MNIST Dataset

Jared Mosley

8/11/2020

Table of Contents

[Overview 2](#_Toc49942296)

[EDA 3](#_Toc49942297)

[Analysis 4](#_Toc49942298)

[Decision Tree 4](#_Toc49942299)

[Decision Tree: Our Baseline 4](#_Toc49942300)

[Decision Tree: With PCA 9](#_Toc49942301)

[Decision Tree: With Pruning 15](#_Toc49942302)

[Decision Tree: With Cross Validation 17](#_Toc49942303)

[Naive Bayes 19](#_Toc49942304)

[Naive Bayes: Baseline 19](#_Toc49942305)

[Naive Bayes: With Less Dimensionality Using PCA of 5 21](#_Toc49942306)

[Naive Bayes: With Less Dimensionality Using PCA of 15 22](#_Toc49942307)

[Naive Bayes: Using k-folds Cross Validation and PCA 25](#_Toc49942308)

[kNN 27](#_Toc49942309)

[kNN: Baseline 27](#_Toc49942310)

[kNN: Optimal k Value Evaluation 28](#_Toc49942311)

[kNN: With Cross Fold Validation 29](#_Toc49942312)

[SVM 31](#_Toc49942313)

[SVM: Model with a Polynomial Kernel 31](#_Toc49942314)

[SVM: Model with a Linear Kernel 32](#_Toc49942315)

[SVM: Model with Radial Kernel 34](#_Toc49942316)

[SVM: Model with Sigmoid Kernel 35](#_Toc49942317)

[SVM: Model with a Linear Kernel and PCA 36](#_Toc49942318)

[Random Forest 38](#_Toc49942319)

[Random Forest: Baseline 38](#_Toc49942320)

[Random Forest: Identifying “Best” mtry 39](#_Toc49942321)

[Random Forest: Using optimal mtry 40](#_Toc49942322)

[Random Forest: Using Optimal mtry and PCA 41](#_Toc49942323)

[Model Performance Comparison 43](#_Toc49942324)

[Conclusion 44](#_Toc49942325)

# Overview

The MNIST (“Modified National Institue of Standards and Technology”) dataset is the prelimanary machine learning dataset for anyone interested in understanding computer vision.

It contains thousands of training and testing images of handwritten numbers from 0-9 written by the American Census Bureau employees for the training data, and by American high school students for the testing data. Each image was normalized and fit into a 28x28 pixel box.

The objective is simple: build various models to see if you can accurately classify the numbers written in the testing data. For this experiment, we will be using a slimmed down MNIST dataset.

# Import the csv  
train <- read.csv('Kaggle-digit-train-sample-small-1400.csv')  
test <- read.csv('Kaggle-digit-test-sample1000.csv')

Checking the dimensionality of the training data, we find 1400 observations of 785 variables. For the testing data, we find 1000 observations of 785 variables. The first column in each is a label of a digit from 0-9, and the remaining variables are pixels represented as integers. These 784 variables will make up a 28 x 28 image grid. Each variable is a value from 0-255 representing the level of shading, with the lower numbers indicating lightness and higher numbers representing darkness.

As the csv’s we imported are already provided as subsets, there is little additional cleaning to be done. At most, we will partition the training data and remove the labels later on.

# check dimensionality  
dim(train)

## [1] 1400 785

dim(test)

## [1] 1000 785

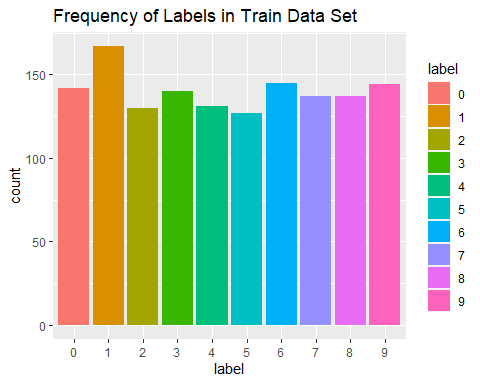
# Convert label into a factor  
train$label <- as.factor(train$label)  
  
ptm <- proc.time()

# EDA

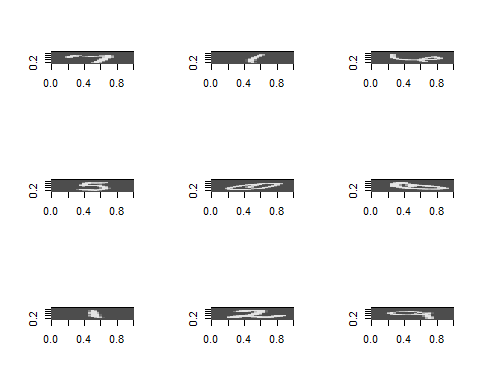
Visualizing the set, we see a relatively even dispersion frequency of the labels in the trainind dataset, with “1” a little higher than the rest, though this does fit Benford’s law.

We also look to vizualize the pixel mapping of each number. You can see how the shading values represented in the 28x28 grid can be restructured back into their image format.

# Visualize dispersion of digits in the train dataset  
ggplot(train, aes(x=label, fill=label)) + geom\_bar() + ggtitle("Frequency of Labels in Train Data Set")



# Visualize the underlying pixels  
par(mfrow=c(3,3))  
loop.vector <- 1:9  
for(i in 1:9){  
 im <- matrix((train[i,2:ncol(train)]), nrow = 28, ncol = 28)  
 im\_numbers <- apply(im, 1, as.numeric)  
 im\_numbers <- rotate.matrix(im\_numbers, 90)  
 image(im\_numbers, col=grey.colors(255), xlim = c(0,1), ylim = c(0.1,1.1))  
}



# Analysis

## Decision Tree

The decision tree is a supervised learning method for classification. It learns from the data by breaking it down into different subsets, branching out as it goes. The deeper the tree, the more decision rules and lower the confidence for the rules gets. The trees can then predict outcomes based off of the decision steps on each branch.

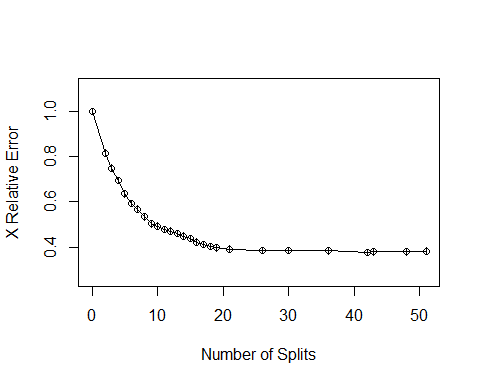
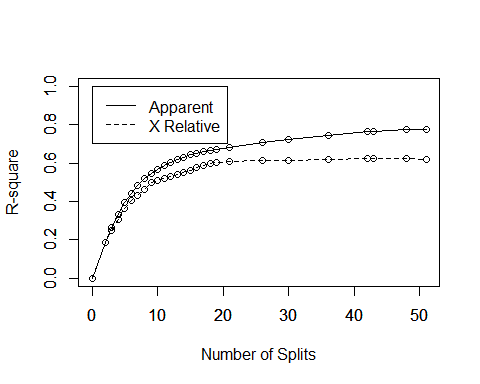
### Decision Tree: Our Baseline

Using the rpart function, we set now limits to our cp, but allow for 100 minimum splits and a max depth of 10. This inital baseline setup returns an accuracy of 80.29% as well as an elapsed computing time of 5.96. It appears around that at 25 splits the relative error begins to deviate more.

set.seed(22)  
# Train the decision tree without PCA  
dtrain1 <- rpart(label ~ ., train, method='class',  
 control = rpart.control(cp=0),  
 minsplit=100, maxdepth=10)  
  
# Run prediction  
pdtrain1 <- data.frame(predict(dtrain1,train))  
  
# Reformat for better evaluation  
pdtrain1 <- as.data.frame(names(pdtrain1[apply(pdtrain1, 1, which.max)]))  
# Reaname column  
colnames(pdtrain1) <- 'prediction'  
pdtrain1$number <- substr(pdtrain1$prediction, 2, 2)  
pdtrain1 <- train %>% bind\_cols(pdtrain1) %>% select(label, number) %>%   
 mutate(label = as.factor(label), number = as.factor(round(as.numeric(number), 0)))  
  
