Code

SVM and KNN Tuning on Credit Card Applications

Mike Huang

May 21, 2017

Assignment Description

The files credit_card_data.txt and credit_card_data-headers.txt contains a dataset of anonymized credit card applications which was either approved(1) or denied(0). This set contains 6 continuous and 4 binary features which has been cleaned up so that it does not contain categorical variables and does not have data points with missing values. The original dataset can be found on the UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets/Credit+Approval).

- Part 1 of our assignment is to use ksvm to find a good classifier for this data and showing the equation associated with the classification.
- Part 2 of our assignment requires the use the kknn to show how well it classifies the data points
- Part 3 of our assignment allows us to practice the use of cross-validation on ksvm or kknn

Data Import and Preprocessing

Before running the algorithms for the homework assignment, I wanted to import, process, and visualize the data so I can get an understanding of what the data looks like and provide me with better intuition as I go forward with future datasets.

Define Libraries

Required libraries:

- kernlab package (https://cran.r-project.org/web/packages/kernlab/kernlab.pdf) - This assignment requires the use of ksvm to practice support vector machine tuning
- kknn package (https://cran.r-project.org/web/packages/kknn/kknn.pdf) - This assignment requries the use of kknn to practice k-nearest neighbor tuning

My Additions:

- caret (https://topepo.github.io/caret/index.html) - A very robust tool that can be used in multiple steps machine learning. Here, we're using it to split the data
- gplots (https://cran.r-project.org/web/packages/gplots/gplots.pdf) - This is used to generate a heatmap.2
- RColorBrewer (https://cran.r-project.org/web/packages/RColorBrewer/RColorBrewer.pdf) - This is used to make the heatmap look cool

Hide

```
library(kernlab)
library(kknn)
library(caret)
library(gplots)
library(RColorBrewer)
mzhSeed=3456
```

Import data and view

The summary command allows us to take a quick look at the data to see what the its general distribution is. From this, we can see that A1, A9, A10, and A12 are all binary data where each of them have more 1's than 0's (mean is higher than 0.5). From our continuous data, we can see that we definitely have some outliers in each of those features.

Hide

```
credTBL <- read.table('credit_card_data.txt')
credHeadTBL <- read.table('credit_card_data-headers.txt',header=TRUE)
summary(credHeadTBL)
```

```
      A1               A2               A3               A8               A9               A10           

 Min.   :0.0000   Min.   :13.75   Min.   : 0.000   Min.   : 0.000   Min.   :0.0000   Min.   :0.0
000  
 1st Qu.:0.0000   1st Qu.:22.58   1st Qu.: 1.040   1st Qu.: 0.165   1st Qu.:0.0000   1st Qu.:0.0
000  
 Median :1.0000   Median :28.46   Median : 2.855   Median : 1.000   Median :1.0000   Median :1.0
000  
 Mean   :0.6896   Mean   :31.58   Mean   : 4.831   Mean   : 2.242   Mean   :0.5352   Mean   :0.5
612  
 3rd Qu.:1.0000   3rd Qu.:38.25   3rd Qu.: 7.438   3rd Qu.: 2.615   3rd Qu.:1.0000   3rd Qu.:1.0
000  
 Max.   :1.0000   Max.   :80.25   Max.   :28.000   Max.   :28.500   Max.   :1.0000   Max.   :1.0
000  
      A11              A12              A14               A15                R1        
 Min.   : 0.000   Min.   :0.0000   Min.   :   0.00   Min.   :     0   Min.   :0.0000  
 1st Qu.: 0.000   1st Qu.:0.0000   1st Qu.:  70.75   1st Qu.:     0   1st Qu.:0.0000  
 Median : 0.000   Median :1.0000   Median : 160.00   Median :     5   Median :0.0000  
 Mean   : 2.498   Mean   :0.5382   Mean   : 180.08   Mean   :  1013   Mean   :0.4526  
 3rd Qu.: 3.000   3rd Qu.:1.0000   3rd Qu.: 271.00   3rd Qu.:   399   3rd Qu.:1.0000  
 Max.   :67.000   Max.   :1.0000   Max.   :2000.00   Max.   :100000   Max.   :1.0000  
```

Data processing

Since this data has already been preprocessed, there is no need to remove NAs or cleanup corrupt data points. Instead, I could proceed with normalizing the data and creating a training, validation, and testing set. I chose to use createDataPartition to create balanced sets that to help minimize an algorithm's performance based on luck. I've also specified a seed to allow reproducibility.

Hide

```
# Rescale all continuous data so one feature does not overwhelm others.
scaleNorm <- function(x)
  {
  return((x-min(x))/(max(x)-min(x)))
  }
credDF <- as.data.frame(lapply(credHeadTBL,scaleNorm))
# Setting the seed allows us to create the same random set everytime to help with reproducibilit
y.
set.seed(mzhSeed)
#Using createDataPartition to separate data to have similar ratio of dependent variables
trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
trainset <- credDF[trainIndex,]
valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)
#Training set
credTrain <- trainset[-valIndex,]
trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
trainLab <- credTrain$R1
#Validation set
credVal <- trainset[valIndex,]
valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
valLab <- credVal$R1
#Testing set
credTest <- credDF[-trainIndex,]
testFeat <- credTest[,!colnames(credTrain) %in% c("R1")]
testLab <- credTest$R1
#Viewing the size of each set
trainSize <- dim(credTrain)
valSize <- dim(credVal)
testSize <- dim(credTest)
sprintf("Training set dim: %i x %i",trainSize[1],trainSize[2])
```

```
[1] "Training set dim: 393 x 11"
```

Hide

```
sprintf("Validation set dim: %i x %i",valSize[1],valSize[2])
```

```
[1] "Validation set dim: 131 x 11"
```

Hide

```
sprintf("Testing set dim: %i x %i",testSize[1],testSize[2])
```

```
[1] "Testing set dim: 130 x 11"
```

SVM

Now I can start getting into the support vector machine tuning. The assignment asked us to vary the C value and during my initial run, I tested it only on vanilladot(linear kernel). I saw absolutely no variations in my accuracy, so I revised this to also loop through other Kernels to see how changing the C value will affect it.

