APTS Statistical Computing 2024/25: Practical Lab 1 (Tuesday)

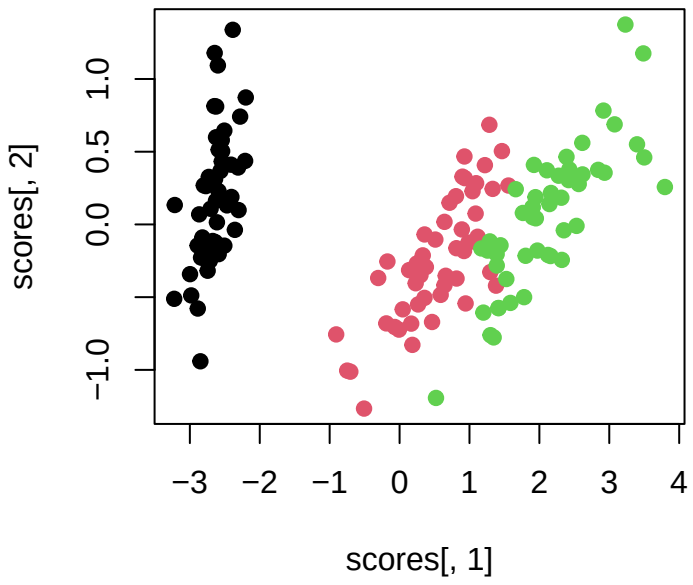
Here are some practical problems which aim to explore and reinforce some of the course material. Several problems here use simulated data: when developing statistical modelling code, it is often best to start out with data where you know what the truth is (and can generate further replicates). Do not consult the solutions[1] until you've made good attempts.

1. **PCA via SVD** The principal components analysis (PCA) of a multivariate data set was traditionally based on the eigen-decomposition of the sample covariance matrix of the data. The matrix of eigenvectors can be used to rotate the (centred) data observations to a set of uncorrelated random quantities ordered by decreasing variance. These rotated data are often known as the *scores*. We can write a small function to implement this as follows:

```
pcScoresEig = function(X) {
  Xc = sweep(as.matrix(X), 2, colMeans(X))
  eig = eigen(crossprod(Xc)/(nrow(Xc)-1), symmetric=TRUE)
  Xc %*% eig$vectors
}
```

Look at the notes/slides to make sure you understand what this function is doing. We can test it on the Iris flower data (see `?iris`), as follows:

```
Xi = iris[,-5] # We exclude the 5th column, which is not numerical
scores = pcScoresEig(Xi)
plot(scores[,1], scores[,2], col=iris[,5], pch=19)
```



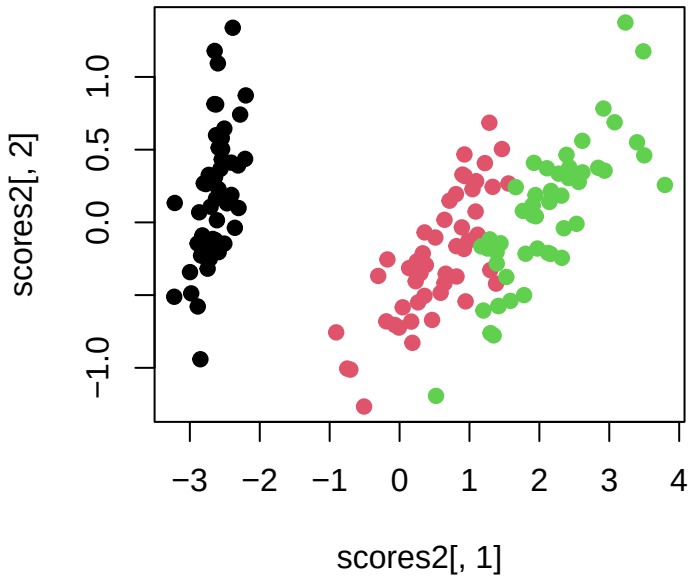
[1] Solutions will be made available from the course website before the end of the session.

