**Support Vector Machines**

In this tutorial, we’ll examine the implementation and application of SVM scheme to either classification or regression problems. Although SVM has been designed initially as a classifier (linear or nonlinear), the Support Vector Regression (SVR) scheme utilizes SVM’s principles and is applied to predict real values rather than a class. The **e1071 package** was the first implementation of SVM in R.

The svm command is used to train a support vector machine. It can be used to carry out general regression and classification (of nu and epsilon-type), as well as density-estimation. Further details for the options that are included into this command can be found at:

https://www.rdocumentation.org/packages/e1071/versions/1.6-8/topics/svm

***The related code has the following general form for SVM:***

#Import Library

require(e1071) #Contains the SVM

Train <- read.csv(file.choose())

Test <- read.csv(file.choose())

# there are various options associated with SVM training, like changing kernel, gamma and C value.

# create model

model <- svm(Target~Predictor1+Predictor2+Predictor3,data=Train,kernel='linear',gamma=0.2,cost=100)

#Predict Output

preds <- predict(model,Test)

table(preds)

**kernel**: We have various options available with kernel like, “linear”, “rbf”, “poly” and others (default value is “rbf”).  Here “rbf” and “poly” are useful for non-linear hyper-plane. Let’s look at our first example, where we’ve used rbf/linear kernel on iris data set to classify their class.

**gamma**: Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’. Higher the value of gamma, will try to exact fit the as per training data set i.e. generalization error and cause over-fitting problem.

**Cost:**Penalty parameter of the error term. It also controls the tradeoff between smooth decision boundary and classifying the training points correctly.

**First example utilizing SVM as classifier**

library("e1071")

head(iris)

## Sepal.Length Sepal.Width Petal.Length Petal.Width Species

## 1 5.1 3.5 1.4 0.2 setosa

## 2 4.9 3.0 1.4 0.2 setosa

## 3 4.7 3.2 1.3 0.2 setosa

## 4 4.6 3.1 1.5 0.2 setosa

## 5 5.0 3.6 1.4 0.2 setosa

Attach the Data

attach(iris)

# Divide Iris data to x (contain the all features) and y only the classes. Beware, in this example we are using all data as training/testing – this is not the usual case (normally we divide the data into training & testing sets)

x <- subset(iris, select=-Species)

y <- Species

Create SVM Model and show summary

svm\_model <- svm(Species ~ ., data=iris)

summary(svm\_model)

##

## Call:

## svm(formula = Species ~ ., data = iris)

##

##

## Parameters:

## SVM-Type: C-classification

## SVM-Kernel: radial

## cost: 1

## gamma: 0.25

##

## Number of Support Vectors: 51

##

## ( 8 22 21 )

##

##

## Number of Classes: 3

##

## Levels:

## setosa versicolor virginica

Or you can use command like this

Create SVM Model and show summary

svm\_model1 <- svm(x,y)

summary(svm\_model1)

Run Prediction and you can measuring the execution time in R

pred <- predict(svm\_model1,x)

system.time(pred <- predict(svm\_model1,x))

See the confusion matrix result of prediction, using command table to compare the result of SVM prediction and the class data in y variable.

table(pred,y)

## y

## pred setosa versicolor virginica

## setosa 50 0 0

## versicolor 0 48 2

## virginica 0 2 48

Tuning SVM to find the best cost and gamma ..

svm\_tune <- tune(svm, train.x=x, train.y=y,

kernel="radial", ranges=list(cost=10^(-1:2), gamma=c(.5,1,2)))

print(svm\_tune)

##

## Parameter tuning of 'svm':

##

## - sampling method: 10-fold cross validation

##

## - best parameters:

## cost gamma

## 1 0.5

##

## - best performance: 0.05333

# After you find the best cost and gamma, you can create svm model again and try to run again

svm\_model\_after\_tune <- svm(Species ~ ., data=iris, kernel="radial", cost=1, gamma=0.5)

summary(svm\_model\_after\_tune)

##

## Call:

## svm(formula = Species ~ ., data = iris, kernel = "radial", cost = 1,

## gamma = 0.5)

##

##

## Parameters:

## SVM-Type: C-classification

## SVM-Kernel: radial

## cost: 1

## gamma: 0.5

##

## Number of Support Vectors: 59

##

## ( 11 23 25 )

##

##

## Number of Classes: 3

##

## Levels:

## setosa versicolor virginica

pred <- predict(svm\_model\_after\_tune,x)

system.time(predict(svm\_model\_after\_tune,x))

## user system elapsed

## 0 0 0

See the confusion matrix result of prediction, using command table to compare the result of SVM prediction and the class data in y variable.

table(pred,y)

## y

## pred setosa versicolor virginica

## setosa 50 0 0

## versicolor 0 48 2

## virginica 0 2 48

**Further Example:** Let’s see differences in results if we have different gamma values like 0, 10 or 100, different cost values (1, 100, 1000), and linear instead of rbf kernel.