# Visualize decision trees  
rsq.rpart(dtrain1)

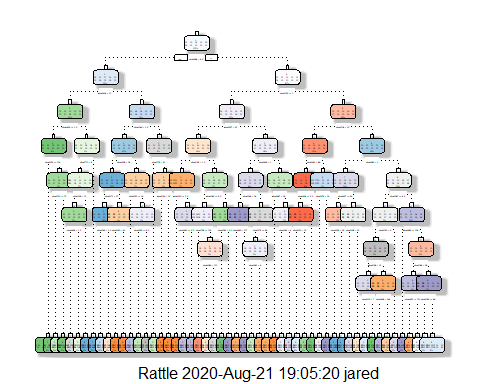
##   
## Classification tree:  
## rpart(formula = label ~ ., data = train, method = "class", control = rpart.control(cp = 0),   
## minsplit = 100, maxdepth = 10)  
##   
## Variables actually used in tree construction:  
## [1] pixel103 pixel155 pixel183 pixel186 pixel206 pixel208 pixel216 pixel239  
## [9] pixel246 pixel269 pixel270 pixel272 pixel290 pixel293 pixel298 pixel319  
## [17] pixel346 pixel350 pixel354 pixel379 pixel405 pixel407 pixel409 pixel429  
## [25] pixel434 pixel455 pixel457 pixel459 pixel461 pixel483 pixel485 pixel488  
## [33] pixel489 pixel496 pixel516 pixel541 pixel542 pixel543 pixel550 pixel553  
## [41] pixel574 pixel597 pixel598 pixel626 pixel657 pixel658 pixel713  
##   
## Root node error: 1233/1400 = 0.88071  
##   
## n= 1400   
##   
## CP nsplit rel error xerror xstd  
## 1 0.09448500 0 1.00000 1.00000 0.0098359  
## 2 0.07542579 2 0.81103 0.81509 0.0136570  
## 3 0.06650446 3 0.73560 0.74858 0.0143825  
## 4 0.06326034 4 0.66910 0.69343 0.0147963  
## 5 0.04703974 5 0.60584 0.63504 0.0150659  
## 6 0.04055150 6 0.55880 0.59205 0.0151590  
## 7 0.03892944 7 0.51825 0.56691 0.0151730  
## 8 0.02676399 8 0.47932 0.53447 0.0151469  
## 9 0.02189781 9 0.45255 0.50284 0.0150736  
## 10 0.01784266 10 0.43066 0.49230 0.0150385  
## 11 0.01622060 11 0.41281 0.47770 0.0149810  
## 12 0.01540957 12 0.39659 0.47040 0.0149484  
## 13 0.01297648 13 0.38118 0.45985 0.0148966  
## 14 0.01135442 14 0.36821 0.44769 0.0148300  
## 15 0.00892133 15 0.35685 0.43715 0.0147662  
## 16 0.00811030 16 0.34793 0.42174 0.0146628  
## 17 0.00729927 17 0.33982 0.41038 0.0145786  
## 18 0.00567721 18 0.33252 0.40146 0.0145077  
## 19 0.00527170 19 0.32685 0.39578 0.0144604  
## 20 0.00486618 21 0.31630 0.39011 0.0144113  
## 21 0.00405515 26 0.29197 0.38443 0.0143604  
## 22 0.00324412 30 0.27575 0.38524 0.0143678  
## 23 0.00283861 36 0.25629 0.38281 0.0143456  
## 24 0.00243309 42 0.23682 0.37551 0.0142769  
## 25 0.00162206 43 0.23439 0.37794 0.0143001  
## 26 0.00081103 48 0.22628 0.37794 0.0143001  
## 27 0.00000000 51 0.22384 0.38200 0.0143381

## Warning in rsq.rpart(dtrain1): may not be applicable for this method



fancyRpartPlot(dtrain1)

## Warning: labs do not fit even at cex 0.15, there may be some overplotting



# Return most valuable pixels  
important <- (varImp(dtrain1, scale=F))  
important$Value <- rownames(important)  
top <- important %>% arrange(desc(Overall))   
head(top)

## Overall Value  
## 1 128.17111 pixel542  
## 2 92.19692 pixel543  
## 3 89.43458 pixel434  
## 4 89.27440 pixel461  
## 5 88.83259 pixel350  
## 6 85.79128 pixel433

# Evaluate   
confusionMatrix(pdtrain1$number, pdtrain1$label)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 127 0 6 0 0 1 0 0 7 1  
## 1 0 146 3 0 3 4 1 0 6 1  
## 2 2 2 99 6 2 5 9 1 4 3  
## 3 0 3 3 103 3 10 0 1 8 3  
## 4 5 1 0 1 102 1 6 6 2 5  
## 5 1 2 5 10 2 94 7 1 3 5  
## 6 2 4 6 8 2 7 117 4 2 2  
## 7 1 5 5 6 5 2 4 121 1 11  
## 8 2 4 2 3 4 1 1 0 104 2  
## 9 2 0 1 3 8 2 0 3 0 111  
##   
## Overall Statistics  
##   
## Accuracy : 0.8029   
## 95% CI : (0.781, 0.8234)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.7808   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.89437 0.8743 0.76154 0.73571 0.77863 0.74016  
## Specificity 0.98808 0.9854 0.97323 0.97540 0.97872 0.97172  
## Pos Pred Value 0.89437 0.8902 0.74436 0.76866 0.79070 0.72308  
## Neg Pred Value 0.98808 0.9830 0.97553 0.97077 0.97718 0.97402  
## Prevalence 0.10143 0.1193 0.09286 0.10000 0.09357 0.09071  
## Detection Rate 0.09071 0.1043 0.07071 0.07357 0.07286 0.06714  
## Detection Prevalence 0.10143 0.1171 0.09500 0.09571 0.09214 0.09286  
## Balanced Accuracy 0.94122 0.9298 0.86738 0.85556 0.87867 0.85594  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.80690 0.88321 0.75912 0.77083  
## Specificity 0.97052 0.96833 0.98496 0.98487  
## Pos Pred Value 0.75974 0.75155 0.84553 0.85385  
## Neg Pred Value 0.97753 0.98709 0.97416 0.97402  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.08357 0.08643 0.07429 0.07929  
## Detection Prevalence 0.11000 0.11500 0.08786 0.09286  
## Balanced Accuracy 0.88871 0.92577 0.87204 0.87785

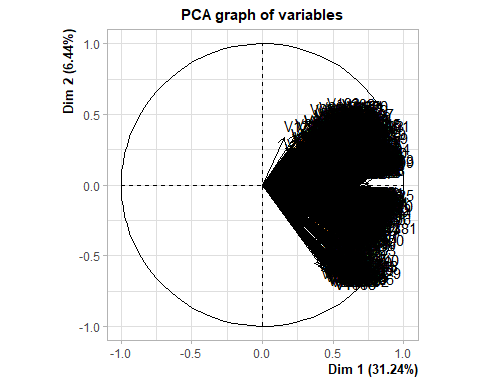
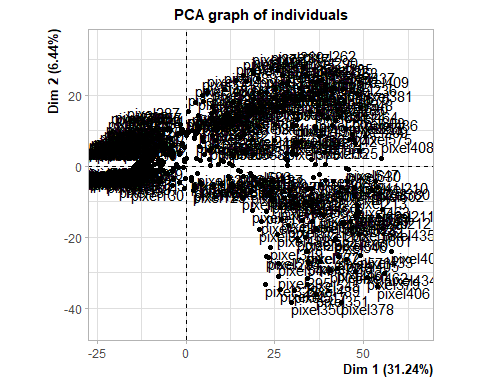
proc.time() - ptm

## user system elapsed   
## 2.92 0.39 3.34

### Decision Tree: With PCA

Using PCA to Reduce Dimensionality, we are able to summarize the patterns in the large dataset, and reduce to smaller (more identifiable) values that will drastically cut the dimensions from the dataset, which can greatly enhance the computing times for each problem. Here we reduce to 6 dimensions.

# Set seed for reproducability  
set.seed(22)  
  
# Set up pca train parameters  
pcadigits <- PCA(t(select(train, -label)))



# Regroup into data frame  
pcatrain <- data.frame(train$label, pcadigits$var$coord)  
  
# Rename first column  
names(pcatrain)[1] <- "label"  
  
# Check new dimensionalityy  
dim(pcatrain)

## [1] 1400 6

proc.time()- ptm

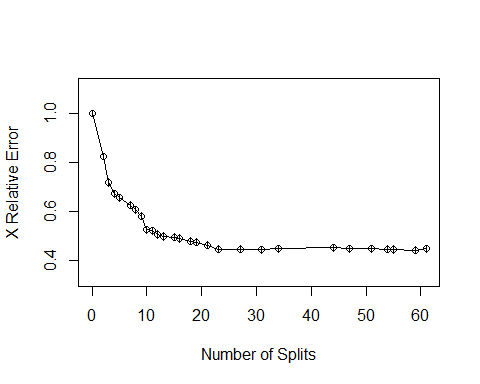
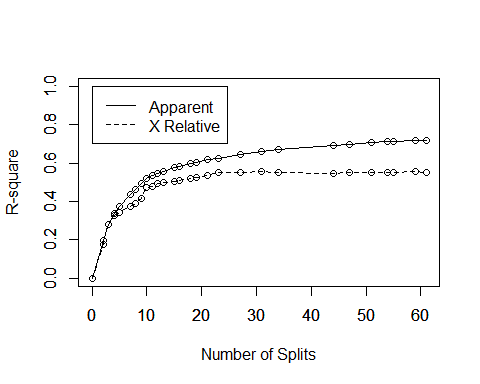
## user system elapsed   
## 5.09 0.84 5.99

Running the same decision tree as above, in this example we are instead using data that has less dimensionality. Using the 6 dimensions, we actually find an accuracy of 75.21% and 2.7 computing time. This is much more efficient, though not as accurate as previously.

