Week-1-SVM-KNN

Anonymous

May 23, 2018

Setup

Loading required libraries, setting intitial seed

```
library(kernlab)
library(kknn)
```

```
## Warning: package 'kknn' was built under R version 3.3.3
```

```
set.seed(829348)
```

Setting Current Working Directory

"setwd" statement Hidden to avoid exposing my directory structure

Data Load

```
data = read.csv("2.2credit_card_data-headersSummer2018.txt", header = TRUE, sep = '\t')
head(data)
```

```
##   A1    A2    A3   A8 A9 A10 A11 A12 A14 A15 R1
## 1  1 30.83 0.000 1.25  1   0   1   1 202   0  1
## 2  0 58.67 4.460 3.04  1   0   6   1  43 560  1
## 3  0 24.50 0.500 1.50  1   1   0   1 280 824  1
## 4  1 27.83 1.540 3.75  1   0   5   0 100   3  1
## 5  1 20.17 5.625 1.71  1   1   0   1 120   0  1
## 6  1 32.08 4.000 2.50  1   1   0   0 360   0  1
```

Exploratory Analysis of Data

How healthy is our data?

```
dim(data)
```

```
## [1] 654  11
```

```
dim(na.omit(data))
```

```
## [1] 654  11
```

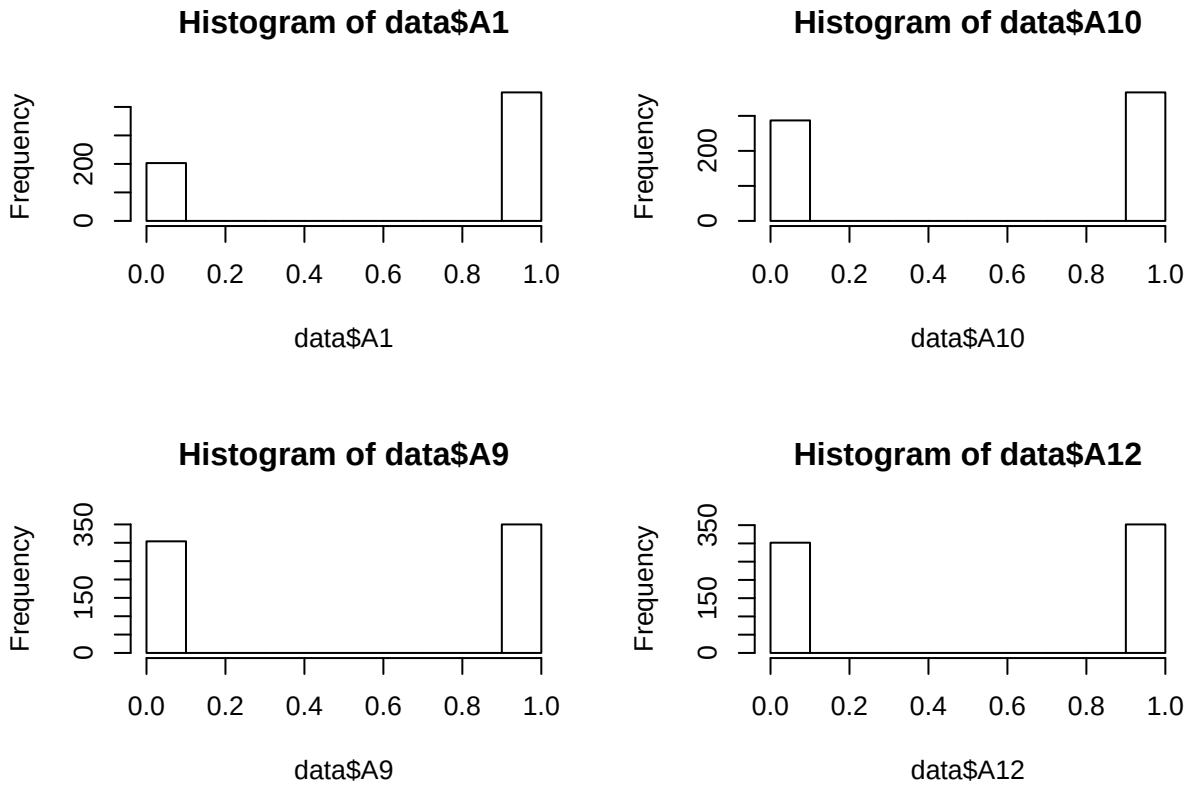
Conclusion:

Okay, Looks like healthy dataset. We should filter for NaN as well to be doubly sure.

Categorical Columns? Verify

A1, A9, A10 and A12 seem to have binary categorical values. Let us verify that.

```
par(mfrow=c(2,2), mfcol=c(2,2))
hist(data$A1)
hist(data$A9)
hist(data$A10)
hist(data$A12)
```

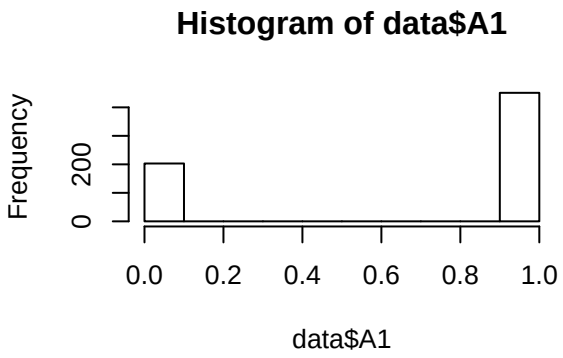


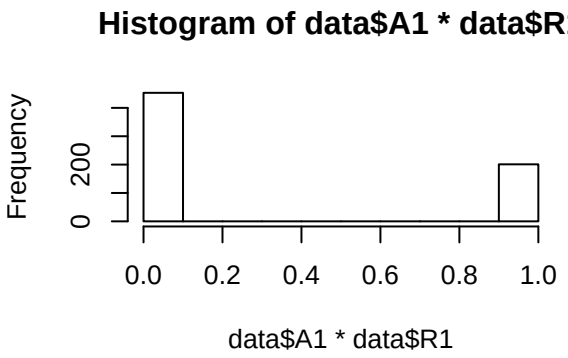
###Conclusion: Yes, these 4 columns are the Binary categorical variables. Confirmed! They take only 2 values : 0 or 1

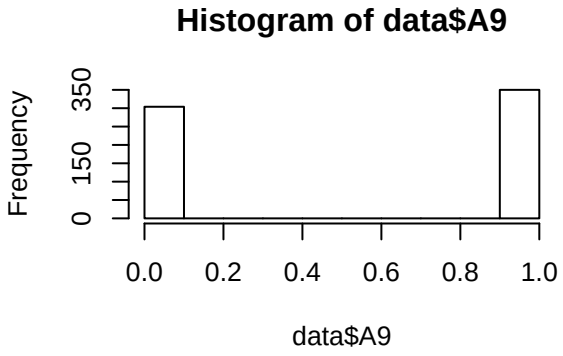
Categorical Columns (vs) Target variable

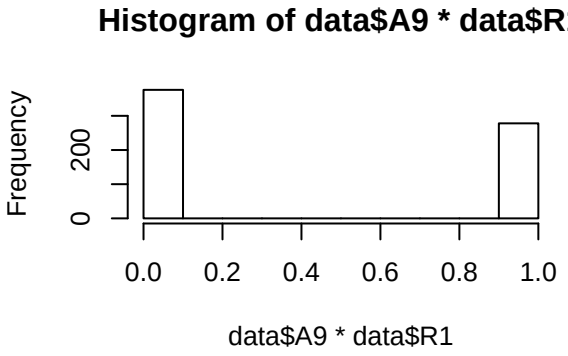
Lets verify how well the binary attributes correlate with the Result column/Target variable ###Check for A1 and A9

```
par(mfrow=c(2,2))
hist(data$A1)
hist(data$A1*data$R1)
hist(data$A9)
hist(data$A9*data$R1)
```



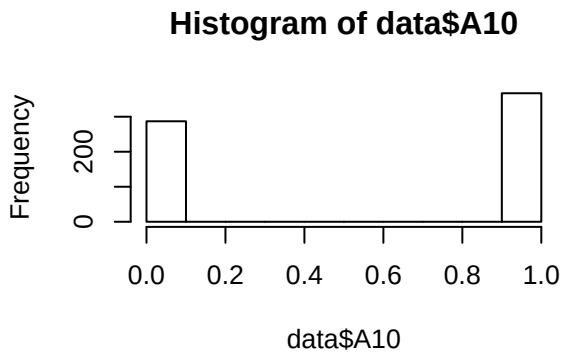


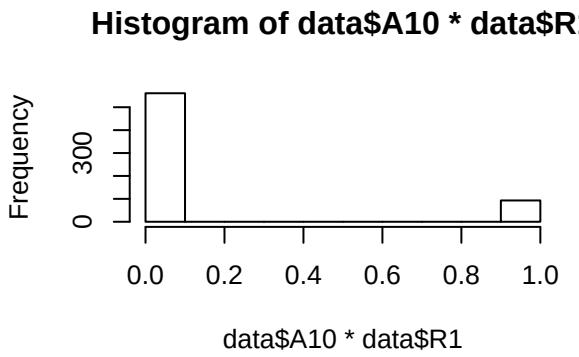


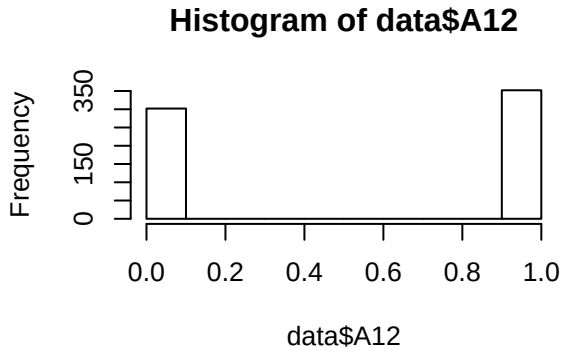


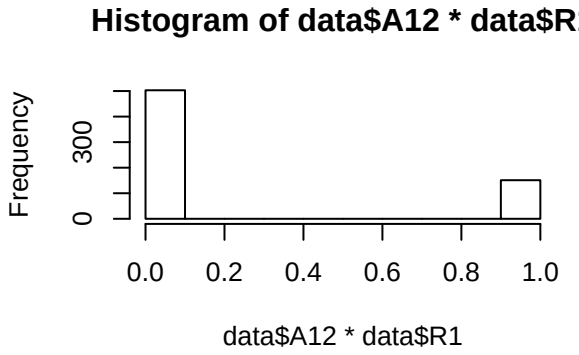
Check for A10 and A12

```
par(mfrow=c(2,2))
hist(data$A10)
hist(data$A10*data$R1)
hist(data$A12)
hist(data$A12*data$R1)
```









