ($= 2\sqrt{2}$), and from the apex to the corners is 1.732051 ($= \sqrt{3}$).

If we project this pyramid onto the plane, we get a square with a point at the centre.
If we use the classical scaling, we get

```
(test.mds <- cmdscale(test.dist))
              [,1]          [,2]
[1,]  0.000000e+00 -1.414214e+00
[2,] -1.414214e+00 -1.372600e-16
[3,]  1.414214e+00 -8.318541e-17
[4,]  1.523304e-16  1.414214e+00
[5,] -2.193883e-16  2.999852e-17
```

These are the *z* values.

The distances for the scaled points on the plane are

```
         1        2        3        4
2 2.000000
3 2.000000 2.828427
4 2.828427 2.000000 2.000000
5 1.414214 1.414214 1.414214 1.414214
```

For this pyramid, it turns out that (except for a rotation) MDS and projection produce the same result.

```
plot(test[,-3], xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), xlab = "", ylab = "",
        type = "n", main = "Classical vs Projection - apex at 1")
text(test[,1],test[,2],1:5, col = "red")
text(test.mds[,1],test.mds[,2],1:5, col = "blue")
```

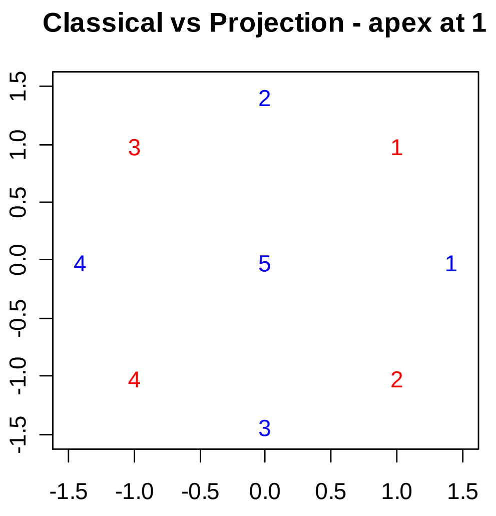

Figure 37. Blue = projected; red = classical

We will look at the stress later on.

This method uses the centered inner product

$$s_{ij} = (x_i - \bar{x}, x_j - \bar{x})$$

and then minimizes $\sum_{i \neq j} \{s_{ij} - (z_i - \bar{z}_i, z_j - \bar{z}_j)\}^2$ over $z_1, ..., z_N \; \varepsilon \; R^k$ (the scaled values). There is an explicit solution in terms of eigenvectors. **S** is the centered inner product matrix with elements $(x_i - \bar{x}, x_j - \bar{x})$.

To see what the classical MDS does, we take the 'distance' values and create a matrix of squared distances as follows

```
tmp <- matrix(0, 5, 5)
row(tmp) > col(tmp)          # to show what 'row(tmp) > col(tmp)' does
      [,1]  [,2]  [,3]  [,4]  [,5]
[1,] FALSE FALSE FALSE FALSE FALSE
[2,]  TRUE FALSE FALSE FALSE FALSE
[3,]  TRUE  TRUE FALSE FALSE FALSE
[4,]  TRUE  TRUE  TRUE FALSE FALSE
[5,]  TRUE  TRUE  TRUE  TRUE FALSE
```

We populate the lower triangle of the matrix with the squared distance

```
tmp[row(tmp) > col(tmp)] <- test.dist^2
tmp
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    4    0    0    0    0
[3,]    4    8    0    0    0
[4,]    8    4    4    0    0
[5,]    3    3    3    3    0
```

and then copy the lower triangle to the upper triangle.

```
(S <- tmp + t(tmp))
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    4    4    8    3
[2,]    4    0    8    4    3
[3,]    4    8    0    4    3
[4,]    8    4    4    0    3
[5,]    3    3    3    3    0
```

We have to do a double centering. We start by finding the row, column, and grand means,

```
(grand.mean <- mean(S))
[1] 3.52
(col.mean <- apply(S, 2, mean))
[1]  3.8 3.8 3.8 3.8 2.4
(row.mean <- apply(S, 1, mean))
[1]  3.8 3.8 3.8 3.8 2.4
```

Then we subtract the mean of each column from the columns (notice the use of `t(t(S)-col.mean)`to do the subtraction. The matrices are stored by columns so using `S-col.mean` would subtract col.mean[1] from S[1,1], col.mean[2] from S[2,1], col.mean[3] from S[3,1], and so on. The use of the `t(S)` means that we subtract col.mean[1] from S[1,1], col.mean[2] from S[1,2], col.mean[3] fromS[1,3], and so on (as we require). The `t(...)` gives us back the correct result.

```
(S <- t(t(S) - col.mean))
     [,1] [,2] [,3] [,4] [,5]
[1,] -3.8  0.2  0.2  4.2  0.6
[2,]  0.2 -3.8  4.2  0.2  0.6
[3,]  0.2  4.2 -3.8  0.2  0.6
[4,]  4.2  0.2  0.2 -3.8  0.6
[5,] -0.8 -0.8 -0.8 -0.8 -2.4
```

and then the mean of each row from the rows.

```
(S <- S - row.mean)
      [,1] [,2] [,3] [,4] [,5]
[1,] -7.6 -3.6 -3.6  0.4 -3.2
[2,] -3.6 -7.6  0.4 -3.6 -3.2
[3,] -3.6  0.4 -7.6 -3.6 -3.2
[4,]  0.4 -3.6 -3.6 -7.6 -3.2
[5,] -3.2 -3.2 -3.2 -3.2 -4.8
```

The process is completed by adding in the grand mean

```
(S <- S + grand.mean)
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] -4.08 -0.08 -0.08  3.92  0.32
[2,] -0.08 -4.08  3.92 -0.08  0.32
[3,] -0.08  3.92 -4.08 -0.08  0.32
[4,]  3.92 -0.08 -0.08 -4.08  0.32
[5,]  0.32  0.32  0.32  0.32 -1.28
```

We can check our result by taking the row and column means

```
apply(S,1,mean)
[1] -8.874737e-17 -8.876363e-17 -8.876363e-17 -8.874737e-17 8.881784e-17
apply(S,2,mean)
[1] -8.874737e-17 -8.876363e-17 -8.876363e-17 -8.874737e-17 8.881784e-17
```

The small values indicate that the matrix is double centred.