# Train the decision tree with PCA  
dtrain2 <- rpart(label ~ ., pcatrain, method='class',  
 control = rpart.control(cp=0),  
 minsplit=100, maxdepth=10)  
  
# Run prediction  
pdtrain2 <- data.frame(predict(dtrain2,pcatrain))  
  
# Reformat for better evaluation  
pdtrain2 <- as.data.frame(names(pdtrain2[apply(pdtrain2, 1, which.max)]))  
# Reaname column  
colnames(pdtrain2) <- 'prediction'  
pdtrain2$number <- substr(pdtrain2$prediction, 2, 2)  
pdtrain2 <- train %>% bind\_cols(pdtrain2) %>% select(label, number) %>%   
 mutate(label = as.factor(label), number = as.factor(round(as.numeric(number), 0)))  
  
# Visualize decision trees  
rsq.rpart(dtrain2)

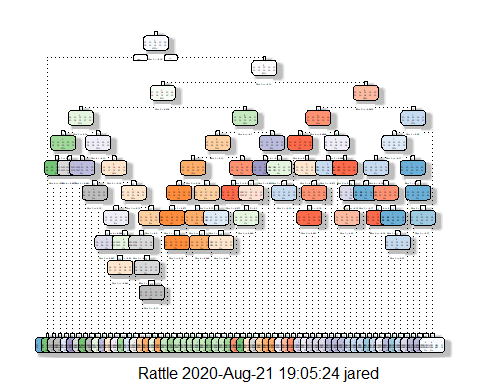
##   
## Classification tree:  
## rpart(formula = label ~ ., data = pcatrain, method = "class",   
## control = rpart.control(cp = 0), minsplit = 100, maxdepth = 10)  
##   
## Variables actually used in tree construction:  
## [1] Dim.1 Dim.2 Dim.3 Dim.4 Dim.5  
##   
## Root node error: 1233/1400 = 0.88071  
##   
## n= 1400   
##   
## CP nsplit rel error xerror xstd  
## 1 0.09813463 0 1.00000 1.00000 0.0098359  
## 2 0.08596918 2 0.80373 0.82482 0.0135280  
## 3 0.05433901 3 0.71776 0.71857 0.0146275  
## 4 0.03730738 4 0.66342 0.67315 0.0149090  
## 5 0.03081914 5 0.62612 0.65693 0.0149845  
## 6 0.02919708 7 0.56448 0.62693 0.0150902  
## 7 0.02757502 8 0.53528 0.60746 0.0151358  
## 8 0.02595296 9 0.50770 0.58232 0.0151680  
## 9 0.01540957 10 0.48175 0.52636 0.0151327  
## 10 0.01378751 11 0.46634 0.52230 0.0151244  
## 11 0.01135442 12 0.45255 0.50770 0.0150880  
## 12 0.00851582 13 0.44120 0.49959 0.0150633  
## 13 0.00811030 15 0.42417 0.49716 0.0150553  
## 14 0.00770479 16 0.41606 0.49067 0.0150326  
## 15 0.00648824 18 0.40065 0.48094 0.0149947  
## 16 0.00567721 19 0.39416 0.47607 0.0149740  
## 17 0.00486618 21 0.38281 0.46391 0.0149172  
## 18 0.00405515 23 0.37307 0.44769 0.0148300  
## 19 0.00364964 27 0.35685 0.44769 0.0148300  
## 20 0.00324412 31 0.34063 0.44526 0.0148158  
## 21 0.00243309 34 0.33090 0.44850 0.0148346  
## 22 0.00216275 44 0.30657 0.45255 0.0148575  
## 23 0.00202758 47 0.30008 0.44850 0.0148346  
## 24 0.00189240 51 0.29197 0.44850 0.0148346  
## 25 0.00162206 54 0.28629 0.44769 0.0148300  
## 26 0.00060827 55 0.28467 0.44688 0.0148253  
## 27 0.00040552 59 0.28224 0.44363 0.0148061  
## 28 0.00000000 61 0.28143 0.44850 0.0148346

## Warning in rsq.rpart(dtrain2): may not be applicable for this method



fancyRpartPlot(dtrain2)

## Warning: labs do not fit even at cex 0.15, there may be some overplotting



confusionMatrix(pdtrain2$number, pdtrain2$label)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 115 1 2 3 0 8 2 2 3 1  
## 1 0 158 4 1 1 7 4 4 7 2  
## 2 5 0 83 1 4 0 10 5 6 1  
## 3 1 1 2 103 2 7 4 0 14 0  
## 4 0 0 3 0 90 0 3 7 1 13  
## 5 10 1 0 14 0 94 0 4 6 2  
## 6 6 1 27 2 5 3 116 2 3 2  
## 7 1 3 2 2 7 2 1 99 4 15  
## 8 4 2 6 14 1 4 4 1 88 1  
## 9 0 0 1 0 21 2 1 13 5 107  
##   
## Overall Statistics  
##   
## Accuracy : 0.7521   
## 95% CI : (0.7287, 0.7746)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.7243   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.80986 0.9461 0.63846 0.73571 0.68702 0.74016  
## Specificity 0.98251 0.9757 0.97480 0.97540 0.97872 0.97093  
## Pos Pred Value 0.83942 0.8404 0.72174 0.76866 0.76923 0.71756  
## Neg Pred Value 0.97862 0.9926 0.96342 0.97077 0.96804 0.97400  
## Prevalence 0.10143 0.1193 0.09286 0.10000 0.09357 0.09071  
## Detection Rate 0.08214 0.1129 0.05929 0.07357 0.06429 0.06714  
## Detection Prevalence 0.09786 0.1343 0.08214 0.09571 0.08357 0.09357  
## Balanced Accuracy 0.89619 0.9609 0.80663 0.85556 0.83287 0.85555  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.80000 0.72263 0.64234 0.74306  
## Specificity 0.95936 0.97070 0.97070 0.96576  
## Pos Pred Value 0.69461 0.72794 0.70400 0.71333  
## Neg Pred Value 0.97648 0.96994 0.96157 0.97040  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.08286 0.07071 0.06286 0.07643  
## Detection Prevalence 0.11929 0.09714 0.08929 0.10714  
## Balanced Accuracy 0.87968 0.84667 0.80652 0.85441

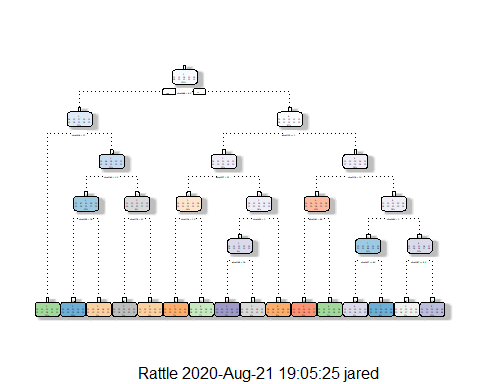
proc.time()- ptm

## user system elapsed   
## 5.64 1.28 7.03

### Decision Tree: With Pruning

Pruning our initial tree, we are increasing the threshold of our cp to .01 in hopes to clear out any overfitting, as well as focusing on a minsplit of 25. This returns a lower accuracy of 68.57% and a 1.7 elapsed computing time. We are however pruning, and not running a “new” model.

set.seed(22)  
# Train the decision tree with pruning  
dtrain3 <- prune(dtrain1, cp=.01, minsplit=25)  
  
# Run prediction  
pdtrain3 <- data.frame(predict(dtrain3,train))  
  
# Reformat for better evaluation  
pdtrain3 <- as.data.frame(names(pdtrain3[apply(pdtrain3, 1, which.max)]))  
# Reaname column  
colnames(pdtrain3) <- 'prediction'  
pdtrain3$number <- substr(pdtrain3$prediction, 2, 2)  
pdtrain3 <- train %>% bind\_cols(pdtrain3) %>% select(label, number) %>%   
 mutate(label = as.factor(label), number = as.factor(round(as.numeric(number), 0)))  
  
# Visualize decision trees  
# rsq.rpart(dtrain3)  
fancyRpartPlot(dtrain3)



confusionMatrix(pdtrain3$number, pdtrain3$label)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 118 1 5 5 0 8 11 10 0 2  
## 1 0 142 6 2 0 2 2 5 15 2  
## 2 0 3 89 20 6 3 11 0 1 3  
## 3 2 0 2 59 0 3 0 0 6 0  
## 4 1 0 1 1 87 2 4 9 0 8  
## 5 8 5 4 24 4 86 17 5 18 4  
## 6 4 8 10 9 11 11 87 1 10 1  
## 7 0 1 4 2 0 3 1 99 0 7  
## 8 5 6 5 10 3 5 8 1 80 4  
## 9 4 1 4 8 20 4 4 7 7 113  
##   
## Overall Statistics  
##   
## Accuracy : 0.6857   
## 95% CI : (0.6607, 0.71)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.6506   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.83099 0.8503 0.68462 0.42143 0.66412 0.67717  
## Specificity 0.96661 0.9724 0.96299 0.98968 0.97951 0.93009  
## Pos Pred Value 0.73750 0.8068 0.65441 0.81944 0.76991 0.49143  
## Neg Pred Value 0.98065 0.9796 0.96756 0.93901 0.96581 0.96653  
## Prevalence 0.10143 0.1193 0.09286 0.10000 0.09357 0.09071  
## Detection Rate 0.08429 0.1014 0.06357 0.04214 0.06214 0.06143  
## Detection Prevalence 0.11429 0.1257 0.09714 0.05143 0.08071 0.12500  
## Balanced Accuracy 0.89880 0.9114 0.82380 0.70556 0.82182 0.80363  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.60000 0.72263 0.58394 0.78472  
## Specificity 0.94821 0.98575 0.96279 0.95303  
## Pos Pred Value 0.57237 0.84615 0.62992 0.65698  
## Neg Pred Value 0.95353 0.97038 0.95522 0.97476  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.06214 0.07071 0.05714 0.08071  
## Detection Prevalence 0.10857 0.08357 0.09071 0.12286  
## Balanced Accuracy 0.77410 0.85419 0.77336 0.86887

proc.time()- ptm

## user system elapsed   
## 6.20 1.70 8.05

### Decision Tree: With Cross Validation

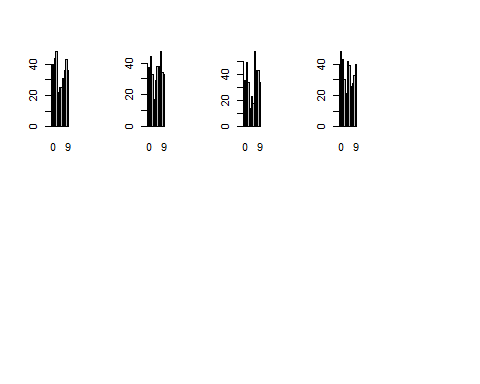
In an attempt to see higher accuracy, we will focus on using cross fold validation, looking at 4 splits and using the average of all scores as our final result. For this instance, we see an accuracy of 62%, and an elapsed computing time of 4.6.