###Conclusion: A9 seems to be correlated well with the Target Dataset.

Lets find by how much A9 is correlated with R1.

```
#Clear the graphics back to normal
par(mfrow=c(1,1), mfcol=c(1,1))

#Definitely A9 seems more correlated with the Output variable.
#Lets find out by how much
A911 = nrow(data[which(data$A9 == 1 & data$R1 == 1), c("A9", "R1")])
A901 = nrow(data[which(data$A9 == 0 & data$R1 == 1), c("A9", "R1")])
A910 = nrow(data[which(data$A9 == 1 & data$R1 == 0), c("A9", "R1")])
A900 = nrow(data[which(data$A9 == 0 & data$R1 == 0), c("A9", "R1")])
```

Conclusion:

Below is the accuracy of a classifier that will just return A9 value as the prediction for R1

```
(A911+A900)/(A911 + A900 + A910 + A901)
```

```
## [1] 0.8623853
```

HOMEWORK Question 2.1

Q: 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.

ANSWER

EXAMPLE-1

A job interview is a classification process that accepts / rejects a candidate for a particular job requirement. It is a **Binary classification** process. Potential features:

1. Does Candidate meet the required eductional qualifications?
2. How many years of prior experience does he/she have?
3. Does he/she possess certifications in relevant areas?
4. Current Salary of the Candidate
5. Hike expected by candidate?
6. Mean of Salary drawn by similar employees already in the company
7. Does the candidate posses required soft-skills to handle the job?
8. How good is his/her communication skills in a grade of 1 to 10?
9. Expertise is Relevant Technical Area #1 in a scale of 1 to 10
10. & So on.

EXAMPLE-2

While Job interview is a binary classification exercise, "appraisals" can be thought of as a **multi-class classification** exercise. An employee may be mapped to different buckets like "Excels", "Exceeds expectation", "Meets Expectation", "Needs improvement" and so on. Potential Features:

1. Customer Feedback
2. Goal Achievements
3. Number of Slip-ups
4. Peer, Team feedback
5. Business KPI metrics

REAL-TIME EXAMPLE-1

Hey! Anonymous Peer Reviewer! Currently my exercises are being graded by you! And that..., My friend! is a real-time example of Classification. Since there are several grades, this is a fine example of **multi-class classification**. Potential Features

1. All Exercises complete or attempted?
2. Correctness for each exercise
3. Extra work done
4. Any advanced R features used?
5. Is the work well presented?

FICTIONAL EXAMPLE-1

In Fiction, the Sorting Hat of Hogwarts (Harry Potter), that classifies people into the 4 houses, is a clean example of classification! Though feature vectors may be known only to Dr. Dumbledore. Muggles like me stand no chance.

HOMEWORK Question 2.2-1

Question: Using the support vector machine function ksvm contained in the R package kernlab, find a good classifier for this data. Show the equation of your classifier, and how well it classifies the data points in the full data set. (Donâ t worry about test/validation data yet; weâ ll cover that topic soon.)

```
#
# We will find a classifier for C=1 and print its equation.
# The "scale" parameter can be an indivudal list specifying scale or not for each feature.
# We use that not to scale the binary categorical variables
# (which also is default behavior according to ksvm doc)
#

m1 <- ksvm(R1 ~ .,
           data=data,
           scaled=c(FALSE,TRUE, TRUE,TRUE, FALSE, FALSE, TRUE,FALSE, TRUE, TRUE),
           type = "C-svc", kernel="vanilladot", C=1)
```

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

```
#SVM Model information for C=1
m1
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 194
##
## Objective Function Value : -180.9056
## Training error : 0.136086
```

```
a <- colSums(m1@xmatrix[[1]] * m1@coef[[1]])
a
```

```
##            A1            A2            A3            A8            A9
## -0.0025233112 -0.0008210401 -0.0016420816  0.0033391989  2.0122532246
##           A10           A11           A12           A14           A15
## -0.0063323612  0.0001025965 -0.0011621332 -0.0013864688  0.1064558021
```

```
# calculate a0
a0 = (m1@b)
a0 = -a0
a0
```

```
## [1] -0.9893221
```

```
equation = paste(a[[1]], "*", names(a[1]), "+\n",
                 a[[2]], "*", names(a[2]), "+\n",
                 a[[3]], "*", names(a[3]), "+\n",
                 a[[4]], "*", names(a[4]), "+\n",
                 a[[5]], "*", names(a[5]), "+\n",
                 a[[6]], "*", names(a[6]), "+\n",
                 a[[7]], "*", names(a[7]), "+\n",
                 a[[8]], "*", names(a[8]), "+\n",
                 a[[9]], "*", names(a[9]), "+\n",
                 a[[10]], "*", names(a[10]), "+\n",
                 a0)


cat(paste("The Equation is: ", equation))
```

```
## The Equation is:  -0.00252331124527218 * A1 +
##  -0.000821040109437116 * A2 +
##  -0.00164208163905245 * A3 +
##  0.0033391988884921 * A8 +
##  2.01225322461612 * A9 +
##  -0.00633236120773255 * A10 +
##  0.000102596541931319 * A11 +
##  -0.00116213318013806 * A12 +
##  -0.00138646876159411 * A14 +
##  0.106455802076176 * A15 +
##  -0.989322106295876
```

Factor the Data

The Data contains 4 categorical predictors and the predicted target is also a binary categorical variable. In order for algorithms to treat them as Categorical variables, we need to explicitly to mark the data so.

We will use the same curated data frame below for KNN and other exercises too.

NOTE: In the TA video, this step is not considered. Possibly because this can overwhelm beginners. But it is perfectly natural for R programmers to do this step.

```
#
# Before we attempt to experiment with different values of C,
# we will take an extra cautious step to convert all categorical columns
# into "factors" so that R functions are careful to not misinterpret
# them as numeric columns
# i.e. is.numeric() on these factor columns will return FALSE
#      and is.factor() returns TRUE
#
```

```
curated_data = data
curated_data$A1 = factor(data$A1)
curated_data$A9 = factor(data$A9)
curated_data$A10 = factor(data$A10)
curated_data$A12 = factor(data$A12)
curated_data$R1 = factor(data$R1)
```

Helper Functions for SVM

```
#
# Helper function to create a geometric progress of Lambdas/Cs
#
makeGeomSequence <- function (base, multiplier, count) {
  r = 1:count
  i = 1
  start = base
  while (i <= count) {
    r[i] = start
    start = start * multiplier
    i = i+ 1
  }
  return(matrix(r, length(r), 1))
}

#
# Helper function to create a SVM model as a function of Soft-classifier constant
#
createSVMModel <- function(LambdaSoft, train_data=curated_data, ker="vanilladot", sig=1.0) {
  kpar_value = "automatic"
  if (ker == "rbfdot-forcesigma") {
    kpar_value = list(sigma=sig)
    ker = "rbfdot"
  }

  m <- ksvm(R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15, data=train_data,
       scaled=c(FALSE,TRUE, TRUE,TRUE, FALSE, FALSE, TRUE,FALSE, TRUE, TRUE, FALSE),
       type = "C-svc", kernel=ker, kpar=kpar_value, C=LambdaSoft)
  return(m)
}

getC <- function(m) {
  return(m$C)
}

getAccuracy <- function(model, test_data=curated_data) {
  pred <- predict(model,
                  test_data[,c("A1", "A2", "A3", "A8", "A9", "A10", "A11", "A12", "A14", "A15")])
  accuracy <- sum(pred == test_data[,"R1"]) / nrow(test_data)
  return (accuracy)
}
```