We find the k largest eigenvalues $\lambda_1 > ... > \lambda_k$ of **S**,

```
k <- 2                              # Dimension of the target space
eig <- eigen(-S/2, symmetric = T)
eig$values
[1] 4.000000e+00 4.000000e+00 8.000000e-01 -2.080358e-16 -1.434345e-15
```

with associated eigenvectors $\mathbf{E}_k = (e_1,...,e_k.)$.

We have

```
(E <- eig$vectors[,1:k])
              [,1]          [,2]
[1,]  6.762109e-01 -2.067338e-01
[2,] -2.067338e-01 -6.762109e-01
[3,]  2.067338e-01  6.762109e-01
[4,] -6.762109e-01  2.067338e-01
[5,] -1.301043e-18  4.580077e-17
```

Let $\mathbf{D}_k$ be a diagonal matrix with diagonal entries $\sqrt{\lambda_i}\,, i = 1,...,k.$

```
(D <- diag(sqrt(eig$values[1:k])))
     [,1] [,2]
[1,]    2    0
[2,]    0    2
```

The solutions z_i to the classical scaling problem are the rows of $\mathbf{E}_k\mathbf{D}_k$

```
(cmds <- E%*%D)
              [,1]          [,2]
[1,]  1.352422e+00 -4.134676e-01
[2,] -4.134676e-01 -1.352422e+00
[3,]  4.134676e-01  1.352422e+00
[4,] -1.352422e+00  4.134676e-01
[5,] -2.602085e-18  9.160153e-17
```

These are not the same values as we obtained from `cmdscale` (above) but, it is equivalent as can be seen from the fact that the distances are the same (it is very close to the projected values).

```
dist(cmds)
         1        2        3        4
2 2.000000
3 2.000000 2.828427
4 2.828427 2.000000 2.000000
5 1.414214 1.414214 1.414214 1.414214
```

```
plot(test[,-3], xlim = c(-1.5,1.5), ylim = c(-1.5,1.5), xlab = "", ylab = "",
     type = "n", main = "cmdscale vs eigen - apex at 1")
text(cmds[,1],cmds[,2],1:5, col = "red")
text(test.mds[,1],test.mds[,2],1:5, col = "blue")
```

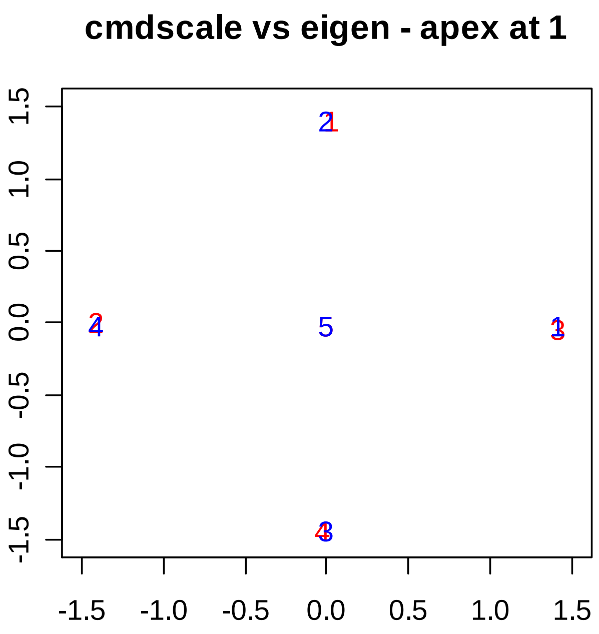

Figure 38. Blue = projected; red = classical

(Note that ***classical scaling is not equivalent to least squares scaling***, since inner products rely on a choice of origin while pairwise distances do not; a set of innner products determines a set of pairwise distances but not vice versa.)

We might wonder if the projection and classical MDS methods will produce the same result in general. To investigate this, move the apex to 2.236. There is a function that can be used to do this.

```
source(paste(code.dir, "ClassicMDS.r", sep = "/"))
Classic.MDS(2.236)
```

When we use the same process as before, we find that we do in fact get the same distance information and a similar image

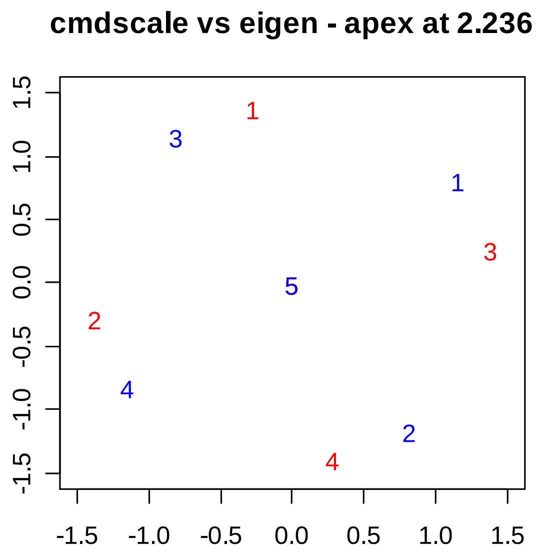

Figure 39. Blue = projected; red = classical

The fun begins if we try again with

```
Classic.MDS(2.237)
```

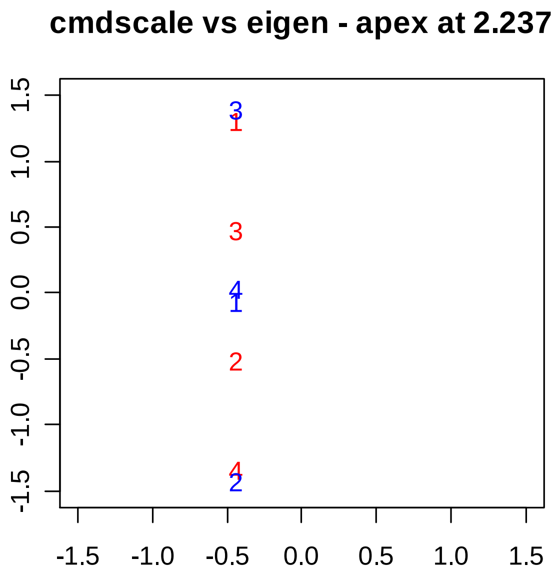

Figure 40. Blue = projected; red = classical

If we look at the distance as printed by the function

```
[1] "dist(cmds)"
          1         2         3         4
2 1.8201571
3 0.8288717 0.9912854
4 2.6490287 0.8288717 1.8201571
5 2.5997129 2.2912509 2.2912509 2.5997129
```

we no longer have the original distance values (as given above) of

```
         1        2        3        4
2 2.000000
3 2.000000 2.828427
4 2.828427 2.000000 2.000000
5 1.414214 1.414214 1.414214 1.414214
```

but we also find that the distance values for the eigenvalue method are different, as can be seen from the figure. **WHY**?