Hide

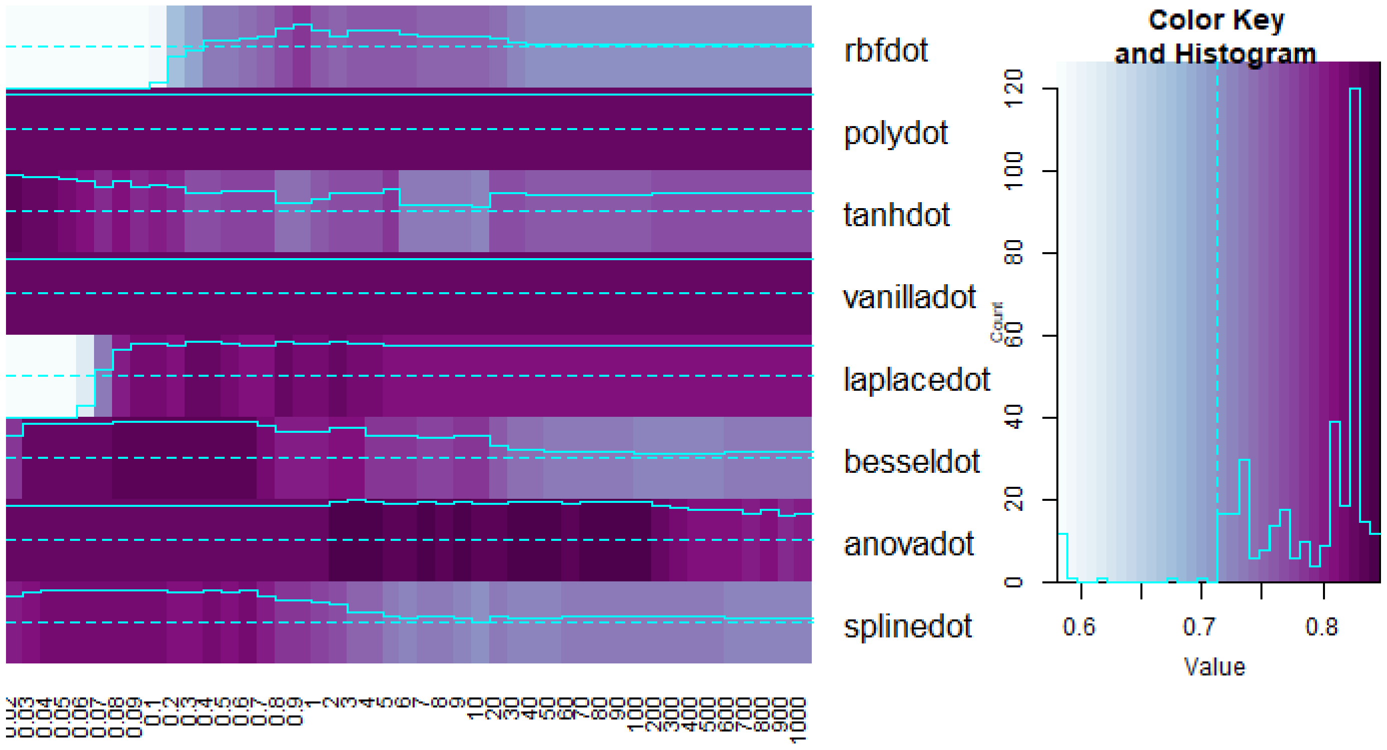
```
set.seed(mzhSeed)
#Define the range of C values
CAry <- c(2:10 %o% 10^(-2:2))
#Define the kernels used
Kern <- c("rbfdot","polydot","tanhdot","vanilladot","laplacedot","besseldot","anovadot","splined
ot")
ModelAcc <- vector()
acc <- matrix(NA,nrow=length(Kern),ncol=length(CAry))
#Loop through each Kernel
for (i in 1:length(Kern))
{
  #Loop through each C value
  for (j in 1:length(CAry))
  {
    #kpar is actually use to prevent printout in R Notebook
    model <- ksvm(as.matrix(trainFeat),as.factor(trainLab),kernel=Kern[i],type="C-svc",scaled=TR
UE,C=CAry[j],kpar=list())
    pred <- predict(model,valFeat)
    acc[i,j] <- sum(pred==valLab)/nrow(valFeat)
  }
}
```

Visualizing the Kernels vs C tuning

Now that I have a matrix of prediction accuracies with the X axis being the varying C values on a log scale and the y axis being the different kernel methods tested, I don't want to just find out which combination of C and kernel gave me the best value, but possibly which kernel method performs best at varying levels of C. I decided to plot this data out in a heatmap to really get a sense of how the values look.