This is essentially how the `princomp` function in R is implemented, and we can verify this:

```
head(scores, 3)

##            [,1]       [,2]        [,3]        [,4]
## [1,] -2.684126  0.3193972 -0.02791483 0.002262437
## [2,] -2.714142 -0.1770012 -0.21046427 0.099026550
## [3,] -2.888991 -0.1449494  0.01790026 0.019968390

head(princomp(Xi)$scores, 3)

##          Comp.1     Comp.2      Comp.3      Comp.4
## [1,] -2.684126  0.3193972  0.02791483 0.002262437
## [2,] -2.714142 -0.1770012  0.21046427 0.099026550
## [3,] -2.888991 -0.1449494 -0.01790026 0.019968390
```

(a) It turns out that the singular value decomposition of the (centred) data matrix can be used to construct the scores directly as $\mathbf{UD}$. Look at the notes/slides and think about why this is true, and write an R function, `pcScoresSvd` to implement this. Test it on the iris data, and don't worry about sign flipping (the sign of the score vectors is arbitrary).

```
## Xc = UDV' => Xc'Xc = VD^2V' which is the symmetric eigen decomp
## So XcV = UDV'V = UD.

pcScoresSvd = function(X) {
    Xc = sweep(as.matrix(X), 2, colMeans(X))
    svx = svd(Xc)
    t(t(svx$u)*svx$d)  ## UD
}
scores2 <- pcScoresSvd(Xi)
scores2 <-  scores2 %*% diag(sign(scores[1,])*sign(scores2[1,])) # Ensure same
plot(scores2[,1], scores2[,2], col=iris[,5], pch=19)
```



(b) This SVD-based method is more numerically stable than the eigendecomposition method, although if we are only interested in the first few components that is rarely a big deal. In the case of wide data ($p > n$) SVD can also be substantially more efficient. It also gives some additional insight into what the PCA "means". This is essentially how the `prcomp` function in R is implemented (which is almost always preferred to the `princomp` function). Compare your function with this (again, do not worry about the sign).

```
head(scores2, 3)
```

```
##             [,1]       [,2]        [,3]        [,4]
## [1,] -2.684126  0.3193972 -0.02791483 0.002262437
## [2,] -2.714142 -0.1770012 -0.21046427 0.099026550
## [3,] -2.888991 -0.1449494  0.01790026 0.019968390
```

```
head(prcomp(Xi)$x, 3)
```

```
##             PC1        PC2         PC3         PC4
## [1,] -2.684126 -0.3193972  0.02791483 0.002262437
## [2,] -2.714142  0.1770012  0.21046427 0.099026550
## [3,] -2.888991  0.1449494 -0.01790026 0.019968390
```

(c) Simulate some random (eg.) $5,000 \times 1,000$ test data, and time your two implementations.

```
X = matrix(rnorm(5000*1000),ncol=1000)
system.time(pcScoresEig(X))
```

```
##    user  system elapsed
##   7.485   0.031   7.518
```

```
system.time(pcScoresSvd(X))

##    user  system elapsed
##  13.910   0.089  13.999
```

(d) Think about how to use the SVD to compute the empirical variances or standard deviations of the scores, then create a function that takes as input the data matrix `X` and that uses its SVD to compute the standard deviations of the scores. Check it against `prcomp` for the iris data to make sure you've done it correctly.

```
## Xc = UDV' => UD = XcV
## So cov(UD) = V'cov(Xc)V. Substituting the empirical
## cov(Xc) = Xc'Xc/(n-1), we have cov(UD) = V'Xc'XcV/(n-1) = V'VD^2V'V/(n-1)
## = D^2/(n-1). i.e. the singular values over sqrt(n-1) give the sd prcomp.

pcSdsSvd = function(X) {
    Xc = sweep(as.matrix(X), 2, colMeans(X))
    SVD = svd(Xc)
    SVD$d/sqrt(nrow(Xc)-1)
}
pcSdsSvd(Xi)

## [1] 2.0562689 0.4926162 0.2796596 0.1543862

prcomp(Xi)$sd

## [1] 2.0562689 0.4926162 0.2796596 0.1543862
```