**Example utilizing SVM: (red wine data – training/testing sets)**

Use redwine.csv file for this simulation.

library(“e1071”)

wine<-read.csv("C:/R\_codes/redwine.csv") # use your own folder

summary(wine)

## fixed.acidity volatile.acidity citric.acid residual.sugar

## Min. : 4.60 Min. :0.1200 Min. :0.000 Min. : 0.900

## 1st Qu.: 7.10 1st Qu.:0.3900 1st Qu.:0.090 1st Qu.: 1.900

## Median : 7.90 Median :0.5200 Median :0.260 Median : 2.200

## Mean : 8.32 Mean :0.5278 Mean :0.271 Mean : 2.539

## 3rd Qu.: 9.20 3rd Qu.:0.6400 3rd Qu.:0.420 3rd Qu.: 2.600

## Max. :15.90 Max. :1.5800 Max. :1.000 Max. :15.500

## chlorides free.sulfur.dioxide total.sulfur.dioxide

## Min. :0.01200 Min. : 1.00 Min. : 6.00

## 1st Qu.:0.07000 1st Qu.: 7.00 1st Qu.: 22.00

## Median :0.07900 Median :14.00 Median : 38.00

## Mean :0.08747 Mean :15.87 Mean : 46.47

## 3rd Qu.:0.09000 3rd Qu.:21.00 3rd Qu.: 62.00

## Max. :0.61100 Max. :72.00 Max. :289.00

## density pH sulphates alcohol

## Min. :0.9901 Min. :2.740 Min. :0.3300 Min. : 8.40

## 1st Qu.:0.9956 1st Qu.:3.210 1st Qu.:0.5500 1st Qu.: 9.50

## Median :0.9968 Median :3.310 Median :0.6200 Median :10.20

## Mean :0.9967 Mean :3.311 Mean :0.6581 Mean :10.42

## 3rd Qu.:0.9978 3rd Qu.:3.400 3rd Qu.:0.7300 3rd Qu.:11.10

## Max. :1.0037 Max. :4.010 Max. :2.0000 Max. :14.90

## quality

## Min. :3.000

## 1st Qu.:5.000

## Median :6.000

## Mean :5.636

## 3rd Qu.:6.000

## Max. :8.000

str(wine$quality)

## int [1:1599] 5 5 5 6 5 5 5 7 7 5 ...

x <- subset(wine, select = -quality)

y <-as.numeric(wine$quality)

# 1a) First classify the data treating the last column as an ordered factor (the wine tasters score).

wine\_factor<-cbind(x, quality=as.factor(y))

str(wine\_factor)

## 'data.frame': 1599 obs. of 12 variables:

## $ fixed.acidity : num 7.4 7.8 7.8 11.2 7.4 7.4 7.9 7.3 7.8 7.5 ...

## $ volatile.acidity : num 0.7 0.88 0.76 0.28 0.7 0.66 0.6 0.65 0.58 0.5 ...

## $ citric.acid : num 0 0 0.04 0.56 0 0 0.06 0 0.02 0.36 ...

## $ residual.sugar : num 1.9 2.6 2.3 1.9 1.9 1.8 1.6 1.2 2 6.1 ...

## $ chlorides : num 0.076 0.098 0.092 0.075 0.076 0.075 0.069 0.065 0.073 0.071 ...

## $ free.sulfur.dioxide : num 11 25 15 17 11 13 15 15 9 17 ...

## $ total.sulfur.dioxide: num 34 67 54 60 34 40 59 21 18 102 ...

## $ density : num 0.998 0.997 0.997 0.998 0.998 ...

## $ pH : num 3.51 3.2 3.26 3.16 3.51 3.51 3.3 3.39 3.36 3.35 ...

## $ sulphates : num 0.56 0.68 0.65 0.58 0.56 0.56 0.46 0.47 0.57 0.8 ...

## $ alcohol : num 9.4 9.8 9.8 9.8 9.4 9.4 9.4 10 9.5 10.5 ...

## $ quality : Factor w/ 6 levels "3","4","5","6",..: 3 3 3 4 3 3 3 5 5 3 ...

wineTrain<-wine\_factor[1:1400,]

wineTest<-wine\_factor[1401:1599,]

x\_factor <- subset(wineTest, select = -quality)

y\_factor <- wineTest$quality

wine\_svm <- svm(quality ~ ., data = wineTrain)

summary(wine\_svm)