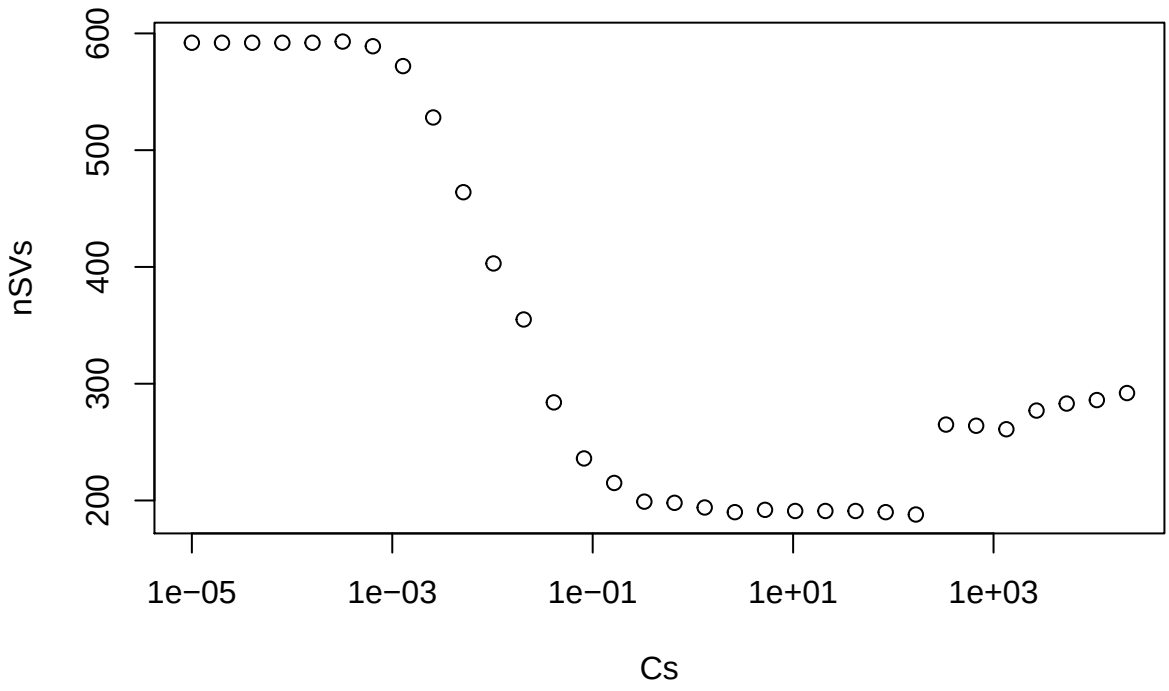
Find Best C for Linear Model

```
#
# Create a list of models, each corresponding to a particular value of Lambda/C
#
models = apply(makeGeomSequence(0.00001,2,32), 1, createSVMModel, train_data = curated_data)
```

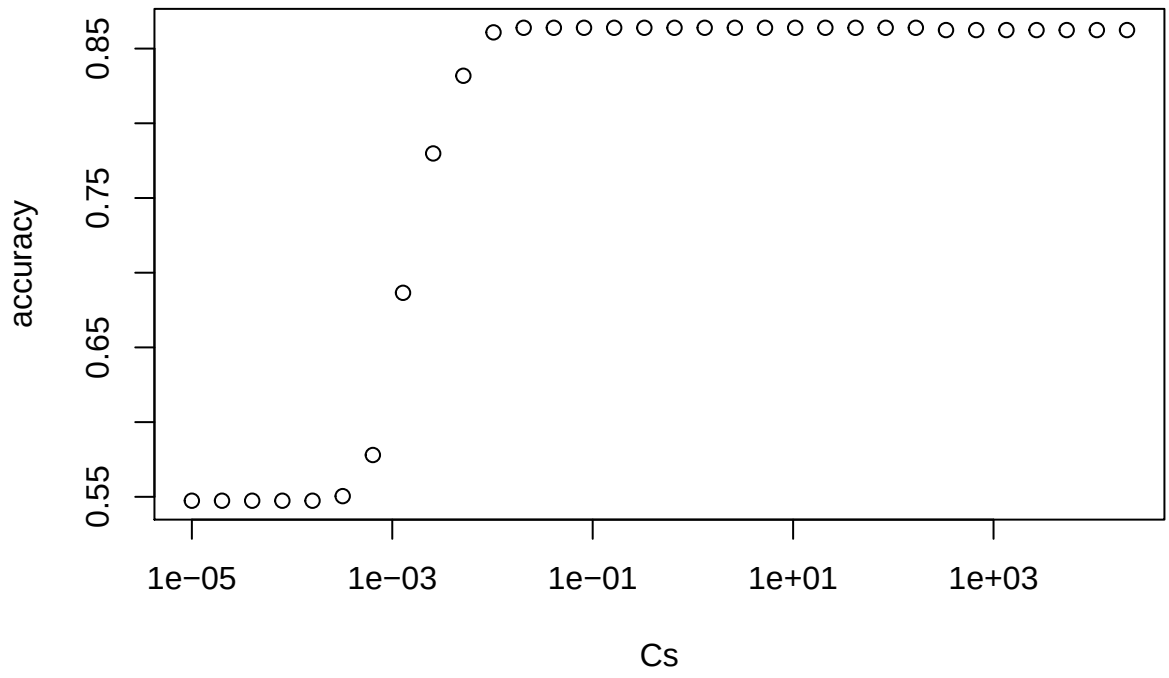
```
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters
##  Setting default kernel parameters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

```
#
# Extract parameters of our interest from the generated models
#
nSVs     = lapply(models, slot, name="nSV")
params   = lapply(models, slot, name="param")
Cs       = lapply(params, getC)
accuracy = lapply(models, getAccuracy)


#
# Let us plot the number of support vectors for the various models
# and see how it looks like
```

```r
# With increase in C, margin will shrink and support vectors will come down.
#
plot(Cs, nSVs, log="x")
```




```r
#
# Let us plot the accuracy of the models and see how it looks like
#
plot(Cs, accuracy, log="x")
```




```
print("RESULTS........")
```

```
## [1] "RESULTS........"
```

```
# C value with good accuracy for VanillaDot kernel and its accuracy value and num Support
print("The Best C Value, its accuracy and support vectors")
```

```
## [1] "The Best C Value, its accuracy and support vectors"
```

```
Cs[[which.max(accuracy)]]
```

```
## [1] 0.02048
```

```
accuracy[[which.max(accuracy)]]
```

```
## [1] 0.8639144
```

```
nSVs[[which.max(accuracy)]]
```

```
## [1] 355
```

```r
print("Just a curiosity to look at model with least support vectors... How good is that? ")
```

```
## [1] "Just a curiosity to look at model with least support vectors... How good is that? "
```

```r
# C value with least amount of support vectors for VanillaDot kernel and its accuracy and num Support
Cs[[which.min(nSVs)]]
```

```
## [1] 167.7722
```

```r
accuracy[[which.min(nSVs)]]
```

```
## [1] 0.8639144
```

```r
nSVs[[which.min(nSVs)]]
```

```
## [1] 188
```

Let us take a look at predictions by models with different C/Lambda/Soft-classifier-constant values

```r
#
# Let us look at some predictions
#
predict(models[[1]], curated_data[,1:10])
```

```
##   [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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 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
## [106] 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
## [141] 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
## [176] 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
## [211] 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
## [246] 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
## [281] 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
## [316] 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
## [351] 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
## [386] 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
## [421] 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
## [456] 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
## [491] 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
## [526] 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
## [561] 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
## [596] 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
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[1]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.547400611620795"
```

```
predict(models[[3]], curated_data[,1:10])
```

```
##   [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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 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
## [106] 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
## [141] 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
## [176] 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
## [211] 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
## [246] 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
## [281] 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
## [316] 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
## [351] 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
## [386] 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
## [421] 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
## [456] 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
## [491] 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
## [526] 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
## [561] 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
## [596] 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
## [631] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[3]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.547400611620795"
```

```
predict(models[[10]], curated_data[,1:10])
```

```
##   [1] 1 1 0 1 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
##  [71] 0 1 1 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 1 1 1 1 1 0 1 1 1
## [106] 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 1 0 0 1 1 1 0 0 0 0 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1
## [211] 1 1 1 1 1 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1
## [246] 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
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
## [351] 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
## [386] 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
## [421] 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [456] 0 0 0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1
## [491] 1 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 0 1 1 0 0 0 1 0 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1
## [561] 1 1 1 1 0 0 0 1 1 1 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
## [596] 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
## [631] 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[10]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.831804281345566"
```

```
predict(models[[16]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [351] 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
## [386] 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
## [421] 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
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
## [561] 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
## [596] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## [631] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[16]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.863914373088685"
```

```
predict(models[[24]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [351] 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
## [386] 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
## [421] 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
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
## [561] 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
## [596] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
## [631] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[24]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.863914373088685"
```

