HW1_GanapathyBalaji

May 22, 2019

1 Homework 1 - Ganapathy Raaman Balaji

1.0.1 Question 2.1

Describe a situation or problem from your job, everyday life, current events, etc., for which a classification model would be appropriate. List some (up to 5) predictors that you might use.

```
I am a Performance Analytics Engineer at Caterpillar. Performing exploratory analysis one high

Some of the factors are:

1. Engine RPM profile in this window (includes factors such as slope of change in a given time
2. How much the engine is loaded in this window
3. Duration the genset engine was running
4. Generator Power Output
5. Fuel Consumption
```

[ ]:

1.0.2 Question 2.2 (a):

Overall Best Solution: rbfdot seems to provide an accuracy of 99.54128% for cost value = 10000

```
Best vanilladot solution: Accuracy of 86.39144% for cost value = 1
Best anovadot solution: Accuracy of 90.82569% for cost value = 10000
Best polydot solution: Accuracy of 86.39144% for cost value = 1

We see different results if we split the data to train and test datasets. I have explained it
```

I have also worked on this problem by splitting the data set to train-test (80-20) split.

Stepwise explanation to the problem approach:

```
In this problem, I have created vectors of different kernels and cost values. I am looping thro
```

The equation of the model, based on the highest model accuracy is:

```
Response, y = (-56.5077762 x A1) + (-90.32754705 x A2) + (-184.3305137 x A3) + (84.07522492 x A
```

[25]:
```
options(scipen = 999)
library(kernlab)
library(e1071)
library(data.table)
library(dplyr)

data <- read.table("credit_card_data-headers.txt", header = TRUE)
cost_values = c(1, 10, 100, 1000, 10000, 0.1, 0.01, 0.001)
kernel_list = c("vanilladot", "rbfdot", "anovadot", "polydot")
accuracy_val <- c()
c_val <- c()
kernel_val <- c()
weights <- c()
bias <- c()

for (i in 1: length(kernel_list)){
    for (j in 1:length(cost_values)) {
      model <- ksvm(as.matrix(data[,1:10]), as.factor(data[,11]), type =␣
 ↪"C-svc", kernel =  kernel_list[i], scaled = TRUE, C=cost_values[j])
      a<- colSums(model@xmatrix[[1]]*model@coef[[1]])
      a0 <- model@b
      pred <- predict(model,data[,1:10])
      accur = sum(pred == data[,11])/nrow(data)
      accuracy_val <- c(accuracy_val, accur)
      c_val <- c(c_val, cost_values[j])
      kernel_val <- c(kernel_val, kernel_list[i])
      weights <- c(weights, a)
      bias <- c(bias, a0)
    }
}
weights1 <- as.data.frame(matrix(weights, 32, 10, byrow = T))
df <- data.frame(kernel_val, c_val, accuracy_val*100, bias)
names(df) <- c("Kernel","Cost Value","Model Accuracy", "a0")
df1 <- merge(df, weights1, by=0)
drops <- c("Row.names")
df1[order(df1["Model Accuracy"], decreasing = TRUE),  !(names(df1) %in% drops)]
names(df1)[names(df1) == "V1"] <- "a1"
names(df1)[names(df1) == "V2"] <- "a2"
names(df1)[names(df1) == "V3"] <- "a3"
names(df1)[names(df1) == "V4"] <- "a8"
names(df1)[names(df1) == "V5"] <- "a9"
names(df1)[names(df1) == "V6"] <- "a10"
names(df1)[names(df1) == "V7"] <- "a11"
names(df1)[names(df1) == "V8"] <- "a12"
names(df1)[names(df1) == "V9"] <- "a14"
names(df1)[names(df1) == "V10"] <- "a15"
```

```
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
```

