Note: In this project, I solved everything with R but for some problems like problem 1, part 1 & 3, I also used Python. Here, R codes begin with '>' symbol and Python codes begin with '»' symbol.

Problem 1:

We want to solve the following LS-SVM problem in R or Python or both using the dataset "pb2.txt":

Part 1:

Use the large scale algorithm (Hestenes-Stiefel algorithm) discussed in Suykens et al. (1999), available on class webcourse, to solve the LS- SVM problem. Provide the training and prediction codes.

Solution:

The standard SVM is solved using quadratic programming methods. On the other hand, a least squares version (LS-SVM) expresses the training in terms of solving a set of linear equations instead of quadratic programming.

Algorithm of LS-SVM is quite like standard SVM, so I wouldn't talk much theory of LS-SVM. My main focus here is to solve the following matrix using conjugate gradient algorithm for $\alpha \; and$ b.

$$\begin{bmatrix} 0 & Y^T \\ Y & ZZ^T + \gamma^{-1} I \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ \vec{1} \end{bmatrix} \quad (1)$$

where, Z = $[\varphi(x_1)y_1; \ldots; \varphi(x_N)y_N], Y = (y_1, \ldots, y_N), \vec{1} = [1,1, \ldots,1], \; \alpha = (\alpha_1, \ldots, \alpha_N)$

The matrix in (1) is dimension (N+1)×(N+1). For large value values of N this matrix cannot be stored, such that an iterative method for solving (1) is needed. The matrix in (1) can be rewrite as,

$$\begin{bmatrix} 0 & Y^T \\ Y & H \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} \quad (2)$$

where, H = $ZZ^T + \gamma^{-1} I, \; d_1 = 0, and \; d_2 = \vec{1}$

A Hestenes-Stiefel conjugate gradient algorithm for solving this problem is used; is given bellow.

Conjugate Gradient Method

i = 0; $x_0 = 0$; $r_0 = B$;
while $r_i \neq 0$
i = i+1
if i=1
$p_1 = r_0$
else
$B_i = \frac{{r_{i-1}}^T r_{i-1}}{{r_{i-2}}^T r_{i-2}}$ (3)
$p_i = r_{i-1} + B_i p_{i-1}$
end
$\lambda_i = \frac{{r_{i-1}}^T r_{i-1}}{{p_i}^T A p_i}$
$x_i = x_{i-1}\lambda_i p_i$
$r_i = r_{i-1} - \lambda_i p_i$
end
x = x_i

Solving LS-SVM using R:

Required packages:

'caTools' is used to split the data into training and test data.

```
> require(caTools)
```

Steps to solve LS-SVM problem:

i) Imported the data, recoded Y (response variable) as (1,-1) and then divided the Boston data into training and test data sets.
ii) Created a function called 'LSSVMtrain' to solve LS-SVM.
iii) Created a kernel function; radial basis function kernel(Gaussian) or RBF kernel. And, using that function created the H matrix.
iv) Created a function called ConjugateGrad that contains Conjugate Gradient Algorithm in (3). (I did this in both R and Python)
v) Calculated s to calculate α, and b. Then found solutions for α and b.
vi) Filtering α on the constraint.
vii) Defined a new function to predict Y's. And, calculated Error rate.

Step 1:

Imported and then recoded Y as (1,-1)

```
> data <- read.table("/Users/mdjibanulhaquejiban/Downloads/pb2.txt")
> require(caTools)
> n=nrow(data)
> for (i in 1:n){
> if (data[i,1] > 1){
> data[i,1]<--1
> }
> }
```

Here, divided the data into training (75%) and test data (25%) as follows

```
> sample = sample.split(data[,1], SplitRatio = .75)
> train_data = subset(data, sample == TRUE)
> test_data  = subset(data, sample == FALSE)
```

Separating the training data set, here Y is training output response and X is the predictors.

```
> Y <- train_data[,1]
> X <- as.matrix(train_data[,2:5],ncol=4)
> N <- length(Y)
```

Step 2:

Created a function called 'LSSVMtrain to solve the LS-SVM problem. Here, X contains predictors and Y is output response from training data.

```
> LSSVMtrain <- function(X,Y,C=Inf, gamma=2,esp=1e-10){
```

Step 3:

Then, I created the RBF kernel function;

$K(\mathbf{x},\mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}}$, where $\sigma^2 = 1$

```
> rbf_kernel <- function(x1,x2,gamma){
> K<-exp(-(1/gamma^2)*t(x1-x2)%*%(x1-x2))
> return(K)
> }
```

Now, using this kernel function, I created the matrix $H_{(n\times n)}(\mathbf{x}, \mathbf{x}')$.

```
> H_matrix <- function(X,Y,C=Inf, gamma=1.5,esp=1e-10){
> N<-length(Y)
> H<-matrix(0,N,N) # Created an empty H matrix
> for(i in 1:N){
> for(j in 1:N){
        H[i,j]<-Y[i]*Y[j]*rbf_kernel(X[i,],X[j,],gamma)
> }
> }
```

Adding a very small number to the diagonal.

```
> H <- H+diag(N)*1e-12
> }
> H <- H_matrix(X,Y,C=12,gamma=13)
```

The output matrix looks like the following matrix

$$H_{(n\times n)} = \begin{bmatrix} H(x_1,x_1) & H(x_1,x_2) & \dots & H(x_1,x_n) \\ H(x_2,x_1) & H(x_2,x_2) & \dots & H(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ H(x_n,x_1) & H(x_n,x_2) & \dots & H(x_n,x_n) \end{bmatrix}$$

Step 4: (in R)

Created a function called ConjugateGrad that contains Conjugate Gradient Algorithm in (3).

The following codes are for solving Ax = B with A $\in \mathbb{R}^{n\times n}$ symmetric positive definite and B$\in$ $\mathbb{R}^n$.