```
predict(models[[32]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [351] 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 0
## [386] 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 0
## [421] 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 0
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
## [561] 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
## [596] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(models[[32]])))
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.862385321100917"
```

Conclusion

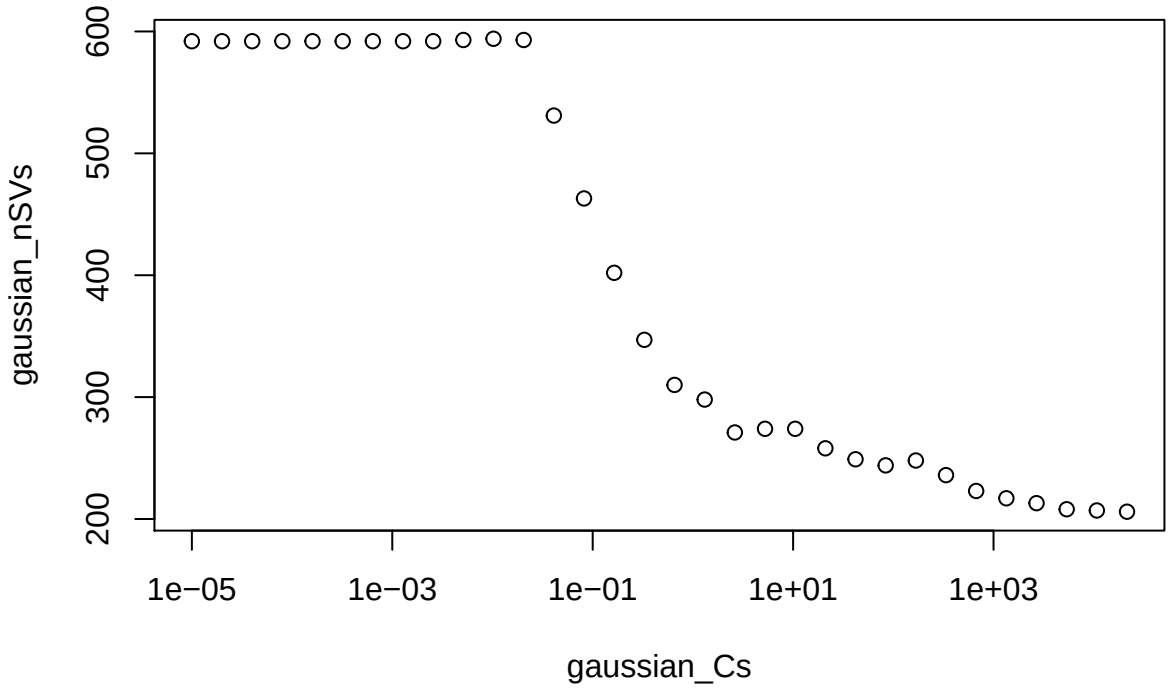
The Linear Kernel's accuracy stagnates after an initial jump in accuracy. The Best C found was 0.02 approx and any other C greater than that produces a very similar accuracy number. Even if we increase C, we don't see any overfitting that is happening with the Linear kernel.

OPTIONAL HOMEWORK 2.2, Question 2

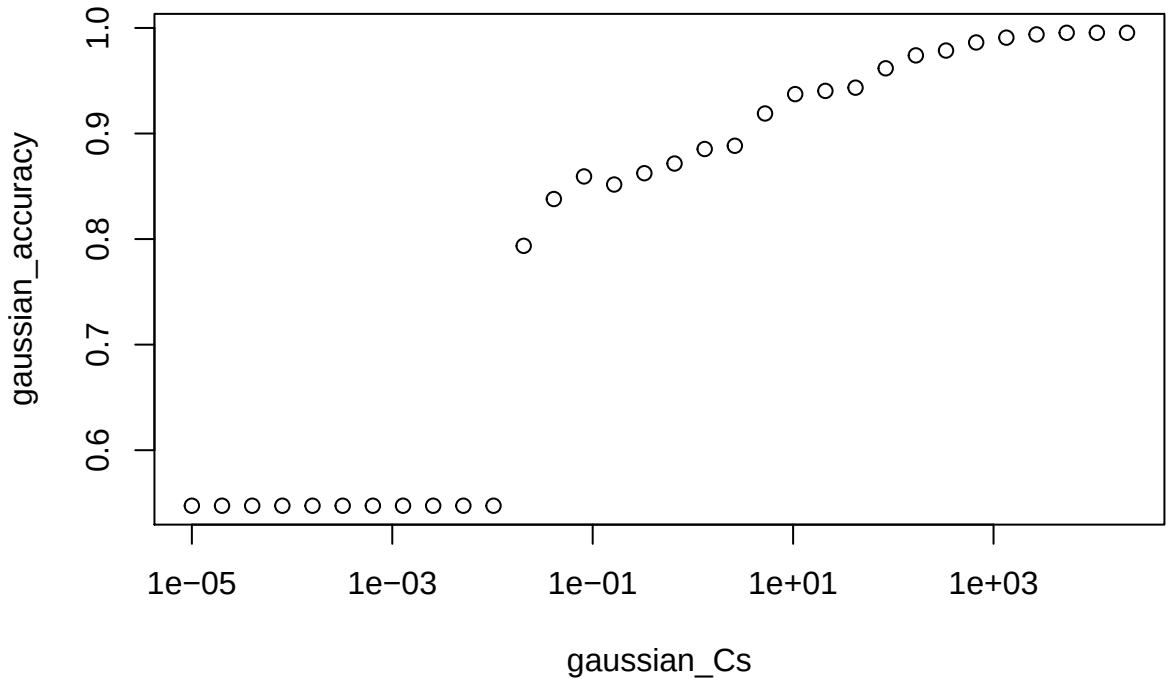
Trying out Gaussian Kernel

```
#
# Create a list of Gaussian SVM models, each corresponding to a particular value of Lambda/C
#
gaussian_models = apply(makeGeomSequence(0.00001,2,32), 1, createSVMModel, ker="rbfdot", train_data=cura
gaussian_nSVs = lapply(gaussian_models, slot, name="nSV")
gaussian_params = lapply(gaussian_models, slot, name="param")
gaussian_errors = lapply(gaussian_models, slot, name="error")
```

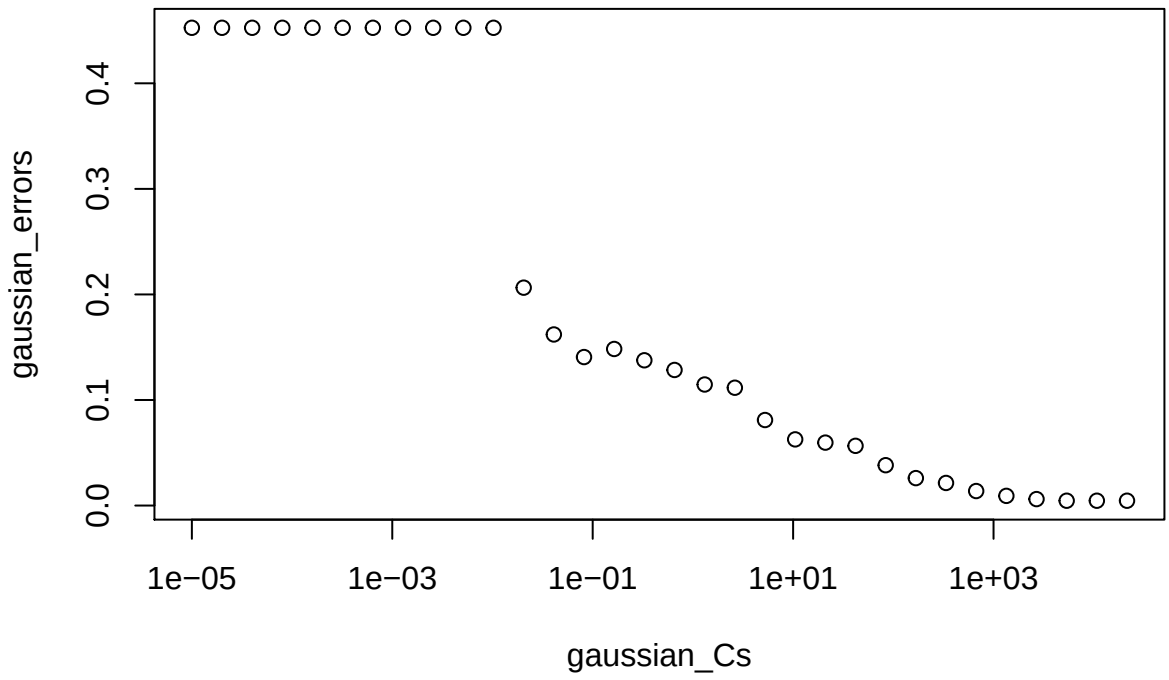
```
gaussian_Cs = lapply(gaussian_params, getC)
gaussian_accuracy = lapply(gaussian_models, getAccuracy)

#
# As we increase C, num support vectors steadily come down
# i.e. the Margin is shrinking - which is to be expected.
# NOTE: This was not the case for vanilladot kernel
#
plot(gaussian_Cs, gaussian_nSVs, log="x")
```




```
#
# Accuracy increases steadily with C.
# But we dont have any test/validation sets and are running this on the whole data.
# Hence accuracy shows up high inspite of possible overfitting
#
plot(gaussian_Cs, gaussian_accuracy, log="x")
```




```
#
# In the Plot below, one can see that the errors keep dropping as we increase C
# We are clearly overfitting the data as we penalize the misclassifications heavily
# Since we dont use validation/training and test sets, we possibly are not seeing the
# effect of unseen data on the model.
#
plot(gaussian_Cs, gaussian_errors, log="x")
```




```
#
# Accuracy is 100%. The full data has been learnt now.
#
print("Accuracy of Gaussian for best C while keeping sigma at 1.0")
```

```
## [1] "Accuracy of Gaussian for best C while keeping sigma at 1.0"
```

```
gaussian_accuracy[[which.max(gaussian_accuracy)]]
```

```
## [1] 0.9954128
```

```
gaussian_Cs[[which.max(gaussian_accuracy)]]
```

```
## [1] 5368.709
```

```
gaussian_nSVs[[which.max(gaussian_accuracy)]]
```

```
## [1] 208
```

```
#
# NOTE: Why only sigma 1.0?
# ========================
```

```
# It is possibly wise to check the effect of "sigma" while keeping C constant
# and prepare the Grid matrix
# But without training and test set it does not make much difference.
# So, not doing it here.
#
```

Viewing Gaussian Kernel Predictions

