STA6106 Statistical Computing Project 3

Robert Norberg, Jung-Han Wang

Monday, November 24, 2014

Problem 1

```
##Robert's Path
mydat <- read.table('/home/robert/cloud/Classes/STA6106 Stat Computing/Project2/Project3/training datas
mydat2 <- read.table('/home/robert/cloud/Classes/STA6106 Stat Computing/Project2/Project3/data_project3

##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
```

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_matrix <- as.matrix(mydat)
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
```

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

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

It is useful to note here that our new function `my_kernel()` is a special kind of function - that is, it belongs to the class `vanillakernel`.

```
class(my_kernel)
```

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

Regular `R` users are familiar with classes like `matrix` and `function`, but what is the `vanillakernel` class?

```
getClass('vanillakernel')
```

```
Class "vanillakernel" [package "kernlab"]

Slots:

Name:     .Data     kpar
Class: function     list

Extends:
Class "kernel", directly
Class "function", by class "kernel", distance 2
Class "OptionalFunction", by class "kernel", distance 3
Class "PossibleMethod", by class "kernel", distance 3
Class "kfunction", by class "kernel", distance 3
```

The `getClass()` command reveals that objects of the class `vanillakernel` should have two slots, one named `.Data` that contains a function and one named `kpar` that contains a list. It also reveals that `vanillakernel` is an extension of the class `kernel`, which is itself an extension of the class `function`. More on this later.

Now that we have defined an `R` 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 (`mydat`) as a matrix.

```
K <- kernelMatrix(kernel=my_kernel, x=as.matrix(mydat))
```

The function returns a $n \times n$ (66×66) matrix of class `kernelMatrix`.

```
dim(K)
```

```
[1] 66 66
```

```r
class(K)
```

```
[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 (a command called `ipop()`) 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 first note that the SVDD problem is a miximization, while the solver computes a minimization. Thus, we re-state the SVDD problem as

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

, subject to the same constraints $\alpha_i \geq 0$ and $\sum \alpha_i = 1$.

Thus, if we set

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

$$H = 2K = 2 \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & ... & k(x_1, x_n) \\ k(x_2, x_1) & k(x_2, x_2) & ... & k(x_2, x_n) \\ ... & ... & ... & ... \\ k(x_n, x_1) & k(x_n, x_2) & ... & k(x_n, x_n) \end{bmatrix}$$

$$c' = (-1)\,[k(x_1, x_1), k(x_2, x_2), ..., k(x_n, x_n)] = (-1)diag(K)$$

then

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

$$= (-1)\,(k(x_1, x_1)\alpha_1 + ... + k(x_n, x_n)\alpha_n) + left(\alpha_1\alpha_1 k(x_1, x_1) + \alpha_1\alpha_2 k(x_1, x_2) + ... + \alpha_n\alpha_n k(x_n, x_n)$$

$$= -\sum_i \alpha_i \; k(x_i, x_i) + \sum_{i,j} \alpha_i \alpha_j \; k(x_i, x_j)$$

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 $0 \leq \alpha_i \leq \infty$.

Since $\alpha_i \geq 0, i = 1, 2, ..., n$ and $\sum_i \alpha_i = 1$, no single α_i may be greater than one, because then a negative valued α_i would be required to offset it to meet the condition $\sum_i \alpha_i = 1$, and negative valued α_i's are not allowed by the first condition. Because of the summation condition, $0 \leq \alpha_i \leq \infty$ is really equivalent to $0 \leq \alpha_i \leq 1$. Thus we can change u from $\begin{bmatrix} \infty \\ \infty \\ ... \\ \infty \end{bmatrix}$ to $\begin{bmatrix} 1 \\ 1 \\ ... \\ 1 \end{bmatrix}$. This will save us considerable search time when running the quadratic solver.

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(K)
my_H <- (2)*K
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=1E-6)
my_alphas <- my_solution@primal # use @ symbol to access s4 slot
```

Note that the `margin` argument specifies how closely the solution obeys the constraints, so the α values found in the solution are not exact, but are accurate to at least six decimal places (since we set `margin` to 1E-6).

```
my_alphas <- round(my_alphas, abs(log10(1E-6)))
```

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

```
sum(my_alphas)
```

```
[1] 1
```

and that none of them are less than zero.

```
any(my_alphas < 0)
```

```
[1] FALSE
```

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`, a kernel function `k`, and a tolerance `tol` (to be passed to `ipop()`'s `margin` argument). The function returns an object of class `SVDD` (an S4 class that we have defined, but its definition is not shown).