Note: For quick understanding, I wrote the equations I coded just before each block of codes.

```
> ConjugateGrad <- function(A = NULL, B = NULL, x0 = NULL, i = 1500,
  thresh = 0.00001){

> r0 <- B
> p_1<- r0

> for(i in 1:i){
```

Coded these equations $\lambda_i = \frac{{r_{i-1}}^T r_{i-1}}{{p_i}^T A p_i}$; $x_i = x_{i-1}\lambda_i p_i$; $r_i = r_{i-1} - \lambda_i p_i$ in R as

```
> lambda_i <- sum(r0*r0) / as.numeric((t(p_1) %*% A) %*% p_1)
> x_i <- x0 + (lambda_i * p_1)
> r_i <- r0 - lambda_i *  A %*% p_1
```

Then coded these $B_i = \frac{{r_{i-1}}^T r_{i-1}}{{r_{i-2}}^T r_{i-2}}$; $p_i = r_{i-1} + B_i p_{i-1}$ in R as

```
> beta_i <- sum(r_i * r_i) / sum(r0*r0)
> p_i <- t(r_i) + beta_i %*% p_1

> x0 <- x_i
> p_1 <- as.numeric(p_i)
> r0 <- r_i
```

The following code is to satisfy this condition while $r_i \neq 0$

```
> if(norm(r0) <= thresh){
> break
> }

> }                 # end of for loop

> return(x_i)       # Stored x_i

> }                 # end of ConjugateGrad function
```

Now, define $d_2 = \vec{1}$ and $x_0 = \vec{0}$ both are $n \times 1$ vectors.

```
> d2 <- rep(1,N)
> x0 <- rep(0, N)
```

Now, solving η from Hη = Y as follows

```
> eta <- ConjugateGrad(H, Y, x0)     # It took 949 iterations to get η
```

and ν from Hν = d_2

```
> v <- ConjugateGrad(H, d2, x0)      # It took 1087 iterations to get ν
```

Step 4: (in Python)

```
» import scipy as sp
» from scipy.linalg import norm

» def ConjugateGrad(A,B,x0,i,thresh):
» r0 = B
» p_1 = r0
» x_i = x0
» k=1
```

Coded these equations $\lambda_i = \frac{{r_{i-1}}^T r_{i-1}}{{p_i}^T A p_i}$; $x_i = x_{i-1}\lambda_i p_i$; $r_i = r_{i-1} - \lambda_i p_i$ in Python as

```
» while(k<i):
» lambda_i = sp.dot(r0,r0) / sp.dot(sp.dot(p_1,A),p_1)
» x_i = x0 + sp.dot(lambda_i,p_1)
» r_i = r0 - sp.dot(sp.dot(lambda_i,A),p_1)
```

Then coded these $B_i = \frac{{r_{i-1}}^T r_{i-1}}{{r_{i-2}}^T r_{i-2}}$; $p_i = r_{i-1} + B_i p_{i-1}$ in Python as

```
» beta_i = sp.dot(r_i,r_i) / sp.dot(r0,r0)
» p_i = r_i + sp.dot(beta_i,p_1)
```

The following code is to satisfy this condition while $r_i \neq 0$

```
» if(norm(r_i) <= thresh):
        break

» x0 = x_i
» p_1 = p_i
» r0 = r_i

» k+=1

» return x_i             # Stored x_i

» x = ConjugateGrad(A,B,x0,1500,1e-9)

» print("ConjugateGrad = {} ".format(x.round(2)))
```

Step 5:

Computing s = $Y^T \eta$ as follows

```
> s <- sum(t(Y)*eta)
```

- Now, finding solution for b and α

Calculated b = $\eta^T d_2 / s$ as follows

```
> b <- sum(t(eta)*d2)/s
```

Calculated $\alpha = \nu$ - η b as follows

```
alpha_org <- v - eta*b
```

Step 6:

The Lagrange multipliers α_i can be either positive or negative due to the equality constraints. So, I am deleting zero values in α_i, if there is any.

The following codes are from SVM, I didn't change anything that's why I am not explaining these much.
Here, 'alpha' is our final vector of α.

```
> Indx <- which(alpha_org != 0,arr.ind=TRUE)
> Alpha <- alpha_org[indx]
> nSV <- length(indx)

> if(nSV==0){
> throw("no solution of alpha for these data points")
> }

> Xv <- X[indx,]
> Yv <- Y[indx]
> Yv <- as.vector(Yv)


> list(alpha=alpha, b=b, nSV=nSV, Xv=Xv, Yv=Yv, gamma=gamma)

> }          # end of LSSVMtrain function
```

Now, calling LS-SVM training model.

```
> LSSVM_Train_model <- LSSVMtrain(X,Y,C=12,gamma=2)
```

Output: b = -1.941031 and α vector is

```
> LSSVM_Train_model$alpha
 [1]   -50.9597968   520.7919430 -2106.5705242   239.7954893    71.0867174   841.9042465    21.7807448
 [8]  -161.7258598    91.4470326   125.9157088   843.5865754 -2383.1509936   -54.6683791   -26.4570378
[15]  -811.6867324     0.7379416  1174.4087059     1.9589955   -21.1197876   235.5798576  3288.5731232
[22]  -211.4087151   537.4617162    61.5204875   -34.6954298  -678.0405339   112.2763908    33.1003806
[29]   -28.6322249   154.9634276  -162.1186518    14.8056271    74.7225650   656.7387436   -24.3298304
[36]    -7.7518572  4136.5647902 -4378.6027535   715.2002141   575.4156283  1115.2454201  -289.7419943
[43]    91.8011277    48.2674319    -8.3745542   -11.0534368
```

Step 7:

Here, I defined a new function called 'LSSVMpredict' to predict Y's.