Hide

```
#assigning row and col names so they appear properly
rownames(acc)=Kern
colnames(acc)=CAry
#adjusting the graph sizes
lmat = rbind(c(1,4),c(2,3))
lhei <- c(10,2)
lwid <- c(10,5)
#Defining the color gradient
jBrewColors <- brewer.pal(n = 9, name = "BuPu")
ramp <- colorRampPalette(jBrewColors)
#Plotting the heatmap
heatmap.2(acc,dendrogram = 'none',Rowv=FALSE, Colv=FALSE,scale="none",trace="row",col=ramp(32),m
argins = c(2,4), lmat = lmat,lwid = lwid, lhei = lhei)
```



Fine tuning C

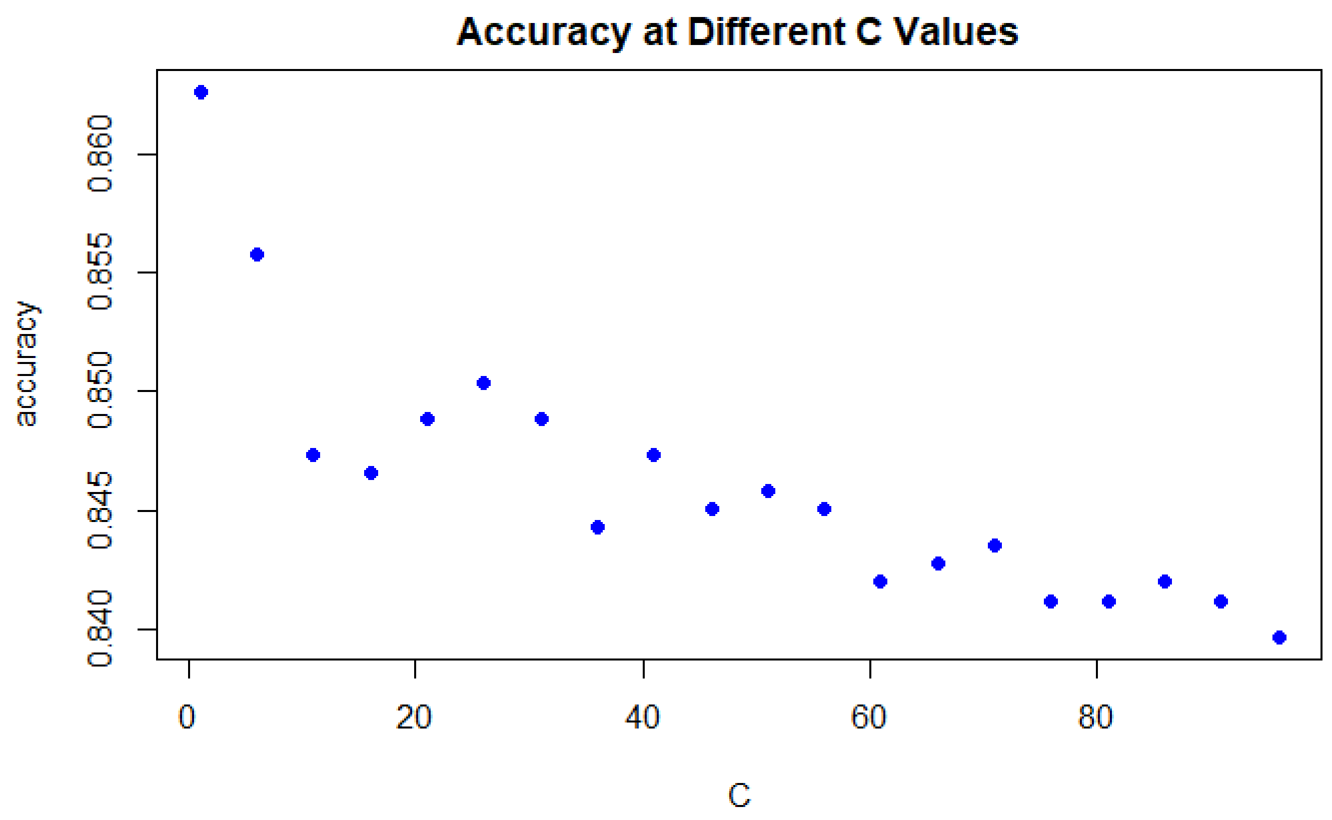
Now I saw that anovadot is a really strong performer at the range of 1 to 100, I can narrow down the range of C and fine tune its value. Additionally, Anovadot actually has two other parameter that can be fine tuned sigma and degree, but that will be research for another time.

Hide

```
#Narrow down the range of values for C
CAry <- seq(1,100,5)
ModelAcc <- vector()
accC <- matrix(NA,nrow=10,ncol=length(CAry))
#Loop through 10 seeds to introduce enough variation
for (i in 1:10)
{
  set.seed(i)
  #Create partitions
  trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
  trainset <- credDF[trainIndex,]
  valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)

  #Training set
  credTrain <- trainset[-valIndex,]
  trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
  trainLab <- credTrain$R1

  #Validation set
  credVal <- trainset[valIndex,]
  valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
  valLab <- credVal$R1

  #Loop through all values of C
  for (j in 1:length(CAry))
  {
    model <- ksvm(as.matrix(trainFeat),as.factor(trainLab),kernel="anovadot",type="C-
svc",scaled=TRUE,C=CAry[j],kpar=list())
    pred <- predict(model,valFeat)
    accC[i,j] <- sum(pred==valLab)/nrow(valFeat)
  }
}
```