There are two factors at work here:

- We believe that we should project from three dimensions to two dimensions (otherwise we have no dimension reduction), but the number of dimensions required is determined by the *k* largest eigenvalues. We saw in the original case that the eigenvalues were [4, 4, 0.8, 0, 0] so that the first two dimensions are dominant. What we did not look at in the 2.236 and 2.237 cases were the eigenvalues which were [4, 4, 3.999757, 0, 0] and [4.003335, 4, 4, 0, 0] (look at the printed output). We see that the 2.236 case was on the borderline of having 2 eigenvalues that are larger than the others, while the 2.237 case suggests that two dimensions are no longer adequate.

- The `cmdscale` method and the *eigenvalue* method should produce the same results but are different. A closer look at the process in both cases indicates that the matrices for which the eigenvectors are found differ only by amount in the order of 10^{-16} and yet they produce somewhat different results. ***We need to watch out for numerical instabilities***.

Sammon Mapping

An alternative to the ***Kruskal-Shepard*** scaling and the ***classical scaling*** is the ***Sammon mapping*** which minimizes

$$\sum_{i \neq j} \frac{(d_{ij} - \|z_i - z_j\|)^2}{d_{ij}}$$

and places more emphasis on **preserving smaller pairwise distances**.

We will apply this method to the vertex at 2.237 case.

```
test <- matrix(c(1,1,0, 1,-1,0, -1,1,0, -1,-1,0, 0,0,2.237), ncol = 3, byrow = T)
test.dist <- dist(test)
(test.mds <- sammon(test.dist))
Initial stress        : 0.11184
stress after  10 iters: 0.06046, magic = 0.500
stress after  20 iters: 0.06037, magic = 0.500
$points
           [,1]          [,2]
[1,] -1.0857805 -8.263581e-01
[2,]  0.1748359 -1.412080e+00
[3,]  0.1748359  1.412080e+00
[4,] -1.0857805  8.263581e-01
[5,]  1.8218892 -2.586420e-13
$stress
[1] 0.06036759
$call
sammon(d = test.dist)
plot(test[,-3], xlim = c(-1.5,1.8), ylim = c(-1.5,1.5), xlab = "", ylab = "",
          type = "n", main = "Projected vs Sammon - apex at 2.237")
text(test[,1],test[,2],1:5, col = "red")
text(test.mds$points[,1],test.mds$points[,2],1:5, col = "blue")
li <- c(1,2,5,4,3,5,1,3,4,2)
lines(test[li,1],test[li,2], col = "red", lwd = 2)
lines(test.mds$points[li,1],test.mds$points[li,2],col = "blue", lwd = 2)
```

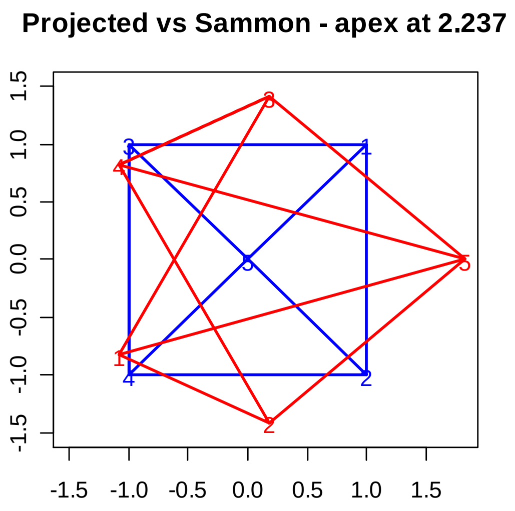

Figure 41. Blue = projected; red = Sammon

The lines indicate the projected pyramid.

```
(dist.sam <- dist(test.mds$points))
         1        2        3        4
2 1.390045
3 2.569000 2.824160
4 1.652716 2.569000 1.390045
5 3.022815 2.169506 2.169506 3.022815
```

To illustrate what is taking place we note that the `sammon` routine in `MASS` uses the `cmdscale` from `stats` as a starting configuration.

```
(test.mds <- cmdscale(test.dist))
        [,1]          [,2]
[1,] -0.4474 -4.050837e-01
[2,] -0.4474 -1.354957e+00
[3,] -0.4474  1.354957e+00
[4,] -0.4474  4.050837e-01
[5,]  1.7896 -7.862374e-14
```

To find the initial stress, we use

```
(total <- sum(test.dist))
[1] 24.24301
```

Compute $d_{ij} - \|z_i - z_j\|$.

```
(diff <- test.dist - dist(cmdscale(test.dist)))
           1          2          3          4
2 1.05012715
3 0.23995978 0.11851406
4 2.01825975 0.23995978 1.05012715
5 0.37315792 0.03118513 0.03118513 0.37315792
```

Next compute

$$\sum_{i \neq j} \frac{(d_{ij} - \|z_i - z_j\|)^2}{d_{ij}}$$

```
(err <- sum((diff^2)/test.dist))
[1] 2.711433
```

The stress function that is used in the Sammon routine is a **scaled stress**,

```
err/total
[1] 0.1118439
```

The routine then uses a gradient descent to minimize the stress.