```
predict(gaussian_models[[3]], curated_data[,1:10])
```

```
##   [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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 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
## [106] 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
## [141] 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
## [176] 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
## [211] 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
## [246] 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
## [281] 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
## [316] 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
## [351] 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
## [386] 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
## [421] 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
## [456] 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
## [491] 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
## [526] 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
## [561] 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
## [596] 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
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(gaussian_models[[3]]]
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.547400611620795"
```

```
predict(gaussian_models[[10]], curated_data[,1:10])
```

```
##   [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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [71] 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
## [106] 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
## [141] 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
## [176] 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
## [211] 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
## [246] 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
## [281] 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
## [316] 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
## [351] 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
## [386] 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
```

```
## [421] 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
## [456] 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
## [491] 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
## [526] 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
## [561] 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
## [596] 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
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(gaussian_models[[10]]
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.547400611620795"
```

```
predict(gaussian_models[[16]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [281] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0
## [316] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
## [351] 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
## [386] 0 0 0 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
## [421] 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
## [456] 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1
## [561] 1 1 1 1 0 1 1 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
## [596] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(gaussian_models[[16]]
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.862385321100917"
```

```
predict(gaussian_models[[28]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 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
## [106] 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [281] 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 1 1 0 0 1 1 1 0 0 0 0
```

```
## [316] 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 0
## [351] 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 0
## [386] 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 0
## [421] 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 0
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 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 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [561] 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
## [596] 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 0
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(gaussian_models[[28]]
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.990825688073395"
```

```
predict(gaussian_models[[32]], curated_data[,1:10])
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [36] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 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 0
## [106] 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [176] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [211] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [246] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [281] 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 1 1 1 0 1 1 1 0 0 0 0
## [316] 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 0
## [351] 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 0
## [386] 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 0
## [421] 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 0
## [456] 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [491] 1 1 1 1 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 1 1 1 1 1 1 1
## [526] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [561] 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
## [596] 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 0
## [631] 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
## Levels: 0 1
```

```
print(paste("The accuracy of the predictions above on full dataset= ", getAccuracy(gaussian_models[[32]]
```

```
## [1] "The accuracy of the predictions above on full dataset=  0.995412844036697"
```

Conclusion

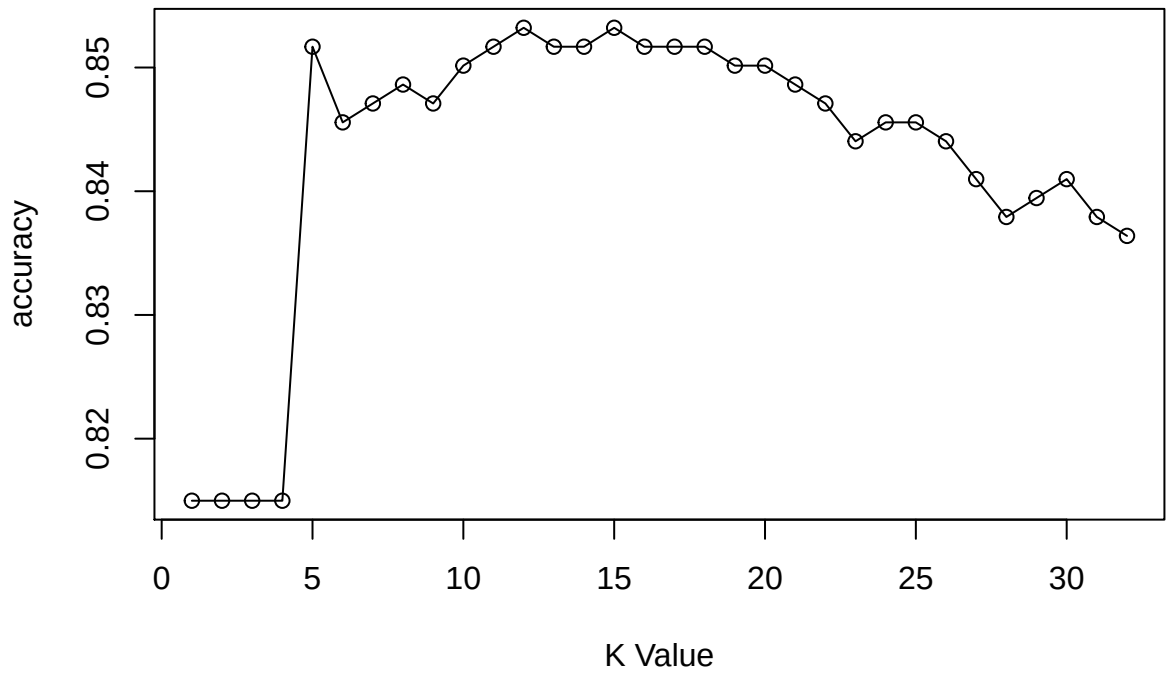
Higher C values are resulting in overfitting for Gaussian kernels. We have still not tuned Gamma yet. We will do that when we have cross-validation and test sets.

HOMEWORK 2.2, Question 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. Do not forget to scale the data (scale=TRUE in kknn).

In the TA video, K-NN prediction is used as a "regression" model. But I choose classification here by marking the categorical variables as Factors. "curated_data" does just that. So, this code will **not** need the 0.5 addition and subsequent rounding using as.integer etc.

```
evaluateKNNAccuracy = function(kvalue) {
  right = 0
  for (i in 1:nrow(curated_data)) {
    m = kknn(formula = R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15,
             train=curated_data[-i,],
             test=curated_data[i,],
             k=kvalue, scale=TRUE)
    p = predict(m)
    a = curated_data[i, "R1"]
    if (p == a) {
      right = right + 1
    }
  }
  return (right/(nrow(curated_data)))
}

accuracy = lapply(1:32, evaluateKNNAccuracy)
plot(1:32, accuracy, xlab="K Value", type="o")
```




```
print("Max Accuracy: ")
```

```
## [1] "Max Accuracy: "
```

```
accuracy[which.max(accuracy)]
```

```
## [[1]]
## [1] 0.853211
```

```
print("K-Value Corresponding to Max Accuracy (bestK): ")
```

```
## [1] "K-Value Corresponding to Max Accuracy (bestK): "
```

```
bestK = which.max(accuracy)
bestK
```

```
## [1] 12
```

Conclusion

We found that by varying K (which is the number of neighbors to consider), we were able to increase accuracy. For this exercise, we have used test-set with only 1 data point. Very similar to 654-fold cross validation.

HOMEWORK 3.1 (a)

Question: Using the same data set (credit_card_data.txt or credit_card_data-headers.txt) as in Question 2.2, use the ksvm or kknn function to find a good classifier: (a) using cross-validation (do this for the k-nearest-neighbors model; SVM is optional); and (b) splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

We use a 16-fold CV here using cv.kknn

```
cvm = cv.kknn(formula = R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15,
              data=curated_data, kcv=16, k=bestK, scale=TRUE)
cv_accuracy = nrow(cvm[[1]][which(cvm[[1]][,"yhat"] == cvm[[1]][,"y"]),]) / nrow(curated_data)
print("Knn accuracy with 16-fold classifcation while holding k at bestK value")
```

```
## [1] "Knn accuracy with 16-fold classifcation while holding k at bestK value"
```

```
cv_accuracy
```

```
## [1] 0.8425076
```

HOMEWORK 3.1 (a) - 2nd Implementation

This is a second implementation of Question 3.1 by implementing Cross-validation manually. We use a 10-fold CV here.

NOTE: This covers both KSVM and K-NN

```
#
# 3.1 (a) looks too simplistic.
# So, Lets hand-code Cross-validation logic ourselves to see how it is working
# Below is 2nd implementation of 3.1 (a)
#

#
# Helper function: Given a list of dataframes, it simply constructs
# a grand dataframe by combining
# all the data frames one above another
#
coalesceDataFrames <- function(listOfDataFrames, exclude=0) {

  grand = data.frame()
  for (i in 1:length(listOfDataFrames)) {
    if (i != exclude) {
      grand = rbind(grand, listOfDataFrames[[i]])
    }
  }
  return (grand)
}

#
# Helper function: This function will input the data and the number of folds
```

```
# and then return a list of Training and Test Datasets that can be used for
# Cross Validation.
#
createCVTrainingSet <- function(input_data, nFolds) {

    foldLength = floor(nrow(input_data)/nFolds)
    remnant    = nrow(input_data) - (foldLength*nFolds)

    #
    # Lets first partition "curated_data" into "nFolds" number of sub data-frames
    #
    folds = list(rep(0,nFolds))
    temp_data = input_data
    for (i in 1:nFolds) {
      f = sample(nrow(temp_data), foldLength, replace=FALSE)
      folds[[i]] = temp_data[f,]
      temp_data = temp_data[-f,]
    }

    #print(paste("nrow of temp_data before adding remnant = ", nrow(temp_data)))

    if (remnant > 0) {
      for (i in 1:remnant) {
        folds[[i]] = rbind(folds[[i]], temp_data[i,])
      }
    }

