MNIST Written Character Classification

Author: Luca Di Carlo

1. Introduction

The goal of this machine learning algorithm is to be able to identify and classify hand-written images containing the numbers zero to nine from the MNIST dataset. A classification problem is when an algorithm ingests an input dataset and the output class of each piece of data in order to classify data points (Louridas and Ebert, 2016). Classification is a supervised learning technique, which infers that the learning process involves input data and output data in order to determine a function to correctly identify new input labels. This mode of machine learning requires that a dataset be separated into two separate groups: train and test data. The reason to split the data is so the computer will study the training data set, and then perform the same task on the test data set (Louridas and Ebert, 2016). A rule of thumb also states that a split of train/test data should be typically between 70/30 or 80/20, and this is to prevent excess variance in training and performance phases of learning.

The MNIST database is a collection of images representing numbers zero to nine that are handwritten. Each image has 784 pixels which are stored in the data set as numerical values, for a total of 70,000 images. The MNIST dataset is a classification problem because it contains input data that corresponds to output class labels which represent numbers zero to nine. Since the output is by class, a classification machine learning algorithm is required to develop a classification model. In order to develop a model, the MNIST data set will need to be split into a training and testing data set separately (in this case 85/15 split). Due to the size of the MNIST dataset, this translates to roughly 60,000 training data points and 10,000 testing points which is a substantial amount of data to develop the model with (Appendix A).

2. Dimensionality Reduction

Dimensionality reduction was performed to reduce the number of dimensions within the dataset. The method of choice was principal component analysis (PCA), which analyzes the multi-dimensional data and offers a reduction based on cumulative variance. Before analysis, data exploration indicated that there was no missing data, and the range of the data was 0 to 255. Data was first split into training and testing data (85/15) and scaled accordingly. Scaling and centering of the data proved no change in accuracy to the model, although it is recommended as good practice. PCA was performed first on the normalized MNIST data set and a graph of all 784 dimensions and the corresponding cumulative variance. Initial results from Figure 1 indicate 300 parameters contain about 98% of the variance (Appendix B).

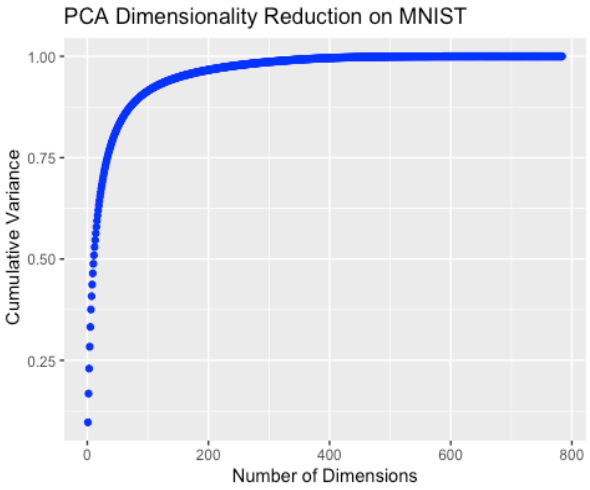

Figure 1

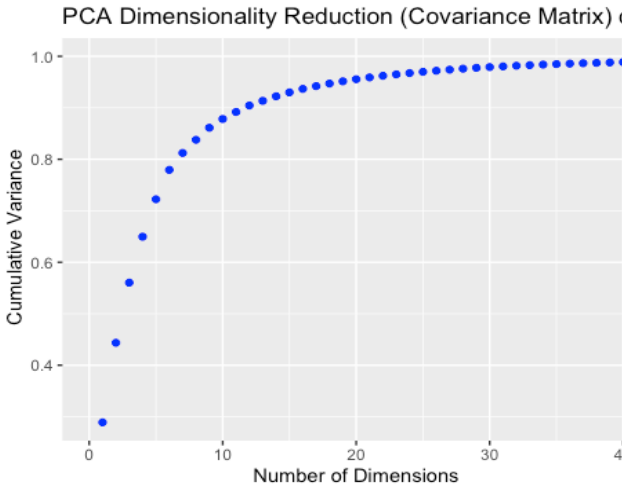

Figure 2

A second PCA was then performed using a covariance matrix, which yielded better results with only 30 parameters combining to represent 97% of the accuracy in Figure 2. The second method is a superior method due to the low number of dimensions and high cumulative variance (Appendix C).