2. **Ridge Regression**. For a linear regression model,

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

we know that the quadratic loss $L_0(\boldsymbol{\beta}) = \|\boldsymbol{\epsilon}\|^2 = \boldsymbol{\epsilon}^\mathsf{T}\boldsymbol{\epsilon}$ is minimised wrt $\boldsymbol{\beta}$ when $\boldsymbol{\beta}$ is a solution to the normal equations,

$$\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

In *ridge regression*, the slightly modified quadratic loss function $L_\lambda(\boldsymbol{\beta}) = \|\boldsymbol{\epsilon}\|^2 + \lambda\|\boldsymbol{\beta}\|^2$ is used, for some ridge penalty $\lambda > 0$, which encourages shrinkage of the regression coefficients towards zero.

(a) Show that for a given fixed $\lambda > 0$, the loss $L_\lambda(\boldsymbol{\beta})$ is minimised when $\boldsymbol{\beta}$ is a solution to

$$(\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} = \mathbf{X}^\mathsf{T}\mathbf{y}.$$

```
## L = e'e + lb'b = (y-Xb)'(y-Xb) + lb'b = y'y - 2b'X'y + b'X'Xb + lb'b
## gradient of L = -2X'y + 2X'Xb + 2lb = -2X'y + 2(X'X + lI)b
## So gradient of L = 0 => (X'X + lI)b = X'y
```

(b) Starting from the singular value decomposition, $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^\mathsf{T}$, show that the optimal $\widehat{\boldsymbol{\beta}}_\lambda$ can be written as

$$\widehat{\boldsymbol{\beta}}_\lambda = \mathbf{V}\mathbf{D}_\lambda\mathbf{U}^\mathsf{T}\mathbf{y},$$

where $\mathbf{D}_\lambda$ is a diagonal matrix with entries $d_i^\lambda = d_i/(d_i^2 + \lambda)$. Note that this means $\widehat{\boldsymbol{\beta}}_\lambda$ can be computed for as many different λ as desired, all for the cost of one single expensive SVD operation.

```
##     (X'X + lI)b = X'y
## => (VDU'UDV' + lI)b = VDU'y
## => (VDDV' + lI)b = VDU'y
## => V(DD + lI)V'b = VDU'y
## => b = V(DD + lI)^{-1}DU'y
## => b = V[(DD + lI)^{-1}D]U'y
## => b = VEU'y, where diagonal E = (DD + lI)^{-1}D
```

(c) In practice, both the data, $\mathbf{y}$, and the covariate matrix $\mathbf{X}$ are centred before ridge regression is applied, since then the model can be fit without an intercept, and typically you would not want to shrink the intercept. Write a function,

```
ridge(y, X, lambda)
```

which expects an n-vector `y`, an $n \times p$ matrix `X`, and a q-vector of λ values where the ridge solution is required. The function should return a $p \times q$ matrix of ridge regression parameters, with each column representing a solution for a given λ.

```
ridge = function(y, X, lambda) {
  y = y - mean(y)
  X = sweep(as.matrix(X), 2, colMeans(X))
  SVD = svd(X)
  uty = as.vector(t(SVD$u) %*% y)
  D = outer(SVD$d, lambda, function(d,l){d/(d*d+l)})
  SVD$v %*% (D * uty) # 1st product is matrix, 2nd is elementwise
}
```

(d) For the `trees` dataset, regress volume on the other two variables for a range of shrinkage parameters.

```
ridge(trees[,3], trees[,1:2], c(0,exp(0:5)))
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## [1,] 4.7081605 4.6868475 4.6507090 4.5554535 4.3166299 3.7858307 2.8648009
## [2,] 0.3392512 0.3444192 0.3531561 0.3760244 0.4322559 0.5504595 0.7226325
```

