1 Singular Value Decomposition

1.1 SVD in R

If $\mathbf{A}$ is an $n \times p$ matrix, and the singular value decomposition of $\mathbf{A}$ is given by $\mathbf{A} = \mathbf{USV}^T$, the columns of the matrix $\mathbf{V}^T$ are the eigenvectors of the square matrix $\mathbf{A}^T\mathbf{A}$ (sometimes refered to as the minor product of $\mathbf{A}$). The singular values of $\mathbf{A}$ are equal to the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$.

The `svd()` function computes the singular value decomposition of an arbitrary rectangular matrix. Below I demonstrate the use of the `svd()` function and confirm the relationships described above:

```
> A <- matrix(c(2,1,2,3),nrow=2)
> A
     [,1] [,2]
[1,]    2    2
[2,]    1    3
> a.svd <- svd(A)
> a.svd$u
           [,1]       [,2]
[1,] -0.6618026 -0.7496782
[2,] -0.7496782  0.6618026
# R uses the notation A = u d v' rather than A = u s v'
> a.svd$d
[1] 4.1306486 0.9683709
> all.equal(A, a.svd$u %*% diag(a.svd$d) %*% t(a.svd$v))
[1] TRUE
> AtA <- t(A) %*% A
> eigen.AtA <- eigen(AtA)
> eigen.AtA
$values
[1] 17.0622577  0.9377423
$vectors
          [,1]       [,2]
[1,] 0.5019268 -0.8649101
[2,] 0.8649101  0.5019268
> all.equal(a.svd$d, sqrt(eigen.AtA$values))
[1] TRUE
```

As we discussed in lecture, the eigenvectors of square matrix, $\mathbf{A}$, point in the directions that are unchanged by the transformation specified by $\mathbf{A}$.

1.1.1 Writing our own PCA function

In lecture we discussed the relationship between SVD and PCA. Let's walk through some code that carries out PCA via SVD, and then we'll impliment our own PCA function.

```
> i.sub <- subset(iris, select=-Species)
> i.ctr <- scale(i.sub, center=T, scale=F)
> i.svd <- svd(i.ctr)

> U <- i.svd$u
> S <- diag(i.svd$d)
> V <- i.svd$v

> pc.scores <- U %*% S
# compare to fig 5.5 in your workbook
> plot(pc.scores, asp=1, col=c('red', 'darkolivegreen', 'blue')[iris$
    Species], pch=16)

> n <- nrow(i.ctr)
> pc.sdev <- sqrt((S**2/(n-1)))
> pc.sdev
         [,1]      [,2]      [,3]      [,4]
[1,] 2.056269 0.0000000 0.0000000 0.0000000
[2,] 0.000000 0.4926162 0.0000000 0.0000000
[3,] 0.000000 0.0000000 0.2796596 0.0000000
[4,] 0.000000 0.0000000 0.0000000 0.1543862


> V
            [,1]        [,2]        [,3]       [,4]
[1,]  0.36138659 -0.65658877  0.58202985  0.3154872
[2,] -0.08452251 -0.73016143 -0.59791083 -0.3197231
[3,]  0.85667061  0.17337266 -0.07623608 -0.4798390
[4,]  0.35828920  0.07548102 -0.54583143  0.7536574
```

For comparison, here's what the builtin `prcomp` function gives us:

```
> i.pca <- prcomp(i.ctr)
> i.pca$sdev
[1] 2.0562689 0.4926162 0.2796596 0.1543862
> i.pca$rotation
                     PC1         PC2         PC3        PC4
Sepal.Length  0.36138659 -0.65658877  0.58202985  0.3154872
Sepal.Width  -0.08452251 -0.73016143 -0.59791083 -0.3197231
Petal.Length  0.85667061  0.17337266 -0.07623608 -0.4798390
Petal.Width   0.35828920  0.07548102 -0.54583143  0.7536574
```