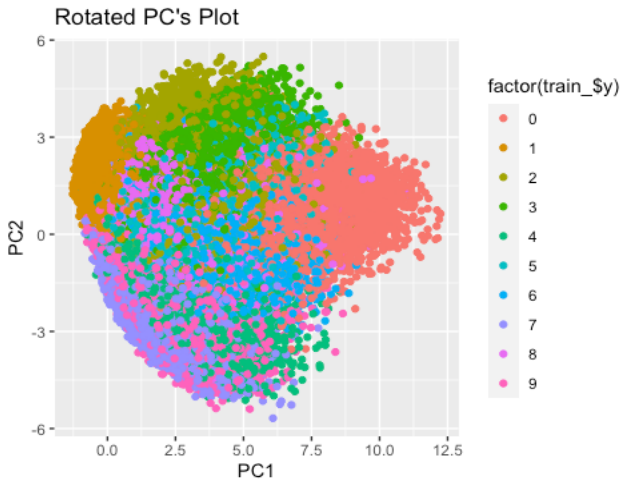

Figure 3

Figure 3 is a plot of principal component 1 and principal component 2 after rotation, with colors representing the numeric value zero to nine. Although it is not possible to see the grouping of all ten numbers due to the high dimensionality of the data, there is substantial visual grouping of the classes in this plot (Appendix D). Figure 4 reconstructs the data based on the 30 principal components that were selected after PCA (note that the data is standardized). The reconstruction was done using the first PCA results, as the second PCA method used a covariance matrix which prevented proper reconstruction of the data visually (Appendix E).

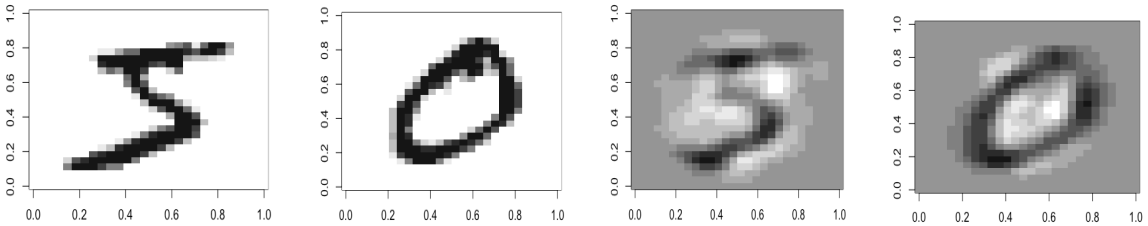

Figure 4

3. K – Nearest Neighbor (KNN)

K – Nearest Neighbor (KNN) is a classic classification algorithm which is simple and scalable, yet a powerful predictive tool to accurately model and classify data structures (Taunk *et al.*, 2019). Therefore, this algorithm was chosen to classify the MNIST data after PCA. First, the estimated value of k was estimated in Equation 1 which is 5. Figure 5 also confirms that the optimal value for k is 5, with an accuracy of 97.54% (~2.5% error) after KNN was performed. Figure 6 is a comparison of the model's predicted MNIST numbers and the actual MNIST numbers (Appendix F). Interestingly, numbers that are similar in shape were misclassified by the model the highest. For example, numbers "4" and "9" were mistaken for each other the most, followed by "7" and "1". This indicates that our model struggles to predict numbers of a similar shape with a high degree of accuracy.

