Sharat_Sripada_HW6_7

```
library(FactoMineR)
library(dplyr)

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

library(rattle)

## Loading required package: tibble

## Loading required package: bitops

## Rattle: A free graphical interface for data science with R.
## Version 5.4.0 Copyright (c) 2006-2020 Togaware Pty Ltd.
## Type 'rattle()' to shake, rattle, and roll your data.

library(e1071)
library(caret)

## Loading required package: lattice

## Loading required package: ggplot2

library(rpart)
library(randomForest)

## randomForest 4.6-14

## Type rfNews() to see new features/changes/bug fixes.

##
## Attaching package: 'randomForest'

## The following object is masked from 'package:ggplot2':
##
##     margin

## The following object is masked from 'package:rattle':
##
##     importance
```

```
## The following object is masked from 'package:dplyr':
##
##     combine

library(class)
```

Introduction

This submission will dive into comparing the performance of various models covered through Week5-8. These include Naive Bayes, Decision Trees, SVM, KSVMs, KNN and Random Forest on a data-set comprising images of digits. The data and the problem statement is defined in the following Kaggle competition: https://www.kaggle.com/c/digit-recognizer/overview

To state the purpose briefly, we will attempt to accurately predict number images by training the models specified above.

EDA

Dimensionality and data reduction

As a first step let us load up the data from Kaggle and run some EDA

```
filename <- '/Users/venkatasharatsripada/Downloads/train.csv'
digits_all <- read.csv(filename, header=TRUE, stringsAsFactors = TRUE)

# Examine the data
str(digits_all)

## 'data.frame':    42000 obs. of  785 variables:
##  $ label   : int  1 0 1 4 0 0 7 3 5 3 ...
##  $ pixel0  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel1  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel2  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel3  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel4  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel5  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel6  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel7  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel8  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel9  : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel10 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel11 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel12 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel13 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel14 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel15 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel16 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel17 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel18 : int  0 0 0 0 0 0 0 0 0 0 ...
```

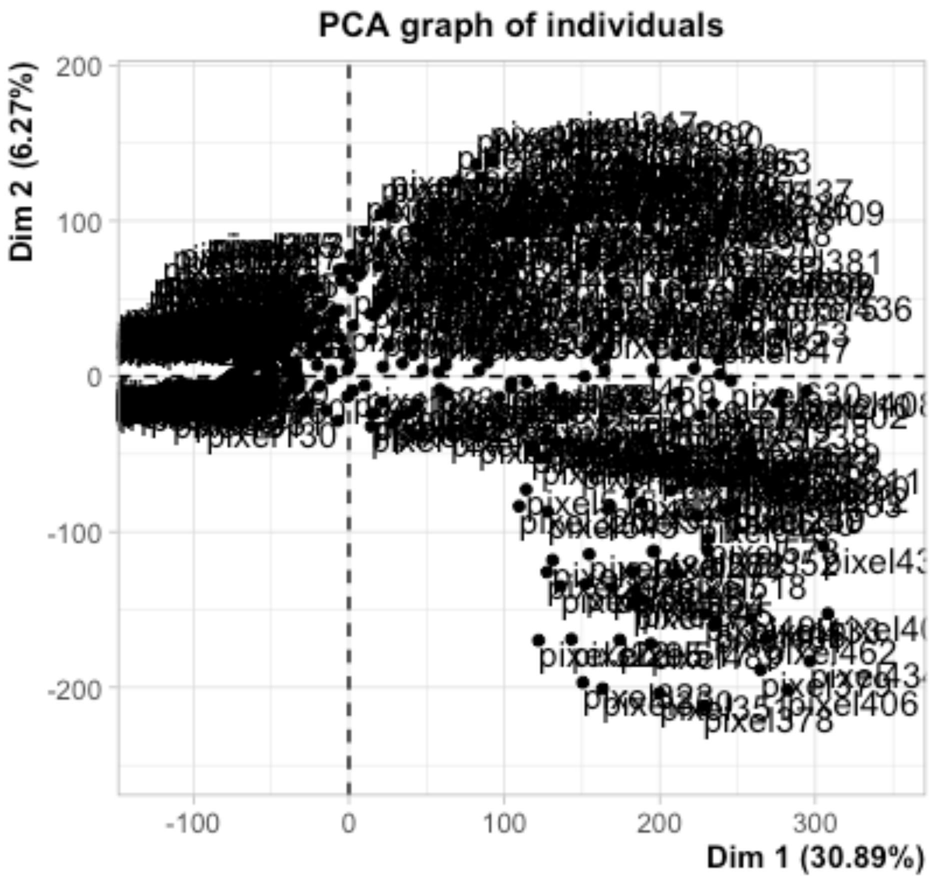
```
##  $ pixel19 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel20 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel21 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel22 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel23 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel24 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel25 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel26 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel27 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel28 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel29 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel30 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel31 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel32 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel33 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel34 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel35 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel36 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel37 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel38 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel39 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel40 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel41 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel42 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel43 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel44 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel45 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel46 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel47 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel48 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel49 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel50 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel51 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel52 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel53 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel54 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel55 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel56 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel57 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel58 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel59 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel60 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel61 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel62 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel63 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel64 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel65 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel66 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel67 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel68 : int  0 0 0 0 0 0 0 0 0 0 ...
```

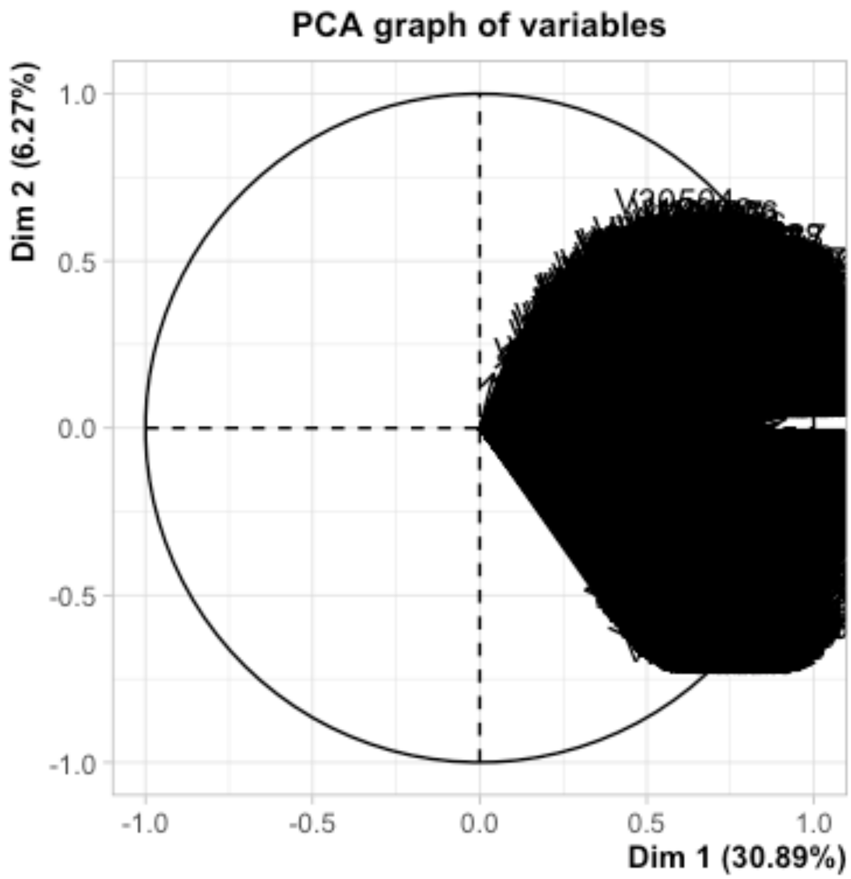
```
##  $ pixel69 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel70 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel71 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel72 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel73 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel74 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel75 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel76 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel77 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel78 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel79 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel80 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel81 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel82 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel83 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel84 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel85 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel86 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel87 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel88 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel89 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel90 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel91 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel92 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel93 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel94 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel95 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel96 : int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel97 : int  0 0 0 0 0 0 0 0 0 0 ...
##   [list output truncated]
```

```r
# Label has data-type as int, changing to factor
digits_all$label <- as.factor(digits_all$label)

# Examine the dimensionality of the data-set
dim(digits_all)
```

```
## [1] 42000   785
```

```r
# Reduce dimensionality of data
# Default ncp=5, play around with ncp values
# to find a suitable number of reduced dimensionality
pca_digits <- PCA(t(select(digits_all, -label)), ncp=30)
```


PCA graph of individuals
Dim 2 (6.27%)
Dim 1 (30.89%)
200
100
0
-100
-200
-100
0
100
200
300



Let's apply PCA and reduce the data-set dimensionality and the number of data samples.

```
digits_reduced <- data.frame(digits_all$label, pca_digits$var$coord)

# Examine digits_reduced
dim(digits_reduced)

## [1] 42000    31

# Reduce number of samples
percent <- 0.25
set.seed(275)
digitsplit <- sample(nrow(digits_reduced), nrow(digits_reduced)*percent)
digits_final <- digits_reduced[digitsplit,]

# Examine the final data-frame we will use for all modelling
dim(digits_final)

## [1] 10500    31
```

Data splitting for Cross-validation

First, we will split the train-set based on a k-value (here k=10, means a 10-fold cross-validation model) and carve out test-set based on a holdout value. The remaining data will serve as the training set.

NOTE:

This is a far more exhaustive validation of machine learning models than the simple 'Hold-Out Test' which would split the data into train/test based on simple ratios.

```
N <- nrow(digits_final)

# Number of splits
kfolds <- 10

# Split the data into train/test
holdout <- split(sample(1:N), 1:kfolds)
```

Cross-validation results across various models

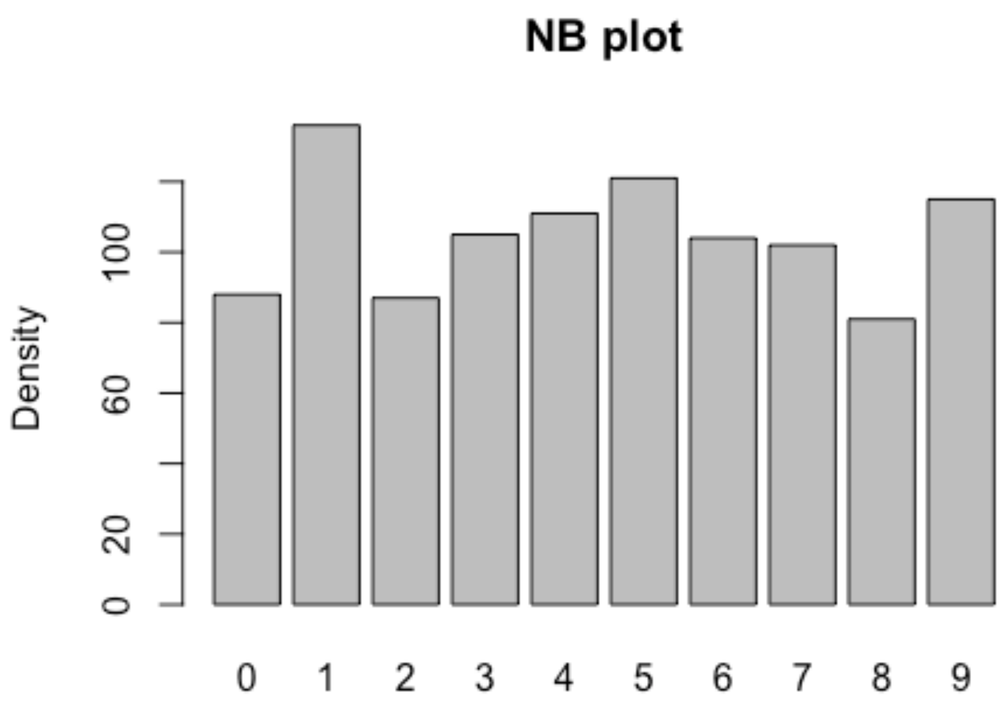
Naive Bayes

Create the test and train data-sets and run the Naive Bayes machine learning model

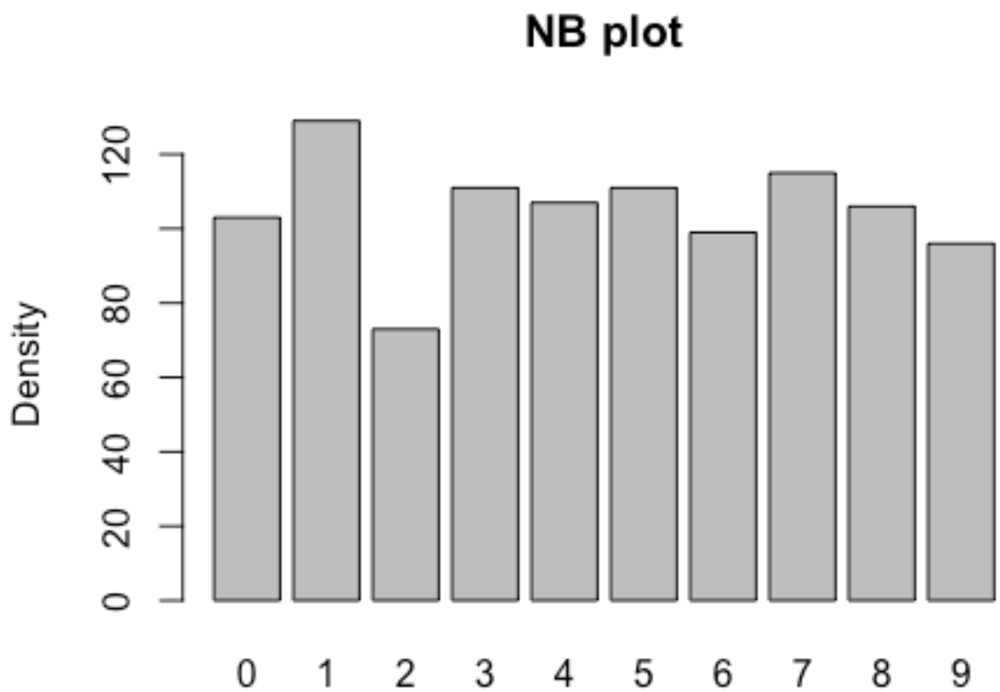
```
all_results <- list()
all_labels <- list()

for (k in 1:kfolds) {
  digits_final_test <- digits_final[holdout[[k]],]
  digits_final_train <- digits_final[-holdout[[k]], ]

  # View the train/test data-sets
  head(digits_final_test)
  head(digits_final_train)

  # Remove the label from the test-data
  digits_final_test_noLabel <- digits_final_test[-c(1)]

  # Just the label
  digits_final_test_Label <- digits_final_test[c(1)]

  # Train the model
  train_nb <- naiveBayes(digits_all.label~., data=digits_final_train,
na.action=na.pass)

  # Make predictions
  predict_nb <- predict(train_nb, digits_final_test_noLabel)
```

```
  # Evaluate accuracy of the model
  comp_table <- data.frame(Actual=digits_final_test_Label$digits_all.label,
Predicted=predict_nb)
  matrix_nb <- confusionMatrix(as.factor(comp_table$Predicted),
as.factor(comp_table$Actual))

  print(matrix_nb$overall)

  # Visualize Naive-Bayes plots
  plot(predict_nb, ylab='Density', main='NB plot')
}
```

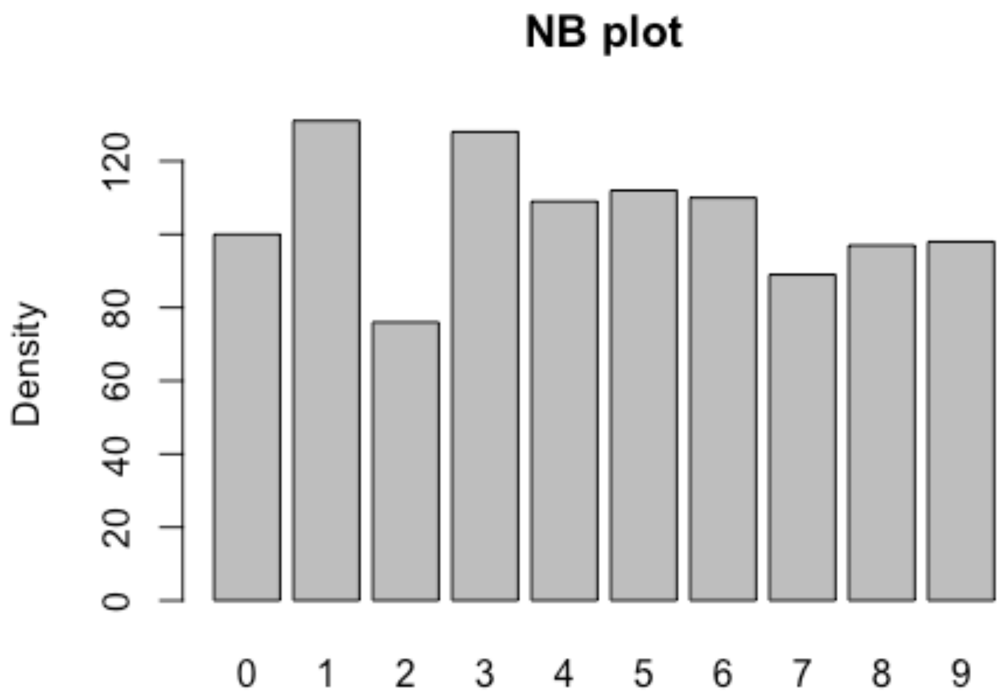
```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8600000      0.8441539      0.8375272      0.8804320      0.1276190
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



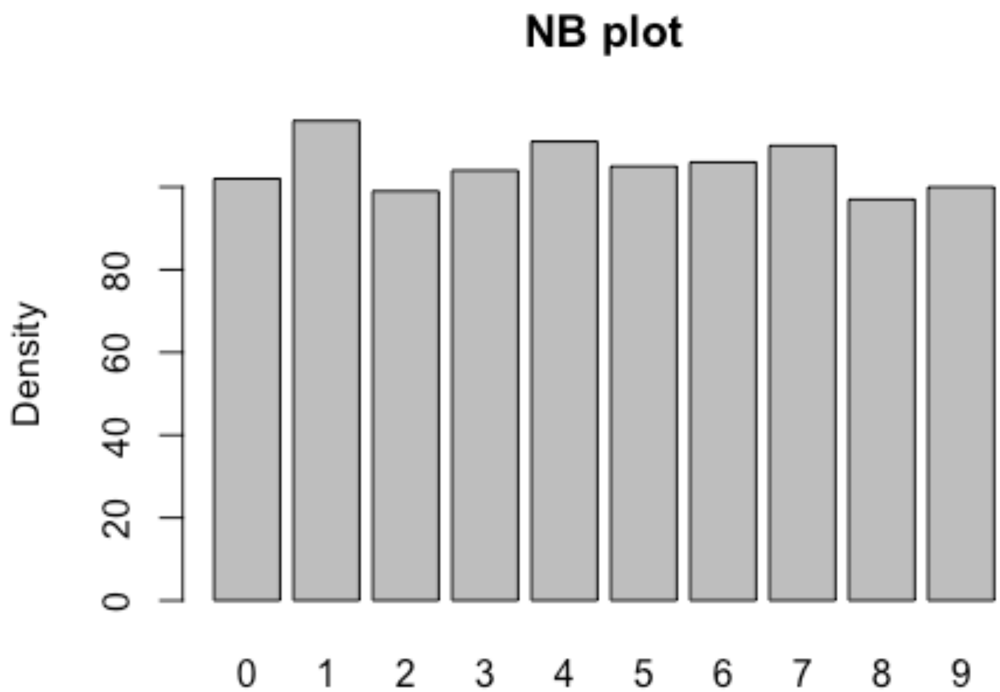

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8476190      0.8304662      0.8244343      0.8688351      0.1238095
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



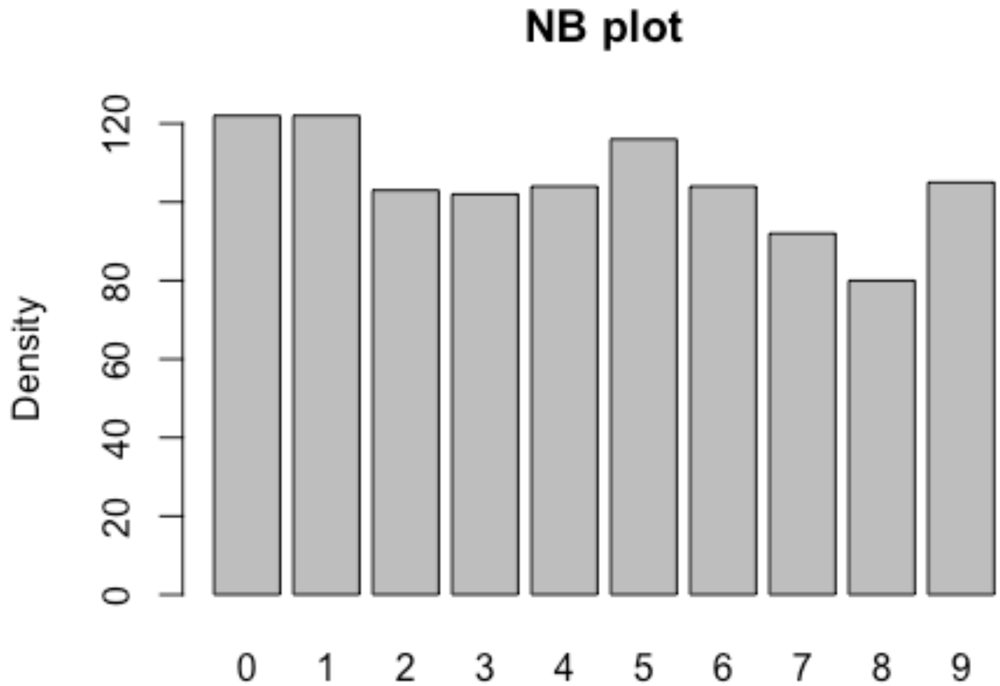

```
##        Accuracy           Kappa   AccuracyLower   AccuracyUpper    AccuracyNull
##       0.8514286       0.8346155       0.8284574       0.8724089       0.1257143
## AccuracyPValue   McnemarPValue
##       0.0000000             NaN
```



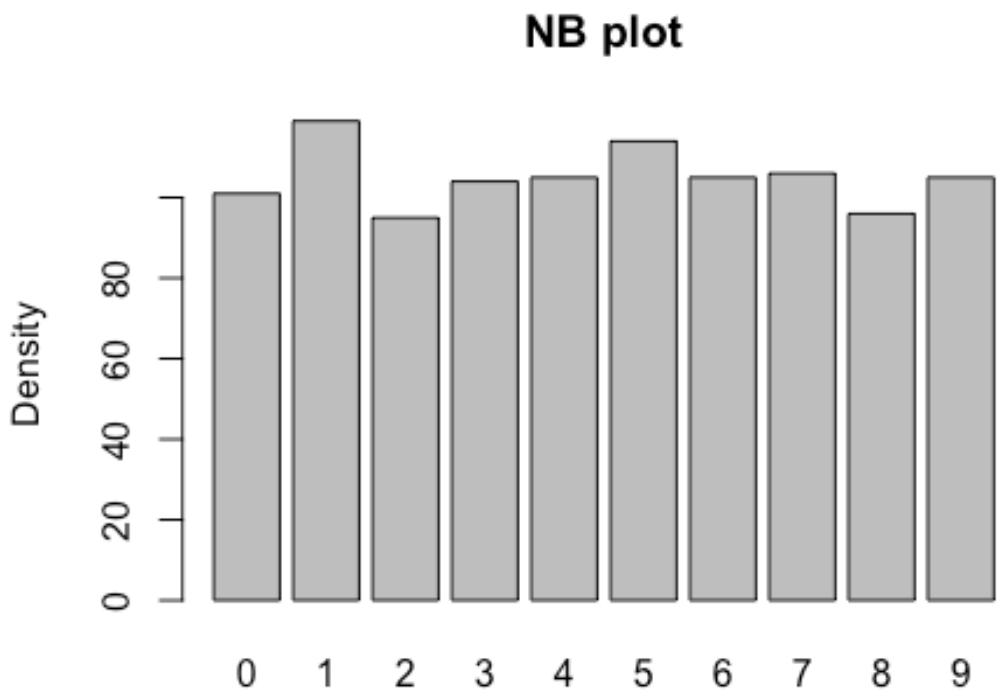

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8476190      0.8306496      0.8244343      0.8688351      0.1114286
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



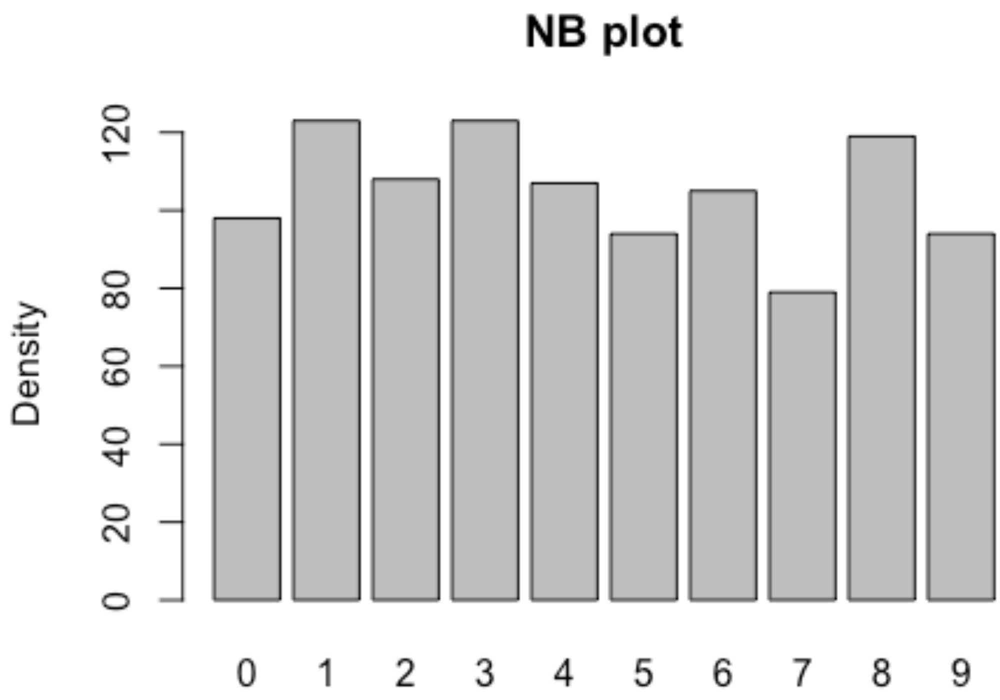

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8638095      0.8485454      0.8415665      0.8839894      0.1209524
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



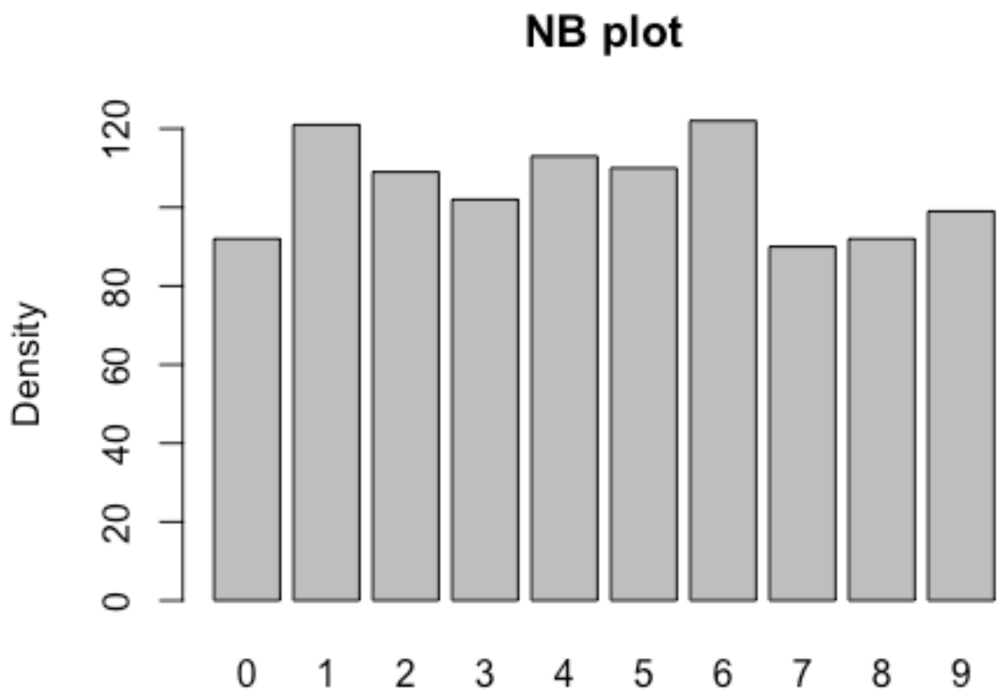

```
##        Accuracy           Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##       0.8580952       0.8423039      0.8355095      0.8786513      0.1171429
## AccuracyPValue  McnemarPValue
##       0.0000000             NaN
```



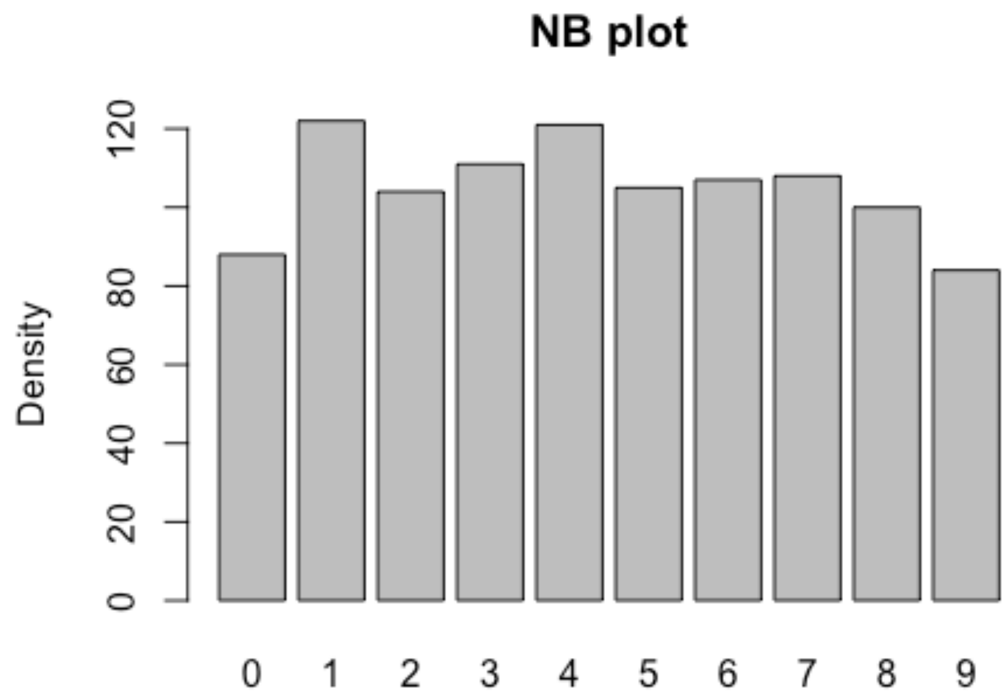

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8542857      0.8378216      0.8314778      0.8750861      0.1247619
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



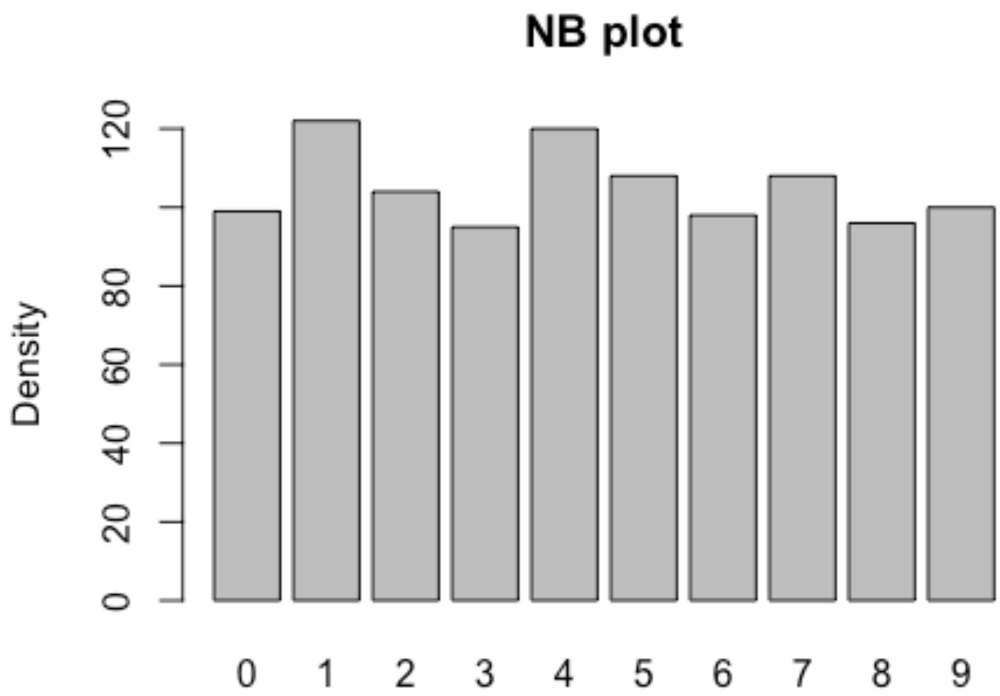

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8647619      0.8496054      0.8425772      0.8848779      0.1142857
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```




```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8790476      0.8654962      0.8577800      0.8981626      0.1161905
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```




```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.8447619      0.8274205      0.8214201      0.8661516      0.1142857
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```



The overall accuracy for this model ranges between 83-88%.

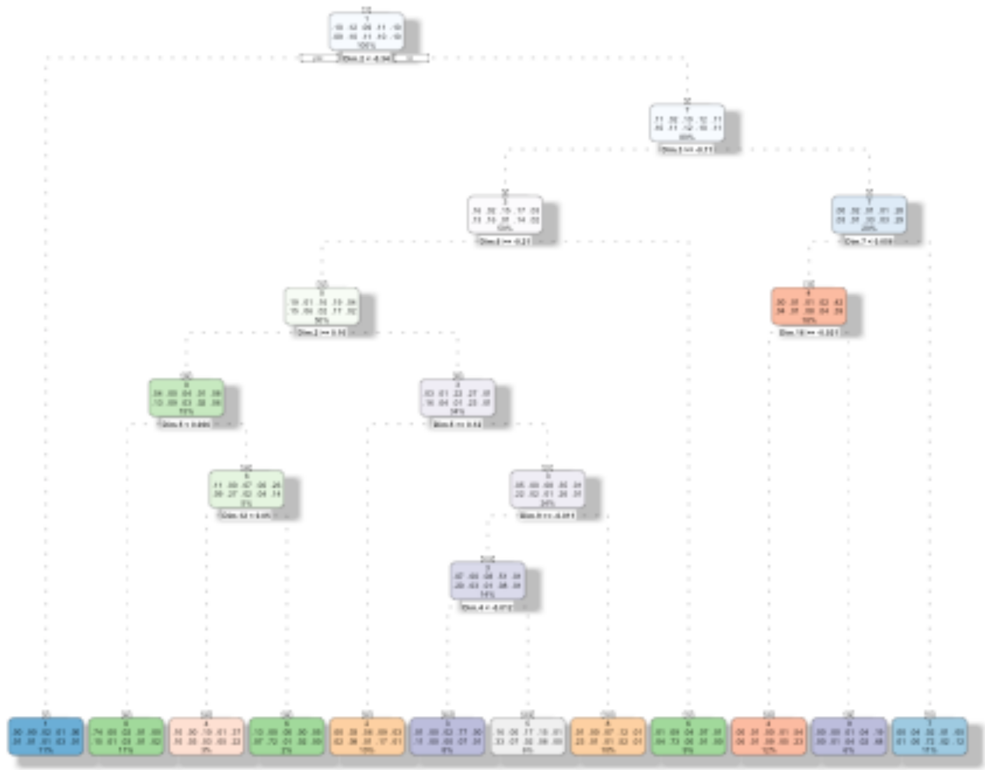
Decision-Trees

For this exercise, we will split the data (sampled data, rather than the original data) with 80:20 rule and then verify the performance of this model.

Model DT-1

```
# 80% of first N rows is train-data
train_rows <- as.integer(0.8 * dim(digits_final)[1])
digits_train <- digits_final[1:train_rows,]
dim(digits_train)

## [1] 8400   31

# 20% of remaining is test-data
digits_test <- digits_final[(train_rows+1):dim(digits_final)[1],]
dim(digits_test)

## [1] 2100   31

train_dt1 <- rpart(digits_all.label~ ., data=digits_train, method='class')
printcp(train_dt1)
```

```
##
## Classification tree:
## rpart(formula = digits_all.label ~ ., data = digits_train, method =
"class")
##
## Variables actually used in tree construction:
## [1] Dim.12 Dim.16 Dim.2  Dim.3  Dim.4  Dim.5  Dim.6  Dim.7  Dim.9
##
## Root node error: 7395/8400 = 0.88036
##
## n= 8400
##
##          CP nsplit rel error  xerror      xstd
## 1 0.102028      0   1.00000 1.00000 0.0040223
## 2 0.081880      2   0.79594 0.79621 0.0056743
## 3 0.070047      4   0.63218 0.63029 0.0061594
## 4 0.055037      5   0.56214 0.56498 0.0061968
## 5 0.045030      6   0.50710 0.51156 0.0061662
## 6 0.034483      7   0.46207 0.46680 0.0060978
## 7 0.012711      8   0.42759 0.43638 0.0060283
## 8 0.010683      9   0.41487 0.42461 0.0059963
## 9 0.010000     11   0.39351 0.41298 0.0059617

# Plot rpart using fancyRpartPlot
fancyRpartPlot(train_dt1)
```



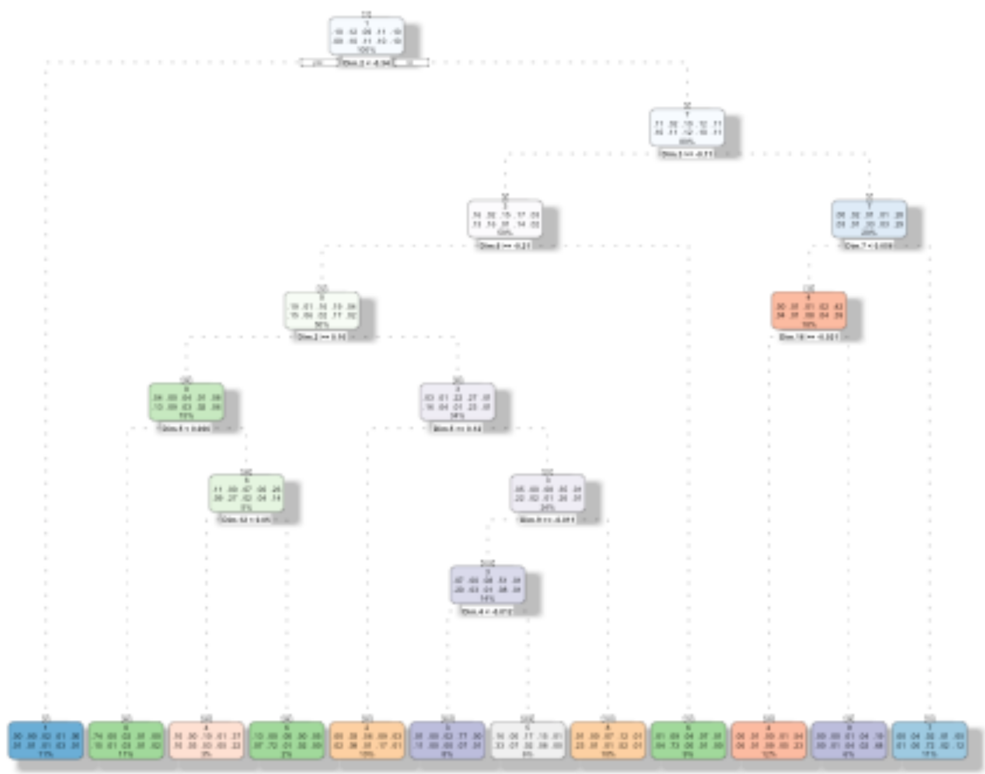
Rattle 2020-Aug-29 23:36:15 venkatasharatsripada

```
# Predict labels using Decision Tree
predict_dt1 <- predict(train_dt1, digits_test, type='class')

# Plot confusion matrix
comp_table <- data.frame(Actual=digits_test$digits_all.label,
Predicted=predict_dt1)
matrix_dt1 <- confusionMatrix(as.factor(comp_table$Predicted),
as.factor(comp_table$Actual))

print(matrix_dt1$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.6519048      0.6129176      0.6310915      0.6722931      0.1104762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

With an accuracy of ~62%, this seems clearly a less accurate model. Next, let us attempt to prune the tree to reduce the node complexity and overall run-time.

```
dt1_prune <- prune(train_dt1,
cp=train_dt1$cptable[which.min(train_dt1$cptable[,"xerror"]),"CP"])
```

```r
# Visualize pruned tree
fancyRpartPlot(dt1_prune)
```



Rattle 2020-Aug-29 23:36:17 venkatasharatsripada

```r
# Predict labels based on the pruned tree
predict_dt1_prune <- predict(dt1_prune, digits_test, type='class')

# Plot confusion matrix
comp_table <- data.frame(Actual=digits_test$digits_all.label,
Predicted=predict_dt1_prune)
matrix_dt1_prune <- confusionMatrix(as.factor(comp_table$Predicted),
as.factor(comp_table$Actual))

print(matrix_dt1_prune$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.6519048      0.6129176      0.6310915      0.6722931      0.1104762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

Pruned tree and actual decision tree turned out to be similar and thus yielded no benefits

Random Forests

Random Forests generates a bunch of bootstrapped sub-trees and combining the results in an Ensemble method.

```
train_dt2 <- randomForest(x=digits_train[2:ncol(digits_train)],
y=digits_train$digits_all.label, data = digits_train,
                          ntree=100, mtry=2, importance=TRUE)

# Predict labels based Random Forest algorithm
predict_dt2 <- predict(train_dt2, digits_test)

# Plot confusion matrix
comp_table <- data.frame(Actual=digits_test$digits_all.label,
Predicted=predict_dt2)
matrix_dt2 <- confusionMatrix(as.factor(comp_table$Predicted),
as.factor(comp_table$Actual))

print(matrix_dt2$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9376190      0.9306534      0.9264122      0.9475842      0.1104762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

By far, the best accuracy so far with prediction accuracy at ~92%

KNN

```
# To prevent model over-fitting, re-model training-set
# Reduce number of samples
percent <- 0.15
set.seed(275)
digitsplit_knn <- sample(nrow(digits_all), nrow(digits_all)*percent)
digits_final_knn <- digits_all[digitsplit_knn,]

# Examine the final data-frame we will use for knn modeling
dim(digits_final_knn)

## [1] 6300  785

# Slice the data-set
N <- nrow(digits_final_knn)
kfolds_knn <- 2
set.seed(30)
holdout_knn <- split(sample(1:N), 1:kfolds_knn)

# Start with some finite k_guess
k_guess <- round(sqrt(N)/10)

all_results <- data.frame(Actual=c(), Predicted=c())
```

```r
for (k in 1:kfolds_knn) {
  digits_final_test <- digits_final_knn[holdout_knn[[k]],]
  digits_final_train <- digits_final_knn[-holdout_knn[[k]], ]

  digits_final_no_label <- digits_final_test[-c(1)]
  digits_final_label <- digits_final_test[c(1)]

  predict_knn <- knn(train=digits_final_train, test=digits_final_test,
cl=digits_final_train$label,
                     k=k_guess)

  # Put results in each iteration in all_results
  all_results <- rbind(all_results,
data.frame(Actual=digits_final_label$label, Predicted=predict_knn))

}

# Get the overall accuracy for k=7
matrix_knn <- confusionMatrix(as.factor(all_results$Predicted),
as.factor(all_results$Actual))
print(matrix_knn$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9114286      0.9014342      0.9041413      0.9183322      0.1201587
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

The accuracy for knn is at ~91%. Repeating the test for k=3, 5, 8 yielded the following results:

- k=3, accuracy - ~92%
- k=5, accuracy - ~92%

SVMs

```r
all_results <- data.frame(Actual=c(), Predicted=c())
for (k in 1:kfolds) {
  digits_final_test <- digits_final[holdout[[k]],]
  digits_final_train <- digits_final[-holdout[[k]], ]

  digits_final_no_label <- digits_final_test[-c(1)]
  digits_final_label <- digits_final_test[c(1)]

  train_svm <- svm(digits_final_train$digits_all.label ~ .,
digits_final_train, na.action = na.pass)

  # Predict using svm modeling
  predict_svm <- predict(train_svm, digits_final_no_label, type=c('class'))
```

```
  # Put results in each iteration in all_results
  all_results <- rbind(all_results,
data.frame(Actual=digits_final_label$digits_all.label,
Predicted=predict_svm))

}

# Get the overall accuracy for SVM
matrix_svm <- confusionMatrix(as.factor(all_results$Predicted),
as.factor(all_results$Actual))
print(matrix_svm$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9686667      0.9651633      0.9651541      0.9719161      0.1164762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

The accuracy here is pretty good at 96.53% and is proving to be the best of the machine-learning models.

KSVMs

Finally, let's run the data-set through the various KSVMs types, namely:

- linear
- polynomial
- sigmoid

```
# Let's start with experimenting with kernel SVM Linear modeling
all_results <- data.frame(Actual=c(), Predicted=c())
for (k in 1:kfolds) {
  digits_final_test <- digits_final[holdout[[k]],]
  digits_final_train <- digits_final[-holdout[[k]], ]

  digits_final_no_label <- digits_final_test[-c(1)]
  digits_final_label <- digits_final_test[c(1)]

  train_ksvm <- svm(digits_final_train$digits_all.label ~ .,
digits_final_train, kernel='linear',
                   na.action = na.pass)

  # Predict using KSVM modeling
  predict_ksvm <- predict(train_ksvm, digits_final_no_label, type=c('class'))

  # Put results in each iteration in all_results
  all_results <- rbind(all_results,
data.frame(Actual=digits_final_label$digits_all.label,
Predicted=predict_ksvm))
```

```
}

# Get the overall accuracy for KSVM (Linear)
matrix_ksvm <- confusionMatrix(as.factor(all_results$Predicted),
as.factor(all_results$Actual))
print(matrix_ksvm$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9124762      0.9026751      0.9069081      0.9178143      0.1164762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN

# Repeat the experiment with kernel SVM as Polynomial
all_results <- data.frame(Actual=c(), Predicted=c())
for (k in 1:kfolds) {
  digits_final_test <- digits_final[holdout[[k]],]
  digits_final_train <- digits_final[-holdout[[k]], ]

  digits_final_no_label <- digits_final_test[-c(1)]
  digits_final_label <- digits_final_test[c(1)]

  train_ksvm <- svm(digits_final_train$digits_all.label ~ .,
digits_final_train, kernel='polynomial',
                   na.action = na.pass)

  # Predict using KSVM modeling
  predict_ksvm <- predict(train_ksvm, digits_final_no_label, type=c('class'))

  # Put results in each iteration in all_results
  all_results <- rbind(all_results,
data.frame(Actual=digits_final_label$digits_all.label,
Predicted=predict_ksvm))

}

# Get the overall accuracy for KSVM (Linear)
matrix_ksvm <- confusionMatrix(as.factor(all_results$Predicted),
as.factor(all_results$Actual))
print(matrix_ksvm$overall)

##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##      0.9686667      0.9651632      0.9651541      0.9719161      0.1164762
## AccuracyPValue  McnemarPValue
##      0.0000000            NaN
```

The accuracy with kernel='polynomial' yielded marginally better results at 96.78% vs linear at 91.39%.

```
# Repeat the experiment with kernel SVM as Sigmoid
all_results <- data.frame(Actual=c(), Predicted=c())
for (k in 1:kfolds) {
  digits_final_test <- digits_final[holdout[[k]],]
  digits_final_train <- digits_final[-holdout[[k]], ]

  digits_final_no_label <- digits_final_test[-c(1)]
  digits_final_label <- digits_final_test[c(1)]

  train_ksvm <- svm(digits_final_train$digits_all.label ~ .,
digits_final_train, kernel='sigmoid',
                   na.action = na.pass)

  # Predict using KSVM modeling
  predict_ksvm <- predict(train_ksvm, digits_final_no_label, type=c('class'))

  # Put results in each iteration in all_results
  all_results <- rbind(all_results,
data.frame(Actual=digits_final_label$digits_all.label,
Predicted=predict_ksvm))

}

# Get the overall accuracy for KSVM (Linear)
matrix_ksvm <- confusionMatrix(as.factor(all_results$Predicted),
as.factor(all_results$Actual))
print(matrix_ksvm$overall)
```

```
##       Accuracy          Kappa  AccuracyLower  AccuracyUpper   AccuracyNull
##   7.932381e-01   7.700444e-01   7.853634e-01   8.009498e-01   1.164762e-01
## AccuracyPValue  McnemarPValue
##   0.000000e+00   6.118680e-17
```

By far, KSVM - Sigmoid model yielded the worst results at 79.62%

Conclusion

The data-set related to images of written digits, was run through all the machine learning algorithms covered within the course-work for IST-707 - Decision Tree, Naive Bayes, KNN, SVMs, KSVMs and Random Forest. While each of them have have their strengths/weaknesses, I’ve ordered them in their ability to accurately predict data (limited to the nature of this data-set), based on the experiments presented thus far:

- KSVMs - Polynomial
- SVMs
- KSVMs - Linear
- Random Forest

- KNN
- Naive Bayes
- KSVMs - Sigmoid
- Decision Trees