```
[1] "summary"
```

```
[1] "summary"
```

```
svdd <- function(x, k, tol=1E-6){
  K <- kernelMatrix(kernel=k, x=x)
  n <- nrow(x)
  solution <- ipop(c=(-1)*diag(K),
                   H=2*K,
                   A=rep(1, n),
                   b=1,
                   l=rep(0, n),
                   u=rep(1, n),
                   r=0,
                   maxiter=300,
                   margin=tol)

  alphas <- solution@primal
  alphas <- round(alphas, abs(log10(tol))) # round based on tol

  svs <- x[alphas > 0, ] # these are the support vectors

  return(new('SVDD',
             data=x,
             alphas=alphas,
             support_vectors=svs,
             kernel=k,
             tolerance=tol,
             fun_call=match.call()
             ))
  }
```

We test the function on our test data set and inspect it using a summary method, which we defined with the class definition for the class SVDD. (SVDD class definition and summary method definition are not shown for the sake of brevity.)

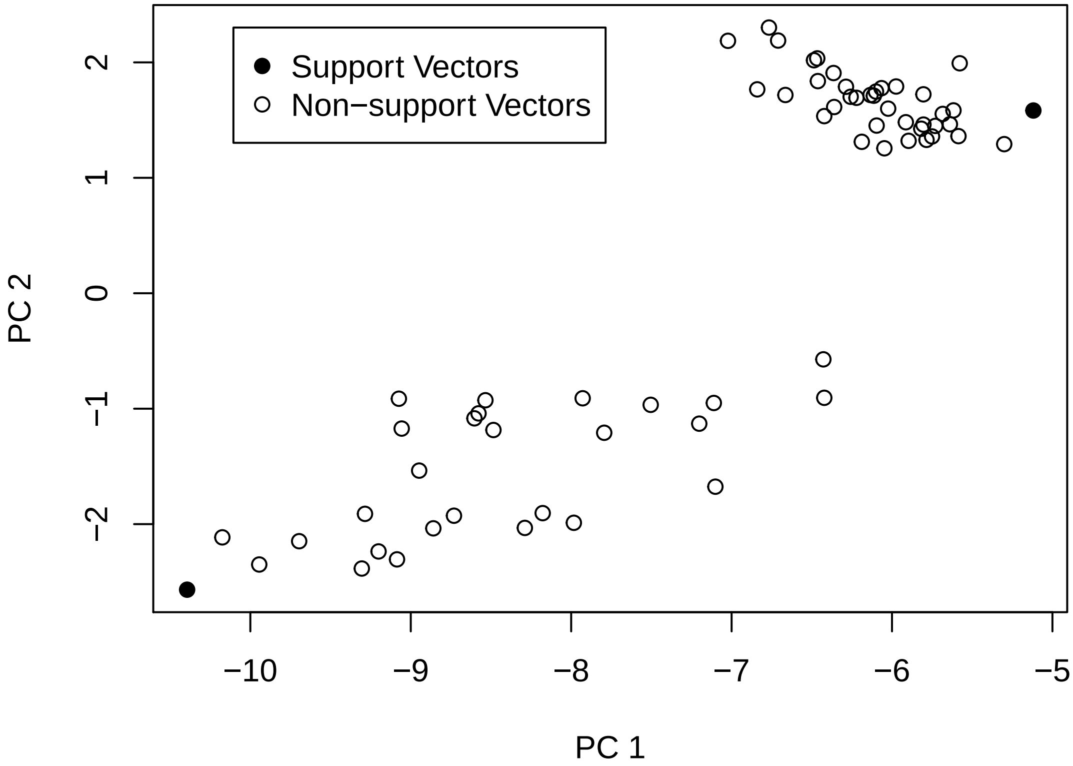
```
# supplying no value to tol implies it should use the default value tol = 1E-6
euclidean_solution <- svdd(x=my_matrix, k=my_kernel)
summary(euclidean_solution)
```

```
SVDD object using kernel function vanillakernel:
2 support vectors identified out of 66 observations.