	Kernel	Cost Value	Model Accuracy	a0	V1	V2	V3
5	rbfdot	10000.000	99.23547	-0.63824146	-76.93709192106	-73.3619155041	-148.4
4	rbfdot	1000.000	98.01223	-0.70466937	-58.53155010645	-12.6283069575	-40.48
3	rbfdot	100.000	95.25994	-0.78805310	-19.19664569632	-37.2038168337	-8.646
2	rbfdot	10.000	91.43731	-0.44146303	-3.01817583732	-17.8291518521	3.7185
14	anovadot	10000.000	90.82569	-21.79686445	0.11124018539	-104.1006008680	-199.0
11	anovadot	100.000	90.67278	-1.17407433	0.01883723961	-22.4979970645	-28.05
13	anovadot	1000.000	90.67278	-8.83058576	0.13349017779	-29.4473951794	-69.10
10	anovadot	10.000	87.30887	-0.02852669	0.01108490367	-8.1314613538	-10.57
32	rbfdot	1.000	87.00306	-0.42067564	0.58153076023	-1.6867985593	3.7221
1	vanilladot	1.000	86.39144	-0.08148382	-0.00110266416	-0.0008980539	-0.001
9	anovadot	1.000	86.39144	-0.37489521	0.00190220241	-1.5386990249	-0.900
12	vanilladot	10.000	86.39144	-0.08157559	-0.00090336713	-0.0007891036	-0.001
18	polydot	1.000	86.39144	-0.08148471	-0.00117790293	-0.0007585829	-0.001
19	polydot	10.000	86.39144	-0.08154590	-0.00096471814	-0.0010953376	-0.001
20	polydot	100.000	86.39144	-0.08157716	-0.00109297047	-0.0012425741	-0.001
23	vanilladot	100.000	86.39144	-0.08158492	-0.00100653481	-0.0011729048	-0.001
24	polydot	0.100	86.39144	-0.08165418	-0.00121245690	-0.0006070979	-0.001
25	polydot	0.010	86.39144	-0.08198853	-0.00015007085	-0.0014818363	0.0014
29	vanilladot	0.100	86.39144	-0.08155226	-0.00116089805	-0.0006366002	-0.001
30	vanilladot	0.010	86.39144	-0.08198854	-0.00015007376	-0.0014818294	0.0014
15	anovadot	0.100	86.23853	-0.03920849	-0.00001012938	-0.1554799502	-0.084
16	anovadot	0.010	86.23853	-0.14978982	-0.00795464422	0.0236412738	0.0247
21	polydot	1000.000	86.23853	-0.07044332	-0.00022971942	-0.0007721707	0.0003
22	polydot	10000.000	86.23853	-0.07188214	-0.00093819800	0.0026823497	0.0082
27	vanilladot	1000.000	86.23853	-0.07017871	-0.00021491854	0.0007097786	0.0011
28	vanilladot	10000.000	86.23853	-0.07046104	0.00089361671	0.0016125725	-0.000
6	rbfdot	0.100	85.93272	-0.49895859	0.44800542306	2.6930386158	2.9894
26	polydot	0.001	83.79205	0.22261555	-0.00215977831	0.0323381698	0.0466
31	vanilladot	0.001	83.79205	0.22261554	-0.00215977831	0.0323381696	0.0466
17	anovadot	0.001	58.86850	0.40814596	-0.00495820271	0.0732439801	0.0901
7	rbfdot	0.010	56.72783	0.37843714	0.19438004769	0.8585445702	0.9615
8	rbfdot	0.001	54.74006	0.94002520	0.01943800477	0.0867650055	0.0980

[ ]:

1.0.3 Question 2.2(a) Alternate method

REPRODUCING THE ABOVE KSVM FUNCTION WITH 80-20 Train-Test Split for the best C-value for each of the above kernel methods.

Similar to the previous cell, I have created a dataframe of kernel, cost values, accuracy, wieghts and bias vectors. I have ordered the dataframe by accuracy in descending order.

The poor model accuracies could be simply a case of overfitting. In the previous cell, the model calculated predicted accuracy on data it was trained in, thus fit that data too well. But in this case, we are testing the model on a test dataset, leading to poorer model accuracy.