Ensure that your solution matches up with that of `lm` in the case $\lambda = 0$.

```
lm(as.vector(trees[,3]) ~ as.matrix(trees[,1:2]))$coefficients
```

```
##                   (Intercept)  as.matrix(trees[, 1:2])Girth
##                    -57.9876589                     4.7081605
## as.matrix(trees[, 1:2])Height
##                     0.3392512
```

3. **Interactively exploring optimisation routines**. For this exercise, we will use the `FLtools` package from:

   `https://bitbucket.org/finnlindgren/FLtools/`

   developed by Finn Lindgren (a previous lecturer for this course). You can install it with:

```
library(devtools)
devtools::install_bitbucket("finnlindgren/FLtools")
```

If you don't have the `devtools` package, first install it with:

```
install.packages("devtools")
```

Once you have installed the `FLtools` package, you should be able to load it with

```
library(FLtools)
```

Make sure you have this package installed before proceeding to the next step.

(a) Start the `optimisation` shiny app:

```
FLtools::optimisation()
```

This should start a Shiny web application. It will also attempt to start up a tab in your browser connected to the session. If this doesn't work, just connect your browser to the URL of the Shiny app. Make sure the Shiny app is running in a browser window before proceeding to the next step.

(b) For the "Simple (1D)" and "Simple (2D)" functions, familiarise yourself with the "Step", "Converge", and "Reset" buttons.

(c) Choose different optimisation starting points by clicking in the figure.

(d) Explore the different optimisation methods and what they display in the figure for each optimisation step[2][3][4]. Also observe the diagnostic output box and how the number of function, gradient, and Hessian evaluations differ between the methods.

(e) For the "Rosenbrock (2D)" function, observe the differences in convergence behaviour for the four different optimisation methods.

(f) For the "Multimodal" functions, explore how the optimisation methods behave for different starting points.

(g) How far out can the optimisation start for the "Spiral" function? E.g., try the "Newton" method, starting in the top right corner of the figure.

4. Write your own code to optimise Rosenbrock's function

$$f(x, z) = 100(z - x^2)^2 + (1 - x)^2$$

by Newton's method. In particular, create the function:

```
newton_step <- function(x0, f, gf, hf, mh = 5, me=0.0001)
```

where:

- `x0` is the initial point;

[2] LS stands for "line search".

[3] The simplex/triangle shapes are shown for each "Simplex" method step in blue. The "best" points for each simplex are connected (magenta).

[4] The Newton methods display the true quadratic Taylor approximations (red) as well as the approximations used to find the proposed steps (blue).

- `f` is a function that evaluates $f(x, z)$;
- `gf` returns the gradient of $f(x, z)$;
- `hf` returns the Hessian of $f(x, z)$;
- `mh` is the maximum number of step-halving step use during back-tracking;
- `me` is the value at which any Hessian eigenvalue falling between 0 and `me` should be fixed.