The final stress is found by

```
(diffs <- test.dist-dist.sam)
             1            2            3            4
2  0.609955357
3 -0.568999721  0.004266701
4  1.175711004 -0.568999721  0.609955357
5 -0.376275909  0.477033374  0.477033374 -0.376275909
(errs <- sum((diffs^2)/test.dist))
[1] 1.463492
errs/total            #stress
[1] 0.06036759
```

The effect of this mapping is to project the 5 points of the 3-dimensional pyramid onto the plane in such a way that the relative differences between the true and scaled distances are as small as possible. Note that the classical routine produced a set of points in the plane that

made the differences between those points in the plane and the scaled points the same, but did not minimize the off-plane distances.

Least squares and classical scaling are *metric* scaling methods (the actual dissimilarities or similarities are approximated.)

Shepard-Kruskal nonmetric scaling effectively uses only *ranks*. Nonmetric scaling minimizes the stress function

$$\sum_{i,j} \frac{[\theta(\|z_i - z_j\|) - d_{ij}]^2}{\sum_{i,j} d_{ij}^2}$$

over the d_{ij} and an arbitrary increasing function $\theta(\bullet)$. Fixing $\theta(\bullet)$, we use gradient descent to minimize over d_{ij}. Fixing d_{ij}, we use **isotonic regression** to find the best monotonic approximation $\theta(\bullet)$. We iterate these steps until the solutions seem to stabilize.

Note: In **principal surfaces** and **SOM**, points close together in our original space should map close together in the manifold, but points far apart in the original space *might also map close together*. This is less likely in *MDS* since it explicitly tries to preserve all pairwise distances.

Examples:

Consider a situation in which you do not (or can not) know the data but do know the dissimilarities.

For example, we might have the following table of distances between European cities as found in the dataset `eurodist`.

```
data(eurodist)
eurodist
                Athens Barcelona Brussels Calais Cherbourg Cologne Copenhagen
Barcelona         3313
Brussels          2963      1318
Calais            3175      1326      204
Cherbourg         3339      1294      583    460
Cologne           2762      1498      206    409       785
Copenhagen        3276      2218      966   1136      1545     760
Geneva            2610       803      677    747       853    1662       1418
Gibralta          4485      1172     2256   2224      2047    2436       3196
Hamburg           2977      2018      597    714      1115     460        460
Hook of Holland   3030      1490      172    330       731     269        269
Lisbon            4532      1305     2084   2052      1827    2290       2971
Lyons             2753       645      690    739       789     714       1458
Madrid            3949       636     1558   1550      1347    1764       2498
Marseilles        2865       521     1011   1059      1101    1035       1778
Milan             2282      1014      925   1077      1209     911       1537
Munich            2179      1365      747    977      1160     583       1104
Paris             3000      1033      285    280       340     465       1176
Rome               817      1460     1511   1662      1794    1497       2050
Stockholm         3927      2868     1616   1786      2196    1403        650
Vienna            1991      1802     1175   1381      1588     937       1455

                Geneva Gibralta Hamburg Hook of Holland Lisbon Lyons Madrid
Barcelona
Brussels
Calais
Cherbourg
Cologne
Copenhagen
Geneva
Gibralta          1975
Hamburg           1118     2897
Hook of Holland    895     2428     550
Lisbon            1936      676    2671            2280
Lyons              158     1817    1159             863   1178
Madrid            1439      698    2198            1730    668  1281
Marseilles         425     1693    1479            1183   1762   320   1157
Milan              328     2185    1238            1098   2250   328   1724
Munich             591     2565     805             851   2507   724   2010
Paris              513     1971     877             457   1799   471   1273
Rome               995     2631    1751            1683   2700  1048   2097
Stockholm         2068     3886     949            1500   3231  2108   3188
Vienna            1019     2974    1155            1205   2937  1157   2409

                Marseilles Milan Munich Paris Rome Stockholm
Barcelona
Brussels
Calais
Cherbourg
Cologne
Copenhagen
Geneva
Gibralta
```

```
Hamburg
Hook of Holland
Lisbon
Lyons
Madrid
Marseilles
Milan                    618
Munich                  1109   331
Paris                    792   856    821
Rome                    1011   586    946  1476
Stockholm               2428  2187   1754  1827 2707
Vienna                  1363   898    428  1249 1209      2105
```

We will use multidimensional scaling on this data. In order to plot the results on a map of Europe, we will need to do some scaling of the results to make them fit on the map.
The following gives us a way of plotting images (in this case Portable GreyMap). We will use the classical, iso, and Sammon mappings.

```
library(pixmap)
d.file <- paste(data.dir, "Europe.pgm", sep = "/")
```

For Figure 42:

```
image <- read.pnm(d.file)
plot(image, main = "Classical MDS")
loc.cmd <- cmdscale(eurodist)
x <- (loc.cmd[,1]-min(loc.cmd[,1]))*.14 + 50
y <- -(loc.cmd[,2]-max(loc.cmd[,2]))*.12 + 80
text(x, y, labels(eurodist), cex = 1, col = "red")
```

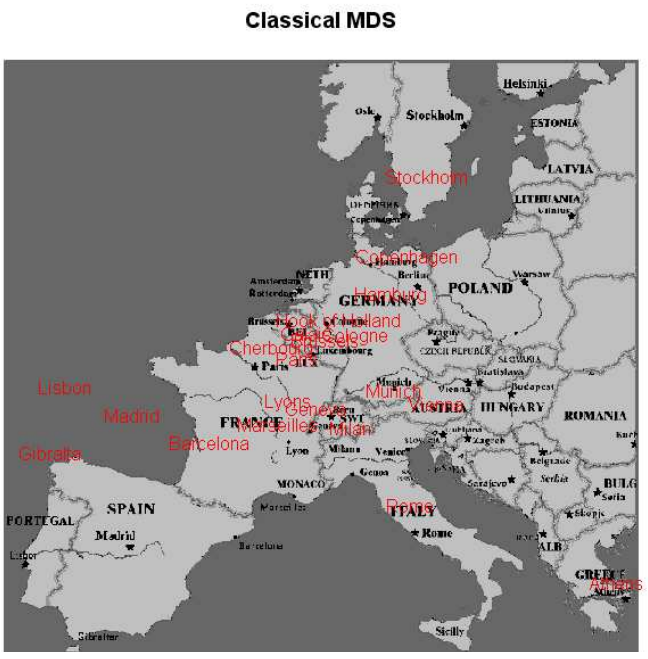

Figure 42.