I used the split technique used in this stackoverflow link: https://stackoverflow.com/questions/17200114/how-to-split-data-into-training-testing-sets-using-sample-function

[73]:
```
options(scipen = 999)
library(kernlab)
library(e1071)
library(data.table)
library(caTools)
require(caTools)
set.seed(101)

sample = sample.split(data$R1, SplitRatio = 0.80)
x_data_train <- subset(data[,1:10], sample == TRUE)
y_data_train <- subset(data[,11], sample == TRUE)
x_data_test <- subset(data[,1:10], sample == FALSE)
y_data_test <- subset(data[,11], sample == FALSE)

data <- read.table("credit_card_data-headers.txt", header = TRUE)
cost_values = c(1, 10, 100, 1000, 10000, 0.1, 0.01, 0.001)
kernel_list = c("vanilladot", "rbfdot", "anovadot", "polydot")
accuracy_val <- c()
c_val <- c()
kernel_val <- c()
weights <- c()
bias <- c()

for (i in 1: length(kernel_list)){
    for (j in 1:length(cost_values)) {
      model <- ksvm(as.matrix(x_data_train), as.factor(y_data_train), type =␣
 ↪"C-svc", kernel =  kernel_list[i], scaled = TRUE, C=cost_values[j])
      a<- colSums(model@xmatrix[[1]]*model@coef[[1]])
      a0 <- model@b
      pred <- predict(model,x_data_test)
      accur = sum(pred == y_data_test)/length(y_data_test)

      accuracy_val <- c(accuracy_val, accur)
      c_val <- c(c_val, cost_values[j])
      kernel_val <- c(kernel_val, kernel_list[i])
      weights <- c(weights, a)
      bias <- c(bias, a0)
    }
}
weights1 <- as.data.frame(matrix(weights, 32, 10, byrow = T))
df <- data.frame(kernel_val, c_val, accuracy_val*100, bias)
names(df) <- c("Kernel","Cost Value","Model Accuracy", "Bias")
df1 <- merge(df, weights1, by=0)
drops <- c("Row.names")
```

```r
df1[order(df1["Model Accuracy"], decreasing = TRUE),  !(names(df1) %in% drops)]
```

```
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
```

	Kernel	Cost Value	Model Accuracy	Bias	V1	V2	V3
11	anovadot	100.000	88.54962	-2.66983539	0.15681584311	-22.4787451448	-25.31
13	anovadot	1000.000	87.78626	-7.73772115	0.20149114886	-62.3291855859	-54.53
10	anovadot	10.000	86.25954	-0.62600087	-0.01508102521	-8.2171063937	-10.36
14	anovadot	10000.000	86.25954	-19.75539577	0.23603209871	-141.3793629830	-86.32
1	vanilladot	1.000	85.49618	-0.09739999	-0.00107433634	-0.0020990654	-0.001
9	anovadot	1.000	85.49618	-0.58628080	-0.00106433194	-1.3151866751	-1.274
12	vanilladot	10.000	85.49618	-0.09726800	-0.00117581013	-0.0021618342	-0.002
15	anovadot	0.100	85.49618	-0.06077585	-0.00003832392	-0.1290328776	-0.114
18	polydot	1.000	85.49618	-0.09729321	-0.00117728585	-0.0020144465	-0.001
19	polydot	10.000	85.49618	-0.09740113	-0.00106567026	-0.0020013082	-0.001
20	polydot	100.000	85.49618	-0.09710195	-0.00131673502	-0.0020654587	-0.001
21	polydot	1000.000	85.49618	-0.08588901	0.00016348864	0.0005152847	-0.000
22	polydot	10000.000	85.49618	-0.08526746	-0.00195078227	0.0038529897	0.001
23	vanilladot	100.000	85.49618	-0.09721200	-0.00107671853	-0.0020602805	-0.001
24	polydot	0.100	85.49618	-0.09834222	-0.00136258606	-0.0014045740	-0.001
25	polydot	0.010	85.49618	-0.09754501	-0.00106415440	-0.0018193278	0.003
27	vanilladot	1000.000	85.49618	-0.08606945	0.00019008636	0.0002998667	0.000
28	vanilladot	10000.000	85.49618	-0.08871878	-0.00175092685	0.0036540001	-0.004
29	vanilladot	0.100	85.49618	-0.09838218	-0.00129612346	-0.0013626013	-0.001
30	vanilladot	0.010	85.49618	-0.09754502	-0.00106413518	-0.0018193307	0.003
32	rbfdot	1.000	85.49618	-0.40957213	-2.04442860421	0.3866227893	2.022
6	rbfdot	0.100	84.73282	-0.47822488	0.31827953404	2.4905123762	2.514
26	polydot	0.001	83.96947	0.29164928	-0.00530297781	0.0338661041	0.060
31	vanilladot	0.001	83.96947	0.29164928	-0.00530297871	0.0338661035	0.060
2	rbfdot	10.000	81.67939	-0.44259458	-12.39184196070	-15.6758957448	4.574
3	rbfdot	100.000	80.15267	-0.65145613	-10.20000313384	-27.7218582700	-5.676
4	rbfdot	1000.000	79.38931	-1.23879027	-23.58541446144	-21.2492144343	-124.1
5	rbfdot	10000.000	77.86260	-2.08644510	-14.95093912344	-79.1986580085	-236.4
16	anovadot	0.010	74.04580	-0.19864180	-0.01535700189	0.0350049147	0.046
17	anovadot	0.001	66.41221	0.50465689	-0.00444237055	0.0598649558	0.087
7	rbfdot	0.010	54.96183	0.47706905	0.12963618386	0.7170428903	0.916
8	rbfdot	0.001	54.96183	0.94854979	0.01512422145	0.0715753134	0.092