Now that we have a sense of the key calculations, let's turn this into a function. Save the following code in file named `mypca.R`.

```
# a user defined version of principal components analysis
PCA <- function(X, center=T, scale=F){
   x <- scale(X, center=center, scale=scale)
   n <- nrow(x)
   p <- ncol(x)

   x.svd <- svd(x)
   U <- x.svd$u
   S <- diag(x.svd$d)
   V <- x.svd$v

   # check for zero eigenvalues
   tolerance = .Machine$double.eps^0.5
   has.zero.singval <- any(x.svd$d <= tolerance)
   if(has.zero.singval)
     print("WARNING: Zero singular values detected")

   pc.scores <- U %*% S
   pc.sdev <- diag(sqrt((S**2/(n-1))))
   return(list(vectors = V, scores=pc.scores, sdev = pc.sdev))
}
```

Note I also included some code to warn the user when the covariance matrix is singular. Use the help to read about variables defined in '.Machine'.

Let's put our function through it's paces:

```
> source('mypca.R')
> iris.pca <- PCA(i.sub)
> plot(iris.pca$scores, asp=1)

> sing.pca <- PCA(t(i.sub))  # should have singular values equal to zero
[1] "WARNING: Zero singular values detected"

> tree.pca <- PCA(trees)
> tree.pca$sdev
[1] 17.1834214  4.9820035  0.7485858
> prcomp(trees)$sdev # compare to prcomp
[1] 17.1834214  4.9820035  0.7485858
```

To bring things full circle, let's make sure that the covariance matrix we reconstruct from our PCA analysis is equal to the covariance matrix calculated directly from the data set:

```
> n <- nrow(i.sub)
> V <- iris.pca$vectors
> S <- diag( sqrt(iris.pca$sdev**2 * (n-1)) ) # turn sdev's back into
    singular values
> reconstructed.cov <- (1/(n-1)) * V %*% S %*% S %*% t(V) # see pg. 11 of
    slides
> all.equal(reconstructed.cov, cov(i.sub), check.attributes=F)
```

```
[1] TRUE
```

Great! I seems like things are working as expected.

1.2 Creating Biplots in R

To illustrate the construction of biplots we'll use the iris data set. The built-in R function is `biplot()`.

```
# leave out the Species variable
> iris.vars <- subset(iris, select=-Species)
# read the prcomp docs and note differnces from princomp
> iris.pca <- prcomp(iris.vars)
> summary(iris.pca)

Importance of components:
                          PC1     PC2    PC3     PC4
Standard deviation     2.0563 0.49262 0.2797 0.15439
Proportion of Variance 0.9246 0.05307 0.0171 0.00521
Cumulative Proportion  0.9246 0.97769 0.9948 1.00000

> ?biplot  # read the help for biplot
> ?biplot.prcomp  # more detailed info on how biplot works with objects
    return by prcomp
> biplot(iris.pca, scale=1)  # scale = 1 - alpha
# change the biplot scaling - how does this differ?
> biplot(iris.pca, scale=0)
```

Note that the `scale` argument to biplot sets the α value we discussed during lecture, however `scale` $= 1 - \alpha$ (i.e. if `scale` $= 1$, $\alpha = 0$, and if `scale` $= 0$, $\alpha = 1$).

Assignment 1.1

1. Apply PCA to the `yeast-subnetwork-clean.txt` data set.
2. Create biplots in the first two principal components using both $\alpha = 0$ and $\alpha = 1$ (i.e. the `scale` argument to biplot).
3. In your biplots change the labels for the observations to integers using the `xlabs` argument to `biplot()`. To make the plot more readable use the `cex` argument to `biplot` to make the font size for the observations half the size of the variable labels.
4. An obvious pattern emerges in the biplot with respect to the gene MEP2. What is this pattern? What subset of conditions (rownames) is most closely related to the vector representing MEP2?

1.3 Data compression and noise filtering using SVD

Two common uses for singular value decomposition are for data compression and noise filtering. Will illustrate these with two examples involving matrices which represent image data. This example is drawn from an article by David Austin, found on a tutorial about SVD at the American Mathematical Society Website (link).

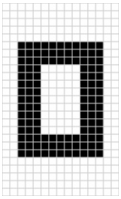
1.3.1 Data compression

Download the file `zeros.dat` from the course wiki. This is a 25×15 binary matrix that represents pixel values in a simple binary (black-and-white) image.