$$k_{optimal\ value} = \sqrt{N} = \sqrt{20} = 4.47$$

$$k_{optimal\ value} = 5$$

Equation 1

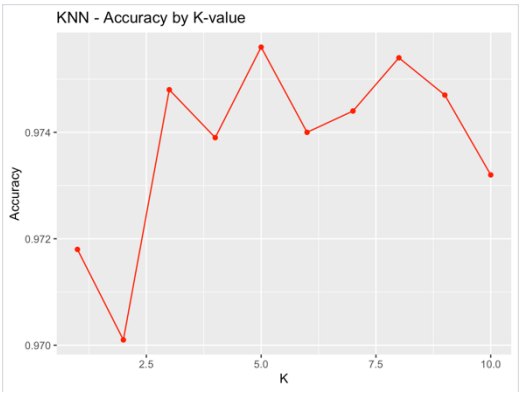

Figure 5

Actual

Predicted	0	1	2	3	4	5	6	7	8	9
0	971	0	7	0	0	3	4	0	2	3
1	1	1131	0	0	0	0	4	18	0	5
2	1	2	998	3	0	0	0	6	2	4
3	0	0	0	974	0	7	0	0	13	4
4	0	0	1	0	948	2	2	2	1	8
5	1	0	0	12	0	865	0	0	5	6
6	5	1	5	0	5	10	948	0	2	1
7	1	0	14	7	3	1	0	987	3	6
8	0	0	7	10	1	1	0	0	941	3
9	0	1	0	4	25	3	0	15	5	969

Figure 5

In order to compare the effect of PCA, KNN was run on the data set before PCA and compared with KNN after PCA (Appendix G). The chosen k value was 27, calculated by taking the square-root of the number of total dimensions (784) and rounding to the nearest odd number. The results indicated that model accuracy decrease without PCA, with lowering rising from 97.54% to 96.06% and runtime increasing from 30 seconds to 2,371 seconds.

4. Random Forest

Random forest was chosen as a second classifier for the MNIST data set due to its simplistic methodology, implementation, and effectiveness. The optimal number of trees is realized in Figure 6 as 60, which is the number at which the error term completely flattens. The random forest model was initially run using the 30 principal components from PCA and yielded an accuracy of 95.21% (Appendix H). In order to improve accuracy, the number of variables that was split at each node was optimized in Figure 7 to 4 since it generated the highest accuracy. The result yielded a slightly improved accuracy of 95.60%. Surprisingly, gradient boosting only produced a model that was 71.65% accurate and was therefore abandoned as a method (Appendix I). Figure 8 displays the error rate of the model when predicting the output class types, or numbers 0-9.

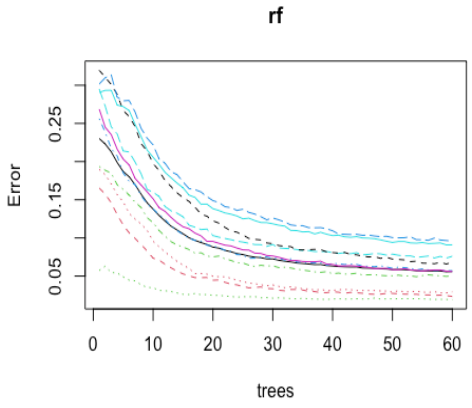

Figure 6

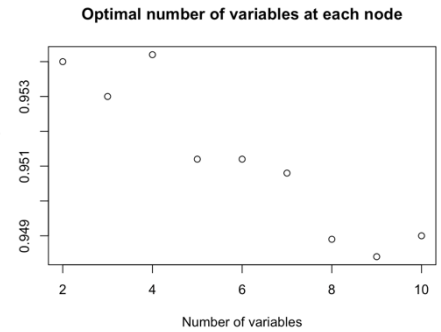

Figure 7

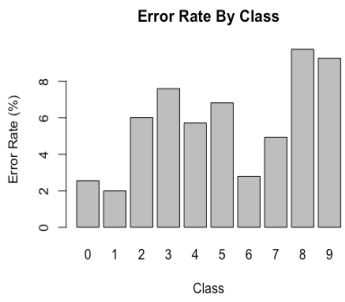

Figure 8

To test the significance of PCA on random forest, the model was run again using all 784 parameters (Appendix J). Unlike KNN, random forest yielded higher accuracy without PCA at 96.93% which was an increase of 1.3%. The increase in accuracy however included a 6-fold increase in runtime to over 1,200 seconds. Overall, PCA had a positive effect on model runtime but a negative effect on accuracy.