1.0.4 Question 2.3:

Using the k-nearest-neighbors classification function kknn contained in the R kknn package, suggest a good value of k, and show how well it classifies that data points in the full data set. Don't forget to scale the data (scale=TRUE in kknn). Note that kknn will read the responses as continuous, and return the fraction of the k closest responses that are 1 (rather than the most common response, 1 or 0).

Stepwise explanation to the problem approach:

```
In this problem, I have created vectors of different kernels and k values. The logic is simila

Since the predicted values have integers with decimal places, I round them to get predicted va
```

Using this approach, 12 and 15 seem to be good values of k, and the model accuracy for both the values of k is 85.32110 %. I get the best possible accuracy when using "optimal" kernel.

[43]:
```
library(kernlab)
library(kknn)

data <- read.table("credit_card_data-headers.txt", header = TRUE)

k_list <- c(1:50)
kernel_list <- c("rectangular" , "triangular", "inv", "gaussian", "rank",␣
 ↪"optimal")

accuracy_val <- c()
k_val <- c()
kernel_val <- c()
pred <- rep(0, nrow(data))

for (m in 1:length(kernel_list)){
    for (i in 1:length(k_list)) {
        for (j in 1:nrow(data)) {
            kknn_model <- kknn(R1~., data[-j,1:11], data[j,1:11], k =␣
 ↪k_list[i], distance = 2, scale = TRUE, kernel = kernel_list[m])
            pred[j] <- round(fitted(kknn_model))
        }
        accur = sum(pred == data[,11])/length(data[,11])
        accuracy_val <- c(accuracy_val, accur)
        k_val <- c(k_val, k_list[i])
        kernel_val <- c(kernel_val, kernel_list[m])
    }
}

df <- data.frame(kernel_val, k_val, accuracy_val*100)
names(df) <- c("Kernel","K Value","Model Accuracy")
df[order(df["Model Accuracy"], decreasing = TRUE), ]

# cat("Max Accuracy is:", max(accuracy[1:length(k_list)]), ",\nand the␣
 ↪corresponding value of k is: ", k_list[which.max(accuracy[1:
 ↪length(k_list)])])
```