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

We examine our solution to see if it makes sense. The support vectors found should be the "edge cases" of the input data. Since the input data is defined in four dimensions, it is difficult to visualize all of its variability on a simple X-Y plot. To capture as much of the data's variability as possible in our plot, we plot the data along it's first two principal components and shade the support vectors. We abstract this plot away by defining a plot method for the SVDD class defined earlier. We invoke our plot method on our SVDD object euclidean_solution and the plot below is produced.

```
plot(euclidean_solution)
```



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

The distance from a data vector to the center of the SVDD spheroid in the kernel space is given by

$$R^2_{x^*} = k(x^*, x^*) - 2\sum_{i=1}^{n} \alpha_i k(x^*, x_j) + \sum_{i,j=1}^{n} \alpha_i \alpha_j k(x_i, x_j)$$

For the purpose of predicting outliers, we are interested in the radius of the SVDD spheroid in the kernel space, that is, the distance from the center of the spheroid to the very edge of the spheroid (in the kernel space). Any new observation farther away than this from the spheroid center would be identified as an outlier.

If we identify just the support vectors of a data set and find their distance from the spheroid center, we can determine the radius of the spheroid and use this to score new observations. We break the equation above into three terms to be calculated.

$$Term\ 1 = k(x^*, x^*)$$

$$Term\ 2 = (-2)\sum_{i=1}^{n} \alpha_i k(x^*, x_j)$$

$$Term\ 3 = \sum_{i,j=1}^{n} \alpha_i \alpha_j k(x_i, x_j)$$

Term 1 is easily calculated using the kernel function passed to the `svdd()` function. Term 2 includes the α_i values, many of which are zero, so its computation can be simplified. Only the support vectors have a non-zero α_i, so we can reduce $(-2)\sum_{i=1}^{n} \alpha_i k(x^*, x_j)$ to $(-2)\sum_{i\in s} \alpha_i k(x^*, x_j)$, where s is the set of support vectors only. Term 3 can similarly be reduced. Also, note that while terms 1 and 2 differ for each data vector, term 3 is the same for every vector in a data set, so it only needs to be calculated once. For a data set with many support vectors, this may save considerable time.

There are two functions in the `kernlab` package that aid in the calculation of terms 1 and 2. The function `kernelMult()` calculates $\sum_{i=1}^{n} z_i k(x_i, x_j)$, so term 2 can be calculated by `(-2)*kernelMult(kernel, support_vectors, alphas)`. The function `kernelPol()` computes $z_i z_j k(x_i, x_j)$, so term 3 can be calculated by `sum(kernelPol(kernel, support_vectors, sv_alphas))`.

Once R^2 is found for each support vector, the set of these may be averaged to find a suitable estimate for the radius of the SVDD spheroid. The calculated R^2 should be the same for all support vectors, but imprecision in the quadratic solver may cause this not to be the case.

Once the radius of the SVDD spheroid is obtained, a new observation may be scored. A new data vector z is accepted when its distance from the center of the spheroid is less than the radius of the spheroid, that is

$$k(z, z) - 2\sum_{i=1}^{n} \alpha_i k(z, x_j) + \sum_{i,j=1}^{n} \alpha_i \alpha_j k(x_i, x_j)$$

This is calculated in the same way as the distances of the support vectors from the center. Note that “term 3” appears here as well and may be reused again, saving even more computation.

We use these results to define a `predict` method for the class `SVDD`. The function takes the argument `type`, which if set to `"boolean"` will cause the method to return a vector of 0’s and 1’s, indicating a new data vector has been accepted or rejected, respectively. If the `type` argument is set to `"radii"`, the method returns a list object containing the radius of the SVDD spheroid and the R^2 values for each new data vector.

```r
setMethod(
  f='predict',
  signature='SVDD',
  definition=function(object, newdata, type=c('radii', 'boolean')){
    # isolate non-zero alpha values (corresponding to support vectors)
    sv_alphas <- object@alphas[object@alphas > 0]
    # calculate k(x_s, x_s) for each support vector
    term1 <- apply(object@support_vectors, 1, function(x) object@kernel(x, x))
    # kernelMult computes (-2) sum(alpha_i k(x_s, x_i)) for each support vector x_s
    term2 <- (-2)*kernelMult(kernel=object@kernel, x=object@support_vectors, z=sv_alphas)
    # kernelPol computes z_i z_j k(x_i, x_j)
    term3 <- sum(kernelPol(kernel=object@kernel, x=object@support_vectors, z=sv_alphas))
    radii <- term1+term2+term3 # term1 and term2 vectors, term3 scalar
    r2 <- mean(radii)

    z1 <- apply(newdata, 1, function(x) object@kernel(x, x))
    z2 <- (-2)*kernelMult(kernel=object@kernel, x=newdata, y=object@support_vectors, z=sv_alphas)
    z_rad <- z1+z2+term3
    if(type=='radii'){
      return(list(mod_r2=r2, newdata_r2=c(z_rad)))
      }else{
        return(as.numeric(z_rad>r2))
        }
    })
```