At first stored X and Y in Test_X and Test_Y from test data I created at the beginning.

```
>  Test_Y <- test_data[,1]
>  Test_X <- as.matrix(test_data[,2:5],ncol=4)
>  n <- nrow(Test_X)
```

Here is the prediction function, 'LSSVMpredict', where we used all the parameters we got from our training model.

```
>  LSSVMpredict <- function(x,model){
>  alpha<-model$alpha
>  b<-model$b
>  Yv<-model$Yv
>  Xv<-model$Xv
>  nSV<-model$nSV
>  gamma<-model$gamma
>  ayK <- numeric(nSV)
```

Used ayK = $\sum_{i \in nSV} y_i \alpha_i^* K(\boldsymbol{x_i'}, \boldsymbol{x})$ to find $\langle w, x \rangle$ in the model.

```
>  Y_hat <- numeric(n)          # defined an empty Ŷ vector
>  for (m in 1:n){
>  for(i in 1:nSV){
>  ayK[i]<- alpha[i]*Yv[i]*rbf_kernel(Xv[i,],x[m,],gamma)
>  }
```

Calculated $\hat{Y} = sign(\langle w, x \rangle + b)$ as follows,

```
>  Y_hat[m] <- sign(sum(ayK)+b)
>  }
>  return(Y_hat)                # Ŷ vector
>  }                        # end of prediction model
```

Calling the prediction function as

```
> Pred_LSSVM <- LSSVMpredict(Test_X, LSSVM_Train_model); Pred_LSSVM
```

Error rate:

```
> table(Pred_LSSVM, Test_Y)
```

Output:

		Test_Y	
		-1	1
Predict_Y	-1	6	3
	1	2	5

```
> mean(Pred_LSSVM != Test_Y)
```

Output: 0.3125

Part 2:

Write a code for updating the QR decomposition after adding one row to the training set. (qr.update on class webcourse).

Solution:

Given A = QR $\in \mathbb{R}^{m\times n}$, with m ≥ n, algorithm 2.6 (from QR update, S. Hammarling and C. Lucas) computes $\widetilde{Q^T}\tilde{A} = \tilde{R} \in \mathbb{R}^{(m+1)\times n}$, where $\tilde{R}$ is upper trapezoidal, $\tilde{Q}$ is orthogonal and $\tilde{A}$ is A with a new row, $u^T \in \mathbb{R}^n$, inserted in the kth position.

Step1: Followed Step 1 to Step 3 from previous question to get H matrix. Where, H = K + $C^{-1}I$

$$H_{(n\times n)} = \begin{bmatrix} H(x_1,x_1) & H(x_1,x_2) & \dots & H(x_1,x_n) \\ H(x_2,x_1) & H(x_2,x_2) & \dots & H(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ H(x_n,x_1) & H(x_n,x_2) & \dots & H(x_n,x_n) \end{bmatrix}$$

Step2: In this step, I Generated K matrix as follows,

$$\text{A} = K_1 = \begin{bmatrix} 0 & \vec{1} \\ \vec{1} & \text{K} + C^{-1}I \end{bmatrix} = \begin{bmatrix} 0 & \vec{1} \\ \vec{1} & \text{H} \end{bmatrix}$$

Adding row = $(0, \vec{1})$ and column = $\vec{1}$ to H matrix to get K_1 matrix.

```
> row=c(0,rep(1,N))
> col = rep(1,N)
> K1 = cbind(col,H)
> K1 = as.matrix(rbind(row,K1))
```

Step 3:

New row which is needed to be added in the data.

```
> x_raw = c(15, 21, 26, 21)
> y_raw = 1
```

Calculating u_i using kernel function

```
> Ur<-numeric(0)
> N<-length(Y)

> for(j in 1:N){
> Ur[j]<-y_raw*Y[j]*rbf_kernel(x_raw,X[j,],gamma)
> }
```

Now, Decomposing K_1 matrix using into QR that we need in algorithm (2.6).

```
> Q = qr.Q(qr(K1))
> R = qr.R(qr(K1))
```

Step 4:

In algorithm (2.6) we need a function calls givens, defining this first.

The following function 'givens' is defined using the algorithm (1.1) (from QR update, S. Hammarling and C. Lucas)

This following function returns scalars c and s such that

$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} d \\ 0 \end{bmatrix}$, where a, b and d are scalars, and $s^2 + c^2 = 1$

```
> givens <- function(a,b) {
> if (b == 0){
> c=1
> s=0
> } else if (abs(b) >= abs(a)) {
> t = -a/b
> s = 1/sqrt(1+t**2)
> c = s*t
> } else {
> t = -b/a
> c = 1/sqrt(1+t**2)
> s = c*t
> }
> list(c=c,s=s)
> }
```

R codes for algorithm (2.6) to update R:

Defining two empty vectors called c and s

```
> c = numeric(0)
> s = numeric(0)
```

Here, generated d = $Q^T b$ as follows,

```
> d = t(Q1)%*%b
> mu = y_raw


> for (j in 1:length(Ur)){
> c[j] = givens(R[j,j], Ur[j])$c
> s[j] = givens(R[j,j], Ur[j])$s
> R[j,j] = c[j]*R[j,j] - s[j]*Ur[j]
```

Updated the jth row of R and μ

```
> i = 1
> while(i != length(Ur)){
> i = i+1
> t1 = R[j,i]
> t2 = Ur[i]
> R[j, i] = c[j]*t1 - s[j]*t2
> Ur[i] = s[j]*t1 + c[j]*t2
> }
```

Update jth row of d and μ

```
> t1 = d[j]
> t2 = mu
> d[j] = c[j]*t1 - s[j]*t2
> mu = as.numeric(s[j]*t1 - c[j]*t2)

> }                # End of For Loop
```

Here, $\tilde{R}$ is the updated form of R when we add a row; $\tilde{R} = \begin{bmatrix} R \\ 0 \end{bmatrix}$. and update $\tilde{d}$ as $\tilde{d} = \begin{bmatrix} d \\ \mu \end{bmatrix}$.

```
> R_tilda = rbind(R, 0)
> D_tilda = c(d, mu)
```

Step 5:

In this step, I am going to update Q.

At first, I added a row and a column to the origina Q I have as

$$\tilde{Q} = \begin{bmatrix} Q & \vec{0} \\ \vec{0} & 1 \end{bmatrix}$$

The following codes added the new raw and column to Q to get $\tilde{Q}$,

```
> temp_row <- rep(0,ncol(Q))
> temp_col <- c(temp_row,1)
> Q1 <- rbind(Q,temp_row)
> Q2 <- cbind(Q1,temp_col)
```

Now, I used algorithm (2.7) (from QR update, S. Hammarling and C. Lucas) to update Q to $\tilde{Q}$

Calculating the number of column Q.

```
> m=ncol(Q)
```

Defining a for loop that updates Q to $\tilde{Q}$, [algorithm 2.7]

```
> for (j in 1:length(Ur)){
> t1 = Q2[1:m+1,j]
> t2 = Q2[1:m,m+1]
> Q2[1:m+1,j] = c[j] * t1 - s[j] *t2
> Q2[1:m,m+1] = s[j] * t1 - c[j] * t2
> }
```

Note: This is not the complete solution for this part of this problem. $\tilde{R}$ and $\tilde{Q}$ both are needed to be updated again by adding one column algorithms (algorithm 2.19 and 2.20, respectively). Due to time shortage, I couldn't finish the rest. However, I am going to work on this.