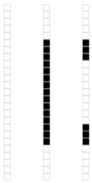
```
> z <- read.delim('zero.dat',header=F)
> z
   V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13 V14 V15
1   1  1  1  1  1  1  1  1  1   1   1   1   1   1   1
2   1  1  1  1  1  1  1  1  1   1   1   1   1   1   1
... output truncated ...

# we'll use the image() function to visualize z
> image(1:15,1:25,t(z),col=c('black','white'),asp=1)
```

This matrix data is shown below in a slightly different form that emphasizes the individual elements of the matrix. As you can see, this matrix can be thought of as being composed of just three types of vectors.



(a) The 'zero' matrix.



(b) The three vector types in the 'zero' matrix.

If SVD is working like expected it should capture that feature of our input matrix, and we should be able to represent the entire image using just three singular values and their associated left- and right-singular vectors.

```
> zsvd <- svd(z)
> round(zsvd$d,2)
 [1] 14.72  5.22  3.31  0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
     0.00  0.00  0.00
[15]  0.00
> D <- diag(zsvd$d[1:3])
> D
         [,1]     [,2]     [,3]
```

```
[1,] 14.72425 0.000000 0.000000
[2,]  0.00000 5.216623 0.000000
[3,]  0.00000 0.000000 3.314094
> U <- zsvd$u[,1:3]
> V <- zsvd$v[,1:3]
> newZ <- U %*% D %*% t(V)
> all.equal(newZ, z, check.attributes=F)
[1] TRUE

# and let's double check using the image() function
> image(1:15,1:25,t(newZ),col=c('black','white'),asp=1)
```

Our original matrix required 25×15 $(= 375)$ storage elements. Using the SVD we can represent the same data using only $15 \times 3 + 25 \times 3 + 3 = 123$ units of storage (corresponding to the truncated U, V, and D in the example above). Thus our SVD allows us to represent the same data with at less than $1/3$ the size of the original matrix. In this case, because all the singular values after the 3rd were zero this is a lossless data compression procedure.

1.3.2 Noise filtering using SVD

The file `noisy-zero.dat` is the same 'zero' image, but now sprinkled with Gaussian noise draw from a normal distribution ($N(0, 0.1)$. As in the data compression case we can use SVD to approximate the input matrix with a lower-dimensional approximation. Here the SVD is 'lossy' as our approximation throws away information. In this case we hope to choose the approximating dimension such that the information we lose corresponds to the noise which is 'polluting' our data.

```
> nz <- as.matrix(read.delim('noisy-zero.dat',header=F))
> dim(nz)
[1] 25 15
> x <- 1:15
> y <- 1:25
# create a gray-scale representation of the matrix
> image(x,y,t(nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq(0,1,0.05)))
> round(nz.svd$d,2)
 [1] 13.63  4.87  3.07  0.40  0.36  0.31  0.27  0.26  0.21  0.19  0.13
     0.11  0.09  0.06
[15]  0.04
# as before the first three singular values dominate
> nD <- diag(nz.svd$d[1:3])
> nU <- nz.svd$u[,1:3]
> nV <- nz.svd$v[,1:3]
> approx.nz <- nU %*% nD %*% t(nV)

# now plot the original and approximating matrix side-by-side
> par(mfrow=c(1,2))
> image(x,y,t(nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq(0,1,0.05)))
```

```
> image(x,y,t(approx.nz),asp=1,xlim=c(1,15),ylim=c(1,25),col=gray(seq
    (0,1,0.05)))
```

As you can see from the images you created the approximation based on the approximation based on the SVD manages to capture the major features of the matrix and filters out much of (but not all) the noise.

1.4 Image Approximation Using SVD in R

R doesn't have native support for common image files like JPEG and PNG. However, there are packages we can install that will allow us to read in such files and treat them as matrices:

```
> install.packages("jpeg", dependencies=T)
```

The `jpeg` library provides simple functions for reading and writing image files. The following code shows how to read in the `chesterbw.jpg` image which can be found in the course datasets.

