Project3

Robert Norberg, Jung-Han Wang

Monday, November 20, 2014

Project 3

Problem 1

```
##Robert's Path
mydat <- read.table('/home/robert/cloud/Classes/STA6106 Stat Computing/Project2/Project3/training datase

##Jung-Han's Path
# mydat <- read.table("E:/Cloud Storage/Dropbox/Life long study/Ph.D/Lecture/2014 Fall/Statistical Comp
# mydat2 <- read.table("E:/Cloud Storage/Dropbox/Life long study/Ph.D/Lecture/2014 Fall/Statistical Com
my_matrix <- as.matrix(mydat)
# my_matrix2 <- as.matrix(mydat2)
```

This problem is to get some codes to perform the support vector data description (SVDD)

a. Write an *R* function to perform the SVDD.

First, we want to compute the kernel matrix

$$\begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & ... & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & ... & k(x_2, x_n) \\ ... & ... & ... & ... \\ k(x_n, x_1) & k(x_n, x_2) & ... & k(x_n, x_n) \end{bmatrix}$$

To do this, we must define a kernel function. This function essentially calculates the distance between each pair of data vectors. For simplicity, we begin by using the simplest distance, the euclidean distance. The euclidean distance between two data vectors is just their dot product. The `kernlab` package includes a function `vanilladot()` that when called, creates another function that will compute these dot products.

```
my_kernel <- vanilladot()
```

We have now created a function `my_kernel()` that will calculate the linear distance between two data vectors for us. We check that this is equivalent to the dot product.

```
my_kernel(my_matrix[1, ], my_matrix[2, ]) # dot prod using kernel function
```

```
      [,1]
[1,] 37.49
```

```
crossprod(my_matrix[1, ], my_matrix[2, ]) # dot prod using base R function
```

```
      [,1]
[1,] 37.49
```

```r
my_matrix[1, ] %*% my_matrix[2, ] # old school matrix multiplication operator
```

```
      [,1]
[1,] 37.49
```

Now that we have defined a function for applying our kernel function to a pair of data vectors, we can easily create a kernel matrix from our data matrix. There is a handy function in the `kernlab` package called `kernelMatrix()` that does exactly this. It requires as arguments `kernel`, the kernel function to be used, and `x`, the data matrix from which to compute the kernel matrix. We pass the function our kernel function `my_kernel()` and our data matrix `my_matrix`. The function returns a *nxn* (66x66) matrix of class `kernelMatrix`.

```r
H <- kernelMatrix(kernel=my_kernel, x=my_matrix)
dim(H)
```

```
[1] 66 66
```

```r
class(H)
```

```
[1] "kernelMatrix"
attr(,"package")
[1] "kernlab"
```

The SVDD problem can be stated mathematically as

$$\underset{\alpha}{max} \sum_i \alpha_i \; k(x_i, x_i) - \sum_{i,j} \alpha_i \alpha_j \; k(x_i, x_j)$$

subject to $\alpha_i \geq 0$ and $\sum \alpha_i = 1$.

The quadratic solver in the `kernlab` package solves quadratic programming problems in the form

$$min(c'x + \frac{1}{2}x'Hx)$$

subject to $b \leq Ax \leq b + r$ and $l \leq x \leq u$.

To re-state the SVDD problem in the form required by the quadratic solver we set

$$x' = [\alpha_1, \alpha_2, ..., \alpha_n]$$

$$H = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & ... & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & ... & k(x_2, x_n) \\ ... & ... & ... & ... \\ k(x_n, x_1) & k(x_n, x_2) & ... & k(x_n, x_n) \end{bmatrix}$$

$$c' = [k(x_1, x_1), k(x_2, x_2), ..., k(x_n, x_n)] = diag(H)$$

then

$$c'x + \frac{1}{2}x'Hx = [k(x_1, x_1), ..., k(x_n, x_n)] \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix} + \frac{1}{2}(2)[\alpha_1, ..., \alpha_n] \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & ... & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & ... & k(x_2, x_n) \\ ... & ... & ... & ... \\ k(x_n, x_1) & k(x_n, x_2) & ... & k(x_n, x_n) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix}$$

To re-state the constraints of the SVDD problem in the form required by the quadratic solver, we set

$$b = 1,\ A = [1, 1, ..., 1],\ x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix}, r = 0$$

$$l = \begin{bmatrix} 0 \\ 0 \\ ... \\ 0 \end{bmatrix}, x = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix}, u = \begin{bmatrix} \infty \\ \infty \\ ... \\ \infty \end{bmatrix}$$

then $b \leq Ax \leq b + r$ is equivalent to

$$1 \leq [1, 1, ..., 1] \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_n \end{bmatrix} \leq 1 + 0$$

or

$$1 \leq \sum \alpha_i \leq 1$$

and $l \leq x \leq u$ is equivalent to $0 \leq \alpha_i \leq \infty$.