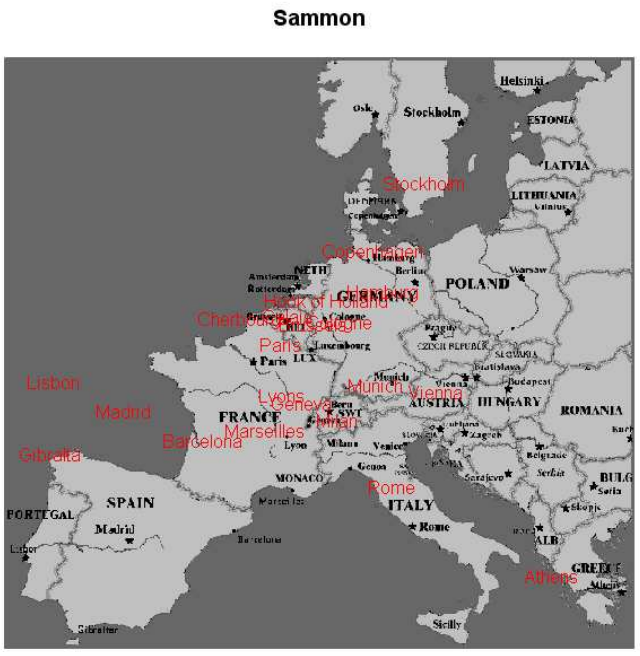

Figure 43.

For Figure 43:

```
plot(image, main = "Sammon")
loc <- sammon(eurodist)
locPt <- loc$points
x <- (loc$points[,1]-min(loc$points[,1]))*.14 + 50
y <- -(loc$points[,2]-max(loc$points[,2]))*.12 + 80
text(x, y, labels(eurodist), cex = 1, col = "red")
```

For Figure 44:

```
plot(image, main = "isoMDS")
loc <- isoMDS(eurodist)
locPt <- loc$points
x <- (locPt[,1]-min(locPt[,1]))*.14 + 50
y <- -(locPt[,2]-max(locPt[,2]))*.15 + 80
text(x, y, labels(eurodist), cex = 1, col = "red")
```

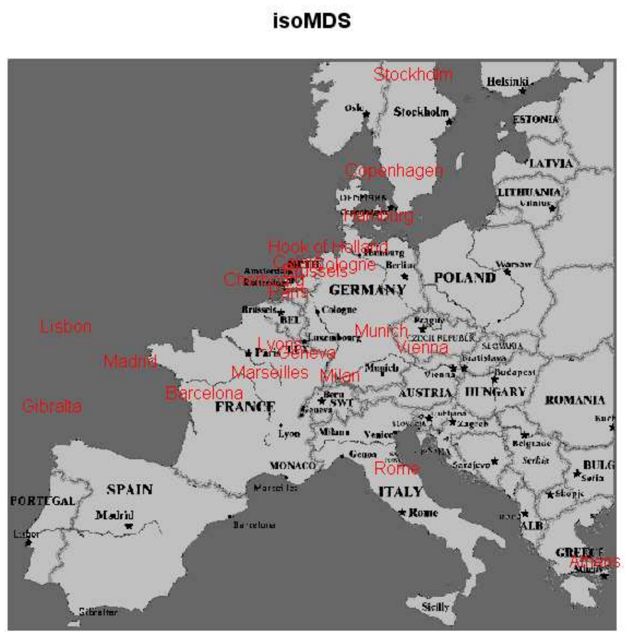

Figure 44.

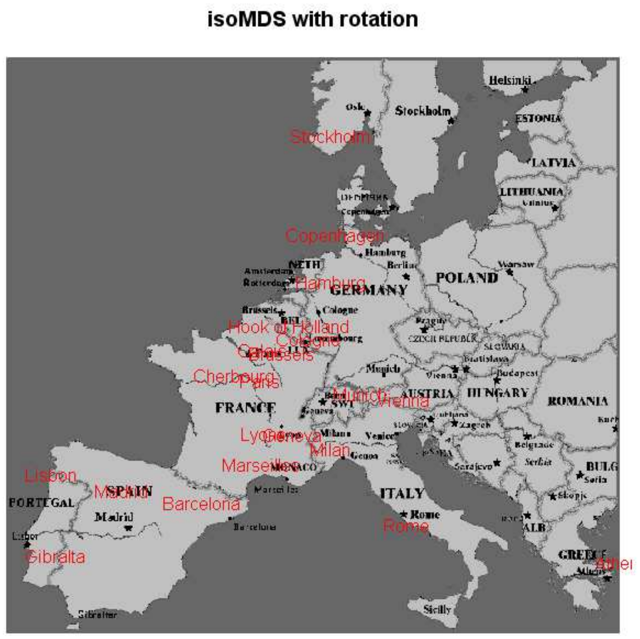

Figure 45.

As can be seen in Figure 44, the information on the distances between points (cities) allows us to place the cities reasonably well. Keep in mind that we are only obtaining relative locations and that the use of rotation might improve the "map".

For Figure 45:

```
source(paste(code.dir, "3Drotations.r", sep = "/"))
plot(image, main = "isoMDS with rotation")
loc <- isoMDS(eurodist)
locPt <- R.2D(loc$points,-.25)
x <- (locPt[,1]-min(locPt[,1]))*.14 + 50
y <- -(locPt[,2]-max(locPt[,2]))*.15 + 80
text(x, y, names(eurodist), cex = 1, col = "red")
```