# Set Seed for reproducability  
set.seed(22)  
N <- nrow(train)  
# Set number of splits  
kfolds <- 4  
# Set holdout variable for split  
holdout <- split(sample(1:N), 1:kfolds)  
# head(holdout)  
  
# Run training and testing for each k-fold  
AllResults <- list()  
AllLabels <- list()  
AllAccuracy <- list()  
par(mfrow = c(2,5))  
for (k in 1:kfolds) {  
 # Segement  
 digitTest <- train[holdout[[k]],]  
 digitTrain <- train[-holdout[[k]],]  
 # Remove labels from Test  
 digitTestWL <- digitTest$label  
 digitTestNL <- digitTest[-c(1)]  
 # Run naive Bayes  
 dtrainCV <- rpart(label~.,data=digitTrain, method="class")  
 # Run Predictiopn  
 pdtrainCV <- predict(dtrainCV, digitTestNL, type="class")  
 pdtrainCV  
 # Confusion Matrix  
 conf1 <- confusionMatrix(pdtrainCV, digitTest$label)  
 conf1  
 # Set up for plot  
 AllResults <- c(AllResults,pdtrainCV)  
 AllLabels <- c(AllLabels, digitTestWL)  
 AllAccuracy <- c(AllAccuracy, conf1$overall[1])  
 plot(pdtrainCV)  
}  
  
# Check Accuracy of all   
mean(unlist(AllAccuracy))

## [1] 0.62

proc.time()- ptm

## user system elapsed   
## 10.36 1.87 12.39



## Naive Bayes

The Naive Bayes algorithm works to classify based on Bayes’ theorem, though with strong independence between attrubutes allowing for a reduction in calculation from Bayes, hence the term “Naive.”

### Naive Bayes: Baseline

For our naive bayes analysis, we first need to segment the training data with and without a label, to learn on itself.

# Remove label to set up clean data  
trainNL <- train[,-1]  
trainWL <- train$label

Our baseline naive bayes analysis resulted in an accuracy of 50.86% and an elapsed computing time of 14.9. These are both the lowest accuracy and the longest computing time of our analysis yet.

set.seed(22)  
# Run the Naive Bayes on the training data  
nb1 <- naiveBayes(label ~ ., data=train, na.action=na.pass)  
  
# Predict the outcomes of the test data  
pnb1 <- predict(nb1, trainNL)  
#pnb1\_df <- as.data.frame(pnb1)  
  
confusionMatrix(pnb1,trainWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 131 0 6 7 0 12 1 1 1 1  
## 1 2 166 9 25 6 24 18 16 42 15  
## 2 0 0 39 0 1 0 1 0 1 0  
## 3 0 0 4 50 0 1 0 0 0 0  
## 4 0 0 0 0 21 1 0 3 0 1  
## 5 1 0 5 4 0 10 0 0 1 0  
## 6 0 0 23 5 1 1 84 0 0 0  
## 7 0 0 1 1 1 2 0 18 0 0  
## 8 2 1 34 32 2 58 2 3 66 0  
## 9 6 0 9 16 99 18 39 96 26 127  
##   
## Overall Statistics  
##   
## Accuracy : 0.5086   
## 95% CI : (0.482, 0.5351)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.4509   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.92254 0.9940 0.30000 0.35714 0.16031 0.078740  
## Specificity 0.97695 0.8727 0.99764 0.99603 0.99606 0.991359  
## Pos Pred Value 0.81875 0.5139 0.92857 0.90909 0.80769 0.476190  
## Neg Pred Value 0.99113 0.9991 0.93299 0.93309 0.91994 0.915156  
## Prevalence 0.10143 0.1193 0.09286 0.10000 0.09357 0.090714  
## Detection Rate 0.09357 0.1186 0.02786 0.03571 0.01500 0.007143  
## Detection Prevalence 0.11429 0.2307 0.03000 0.03929 0.01857 0.015000  
## Balanced Accuracy 0.94974 0.9333 0.64882 0.67659 0.57818 0.535050  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.57931 0.13139 0.48175 0.88194  
## Specificity 0.97610 0.99604 0.89390 0.75398  
## Pos Pred Value 0.73684 0.78261 0.33000 0.29128  
## Neg Pred Value 0.95257 0.91358 0.94083 0.98237  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.06000 0.01286 0.04714 0.09071  
## Detection Prevalence 0.08143 0.01643 0.14286 0.31143  
## Balanced Accuracy 0.77770 0.56371 0.68783 0.81796

proc.time()- ptm

## user system elapsed   
## 22.11 1.87 24.16

### Naive Bayes: With Less Dimensionality Using PCA of 5

Similar to the last example, we are instead segmenting our training data for the PCA version.

# Remove label to set up clean data  
pcatrainNL <- pcatrain[,-1]  
pcatrainWL <- pcatrain$label

This analysis with the PCA data improved our basline naive bayes, returning an accuracy of 64.43% and 1.2 elapsed computing time.

set.seed(22)  
# Run the Naive Bayes on the training data  
nb2 <- naiveBayes(label ~ ., data=pcatrain, na.action=na.pass)  
  
# Predict the outcomes of the test data  
pnb2 <- predict(nb2, pcatrainNL)  
#pnb2\_df <- as.data.frame(pnb2)  
  
confusionMatrix(pnb2,pcatrainWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 119 0 6 1 0 12 3 4 6 1  
## 1 0 154 2 1 1 2 7 7 6 2  
## 2 1 0 59 3 3 3 24 0 9 0  
## 3 1 2 5 102 0 20 1 0 21 2  
## 4 0 0 6 1 67 2 4 12 2 27  
## 5 9 0 3 5 1 60 1 3 5 2  
## 6 4 6 41 1 13 9 90 1 6 1  
## 7 0 1 0 1 6 4 0 87 1 20  
## 8 8 4 7 21 3 14 15 2 76 1  
## 9 0 0 1 4 37 1 0 21 5 88  
##   
## Overall Statistics  
##   
## Accuracy : 0.6443   
## 95% CI : (0.6186, 0.6694)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.6042   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.8380 0.9222 0.45385 0.72857 0.51145 0.47244  
## Specificity 0.9738 0.9773 0.96614 0.95873 0.95745 0.97722  
## Pos Pred Value 0.7829 0.8462 0.57843 0.66234 0.55372 0.67416  
## Neg Pred Value 0.9816 0.9893 0.94530 0.96950 0.94996 0.94889  
## Prevalence 0.1014 0.1193 0.09286 0.10000 0.09357 0.09071  
## Detection Rate 0.0850 0.1100 0.04214 0.07286 0.04786 0.04286  
## Detection Prevalence 0.1086 0.1300 0.07286 0.11000 0.08643 0.06357  
## Balanced Accuracy 0.9059 0.9497 0.70999 0.84365 0.73445 0.72483  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.62069 0.63504 0.55474 0.61111  
## Specificity 0.93466 0.97387 0.94062 0.94506  
## Pos Pred Value 0.52326 0.72500 0.50331 0.56051  
## Neg Pred Value 0.95521 0.96094 0.95116 0.95495  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.06429 0.06214 0.05429 0.06286  
## Detection Prevalence 0.12286 0.08571 0.10786 0.11214  
## Balanced Accuracy 0.77768 0.80445 0.74768 0.77809

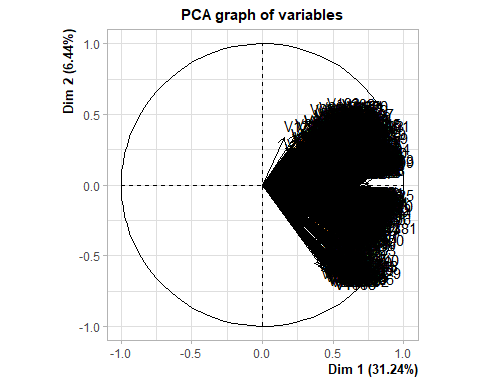
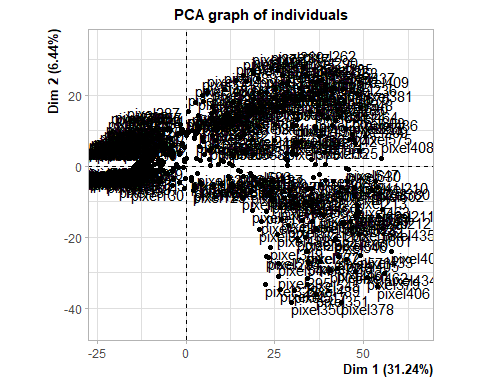
proc.time()- ptm

## user system elapsed   
## 22.26 1.89 24.33

### Naive Bayes: With Less Dimensionality Using PCA of 15

Seeing positive results with the dimensionality, we are now increasing the dimensionality to a PCA of 15 in hopes of improving our accuracy.