We demonstrate the function below on the test data set.

```r
predict(euclidean_solution, newdata=as.matrix(mydat2), type='boolean')
```

```
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[36] 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
[71] 0 0 0 0 0 0
```

```r
predict(euclidean_solution, newdata=as.matrix(mydat2), type='radii')
```

```
$mod_r2
[1] 11.285

$newdata_r2
 [1]  9.755  6.465  5.655  7.965  7.235  8.675  7.245  7.805  0.845  0.235
[11]  0.965  0.705  0.185  1.285  0.375  1.435  0.125  1.635  0.795  0.455
[21]  0.815  1.755  2.245  0.595  0.355  0.565  0.595  0.125  1.905  0.885
[31]  0.835  1.455  0.955  0.225  0.515  0.675  0.675  0.215  1.705  0.375
[41]  0.195  0.235  0.315  1.695  0.205  3.785  3.515 13.035 13.965  2.125
[51]  5.805  2.075 12.035  1.805  5.075  6.715  1.495  1.615  4.305  5.615
[61]  7.845 11.755  4.515  1.885  3.335  9.565  5.035  3.425  1.395  4.315
[71]  5.325  3.915  2.315  6.405  6.035  3.825
```

c. Write an R function for detecting potential outliers for a new set of observations, along with the upper threshold.

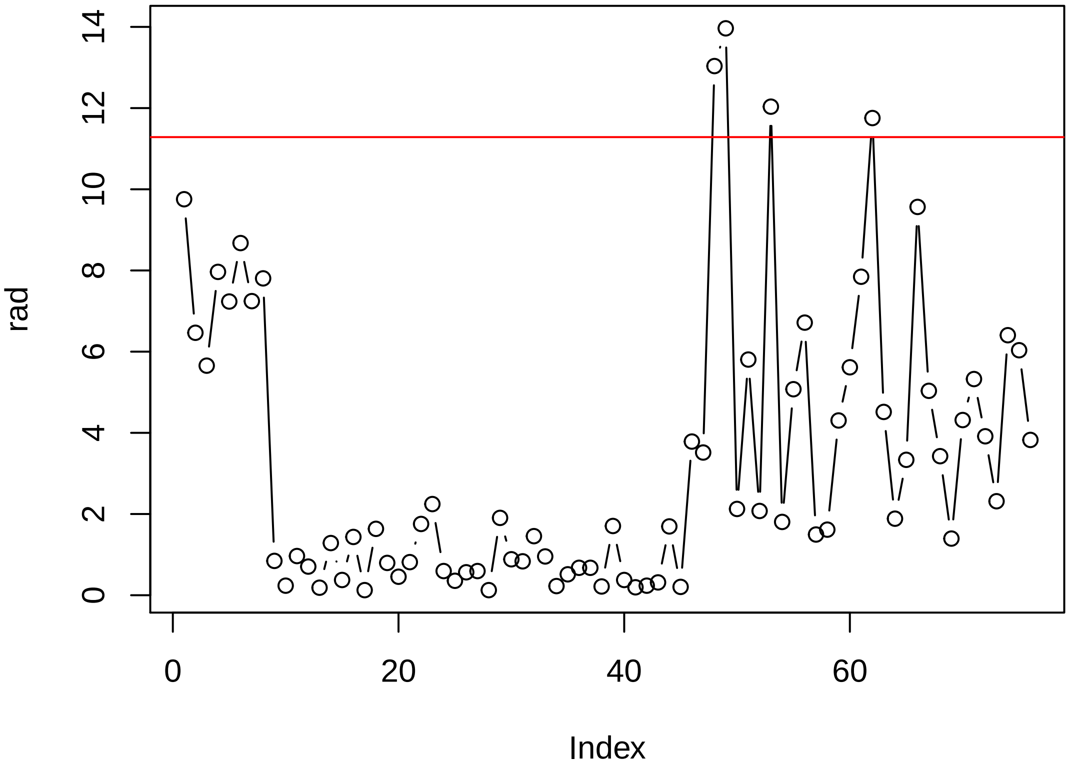
This task is trivial once the `predict` method above is defined. We call the `predict` method to return the R^2 for each

```r
detect_outliers <- function(SVDD_obj, newdata, plot=T){
  x_mat <- as.matrix(newdata) # in case newdata is a data.frame
  pred <- predict(SVDD_obj, newdata=x_mat, type='radii')
  rad <- pred$newdata_r2
  r2 <- pred$mod_r2
  outlier_ind <- rad>r2
  num_outliers <- sum(outlier_ind)

  if(plot==T){
    plot(rad, type='b')
    abline(h=r2, col='red')
  }

  msg <- paste(num_outliers, 'outliers out of', nrow(x_mat), 'observations detected. \n')
  cat(msg)
  return('outliers'=which(outlier_ind))
}

detect_outliers(SVDD_obj=euclidean_solution, newdata=mydat2)
```



```
4 outliers out of 76 observations detected.
```

```
[1] 48 49 53 62
```