	Kernel	K Value	Model Accuracy
262	optimal	12	85.32110
265	optimal	15	85.32110
60	triangular	10	85.16820
255	optimal	5	85.16820
261	optimal	11	85.16820
263	optimal	13	85.16820
264	optimal	14	85.16820
266	optimal	16	85.16820
267	optimal	17	85.16820
268	optimal	18	85.16820
22	rectangular	22	85.01529
44	rectangular	44	85.01529
61	triangular	11	85.01529
62	triangular	12	85.01529
64	triangular	14	85.01529
142	inv	42	85.01529
158	gaussian	8	85.01529
222	rank	22	85.01529
244	rank	44	85.01529
260	optimal	10	85.01529
269	optimal	19	85.01529
270	optimal	20	85.01529
48	rectangular	48	84.86239
63	triangular	13	84.86239
65	triangular	15	84.86239
66	triangular	16	84.86239
67	triangular	17	84.86239
106	inv	6	84.86239
143	inv	43	84.86239
248	rank	48	84.86239
...	...	...	...
178	gaussian	28	82.72171
4	rectangular	4	82.56881
15	rectangular	15	82.56881
176	gaussian	26	82.56881
204	rank	4	82.56881
215	rank	15	82.56881
3	rectangular	3	82.26300
17	rectangular	17	82.26300
19	rectangular	19	82.26300
103	inv	3	82.26300
153	gaussian	3	82.26300
203	rank	3	82.26300
217	rank	17	82.26300
219	rank	19	82.26300
54	triangular	4	81.95719
1	rectangular	1	81.49847
51	triangular	1	81.49847
52	triangular	2	81.49847
101	inv	1	81.49847
102	inv	2	81.49847
151	gaussian	1	81.49847

[ ]:

1.0.5 Question 3.1 (a):

Problem:

Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 1
Using cross-validation (do this for the k-nearest-neighbors model; SVM is optional)

Solution:

Here I am using train.kknn and cv.kknn methods.
train.kknn - Training of kknn method via leave-one-out crossvalidation
cv.kknn - k-fold cross-validation, where k is the number number of data points (HERE I AM USIN

I am also plotting all the "train.kknn" models in one plot. This shows mean squared error valu

From the train.kknn approach, attributes(model) shows that the Minimal mean squared error is 1

Websites I referred to other than documentation, to work on this problem:

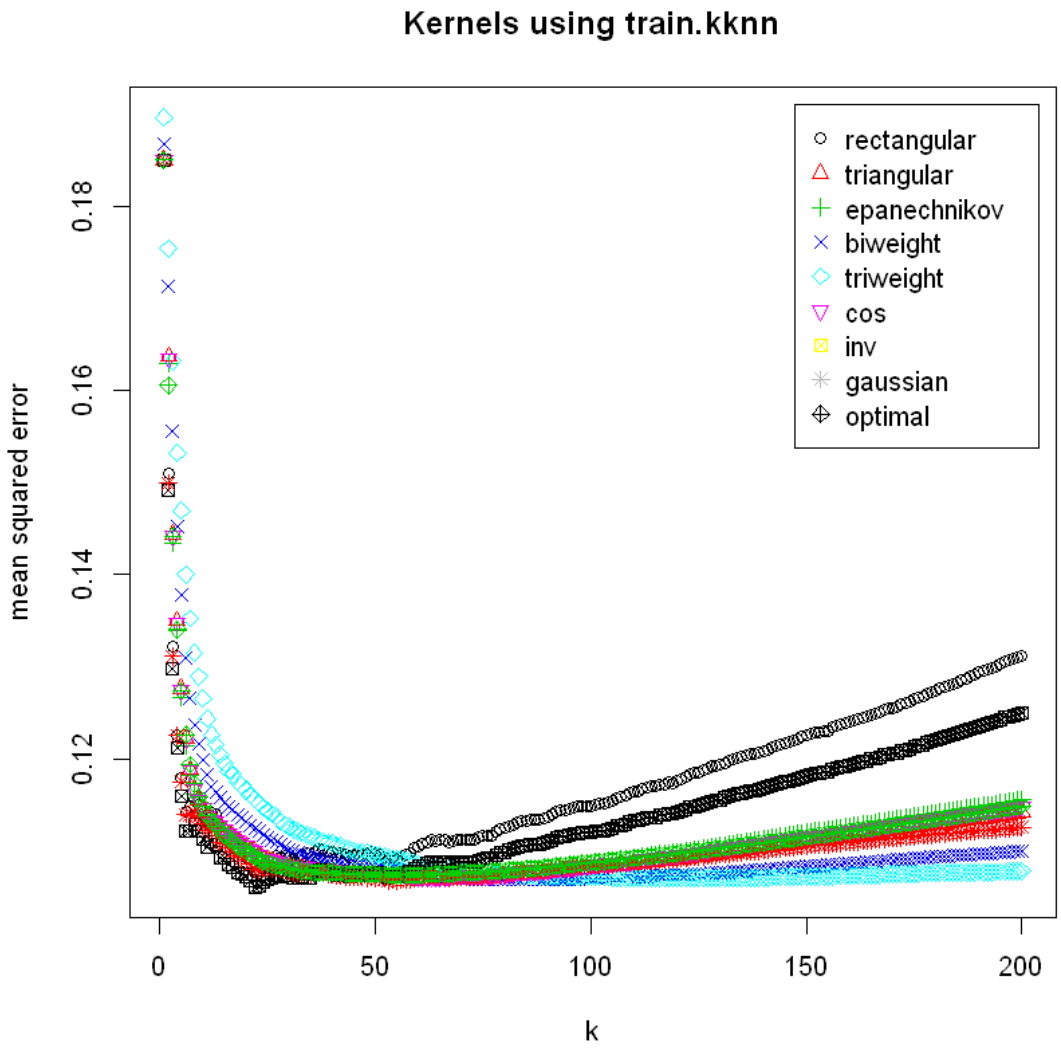
1. train.kknn - http://course1.winona.edu/bdeppa/Stat%20425/Handouts/Section%2013%20-%20Neares
2. cv.kknn - https://www.kaggle.com/jaybee79/sf-crime-class-knn-s-only-benchmark