##

## Call:

## svm(formula = quality ~ ., data = wineTrain)

##

##

## Parameters:

## SVM-Type: C-classification

## SVM-Kernel: radial

## cost: 1

## gamma: 0.09090909

##

## Number of Support Vectors: 1166

##

## ( 430 496 172 46 15 7 )

##

##

## Number of Classes: 6

##

## Levels:

## 3 4 5 6 7 8

# gamma: 0.0909 cost: 1

wine\_factor\_predict <- predict(wine\_svm, x\_factor);

1-sum(wine\_factor\_predict == y\_factor)/length(y\_factor)

## [1] 0.4371859

## tune `svm' for classification with RBF-kernel (default in svm), using one split for training/validation set gamma = 0.06 0.07 0.08 0.09 0.10 0.11; cost = 1.0 1.5 2.0 2.5 3.0 3.5

wine\_svm\_tuned <- tune(svm, quality~., data = wineTrain,

ranges = list(gamma = seq(.05,.11,.01), cost = seq(1,4,0.5)),

tunecontrol = tune.control(sampling = "cross"))

summary(wine\_svm\_tuned)

##

## Parameter tuning of 'svm':

##

## - sampling method: 10-fold cross validation

##

## - best parameters:

## gamma cost

## 0.1 1.5

##

## - best performance: 0.3621429

##

## - Detailed performance results:

## gamma cost error dispersion

## 1 0.05 1.0 0.3764286 0.03595948

## 2 0.06 1.0 0.3742857 0.03353679

## 3 0.07 1.0 0.3728571 0.03363806

## 4 0.08 1.0 0.3735714 0.03384807

## 5 0.09 1.0 0.3735714 0.04041312

## 6 0.10 1.0 0.3678571 0.04127366

## 7 0.11 1.0 0.3678571 0.03842860

## 8 0.05 1.5 0.3771429 0.03746503

## 9 0.06 1.5 0.3757143 0.03502510

## 10 0.07 1.5 0.3714286 0.03779645

## 11 0.08 1.5 0.3642857 0.03897787

## 12 0.09 1.5 0.3628571 0.03981249

## 13 0.10 1.5 0.3621429 0.03810268

## 14 0.11 1.5 0.3628571 0.03850965

## 15 0.05 2.0 0.3750000 0.03661561

## 16 0.06 2.0 0.3707143 0.03649154

## 17 0.07 2.0 0.3692857 