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 $d_g(x)$

We will do you one better - we first define a class `malhalanobis` and then define a function to define a Mahalanobis kernel function given a covariance matrix. It's a function that defines a function that is of class `malhalanobis` (class definition not shown).

```
malhalanobis <- function(covmat){
  rval <- function(x, y = NULL) {
    if (!is(x, 'vector'))
      stop('x must be a vector')
    if (!is(y, 'vector') && !is.null(y))
      stop('y must be a vector')
    if (is(x, 'vector') && is.null(y)) {
      t(x)%*%solve(covmat)%*%x
      }
    if (is(x, 'vector') && is(y, 'vector')) {
      if (!length(x) == length(y))
        stop('number of dimension must be the same on both data points')
      t(x)%*%solve(covmat)%*%y
      }
    }
  return(new('malhalanobis', .Data=rval, kpar=list(covmat=covmat)))
  }
```

We test the function by passing it the covariance matrix of our training data set and then an arbitrary point.

```
my_mal_kernel <- malhalanobis(covmat=cov(mydat))
class(my_mal_kernel)
```

```
[1] "malhalanobis"
attr(,"package")
[1] ".GlobalEnv"
```

```
my_mal_kernel(c(1, 1, 1, 1), c(1, 1, 1, 1)) # test case
```

```
         [,1]
[1,] 17.01475
```

b. Write an R function to perform the Mahalanobis kernel SVDD.

The flexibility of our earlier code pays off here. The function `svdd()` defined earlier takes as an argument `kernel`, so we pass it our kernel function `my_mal_kernel` and our data set `mydat` and reuse the function.

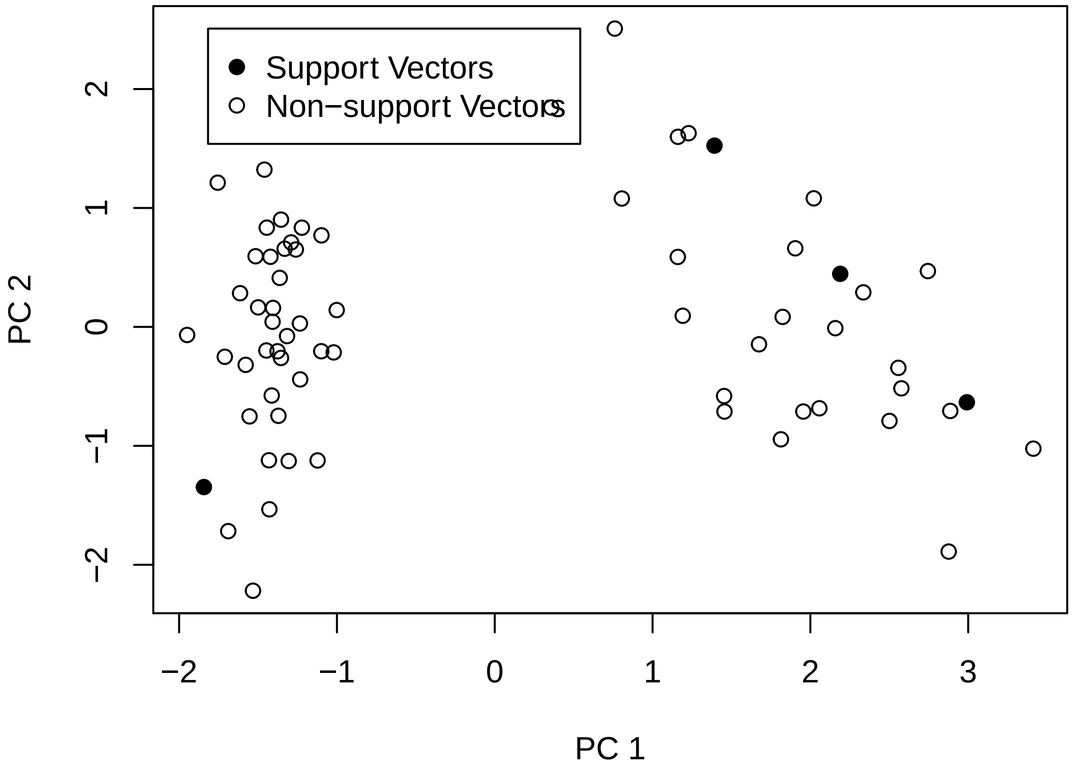
```
mal_solution <-  svdd(as.matrix(mydat), k=my_mal_kernel)
summary(mal_solution)
```

```
SVDD object using kernel function malhalanobis:
4 support vectors identified out of 66 observations.
```

```
      V1  V2  V3  V4
[1,] 5.2 4.1 1.5 0.1
```

```
[2,] 6.0 2.2 4.0 1.0
[3,] 7.3 2.9 6.3 1.8
[4,] 5.8 2.8 5.1 2.4
```

```r
plot(mal_solution)
```