Your function should return the estimated minimiser $[x^*, y^*]$ of the Rosenbrock function. Make sure that your function implements backtracking for step-length selection and that the Hessian is perturbed to positive definiteness. Ensure that you have implemented it correctly by comparing your output (and implementation) with that of the Shiny app from the first exercise.

```
## One step of a Newton method
newton_step <- function(x0, f, gf, hf, mh=5, me=0.0001) {
    h0 = hf(x0) # raw hessian
    eig = eigen(h0, symmetric=TRUE)
    e0 = eig$values # raw e-vals
    e1 = abs(e0) # flip negative e-vals
    e1[e1 < me] = me # inflate small e-vals
    ## since we have eigendecomposition, directly compute Newton step
    ei = 1/e1 # eigenvalues of inverse
    delta = -(eig$vectors %*% (ei * (t(eig$vectors) %*% gf(x0))))
    x1 = x0 + delta
    if (f(x1) > f(x0)) {
        for (i in 1:mh) {
            delta = delta / 2 # halve step length
            x1 = x0 + delta
            if (f(x1) < f(x0)) return(x1)
        }
        warning("Newton step didn't descend.")
    }
    x1
}

## Newton iteration
newton <- function(x0, f, gf, hf, mi=100, eps=1e-10, plt=FALSE, ...) {
    x = x0
    for (i in 1:mi) {
        x = newton_step(x, f, gf, hf, ...)
        if (plt) {
            lines(c(x0[1],x[1]), c(x0[2],x[2]), col=2)
            message(paste(i, " "), appendLF=FALSE)
            print(x)
        }
        ## mixed relative/absolute convergence check
        if (sum(abs(x-x0)) < eps*sum(abs(x0)) + eps) return(x)
        x0=x
    }
```

```
    warning("Max iterations exceeded before convergence.")
    x
}

## Application to the Rosenbrock function

## Define function
rb <- function(x,z) {
  100*(z-x^2)^2 + (1-x)^2
}
## Clearly has min value of 0 at (1,1)

## Plot contours of the function (logged)
n <- 100
x <- seq(-1.5,1.5,length=n)
z <- seq(-.5,1.5,length=n)
f <- outer(x,z,rb)
contour(x,z,matrix(log10(f),n,n),levels=(1:10/2))

## Gradient
rb.grad <- function(x,z) {
  g <- rep(NA,2)
  g[1] <- 400*(x^3-z*x) + 2*(x-1)
  g[2] <- 200*(z-x^2)
  g
}

## Hessian
rb.hess <- function(x,z) {
  H <- matrix(NA,2,2)
  H[1,1] <- 1200*x^2 - 400*z + 2
  H[2,1] <- H[1,2] <- -400*x
  H[2,2] <- 200
  H
}

## Versions with a single vector argument:
rbv <- function(x) rb(x[1],x[2])
rbg <- function(x) rb.grad(x[1],x[2])
rbh <- function(x) rb.hess(x[1],x[2])

xx <- newton(c(-.5,1), rbv, rbg, rbh, plt=TRUE)

## 1

##             [,1]
## [1,] -1.0873529
## [2,]  0.5498683
```

```
## 2

##           [,1]
## [1,] -1.070981
## [2,]  1.146732

## 3

##            [,1]
## [1,] -0.8252801
## [2,]  0.6204839

## 4

##            [,1]
## [1,] -0.6861652
## [2,]  0.4514698

## 5

##            [,1]
## [1,] -0.5130686
## [2,]  0.2236005

## 6

##             [,1]
## [1,] -0.34358995
## [2,]  0.08933102

## 7

##             [,1]
## [1,] -0.14438037
## [2,] -0.01883876

## 8

##             [,1]
## [1,] -0.01632909
## [2,] -0.01613049

## 9

##              [,1]
## [1,]  0.221162804
## [2,] -0.007489412

## 10
```

```
##             [,1]
## [1,] 0.28458355
## [2,] 0.07696561
```

```
## 11
```

```
##           [,1]
## [1,] 0.4828215
## [2,] 0.1918072
```

```
## 12
```

```
##           [,1]
## [1,] 0.5386610
## [2,] 0.2870376
```

```
## 13
```

```
##           [,1]
## [1,] 0.6807331
## [2,] 0.4416540
```

```
## 14
```

```
##           [,1]
## [1,] 0.7404237
## [2,] 0.5446642
```

```
## 15
```

```
##           [,1]
## [1,] 0.8919928
## [2,] 0.7726780
```

```
## 16
```