5. Conclusion

Using a covariance matrix for PCA proved an effective method for dimensionality reduction, with overall quicker runtime and better accuracy. Reconstructions however were not possible using the covariance matrix, therefore the principal components from the non-covariance matrix were used instead. This classification problem required the usage of two classification machine learning techniques: KNN and random forest. KNN produced the most accurate model of the two algorithms (97.5% accuracy) with an optimized number of clusters set to 5, and the quickest runtime (30 seconds). Random forest was highly accurate as well (95.6%) although it required far longer to produce results. PCA was found to improve runtimes significantly and impact accuracy slightly. KNN saw decreased accuracy from PCA, whilst the opposite occurred for random forest.

Other models of the MNIST dataset were made available by LeCun, Cortes and Burges on an online database for the MNIST dataset. KNN results on the webpage indicate similar errors to the results above (<5% error). Preprocessing the data however resulted in errors less than 2%, especially with multiple preprocessing techniques such as de-skewing, noise removal, blurring, and pixel shifting producing errors around 1% and less. The best performing algorithms overall are convolutional neural networks (CNN's), deep neural networks (DNN's), and support vector machines (SVM's), which all boasted errors for the most part less than 1%. It is evident that in each method the more preprocessing and complexity in the model, the better the performance of the model. This raises the question of what else is important besides model accuracy? The answer is computing resources, time, and model simplicity. The speed and processing power of a computer will directly impact the amount of time it takes to run a model. KNN will most definitely produce results faster than optimizing the parameters of a committee of 35 convolutional networks using width normalization. Simple models like random forest and KNN are powerful, easy to implement, and computationally resourceful.

Improvements may be made upon the models performed in this report, particularly in the preprocessing of the data. Techniques such as blurring, noise removal, and de-skewing the data has proven effective in reducing error and generating a more robust model. A current limitation of the current KNN model is its inability to differentiate between similar numbers such as "1" and "7". This is because the model is grouping the data based on each individual pixel, rather than looking at the relationships of each pixel to the other pixels in an image. A more complex algorithm would capture these relationships better, for example a neural network.

References

LeCun, Y., Cortes, C. and Burges, C. J. C. *MNIST database of handwritten digits.* Google Labs, New York (Accessed: December).

Louridas, P. and Ebert, C. (2016) 'Machine Learning', *IEEE Software,* 33(5), pp. 110-115. doi: 10.1109/MS.2016.114.

Taunk, K.*, et al.* 'A Brief Review of Nearest Neighbor Algorithm for Learning and Classification'. *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 15-17 May 2019, 1255-1260.

Appendix

A.

```
# DALT7011 ML-Assignment 1
# Student ID: 19137568
# Student Name: Luca Di Carlo

#####################################################
# Load function provided by brendan o'connor - gist.github.com/39760 - anyall.org
#####################################################
# Insert the path to the directory that holds the MNIST data files
setwd("/Users/lucadicarlo/Documents/OxfordBrookes/classes/DALT7011-ML Intro/assignments/data/")

# load function
load_mnist <- function() {
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    ret$n = readBin(f,'integer',n=1,size=4,endian='big')
    nrow = readBin(f,'integer',n=1,size=4,endian='big')
    ncol = readBin(f,'integer',n=1,size=4,endian='big')
    x = readBin(f,'integer',n=ret$n*nrow*ncol,size=1,signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }
  load_label_file <- function(filename) {
    f = file(filename,'rb')
    readBin(f,'integer',n=1,size=4,endian='big')
    n = readBin(f,'integer',n=1,size=4,endian='big')
    y = readBin(f,'integer',n=n,size=1,signed=F)
    close(f)
    y
  }
  train_ <<- load_image_file('train-images-idx3-ubyte')
  test_ <<- load_image_file('t10k-images-idx3-ubyte')

  train_$y <<- load_label_file('train-labels-idx1-ubyte')
  test_$y <<- load_label_file('t10k-labels-idx1-ubyte')
}

show_digit <- function(arr784, col=gray(12:1/12), ...) {
  image(matrix(arr784, nrow=28)[,28:1], col=col, ...)
}
```