0.03719929

## 18 0.08 2.0 0.3678571 0.03722975

## 19 0.09 2.0 0.3635714 0.03904326

## 20 0.10 2.0 0.3628571 0.03512207

## 21 0.11 2.0 0.3650000 0.03554723

## 22 0.05 2.5 0.3757143 0.03598312

## 23 0.06 2.5 0.3721429 0.03756325

## 24 0.07 2.5 0.3700000 0.03654587

## 25 0.08 2.5 0.3642857 0.03955535

## 26 0.09 2.5 0.3671429 0.03486287

## 27 0.10 2.5 0.3664286 0.03532326

## 28 0.11 2.5 0.3657143 0.03607752

## 29 0.05 3.0 0.3714286 0.03688556

## 30 0.06 3.0 0.3728571 0.03761603

## 31 0.07 3.0 0.3692857 0.03970555

## 32 0.08 3.0 0.3664286 0.03467537

## 33 0.09 3.0 0.3671429 0.03598312

## 34 0.10 3.0 0.3685714 0.03566663

## 35 0.11 3.0 0.3657143 0.03821410

## 36 0.05 3.5 0.3742857 0.03782643

## 37 0.06 3.5 0.3728571 0.03670066

## 38 0.07 3.5 0.3700000 0.03607752

## 39 0.08 3.5 0.3671429 0.03453612

## 40 0.09 3.5 0.3700000 0.03654587

## 41 0.10 3.5 0.3685714 0.03767627

## 42 0.11 3.5 0.3721429 0.03875178

## 43 0.05 4.0 0.3728571 0.03731341

## 44 0.06 4.0 0.3692857 0.04027260

## 45 0.07 4.0 0.3707143 0.03602248

## 46 0.08 4.0 0.3700000 0.03463447

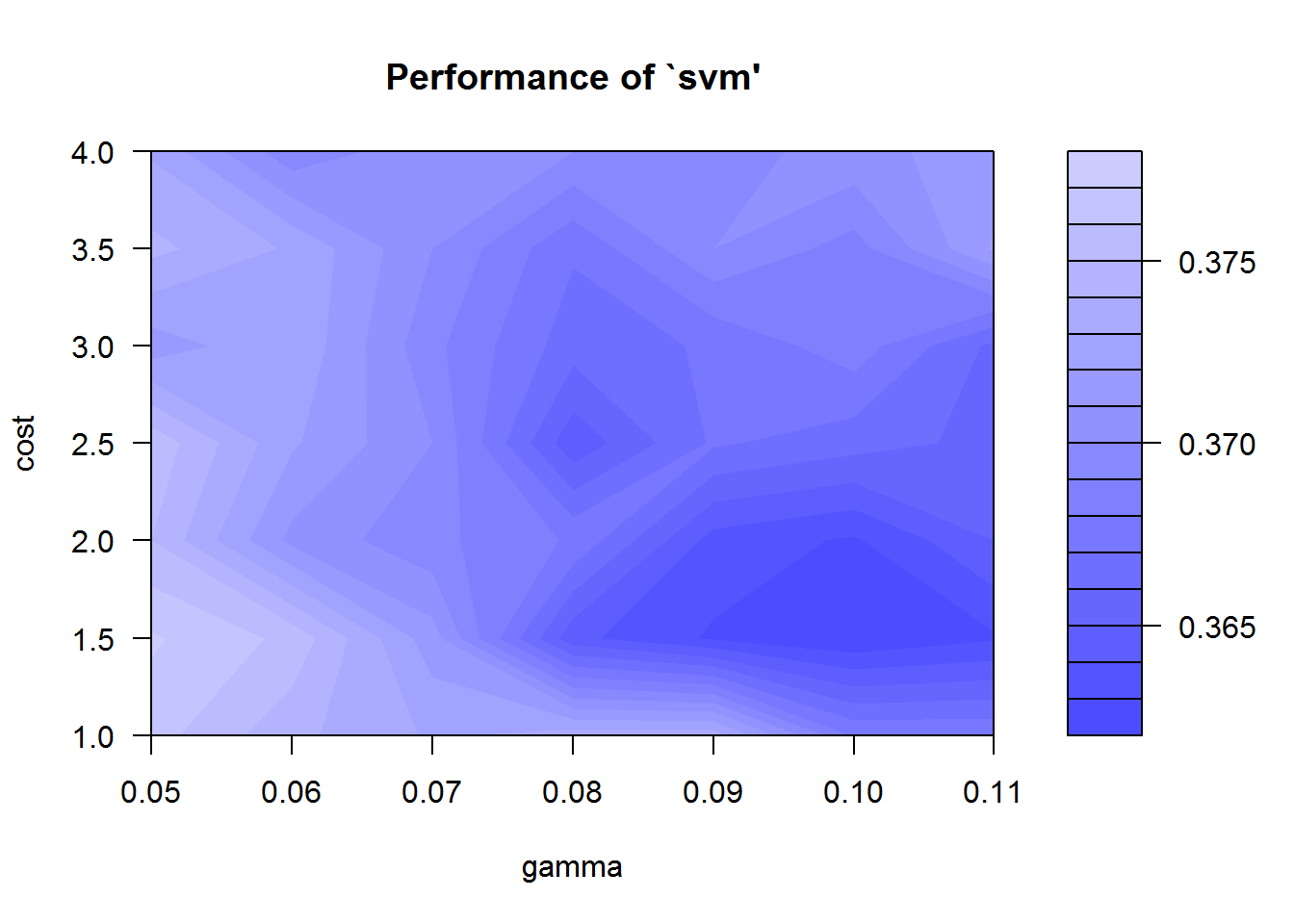
## 47 0.09 4.0 0.3692857 0.03689324

## 48 0.10 4.0 0.3707143 0.03933258

## 49 0.11 4.0 0.3714286 0.04123930

plot(wine\_svm\_tuned)

# takes some time to create it...



wine\_svm\_tuned$best.parameters

## gamma cost

## 13 0.1 1.5

# using gamma = 0.07 cost = 1.5

wine\_svm <- svm(quality ~ ., data = wineTrain, gamma = 0.07, cost = 1.5)

wine\_factor\_predict <- predict(wine\_svm, x\_factor);

1-sum(wine\_factor\_predict == y\_factor)/length(y\_factor)

## [1] 0.4371859

# Next treat the last column as a numeric.

wine\_numeric<-cbind(x, quality=y)

str(wine\_numeric)