Hide

```
#cbind the array of C value with the column mean of the accuracy matrix
meansVal=cbind(seq(1,100,5),colMeans(accC))
plot(meansVal,main="Accuracy at Different C Values",xlab="C",ylab="accuracy", pch=19,
col="blue")
```



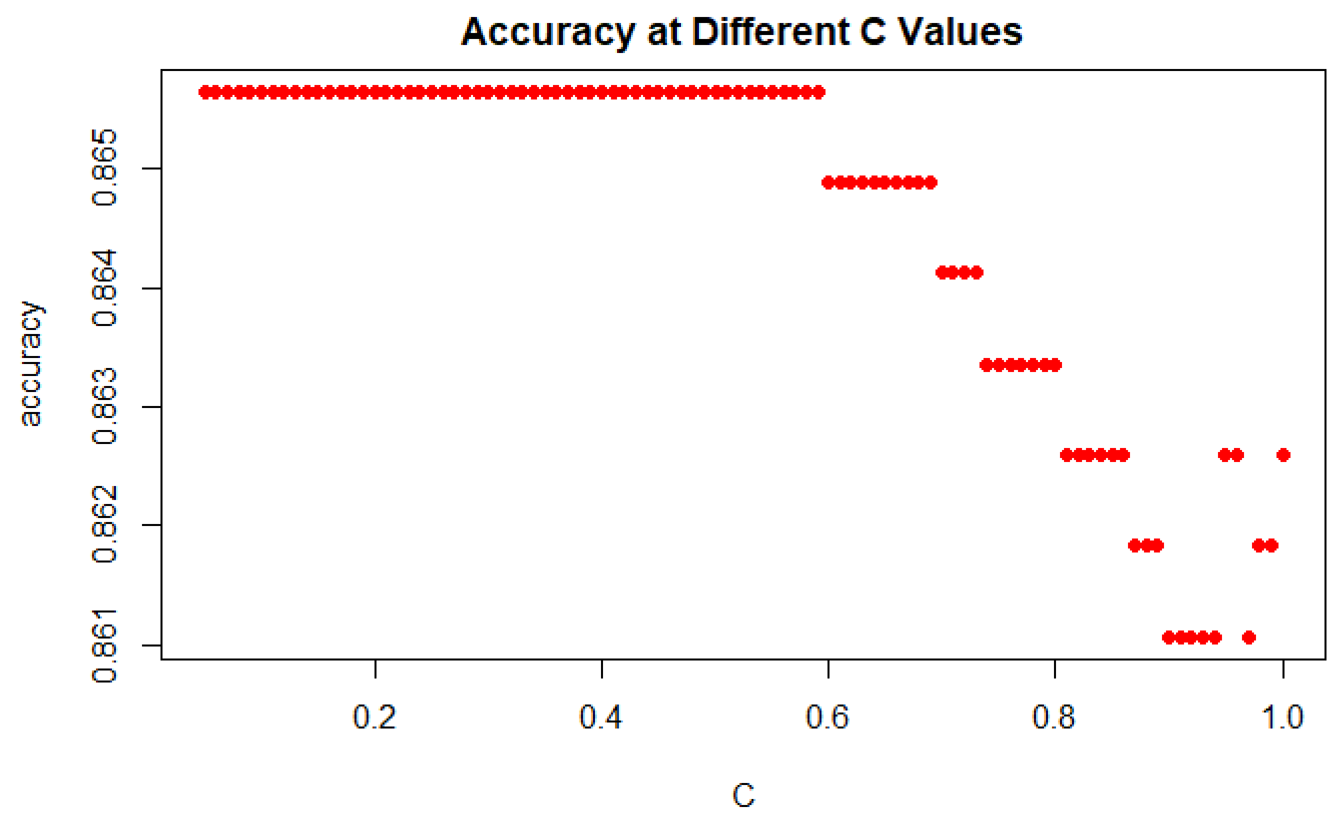
From the above graph, it’s starting to look like we actually have more accuracy from lower values of C than higher. So I’m going to run this again with a smaller set of C values.

Hide

```
#redefine the narrow region of C values
CAry <- seq(0.05,1,0.01)
accD <- matrix(NA,nrow=10,ncol=length(CAry))
for (i in 1:10)
{
  set.seed(i)
  #Create partitions
  trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
  trainset <- credDF[trainIndex,]
  valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)

  #Training set
  credTrain <- trainset[-valIndex,]
  trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
  trainLab <- credTrain$R1

  #Validation set
  credVal <- trainset[valIndex,]
  valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
  valLab <- credVal$R1
  for (j in 1:length(CAry))
  {
    model <- ksvm(as.matrix(trainFeat),as.factor(trainLab),kernel="anovadot",type="C-
svc",scaled=TRUE,C=CAry[j],kpar=list())
    pred <- predict(model,valFeat)
    accD[i,j] <- sum(pred==valLab)/nrow(valFeat)
  }
}
```

Hide

```
meansVal=cbind(seq(0.05,1,0.01),colMeans(accD))
plot(meansVal,main="Accuracy at Different C Values",xlab="C",ylab="accuracy", pch=19, col="red")
```



It looks like I can really pick any values of C between 0.1 and 0.6. So I'm going to hit the middle with 0.3

Testing Data

Now that we've picked a solid kernel and a solid C value, we're going to run it through 100 different seeds of training and testing to see what the mean value of our test accuracy would be.