Soft drink taste test example

Consider a taste test in which 10 students did a taste test on 10 soft drinks Diet Pepsi, RC Cola, Yukon, Dr. Pepper, Shasta, Coca-Cola, Diet Dr. Pepper, Tab, Pepsi-Cola, Diet-Rite. The similarity matrix represents the perception of the students as to the similarity of the tastes.

```
comp <- c(”Diet Pepsi”,”RC Cola”, ”Yukon”,”Dr. Pepper”,”Shasta”, ”Coca-Cola”,
          ”Diet Dr. Pepper”,”Tab”,”Pepsi-Cola”,”Diet-Rite”)
p <- length(comp)
dat <- scan(paste(data.dir,”softdrinks.dat”,sep = ”/”))
par(mfrow = c(2,3), mar = c(1,0,1,0), xaxt = ”n”, yaxt = ”n”)
k <- 1
      # Repeat 9 times to get 9 students
for (kk in 1:9) {
   Dis <- matrix(0, p, p)
   Dis[col(Dis) >= row(Dis)] <- dat[k:(k+54)]
   Dis <- t(Dis)+Dis
   diag(Dis) <- diag(Dis/2)
   k <- k + 55
   dimnames(Dis) <- list(comp, comp)
   coords <- cmdscale(Dis)
   coord1 <- -coords[,1]
   coord2 <- -coords[,2]
   plot(coord2,coord1, xlab = ””,main = ”cmdscale”)
   text(coord2,coord1, comp)
   coords <- isoMDS(Dis)
   coord1 <- -coords$points[,1]
   coord2 <- -coords$points[,2]
   plot(coord2,coord1, xlab = ””, main = paste(”isoMDS ”, kk))
   text(coord2,coord1, comp)
   coords <- sammon(Dis)
   coord1 <- -coords$points[,1]
   coord2 <- -coords$points[,2]
   plot(coord2,coord1, xlab = ””, main = ”sammon”)
   text(coord2,coord1, comp)
   readline(”Press any key”)
}
par(oldpar)
```

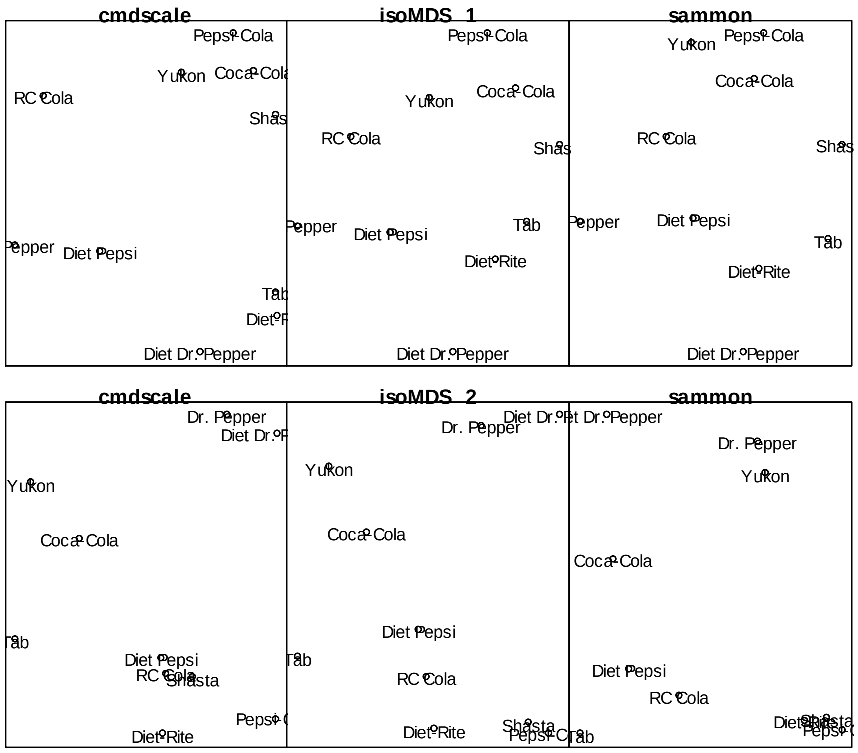

Figure 46.

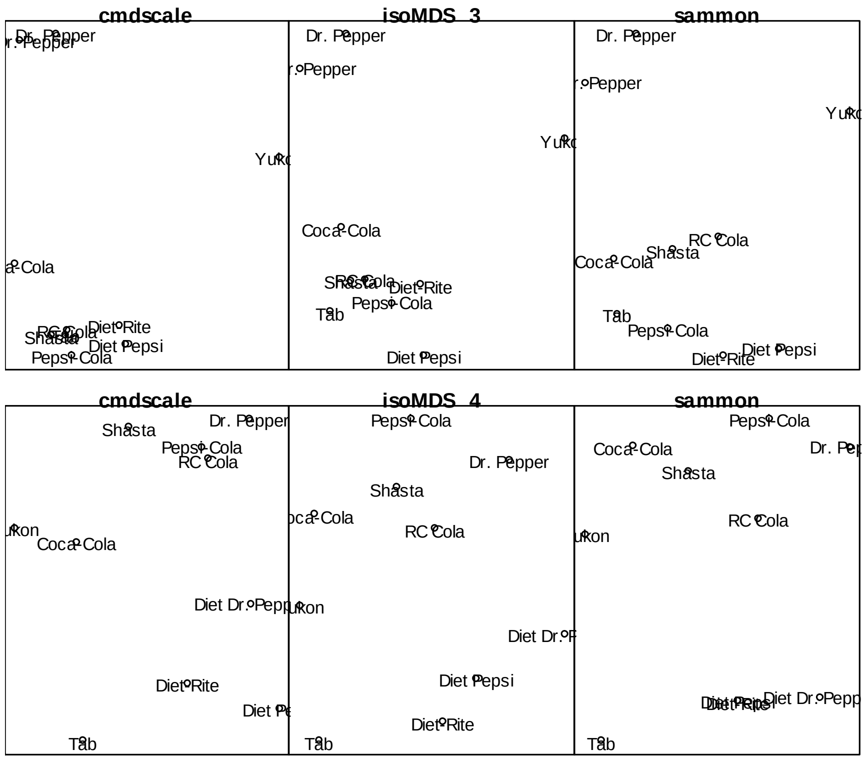

Figure 47.