## 'data.frame': 1599 obs. of 12 variables:

## $ fixed.acidity : num 7.4 7.8 7.8 11.2 7.4 7.4 7.9 7.3 7.8 7.5 ...

## $ volatile.acidity : num 0.7 0.88 0.76 0.28 0.7 0.66 0.6 0.65 0.58 0.5 ...

## $ citric.acid : num 0 0 0.04 0.56 0 0 0.06 0 0.02 0.36 ...

## $ residual.sugar : num 1.9 2.6 2.3 1.9 1.9 1.8 1.6 1.2 2 6.1 ...

## $ chlorides : num 0.076 0.098 0.092 0.075 0.076 0.075 0.069 0.065 0.073 0.071 ...

## $ free.sulfur.dioxide : num 11 25 15 17 11 13 15 15 9 17 ...

## $ total.sulfur.dioxide: num 34 67 54 60 34 40 59 21 18 102 ...

## $ density : num 0.998 0.997 0.997 0.998 0.998 ...

## $ pH : num 3.51 3.2 3.26 3.16 3.51 3.51 3.3 3.39 3.36 3.35 ...

## $ sulphates : num 0.56 0.68 0.65 0.58 0.56 0.56 0.46 0.47 0.57 0.8 ...

## $ alcohol : num 9.4 9.8 9.8 9.8 9.4 9.4 9.4 10 9.5 10.5 ...

## $ quality : num 5 5 5 6 5 5 5 7 7 5 ...

wineTrain<-wine\_numeric[1:1400,]

wineTest<-wine\_numeric[1401:1599,]

x\_factor <- subset(wineTest, select = -quality)

y\_factor <- wineTest$quality

wine\_svm <- svm(quality ~ ., data = wineTrain)

summary(wine\_svm)

##

## Call:

## svm(formula = quality ~ ., data = wineTrain)

##

##

## Parameters:

## SVM-Type: eps-regression

## SVM-Kernel: radial

## cost: 1

## gamma: 0.09090909

## epsilon: 0.1

##

##

## Number of Support Vectors: 1162

wine\_factor\_predict <- predict(wine\_svm, x\_factor);

sqrt( sum((wineTest$quality-wine\_factor\_predict)^2))/length(wine\_factor\_predict)

## [1] 0.04847373

## tune `svm' for classification with RBF-kernel (default in svm),

## using one split for training/validation set

# gamma = 0.06 0.07 0.08 0.09 0.10 0.11; cost = 1.0 1.5 2.0 2.5 3.0 3.5

wine\_svm\_tuned <- tune(svm, quality~., data = wineTrain,

ranges = list(gamma = seq(.05,.11,.01), cost = seq(1,4,0.5)),

tunecontrol = tune.control(sampling = "cross"))

# this takes a bit of time …

summary(wine\_svm\_tuned)