Using the re-formulation of the problem, we pass the appropriate objects to the `ipop()` quadratic programming solver included in the `kernlab` package. this returns the vector of α's that minimize the stated problem $min(c'x + \frac{1}{2}x'Hx)$.

```
my_c <- (-1)*diag(H)
my_H <- (2)*H
my_A <- rep(1, nrow(my_matrix))
my_b <- 1
my_l <- rep(0, nrow(my_matrix))
my_u <- rep(1, nrow(my_matrix))
my_r <- 0
my_solution <- ipop(c=my_c, H=my_H, A=my_A, b=my_b, l=my_l, u=my_u, r=my_r, maxiter=300, margin=0.001)
my_alphas <- my_solution@primal # use @ symbol to access s4 slot
```

We check to make sure our α_i's sum to one,

```
sum(my_alphas)
```

```
[1] 1
```

We believe we have found a solution to the SVDD problem, so we wrap the commands shown above into a function that takes as input a data matrix `x` and a kernel function `k`.

```
SVDD <- function(x, k){
  H <- kernelMatrix(kernel=k, x=x)
  n <- nrow(x)
  solution <- ipop(c=(-1)*diag(H),
                   H=2*H,
                   A=rep(1, n),
                   b=1,
                   l=rep(0, n),
                   u=rep(1, n),
                   r=0,
                   maxiter=300,
                   margin=0.001)

  alphas <- solution@primal

  # catch errors
  if(signif(sum(alphas), 7) != 1){
    stop("An error has occurred! The soultion values do not sum to one.")
    }
##  if(any(signif(alphas, 7) < 0)){
##    stop("An error has occurred! The solution values include at least one negative value, which is no
##    }

  return(alphas)
  }

test_alphas <- SVDD(my_matrix, my_kernel)
```

b. Write an `R` function to perform the prediction of a new observation using SVDD.

When x_s is Support Vector when $\alpha_s > 0$

Followed by, we want to select support vector x_S based on α_s.

Therefore, we combined α with original dataset, then make the selection.

```
sv_indices <- which(round(test_alphas, 7)>0)
my_matrix[sv_indices, ] # these are the support vectors
```

```
      V1 V2  V3  V4
[1,] 4.3  3 1.1 0.1
[2,] 7.6  3 6.6 2.1
```

The prediction of new observation can be calculated by solving R. The function of R can be written as:

$$R^2 = (x_s' \cdot x_s) - 2\sum_{i=1}^{N} \alpha_i (x_s' \cdot x_i) + \sum_{i,j=1}^{N} \alpha_i \alpha_j (x_i' \cdot x_j)$$

```r
dim(t(my_matrixa_s))
dim(my_matrix)

dim(t(my_matrixa_s))

xs<-c(5.8,2.8,5.1,2.4)

## 1. Calculate Xs dot xs
r1<-crossprod(xs,xs)

## 2. Calculate {   -2 sum(alpha(x_s dot x_i))   }
r2<-0
r2.old<-0
r2.func<-function (xs,test_alphas,my_matrix){
for (i in 1:length(test_alphas)){
r2.old<-r2
r2<- -2*test_alphas[i]%*%crossprod(xs,my_matrix[i,])
r2<- r2.old+r2
return (r2)
}
}
r2<-r2.func(xs,test_alphas,my_matrix)

## 3. Calculate {   sum(alpha_i*alpha_j (x_i dot x_j))   }

for(i in 1:3){for(j in 1:3){
a=i+j}}

r3<-0
r3.old<-0

r3.func<-function (test_alphas,my_matrix){
for (i in 1:length(test_alphas)){for (j in 1:length(test_alphas)){
r3.old<-r3
r3<- test_alphas[i]*test_alphas[j]*crossprod(my_matrix[i,],my_matrix[j,])
r3<- r3.old+r3
return (r3)
}
}
}
r3<-r3.func(test_alphas,my_matrix)

r.sqr<-r1+r2+r3
```

A test sample z is accepted when $\leq R^2$

```r
z<-c(1,1,1,1)
z.func<-function(z,r3){
  z1<-crossprod(z,z)
  z2<-r2.func(z,test_alphas,my_matrix)
  z3<-r3
  return(z1+z2+z3)
}
```

```
deci<-z.func(z,r3)

if(deci <= r.sqr){
    ("z is accepted")
    }
```

Apply function developed in problem 1-a, c. Write an R function for detecting potential outliers for a new set of observations, along with the upper threshold.