    #
    # Now "folds" is a list that holds the sub-data frames of "curated_data"
    # Now, we need to pair them as "nFolds" number of training and validation sets
    #
    cv = list(rep(0, nFolds))
    for (i in 1:nFolds) {
      validationSet = folds[[i]]
      trainingSet   = coalesceDataFrames(folds, exclude=i)
      cv[[i]] = list(train_data=trainingSet, validation_data=validationSet)
    }

    return(cv)
}


#
# HOMEWORK 3.1 (a) - 2nd Implementation - Hand-coded CV implementation
#
# We will now implement cross-validation by hand. cv.kknn does not return the actual model
# But, we just used it to check what accuracy that it is giving.
# For this exercise, we will use cross-validation and pick the model based on "accuracy" in the
# validation data.
# For this exercise, we still don't have to evaluate "test" data. We will leave that for the next
# exercise
#
```

```
CVSET           = createCVTrainingSet(curated_data, 10)
CV_KNN_MODELS   = list(rep(0,length(CVSET)))
CV_KNN_ACCURACY = list(rep(0,length(CVSET)))
CV_KSVM_MODELS   = list(rep(0,length(CVSET)))
CV_KSVM_ACCURACY = list(rep(0,length(CVSET)))

for(i in 1:length(CVSET)) {
  m = kknn(formula = R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15, train=CVSET[[i]]$train_data,
           test=CVSET[[i]]$validation_data, k=bestK, scale=TRUE)
  pred = predict(m)
  actual = CVSET[[i]]$validation_data[,"R1"]
  CV_KNN_MODELS[[i]] = m
  CV_KNN_ACCURACY[[i]] = sum(pred==actual)/length(actual)

  m = createSVMModel(1.0, train_data=CVSET[[i]]$train_data, ker = "rbfdot") # For C=1.0 and Gaussian Ke
  #pred = predict(m, CVSET[[i]]$validation_data[,c("A1", "A2", "A3", "A8", "A9", "A10", "A11", "A12", "
  #accuracy = sum(pred == CVSET[[i]]$validation_data[,"R1"]) / length(actual)
  accuracy = getAccuracy(m, test_data = CVSET[[i]]$validation_data)
  CV_KSVM_MODELS[[i]] = m
  CV_KSVM_ACCURACY[[i]] = accuracy
}

bestKnnModel = CV_KNN_MODELS[[which.max(CV_KNN_ACCURACY)]]
bestKnnAccuracy = CV_KNN_ACCURACY[[which.max(CV_KNN_ACCURACY)]]
#
# Accuracy of the best KNN model found by cross-validation
#
print("This is the best KNN Accuracy that we got on Validation Set")
```

```
## [1] "This is the best KNN Accuracy that we got on Validation Set"
```

```
bestKnnAccuracy
```

```
## [1] 0.9076923
```

```
bestKsvmModel = CV_KSVM_MODELS[[which.max(CV_KSVM_ACCURACY)]]
bestKsvmAccuracy = CV_KSVM_ACCURACY[[which.max(CV_KSVM_ACCURACY)]]
#
# Accuracy of the best KNN model found by cross-validation
#
print("This is the best KSVM Accuracy that we got on Validation Set (C=1.0, Gaussian)")
```

```
## [1] "This is the best KSVM Accuracy that we got on Validation Set (C=1.0, Gaussian)"
```

```
bestKsvmAccuracy
```

```
## [1] 0.8787879
```

Conclusion

We created several Cross-validation models using hand-coded methods, evaluated them on the corresponding validation set and identified the model with the Best accuracy on its Validation set. This was done for both KNN and KSVM models.

HOMEWORK 3.1 (b)

Question: (b) splitting the data into training, validation, and test data sets (pick either KNN or SVM; the other is optional).

We choose to demonstrate this with both SVM and KNN.

Some recollections from the Lecture: In 3.1 a), We selected the model based on "accuracy on the validation set" and reported the same accuracy. But this accuracy may not be truly reflective of the model's true predictive properties. Due to randomness, we might have got a bit lucky with the split or so. So, to get the real accuracy of the CV model, we will use a test-set to test its accuracy and report that accuracy.

Here is the deal:

1. We will split the data into Training, validation and test.
2. We will build different KSVM classifiers using the Training set for different C and Gamma values.
3. We will select the best C and Gamma value based on accuracy on Validation set
4. Once we select the model, we will run it on Test dataset to report it as "closer to reality accuracy"

NOTE: This execise does not need to run Cross-validation (thats for the earlier exercise). The same was confirmed by the Instructor in the verified learner forums, in the thread titled "Training, Test, Validation Sets"

Split Scheme

```
#split strategy: Rotating splits (as against to randomized splits)
#split scheme: 70% in Training, 15% in Validation, 15% in Testing
#For split function, we will pass a pattern having
# 14/20 for training,
# 3/20 for validation,
# 3/20 for testing
#And then simply repeat the pattern
#NOTE for split below : 654 is not a multiple of 20. Split emits a warning.
#Its just a benign warning.
#The split happens well and adds up to 654 as shown below.
#
splitscheme = split(1:nrow(curated_data), rep(c(1,3,1,2,1,1,1,1,2,1,1,1,1,3,1,1,2,3,1,1),
                    ceiling(nrow(curated_data)/20)))
```

```
## Warning in split.default(1:nrow(curated_data), rep(c(1, 3, 1, 2, 1, 1, 1, :
## data length is not a multiple of split variable
```

```
trl = length(splitscheme$`1`)
vl = length(splitscheme$`2`)
tel = length(splitscheme$`3`)


print("Sum of training + validation + test")
```

```
## [1] "Sum of training + validation + test"
```

```
trl + vl + tel
```

```
## [1] 654
```

```
trainingSet = curated_data[splitscheme$`1`,]
validationSet = curated_data[splitscheme$`2`,]
testSet = curated_data[splitscheme$`3`,]

print("Size of training , validation , test dataset respectively")
```

```
## [1] "Size of training , validation , test dataset respectively"
```

```
nrow(trainingSet)
```

```
## [1] 458
```

```
nrow(validationSet)
```

```
## [1] 98
```

```
nrow(testSet)
```

```
## [1] 98
```

```
#
# Verify how the rotation based split is working by inspecting the "rownames"
# output below.
# Note:
# The split pattern given is : c(1,3,1,2,1,1,1,1,2,1,1,1,1,3,1,1,2,3,1,1)
# (repeats itself until 654)
#
# Training is factor level 1 and hence should have index in the pattern of:
# (1,3,5,6,7,8,10,11,12,13,15,16,19,20), first-set + 20, first-set + 40 and so on.
#
# Test set is factor level 3 and hence should have index in the pattern of:
# (2,14,18),(2,14,18)+20,(2,14,18)+40,...
#
# Validatiion is factor level 2 and hence :
# (4,9,17), (4,9,17)+20, (4,9,17)+40 and so on.
#

rownames(trainingSet)
```

```
##   [1] "1"   "3"   "5"   "6"   "7"   "8"   "10"  "11"  "12"  "13"  "15"
##  [12] "16"  "19"  "20"  "21"  "23"  "25"  "26"  "27"  "28"  "30"  "31"
##  [23] "32"  "33"  "35"  "36"  "39"  "40"  "41"  "43"  "45"  "46"  "47"
##  [34] "48"  "50"  "51"  "52"  "53"  "55"  "56"  "59"  "60"  "61"  "63"
##  [45] "65"  "66"  "67"  "68"  "70"  "71"  "72"  "73"  "75"  "76"  "79"
##  [56] "80"  "81"  "83"  "85"  "86"  "87"  "88"  "90"  "91"  "92"  "93"
##  [67] "95"  "96"  "99"  "100" "101" "103" "105" "106" "107" "108" "110"
```

```
##  [78] "111" "112" "113" "115" "116" "119" "120" "121" "123" "125" "126"
##  [89] "127" "128" "130" "131" "132" "133" "135" "136" "139" "140" "141"
## [100] "143" "145" "146" "147" "148" "150" "151" "152" "153" "155" "156"
## [111] "159" "160" "161" "163" "165" "166" "167" "168" "170" "171" "172"
## [122] "173" "175" "176" "179" "180" "181" "183" "185" "186" "187" "188"
## [133] "190" "191" "192" "193" "195" "196" "199" "200" "201" "203" "205"
## [144] "206" "207" "208" "210" "211" "212" "213" "215" "216" "219" "220"
## [155] "221" "223" "225" "226" "227" "228" "230" "231" "232" "233" "235"
## [166] "236" "239" "240" "241" "243" "245" "246" "247" "248" "250" "251"
## [177] "252" "253" "255" "256" "259" "260" "261" "263" "265" "266" "267"
## [188] "268" "270" "271" "272" "273" "275" "276" "279" "280" "281" "283"
## [199] "285" "286" "287" "288" "290" "291" "292" "293" "295" "296" "299"
## [210] "300" "301" "303" "305" "306" "307" "308" "310" "311" "312" "313"
## [221] "315" "316" "319" "320" "321" "323" "325" "326" "327" "328" "330"
## [232] "331" "332" "333" "335" "336" "339" "340" "341" "343" "345" "346"
## [243] "347" "348" "350" "351" "352" "353" "355" "356" "359" "360" "361"
## [254] "363" "365" "366" "367" "368" "370" "371" "372" "373" "375" "376"
## [265] "379" "380" "381" "383" "385" "386" "387" "388" "390" "391" "392"
## [276] "393" "395" "396" "399" "400" "401" "403" "405" "406" "407" "408"
## [287] "410" "411" "412" "413" "415" "416" "419" "420" "421" "423" "425"
## [298] "426" "427" "428" "430" "431" "432" "433" "435" "436" "439" "440"
## [309] "441" "443" "445" "446" "447" "448" "450" "451" "452" "453" "455"
## [320] "456" "459" "460" "461" "463" "465" "466" "467" "468" "470" "471"
## [331] "472" "473" "475" "476" "479" "480" "481" "483" "485" "486" "487"
## [342] "488" "490" "491" "492" "493" "495" "496" "499" "500" "501" "503"
## [353] "505" "506" "507" "508" "510" "511" "512" "513" "515" "516" "519"
## [364] "520" "521" "523" "525" "526" "527" "528" "530" "531" "532" "533"
## [375] "535" "536" "539" "540" "541" "543" "545" "546" "547" "548" "550"
## [386] "551" "552" "553" "555" "556" "559" "560" "561" "563" "565" "566"
## [397] "567" "568" "570" "571" "572" "573" "575" "576" "579" "580" "581"
## [408] "583" "585" "586" "587" "588" "590" "591" "592" "593" "595" "596"
## [419] "599" "600" "601" "603" "605" "606" "607" "608" "610" "611" "612"
## [430] "613" "615" "616" "619" "620" "621" "623" "625" "626" "627" "628"
## [441] "630" "631" "632" "633" "635" "636" "639" "640" "641" "643" "645"
## [452] "646" "647" "648" "650" "651" "652" "653"
```