##

## Parameter tuning of 'svm':

##

## - sampling method: 10-fold cross validation

##

## - best parameters:

## gamma cost

## 0.11 1.5

##

## - best performance: 0.3779477

##

## - Detailed performance results:

## gamma cost error dispersion

## 1 0.05 1.0 0.3933767 0.03111975

## 2 0.06 1.0 0.3894143 0.03040395

## 3 0.07 1.0 0.3859173 0.02794516

## 4 0.08 1.0 0.3836139 0.02557613

## 5 0.09 1.0 0.3820125 0.02484204

## 6 0.10 1.0 0.3806449 0.02437718

## 7 0.11 1.0 0.3798551 0.02417274

## 8 0.05 1.5 0.3887931 0.02951885

## 9 0.06 1.5 0.3860247 0.02670351

## 10 0.07 1.5 0.3827143 0.02435098

## 11 0.08 1.5 0.3805294 0.02319417

## 12 0.09 1.5 0.3789467 0.02276341

## 13 0.10 1.5 0.3781318 0.02331553

## 14 0.11 1.5 0.3779477 0.02422690

## 15 0.05 2.0 0.3876178 0.02826725

## 16 0.06 2.0 0.3834809 0.02450302

## 17 0.07 2.0 0.3809698 0.02302295

## 18 0.08 2.0 0.3788462 0.02231663

## 19 0.09 2.0 0.3783697 0.02356869

## 20 0.10 2.0 0.3786191 0.02504706

## 21 0.11 2.0 0.3797492 0.02677891

## 22 0.05 2.5 0.3860613 0.02628725

## 23 0.06 2.5 0.3824666 0.02356673

## 24 0.07 2.5 0.3795438 0.02291436

## 25 0.08 2.5 0.3787625 0.02406585

## 26 0.09 2.5 0.3791854 0.02545808

## 27 0.10 2.5 0.3804509 0.02705919

## 28 0.11 2.5 0.3820752 0.02840122

## 29 0.05 3.0 0.3845686 0.02497025

## 30 0.06 3.0 0.3809081 0.02367164

## 31 0.07 3.0 0.3790676 0.02372830

## 32 0.08 3.0 0.3790282 0.02550304

## 33 0.09 3.0 0.3804384 0.02712539

## 34 0.10 3.0 0.3824476 0.02829878

## 35 0.11 3.0 0.3835673 0.02974205

## 36 0.05 3.5 0.3835795 0.02423519

## 37 0.06 3.5 0.3802913 0.02390804

## 38 0.07 3.5 0.3786601 0.02483396

## 39 0.08 3.5 0.3800767 0.02652691

## 40 0.09 3.5 0.3820321 0.02776698

## 41 0.10 3.5 0.3831677 0.02949035

## 42 0.11 3.5 0.3854873 0.03128188

## 43 0.05 4.0 0.3822812 0.02415505

## 44 0.06 4.0 0.3797801 0.02470262

## 45 0.07 4.0 0.3792003 0.02592417

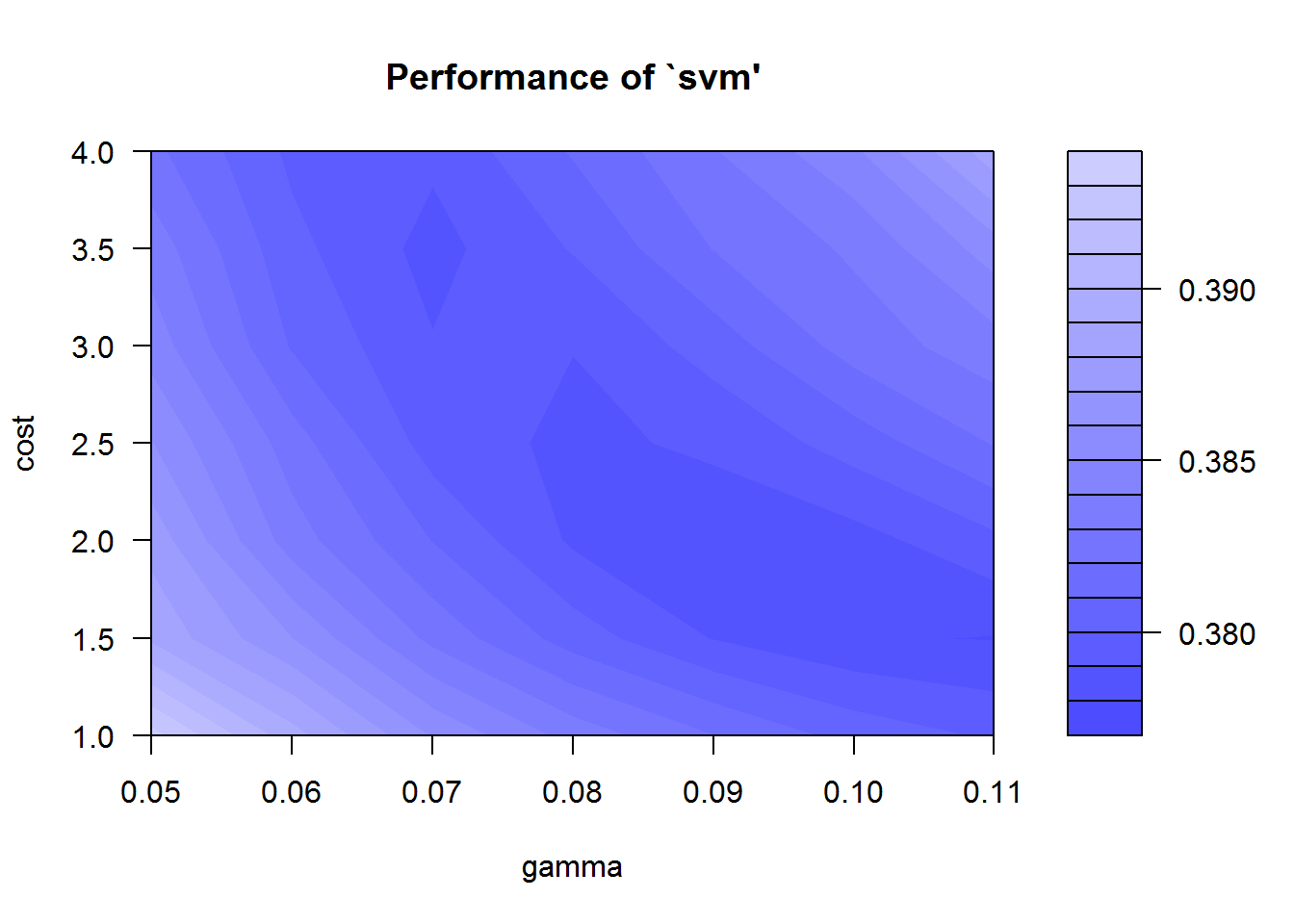