Problem 2

The goal of problem 2 is to perform the support vector data description (SVDD) using the Mahalanobis kernel function. We will simplify the problem by using the identity function for g.

a. Write an R function to compute the Mahalanobis kernel distance $\boldsymbol{d_g(x)}$
b. Write an R function to perform the Mahalanobis kernel SVDD.
c. Write an R function to perform the prediction of a new observation using the Mahalanobis kernel SVDD.
d. Write an R function for detecting potential outliers for a new set of observations, along with the upper threshold.

Appendix with R code

```
# Clear working environment
rm(list=ls())
library(ggplot2) # for plots
library(kernlab)

# Options for document compilation
knitr::opts_chunk$set(warning=FALSE, message=FALSE, comment=NA, fig.width=4, fig.height=3)
##Robert's Path
mydat <- read.table('/home/robert/cloud/Classes/STA6106 Stat Computing/Project2/Project3/training datase

##Jung-Han's Path
# mydat <- read.table("E:/Cloud Storage/Dropbox/Life long study/Ph.D/Lecture/2014 Fall/Statistical Comp
# mydat2 <- read.table("E:/Cloud Storage/Dropbox/Life long study/Ph.D/Lecture/2014 Fall/Statistical Com
my_matrix <- as.matrix(mydat)
# my_matrix2 <- as.matrix(mydat2)

my_kernel <- vanilladot()


my_kernel(my_matrix[1, ], my_matrix[2, ]) # dot prod using kernel function
crossprod(my_matrix[1, ], my_matrix[2, ]) # dot prod using base R function
my_matrix[1, ] %*% my_matrix[2, ] # old school matrix multiplication operator
H <- kernelMatrix(kernel=my_kernel, x=my_matrix)
dim(H)
class(H)
my_c <- (-1)*diag(H)
```

```r
my_H <- (2)*H
my_A <- rep(1, nrow(my_matrix))
my_b <- 1
my_l <- rep(0, nrow(my_matrix))
my_u <- rep(1, nrow(my_matrix))
my_r <- 0
my_solution <- ipop(c=my_c, H=my_H, A=my_A, b=my_b, l=my_l, u=my_u, r=my_r, maxiter=300, margin=0.001)
my_alphas <- my_solution@primal # use @ symbol to access s4 slot
sum(my_alphas)
SVDD <- function(x, k){
  H <- kernelMatrix(kernel=k, x=x)
  n <- nrow(x)
  solution <- ipop(c=(-1)*diag(H),
                   H=2*H,
                   A=rep(1, n),
                   b=1,
                   l=rep(0, n),
                   u=rep(1, n),
                   r=0,
                   maxiter=300,
                   margin=0.001)

  alphas <- solution@primal

  # catch errors
  if(signif(sum(alphas), 7) != 1){
    stop("An error has occurred! The soultion values do not sum to one.")
    }
##  if(any(signif(alphas, 7) < 0)){
##    stop("An error has occurred! The solution values include at least one negative value, which is no
##    }

  return(alphas)
  }

test_alphas <- SVDD(my_matrix, my_kernel)
sv_indices <- which(round(test_alphas, 7)>0)
my_matrix[sv_indices, ] # these are the support vectors
dim(t(my_matrixa_s))
dim(my_matrix)

dim(t(my_matrixa_s))

xs<-c(5.8,2.8,5.1,2.4)

## 1. Calculate Xs dot xs
r1<-crossprod(xs,xs)

## 2. Calculate {   -2 sum(alpha(x_s dot x_i))   }
r2<-0
r2.old<-0
r2.func<-function (xs,test_alphas,my_matrix){
for (i in 1:length(test_alphas)){
```

```r
r2.old<-r2
r2<- -2*test_alphas[i]%*%crossprod(xs,my_matrix[i,])
r2<- r2.old+r2
return (r2)
}
}
r2<-r2.func(xs,test_alphas,my_matrix)

## 3. Calculate {   sum(alpha_i*alpha_j (x_i dot x_j))   }

for(i in 1:3){for(j in 1:3){
a=i+j}}

r3<-0
r3.old<-0

r3.func<-function (test_alphas,my_matrix){
for (i in 1:length(test_alphas)){for (j in 1:length(test_alphas)){
r3.old<-r3
r3<- test_alphas[i]*test_alphas[j]*crossprod(my_matrix[i,],my_matrix[j,])
r3<- r3.old+r3
return (r3)
}
}
}
r3<-r3.func(test_alphas,my_matrix)

r.sqr<-r1+r2+r3
z<-c(1,1,1,1)
z.func<-function(z,r3){
  z1<-crossprod(z,z)
  z2<-r2.func(z,test_alphas,my_matrix)
  z3<-r3
  return(z1+z2+z3)
}
deci<-z.func(z,r3)

if(deci <= r.sqr){
    ("z is accepted")
    }
```