Part 3:

Our next challenge is to solve the LSSVM using matrix inversion and apply it to incremental/decremental LSSVM. Use the code from ques- tion 2 on QR update to solve the incremental LSSVM problem. (increment.lssvm on class webcourse.)

Solution:

Note: I couldn't solve the previous question completely, but I know how to solve this part. Assuming that, I finished previous question completely and got my updated Q^* and R^*. I continued to this one. Here I used Q and R; these are calculated by decomposing H matrix from part 1 in this problem.

Our purpose here is to solve $\boldsymbol{x}^*$ in the following equation,

$$R^* x^* = (Q^*)^T \begin{bmatrix} 0 \\ y^* \end{bmatrix}$$

or, $R^* x^* = \text{b}$ (1)

where, y^* is the updated y vector and the matrices Q^* and R^* are obtained by updating QR decomposition in previous part of this problem, and $\text{b} = (Q^*)^T \begin{bmatrix} 0 \\ y^* \end{bmatrix}$.

Here,

$$R^* = \begin{bmatrix} R_{11} & R_{12} & \dots & R_{1(n+1)} \\ 0 & R_{22} & \dots & R_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & R_{(n+1)(n+1)} \end{bmatrix};\ x^* = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n+1} \end{bmatrix};\ Q^* = \begin{bmatrix} Q_{11} & Q_{12} & \dots & Q_{1(n+1)} \\ Q_{21} & Q_{22} & \dots & Q_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{(n+1)1} & Q_{(n+1)2} & \dots & Q_{(n+1)(n+1)} \end{bmatrix};$$

$$y = \begin{bmatrix} -u_{m1} \\ u_{m2} \end{bmatrix};\ \text{and}\ b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n+1} \end{bmatrix}$$

we can write (1) as

$$\begin{bmatrix} R_{11} & R_{12} & \dots & R_{1(n+1)} \\ 0 & R_{22} & \dots & R_{2(n+1)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & R_{(n+1)(n+1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n+1} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n+1} \end{bmatrix}$$

Algorithm to solve for $\boldsymbol{x}^*$:

1) x_{n+1} can be found directly as $x_{n+1} = b_{n+1}/R_{(n+1)(n+1)}$
2) Now, solving for $x_n = (b_n - R_{(n)(n+1)}x_{n+1})/R_{nn}$. We can just substitute x_{n+1} in this equation to get x_n.
3) We can repeat step 2 until we find x_1.

R code to solve this problem for $\boldsymbol{x}^*$:

I got this H from part 1 in this problem. Then decomposed into Q and R using 'qr' statement in R.

```
> H <- LSSVM_Train_model$H
> R <- qr.R(qr(H))
> Q <- qr.Q(qr(H))
```

Created b = $(Q^*)^T \begin{bmatrix} 0 \\ y^* \end{bmatrix}$

```
> b <- t(Q)%*%y
```

Defined a function called 'incremental' to find $\boldsymbol{x}^*$

```
> incremental=function(R,b)
> {
> p= nrow(R)                    # number of rows in R matrix
> b=as.matrix(b)

> for (j in seq(p,1,-1))
> {
> b[j,1]=b[j,1]/R[j,j]              # This is step 1 of the algorithm
  I mentioned earlier
> if((j-1)>0)
> b[1:(j-1),1]=b[1:(j-1),1]-(b[j,1]*R[1:(j-1),j])          # This is
  step 2
> }
> return(b)                     # getting x*
> }

> x <- incremental(A,b); x
```

Python code to solve this problem for x^*:

```
» import numpy as np
```

Defining a function called 'incremental' as

```
» def incremental(A, b):
» n = np.size(b)
» x = np.zeros_like(b)

» x[-1] = 1. / A[-1, -1] * b[-1]                                  # Step 1 in
  algorithm
» for i in xrange(n-2, -1, -1):
» x[i] = 1. / A[i, i] * (b[i] - np.sum(A[i, i+1:] * x[i+1:])) # Step 2 in
                                                                   algorithm

» return x               # storing x*
```

Calling the function to get x^*

```
» print("incremental = {} ")
```

Output:
Both outputs are same so I am only proving part of one because the whole output is very large,

```
> incremental(A,b)
              [,1]
 [1,]  1.06236417
 [2,] -1.20640982
 [3,]  0.31833706
 [4,]  2.83268275
 [5,]  0.49316744
 [6,] -2.08213613
 [7,] -0.57422204
 [8,]  1.67484510
 [9,] -1.22153611
[10,] -0.59881069
[11,] -1.43334965
[12,] -0.04744487
[13,] -0.68665418
[14,] -0.07880984
[15,]  0.25398672
[16,] -0.51519100
[17,]  1.03460803
[18,]  0.39733552
[19,] -0.72409676
[20,]  0.59930313
[21,]  1.18113870
[22,]  0.38779223
[23,]  0.40417719
[24,] -0.53872667
[25,]  1.08182775
[26,] -0.20283716
[27,] -0.40320001
[28,] -0.19902526
[29,] -0.05342628
[30,]  1.25145692
[31,]  0.19603629
```

Problem 2:

Write a code similar to the SVM discussed in class ("svmtrain.r" and "svmpredict.r") to solve the SVR problem. The dataset to be used is the Boston Housing Dataset.