# Set up pca train parameters  
pcadigits <- PCA(t(select(train, -label)), ncp = 15)



# Regroup into data frame  
pcatrain <- data.frame(train$label, pcadigits$var$coord)  
# Rename first column  
names(pcatrain)[1] <- "label"  
dim(pcatrain) # actually runs at 16 dimensions

## [1] 1400 16

# Remove label to set up clean data  
pcatrainNL <- pcatrain[,-1]  
pcatrainWL <- pcatrain$label  
proc.time()- ptm

## user system elapsed   
## 24.23 2.05 26.45

Increasing the PCA, we drastically improve our results, with an accuracy of 82.57% and only 2.02 elapsed computing time.

set.seed(22)  
# Run the Naive Bayes on the training data  
nb3 <- naiveBayes(label ~ ., data=pcatrain, na.action=na.pass)  
  
# Predict the outcomes of the test data  
pnb3 <- predict(nb3, pcatrainNL)  
#pnb2\_df <- as.data.frame(pnb2)  
  
confusionMatrix(pnb3, pcatrainWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 124 0 0 0 1 3 5 4 0 1  
## 1 0 162 2 1 0 2 4 5 1 0  
## 2 0 1 102 3 1 2 0 1 1 1  
## 3 4 0 5 112 0 9 0 0 12 2  
## 4 0 0 2 0 92 2 1 1 5 18  
## 5 6 2 2 4 4 104 3 1 6 4  
## 6 8 1 3 1 8 0 131 0 1 1  
## 7 0 1 3 0 1 2 0 113 0 8  
## 8 0 0 11 14 1 1 1 1 108 1  
## 9 0 0 0 5 23 2 0 11 3 108  
##   
## Overall Statistics  
##   
## Accuracy : 0.8257   
## 95% CI : (0.8048, 0.8452)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.8062   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.87324 0.9701 0.78462 0.8000 0.70229 0.81890  
## Specificity 0.98887 0.9878 0.99213 0.9746 0.97715 0.97486  
## Pos Pred Value 0.89855 0.9153 0.91071 0.7778 0.76033 0.76471  
## Neg Pred Value 0.98574 0.9959 0.97826 0.9777 0.96951 0.98180  
## Prevalence 0.10143 0.1193 0.09286 0.1000 0.09357 0.09071  
## Detection Rate 0.08857 0.1157 0.07286 0.0800 0.06571 0.07429  
## Detection Prevalence 0.09857 0.1264 0.08000 0.1029 0.08643 0.09714  
## Balanced Accuracy 0.93106 0.9789 0.88837 0.8873 0.83972 0.89688  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.90345 0.82482 0.78832 0.75000  
## Specificity 0.98167 0.98812 0.97625 0.96497  
## Pos Pred Value 0.85065 0.88281 0.78261 0.71053  
## Neg Pred Value 0.98876 0.98113 0.97702 0.97115  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.09357 0.08071 0.07714 0.07714  
## Detection Prevalence 0.11000 0.09143 0.09857 0.10857  
## Balanced Accuracy 0.94256 0.90647 0.88228 0.85748

proc.time()- ptm

## user system elapsed   
## 24.65 2.20 27.03

### Naive Bayes: Using k-folds Cross Validation and PCA

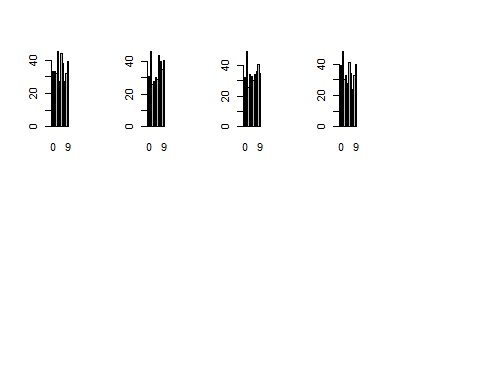
In order to see if we can elaborate on our last model, we will now add cross validation with 4 folds to see if it can improve our results. We oddly find .81 elapsed computing time, but 80.71% accuracy.

# Set Seed for reproducability  
set.seed(22)  
# Set number of splits  
kfolds <- 4  
# Set holdout variable for split  
holdout <- split(sample(1:N), 1:kfolds)  
# head(holdout)  
  
# Run training and testing for each k-fold  
AllResults <- list()  
AllLabels <- list()  
AllAccuracy <- list()  
par(mfrow = c(2,5))  
for (k in 1:kfolds) {  
 # Segement  
 pdigitTest <- pcatrain[holdout[[k]],]  
 pdigitTrain <- pcatrain[-holdout[[k]],]  
 # Remove labels from Test  
 pdigitTestWL <- pdigitTest$label  
 pdigitTestNL <- pdigitTest[-c(1)]  
 # Run naive Bayes  
 nbtrain <- naiveBayes(label~.,data=pdigitTrain,na.action=na.pass)  
 # Run Predictiopn  
 nbpred <- predict(nbtrain, pdigitTestNL)  
 nbpred  
 # Confusion Matrix  
 conf1 <- confusionMatrix(nbpred, pdigitTest$label)  
 conf1  
 # Set up for plot  
 AllResults <- c(AllResults,nbpred)  
 AllLabels <- c(AllLabels, pdigitTestWL)  
 AllAccuracy <- c(AllAccuracy, conf1$overall[1])  
 plot(nbpred)  
}  
  
# Check Accuracy of all   
mean(unlist(AllAccuracy))

## [1] 0.8071429

proc.time()- ptm

## user system elapsed   
## 25.03 2.23 27.44



## kNN

The k Nearest Neighbor algorithm stores available cases as data based on the distance between the values, or a similarity measure. It aims to classify like-variables based on proximity to how their neighbors are classified. It adds value by being a good algorithm for a complex decision, but can be less useful on noisy data. A large k value can cause underfitting, and the best results are typically from a k value less than 10.

### kNN: Baseline

Running our baseline kNN, we are setting the k value equal to the rounded square root of the train rows, or 37. This returns an accuracy of 85% and an elapsed computing time of 2.67.

set.seed(22)  
# set k equal to the rounded square root of the train rows  
# K is set to 37  
k <- round(sqrt(nrow(train)))  
# Set the prediction of kNN  
kNN1 <- knn(trainNL, trainNL, trainWL, k=k)  
# Evaluate  
confusionMatrix(kNN1, trainWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 133 0 1 1 0 0 3 1 0 1  
## 1 0 166 20 8 5 11 9 12 15 3  
## 2 0 0 91 0 0 0 0 0 0 0  
## 3 0 0 2 118 0 11 0 0 6 0  
## 4 0 0 3 1 103 1 1 1 0 4  
## 5 2 0 0 1 0 96 1 0 5 0  
## 6 4 0 2 1 0 3 131 0 1 0  
## 7 0 1 7 1 0 2 0 118 0 4  
## 8 1 0 4 4 0 0 0 0 102 0  
## 9 2 0 0 5 23 3 0 5 8 132  
##   
## Overall Statistics  
##   
## Accuracy : 0.85   
## 95% CI : (0.8302, 0.8683)  
## No Information Rate : 0.1193   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.8329   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.9366 0.9940 0.70000 0.84286 0.78626 0.75591  
## Specificity 0.9944 0.9327 1.00000 0.98492 0.99133 0.99293  
## Pos Pred Value 0.9500 0.6667 1.00000 0.86131 0.90351 0.91429  
## Neg Pred Value 0.9929 0.9991 0.97021 0.98258 0.97823 0.97606  
## Prevalence 0.1014 0.1193 0.09286 0.10000 0.09357 0.09071  
## Detection Rate 0.0950 0.1186 0.06500 0.08429 0.07357 0.06857  
## Detection Prevalence 0.1000 0.1779 0.06500 0.09786 0.08143 0.07500  
## Balanced Accuracy 0.9655 0.9633 0.85000 0.91389 0.88880 0.87442  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.90345 0.86131 0.74453 0.91667  
## Specificity 0.99124 0.98812 0.99287 0.96338  
## Pos Pred Value 0.92254 0.88722 0.91892 0.74157  
## Neg Pred Value 0.98887 0.98500 0.97285 0.99018  
## Prevalence 0.10357 0.09786 0.09786 0.10286  
## Detection Rate 0.09357 0.08429 0.07286 0.09429  
## Detection Prevalence 0.10143 0.09500 0.07929 0.12714  
## Balanced Accuracy 0.94734 0.92472 0.86870 0.94002

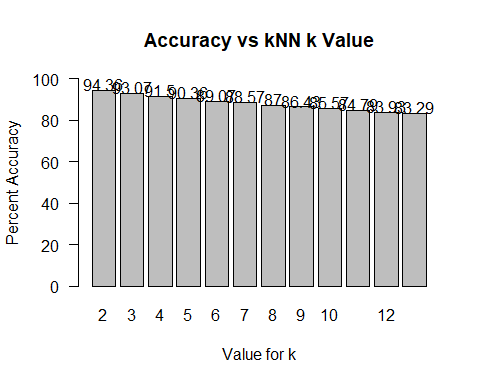
proc.time()- ptm

## user system elapsed   
## 26.65 2.23 29.06

### kNN: Optimal k Value Evaluation

This section shows visually the accuracy output for different k values from 2-13. We see a declining accuracy, which could be alarming. We want to be cautious with too low of a k value as it may cause overfitting. The optimal k value is not visually apparent, so we will choose a k value of 5.