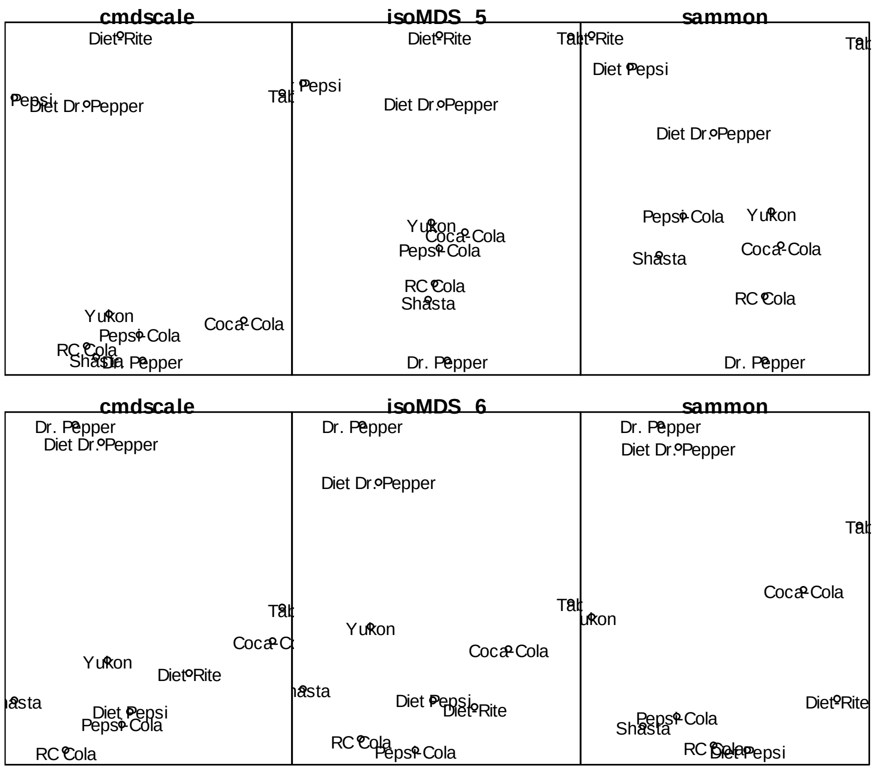

Figure 48.

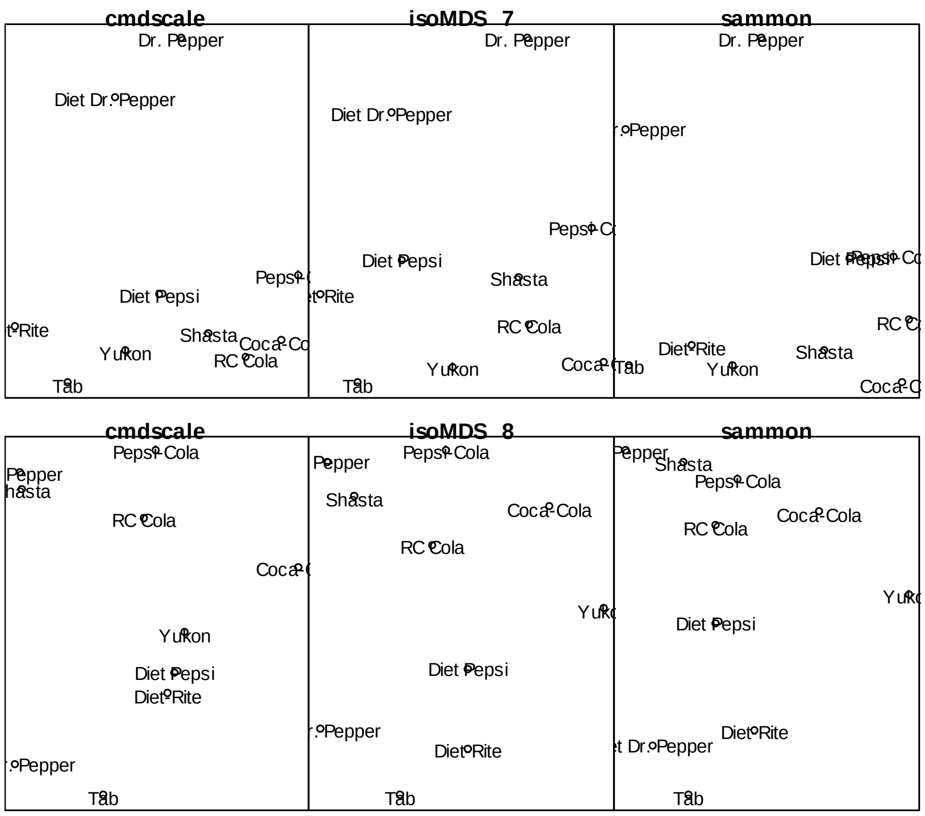

Figure 49.

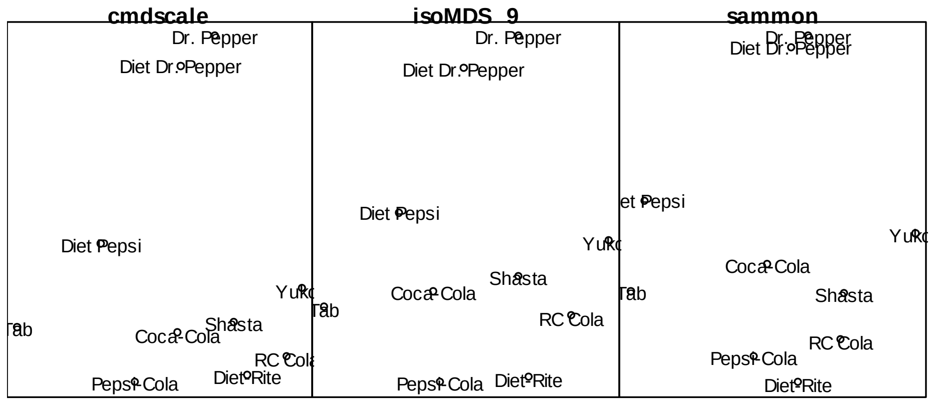
cmdscale
isoMDS_9
sammon
Dr. Pepper
Diet Dr. Pepper
Diet Pepsi
Yukon
Tab
Coca-Cola
Shasta
RC Cola
Pepsi-Cola
Diet-Rite

Figure 50.