```
rownames(validationSet)
```

```
##  [1] "4"   "9"   "17"  "24"  "29"  "37"  "44"  "49"  "57"  "64"  "69"
## [12] "77"  "84"  "89"  "97"  "104" "109" "117" "124" "129" "137" "144"
## [23] "149" "157" "164" "169" "177" "184" "189" "197" "204" "209" "217"
## [34] "224" "229" "237" "244" "249" "257" "264" "269" "277" "284" "289"
## [45] "297" "304" "309" "317" "324" "329" "337" "344" "349" "357" "364"
## [56] "369" "377" "384" "389" "397" "404" "409" "417" "424" "429" "437"
## [67] "444" "449" "457" "464" "469" "477" "484" "489" "497" "504" "509"
## [78] "517" "524" "529" "537" "544" "549" "557" "564" "569" "577" "584"
## [89] "589" "597" "604" "609" "617" "624" "629" "637" "644" "649"
```

```
rownames(testSet)
```

```
##  [1] "2"   "14"  "18"  "22"  "34"  "38"  "42"  "54"  "58"  "62"  "74"
## [12] "78"  "82"  "94"  "98"  "102" "114" "118" "122" "134" "138" "142"
## [23] "154" "158" "162" "174" "178" "182" "194" "198" "202" "214" "218"
```

```
## [34] "222" "234" "238" "242" "254" "258" "262" "274" "278" "282" "294"
## [45] "298" "302" "314" "318" "322" "334" "338" "342" "354" "358" "362"
## [56] "374" "378" "382" "394" "398" "402" "414" "418" "422" "434" "438"
## [67] "442" "454" "458" "462" "474" "478" "482" "494" "498" "502" "514"
## [78] "518" "522" "534" "538" "542" "554" "558" "562" "574" "578" "582"
## [89] "594" "598" "602" "614" "618" "622" "634" "638" "642" "654"
```

SVM

```
bestAccuracy = 0
bestLambda = 0
bestGamma = 0
bestModel = 0

for (lambda in c(0.0001, 0.001, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 5, 10, 50, 100, 1000, 10000)
  for (gamma in c(0.0001, 0.001, 0.01, 0.1, 1, 5, 10, 50, 100, 500, 1000, 10000)) {
    m = createSVMModel(lambda, trainingSet, ker = "rbfdot-forcesigma", sig=gamma)
    accuracy = getAccuracy(m, test_data = validationSet)
    print(paste(lambda, ",", gamma, ",", accuracy))
    if (accuracy > bestAccuracy) {
      bestAccuracy = accuracy
      bestLambda = lambda
      bestGamma = gamma
      bestModel = m
    }
  }
}
```

```
## [1] "1e-04 , 1e-04 , 0.551020408163265"
## [1] "1e-04 , 0.001 , 0.551020408163265"
## [1] "1e-04 , 0.01 , 0.551020408163265"
## [1] "1e-04 , 0.1 , 0.551020408163265"
## [1] "1e-04 , 1 , 0.551020408163265"
## [1] "1e-04 , 5 , 0.551020408163265"
## [1] "1e-04 , 10 , 0.551020408163265"
## [1] "1e-04 , 50 , 0.551020408163265"
## [1] "1e-04 , 100 , 0.551020408163265"
## [1] "1e-04 , 500 , 0.551020408163265"
## [1] "1e-04 , 1000 , 0.551020408163265"
## [1] "1e-04 , 10000 , 0.551020408163265"
## [1] "0.001 , 1e-04 , 0.551020408163265"
## [1] "0.001 , 0.001 , 0.551020408163265"
## [1] "0.001 , 0.01 , 0.551020408163265"
## [1] "0.001 , 0.1 , 0.551020408163265"
## [1] "0.001 , 1 , 0.551020408163265"
## [1] "0.001 , 5 , 0.551020408163265"
## [1] "0.001 , 10 , 0.551020408163265"
## [1] "0.001 , 50 , 0.551020408163265"
## [1] "0.001 , 100 , 0.551020408163265"
## [1] "0.001 , 500 , 0.551020408163265"
## [1] "0.001 , 1000 , 0.551020408163265"
## [1] "0.001 , 10000 , 0.551020408163265"
```

```
## [1] "0.005 , 1e-04 , 0.551020408163265"
## [1] "0.005 , 0.001 , 0.551020408163265"
## [1] "0.005 , 0.01 , 0.551020408163265"
## [1] "0.005 , 0.1 , 0.551020408163265"
## [1] "0.005 , 1 , 0.551020408163265"
## [1] "0.005 , 5 , 0.551020408163265"
## [1] "0.005 , 10 , 0.551020408163265"
## [1] "0.005 , 50 , 0.551020408163265"
## [1] "0.005 , 100 , 0.551020408163265"
## [1] "0.005 , 500 , 0.551020408163265"
## [1] "0.005 , 1000 , 0.551020408163265"
## [1] "0.005 , 10000 , 0.551020408163265"
## [1] "0.01 , 1e-04 , 0.551020408163265"
## [1] "0.01 , 0.001 , 0.551020408163265"
## [1] "0.01 , 0.01 , 0.551020408163265"
## [1] "0.01 , 0.1 , 0.551020408163265"
## [1] "0.01 , 1 , 0.551020408163265"
## [1] "0.01 , 5 , 0.551020408163265"
## [1] "0.01 , 10 , 0.551020408163265"
## [1] "0.01 , 50 , 0.551020408163265"
## [1] "0.01 , 100 , 0.551020408163265"
## [1] "0.01 , 500 , 0.551020408163265"
## [1] "0.01 , 1000 , 0.551020408163265"
## [1] "0.01 , 10000 , 0.551020408163265"
## [1] "0.02 , 1e-04 , 0.551020408163265"
## [1] "0.02 , 0.001 , 0.551020408163265"
## [1] "0.02 , 0.01 , 0.551020408163265"
## [1] "0.02 , 0.1 , 0.653061224489796"
## [1] "0.02 , 1 , 0.551020408163265"
## [1] "0.02 , 5 , 0.551020408163265"
## [1] "0.02 , 10 , 0.551020408163265"
## [1] "0.02 , 50 , 0.551020408163265"
## [1] "0.02 , 100 , 0.551020408163265"
## [1] "0.02 , 500 , 0.551020408163265"
## [1] "0.02 , 1000 , 0.551020408163265"
## [1] "0.02 , 10000 , 0.551020408163265"
## [1] "0.05 , 1e-04 , 0.551020408163265"
## [1] "0.05 , 0.001 , 0.551020408163265"
## [1] "0.05 , 0.01 , 0.571428571428571"
## [1] "0.05 , 0.1 , 0.826530612244898"
## [1] "0.05 , 1 , 0.551020408163265"
## [1] "0.05 , 5 , 0.551020408163265"
## [1] "0.05 , 10 , 0.551020408163265"
## [1] "0.05 , 50 , 0.551020408163265"
## [1] "0.05 , 100 , 0.551020408163265"
## [1] "0.05 , 500 , 0.551020408163265"
## [1] "0.05 , 1000 , 0.551020408163265"
## [1] "0.05 , 10000 , 0.551020408163265"
## [1] "0.1 , 1e-04 , 0.551020408163265"
## [1] "0.1 , 0.001 , 0.551020408163265"
## [1] "0.1 , 0.01 , 0.673469387755102"
## [1] "0.1 , 0.1 , 0.846938775510204"
## [1] "0.1 , 1 , 0.591836734693878"
## [1] "0.1 , 5 , 0.551020408163265"
```