set.seed(22)  
AllAccuracy.kNN <- list()  
# for loop to evaluate different accuracies based on different k values  
for (i in seq(2,46,4)) {  
 knn2 <- knn(trainNL, trainNL, trainWL, k=i)  
 knn2CM <- confusionMatrix(knn2, trainWL)  
 knn2ACC <- round(knn2CM$overall[1]\*100,2)  
 AllAccuracy.kNN <- c(AllAccuracy.kNN, knn2CM$overall[1])  
}  
knnACC.df <- data.frame(kvalue=seq(2,13,1), as.data.frame((unlist(AllAccuracy.kNN))))  
colnames(knnACC.df)[2] <- 'kNN.Percent.Accuracy'  
knnACC.df$kNN.Percent.Accuracy <- round(knnACC.df$kNN.Percent.Accuracy\*100,2)  
x <- barplot(knnACC.df$kNN.Percent.Accuracy, names.arg = knnACC.df$kvalue, xlab='Value for k', ylab = 'Percent Accuracy', main = 'Accuracy vs kNN k Value', las=1, ylim=c(0,100))  
y <- knnACC.df$kNN.Percent.Accuracy  
text(x,y+3,labels=as.character(y))



proc.time()- ptm

## user system elapsed   
## 45.64 2.25 48.08

### kNN: With Cross Fold Validation

Using our k value of 5, we are now running our kNN algorithm with cross fold validation. This returns a strong accuracy of 89.21% with an elapsed computing time of 4.08.

set.seed(22)  
AllResults <- data.frame(orig=c(), pred=c())  
for (k in 1:kfolds) {  
 # Segement  
 digitTest <- train[holdout[[k]],]  
 digitTrain <- train[-holdout[[k]],]  
 # Remove labels from Test  
 digitTestWL <- digitTest$label  
 digitTestNL <- digitTest[-c(1)]  
 # Remove labels from train  
 digitTrainWL <- digitTrain$label  
 digitTrainNL <- digitTrain[-c(1)]  
 kNN3 <- knn(digitTrain, digitTest, digitTrainWL, k=5)  
 AllResults <- rbind(AllResults, data.frame(orig=digitTestWL, pred=kNN3))  
}  
confusionMatrix(AllResults$orig, AllResults$pred)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 138 0 0 0 1 0 2 0 0 1  
## 1 0 166 0 0 0 0 0 1 0 0  
## 2 0 5 105 4 1 2 2 7 4 0  
## 3 0 1 0 128 0 3 1 0 5 2  
## 4 0 2 0 0 110 0 2 0 0 17  
## 5 0 3 0 7 1 104 3 1 5 3  
## 6 2 2 0 0 0 2 139 0 0 0  
## 7 1 7 0 0 0 0 0 124 0 5  
## 8 1 6 0 9 1 9 2 2 102 5  
## 9 1 1 0 1 3 0 0 5 0 133  
##   
## Overall Statistics  
##   
## Accuracy : 0.8921   
## 95% CI : (0.8747, 0.9079)  
## No Information Rate : 0.1379   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.88   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.96503 0.8601 1.00000 0.85906 0.94017 0.86667  
## Specificity 0.99682 0.9992 0.98069 0.99041 0.98363 0.98203  
## Pos Pred Value 0.97183 0.9940 0.80769 0.91429 0.83969 0.81890  
## Neg Pred Value 0.99603 0.9781 1.00000 0.98333 0.99448 0.98743  
## Prevalence 0.10214 0.1379 0.07500 0.10643 0.08357 0.08571  
## Detection Rate 0.09857 0.1186 0.07500 0.09143 0.07857 0.07429  
## Detection Prevalence 0.10143 0.1193 0.09286 0.10000 0.09357 0.09071  
## Balanced Accuracy 0.98093 0.9296 0.99035 0.92473 0.96190 0.92435  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.92053 0.88571 0.87931 0.8012  
## Specificity 0.99520 0.98968 0.97274 0.9911  
## Pos Pred Value 0.95862 0.90511 0.74453 0.9236  
## Neg Pred Value 0.99044 0.98733 0.98892 0.9737  
## Prevalence 0.10786 0.10000 0.08286 0.1186  
## Detection Rate 0.09929 0.08857 0.07286 0.0950  
## Detection Prevalence 0.10357 0.09786 0.09786 0.1029  
## Balanced Accuracy 0.95786 0.93770 0.92603 0.8961

proc.time()- ptm

## user system elapsed   
## 48.59 2.27 51.06

## SVM

Support Vector Machines (SVM) work to visualize the data into multidimensional space, then segment the data points based off of optimal hyperplanes, maximizing the margins between the closest points and the “middle” hyperplane.

### SVM: Model with a Polynomial Kernel

Our first SVM we will perform with a polynomial kernel. The Polynomial kernel is supposed to be best used for image processing. This evaluation returns an accuracy of 92% with 2.49 elapsed computing time.

# set seed for reproducability  
set.seed(22)  
  
# Using the already partitioned digiTrain and digitTest  
# svm for a polynomial kernel  
svm1 <- svm(label~., data=digitTrain, kernel="polynomial", cost=.1, scale=FALSE)  
  
# predict  
psvm1 <- predict(svm1, digitTestNL, type="class")  
  
# check performance  
confusionMatrix(psvm1, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 41 0 0 0 0 0 0 0 0 0  
## 1 3 46 1 2 0 2 2 2 0 0  
## 2 0 0 28 0 0 0 0 1 0 0  
## 3 0 0 0 30 0 0 0 0 1 0  
## 4 0 0 1 0 27 0 0 0 0 2  
## 5 0 0 0 0 0 34 0 0 0 0  
## 6 1 0 0 0 1 0 30 0 0 0  
## 7 0 0 0 1 0 0 0 23 0 0  
## 8 0 0 0 1 0 1 0 0 29 0  
## 9 0 0 0 1 4 0 0 0 1 34  
##   
## Overall Statistics  
##   
## Accuracy : 0.92   
## 95% CI : (0.8865, 0.9462)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9107   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.9111 1.0000 0.93333 0.85714 0.84375 0.91892  
## Specificity 1.0000 0.9605 0.99687 0.99683 0.99057 1.00000  
## Pos Pred Value 1.0000 0.7931 0.96552 0.96774 0.90000 1.00000  
## Neg Pred Value 0.9871 1.0000 0.99377 0.98433 0.98438 0.99051  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1171 0.1314 0.08000 0.08571 0.07714 0.09714  
## Detection Prevalence 0.1171 0.1657 0.08286 0.08857 0.08571 0.09714  
## Balanced Accuracy 0.9556 0.9803 0.96510 0.92698 0.91716 0.95946  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.93750 0.88462 0.93548 0.94444  
## Specificity 0.99371 0.99691 0.99373 0.98089  
## Pos Pred Value 0.93750 0.95833 0.93548 0.85000  
## Neg Pred Value 0.99371 0.99080 0.99373 0.99355  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.08571 0.06571 0.08286 0.09714  
## Detection Prevalence 0.09143 0.06857 0.08857 0.11429  
## Balanced Accuracy 0.96561 0.94076 0.96461 0.96267

proc.time()- ptm

## user system elapsed   
## 50.15 2.27 52.63

### SVM: Model with a Linear Kernel

The most simplistic form of SVM is the linear kernel. Despite its simplicity, we actually return an improved 92.29% accuracy and 2.47 elapsed computing time.

# set seed for reproducability  
set.seed(22)  
  
# Using the already partitioned digiTrain and digitTest  
# svm for a linear kernel  
svm2 <- svm(label~., data=digitTrain, kernel="linear", cost=.1, scale=FALSE)  
  
# predict  
psvm2 <- predict(svm2, digitTestNL, type="class")  
  
# check performance  
confusionMatrix(psvm2, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 45 0 0 0 0 0 0 0 0 0  
## 1 0 46 0 1 0 0 0 0 0 0  
## 2 0 0 26 0 0 0 0 1 1 0  
## 3 0 0 0 30 0 1 0 0 1 0  
## 4 0 0 1 0 28 2 0 0 0 3  
## 5 0 0 1 2 0 32 0 0 1 0  
## 6 0 0 0 0 0 1 32 0 0 0  
## 7 0 0 1 0 0 0 0 23 0 0  
## 8 0 0 1 2 0 1 0 0 28 0  
## 9 0 0 0 0 4 0 0 2 0 33  
##   
## Overall Statistics  
##   
## Accuracy : 0.9229   
## 95% CI : (0.8897, 0.9485)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.914   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 1.0000 1.0000 0.86667 0.85714 0.87500 0.86486  
## Specificity 1.0000 0.9967 0.99375 0.99365 0.98113 0.98722  
## Pos Pred Value 1.0000 0.9787 0.92857 0.93750 0.82353 0.88889  
## Neg Pred Value 1.0000 1.0000 0.98758 0.98428 0.98734 0.98408  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1286 0.1314 0.07429 0.08571 0.08000 0.09143  
## Detection Prevalence 0.1286 0.1343 0.08000 0.09143 0.09714 0.10286  
## Balanced Accuracy 1.0000 0.9984 0.93021 0.92540 0.92807 0.92604  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 1.00000 0.88462 0.90323 0.91667  
## Specificity 0.99686 0.99691 0.98746 0.98089  
## Pos Pred Value 0.96970 0.95833 0.87500 0.84615  
## Neg Pred Value 1.00000 0.99080 0.99057 0.99035  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.09143 0.06571 0.08000 0.09429  
## Detection Prevalence 0.09429 0.06857 0.09143 0.11143  
## Balanced Accuracy 0.99843 0.94076 0.94534 0.94878

proc.time()- ptm

## user system elapsed   
## 51.64 2.28 54.13

### SVM: Model with Radial Kernel

The Radial kernel, or the Gaussian RBG kernel is best used when there is no priod knowledge of the data. This SVM however, mistakes all values for a “1” and is one of the worst performing algorithm yet, with an accuracy of 13.14% and 4.07 elapsed computing time.