Solution:

i) Theory and concept of Support Vector Regression (SVR):

Define, our training data $\{(x_1, y_1), (x_2, y_2), \dots \dots, (x_n, y_n)\} \subset \boldsymbol{\chi}$ x $\mathbb{R}$, where $\boldsymbol{\chi}$ denotes the space of the input patterns (e.g. $\boldsymbol{\chi} = \mathbb{R}^n$). In ε-SV regression our goal is to find a function $f(x)$ that has at most ε deviation from the actually obtained targets y_i for all the training data, and at the same time is as flat as possible.

The function is defined as,

$$f(x) = \langle w, x \rangle + b \quad with\, w \in \boldsymbol{\chi}, b \in \mathbb{R} \tag{1}$$

Flatness in the case of (1) means that one seeks a small w. We can write this problem as a convex optimization problem:

minimize $\frac{1}{2} ||w||^2$

subject to constraints, $\begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon \end{cases}$ (2)

The above convex optimization problem is feasible in cases where *f* actually exists and approximates all pairs (x_i, y_i) with ε precision. Sometimes, some errors are allowed. Introducing slack variables ξ_i, ξ_i^* to cope with otherwise infeasible constraints of the optimization problem (2), the formulation becomes

Minimize $\frac{1}{2} ||w||^2 + C \sum_{i=1}^{n} (\xi_i + \xi_i^*)$

subject to constraints, $\begin{cases} y_i - \langle w, x_i \rangle - b \leq \varepsilon + \xi_i \\ \langle w, x_i \rangle + b - y_i \leq \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases}$ (3)

The constant C > 0 determines the tradeoff between the flatness of *f* and the amount up to which deviations larger than ε are tolerated. ε-intensive loss function has been described as follows:

$$|\xi|_\varepsilon = \begin{cases} 0 & if\ |\xi| - \varepsilon \\ |\xi| - \varepsilon & \text{otherwise} \end{cases} \tag{4}$$

The following figure depicts the situation graphically.

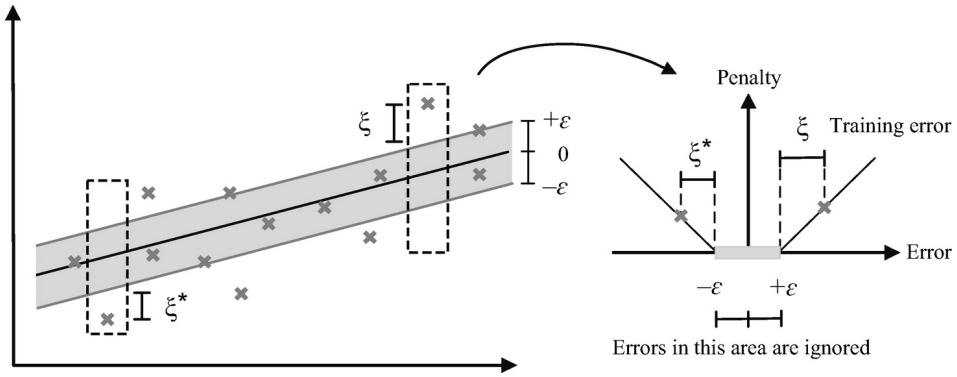

Figure: *The soft margin loss setting for a linear SVM (from Schölkopf and Smola, 2002)*

The dual formulation provides the key for extending SV machine to nonlinear functions. The standard dualization method utilizing Lagrange multipliers has been described as follows:

$$L = \frac{1}{2}\left\|w\right\|^2 + C\sum_{i=1}^{n}(\xi_i + \xi_i^*) - \sum_{i=1}^{n}\alpha_i(\varepsilon + \xi_i - y_i + \langle w, x_i\rangle + b) - \sum_{i=1}^{n}\alpha_i^*(\varepsilon + \xi_i^* + y_i - \langle w, x_i\rangle - b) - \sum_{i=1}^{n}(\eta_i\xi_i + \eta_i^*\xi_i^*) \quad (5)$$

Here, L is the Lagrangian and the Lagrange multipliers/dual variables in (5) must satisfy positivity constraints, i.e.

$$\alpha_i, \alpha_i^*, \eta_i, \eta_i^* \geq 0 \quad (6)$$

It follows from the saddle point condition that the partial derivatives of L with respect to the primal variables (w,b, ξ_i, ξ_i^*) have to vanish for optimality.

$$\partial_b L = \sum_{i=1}^{n}(\alpha_i^* - \alpha_i) = 0 \quad (7)$$

$$\partial_w L = w - \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i = 0 \quad (8)$$

$$\partial_{\xi_i} L = C - \alpha_i - \eta_i = 0 \quad (9)$$

$$\partial_{\xi_i^*} L = C - \alpha_i^* - \eta_i^* = 0 \quad (10)$$

From equation (8), we get,

$$w = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i \quad (11)$$

Therefore, $f(x) = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)x_i\, x + b$

$$= \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)\, K(x_i, x_j) + b \quad (12)$$

Substituting (7),(8),(9), and (10) into (5) yields the dual optimization problem,

$$\text{L} = \text{maximize}\left\{-\frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)\langle x_i,x_j\rangle-\varepsilon\sum_{i=1}^{n}(\alpha_i+\alpha_i^*)+\sum_{i=1}^{n}y_i(\alpha_i-\alpha_i^*)\right.$$

$$=\text{maximize}\left\{-\frac{1}{2}\sum_{i,j=1}^{n}(\alpha_i-\alpha_i^*)(\alpha_j-\alpha_j^*)\langle x_i,x_j\rangle-\sum_{i=1}^{n}(\varepsilon-y_i)\alpha_i-\sum_{i=1}^{n}(\varepsilon+y_i)\alpha_j^*\right.$$