Flea Beetles

As a different type of example, we can look at the flea beetle data (again!).
In this case we have high dimensional (6) data that has clustering.

```
source(paste(code.dir, "ReadFleas.r", sep = "/"))
(flea.dist <- dist(d.flea))
           1         2        3        4        5         6         7         8
2   8.246211
3  12.609520 16.031220
4  20.856654 16.583124 29.35984
5  24.535688 22.135944 36.18011 16.03122
6  35.930488 31.288976 44.85532 16.73320 18.62794
7   7.549834  9.539392 14.89966 18.54724 25.27845 34.813790
8  10.908712  8.306624 18.05547 18.65476 21.61018 30.724583 15.684387
9  21.702534 16.278821 30.13304 19.64688 19.20937 26.381812 24.124676 13.638182
10 34.942810 31.352831 43.95452 16.18641 17.97220  6.164414 33.555923 31.112698
11 16.000000 13.490738 12.44990 23.89561 34.46738 39.433488 13.820275 18.411953
12 17.291616 11.958261 27.60435 10.86278 13.89244 22.583180 16.492423 14.966630
13 12.206556 17.291616 11.48913 31.43247 33.00000 45.055521 18.654758 15.874508
14  5.000000  9.433981 15.29706 22.71563 24.63737 37.496667  8.944272 13.190906
15  9.848858  5.196152 19.79899 12.96148 17.52142 27.018512 11.224972  7.615773
16 17.435596 14.560220 19.77372 13.96424 26.83282 28.053520 15.459625 16.401219
17 22.022716 16.941074 27.34959 13.49074 21.97726 21.118712 23.108440 14.000000
...
flea.mds <- isoMDS(flea.dist)
initial  value 4.053595
final  value 3.626536
converged
```

The `plot` command below sets up the plot but the `type = "n"` prevents any data from being displayed.

```
plot(flea.mds$points, type = "n", main = "isoMDS for Flea Beetles")
text(flea.mds$points, labels = as.character(1:nrow(d.flea)),col = species+1)
```

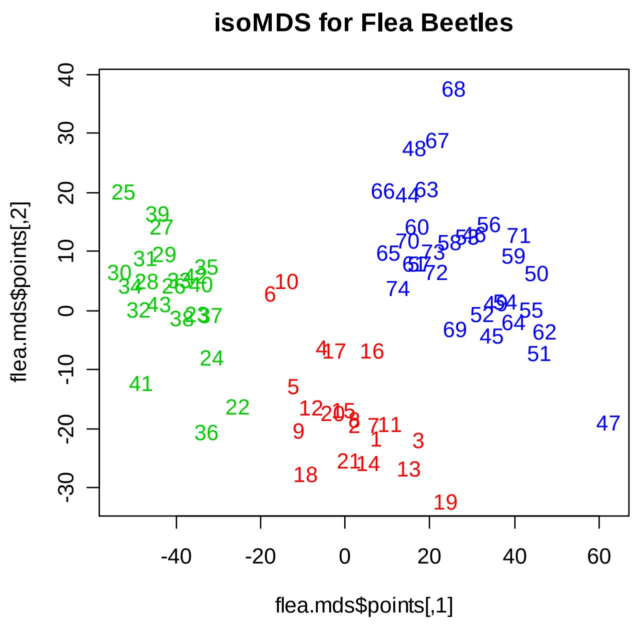

Figure 51.

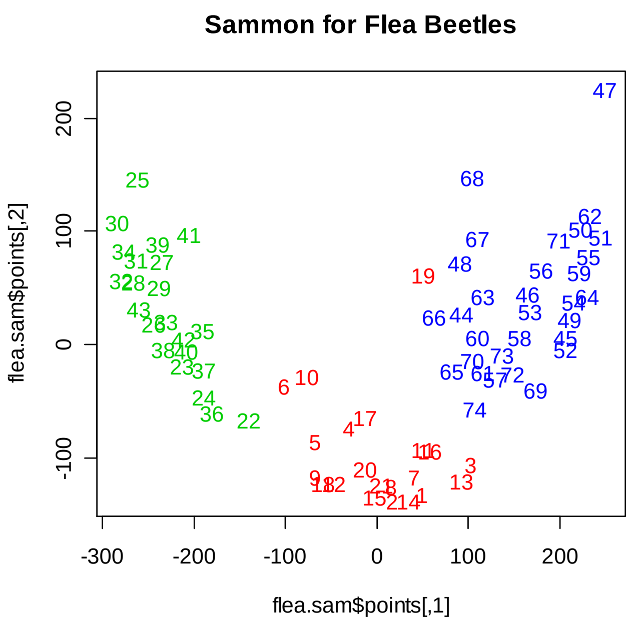

Figure 52.

```
flea.sam <- sammon(dist(flea.dist))
Initial stress : 0.02439
stress after 9 iters: 0.01203
plot(flea.sam$points, type = "n", main = "Sammon for Flea Beetles")
text(flea.sam$points, labels = as.character(1:nrow(d.flea)),col = species+1)
```

For comparison, we can look at the projection on the plane that produces one of the best separations of the species.

```
plot(-d.flea[,c(1,6)], col = species+1)
```

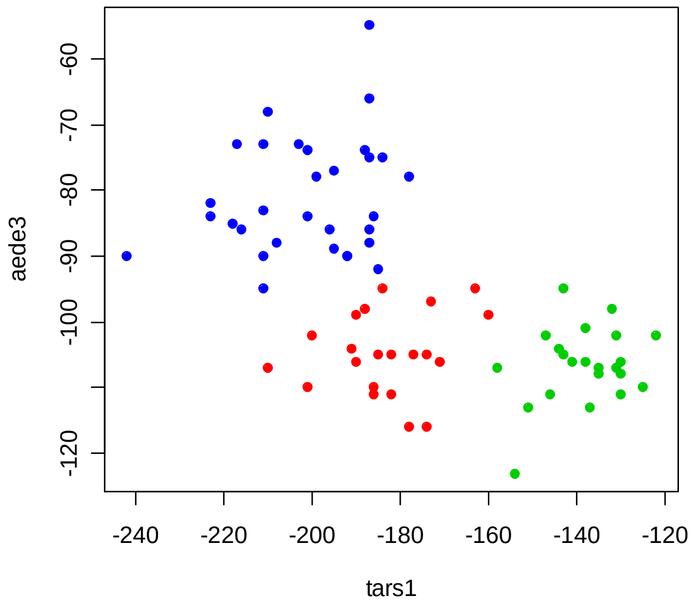

Figure 53.

The use of multidimensional scaling may enable us to see the clusters with better separation.