[14]:
```
library(kernlab)
library(kknn)
library(data.table)
library(e1071)
library(caTools)
require(caTools)
set.seed(101)

data <- read.table("credit_card_data-headers.txt", header = TRUE)
kernels = c("rectangular", "triangular", "epanechnikov", "biweight",␣
 ↪"triweight", "cos", "inv", "gaussian", "optimal")

cat("-------------------------------------BEGIN train.kknn method␣
 ↪----------------------------------------\n")
train_model <- train.kknn(R1~., data, kmax = 200, distance = 2, kernel =␣
 ↪kernels, scale = TRUE)
train_model
plot(train_model)
title("Kernels using train.kknn")

best_train_kernel <- train_model$best.parameters$kernel
best_train_kval <- train_model$best.parameters$k
kknn_train_model <- train.kknn(R1~., data, ks=best_train_kval, distance = 2,␣
 ↪kernel = best_train_kernel, scale = TRUE)
```

```
pred <- round(predict(kknn_train_model, data[,1:10]))
train_model_accuracy <- sum(pred == data[,11])/length(data[,11])
cat("\nBest Kernel is:",best_train_kernel,", Best K Value is:
 ↪",best_train_kval,", and the accuracy is:",train_model_accuracy*100,"%\n")
cat("\n-------------------------------------END train.kknn method␣
 ↪-----------------------------------------\n\n")
cat("\n-------------------------------------BEGIN cv.kknn method␣
 ↪-----------------------------------------\n")
kvals <- c(1:100)
model_accuracy <- c()

for (i in 1:length(kernels)){
  for (j in 1:length(kvals)){
    cv_model <- cv.kknn(R1~., data= data, kcv = 10, scale = TRUE, kernel =␣
 ↪kernels[i], k = kvals[j])
    cv_model <- data.table(cv_model[[1]])
    cv_model_accuracy <- sum(round(cv_model$yhat) == data$R1)/length(data$R1)
    model_accuracy <- c(model_accuracy, cv_model_accuracy)
  }
  cat("\nFor", kernels[i], "kernel, model accuracy is:",␣
 ↪max(model_accuracy)*100,"% and corresponding k-value is:",kvals[which.
 ↪max(model_accuracy[1:length(kvals)])])

}
cat("\n-------------------------------------END cv.kknn method␣
 ↪-----------------------------------------\n")
```

```
Warning message:

-------------------------------------BEGIN train.kknn method
-----------------------------------------

Call:
train.kknn(formula = R1 ~ ., data = data, kmax = 200, distance = 2,     kernel = kernels, scale

Type of response variable: continuous
minimal mean absolute error: 0.1850153
Minimal mean squared error: 0.1061155
Best kernel: inv
Best k: 22



Best Kernel is: inv , Best K Value is: 22 , and the accuracy is: 100 %

-------------------------------------END train.kknn method
-----------------------------------------
```

```
-------------------------------------BEGIN cv.kknn method
-----------------------------------------

For rectangular kernel, model accuracy is: 85.01529 % and corresponding k-value
is: 38
For triangular kernel, model accuracy is: 85.3211 % and corresponding k-value
is: 38
For epanechnikov kernel, model accuracy is: 85.47401 % and corresponding k-value
is: 38
For biweight kernel, model accuracy is: 85.47401 % and corresponding k-value is:
38
For triweight kernel, model accuracy is: 85.77982 % and corresponding k-value
is: 38
For cos kernel, model accuracy is: 85.93272 % and corresponding k-value is: 38
For inv kernel, model accuracy is: 85.93272 % and corresponding k-value is: 38
For gaussian kernel, model accuracy is: 85.93272 % and corresponding k-value is:
38
For optimal kernel, model accuracy is: 85.93272 % and corresponding k-value is:
38
-------------------------------------END cv.kknn method
-----------------------------------------
```