## 46 0.08 4.0 0.3811046 0.02724750

## 47 0.09 4.0 0.3829279 0.02873294

## 48 0.10 4.0 0.3847974 0.03081793

## 49 0.11 4.0 0.3886604 0.03264213

plot(wine\_svm\_tuned)



wine\_svm\_tuned$best.parameters

## gamma cost

## 14 0.11 1.5

# using gamma = 0.1 cost = 2

wine\_svm <- svm(quality ~ ., data = wineTrain, gamma = 0.1, cost = 2)

wine\_factor\_predict <- predict(wine\_svm, x\_factor);

sqrt(sum((wineTest$quality-wine\_factor\_predict)^2))/length(wine\_factor\_predict)

## [1] 0.04952251

**Example utilizing SVR (regression)**

To use SVR in R, in this example, we have created a simple one input-one output problem. We took all the values of x as just a sequence from 1 to 20 and the corresponding values of y were derived using the formula y(t)=y(t-1) + r(-1:9) where r(a,b) generates a random integer between a and b. Initial value for y(1) was set to 3.

library(e1071)

x=c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)

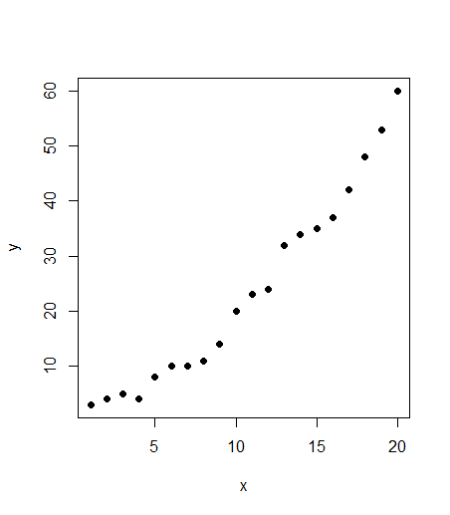
y=c(3,4,5,4,8,10,10,11,14,20,23,24,32,34,35,37,42,48,53,60)

#Create a data frame of the data

train=data.frame(x,y)

#Let’s see how our data looks like. For this we use the plot function

plot(train, pch=16)



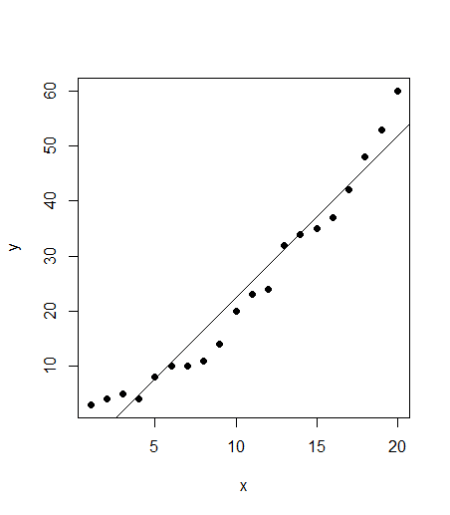
It seems to be a fairly good data (not difficult to be modeled). Looking at the plot, it also seems like a linear regression could also be a good fit to the data. Let’s see if we may have advantaged by using SVR. First, the code for linear regression:

#linear regression

 model <- lm(y ~ x, train)

 # add the fitted line and plot the model using abline

abline(model)



# make a prediction for each x

pred\_lm <- predict(model, train)

 # display the predictions

points(train$x, pred\_lm, col = "blue", pch=4)

#define RMSE function

rmse <- function(error)

{

  sqrt(mean(error^2))

}

 error <- model$residuals  # same as (train$y – pred\_lm)

pred\_RMSE <- rmse(error)   # 3.832974

#fit a model with SVM

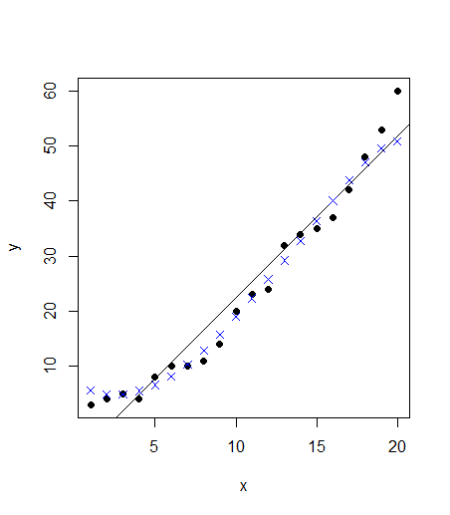
 model\_svm <- svm(y ~ x , train)