c. Write an R function to perform the prediction of a new observation using the Mahalanobis kernel SVDD.

Again, our flexible code earlier pays off. We can reuse our `predict` method for any `SVDD` object.

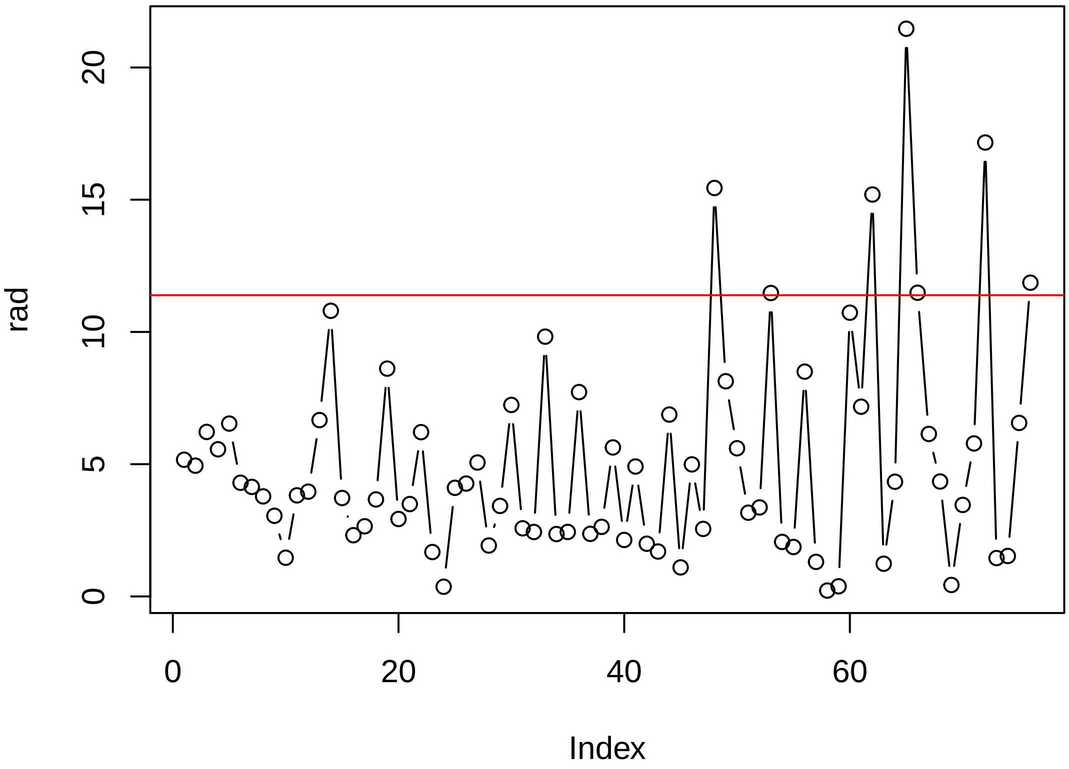
```r
predict(mal_solution, newdata=as.matrix(mydat2), type='boolean')
```

```
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
[36] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0
[71] 0 1 0 0 0 1
```

d. Write an R function for detecting potential outliers for a new set of observations, along with the upper threshold.

We reuse our function from earlier again.

```r
detect_outliers(SVDD_obj=mal_solution, newdata=mydat2)
```



```
7 outliers out of 76 observations detected.
```

```
[1] 48 53 62 65 66 72 76
```

Appendix with R code