Subject to constraints

$$\sum_{i,j=1}^{n}(\alpha_i-\alpha_i^*)=0,$$

$$0\le\alpha_i\le C\quad i=1,2,\dots,n, \tag{13}$$

$$0\le\alpha_i^*\le C\quad i=1,2,\dots,n.$$

There are 2n unknown dual variables (n α_i-s and n α_i^*-s) for a linear regression and the Hessian Matrix H of the quadratic optimization problem in the case of regression is a (2n,2n) matrix. The above problem can be expressed in the standard quadratic optimization problem in a matrix notation/form as follows:

$$\text{Min L}=\frac{1}{2}\alpha'T'KT\alpha+d'\alpha$$

$$=\frac{1}{2}\alpha'H\alpha+d'\alpha \tag{14}$$

where,

$$\alpha_{(2n\times1)}=\begin{bmatrix}\alpha_1\\ \alpha_2\\ \vdots\\ \alpha_n\\ \alpha_1^*\\ \alpha_2^*\\ \vdots\\ \alpha_n^*\end{bmatrix},\ \text{T}_{(2n\times2n)}=\begin{bmatrix}1&0&\dots&0&-1&0&\dots&0\\ 0&1&\dots&0&0&-1&\dots&0\\ \vdots&\vdots&\ddots&\vdots&\vdots&\vdots&\ddots&\vdots\\ 0&0&\dots&1&0&0&\dots&-1\\ 1&0&\dots&0&-1&0&\dots&0\\ 0&1&\dots&0&0&-1&\dots&0\\ \vdots&\vdots&\ddots&\vdots&\vdots&\vdots&\ddots&\vdots\\ 0&0&\dots&1&0&0&\dots&-1\end{bmatrix},$$

$$K_{(n\times n)}=\begin{bmatrix}K(x_1,x_1)&K(x_1,x_2)&\dots&K(x_1,x_n)\\ K(x_2,x_1)&K(x_2,x_2)&\dots&K(x_2,x_n)\\ \vdots&\vdots&\ddots&\vdots\\ K(x_n,x_1)&K(x_n,x_2)&\dots&K(x_n,x_n)\end{bmatrix},$$

$d'_{(1\times2n)}=[(\varepsilon-y_1),(\varepsilon-y_2),\dots,(\varepsilon-y_n),(\varepsilon+y_1),(\varepsilon+y_2),\dots,(\varepsilon+y_n)]$, and

$$\mathbf{H}_{(2n\times 2n)} = T' \begin{bmatrix} K & -K \\ -K & K \end{bmatrix} T,$$

where K is an (n×n) kernel matrix, and T is a $(2n\times 2n)$ $matrix$ defined above.

subject to constrains in (13) can be written in this form,

$$A'b \geq b_0$$

where, $A'_{((4n+1)\times 2n)} = \begin{bmatrix} 1 & 1 & \dots & 1 & -1 & -1 & \dots & -1 \\ 1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & 1 \\ -1 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & -1 & \dots & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & -1 & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & -1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & -1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 0 & 0 & \dots & -1 \end{bmatrix}$,

$b_{(2n\times 1)} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \\ \alpha_1^* \\ \alpha_2^* \\ \vdots \\ \alpha_n^* \end{bmatrix}$, and $b_{0_{((4n+1)\times 1)}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ -C \\ -C \\ \vdots \\ -C \\ -C \\ -C \\ \vdots \\ -C \end{bmatrix}$

Now we can solve the Lagrangian equation (14) for α in R using quadratic programing package (‘quadprog’).

ii) Solving SVR using R:

Required packages:

'MASS' package is used to get the 'Boston' data file, 'caTools' is used to split the data into training and test data, and 'quadprog' package is used to solve the quadratic problem in SVR.

```
> library(MASS)
> require(caTools)
> library(quadprog)
```

Steps to solve SVR problem:

i) Divided the Boston data into training and test data sets.
ii) Created a function called 'SVRtrain' to solve SVR.
iii) Created a kernel function; radial basis function kernel(Gaussian) or RBF kernel. And, using that function created the kernel matrix.
iv) Created the Hessian Matrix $\mathbf{H}_{(2n\times 2n)}$ (dm).
v) Created all other necessary matrices, vectors and parameters to solve the quadratic problem in R.
vi) Got $\alpha_i \; and \; \alpha_i^*$ using the constraint $(\alpha_i \times \alpha_i^*) > \varepsilon$.
vii) Calculated constant b. End of training model function.
viii) Defined a new function to predict Y's. And, calculated MSE.

Step 1:

In the following codes, I divided the data into training and test data.

```
> set.seed(351969)
> sample = sample.split(Boston[,14], SplitRatio = .75)
> train_data = subset(Boston, sample == TRUE)
> test_data  = subset(Boston, sample == FALSE)
```

Now, I separated the output response 'housing value' (stored in Y) and predictors (stored in X) the following codes, I divided the data into training and test data.

```
> Y <- train_data[,14]
> X <- as.matrix(train_data[,1:13],ncol=13)
> N <- length(Y)          # N is sample size
```

Step 2:

Created a function called 'SVRtrain' to solve the SVR problem. Here, X is predictors and Y is output response from training data and $\varepsilon = e^{-10}$.

```
> SVRtrain <- function(X,Y,C=Inf, gamma=2,esp=1e-10){
```

Step 3:

Then, I created the RBF kernel function;

$K(\mathbf{x},\mathbf{x}') = e^{-\frac{\|\mathbf{x}-\mathbf{x}'\|^2}{2\sigma^2}}$, where $\sigma^2 = 1$

```
> rbf_kernel <- function(x1,x2,gamma){
> kernel<-exp(-(1/gamma^2)*t(x1-x2)%*%(x1-x2))
> return(kernel)
> }
```

Now, using this kernel function, I created the kernel matrix $K_{(n\times n)}(\mathbf{x},\mathbf{x}')$.

```
> k<-matrix(0,N,N) # Defined an (n×n) empty matrix
> for(i in 1:N){
> for(j in 1:N){
>       k[i,j]<-rbf_kernel(X[i,],X[j,],gamma)
> }
> }
```

The output kernel matrix looks like the following matrix

$$K_{(n\times n)} = \begin{bmatrix} K(x_1,x_1) & K(x_1,x_2) & \dots & K(x_1,x_n) \\ K(x_2,x_1) & K(x_2,x_2) & \dots & K(x_2,x_n) \\ \vdots & \vdots & \ddots & \vdots \\ K(x_n,x_1) & K(x_n,x_2) & \dots & K(x_n,x_n) \end{bmatrix}$$

Step 4:

In support vector classification problem **H** is an $(n \times n)$ matrix because there were only n numbers Lagrange multiplier $(\alpha_i, where\ i = 1, 2, \dots, n)$. However, in support vector regression problem **H** is a $(2n \times 2n)$ matrix because here we have 2n numbers Lagrange multiplier $(\alpha_i, \alpha_i^*, where\ i = 1, 2, \dots, n)$. That's why, I defined Hessian Matrix **H** as

$$\mathbf{H}_{(2n \times 2n)} = T' \begin{bmatrix} K & -K \\ -K & K \end{bmatrix} T$$

where K is an (n×n) kernel matrix, and T is a $(2n \times 2n)$ $matrix$ defined above.

```
> K<-rbind(cbind(k,-k),(cbind(-k,k)))
> K<-K+(diag(2*N)*1e-7)
> T <- rbind(cbind(diag(N),-diag(N)),(cbind(diag(N),-diag(N))))
> H <- t(T) %*% K %*% T
```

Here, **H** is not a positive definite matrix. So, I added a small value to the diagonal of **H** to make it positive definite matrix; which condition should be satisfied to use quadratic programming in R.

```
> H<-H+(diag(2*N)*1e-7)
```

Step 5:

Defined the vector (**dv**) appearing in the quadratic function to be minimized.

```
> dv = t(as.vector(cbind((esp - Y),(esp + Y))))
```

The output vector looks like the following vector,

$$\boldsymbol{dv} = d'_{(1 \times 2n)} = [(\varepsilon - y_1), (\varepsilon - y_2), \dots, (\varepsilon - y_n), (\varepsilon + y_1), (\varepsilon + y_2), \dots, (\varepsilon + y_n)]$$

Defined **meq**, our first constraint is an equality constraint so **meq** is equal to 1 and all further will be treated as inequality constraints.

```
> meq<-1
```

Then, defined **Am**; matrix $((4n + 1) \times 2n)$ defining the constraints under which we want to minimize the quadratic function. And, **bv**; vector $((4n + 1) \times 1)$ holding the values of b_0.

Here, our constraint is $A'b \geq b_0$

Where, Am = A matrix & bv = b_0 vector, which are defined above.

```
> Am <- cbind(matrix(c(rep(1,N),rep(-1,N))),diag(2*N))
> bv <- rep(0,1+2*N)
> if(C!=Inf){
> Am <- cbind(Am,-1*diag(2*N))
> bv <- c(cbind(matrix(bv,1),matrix(rep(-C,2*N),1)))
> }
```

Now, we used solve.QP from 'quadprog' to solve the quadratic problem we have. 'alpha_org' stored all the alphas (2n) from the solution of QP problem.

```
> alpha_org<-solve.QP(Dmat=H, dvec=dv,
                Amat=Am,meq=meq,bvec=bv)$solution
```

Step 6:

Divided the α (alpha_org) into 'alp_i' and 'alp_i_star'

```
> alp_i <- alpha_org[1:N]
> alp_i_star <- alpha_org[(N+1):(2*N)]
```

Filtered $\alpha_i \; and \; \alpha_i{}^*$ using the constraint $(\alpha_i \times \alpha_i{}^*) > \varepsilon$.

```
> alp_constraint <- alp_i*alp_i_star
> indx2<-which(alp_constraint>esp, arr.ind=TRUE)
> alpha_i <- alp_i[indx2]
> alpha_i_star <- alp_i_star[indx2]
```

Here, alpha_i is our final α_i and alpha_i_star is our final $\alpha_i{}^*$.

Step 7:

In this step, our goal is to calculate constant b from the following model we defined in (1),

$$f(x) = \langle w, x \rangle + b \quad with \, w \in \chi, b \in \mathbb{R}$$

The following codes are similar to the codes we used in SVM classification problem.

```
> nSV<-length(indx2)
> if(nSV==0){
> throw("QP is not able to give a solution for these data points")
> }
```

```
> Xv<-X[indx2,]
> Yv<-Y[indx2]
> Yv<-as.vector(Yv)
> b <- numeric(nSV)
> wx <- numeric(nSV)
```

Created a function to calculate $\langle w, x \rangle = \sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x_j)$, here K is the RBF kernel matrix.

```
> for (i in 1:nSV){
> for (m in 1:nSV){
> wx[m] <- (alpha_i[m]-
  alpha_i_star[m])*rbf_kernel(Xv[m,],Xv[i,],gamma)
> }
```

Here, I calculated b as b = Y - $\sum_{i=1}^{n} (\alpha_i - \alpha_i^*) K(x_i, x_j)$. And, took the average of b vector I got from previous function.

```
> b[i]<-Yv[i]-sum(wx)
> }
> w0 <- mean(b)
```

This is the end of the training model we defined as a function in **Step 2**.

```
> list(alpha_org=alpha_org,alpha_i=alpha_i,alpha_i_star=alpha_i_star,b
  =b, w0=w0, nSV=nSV, Xv=Xv, Yv=Yv, gamma=gamma)
> }
> Training_model<-SVRtrain(X,Y,C=15,gamma=2)
> Training_model
```

Output: The output matrices are very large so I am just providing part of the outputs.