#Use the predictions on the data

 pred <- predict(model\_svm, train)

 #Plot the predictions and the plot to see our SVM model fit

 points(train$x, pred, col = "blue", pch=4)



The points follow the actual values much more closely than the abline. Can we verify that? Let’s calculate the RMSE error for SVM model

 #For svm, we can calculate the difference between actual values (train$y) with our predictions (pred)

error\_2 <- (train$y – pred)

 svm\_RMSE <- rmse(error\_2) # 2.696281

summary(model\_svm) # to check the details of our SVM model

In this case, the RMSE for linear model is greater than SVM. A straightforward implementation of SVM has accuracy higher than the linear regression model. However, the SVM model goes far beyond that. We can further improve our SVM model and tune it so that the error is even lower. We will now go deeper into the SVM function and the tune function. We can specify the values for the cost parameter and epsilon which is 0.1 by default. A simple way is to try for each value of epsilon between 0 and 1 (we can take steps of 0.01) and similarly try for cost function from 4 to 2^9 (I will take exponential steps of 2 here). We have totally 101 values of epsilon and 8 values of cost function. Therefore, we need to test 808 models and see which ones perform best. The code may take a short while to run all the models and find the best version. The corresponding code will be:

svm\_tune <- tune(svm, y ~ x, data = train, ranges = list(epsilon = seq(0,1,0.01), cost = 2^(2:9)))

print(svm\_tune)

#Printing gives the output:

#Parameter tuning of ‘svm’:

# - sampling method: 10-fold cross validation

#- best parameters:

# epsilon cost

#0.09 256

#- best performance: 2.569451

#This best performance denotes the MSE. The corresponding RMSE is 1.602951 which is square root of MSE

An advantage of tuning in R is that it lets us extract the best function directly. We don’t have to do anything and just extract the best function from the svm\_tune list. We can now see the improvement in our model by calculating its RMSE error using the following code

#the best model

best\_mod <- svm\_tune$best.model

best\_mod\_pred <- predict(best\_mod, train)

error\_best\_mod <- (train$y - best\_mod\_pred)

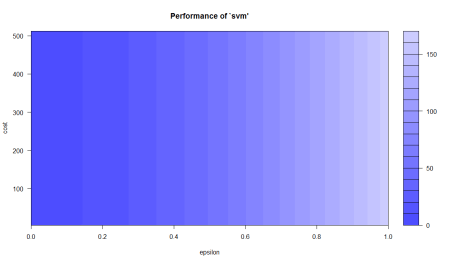
# this value can be different on your computer

# because the tune method randomly shuffles the data

best\_mod\_RMSE <- rmse(error\_best\_mod) # 1.159186

This tuning method is known as grid search. R runs all various models with all the possible values of epsilon and cost function in the specified range and gives us the model which has the lowest error. We can also plot our tuning model to see the performance of all the models together

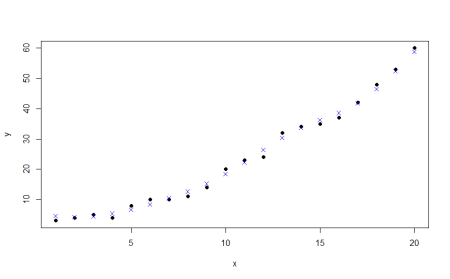
plot(svm\_tune)



This plot shows the performance of various models using colour coding. Darker regions imply better accuracy. The use of this plot is to determine the possible range where we can narrow down our search to and try further tuning if required. For instance, this plot show that we can run tuning for epsilon in the new range of 0 to 0.2 and while we are doing this, we can also use even lower steps (say 0.002) but going further may lead to over-fitting. Let’s see how the best model looks like when plotted.

plot(train,pch=16)

points(train$x, best\_mod\_pred, col = "blue", pch=4)



SVR is a useful and flexible technique, helping the user to deal with the limitations pertaining to distributional properties of underlying variables, geometry of the data and the common problem of model over-fitting. The choice of kernel function is critical for SVR modeling. We could use linear and RBF kernel for linear and non-linear relationship respectively.