Hide

```
trainAcc <- vector()
valAcc <- vector()
testAcc <- vector()
for (i in 1:100)
{
  set.seed(i)
  #Create partitions
  trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
  trainset <- credDF[trainIndex,]
  valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)

  #Training set
  credTrain <- trainset[-valIndex,]
  trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
  trainLab <- credTrain$R1

  #Validation set
  credVal <- trainset[valIndex,]
  valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
  valLab <- credVal$R1
  #Test set
  credTest <- credDF[-trainIndex,]
  testFeat <- credTest[,!colnames(credTrain) %in% c("R1")]
  testLab <- credTest$R1
  model <- ksvm(as.matrix(trainFeat),as.factor(trainLab),kernel="anovadot",type="C-svc",scaled=T
RUE,C=0.3,kpar=list())
  trainPred <- predict(model,trainFeat)
  trainAcc[i] <- sum(trainPred==trainLab)/nrow(trainFeat)
  valPred <- predict(model,valFeat)
  valAcc[i] <- sum(valPred==valLab)/nrow(valFeat)
  testPred <- predict(model,testFeat)
  testAcc[i] <- sum(testPred==testLab)/nrow(testFeat)
}
sprintf("Train Accuracy: %s%%",mean(trainAcc)*100)
```

```
[1] "Train Accuracy: 86.3027989821883%"
```

Hide

```
sprintf("Validation Accuracy: %s%%",mean(valAcc)*100)
```

```
[1] "Validation Accuracy: 86.1450381679389%"
```

Hide

```
sprintf("Test Accuracy: %s%%",mean(testAcc)*100)
```

```
[1] "Test Accuracy: 86.1692307692308%"
```

SVM Homework Equation and Accuracy

Finally, we'll get back to what the homework asked us to do which is to gather the accuracy of the model tested on the same dataset it is trained on without doing cross-validation. Additionally, the homework asked for the coefficients of each variable that the model trained on.

Hide

```
#Using the full dataset, separate it into features and labels
credFeat=credDF[,1:10]
credLab=credDF$R1
#Run the model on everything with our tuned parameters
model <- ksvm(as.matrix(credFeat),as.factor(credLab),kernel="anovadot",type="C-
svc",scaled=TRUE,C=0.3,kpar=list())
#Obtain the coefficient values and print them
a <- colSums(credDF[model@SVindex,1:10] * model@coef[[1]])
a0 <- sum(a*credDF[1,1:10])-model@b
colmns=c(names(a),"Intercept")
coeff=c(a,a0)
coeffTBL=cbind(colmns,coeff)
rownames(coeffTBL) <- NULL
print(coeffTBL)
```

```
      colmns      coeff
 [1,] "A1"        "0.000235913646850194"
 [2,] "A2"        "-0.0841757228903852"
 [3,] "A3"        "-0.0460680570260616"
 [4,] "A8"        "0.0237325453600105"
 [5,] "A9"        "1.01945018406938"
 [6,] "A10"       "-0.00322700142121967"
 [7,] "A11"       "0.0173474821632253"
 [8,] "A12"       "0.000139063620248823"
 [9,] "A14"       "-0.0689547404818019"
[10,] "A15"       "0.294158919580667"
[11,] "Intercept" "1.1054062244456"
```

Sigma=1, degree=1, C=0.3, kernel = "anovadot" are the parameters used to tune this SVM which maintained an accuracy of 86.2%

Hide

```
#Generate a prediction accuracy
credPred <- predict(model,credFeat)
credAcc <- sum(credPred==credLab)/nrow(credFeat)
sprintf("Cred Accuracy: %s%%",mean(credAcc)*100)
```

```
[1] "Cred Accuracy: 86.2385321100918%"
```

KNN

Next component is to look at the K-Nearest Neighbor algorithm.

Model training

Reviewing the documentations, I saw the train.kknn documentation (https://www.rdocumentation.org/packages/kknn/versions/1.3.1/topics/train.kknn) which provides an automatic way to handle the leave-one-out method and also automatic parsing of kernels and k-values to provide you with the best method. I ran this on the full training dataset and found that the most optimal value of K was 55 and the kernel was gaussian. This provided an accuracy value of 88.9% when tested on itself.

Hide

```
#Normalize the data
scaleNorm <- function(x)
  {
  return((x-min(x))/(max(x)-min(x)))
  }
credDF <- as.data.frame(lapply(credHeadTBL,scaleNorm))
credLab=credDF$R1
#Train the model
knnModel <- train.kknn(formula=R1~., data=credDF, kmax=100, kernel= c("rectangular", "triangula
r", "epanechnikov",
"gaussian", "rank", "optimal"), distance = 2)
#Test the model
predVal=predict(knnModel,credDF)
#Get accuracy values
predVal[predVal>=0.5]=1
predVal[predVal<0.5]=0
Acc <- sum(predVal==credLab)/nrow(credDF)
#Print
sprintf("Accuracy: %s%%",mean(Acc)*100)
```

```
[1] "Accuracy: 88.9908256880734%"
```

Hide

```
sprintf("Best Kernel is: %s",knnModel$best.parameters$kernel)
```

```
[1] "Best Kernel is: gaussian"
```