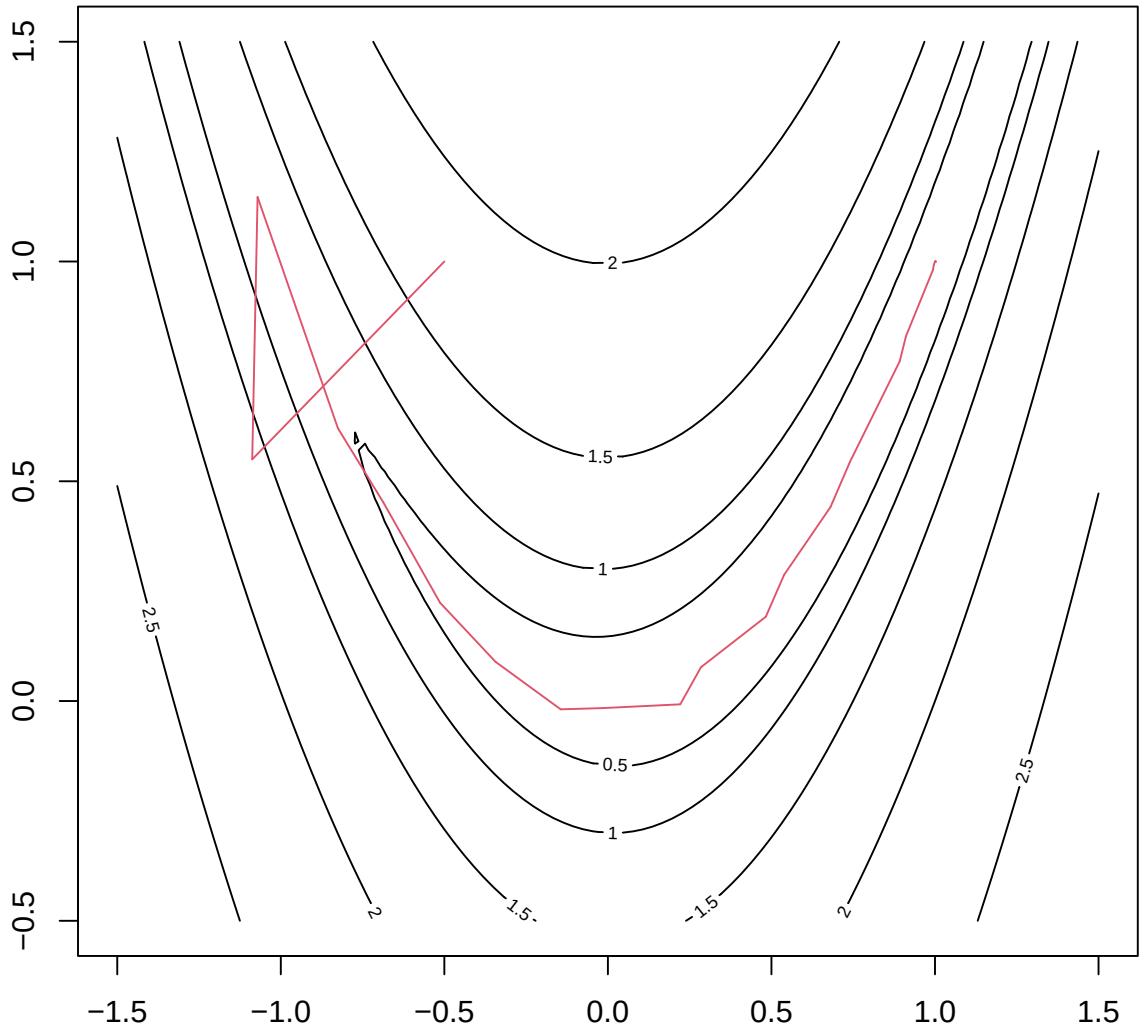
```
##           [,1]
## [1,] 0.9112983
## [2,] 0.8300919
```

```
## 17
```

```
##           [,1]
## [1,] 0.9938468
## [2,] 0.9809172
```

```
## 18
```

```
##           [,1]
## [1,] 0.9964510
## [2,] 0.9929077

## 19

##           [,1]
## [1,] 0.9999952
## [2,] 0.9999778

## 20

##      [,1]
## [1,]    1
## [2,]    1

## 21

##      [,1]
## [1,]    1
## [2,]    1

## 22
```




```
##      [,1]
## [1,]    1
## [2,]    1

print(xx)

##      [,1]
## [1,]    1
## [2,]    1

print(rbv(xx))
```

```
## [1] 0
```