1.0.6 Question 3.1 (b):

Here, I am splitting the data to train, validate and test. I am using 4 different ratios : (50%,25%,25%), (60%,20%,20%), (70%,15%,15%), and (80%,10%,10%).
I am going to run the ksvm model on the split datasets to predict the best model classifier.
I am not going to use Rotating method at this point.

For the different ratios, I am predicting model accuracies for four different kernels. Iterati

From Problem 2.1 (a) we concluded that rbfdot with C = 10000 had the best model accuracy. Howe

Documentation on Stackoverflow showing examples to split data frame into train,validation and test datasets. https://stackoverflow.com/questions/36068963/r-how-to-split-a-data-frame-into-training-validation-and-test-sets

[62]:
```
library(kernlab)
library(e1071)
library(caTools)
require(caTools)
set.seed(101)

data <- read.table("credit_card_data-headers.txt", header = TRUE)
kernel_list = c("vanilladot", "rbfdot", "anovadot", "polydot")

w <- c(0.5,0.25,0.25)
x <- c(0.6,0.2,0.2)
y <- c(0.7,0.15,0.15)
z <- c(0.8,0.1,0.1)
m <- list(w,x,y,z)

cost_value <- c()
kernel_val <- c()
train_frac_list <- c()
validate_frac_list <- c()
test_frac_list <- c()
Train_Accuracy  <- c()
Validation_Accuracy  <- c()
Test_Accuracy  <- c()

for (k1 in 1:length(kernel_list)){
    for (p in 1:length(m)){
        fractionTraining <- m[[p]][1]
        fractionValidation <- m[[p]][2]
        fractionTest <- m[[p]][3]

        sampleSizeTraining <- floor(fractionTraining * nrow(data))
        sampleSizeValidation <- floor(fractionValidation * nrow(data))
        sampleSizeTest <- floor(fractionTest * nrow(data))

        indicesTraining <- sort(sample(seq_len(nrow(data)),␣
↪size=sampleSizeTraining))
        indicesNotTraining <- setdiff(seq_len(nrow(data)), indicesTraining)
        indicesValidation <- sort(sample(indicesNotTraining,␣
↪size=sampleSizeValidation))
        indicesTest <- setdiff(indicesNotTraining, indicesValidation)

        dfTraining   <- data[indicesTraining, ]
        x_dfTraining <- dfTraining[,1:10]
        y_dfTraining <- dfTraining[,11]
```

```
        dfValidation <- data[indicesValidation, ]
        x_dfValidation <- dfValidation[,1:10]
        y_dfValidation <- dfValidation[,11]
        dfTest       <- data[indicesTest, ]
        x_dfTest <- dfTest[,1:10]
        y_dfTest <- dfTest[,11]

        cost_values = c(1, 10, 100, 1000, 10000, 0.1, 0.01, 0.001)
        train_accuracy <- c()
        prediction_train <- c()
        train_model <- c()

        for (i in 1:length(cost_values)){
          ksvm_model <- ksvm(as.matrix(x_dfTraining), as.factor(y_dfTraining),␣
↪type = "C-svc", kernel = kernel_list[k1], C = cost_values[i], scaled = TRUE)
          a <- colSums(ksvm_model@xmatrix[[1]]*ksvm_model@coef[[1]])