The function `grid.raster` in the `grid` library can be used to draw the matrix of image data returned from the `readJPEG`. There is also a lower-level `rasterImage()` function that can be used to draw images, as shown below. The `image()` function included in R base will also draw images, but to do so conveniently we'll write a simple wrapper function called `GreyscaleImage()`.

```
> library(jpeg)
> img <- readJPEG("chesterbw.jpg")
> dim(img)
[1] 556 605
> typeof(img)
[1] "double"
> class(img)
[1] "matrix"
> ny <- dim(img)[1]  # rasterImage will draw rows along vertical axis
> nx <- dim(img)[2]
> max.pixels <- max(nx,ny)
> plot(0:max.pixels, 0:max.pixels, type='n', xlab='', ylab='',asp=1)
> ?rasterImage
> rasterImage(img, 0, 0, nx, ny)
> library(grid) # provides grid.raster function
> ?grid.raster
> grid.raster(img)  # more convenient but less flexible than rasterImage
> GreyscaleImage <- function(im){
+    rotated <- t(im[rev(1:nrow(im)), 1:ncol(im)])
+    image( rotated, col= grey(seq(0,1, length=256)), useRaster=TRUE )
+ }
> GreyscaleImage(img)
```

The output of the code above is shown in Fig 1.2.

Now we'll use SVD to create a low-dimensional approximation of this image.

Figure 1.2: My ever-faithful companion Chester.

```
> img.svd <- svd(img)
> U <- img.svd$u
> S <- diag(img.svd$d)
> Vt <- t(img.svd$v)

> U15 <- U[,1:15]  # first 15 left singular vectors
> S15 <- S[1:15,1:15]  # first 15 singular values
> Vt15 <- Vt[1:15,]  # first 15 right singular values, NOTE: we're getting
    rows rather than columns here

> approx15 <- U15 %*% S15 %*% Vt15
> GreyScaleImage(approx15)
```

The output of our approximate image is shown in Fig 1.3.

Figure 1.3: A low-dimensional approximation of Chester.

Above we created a rank 15 approximation to the rank 556 original image matrix. This approximation is crude (as judged by the visual quality of the approximating image) but it does represent a very large savings in space. Our original image required the storage of $605 \times 556 = 336380$ integer values. Our approximation requires the storage of only $15 \times 556 + 15 \times 605 + 15 = 17430$ integers. This is a saving of roughly 95%. Of course, as with any lossy compression algorithm, you need to decide what is the appropriate tradeoff between compression and data loss for your given application.

Finally, let's look at the 'error term' associated with our approximation, i.e. what we *did not* capture in the 15 singular vectors.

```
> img.diff <- img - approx15
> GreyScaleImage(img.diff)
```

An image representing the information our approximation didn't capture is shown in Fig 1.4.

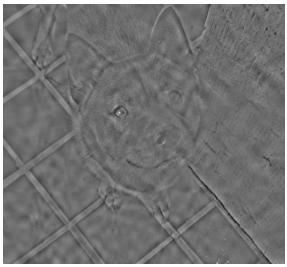


Figure 1.4: A representation of the information *not* captured by our approximation.

Assignment 1.2

Write a function, `svd_img()`, that automates the creation of a lower dimensional approximation of a grayscale image using SVD.

1. Your function should take as input a matrix representing the original image and an integer specifying the approximating dimension – i.e. function will be called as `svd_img(imgmtx,dim)`.
2. Your function should return a list of two objects: 1) an array representing the approximated image; and 2) an array representing the difference between the original and approximating images (i.e. original - approximation).
3. Test your function on various images using a variety of approximating dimensions (e.g. 5,10, 25, 50, 100, 250) on the `chesterbw.jpg` image.

In addition to your code consider the following questions:

- When analyzing `chesterbw.jpg`, at some approximating dimensions you'll notice interesting artifacts. How do these relate to the original image?
- What is the lowest approximating dimension where you would you consider the image to be recognizable as a dog?
- At what approximating dimension would you judge the image to be "close enough" to the original by the casual observer? What is the storage saving of this approximation relative to the original image?
- How does the difference array change as the approximating dimension changes? Is there a particular type of image information that seems most prominent in the difference array?