Hide

```
sprintf("Best K value is: %s",knnModel$best.parameters$k)
```

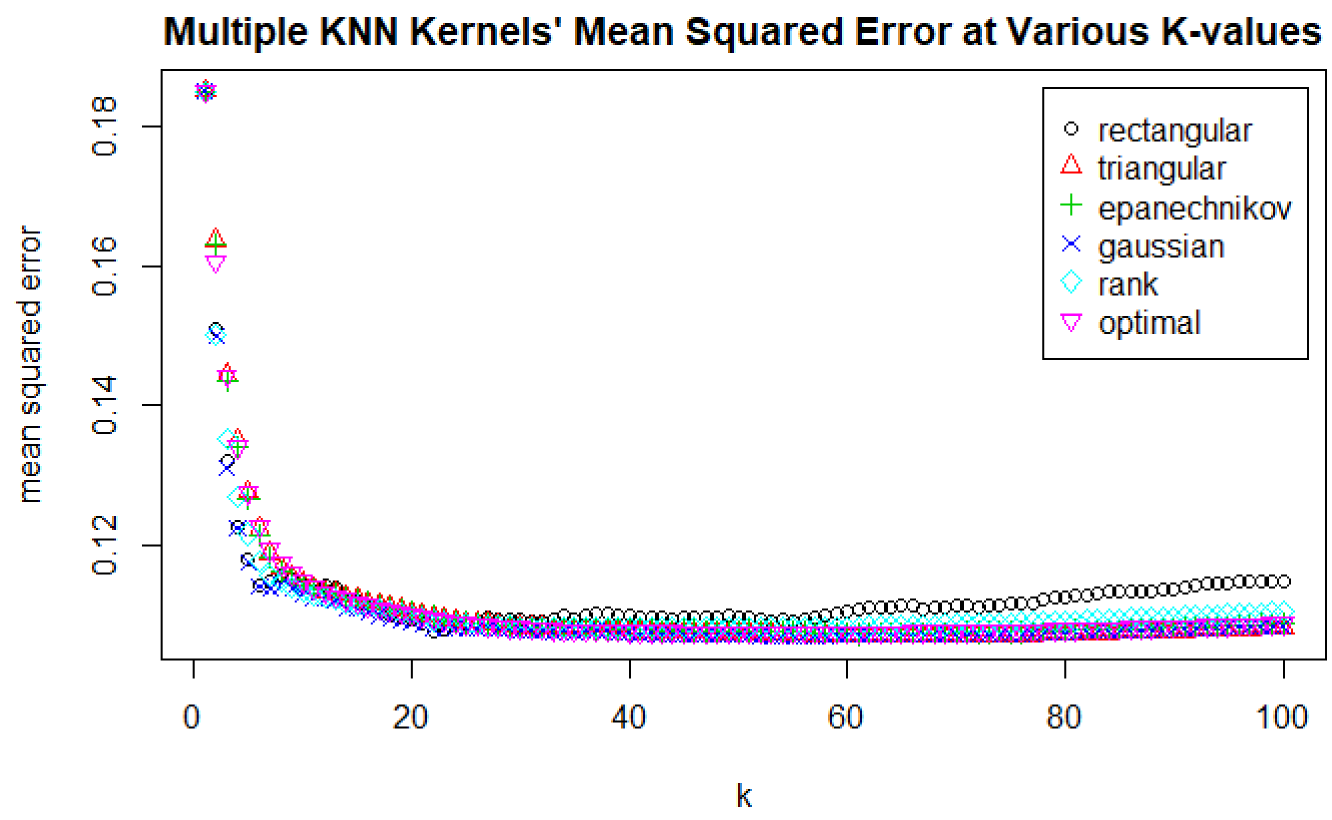
```
[1] "Best K value is: 55"
```

Visualize different values of K

Here we'll take a look at all the kernels plotted out along each individual K value. It's really interesting to see that multiple methods actually performs worse as the K value goes up.

Hide

```
plot(knnModel,main="Multiple KNN Kernels' Mean Squared Error at Various K-values")
```



Cross-Validation

For Cross-Validation, I'm going to separate out the data into Training, Validation, and Testing set. ### Training and Validation I've iterated across 100 different seeds for randomization and tuned for 100 different K values. Once this value is aggregated in the knnAcc matrix where the rows are each individual seed and columns are different K values, I can average across the rows and generate a graph to show the accuracy at each K value.

Hide

```
#Create empty matrix
knnAcc <- matrix(NA,nrow=100,ncol=100)
#iterate 100 times to generate 100 seeds to average over
for (i in 1:100)
{
  set.seed(i)

  #Using createDataPartition to separate data to have similar ratio of dependent variables
  trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
  trainset <- credDF[trainIndex,]
  valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)

  #Training set
  credTrain <- trainset[-valIndex,]
  trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
  trainLab <- credTrain$R1

  #Validation set
  credVal <- trainset[valIndex,]
  valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
  valLab <- credVal$R1

  #Testing set
  credTest <- credDF[-trainIndex,]
  testFeat <- credTest[,!colnames(credTrain) %in% c("R1")]
  testLab <- credTest$R1

  #loop through k=1 to 100 to see which one is most optimal
  for (j in 1:100)
  {
    cvModel <- kknn(R1~.,credTrain,credVal,kernel="gaussian",k=j,distance=2)
    fit <- fitted(cvModel)
    #any prediced value above or equal .5 is 1, below .5 is 0
    fit[fit>=0.5] <- 1
    fit[fit<0.5] <- 0
    #each row is a different seed and each column is a different K value.
    knnAcc[i,j] <- sum(fit==valLab)/nrow(credVal)
  }
}
```