# set seed for reproducability  
set.seed(22)  
  
# Using the already partitioned digiTrain and digitTest  
# svm for a radial kernel  
svm3 <- svm(label~., data=digitTrain, kernel="radial", cost=.1, scale=FALSE)  
  
# predict  
psvm3 <- predict(svm3, digitTestNL, type="class")  
  
# check performance  
confusionMatrix(psvm3, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 0 0 0 0 0 0 0 0 0 0  
## 1 45 46 30 35 32 37 32 26 31 36  
## 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  
##   
## Overall Statistics  
##   
## Accuracy : 0.1314   
## 95% CI : (0.0979, 0.1714)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : 0.5238   
##   
## Kappa : 0   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.0000 1.0000 0.00000 0.0 0.00000 0.0000  
## Specificity 1.0000 0.0000 1.00000 1.0 1.00000 1.0000  
## Pos Pred Value NaN 0.1314 NaN NaN NaN NaN  
## Neg Pred Value 0.8714 NaN 0.91429 0.9 0.90857 0.8943  
## Prevalence 0.1286 0.1314 0.08571 0.1 0.09143 0.1057  
## Detection Rate 0.0000 0.1314 0.00000 0.0 0.00000 0.0000  
## Detection Prevalence 0.0000 1.0000 0.00000 0.0 0.00000 0.0000  
## Balanced Accuracy 0.5000 0.5000 0.50000 0.5 0.50000 0.5000  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.00000 0.00000 0.00000 0.0000  
## Specificity 1.00000 1.00000 1.00000 1.0000  
## Pos Pred Value NaN NaN NaN NaN  
## Neg Pred Value 0.90857 0.92571 0.91143 0.8971  
## Prevalence 0.09143 0.07429 0.08857 0.1029  
## Detection Rate 0.00000 0.00000 0.00000 0.0000  
## Detection Prevalence 0.00000 0.00000 0.00000 0.0000  
## Balanced Accuracy 0.50000 0.50000 0.50000 0.5000

proc.time()- ptm

## user system elapsed   
## 54.65 2.28 57.14

### SVM: Model with Sigmoid Kernel

The Sigmoid is best used for neural networks, but we will examine it, because why not! This performs similarly to the radial kernel, with the same accuracy of 13.14% and 4.05 elapsed computing time.

# set seed for reproducability  
set.seed(22)  
  
# Using the already partitioned digiTrain and digitTest  
# svm for a sigmoid kernel  
svm4 <- svm(label~., data=digitTrain, kernel="sigmoid", cost=.1, scale=FALSE)  
  
# predict  
psvm4 <- predict(svm4, digitTestNL, type="class")  
  
# check performance  
confusionMatrix(psvm4, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 0 0 0 0 0 0 0 0 0 0  
## 1 45 46 30 35 32 37 32 26 31 36  
## 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  
##   
## Overall Statistics  
##   
## Accuracy : 0.1314   
## 95% CI : (0.0979, 0.1714)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : 0.5238   
##   
## Kappa : 0   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.0000 1.0000 0.00000 0.0 0.00000 0.0000  
## Specificity 1.0000 0.0000 1.00000 1.0 1.00000 1.0000  
## Pos Pred Value NaN 0.1314 NaN NaN NaN NaN  
## Neg Pred Value 0.8714 NaN 0.91429 0.9 0.90857 0.8943  
## Prevalence 0.1286 0.1314 0.08571 0.1 0.09143 0.1057  
## Detection Rate 0.0000 0.1314 0.00000 0.0 0.00000 0.0000  
## Detection Prevalence 0.0000 1.0000 0.00000 0.0 0.00000 0.0000  
## Balanced Accuracy 0.5000 0.5000 0.50000 0.5 0.50000 0.5000  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.00000 0.00000 0.00000 0.0000  
## Specificity 1.00000 1.00000 1.00000 1.0000  
## Pos Pred Value NaN NaN NaN NaN  
## Neg Pred Value 0.90857 0.92571 0.91143 0.8971  
## Prevalence 0.09143 0.07429 0.08857 0.1029  
## Detection Rate 0.00000 0.00000 0.00000 0.0000  
## Detection Prevalence 0.00000 0.00000 0.00000 0.0000  
## Balanced Accuracy 0.50000 0.50000 0.50000 0.5000

proc.time()- ptm

## user system elapsed   
## 57.32 2.30 59.83

### SVM: Model with a Linear Kernel and PCA

With the linear kernel performing the best, we will re-run the model with the PCA (less dimensional) data. This returns a lowered 79.43% accuracy, and 1.2 elapsed computing time.

# set seed for reproducability  
set.seed(22)  
  
# Using the already partitioned digiTrain and digitTest  
# svm for a linear kernel  
svm5 <- svm(label~., data=pdigitTrain, kernel="linear", cost=.1, scale=FALSE)  
  
# predict  
psvm5 <- predict(svm5, pdigitTestNL, type="class")  
  
# check performance  
confusionMatrix(psvm5, pdigitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 38 0 0 0 0 0 0 1 0 0  
## 1 0 44 0 0 0 2 0 1 0 0  
## 2 0 0 27 0 0 1 2 1 0 0  
## 3 0 0 0 29 0 11 0 0 6 0  
## 4 0 0 1 0 17 0 0 0 0 2  
## 5 3 0 0 1 1 18 0 0 0 0  
## 6 3 0 0 0 2 2 30 0 0 0  
## 7 0 0 0 0 0 0 0 21 0 3  
## 8 1 2 1 3 0 0 0 1 23 0  
## 9 0 0 1 2 12 3 0 1 2 31  
##   
## Overall Statistics  
##   
## Accuracy : 0.7943   
## 95% CI : (0.7481, 0.8354)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.7707   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.8444 0.9565 0.90000 0.82857 0.53125 0.48649  
## Specificity 0.9967 0.9901 0.98750 0.94603 0.99057 0.98403  
## Pos Pred Value 0.9744 0.9362 0.87097 0.63043 0.85000 0.78261  
## Neg Pred Value 0.9775 0.9934 0.99060 0.98026 0.95455 0.94190  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1086 0.1257 0.07714 0.08286 0.04857 0.05143  
## Detection Prevalence 0.1114 0.1343 0.08857 0.13143 0.05714 0.06571  
## Balanced Accuracy 0.9206 0.9733 0.94375 0.88730 0.76091 0.73526  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.93750 0.80769 0.74194 0.86111  
## Specificity 0.97799 0.99074 0.97492 0.93312  
## Pos Pred Value 0.81081 0.87500 0.74194 0.59615  
## Neg Pred Value 0.99361 0.98466 0.97492 0.98322  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.08571 0.06000 0.06571 0.08857  
## Detection Prevalence 0.10571 0.06857 0.08857 0.14857  
## Balanced Accuracy 0.95774 0.89922 0.85843 0.89712

proc.time()- ptm

## user system elapsed   
## 57.43 2.31 59.95

## Random Forest

The random forest is a very diverse ensemble learning algorithm for machine learning due to its ease-of-use adn often great results. It performs supervised learning to build decision trees via the bagging method. It then predicts based on “votes” from the created subsets of decision trees.

### Random Forest: Baseline

Our baseline random forest is set with 500 trees and 2 splits. This returns an accuracy of 91.71% and a much larger than previous elapsed computing time of 19.27.

# set seed for reproducability  
set.seed(22)  
  
rf1 <- randomForest(label ~ ., digitTrain, importance = T)  
prf1 <- predict(rf1, digitTestNL)  
confusionMatrix(prf1, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 42 0 0 0 0 0 0 0 0 0  
## 1 0 46 0 1 0 0 0 1 0 0  
## 2 0 0 27 0 0 0 0 1 0 0  
## 3 0 0 1 31 0 1 0 0 2 0  
## 4 0 0 1 1 26 1 0 0 0 3  
## 5 0 0 0 0 0 34 0 0 0 0  
## 6 2 0 0 0 1 0 32 0 0 0  
## 7 0 0 0 1 0 0 0 22 0 0  
## 8 1 0 0 1 0 1 0 1 28 0  
## 9 0 0 1 0 5 0 0 1 1 33  
##   
## Overall Statistics  
##   
## Accuracy : 0.9171   
## 95% CI : (0.8832, 0.9438)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9076   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.9333 1.0000 0.90000 0.88571 0.81250 0.91892  
## Specificity 1.0000 0.9934 0.99687 0.98730 0.98113 1.00000  
## Pos Pred Value 1.0000 0.9583 0.96429 0.88571 0.81250 1.00000  
## Neg Pred Value 0.9903 1.0000 0.99068 0.98730 0.98113 0.99051  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1200 0.1314 0.07714 0.08857 0.07429 0.09714  
## Detection Prevalence 0.1200 0.1371 0.08000 0.10000 0.09143 0.09714  
## Balanced Accuracy 0.9667 0.9967 0.94844 0.93651 0.89682 0.95946  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 1.00000 0.84615 0.90323 0.91667  
## Specificity 0.99057 0.99691 0.98746 0.97452  
## Pos Pred Value 0.91429 0.95652 0.87500 0.80488  
## Neg Pred Value 1.00000 0.98777 0.99057 0.99029  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.09143 0.06286 0.08000 0.09429  
## Detection Prevalence 0.10000 0.06571 0.09143 0.11714  
## Balanced Accuracy 0.99528 0.92153 0.94534 0.94559

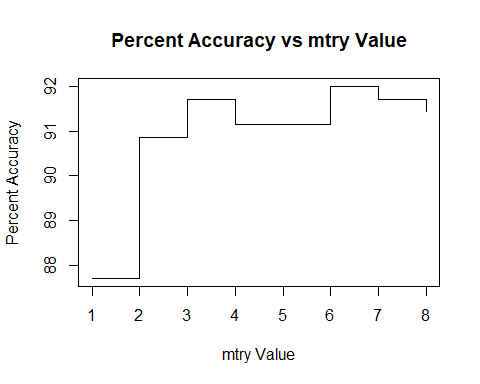
proc.time()- ptm

## user system elapsed   
## 75.42 2.31 77.94

### Random Forest: Identifying “Best” mtry

In order to identify the “best” value for mtry, we run a for loop for different values, and plot those with their accuracy. This is taxing on the computer’s performance, but should help us isolate a value for better predictions in the future. We see the value for 6 and 7 seems to be the best.