```
## [1] "0.1 , 10 , 0.551020408163265"
## [1] "0.1 , 50 , 0.551020408163265"
## [1] "0.1 , 100 , 0.551020408163265"
## [1] "0.1 , 500 , 0.551020408163265"
## [1] "0.1 , 1000 , 0.551020408163265"
## [1] "0.1 , 10000 , 0.551020408163265"
## [1] "0.2 , 1e-04 , 0.551020408163265"
## [1] "0.2 , 0.001 , 0.561224489795918"
## [1] "0.2 , 0.01 , 0.785714285714286"
## [1] "0.2 , 0.1 , 0.857142857142857"
## [1] "0.2 , 1 , 0.806122448979592"
## [1] "0.2 , 5 , 0.551020408163265"
## [1] "0.2 , 10 , 0.551020408163265"
## [1] "0.2 , 50 , 0.551020408163265"
## [1] "0.2 , 100 , 0.551020408163265"
## [1] "0.2 , 500 , 0.551020408163265"
## [1] "0.2 , 1000 , 0.551020408163265"
## [1] "0.2 , 10000 , 0.551020408163265"
## [1] "0.5 , 1e-04 , 0.551020408163265"
## [1] "0.5 , 0.001 , 0.581632653061224"
## [1] "0.5 , 0.01 , 0.846938775510204"
## [1] "0.5 , 0.1 , 0.86734693877551"
## [1] "0.5 , 1 , 0.816326530612245"
## [1] "0.5 , 5 , 0.551020408163265"
## [1] "0.5 , 10 , 0.551020408163265"
## [1] "0.5 , 50 , 0.551020408163265"
## [1] "0.5 , 100 , 0.551020408163265"
## [1] "0.5 , 500 , 0.551020408163265"
## [1] "0.5 , 1000 , 0.551020408163265"
## [1] "0.5 , 10000 , 0.551020408163265"
## [1] "1 , 1e-04 , 0.551020408163265"
## [1] "1 , 0.001 , 0.663265306122449"
## [1] "1 , 0.01 , 0.877551020408163"
## [1] "1 , 0.1 , 0.86734693877551"
## [1] "1 , 1 , 0.877551020408163"
## [1] "1 , 5 , 0.826530612244898"
## [1] "1 , 10 , 0.581632653061224"
## [1] "1 , 50 , 0.551020408163265"
## [1] "1 , 100 , 0.551020408163265"
## [1] "1 , 500 , 0.551020408163265"
## [1] "1 , 1000 , 0.551020408163265"
## [1] "1 , 10000 , 0.551020408163265"
## [1] "5 , 1e-04 , 0.581632653061224"
## [1] "5 , 0.001 , 0.86734693877551"
## [1] "5 , 0.01 , 0.877551020408163"
## [1] "5 , 0.1 , 0.877551020408163"
## [1] "5 , 1 , 0.877551020408163"
## [1] "5 , 5 , 0.816326530612245"
## [1] "5 , 10 , 0.581632653061224"
## [1] "5 , 50 , 0.551020408163265"
## [1] "5 , 100 , 0.551020408163265"
## [1] "5 , 500 , 0.551020408163265"
## [1] "5 , 1000 , 0.551020408163265"
## [1] "5 , 10000 , 0.551020408163265"
```

```
## [1] "10 , 1e-04 , 0.653061224489796"
## [1] "10 , 0.001 , 0.877551020408163"
## [1] "10 , 0.01 , 0.877551020408163"
## [1] "10 , 0.1 , 0.877551020408163"
## [1] "10 , 1 , 0.86734693877551"
## [1] "10 , 5 , 0.816326530612245"
## [1] "10 , 10 , 0.581632653061224"
## [1] "10 , 50 , 0.551020408163265"
## [1] "10 , 100 , 0.551020408163265"
## [1] "10 , 500 , 0.551020408163265"
## [1] "10 , 1000 , 0.551020408163265"
## [1] "10 , 10000 , 0.551020408163265"
## [1] "50 , 1e-04 , 0.86734693877551"
## [1] "50 , 0.001 , 0.877551020408163"
## [1] "50 , 0.01 , 0.86734693877551"
## [1] "50 , 0.1 , 0.86734693877551"
## [1] "50 , 1 , 0.857142857142857"
## [1] "50 , 5 , 0.826530612244898"
## [1] "50 , 10 , 0.581632653061224"
## [1] "50 , 50 , 0.551020408163265"
## [1] "50 , 100 , 0.551020408163265"
## [1] "50 , 500 , 0.551020408163265"
## [1] "50 , 1000 , 0.551020408163265"
## [1] "50 , 10000 , 0.551020408163265"
## [1] "100 , 1e-04 , 0.877551020408163"
## [1] "100 , 0.001 , 0.877551020408163"
## [1] "100 , 0.01 , 0.86734693877551"
## [1] "100 , 0.1 , 0.826530612244898"
## [1] "100 , 1 , 0.857142857142857"
## [1] "100 , 5 , 0.826530612244898"
## [1] "100 , 10 , 0.581632653061224"
## [1] "100 , 50 , 0.551020408163265"
## [1] "100 , 100 , 0.551020408163265"
## [1] "100 , 500 , 0.551020408163265"
## [1] "100 , 1000 , 0.551020408163265"
## [1] "100 , 10000 , 0.551020408163265"
## [1] "1000 , 1e-04 , 0.877551020408163"
## [1] "1000 , 0.001 , 0.86734693877551"
## [1] "1000 , 0.01 , 0.857142857142857"
## [1] "1000 , 0.1 , 0.806122448979592"
## [1] "1000 , 1 , 0.836734693877551"
## [1] "1000 , 5 , 0.826530612244898"
## [1] "1000 , 10 , 0.581632653061224"
## [1] "1000 , 50 , 0.551020408163265"
## [1] "1000 , 100 , 0.551020408163265"
## [1] "1000 , 500 , 0.551020408163265"
## [1] "1000 , 1000 , 0.551020408163265"
## [1] "1000 , 10000 , 0.551020408163265"
## [1] "10000 , 1e-04 , 0.877551020408163"
## [1] "10000 , 0.001 , 0.86734693877551"
## [1] "10000 , 0.01 , 0.877551020408163"
## [1] "10000 , 0.1 , 0.806122448979592"
## [1] "10000 , 1 , 0.806122448979592"
## [1] "10000 , 5 , 0.826530612244898"
```

```
## [1] "10000 , 10 , 0.581632653061224"
## [1] "10000 , 50 , 0.551020408163265"
## [1] "10000 , 100 , 0.551020408163265"
## [1] "10000 , 500 , 0.551020408163265"
## [1] "10000 , 1000 , 0.551020408163265"
## [1] "10000 , 10000 , 0.551020408163265"
```

```
print("Best Parameters for SVM Classifier:")
```

```
## [1] "Best Parameters for SVM Classifier:"
```

```
print(paste("     Validation Accuracy: ", bestAccuracy))
```

```
## [1] "     Validation Accuracy:  0.877551020408163"
```

```
print(paste("     Lambda: ", bestLambda))
```

```
## [1] "     Lambda:  1"
```

```
print(paste("     Gamma: ", bestGamma))
```

```
## [1] "     Gamma:  0.01"
```

```
# Time to report accuracy of the best model on the "Test" data
bestAccuracyOnTest = getAccuracy(bestModel, test_data = testSet)
print("Test set Accuracy of Best Model ")
```

```
## [1] "Test set Accuracy of Best Model "
```

```
bestAccuracyOnTest
```

```
## [1] 0.877551
```

KNN

Do the same on KNN model too! Iterate over various k, select best model using validation set. After selecting the best model, try it on the Testing set and report "accuracy"

```
bestKNNAccuracy = 0
bestKValueFoundByValidationSet = 0

for (kvalue in 1:50) {
  m = kknn(formula = R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15,
           train=trainingSet,
           test=validationSet, k=kvalue, scale=TRUE)
 pred = predict(m)
 actual = validationSet[,"R1"]
 thisAccuracy = sum(pred==actual)/length(actual)
```

```
 if (thisAccuracy > bestKNNAccuracy) {
   bestKNNAccuracy = thisAccuracy
   bestKValueFoundByValidationSet = kvalue
 }
}

print("Best Parameters for KNN  Classifier selected by Validation Set:")
```

```
## [1] "Best Parameters for KNN  Classifier selected by Validation Set:"
```

```
print(paste("     Validation Accuracy: ", bestKNNAccuracy))
```

```
## [1] "     Validation Accuracy:  0.86734693877551"
```

```
print(paste("     Best K Value: ", bestKValueFoundByValidationSet))
```

```
## [1] "     Best K Value:  12"
```

```
#Time to report accuracy on Test data.
m = kknn(formula = R1 ~ A1+A2+A3+A8+A9+A10+A11+A12+A14+A15,
     train=trainingSet,
     test=testSet, k=bestKValueFoundByValidationSet, scale=TRUE)
pred = predict(m)
actual = testSet[,"R1"]
testKNNAccuracy = sum(pred==actual)/length(actual)
print(paste("Test Accuracy of selected model = ", testKNNAccuracy))
```

```
## [1] "Test Accuracy of selected model =  0.887755102040816"
```

Conclusion

We created a training set, a validation set and a test-set in a 70,15,15 proportion. We trained several models on the training set by varying hyper-parameters, C and Gamma (soft-classifier constant and Gaussian kernel bandwidth) for SVM and the K value for KNN and evaluated them on the Validation set. We chose the model and hence the hyperparameters that had best performance on the Validation set. We then tested the model on the Testing set and reported the accuracy of that model on the Test set. The Testing accuracy appears same or slightly better than the Validation accuracy for 70,15,15 split. Earlier, I had used 60%, 20% and 20% split and observed that "test" accuracy decreased compared to "validation" accuracy - which is one of the things that was discussed during the lecture as well. I guess it is safe to say that training with more data can create better models.

Thanks

THANKS to Professor, EdX, Instructors and GATech. THANKS to all Students and Peer Reviewers for your time! All is well; that ends well!