Visualize K-value vs Accuracy for Train vs Val

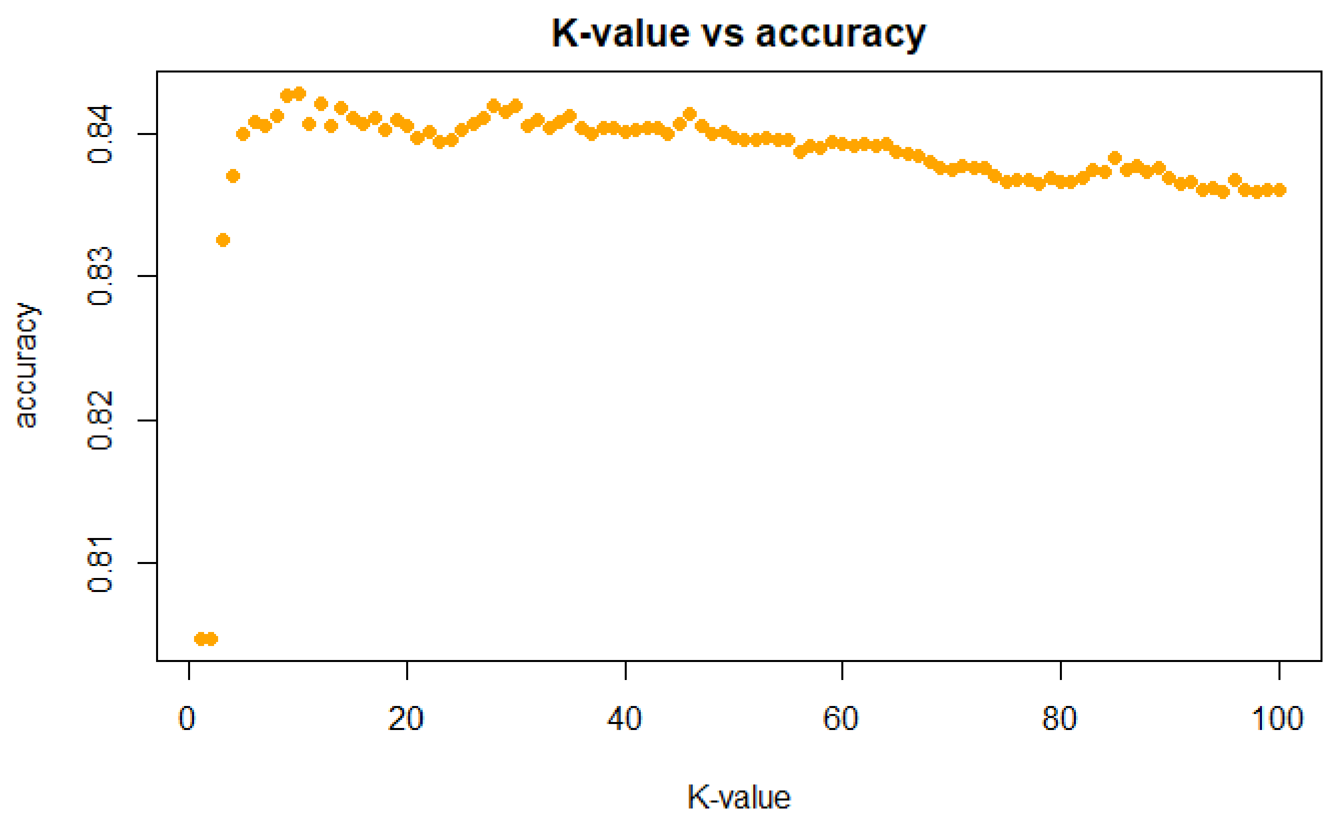
Hide

```
#average across all seeds
kAvgAcc=colMeans(knnAcc)
which(kAvgAcc==max(kAvgAcc))
```

```
[1] 10
```

Hide

```
plot(kAvgAcc,main="K-value vs accuracy",xlab="K-value",ylab="accuracy",pch=19, col="orange")
```



Testing Accuracy

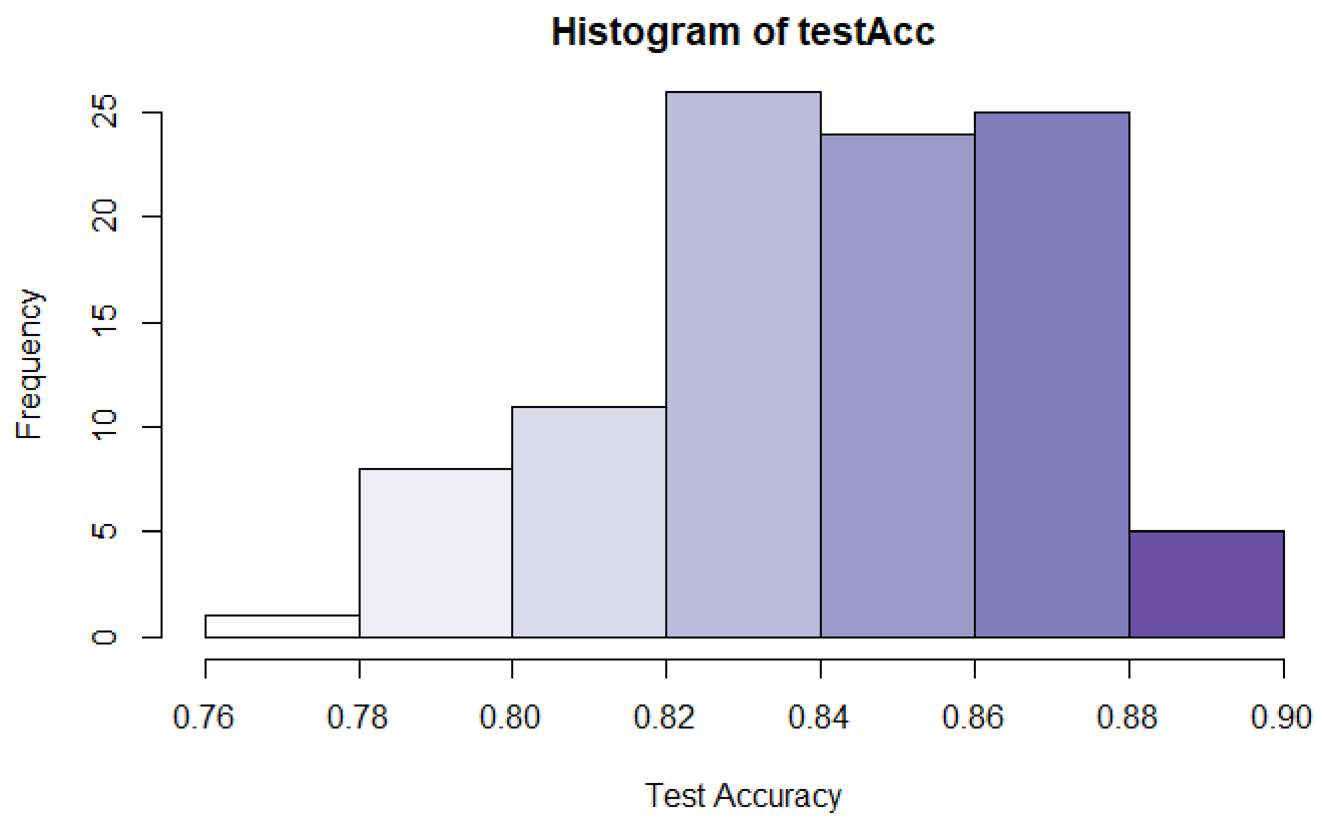
Next, I use the best value of K(10) on the test data and ran this on different cuts of the test data. From there, I produced a histogram to take a look at the distribution of my results and provide an average value of the testing accuracy

Hide

```
testAcc <- vector()
for (i in 1:100)
{
    set.seed(i)

  #Using createDataPartition to separate data to have similar ratio of dependent variables
  trainIndex <- createDataPartition(credDF$R1, p = .8, list = FALSE, times = 1)
  trainset <- credDF[trainIndex,]
  valIndex <- createDataPartition(trainset$R1, p = .25, list = FALSE, times = 1)

  #Training set
  credTrain <- trainset[-valIndex,]
  trainFeat <- credTrain[,!colnames(credTrain) %in% c("R1")]
  trainLab <- credTrain$R1

  #Validation set
  credVal <- trainset[valIndex,]
  valFeat <- credVal[,!colnames(credTrain) %in% c("R1")]
  valLab <- credVal$R1

  #Testing set
  credTest <- credDF[-trainIndex,]
  testFeat <- credTest[,!colnames(credTrain) %in% c("R1")]
  testLab <- credTest$R1

  cvModel <- kknn(R1~.,credTrain,credTest,kernel="gaussian",k=10,distance=2)
  fit <- fitted(cvModel)
  fit[fit>=0.5] <- 1
  fit[fit<0.5] <- 0
  testAcc[i] <- sum(fit==testLab)/nrow(credTest)
}
```

Hide

```
colsPal <- brewer.pal(9,"Purples")
hist(testAcc,breaks=5,xlab="Test Accuracy",col=colsPal)
```



Hide

```
sprintf("Average Testing Accuracy is: %s%%",100*mean(testAcc))
```

```
[1] "Average Testing Accuracy is: 84.2153846153846%"
```

Summary

In this HW assignment, we learned about SVM, KNN, and CV.

Support Vector Machines are large margin classifiers that attempt to find the hyperplane that best separates two sets of data with the largest margin between them. This classifier is capable of implementing multiple kernels that can “lift” the data up into a higher dimension to help separate them with a hyperplane. It is commonly used with small datasets and medium number of features. However, in a higher dimensional space and a large dataset, it suffers from performance issues. There are multiple parameters to watch out for when tuning this algorithm. C parameter defines how much you want to avoid misclassifying each training example. A large value of C is used if you want to misclassify less percentage of your data but this can overfit your data. A small value of C helps you generalize your data but can underfit your data.

K-Nearest Neightbors is a algorithm that looks to the K closest training examples in the feature space. This algorithm is especially useful in classifying geolocation data (https://www.kaggle.com/xchiron/rental-list-knn-on-lat-long-data). The primary parameter to tune for this algorithm is the K value which train.kknn does a great job of doing. You can also specify the kernel to use and also the distance calculation method as l1, l2, etc.

Cross-Validation is a method used to estimate model accuracy, check for bias and variance, and help decide what to do next in algorithm testing. There are various forms of Cross Validation such as K-fold, Leave one out, data split, and bootstrap. The main goal here is to separate data into training, validation, and testing sets so you can train on the training set and test on the validation set to help select the best performing model. Finally, you can use the testing set to check your model’s accuracy. You can also review the training data’s accuracy vs validation data’s accuracy across multiple iterations to determine whether you have high bias or high variance.