b = 23.04

Output of α_i *and* α_i^* is

```
$alpha_i
  [1] 6.337264e-09 1.216337e-07 1.669341e-08 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [10] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 6.700903e-09 1.500000e+01 1.500000e+01 1.937269e-08 7.545859e-08
 [19] 1.171310e-08 1.099244e-07 3.217276e-09 3.661517e-08 3.134650e-09 3.877213e-08 1.251188e-08 1.572649e-08 8.477849e-09
 [28] 1.300586e-07 5.837753e-09 1.500000e+01 1.500000e+01 1.500000e+01 3.017566e-09 1.500000e+01 1.500000e+01 1.500000e+01
 [37] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [46] 1.500000e+01 2.804819e-07 1.593506e-09 3.999432e-07 1.724730e-09 6.798153e-10 1.418711e-07 1.325167e-07 1.515241e-07
 [55] 3.327460e-08 6.462963e-08 8.627207e-08 3.170698e-08 2.529747e-07 7.366999e-08 1.762993e-07 1.124148e-07 1.304869e-07
 [64] 1.076185e-07 1.095250e-08 6.886070e-08 2.052738e-08 4.008930e-08 8.745802e-10 3.613847e-09 7.954362e-08 1.261926e-07
 [73] 3.114276e-07 7.782413e-08 1.014557e-07 6.888956e-08 1.232200e-07 9.153847e-08 1.063167e-07 7.038636e-09 5.838183e-09
 [82] 2.300322e-08 1.500000e+01 2.694217e-08 1.361806e-09 2.309611e-08 6.145523e-07 1.089409e-08 1.580037e-07 4.855957e-08
 [91] 9.030132e-09 8.558067e-08 5.277854e-08 1.270064e-07 2.071357e-07 8.566143e-08 9.536695e-08 4.416850e-09 8.016694e-10
[100] 1.341237e-08 1.442299e-07 2.758903e-07 7.050528e-09 6.769048e-08 9.938471e-09 2.173383e-08 3.268428e-08 1.177244e-07
[109] 9.336994e-08 2.844621e-08 1.500000e+01 1.500000e+01 9.946358e-09 1.500000e+01 8.693167e-09 1.500000e+01 1.500000e+01
[118] 1.500000e+01 2.301710e-08 1.500000e+01 4.609254e-08 1.500000e+01 1.500000e+01 6.430573e-09 2.484477e-08 1.500000e+01
[127] 1.870574e-09 3.715387e-08 1.255527e-08 6.738694e-08 1.420874e-07 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[136] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[145] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[154] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[163] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[172] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 6.880455e-09 1.609967e-08 6.173483e-09 1.590907e-08
[181] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.038121e-08 1.500000e+01 1.500000e+01 1.251673e-08 1.517097e-08
[190] 3.607531e-09

$alpha_i_star
  [1] 1.500000e+01 1.500000e+01 1.500000e+01 7.811971e-08 1.984916e-08 4.080486e-08 4.014293e-08 4.415588e-09 6.268357e-08
 [10] 7.062292e-09 2.678089e-08 2.250504e-08 1.075296e-08 1.500000e+01 3.094140e-08 1.860803e-09 1.500000e+01 1.500000e+01
 [19] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [28] 1.500000e+01 1.500000e+01 9.544851e-11 3.877698e-08 5.433044e-09 1.500000e+01 1.139622e-08 3.909340e-08 5.365740e-09
 [37] 3.356926e-09 6.771324e-08 1.817458e-08 5.560331e-09 3.178793e-08 4.259805e-08 5.295979e-08 3.210691e-08 5.573425e-09
 [46] 7.043106e-08 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [55] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [64] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [73] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [82] 1.500000e+01 7.744848e-09 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
 [91] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[100] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[109] 1.500000e+01 1.500000e+01 7.181970e-09 4.683491e-10 1.500000e+01 8.281905e-09 1.500000e+01 3.969042e-09 1.975203e-08
[118] 1.394556e-08 1.500000e+01 2.457704e-08 1.500000e+01 3.900221e-08 1.756489e-08 1.500000e+01 1.500000e+01 1.720062e-08
[127] 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01 8.209543e-09 1.484589e-08 1.275127e-07 4.221889e-08
[136] 3.920245e-09 5.635584e-09 1.110243e-07 4.508833e-08 3.186973e-08 5.021617e-08 1.318284e-08 9.896047e-08 7.287903e-08
[145] 5.424741e-08 1.712326e-09 1.465480e-09 3.122670e-09 1.013773e-07 1.722572e-07 1.665096e-08 4.288981e-08 8.079579e-08
[154] 1.004845e-07 2.300111e-08 1.113875e-09 1.009084e-07 7.178702e-08 2.856179e-08 1.151011e-09 9.087060e-08 1.061785e-07
[163] 1.423517e-07 6.126292e-09 2.005048e-08 1.185799e-08 1.642811e-08 2.102234e-08 1.092568e-07 5.076566e-08 5.846198e-08
[172] 1.140097e-07 1.723194e-08 2.187606e-08 5.232419e-08 3.156897e-08 1.500000e+01 1.500000e+01 1.500000e+01 1.500000e+01
[181] 1.880273e-08 9.257113e-08 1.455807e-07 7.427564e-08 1.500000e+01 2.474504e-09 3.221385e-08 1.500000e+01 1.500000e+01
[190] 1.500000e+01
```

Step 8:

Here, I defined a new function called 'SVR_predict' to predict Y's.

At first stored Test X and Y in Test_X and Test_Y.

```
> Test_X <- as.matrix(test_data[,1:13],ncol=13)
> Test_Y <- test_data[,14]
> n <- length(Test_Y)
```

Here is the prediction function, where we used all the parameters we got from our training model.

```
> SVR_predict <- function(x,model){
> alpha_i<-model$alpha_i
> alpha_i_star<-model$alpha_i_star
> b<-model$w0
> Yv<-model$Yv
> Xv<-model$Xv
> nSV<-model$nSV
> gamma<-model$gamma
```

Used the same function that we used in training model to calculate $\langle w, x\rangle = \sum_{i=1}^{n}(\alpha_i - \alpha_i^*)\, K(x_i\,,\, x_j)$, here K is the RBF kernel matrix.

```
> wx <- numeric(nSV)
> for (i in 1:n){
> for (m in 1:nSV){
> wx[m] <- (alpha_i[m]-alpha_i_star[m])*rbf_kernel(Xv[m,],x[i,],gamma)
> }
```

Calculated $\hat{Y} = \langle w, x\rangle + b$

```
> Y_hat[i] <- sum(wx) + b
> }
> return(Y_hat)
> }
```

Calling the function 'SVR_predict' to create $\hat{Y}$.

```
> Predicted_Y <- SVR_predict(Test_X,Training_model)
> Y_hat <- t(as.data.frame(Predicted_Y))
```

Finally, calculated Mean Squared Error (MSE)

```
> MSE <- (sum((Y_hat - Test_Y)**2))/n
> MSE
```

Output:

MSE = 64.14

This model's accuracy can be improved by tuning the parameters.

Reference:

1) http://www.svms.org/regression/SmSc98.pdf
2) https://pdfs.semanticscholar.org/43ff/a2c1a06a76e58a333f2e7d0bd498b24365ca.pdf