          a0 <- ksvm_model@b
          accuracy <- sum(predict(ksvm_model,x_dfTraining) == y_dfTraining)/
↪length(y_dfTraining)
          train_accuracy <- c(train_accuracy,accuracy)
          train_model <- c(train_model, ksvm_model)
          prediction_train <- c(prediction_train,␣
↪predict(ksvm_model,x_dfTraining))
        }

        model <- train_model[which.max(train_accuracy[1:length(cost_values)])]
        C_val = cost_values[which.max(train_accuracy[1:length(cost_values)])]
        train_accuracy <- max(train_accuracy[1:length(cost_values)])

        validation_predict <- predict(model[[1]],x_dfValidation)
        validation_accuracy <- sum(validation_predict == y_dfValidation)/
↪length(y_dfValidation)

        test_predict <- predict(model[[1]],x_dfTest)
        test_accuracy <- sum(test_predict == y_dfTest)/length(y_dfTest)

        Train_Accuracy <- c(Train_Accuracy, train_accuracy)
        Validation_Accuracy <- c(Validation_Accuracy, validation_accuracy)
        Test_Accuracy <- c(Test_Accuracy, test_accuracy)
        cost_value <- c(cost_value, C_val)
        train_frac_list <- c(train_frac_list, fractionTraining*100)
        validate_frac_list <- c(validate_frac_list, fractionValidation*100)
        test_frac_list <- c(test_frac_list, fractionTest*100)
        kernel_val <- c(kernel_val, kernel_list[k1])
    }
}
```

```
df <- data.frame(kernel_val, cost_value, train_frac_list, validate_frac_list, test_frac_list, Train_Accuracy*100, Validation_Accuracy*100, Test_Accuracy*100)
names(df) <- c("Kernel","Cost Value","Train Split %","Validate Split %","Test Split %","Train Accuracy %","Validate Accuracy %","Test Accuracy %")
df[order(df["Test Accuracy %"], decreasing = TRUE), ]
```

```
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
```

```
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
```

```
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
Setting default kernel parameters
```

	Kernel	Cost Value	Train Split %	Validate Split %	Test Split %	Train Accuracy %	Validate
16	polydot	1	80	10	10	85.27725	90.76923
14	polydot	1	60	20	20	85.96939	83.07692
2	vanilladot	1	60	20	20	85.96939	84.61538
10	anovadot	10000	60	20	20	92.34694	83.07692
15	polydot	1	70	15	15	85.12035	89.79592
11	anovadot	10000	70	15	15	89.71554	90.81633
13	polydot	1	50	25	25	86.23853	87.11656
4	vanilladot	1	80	10	10	86.61568	86.15385
1	vanilladot	1	50	25	25	85.32110	90.18405
9	anovadot	10000	50	25	25	94.80122	80.36810
12	anovadot	10000	80	10	10	92.92543	81.53846
3	vanilladot	1	70	15	15	87.52735	86.73469
7	rbfdot	10000	70	15	15	99.34354	77.55102
8	rbfdot	10000	80	10	10	99.42639	73.84615
5	rbfdot	10000	50	25	25	100.00000	73.61963
6	rbfdot	10000	60	20	20	99.23469	75.38462

[ ]: