Handwriting Recognition using naive Bayes, Decision Trees, SVMs, kNNs, and Random Forest

Sean Dunn

May 29, 2020

Table of Contents

```
# Packages used
# caret      - Classification and regression training
# class      - Various functions for classification, including k-nearest
neighbour, Learning Vector Quantization and Self-Organizing Maps
# CORElearn  - A suite of machine learning algorithms
# dplyr      - Tool for working with data frame like objects, both in memory
and out of memory.
# e1071      - Functions for naive Bayes classifier, among others
# FactoMineR - Multivariate Exploratory Data Analysis and Data Mining
# ggplot2    - A system for 'declaratively' creating graphics, based on "The
Grammar of Graphics".
# magrittr - data manipulation
# MASS       - Functions and datasets to support Venables and Ripley, `Modern
Applied Statistics with S'' (4th edition, 2002).
# mlr        - Interface to a large number of classification and regression
techniques, including machine-readable parameter descriptions.
# naivebayes - Implementation of the Naive Bayes classifier
# plyr       - A set of tools to break a big problem down into manageable
pieces,
# pROC       - Display and Analyze ROC Curves
# randomForest - implements Breiman's random forest algorithm (based on
Breiman and Cutler's original Fortran code) for classification and regression
# RColorBrewer Color package
# rattle     - Collection of utilities functions
# rpart      - Recursive partitioning for classification, regression and
survival trees
# rpart.plot - Extends plot.rpart() and text.rpart() in the 'rpart' package
# Thermimage - Rotating matrices
# tree       - Classification and regression trees
```

All code used in this analysis and commentary can be found with a static link to an RPubs website for code navigation and commentary:

httpd://rubs.com/consultation/MNIST

Introduction

Imagine a time when checks could not be deposited in an ATM, let alone photos taken at the customer's convenience for depositing via a mobile device. Also, consider the enormous volume of tax forms the IRS processes annually. These days, we take such technology for granted. Vast numbers of checks with digits and words are analyzed and automatically processed routinely. How is all of this possible?

While mechanical, optical character recognition had its early days in 1809 when the first patents for devices to aid the blind were awarded. In 1912, there was a machine that read characters, converted them into standard telegraph code, and then transmitted telegraphic messages over wires without human intervention. Further progress ensued, and in 1914 an OCR device called the optophone that produced sounds. Each sound corresponded to a specific letter or character, which allowed visually impaired people to "read" the printed

material. Developments in OCR continued throughout the 1930s, gaining prominence with the beginnings of the computer industry in the 1940s. OCR became more focused in the 1950s. Ray Kurzweil, who is known is known as the inventor of OCR, in the 70's used a program which guessed at each letter based on stored information and tries to make words from the images.[1]

Over the last several decades, there has been progress within computer vision, the field of computer science that focuses on replicating parts of human vision to enable computers to identify and process objects in images. Computer vision is centered around pattern recognition. Until recently, computer vision only worked in a limited capacity, but advances came in computer vision have accelerated, particularly with the ImageNet challenges that started a revolution including Geoffrey Hinton et al.'s submission of a deep convolutional neural network architecture called AlexNet which was a breakthrough both for neural nets and just for recognition in general.[2,3,4]

Since its release in 1999, this classic data set of handwritten images has served as the basis for bench marking classification algorithms. The Modified National Institute of Standards and Technology (MNIST) database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training data set, while the other half of the training set and the other half of the test set were taken from NIST's testing data set. The focus of this paper is to use naive Bayes, Decision Trees, SVMs, kNNs, and Random Forest to classify digits in the MNIST data set.

Despite the progress and low error rates from neural nets and convolutional neural nets (CNN) with sub-1% error rates, by some studies, for the MNIST database[5], there are other fundamental classifiers to learn with respectable results with MNIST, such as linear classifiers with around 7% error rates, kNN with 0.52, Random Forest around 2%, and SVM around 0.56, and that is the focus of this paper using non-connectivist methods for classification of the MNIST data set. The focus of this paper is to investigate classification algorithms; specifically, naive Bayes, Decision Trees, SVMs, kNNs, and Random Forest, to classify digits in the MNIST data set.

---

[1] https://www.afb.org/aw/5/5/14692

[2] https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/

[3] http://www.edubilla.com/invention/optical-character-recognition/

[4] https://history-computer.com/ModernComputer/Basis/OCR.html

[5] https://github.com/Matuzas77/MNIST-0.17

Analysis and Models

About the Data

```
# Load the MNIST data from Kaggle file**
# Set the working directory
# The data was loaded and returned no missing values.
setwd("~/")
mnist <- read.csv("Kaggle-digit-train.csv")
# nrow(mnist) - sum(complete.cases(mnist)) # determine if there are any
uncomplete cases - returns 0
# any(is.na(mnist)) # checks for NAs and returns FALSE
```

Again, since its release in 1999, this classic data set of handwritten images has served as the basis for bench-marking classification algorithms. The data set includes monochromatic images of hand-drawn digits, zero tp nine. Each image is 28 pixels by 28 pixels for a total of 784 pixels. There is an integer pixel-value associated with each pixel which ranges from 0 to 255, lightest to darkest, respectively.[6]

The training data set, has 785 columns and 42,000 observations. The first column, called "label", is the digit that was drawn by the user. The rest of the columns contain the pixel-values of the associated image. Each pixel column in the training set has a name like pixelx, where x is an integer between 0 and 783, inclusive. To locate this pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27, inclusive. Then pixelx is located on row i and column j of a 28 x 28 matrix, (indexing by zero).[7]. The test set has 28,000 observations.

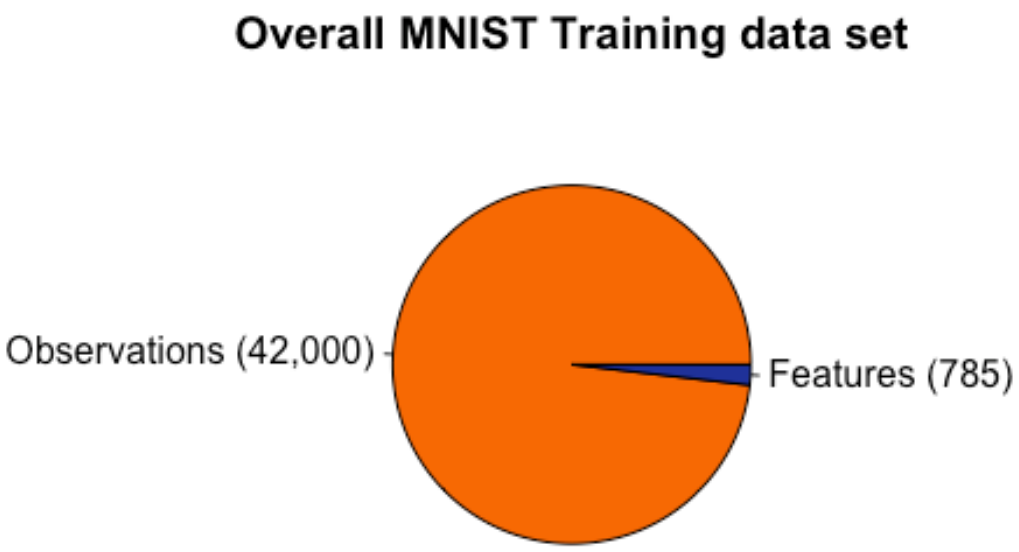
The number of observations related to each digit is represented by the Exhibit 1: Observations Per Digit and Exhibit 2: Bar chart of Digit Observations.

Exploratory data analysis

```
col.vec = c("#F76900", "#203299")
slices <- c(42000, 785)
lbls <- c("Observations (42,000)", "Features (785)")
pie(slices, labels = lbls, main="Overall MNIST Training data set", col =
col.vec)
```

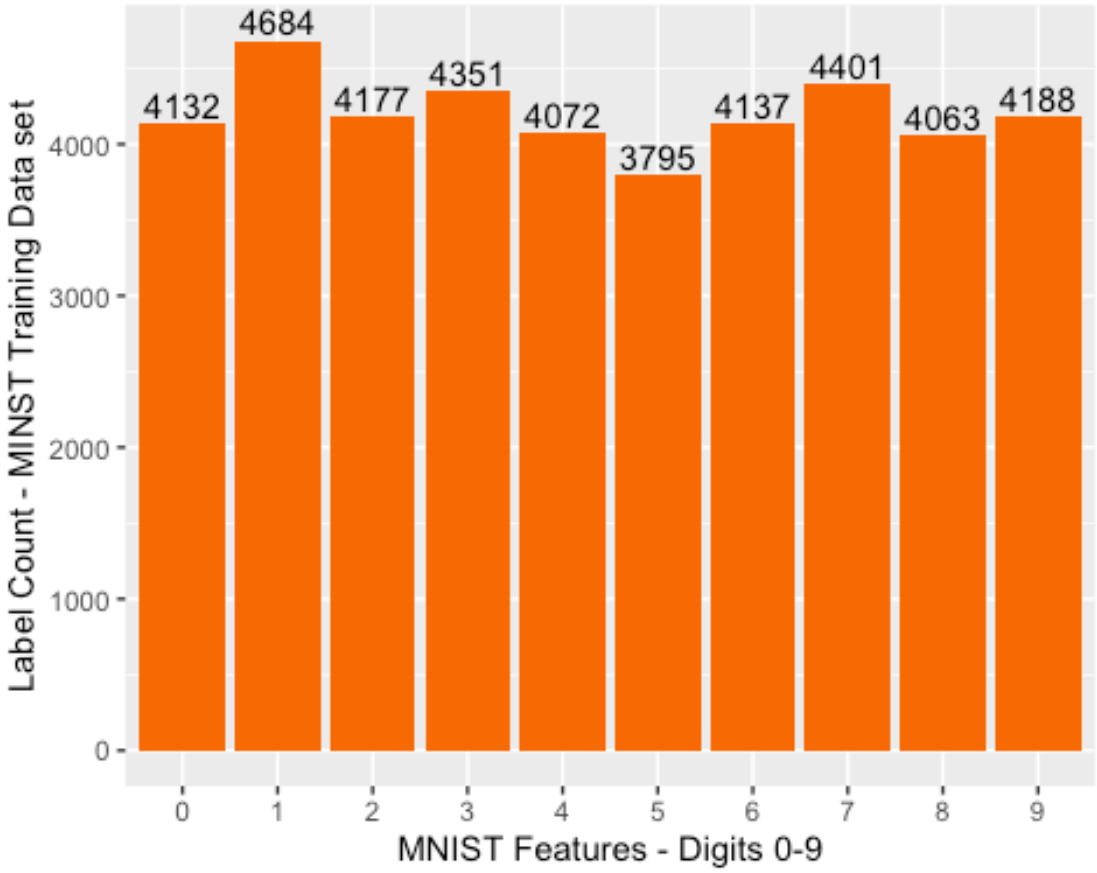
[6] https://www.kaggle.com/aconsapart/minst

[7] https://www.kaggle.com/aconsapart/minst



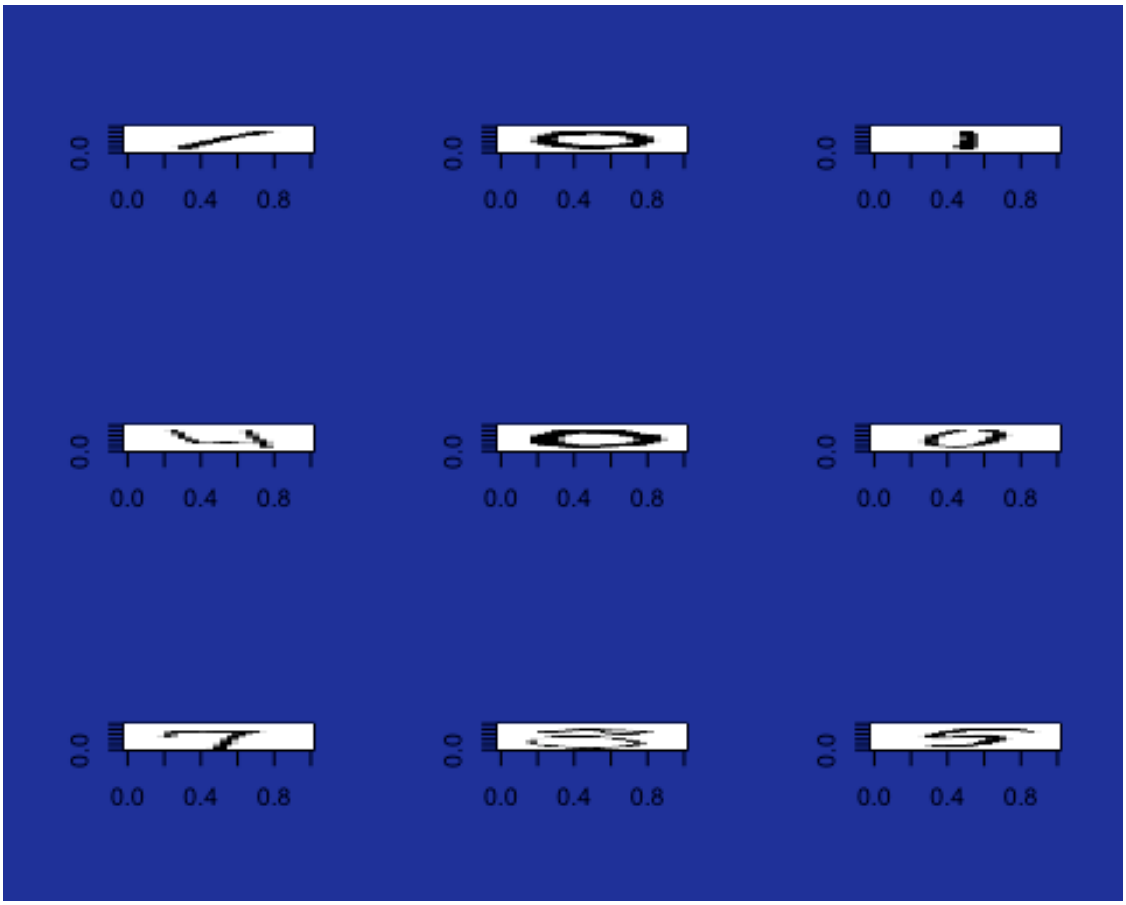
Visualize the label frequency counts - demonstrates balanced data

```
# ggplot2::theme_set(theme_gray()) # Set all ggplot to theme gray
minst.label.table <- as.data.frame(table(mnist$label))
names(minst.label.table)[1] <- "label"
# Create a barchart of the digit frequency of the original dataset
ggplot(minst.label.table, aes(label, Freq)) + geom_bar(stat = "identity",
fill = "#F76900") +
  xlab("MNIST Features - Digits 0-9") + ylab("Label Count - MINST Training
Data set") + geom_text(aes(label=Freq), position=position_dodge(width=0.9),
vjust=-0.25)
```



Visualize the number images from the first 9 rows of the data set

```
par(mfrow = c(3,3), bg ="#203299")
loop.vector <- 1:9
for (i in 1:9){
    im <- matrix((mnist[i,2:ncol(mnist)]), nrow=28, ncol=28)
    im_numbers <- apply(im, 1, as.numeric)
    im_numbers2  <- rotate270.matrix(im_numbers)
    image(im_numbers2, col=gray(seq(1,0, length = 255)))
}
```



Pre-processing steps

The pre-processing for the models used for naive Bayes, Decision Trees, SVMs, kNNs, and Random Forest included changing the labels (digits 0-9) from an integer to a factor with 10 levels.

```
mnist.df<- mnist # create new df to factor
mnist.df$label<- as.factor(mnist.df$label) # need to create factors
# str(mnist.df$label) confirm factor  Factor w/ 10 levels "0","1","2","3",..:
2 1 2 5 1 1 8 4 6 4 ..
```

The train and test data sets for each fold were built. The caret library allows the creation of test and training partitions (p) by creating balanced splits of the data.[8,9,10]

```
# In this case, partition at 75/25% split
set.seed(4321)
```

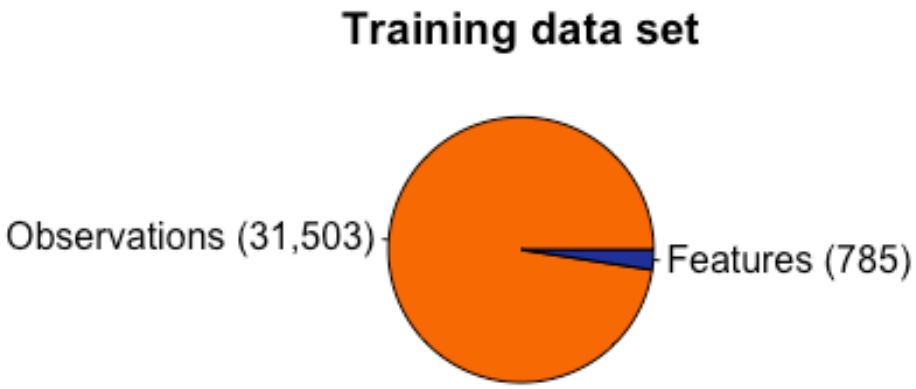
---

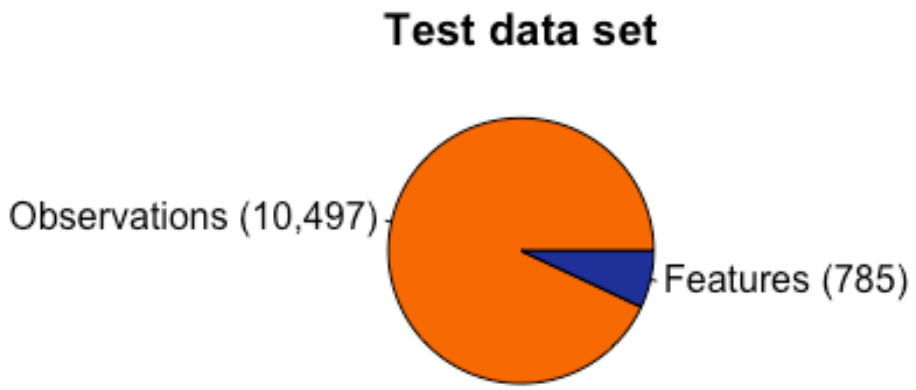
[8] https://www.guru99.com/r-decision-trees.html

[9] https://www.edureka.co/blog/implementation-of-decision-tree/

[10] https://www.edureka.co/blog/decision-tree-algorithm/

```r
Partition<- createDataPartition(y = mnist.df$label, p = 0.75, list = FALSE)
# Create the train sample
train<- mnist.df[Partition,]
# dim(train) # [1] 31503   785
# table(train$label)
# 0    1    2    3    4    5    6    7    8    9
# 3099 3513 3133 3264 3054 2847 3103 3301 3048 3141
# Create the test sample
test<- mnist.df[-Partition,]
# dim(test) # [1] 10497   785
# table(test$label)
#    0    1    2    3    4    5    6    7    8    9
# 1033 1171 1044 1087 1018  948 1034 1100 1015 1047
# checking the difference of the dimensions of split to ensure balance
# round(prop.table(table(train$label))*100, digits = 2) -
round(prop.table(table(test$label))*100, digits = 2)
 #  0     1     2     3     4     5     6     7     8     9
 # 0.00 -0.01  0.00  0.00 -0.01  0.01  0.00  0.00  0.01  0.00
# Compare with original data set labels, confirms appropriate balance.
# round(prop.table(table(mnist.df$label))*100, digits = 2) -
round(prop.table(table(train$label))*100, digits = 2)
 #    0     1     2     3     4     5     6     7     8     9
 # 0.00  0.00  0.00  0.00  0.01  0.00  0.00  0.00 -0.01  0.00

set.seed(4321)
par(mfrow = c(2,1), mar = c(0,0,2,0))
col.vec = c("#F76900", "#203299")
slices <- c(31503, 785)
lbls <- c("Observations (31,503)", "Features (785)")
pie(slices, labels = lbls, main="Training data set", col = col.vec)
col.vec = c("#F76900", "#203299")
slices <- c(10497, 785)
lbls <- c("Observations (10,497)", "Features (785)")
pie(slices, labels = lbls, main="Test data set", col = col.vec)
```





Models

Decision tree Models

The decision tree model is developed using rpart. The method "class" was used for a decision tree. Minisplit is the minimum number of observations that must exist in a node in order for a split to be attempted, which initially is set to one. The complexity parameter, cp, provides a threshold in which any split that does not decrease the overall lack of fit by a factor of cp is not attempted. Other arguments are removed for this particular analysis.[11]

Recursive partitioning is a fundamental tool in data mining. It helps us explore the structure of a set of data, while developing easy to visualize decision rules for predicting a categorical (classification tree) or continuous (regression tree) outcome. This section briefly describes CART modeling, conditional inference trees, and random forests.

It follows these steps:

1. Grow the Tree - using: rpart(formula, data=, method=,control=) where
2. Examine the results - Using a function such as summary(fit)

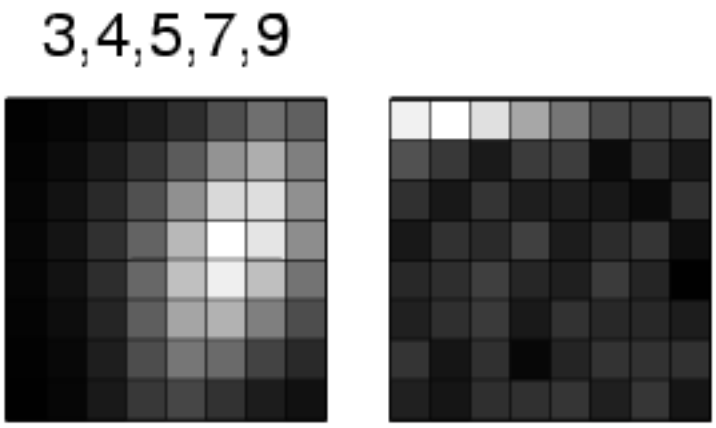
---

[11] https://stat.ethz.ch/R-manual/R-devel/library/rpart/html/rpart.control.html

3. prune tree - Prune back the tree to avoid over-fitting the data. Select a tree size that minimizes the cross-validated error.

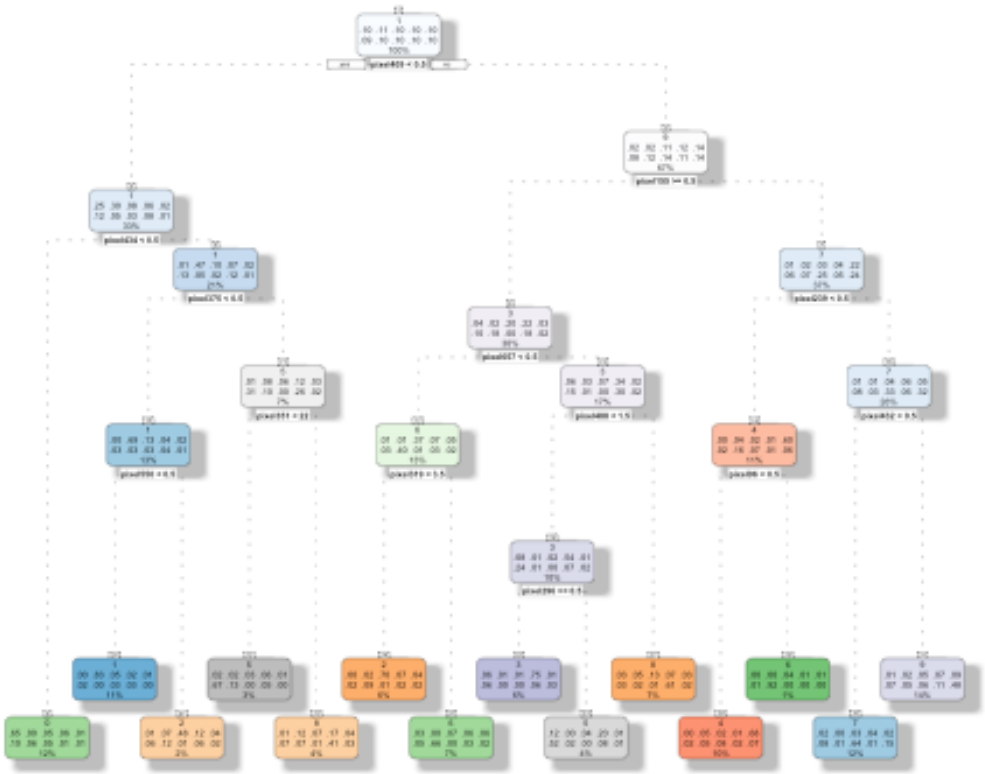
```
# Building the classification tree with rpart
set.seed(4321)
dt.model.1 <- rpart(label~., data=train, method = "class")
# summary(tree)
```

Decision Tree - Model 1: Baseline Decision tree



In the first model, the label is predicted from all features.

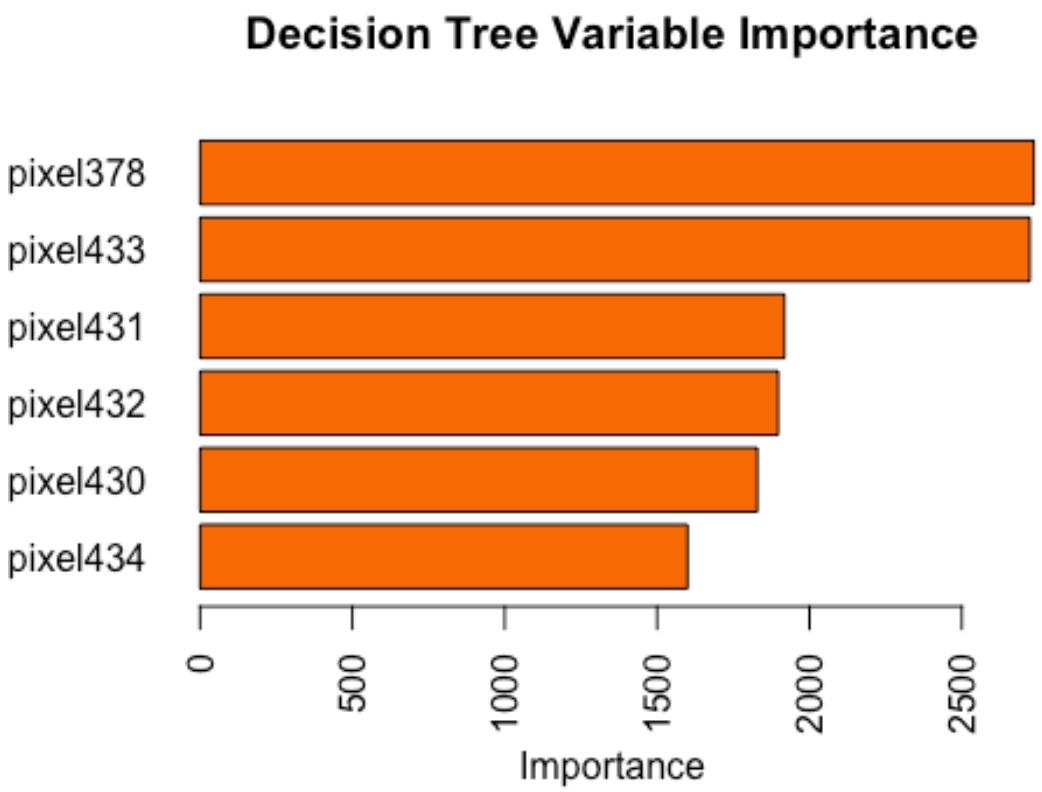
```
# Visualize the decision tree
fancyRpartPlot(dt.model.1, sub = "Classification Decision Tree - MNIST
training data")
```



Classification Decision Tree - MNIST training data

Evaluate decision tree variable importance (utilizing the caret package). It's clear the feature pixel378 has the highest from the bar plot.

```
set.seed(4321)
imp <- varImp(dt.model.1, scale = TRUE) # most important variables
imp$pixel <- rownames(imp)
imp <- imp[order(imp$Overall),]

imp.df <- as.data.frame(tail(imp))

barplot(imp.df$Overall,
        main = "Decision Tree Variable Importance",
        xlab = "Importance",
        names.arg = imp.df$pixel,
        las = 2,
        col = "#F76900",
        horiz = TRUE)
```



Decision tree model 1 classification results using a confusion matrix.

```
set.seed(4321)
pred<- predict(object=dt.model.1,test[-1],type="class")
dt.model.1.cm <- confusionMatrix(pred, test$label)
dt.model.1.cm$table
```

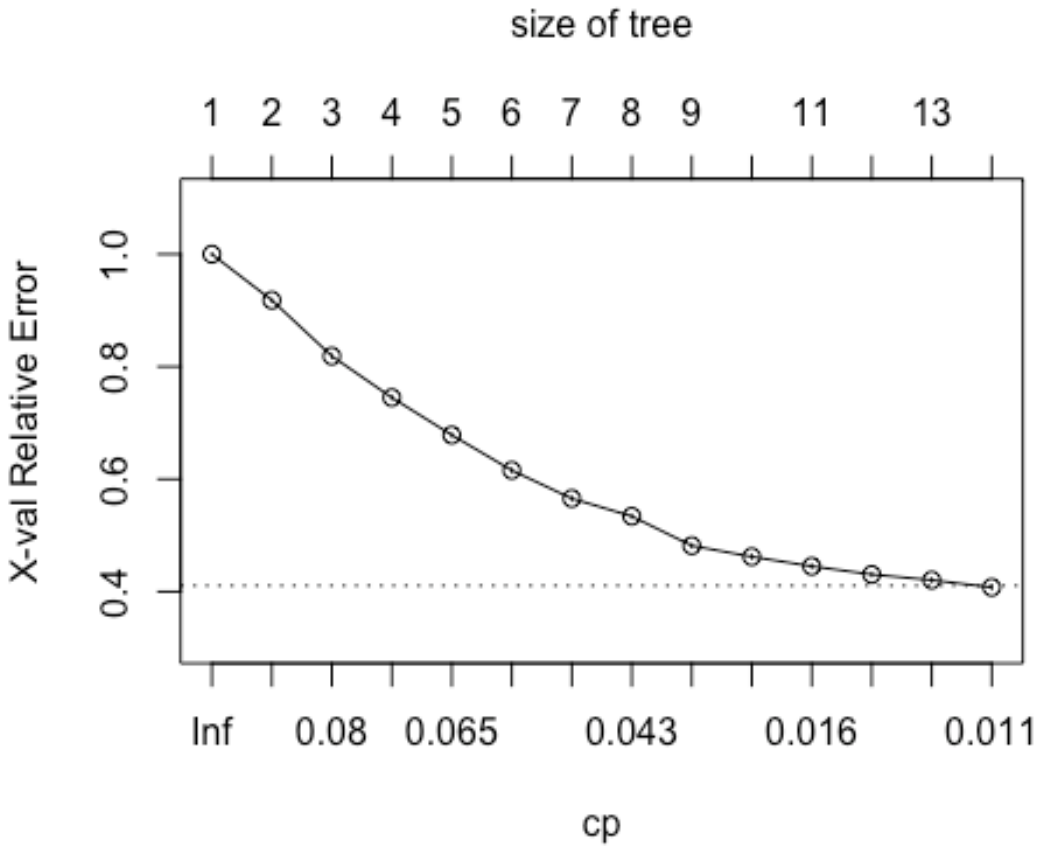
```
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 845   3  66  80  12 112  69  76  11  23
##          1   1 955  72  15   8  19   6  38  46  12
##          2   2  32 541  70  39  26 106   9  22  15
##          3  40   4   7 487   1  45   1   0  44  17
##          4   6  52  26  19 686  15  47  96  12  64
##          5  50   4  35  88  13 434  52   1  61  15
##          6  24   3  56  46  64  37 617   0  22  17
##          7  29  11  35  56  14 102  14 786  10 192
##          8  25  68 149 125  44  53  64   5 648  34
##          9  11  39  57 101 137 105  58  89 139 658
```

```
dt.model.1.acc <- round(dt.model.1.cm$overall[1]*100,2)
```

Decision tree model 1's accuracy is 63.42%.

One can tune the parameters by pruning. Another approach over trial and error is validation of the decision tree parameters using the printcp(x) function, where x is the rpart object. This function provides the optimal pruning based on the cp value. Then, the data is pruned to avoid over-fitting of the data. The convention is to have a small tree and the one with the least cross validated error given by the printcp() function (i.e. x-error or cross-validation error).[12] Plot the x-val relative error by the cp to determine optimal parameter adjustments.

```
plotcp(dt.model.1)
```

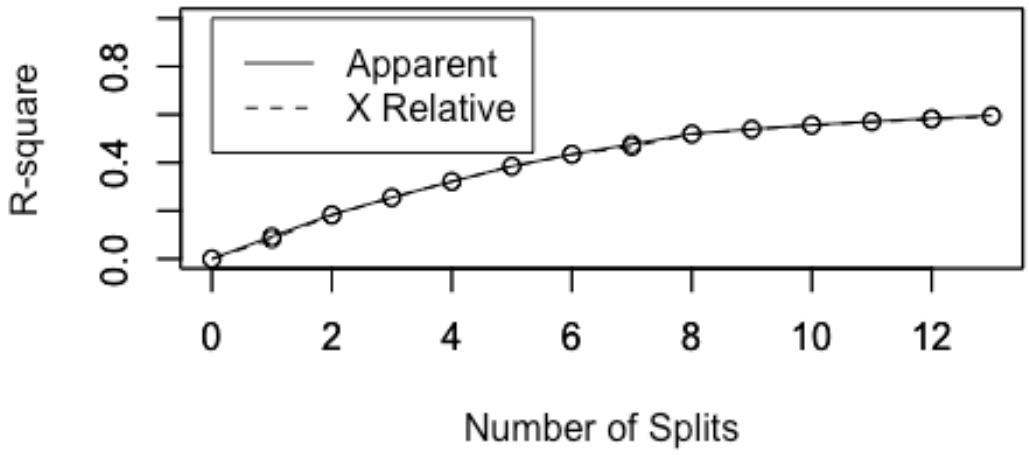


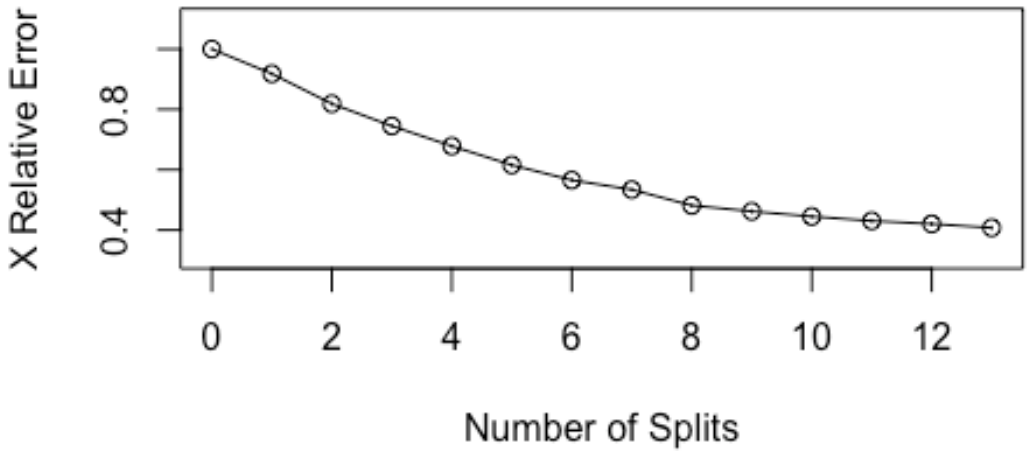
Rather than use stop criteria (e.g. maxdepth, etc.) in this case for pre-pruning, use post-pruning by evaluating the appropriate splits by plotting against r-squared and xerror. Select the lowest xerror for the parameter cp. Also, the printcp() function can be used.

```
##
## Classification tree:
## rpart(formula = label ~ ., data = train, method = "class")
##
## Variables actually used in tree construction:
```

[12] https://www.edureka.co/blog/implementation-of-decision-tree/

```
##  [1] pixel155 pixel239 pixel296 pixel319 pixel351 pixel375 pixel409
pixel432
##  [9] pixel434 pixel488 pixel550 pixel657 pixel96
##
## Root node error: 27990/31503 = 0.88849
##
## n= 31503
##
##          CP nsplit rel error  xerror      xstd
## 1  0.093140      0   1.00000 1.00000 0.0019960
## 2  0.090818      1   0.90686 0.91811 0.0024585
## 3  0.071025      2   0.81604 0.81876 0.0028235
## 4  0.066595      3   0.74502 0.74502 0.0029997
## 5  0.063380      4   0.67842 0.67842 0.0031029
## 6  0.049482      5   0.61504 0.61561 0.0031566
## 7  0.042730      6   0.56556 0.56556 0.0031706
## 8  0.042658      7   0.52283 0.53401 0.0031665
## 9  0.019578      8   0.48017 0.48164 0.0031375
## 10 0.017006      9   0.46059 0.46206 0.0031194
## 11 0.014541     10   0.44359 0.44505 0.0031005
## 12 0.013219     11   0.42905 0.43051 0.0030818
## 13 0.011147     12   0.41583 0.42047 0.0030676
## 14 0.010000     13   0.40468 0.40757 0.0030477
```





.

Decision Tree - Model 2: with pruning

Given the above parameter determination, change the cp parameter to 0.01

```
set.seed(4321)
dt.model.2 <- prune(dt.model.1, cp = 0.010000)

pred<- predict(object=dt.model.2,test[-1],type="class")
dt.model.2.cm <- confusionMatrix(pred, test$label)
dt.model.2.acc <- round(dt.model.2.cm$overall[1]*100,2)
dt.model.2.cm$table
```

```
##           Reference
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0 845   3  66  80  12 112  69  76  11  23
##          1   1 955  72  15   8  19   6  38  46  12
##          2   2  32 541  70  39  26 106   9  22  15
##          3  40   4   7 487   1  45   1   0  44  17
##          4   6  52  26  19 686  15  47  96  12  64
##          5  50   4  35  88  13 434  52   1  61  15
##          6  24   3  56  46  64  37 617   0  22  17
##          7  29  11  35  56  14 102  14 786  10 192
##          8  25  68 149 125  44  53  64   5 648  34
##          9  11  39  57 101 137 105  58  89 139 658
```

In this case, both the pre and post-pruned model's accuracy is equal at 63.42%.

Decision Tree - Model 3: Cross-validation evaluation of model 1

Cross validation (CV) is one of the technique used to test the effectiveness of a machine learning models. The evaluation provides a rotation estimation. CV we need to keep aside a sample/portion of the data on which is do not use to train the model, later use this sample for testing/validating. There are many methods. Unlike the train-test-split, which was applied earlier using caret. K-folds cross validation generally results in a less biased model compare to other methods, as it ensures that every observation from the original data set has the chance of appearing in training and test set. The k-folds re samples by randomly splitting the data in K-folds. Then, the error rate from this process is the mean of each fold error rate.[13,14]

```
n <- nrow(mnist)
K <- 3
size <- n%/%K
set.seed(4321)
rand_value <- runif(n)
rank <- rank(rand_value)
block <- (rank-1)%/%size+1
block <- as.factor(block)
all.err<- numeric(0)
for (k in 1:K) {
        # learn the model on all individuals except the k block
        model.1<- rpart(label~., data=mnist[block!=k,], method="class")
        # apply the model to the block number k
        pred.1<- predict(model.1, newdata=mnist[block==k,], type="class")
        # confusion matrix
```

[13] https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f

[14] http://www.sthda.com/english/articles/38-regression-model-validation/157-cross-validation-essentials-in-r/

```
        mc<- table(mnist$label[block==k],pred.1)
        # error rate
        err<- 1.0 - (mc[1,1]+mc[2,2])/sum(mc)
        # keep
        all.err<- rbind(all.err,err)
}
```

Overall, error rate is 82.85%, which is very large.

```
par(mar=c(.5,2,1,.5))
k.fold.error <-as.data.frame(all.err)
boxplot(k.fold.error, main="3-fold CV error rate")
```

Naïve Bayes - Model 1

Naïve Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naïve) independence assumptions between the features. They are among the simplest Bayesian network models. However, when coupled with kernel density estimation higher accuracy levels are achieved.[15] Pre-processing for a naïve Bayes

---

[15] https://en.wikipedia.org/wiki/Naive_Bayes_classifier

model includes removing the labels from the test data frame, and create a vector with those labels.[16]

Advantages: This algorithm requires a small amount of training data to estimate the necessary parameters. Naive Bayes classifiers are extremely fast compared to more sophisticated methods. Disadvantages: Naive Bayes is is known to be a bad estimator.[17]

```
test_nolabel<- test[,-1]
test_justlabel<- test$label
```

Remove labels from the test data frame, train, and test the model using naïve Bayes.

Confusion matrix for Naïve Bayes - Model 1

```
test_nolabel<- test[,-1]
test_justlabel<- test$label
NB_e1071<- naiveBayes(label~., data=train, na.action = na.pass)
# Testing the model
NB_e1071_Pred<- predict(NB_e1071, test_nolabel)
NB.p.df<- as.data.frame(NB_e1071_Pred)
table(NB_e1071_Pred,test_justlabel)

##               test_justlabel
## NB_e1071_Pred    0    1    2    3    4    5    6    7    8    9
##             0  956    1  142   89   49  128   11   15   25   11
##             1    1 1129   33   73    8   34   17   20  179   34
##             2    5    1  171    5    1    2    2    0    1    0
##             3    3    2   96  324    9   16    0    5    9    2
##             4    0    0    3    1   96    6    0    4    3    4
##             5    4    1   14    2    4   39    1    0    2    1
##             6   28   10  325   73  137   75  976   11   16    5
##             7    0    0    3    6    1    2    2  322    0    2
##             8   25   19  242  431  233  575   23   55  666   35
##             9   11    8   15   83  480   71    2  668  114  953

# Calculating accuracy
nb.model4 <- confusionMatrix(NB_e1071_Pred, test_justlabel)
nb.model4.acc <- round(nb.model4$overall[1]*100,2)
nb.model4$table

##            Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  956    1  142   89   49  128   11   15   25   11
##          1    1 1129   33   73    8   34   17   20  179   34
##          2    5    1  171    5    1    2    2    0    1    0
##          3    3    2   96  324    9   16    0    5    9    2
```

[16] http://uc-r.github.io/naive_bayes

[17] https://en.wikipedia.org/wiki/Naive_Bayes_classifier

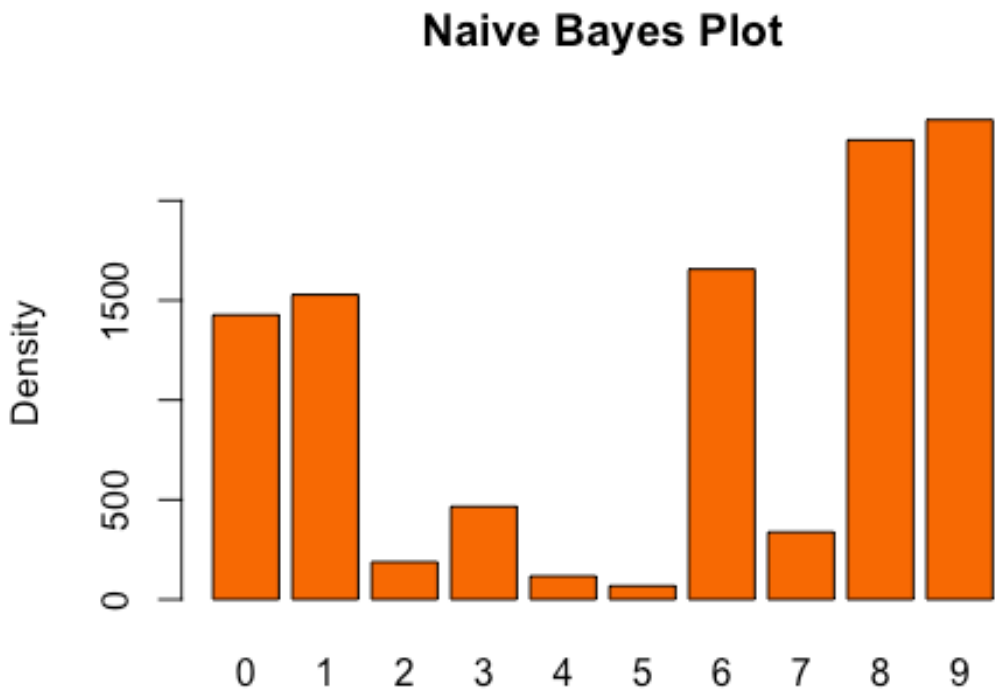
```
##          4    0    0    3    1   96    6    0    4    3    4
##          5    4    1   14    2    4   39    1    0    2    1
##          6   28   10  325   73  137   75  976   11   16    5
##          7    0    0    3    6    1    2    2  322    0    2
##          8   25   19  242  431  233  575   23   55  666   35
##          9   11    8   15   83  480   71    2  668  114  953
```

The accuracy of Model 4, the initial naïve Bayes is 53.65%.

```
rowSums(nb.model4$table[, c(1:10)])

##    0    1    2    3    4    5    6    7    8    9
## 1427 1528  188  466  117   68 1656  338 2304 2405
```

Plot of the predictions

```
plot(NB_e1071_Pred, ylab = "Density", main = "Naive Bayes Plot",
col='#F76900')
```



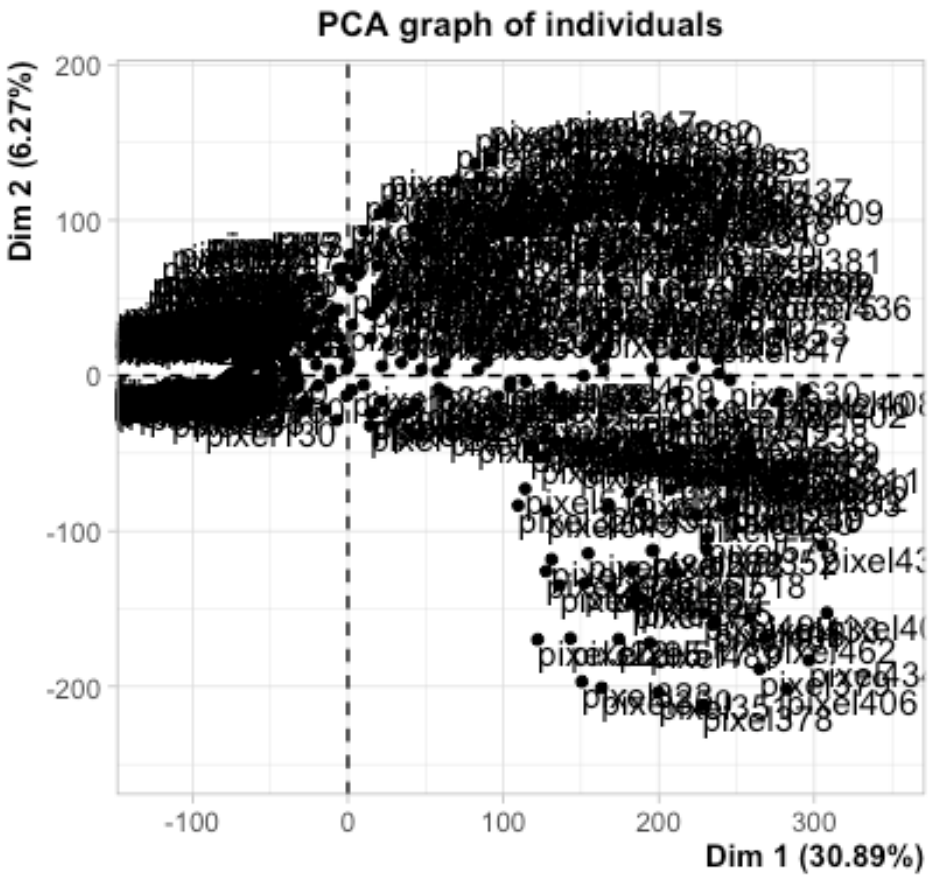
Naïve Bayes - Model 2: Principal Component Analysis at 5 dimensions

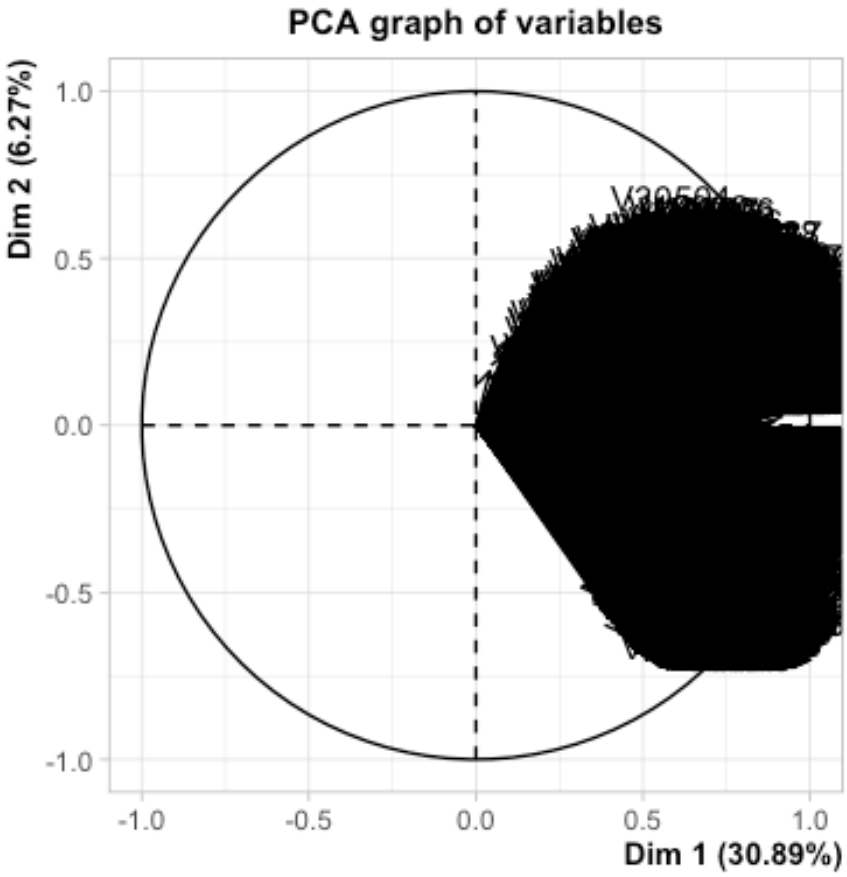
Dimensionality reduction and feature selection are important concepts when dealing with high dimensional data. The goal is to avoid the curse of high dimensionality. Principal Components Analysis is one way to select features based on variance. Example of reducing

the dimensionality from 784 to single or double digit features and compare results. The default number of dimensions kept in the results is 5 for FactoMineR PCA() function.[18]

```
DigitTotalDF <- mnist
DigitTotalDF$label<-as.factor(DigitTotalDF$label)
# dim(DigitTotalDF)

pca_digits = PCA(t(dplyr::select(DigitTotalDF,-label)))
```

[18] https://cran.r-project.org/web/packages/FactoMineR/FactoMineR.pdf

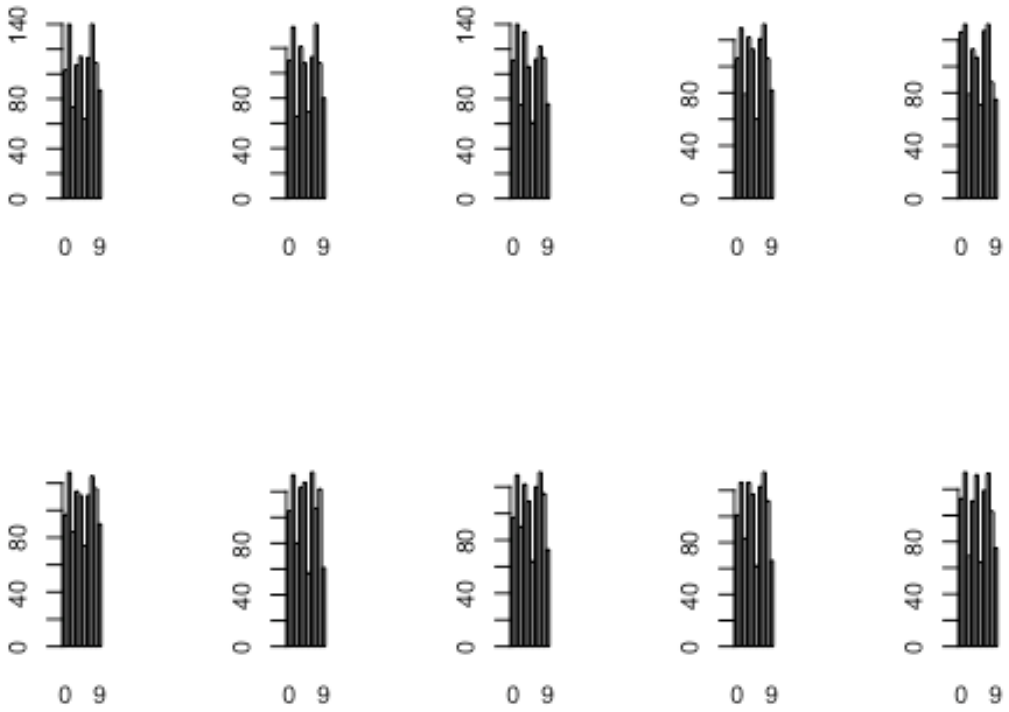

PCA graph of individuals
Dim 2 (6.27%)
Dim 1 (30.89%)
200
100
0
-100
-200
-100
0
100
200
300




```
# Feature Selection / Dim Reduction
# Feature Selection
# Create k-folds for k-fold cross validation
## Number of observations
N <- nrow(DigitDF)
## Number of desired splits
kfolds <- 10
## Generate indices of holdout observations
## Note if N is not a multiple of folds you will get a warning, but is OK.
holdout <- split(sample(1:N), 1:kfolds)
#head(holdout)
```

Build the train and test data sets for each fold. Next, train Naive Bayes Classifier on Train Set and test on Test Set. Lastly, present classification results using confusion matrix.

"10 trials Naive Bayes Plot Digit labels by Density (default dimensions retained = 5)"




```
mod.5.mean.acc.pca <- round(mean(unlist(AllAccuracy)*100),2)
```

Find the overall accuracy of Model 5 is 64.41

Naïve Bayes - Model 3: Principal Component Analysis at 15 dimensions

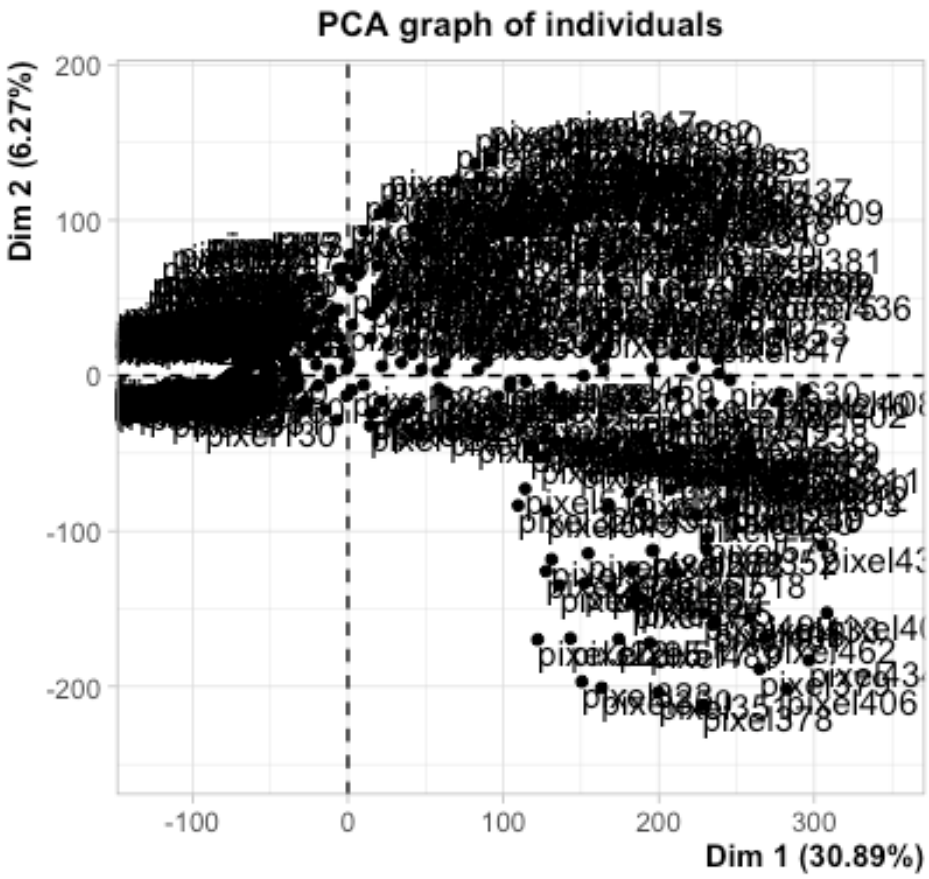
```
DigitTotalDF <- mnist
DigitTotalDF$label<-as.factor(DigitTotalDF$label)
```

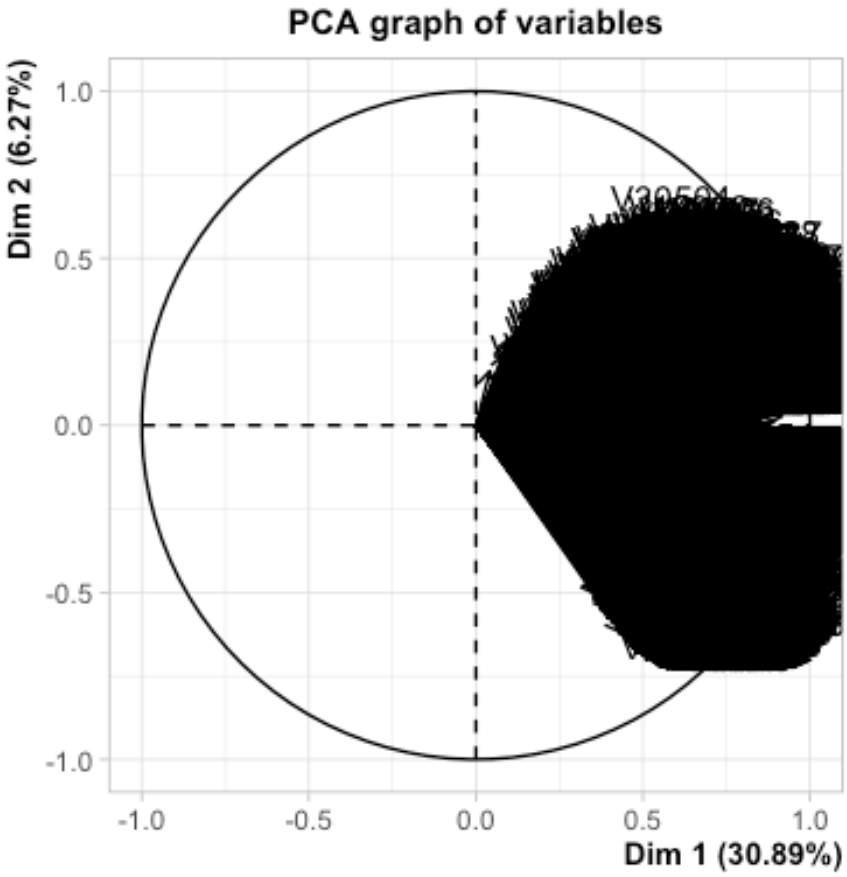
Change the dimensions by specifying, respectively, the indexes of the quantitative and qualitative variable.[19]

s

```
pca_digits = PCA(t(dplyr::select(DigitTotalDF,-label)), ncp = 15 )
```

[19] http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-principal-component-analysis-essentials/#specification-in-pca

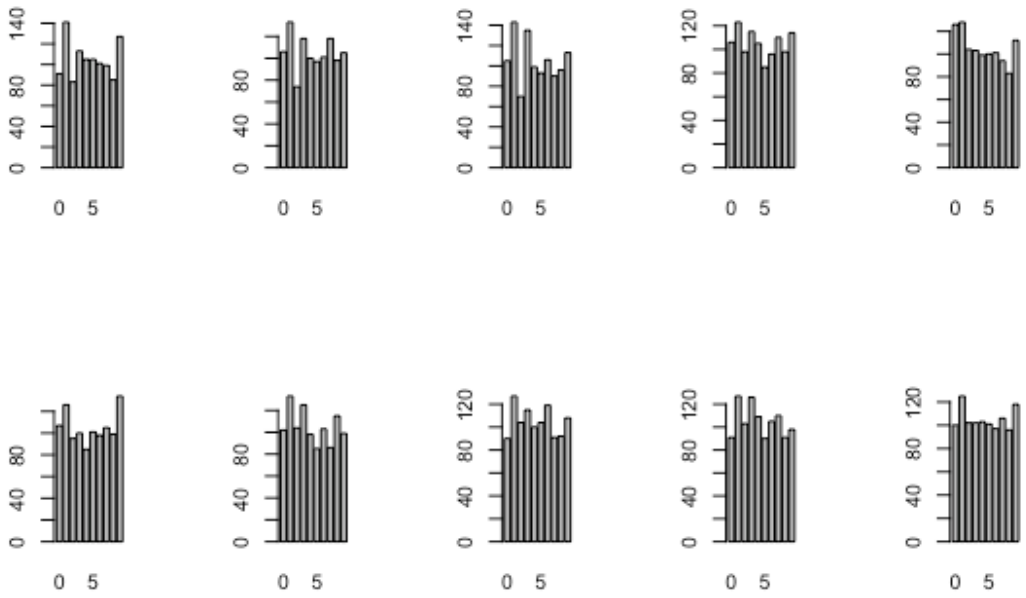

PCA graph of individuals
Dim 2 (6.27%)
Dim 1 (30.89%)
200
100
0
-100
-200
-100
0
100
200
300




```
DigitTotalDF = data.frame(DigitTotalDF$label,pca_digits$var$coord)

percent <- .25
set.seed(275)
DigitSplit <- sample(nrow(DigitTotalDF),nrow(DigitTotalDF)*percent)
DigitDF <- DigitTotalDF[DigitSplit,]
dim(DigitDF)

## [1] 10500    16

# Feature Selection / Dim Reduction
# Feature Selection
# Create k-folds for k-fold cross validation
## Number of observations
N <- nrow(DigitDF)
## Number of desired splits
kfolds <- 10
## Generate indices of holdout observations
## Note if N is not a multiple of folds you will get a warning, but is OK.
holdout <- split(sample(1:N), 1:kfolds)
#head(holdout)
```

"10 trials Naive Bayes Plot Digit labels by Density (default dimensions retained = 15)"

```r
#Run training and Testing for each of the k-folds
AllResults<-list()
AllLabels<-list()
AllAccuracy <-list()
par(mfrow = c(2,5))
for (k in 1:kfolds){

  DigitDF_Test <- DigitDF[holdout[[k]], ]
  DigitDF_Train=DigitDF[-holdout[[k]], ]
## View the created Test and Train sets
#(head(DigitDF_Train))
#(table(DigitDF_Test$DigitTotalDF.label))
## Make sure you take the labels out of the testing data
#
  DigitDF_Test_noLabel<-DigitDF_Test[-c(1)]
  DigitDF_Test_justLabel<-DigitDF_Test$DigitTotalDF.label
#(head(DigitDF_Test_noLabel))
#### Naive Bayes prediction ussing e1071 package
#Naive Bayes Train model
  train_naibayes<-naiveBayes(DigitTotalDF.label~., data=DigitDF_Train,
na.action = na.pass)
#train_naibayes
#summary(train_naibayes)
#Naive Bayes model Prediction
  nb_Pred <- predict(train_naibayes, DigitDF_Test_noLabel)
#nb_Pred
#Testing accurancy of naive bayes model with Kaggle train data sub set
  (confusionMatrix(nb_Pred, DigitDF_Test$DigitTotalDF.label))
# Accumulate results from each fold, if you like
  AllResults<- c(AllResults,nb_Pred)
  AllLabels<- c(AllLabels, DigitDF_Test_justLabel)
# Evaluate accuracy
  confusion_m <- confusionMatrix(nb_Pred, DigitDF_Test$DigitTotalDF.label)
  AllAccuracy <- c(AllAccuracy,confusion_m$overall[1])
##Visualize
  plot(nb_Pred, fill = "#F76900")
}
```



```
mod.6.mean.acc.pca <- mean(unlist(AllAccuracy))
```

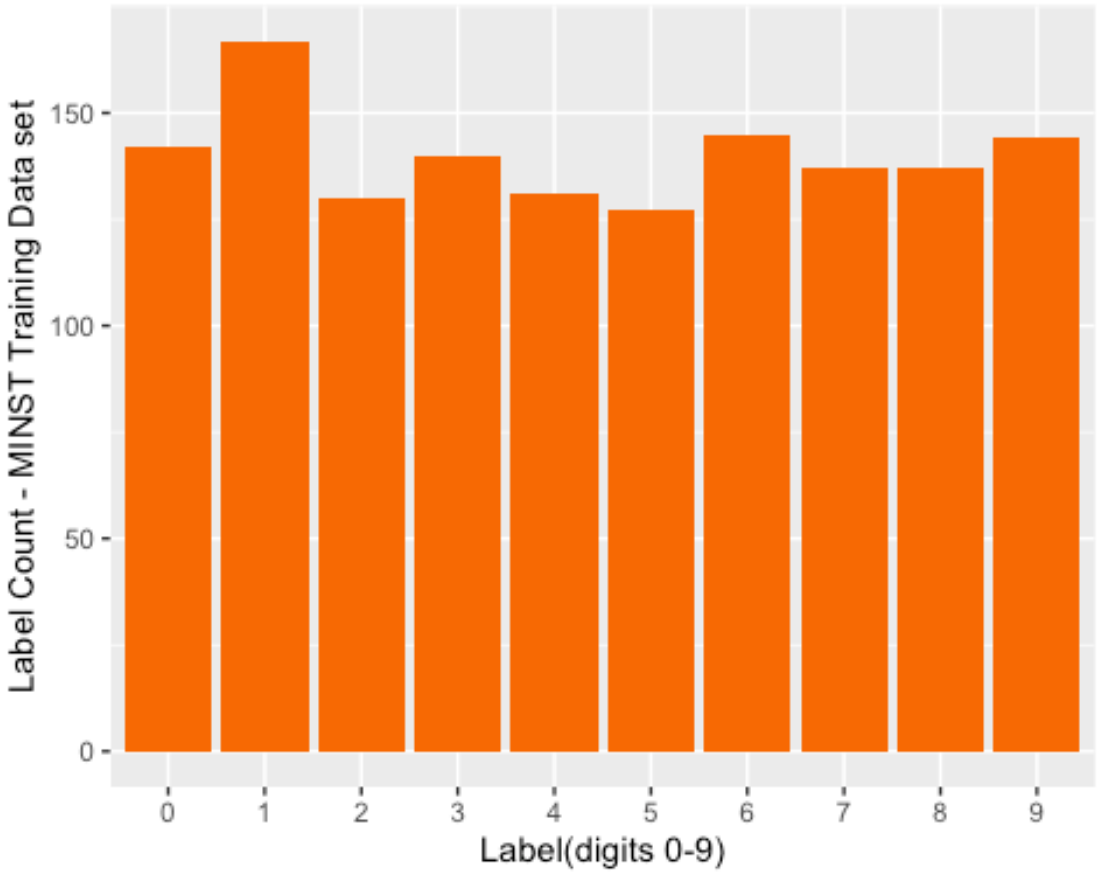
The overall accuracy of Model 5 is 81.32%.

Preprocess for SVM

Use a small data set for processing and educational purposes

```
setwd("~/")
small.mnist <- read.csv("Kaggle-digit-train-sample-small-1400.csv")
# dim(small.mnist) #[1] 1400  785

# number of rows with missing values
# nrow(small.mnist) - sum(complete.cases(small.mnist)) # returns 0

# Visualizing Label
table(small.mnist$label)

##
##   0   1   2   3   4   5   6   7   8   9
## 142 167 130 140 131 127 145 137 137 144

# ggplot2::theme_set(theme_gray()) # Set all ggplot to theme gray
minst.label.table <- as.data.frame(table(small.mnist$label))
names(minst.label.table)[1] <- "label"
# Create a barchart of the digit frequency of the original dataset
```

```
ggplot(minst.label.table, aes(label, Freq)) + geom_bar(stat = "identity",
fill = "#F76900") +
  xlab("Label(digits 0-9)") + ylab("Label Count - MINST Training Data set")
```




```
# Create training and test sets again
set.seed(4321)
Partition<- createDataPartition(y = small.mnist.fac$label, p = 0.80, list =
FALSE)
train<- small.mnist.fac[Partition,] # Create the train sample
# dim(train)
# table(train$label)
# round(prop.table(table(train$label))*100, digits = 2) # checking dimensions
of split

test<- small.mnist.fac[-Partition,] # Create the test sample
# dim(test)
# table(test$label)
# round(prop.table(table(test$label))*100, digits = 2) # checking dimensions
of split

# Remove labels for SVM processing
train_nolabel<- train[,-1]
train_justlabel<- train$label
```

```
test_nolabel<- test[,-1]
test_justlabel<- test$label
# Compairing proportions of both train and test
# round(prop.table(table(test$label))*100, digits = 2) -
round(prop.table(table(train$label))*100, digits = 2)
#     0     1     2     3     4     5     6     7     8     9
# -0.04 -0.02  0.13  0.14  0.04 -0.05  0.14 -0.05 -0.05 -0.22
```

Support Vector Machines (SVM) - Model 1: tuning with a polynomial kernel

The cost parameter in the SVM means the trade off between misclassification and simplicity of the mode, and the cost parameter decides how much an SVM should be allowed to "bend" with the data. For a low cost, one would aim for a smooth decision surface and for a higher cost, you aim to classify more points correctly. It is considered the cost of misclassification. The tune function from e1071 allows altering the SVM by adjusting cost with a sampling method of 10-fold cross validation. One optimizes both the fit of the line to data and penalizing the amount of samples inside the margin at the same time, where c defines the weight of how much samples inside the margin contribute to the overall error.[20]

```
set.seed(4321)
tune.model.1 <- tune(svm,label~., data=train,
                     kernel="polynomial",
                     ranges=list(cost=c(.01,.1,1,10,100,1000)))
tune1.best.performance <- round(tune.model.1$best.performance, 3)
```

The best cost is 0.103. This is the value for c, which will be used in this model.

```
set.seed(4321)
SVM.1 <- svm(label~., data=train, kernel="polynomial", cost=
tune1.best.performance, scale=FALSE) #585 SV
```

Use in model 6, a polynomial kernel SVM there are 616 support vectors out of 1123. The fewer support vectors means faster classification, as the support vectors are the points that are close to the boundary or within the boundary. The SVM used 54.85% of the training sample to encode the training set.[21]

When an SVM model is constructed, the number of support vectors as an indicator of model complexity.

Prediction with SVM Model 1 with a polynomial kernel

---

[20] https://stats.stackexchange.com/questions/225409/what-does-the-cost-c-parameter-mean-in-svm

[21] https://stackoverflow.com/questions/9480605/what-is-the-relation-between-the-number-of-support-vectors-and-training-data-and

```
set.seed(4321)
pred.1 <- predict(SVM.1, test_nolabel, type="class")
Ptable.1 <- table(pred.1, test_justlabel)
svm.table.1 <- confusionMatrix(pred.1, test_justlabel)
svm.table.1.acc <- round(svm.table.1$overall[1]*100,2)
svm.table.1$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 25  0  0  0  0  0  1  0  0  0
##          1  0 32  1  0  0  0  1  0  1  2
##          2  0  0 23  0  0  1  0  0  1  0
##          3  0  0  0 26  0  0  0  0  0  0
##          4  0  0  0  0 24  0  0  0  0  3
##          5  2  1  0  0  0 24  2  0  0  0
##          6  1  0  1  1  1  0 25  0  0  0
##          7  0  0  1  1  0  0  0 25  0  1
##          8  0  0  0  0  0  0  0  0 24  0
##          9  0  0  0  0  1  0  0  2  1 22
```

The accuracy of Model 1: SVM with a polynomial kernel and cost is 90.25%. The misclassification rate for of Model 1 is: 9.75%.

SVM - Model 2 - tuning with a linear kernel

Again, we can the tune function from e1071 to tune the SVM by altering the cost with a sampling method of 10-fold cross validation.

The best cost is 0.1246444. This is the value for c, which will be used in this model.

```
set.seed(4321)
SVM.2 <- svm(label~., data=train, kernel="linear", cost=
tune2.best.performance, scale=FALSE)
```

Use in SVM model 2, a linear kernel SVM there are 635 support vectors out of 1123. The fewer support vectors means faster classification, as the support vectors are the points that are close to the boundary or within the boundary. The SVM used 56.5449688% of the training sample to encode the training set.

Prediction for SVM - Model 2 - tuning with a linear kernel

```
set.seed(4321)
pred.2 <- predict(SVM.2, test_nolabel, type="class")
Ptable.2 <- table(pred.2, test_justlabel)
svm.table.2 <- confusionMatrix(pred.2, test_justlabel)
svm.table.2.acc <- round(svm.table.2$overall[1]*100,2)
svm.table.2$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 27  0  0  0  0  1  0  0  0  0
```

```
##          1  0 33  0  0  0  0  1  0  0  1
##          2  0  0 24  0  0  1  0  0  1  1
##          3  0  0  0 25  0  1  0  0  0  0
##          4  0  0  0  0 24  0  0  0  0  3
##          5  0  0  0  2  0 20  1  0  1  1
##          6  1  0  1  0  1  0 27  0  0  0
##          7  0  0  1  1  0  0  0 26  0  1
##          8  0  0  0  0  0  2  0  0 24  0
##          9  0  0  0  0  1  0  0  1  1 21
```

The accuracy of Model 2: SVM with a linear kernel and cost is 90.61%. The miscalssification rate for of Model 2 is: 9.39%.

SVM - Model 3: tuning with a radial kernel

Again, we can the tune function from e1071 to tune the SVM by altering the cost with a sampling method of 10-fold cross validation.

The best cost is 0.8806416. This is the value for c, which will be used in this model.

```
set.seed(4321)
SVM.3 <- svm(label~., data=train, kernel="radial", cost=
tune3.best.performance , scale=FALSE)
```

Use in SVM model 3, a radial kernel SVM there are 1123 support vectors out of 1123. The fewer support vectors means faster classification, as the support vectors are the points that are close to the boundary or within the boundary. The SVM used 100% of the training sample to encode the training set.

Prediction for SVM - Model 3 - tuning with a radial kernel

```
set.seed(4321)
pred.3 <- predict(SVM.3, test_nolabel, type="class")
Ptable.3 <- table(pred.3, test_justlabel)
svm.table.3 <- confusionMatrix(pred.3, test_justlabel)
svm.table.3.acc <- round(svm.table.3$overall[1]*100,2)
svm.table.3$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0  0  0  0  0  0  0  0  0  0  0
##          1 28 33 26 28 26 25 29 27 27 28
##          2  0  0  0  0  0  0  0  0  0  0
##          3  0  0  0  0  0  0  0  0  0  0
##          4  0  0  0  0  0  0  0  0  0  0
##          5  0  0  0  0  0  0  0  0  0  0
##          6  0  0  0  0  0  0  0  0  0  0
##          7  0  0  0  0  0  0  0  0  0  0
##          8  0  0  0  0  0  0  0  0  0  0
##          9  0  0  0  0  0  0  0  0  0  0
```

The accuracy of Model 3: SVM with a radial kernel and cost is 11.91%. The miscalssification rate for of Model 3 is: 88.09%.

SVM - Model 4: tuning with a sigmoid kernel

Again, we can the tune function from e1071 to tune the SVM by altering the cost with a sampling method of 10-fold cross validation.

The best cost is 0.8806416. This is the value for c, which will be used in this model.

```
set.seed(4321)
SVM.4 <- svm(label~., data=train, kernel="sigmoid", cost=
tune4.best.performance, scale=FALSE)
```

Use in SVM model 4, a sigmoid kernel SVM there are 1105 support vectors out of 1123. The fewer support vectors means faster classification, as the support vectors are the points that are close to the boundary or within the boundary. The SVM used 98.3971505% of the training sample to encode the training set.

Prediction for SVM - Model 4 - tuning with a sigmoid kernel

```
set.seed(4321)
pred.4 <- predict(SVM.4, test_nolabel, type="class")
Ptable.4 <- table(pred.4, test_justlabel)
svm.table.4 <- confusionMatrix(pred.4, test_justlabel)
svm.table.4.acc <- round(svm.table.4$overall[1]*100,2)
svm.table.4$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0  0  0  0  0  0  0  0  0  0  0
##          1 28 33 26 28 26 25 29 27 27 28
##          2  0  0  0  0  0  0  0  0  0  0
##          3  0  0  0  0  0  0  0  0  0  0
##          4  0  0  0  0  0  0  0  0  0  0
##          5  0  0  0  0  0  0  0  0  0  0
##          6  0  0  0  0  0  0  0  0  0  0
##          7  0  0  0  0  0  0  0  0  0  0
##          8  0  0  0  0  0  0  0  0  0  0
##          9  0  0  0  0  0  0  0  0  0  0
```

The accuracy of SVM Model 4: SVM with a sigmoid kernel and cost is 11.91%. The miscalssification rate for of Model 3 is: 88.09%.

---

k nearest neighbor (kNN) - Model 1 k=33.5

K Nearest neighbor is a simple algorithm that stores all the available cases and classifies the new data or case based on a similarity measure. It is mostly used to classify a data point

based on how its neighbors are classified. The major advantage of k-nearest neighbor is that it doesn't have any assumptions made, so unlike Naive Bayes, which assume the independence assumption. kNN can work well when the decision function to be learned is very complex. The ideal aspect of kNN, unlike SVM, is its decision boundary does not have any shape because it's determined by the neighbors.

The number one problem for a K-NN is that it is sensitive to noisy training data. We have to involve all of the attributes in the classification for K-NN unless we remove some attributes before sending them to the algorithm.

A large k value has benefits which include reducing the variance due to the noisy data. The value for k is generally chosen as the square root of the number of observations.[22] In this case the observations are 1123, with the square root of this value equaling 33.51.[23] If we use a k too small, we risk over-fitting and too large mis-classification is a risk.

```
set.seed(4321)
knn.1<- knn(train, test, train_justlabel, k= (round(sqrt(nrow(train)),2)))
knn.1.cm <- confusionMatrix(knn.1, test_justlabel)
knn.1.acc <- round(knn.1.cm$overall[1]*100,2)
knn.1.cm$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 23  0  0  0  0  0  2  0  0  0
##          1  0 33  4  1  1  4  2  2  1  1
##          2  0  0 17  0  0  0  0  0  0  0
##          3  0  0  0 25  0  2  0  0  1  0
##          4  0  0  0  0 19  0  0  0  0  1
##          5  1  0  0  1  0 15  0  0  1  0
##          6  3  0  1  0  0  1 25  0  0  0
##          7  0  0  2  1  0  1  0 24  0  0
##          8  0  0  1  0  0  1  0  0 22  0
##          9  1  0  1  0  6  1  0  1  2 26
```

The accuracy of Model 1 kNN is 82.67%.

k nearest neighbor (kNN) - Model 2: Search for optimal k value

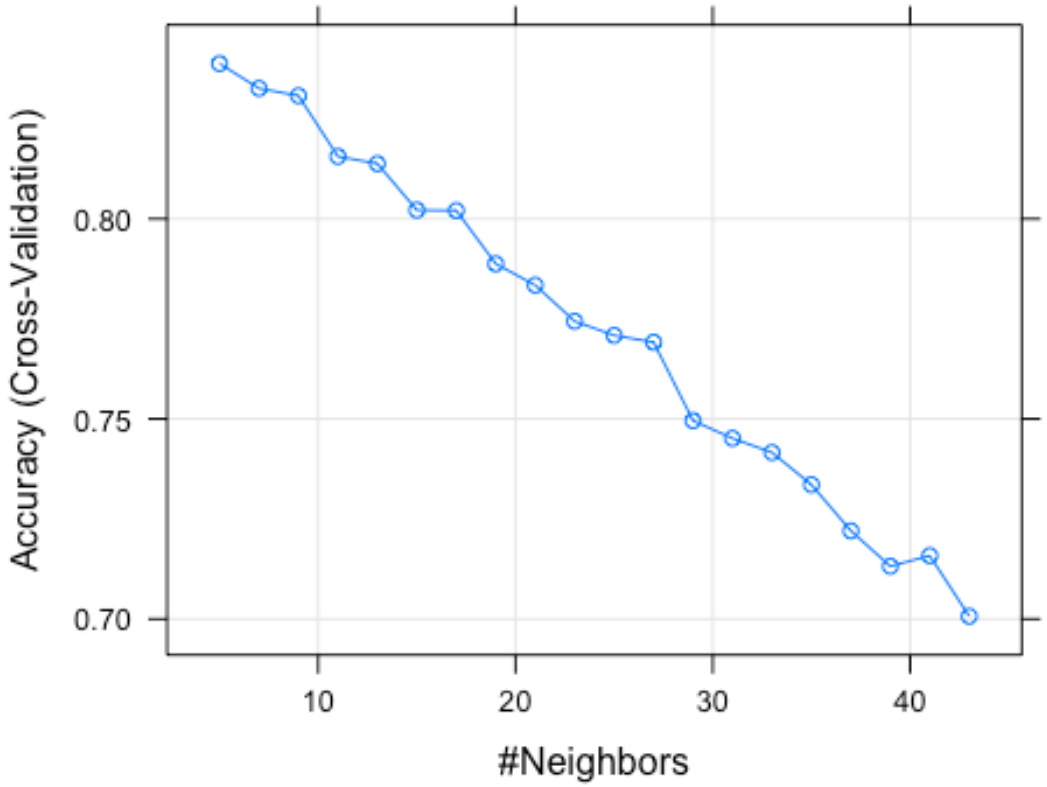
Using caret package for 10-fold cross validation to determine k[24]

---

[22] https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e

[23] https://towardsdatascience.com/a-simple-introduction-to-k-nearest-neighbors-algorithm-b3519ed98e

[24] http://www.sthda.com/english/articles/35-statistical-machine-learning-essentials/142-knn-k-nearest-neighbors-essentials/



The best parameter for k that maximizes model accuracy 5.

k nearest neighbor (kNN) - Model 3: Using optimal k value

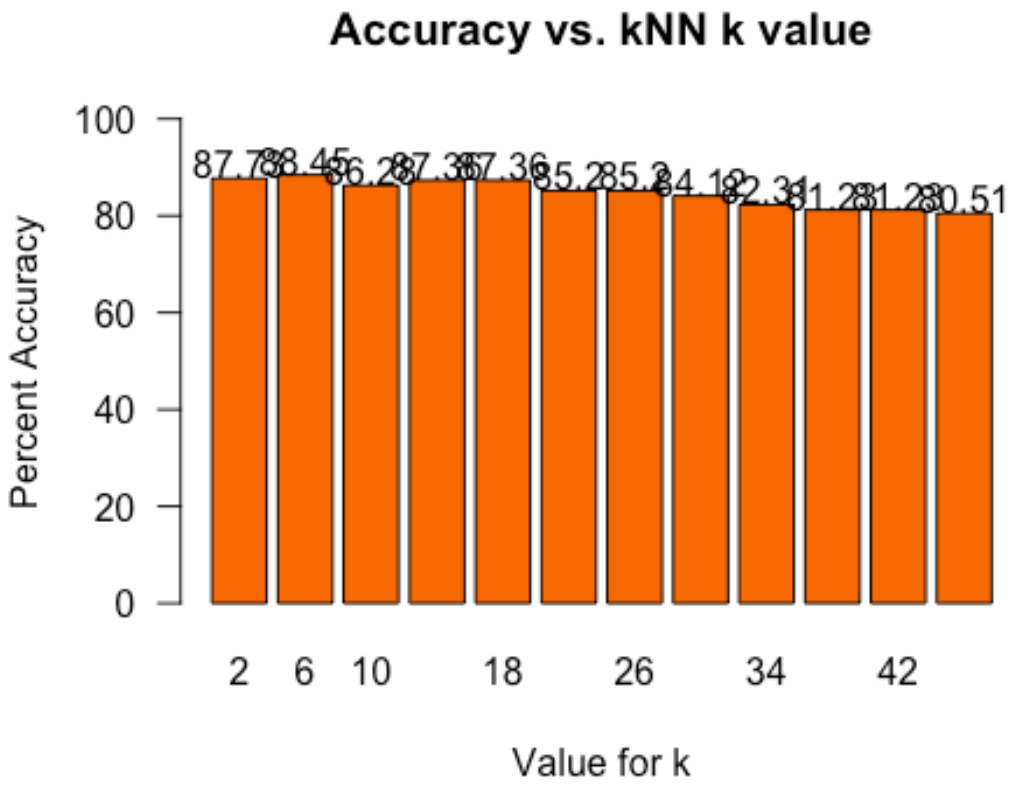
```
set.seed(4321)
knn.x<- knn(train, test, train_justlabel, k= 9)
knn.x.cm <- confusionMatrix(knn.x, test_justlabel)
knn.x.acc <- round(knn.x.cm$overall[1]*100,2)
knn.x.cm$table
```

```
##            Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 27  0  0  0  0  0  2  0  0  0
##          1  0 33  2  0  1  0  2  2  1  1
##          2  0  0 20  0  0  0  0  0  0  0
##          3  0  0  0 26  0  1  0  0  1  0
##          4  0  0  0  0 20  0  0  0  0  1
##          5  0  0  0  1  0 21  0  0  2  0
##          6  1  0  1  0  0  1 25  0  0  0
##          7  0  0  2  1  0  1  0 24  0  0
##          8  0  0  1  0  0  1  0  0 21  0
##          9  0  0  0  0  5  0  0  1  2 26
```

The accuracy of Model 1 kNN is 87.73%. A large increase in accuracy using the optimal k.

Alternate method for comparison without cross validation

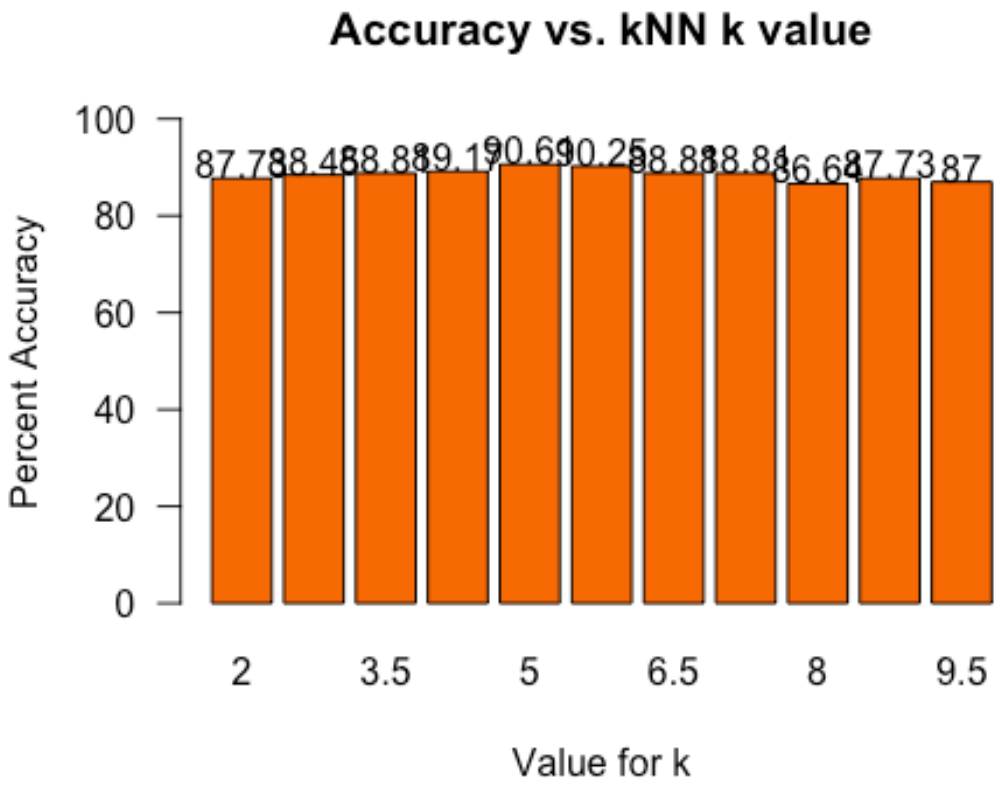
```
set.seed(4321)
AllAccuracy.knn <-list()
for (i in seq(2,46,4)) {
  knn.k<- knn(train, test, train_justlabel, k = i)
  knn.k.cm <- confusionMatrix(knn.k, test_justlabel)
  knn.k.acc <- round(knn.k.cm$overall[1]*100,2)
  AllAccuracy.knn <- c(AllAccuracy.knn, knn.k.cm$overall[1])
}
k.acc.df <- data.frame(kvalue = seq(2,46,4),
as.data.frame((unlist(AllAccuracy.knn))))
colnames(k.acc.df)[2] <- 'kNN.Pct.Accuracy'
k.acc.df$kNN.Pct.Accuracy <- round(k.acc.df$kNN.Pct.Accuracy*100, 2)
x<-barplot(k.acc.df$kNN.Pct.Accuracy, names.arg  = k.acc.df$kvalue, xlab =
'Value for k', ylab ='Percent Accuracy', main='Accuracy vs. kNN k value',
col='#F76900', las=1, ylim = c(0,100))
y <- k.acc.df$kNN.Pct.Accuracy
text(x,y+3,labels=as.character(y))
```



The value of k = 6 would provide the highest accuracy. Further evaluate the appropriate k:

```
set.seed(4321)
AllAccuracy.knn2 <-list()
```

```
for (i in seq(2,10,.75)) {
  knn.k<- knn(train, test, train_justlabel, k = i)
  knn.k.cm <- confusionMatrix(knn.k, test_justlabel)
  knn.k.acc <- round(knn.k.cm$overall[1]*100,2)
  AllAccuracy.knn2 <- c(AllAccuracy.knn2,knn.k.cm$overall[1])
}
k.acc.df <- data.frame(kvalue = seq(2,10,.75),
as.data.frame((unlist(AllAccuracy.knn2))))
colnames(k.acc.df)[2] <- 'kNN.Pct.Accuracy'
k.acc.df$kNN.Pct.Accuracy <- round(k.acc.df$kNN.Pct.Accuracy*100, 2)
x<-barplot(k.acc.df$kNN.Pct.Accuracy, names.arg  = k.acc.df$kvalue, xlab =
'Value for k', ylab ='Percent Accuracy', main='Accuracy vs. kNN k value',
col='#F76900', las=1, ylim = c(0,100))
y <- k.acc.df$kNN.Pct.Accuracy
text(x,y+3,labels=as.character(y))
```



The optimal value at k = 5. The highest bar 90.61%.

Random Forest - Model 1 mtry = 2

Random forest, an ensemble method, is based on the decision tree algorithm. The idea is that the original data is randomized, and then randomized subsets are created. These subsets a decision tree is created and then combined to a "forest", so that a prediction is made based on majority vote.

Create a Random Forest model with default parameters. By default, number of trees is 500 and number of variables tried at each split is 2 in this case.[25]

```
set.seed(4321)
rf.1 <- randomForest(label ~ ., data = train, importance = TRUE)
rf.p.1 <- predict(rf.1, test_nolabel)
rf.cm <- confusionMatrix(rf.p.1, test_justlabel)
rf.1.acc <- round(rf.cm$overall[1]*100,2)
rf.cm$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 28  0  0  0  0  0  1  0  0  0
##          1  0 32  0  0  0  0  2  0  0  1
##          2  0  0 25  0  0  0  0  0  0  0
##          3  0  0  0 27  0  1  0  0  0  1
##          4  0  0  0  0 23  0  0  0  0  2
##          5  0  1  0  1  0 23  1  0  0  0
##          6  0  0  1  0  1  1 25  0  0  0
##          7  0  0  0  0  0  0  0 25  0  0
##          8  0  0  0  0  0  0  0  1 26  0
##          9  0  0  0  0  2  0  0  1  1 24
```

The accuracy of random forest model 1 is 93.14%.

Random Forest - Model 2 - Tuning mtry = 6

The number of variables available for splitting at each tree node in the random forests, this is referred to as the mtry parameter.

```
rf.2<- randomForest(label ~ ., data = train, ntree = 500, mtry = 6,
importance = TRUE)
rf.p.2<- predict(rf.2, test_nolabel)
rf.cm.2<- confusionMatrix(rf.p.2, test_justlabel)
rf.2.acc <- round(rf.cm.2$overall[1]*100,2)
rf.cm.2$table
```

```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 27  0  0  0  0  0  1  0  0  0
##          1  0 33  0  0  0  0  2  0  1  1
##          2  0  0 25  0  0  0  0  0  1  0
```

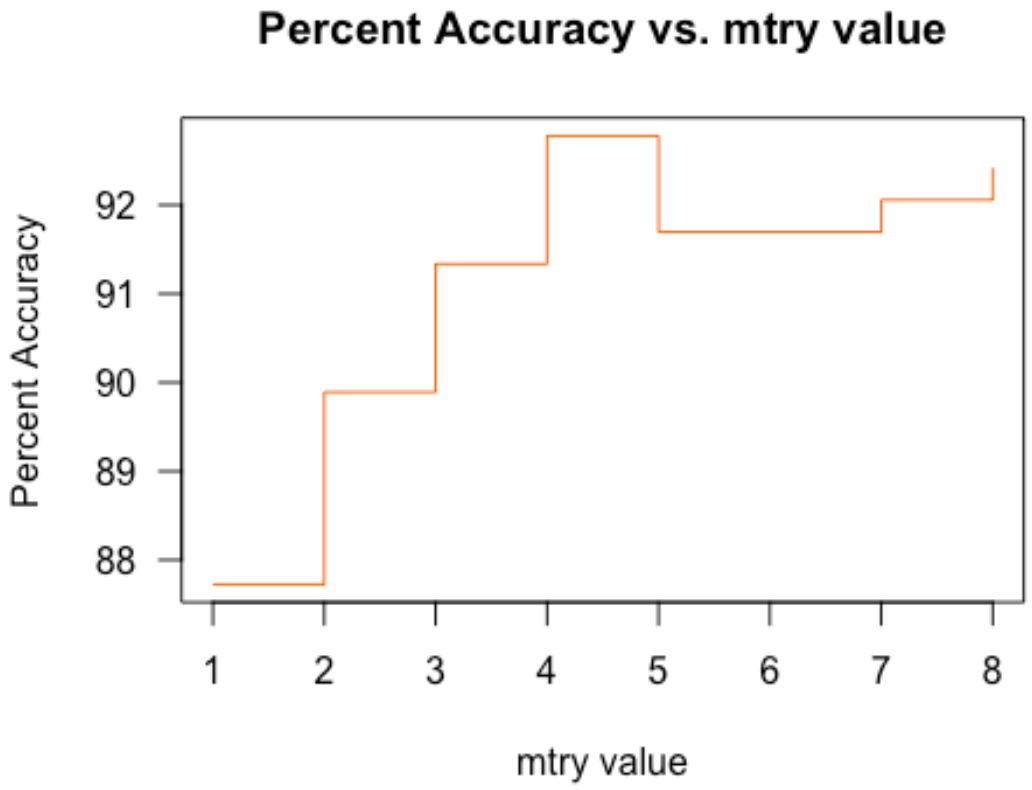
[25] https://www.r-bloggers.com/how-to-implement-random-forests-in-r/

```
##          3  0  0  0 27  0  1  0  0  0  0
##          4  0  0  0  0 23  0  0  0  0  1
##          5  0  0  0  1  0 23  1  0  0  0
##          6  1  0  1  0  1  1 25  0  0  0
##          7  0  0  0  0  0  0  0 25  0  0
##          8  0  0  0  0  0  0  0  1 24  0
##          9  0  0  0  0  2  0  0  1  1 26
```

The accuracy of random forest model 2 is 93.14%.

Identify the most appropriate mtry for the model

```
set.seed(4321)
a=c()
i=5
for (i in 3:10) {
  rf.xx <- randomForest(label ~ ., data = train, ntree = 500, mtry = i,
importance = TRUE)
  predValid <- predict(rf.xx, test_nolabel, type = "class")
  a[i-2] = mean(predValid == test_justlabel)
}

plot(a*100, type='s', las=1, ylab="Percent Accuracy", xlab = "mtry value",
main="Percent Accuracy vs. mtry value", col = "#F76900")
```



The accuracy of mtry increased from 1 to between 4 and 5 and then decreased after 5, and another a 8. Maximum accuracy is at mtry equal to 4 or 5.

Random Forest - Model 3 - Tuning mtry = 4

```
rf.3<- randomForest(label ~ ., data = train, ntree = 500, mtry = 4,
importance = TRUE)
rf.p.3<- predict(rf.3, test_nolabel)
rf.cm.3<- confusionMatrix(rf.p.3, test_justlabel)
rf.3.acc <- round(rf.cm.3$overall[1]*100,2)
rf.cm.3$table
```

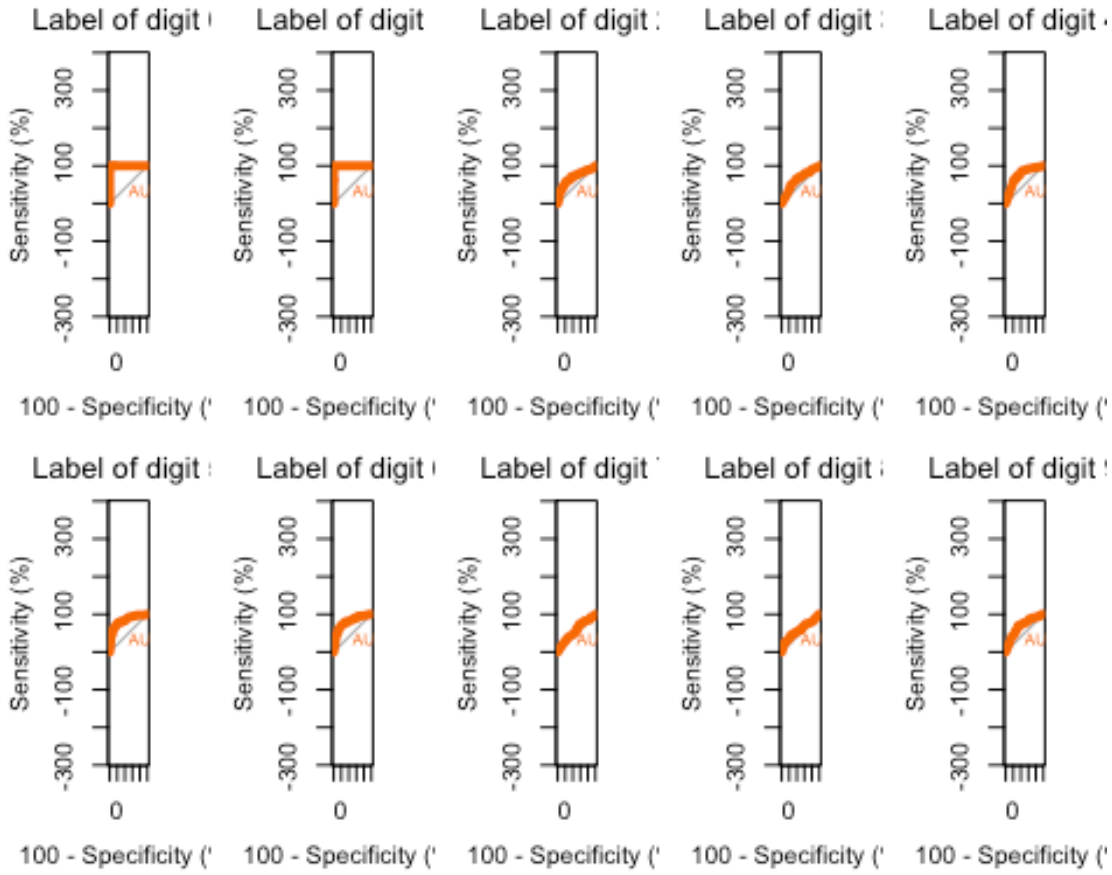
```
##           Reference
## Prediction  0  1  2  3  4  5  6  7  8  9
##          0 26  0  0  0  0  0  2  0  0  0
##          1  0 33  0  0  0  0  2  0  1  1
##          2  0  0 25  0  0  0  0  0  0  0
##          3  0  0  0 27  0  3  0  0  1  1
##          4  0  0  0  0 23  0  0  0  0  2
##          5  1  0  0  1  0 21  1  0  0  0
##          6  1  0  1  0  1  1 24  0  0  0
##          7  0  0  0  0  0  0  0 25  0  0
##          8  0  0  0  0  0  0  0  1 24  0
##          9  0  0  0  0  2  0  0  1  1 24
```

The accuracy of random forest model 2 is 90.97%.

Example of AUROC for the labels True Positive Percentage (y-axis) vs. False Positive Percentage (x-axis)

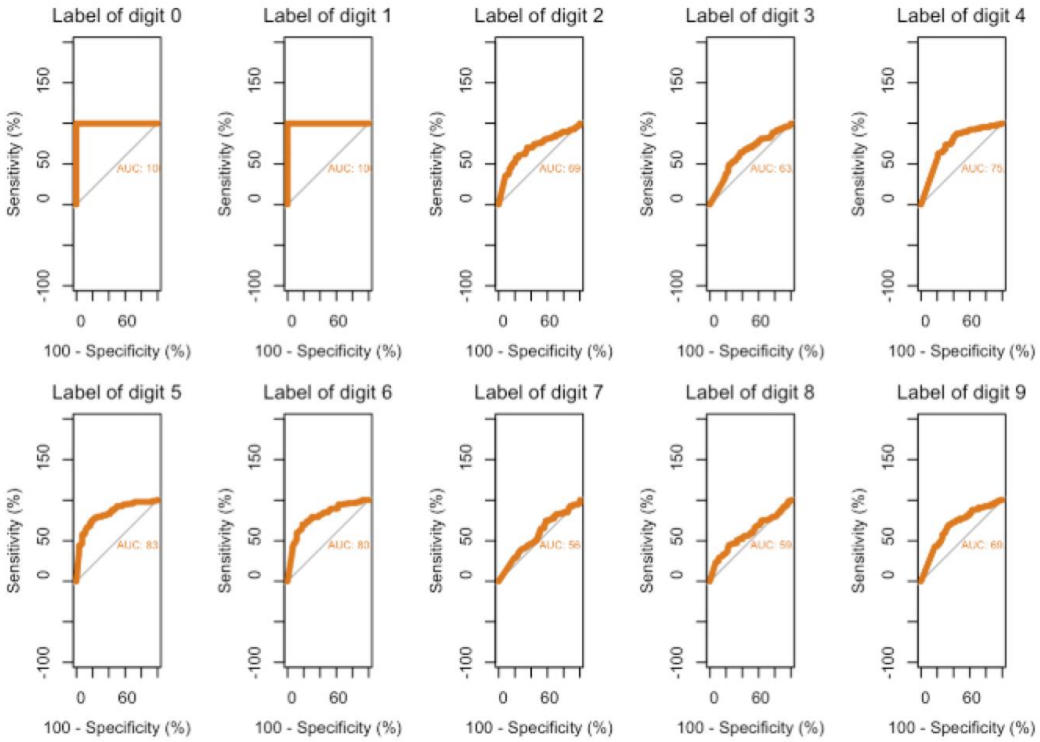
```
## Setting levels: control = 0, case = 1

## Setting direction: controls > cases

## Setting levels: control = 0, case = 1

## Setting direction: controls < cases

## Setting levels: control = 0, case = 1

## Setting direction: controls > cases

## Setting levels: control = 0, case = 1

## Setting direction: controls > cases

## Setting levels: control = 0, case = 1

## Setting direction: controls > cases

## Setting levels: control = 0, case = 1

## Setting direction: controls > cases
```

```
## Setting levels: control = 0, case = 1
## Setting direction: controls > cases
## Setting levels: control = 0, case = 1
## Setting direction: controls > cases
## Setting levels: control = 0, case = 1
## Setting direction: controls > cases
## Setting levels: control = 0, case = 1
## Setting direction: controls > cases
```



Instead of many confusion matrices receiver operator characteristic (ROC) graphs provide a simple way to summarize all of the information. The y-axis shows the sensitivity (true positive rate), the x-axis shows the false positive rate (1-specificity).The ROC enables one to evaluate the thresholds to make decisions about tolerance of false positives one is

willing to accept. Area under the curve, when greater, represents a more predictive model. The diagonal line shows where the true positive rate = the false positive rate.[26]



AUC

Results

This investigation involved the evaluation of classification algorithms; specifically, naive Bayes, Decision Trees, SVMs, kNNs, and Random Forest, to classify digits in the Modified National Institute of Standards and Technology (MNIST) data set. Here are the results:

Section 4: Algorithm performance comparison

Decision Trees

The accuracy of Decision tree - Model 1 1: accuracy was **63.42%**

Decision Tree - Model 2: with pruning was **63.42%**

---

[26] https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

Naïve Bayes

The accuracy of Naïve Bayes - Model 1 was **53.65%**

The accuracy of Naïve Bayes - Model 2: Principal Component Analysis at 5 dimensions was **64.41**

The accuracy of Naïve Bayes - Model 3: Principal Component Analysis at 15 dimensions was **81.32%**

SVM

The accuracy of SVM - Model 1: tuning with a polynomial kernel** was **90.25%**

The accuracy of SVM - Model 2: tuning with a linear kernel was **90.61%**

The accuracy of SVM - Model 3: tuning with a radial kernel was **11.91%**

The accuracy of SVM - Model 4: tuning with a sigmoid kernel was **11.91%**

kNN

The accuracy of kNN Model 1: k=33.5 was **82.67%**

The accuracy of kNN Model 2: Search for optimal k value was **89.17%**

The accuracy of kNN Model 3: Using optimal k value was **87.73%.**

Random Forest

The accuracy of Random Forest - Model 1: mtry = 2 was **93.14%**

The accuracy of Random Forest - Model 2: Tuning mtry = 6 was **93.14%**

The accuracy of Random Forest - Model 2: Tuning mtry = 4 was **90.97%**

Naïve Bayes classifiers depend kernel density estimation, and it's clear when the appropriate kernel is used and cost, higher accuracy levels are achieved. Naive Bayes classifiers are extremely fast compared to more sophisticated methods, but the estimation is not as strong as some others.

K Nearest neighbor is an simple non-parametric type of instance-based learning algorithm that stores all the available cases and classifies the new data or case based on a similarity measure. It is mostly used to classify a data point based on how its neighbors are classified. The major advantage of k-nearest neighbor is that it doesn't have any assumptions made, so unlike Naive Bayes, which assume the independence assumption. kNN can work well when the decision function to be learned is very complex. The ideal aspect of kNN, unlike SVM, is its decision boundary does not have any shape because it's determined by the neighbors. The number one problem for a K-NN is that it is sensitive to noisy training data.

Because no prior knowledge was inferred from the training data and stored in any model. So all of the calculations are in the testing step in the prediction step. And then every test example has to be compared with all of the training examples. One does not distill the values down, so the information is retrained, but the processing takes much time. While kNN is a highly flexible algorithm that could be suitable for many tasks, it takes a long time to run. The random forest uses many decision trees. Going beyond just averaging the prediction of trees, the model randomly samples the training data points when building trees and uses a subset of features when splitting nodes. The accuracy of Random Forest was highest with mtry = 6 ntree = 500 at 93.14%.

Conclusions

Despite the progress and low error rates from neural nets and convolutional neural nets, this analysis demonstrated that other classifiers achieved relatively respectable results with MNIST data set. The results with Random Forest were most impressive.

While there were differences in the size of the training and testing data sets, given the training time for SVM, kNN, nb random forest, the accuracy was still higher. The nature of the Random forest algorithm, an ensemble method, includes prediction made based on a majority vote. Overall, there are pros and cons to the various algorithms and considerations based on the data. It's important to know what is sacrificed when using a given algorithm even if the cost is time, more sophisticated knowledge of parameter tuning, among other aspects to consider. As Professor Yu notes, "there is no silver bullet" when we select a classification algorithm.