# set seed for reproducability  
set.seed(22)  
a <- c()  
i <- 5  
for (i in 3:10) {  
 rf2 <- randomForest(label ~.,digitTrain, ntree=500,   
 mtry=i, importance = T)  
 prf2 <- predict(rf2, digitTestNL, type="class")  
 a[i-2] <- mean(prf2 == digitTestWL)  
}  
  
plot(a\*100, type='s', ylab = "Percent Accuracy",   
 xlab = "mtry Value", main = "Percent Accuracy vs mtry Value")



proc.time()- ptm

## user system elapsed   
## 229.82 2.52 232.59

### Random Forest: Using optimal mtry

Using the optimal value selected above, our new random forest returns an accuracy of 91.14% with an elapsed computing time of 21.82.

# set seed for reproducability  
set.seed(22)  
# run random forest with 6 and 7 mtry   
# (they produce the same accuracy)  
rf3 <- randomForest(label ~ ., digitTrain, ntree=500, mtry=7, importance = T)  
prf3 <- predict(rf3, digitTestNL)  
confusionMatrix(prf3, digitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 42 0 0 0 0 0 0 0 0 0  
## 1 0 46 0 1 0 0 0 1 0 0  
## 2 0 0 28 0 0 0 0 1 0 0  
## 3 0 0 0 30 0 1 0 0 2 0  
## 4 0 0 2 1 26 1 0 0 0 3  
## 5 0 0 0 0 0 33 0 0 0 0  
## 6 3 0 0 0 1 1 32 0 0 0  
## 7 0 0 0 1 0 0 0 22 0 1  
## 8 0 0 0 2 0 1 0 1 28 0  
## 9 0 0 0 0 5 0 0 1 1 32  
##   
## Overall Statistics  
##   
## Accuracy : 0.9114   
## 95% CI : (0.8766, 0.939)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9013   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.9333 1.0000 0.93333 0.85714 0.81250 0.89189  
## Specificity 1.0000 0.9934 0.99687 0.99048 0.97799 1.00000  
## Pos Pred Value 1.0000 0.9583 0.96552 0.90909 0.78788 1.00000  
## Neg Pred Value 0.9903 1.0000 0.99377 0.98423 0.98107 0.98738  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1200 0.1314 0.08000 0.08571 0.07429 0.09429  
## Detection Prevalence 0.1200 0.1371 0.08286 0.09429 0.09429 0.09429  
## Balanced Accuracy 0.9667 0.9967 0.96510 0.92381 0.89524 0.94595  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 1.00000 0.84615 0.90323 0.88889  
## Specificity 0.98428 0.99383 0.98746 0.97771  
## Pos Pred Value 0.86486 0.91667 0.87500 0.82051  
## Neg Pred Value 1.00000 0.98773 0.99057 0.98714  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.09143 0.06286 0.08000 0.09143  
## Detection Prevalence 0.10571 0.06857 0.09143 0.11143  
## Balanced Accuracy 0.99214 0.91999 0.94534 0.93330

proc.time()- ptm

## user system elapsed   
## 250.89 2.53 253.67

### Random Forest: Using Optimal mtry and PCA

In hopes of reducing computing time, we will re run our previous random forest, but this time using the less dimensional PCA data. This returns a slightly lowered accuracy of 87.14% but a much improved computing time of 1.96.

# set seed for reproducability  
set.seed(22)  
# run random forest with 6 and 7 mtry, this time for pca data  
# (they produce the same accuracy)  
rf4 <- randomForest(label ~ ., pdigitTrain, ntree=500, mtry=7, importance = T)  
prf4 <- predict(rf4, pdigitTestNL)  
confusionMatrix(prf4, pdigitTestWL)

## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 40 0 0 0 0 0 0 1 0 1  
## 1 0 46 0 0 0 0 0 0 0 0  
## 2 0 0 27 0 0 1 2 3 0 0  
## 3 0 0 0 28 0 1 0 0 2 0  
## 4 0 0 2 1 26 1 1 0 0 2  
## 5 2 0 0 0 0 32 0 0 0 1  
## 6 3 0 0 0 1 0 29 0 0 0  
## 7 0 0 0 0 0 0 0 20 0 3  
## 8 0 0 1 4 0 2 0 1 28 0  
## 9 0 0 0 2 5 0 0 1 1 29  
##   
## Overall Statistics  
##   
## Accuracy : 0.8714   
## 95% CI : (0.8318, 0.9047)  
## No Information Rate : 0.1314   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.8567   
##   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5  
## Sensitivity 0.8889 1.0000 0.90000 0.80000 0.81250 0.86486  
## Specificity 0.9934 1.0000 0.98125 0.99048 0.97799 0.99042  
## Pos Pred Value 0.9524 1.0000 0.81818 0.90323 0.78788 0.91429  
## Neg Pred Value 0.9838 1.0000 0.99054 0.97806 0.98107 0.98413  
## Prevalence 0.1286 0.1314 0.08571 0.10000 0.09143 0.10571  
## Detection Rate 0.1143 0.1314 0.07714 0.08000 0.07429 0.09143  
## Detection Prevalence 0.1200 0.1314 0.09429 0.08857 0.09429 0.10000  
## Balanced Accuracy 0.9412 1.0000 0.94063 0.89524 0.89524 0.92764  
## Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.90625 0.76923 0.90323 0.80556  
## Specificity 0.98742 0.99074 0.97492 0.97134  
## Pos Pred Value 0.87879 0.86957 0.77778 0.76316  
## Neg Pred Value 0.99054 0.98165 0.99045 0.97756  
## Prevalence 0.09143 0.07429 0.08857 0.10286  
## Detection Rate 0.08286 0.05714 0.08000 0.08286  
## Detection Prevalence 0.09429 0.06571 0.10286 0.10857  
## Balanced Accuracy 0.94684 0.87999 0.93907 0.88845

proc.time() -ptm

## user system elapsed   
## 251.92 2.53 254.70

# Model Performance Comparison

Below I will list the different accuracies, along with their elapsed computing times (ECT).

**Decision Trees:**

* **Our Baseline. 80.29%, 5.96 ECT**
* With PCA. 75.21%, 2.7 ECT
* With Pruning. 68.57%, 1.7 ECT
* With Cross Validation. 62%, 4.6 ECT

**Naive Bayes:**

* Baseline. 50.86%, 14.9 ECT
* With Less Dimensionality Using PCA of 5. 64.43%, 1.2 ECT
* **With Less Dimensionality Using PCA of 15. 82.57%, 2.02 ECT**
* Using k-folds Cross Validation and PCA. 80.71%, 0.81 ECT

**kNN:**

* Baseline. 85%, 2.67 ECT
* **With Cross Fold Validation. 89.21%, 4.08 ECT**

**SVM:**

* Model with a Polynomial Kernel. 92%, ECT 2.49
* **Model with a Linear Kernel. 92.29%, 2.47 ECT**
* Model with Radial Kernel. 13.14%, 4.07 ECT
* Model with Sigmoid Kernel. 13.14%, 4.05 ECT
* Model with a Linear Kernel and PCA. 79.43%, 1.2 ECT

**Random Forest:**

* **Baseline. 91.71%, 19.27 ECT**
* Using optimal mtry. 91.14%, 21.82 ECT
* Using Optimal mtry and PCA. 87.14%, 1.96 ECT

# Conclusion

Setting out to conquer the famed MNIST dataset, we can confirm that machine learning can be used to recognize hand-written digits. Reaching a maximum accuracy of 92.29% using linear kernel SVM, we can now apply that to identify the remaining test data of the MNIST dataset.

While the SVM reached the highest accuracy, the Random Forest had the highest accuracy consistently, though also ate up the most computing power. This makes the SVM model the most efficient under these circumstances, though if you could afford it I would personally recommend performing an average of the two, as I think they are truly too close to judge.

Though we used different classifying algorithms, and several models each, it is obvious to see that not all algorithms performed well on the image categorization for the hand-writing samples. The Decision Tree and the Naive Bayes had the lowest overall performances, and would not be recommeneded for image recognition, though each algorithm had at least one model with above 80% recognition accuracy.