B.

```
#####################################################
# Dimension Reduction
#####################################################
library(ggplot2)
# load data
load_mnist()

# Exploratory Analysis
summary(train_$x)
train_max <- max(train_$x)
train_min <- min(train_$x)
test_max <- max(test_$x)



# check for missing values
sum(is.na(train_$x))
## [1] 0
# standardize the data
digits_X <- train_$x/train_max
train_scaled <- as.data.frame(scale(digits_X,scale=FALSE,center=FALSE))
test_scaled <- as.data.frame(scale(test_$x/test_max,scale=FALSE,center=FALSE))


#collect the train/test labels for verification
train_labels <- factor(train_$y)
test_labels <- factor(test_$y)


############## run PCA
pca_train_ <- prcomp(as.matrix(train_scaled))
plot(pca_train_,type="lines",main="Scree Plot")# Check the variance explained by the dimensions
# create dataframe with cumulative variance and dimensions
varianceExplained <- as.data.frame(pca_train_$sdev^2/sum(pca_train_$sdev^2))
```

```r
varianceExplained = cbind(c(1:784),cumsum(varianceExplained))
colnames(varianceExplained) <- c("PCs","Var")

# create plots for variance
ggplot(varianceExplained, aes(PCs,Var)) +
  geom_point(color="blue") + labs(x= "Number of Dimensions",y= " Cumulative Variance") + ggtitle("PCA Dimensionality
Reduction on MNIST")
# 300 PC's =~ 98% of variance
```

C.

```r
############## Run PCA on  covariance matrix
digits_covMatrix <- cov(train_scaled)

#run PCA with Covariance Matrix
pca_train <- prcomp(digits_covMatrix,scale=FALSE,center=FALSE)
plot(pca_train,type="lines",main="Scree Plot (Covariance Matrix)")# Check the variance explained by the dimensions
# create dataframe with cumulative variance and dimensions
varianceExplained <- as.data.frame(pca_train$sdev^2/sum(pca_train$sdev^2))
varianceExplained = cbind(c(1:784),cumsum(varianceExplained))
colnames(varianceExplained) <- c("PCs","Var")

# create plots for variance
ggplot(varianceExplained, aes(PCs,Var)) +
  geom_point(color="blue") + labs(x= "Number of Dimensions",y= " Cumulative Variance") + ggtitle("PCA Dimensionality Red
uction (Covariance Matrix) on MNIST")ggplot(varianceExplained, aes(PCs,Var)) +
  geom_point(color="blue") + labs(x= "Number of Dimensions",y= " Cumulative Variance") + ggtitle("PCA Dimensionality Red
uction (Covariance Matrix) on MNIST Zoomed")+
  xlim(c(0,40))## Warning: Removed 744 rows containing missing values (geom_point).# 30 PC's == 97.7% of variance
######
## Use the covariance matrix since it accounts for more variance using significantly less PC's
######
```

D.

```r
# Do the dimensional reduction of the matrix to 30 columns for Covariance Matrix
rotate <- pca_train$rotation[,1:30]
trainFinal <- data.frame(as.matrix(train_scaled) %*% (rotate))
testFinal <- as.matrix(test_scaled) %*% (rotate)
trainFinal <- data.frame(trainFinal)
testFinal <- data.frame(testFinal)

ggplot(trainFinal, aes(PC1,PC2,colour=factor(train_$y))) + geom_point() +
  labs(x= "PC1",y= "PC2") + ggtitle("Rotated PC's Plot")
```

E.

```r
########## Reconstruction
xhat <- t(t(pca_train_$x[,1:30] %*% t(pca_train_$rotation[,1:30])) + pca_train$center)

# show original digit
show_digit(train_$x[1,])show_digit(train_$x[2,])

# show digit after PCA analysis
show_digit(xhat[1,])

show_digit(xhat[2,])
```

F.

```r
########################################################
# KNN
########################################################
library(class)
set.seed(123)


# run KNN for PC's with K varying from 1 through 20 with PCA parameters
knnAcc <- matrix(nrow=10,ncol=2)
i=1
ptm <- proc.time()
for (k in 1:10){
  prediction = knn(trainFinal,testFinal,train_labels,k=k)
  knnAcc[i,] <- c(k,mean(prediction==test_labels))
  i = i+1
}
elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))

## [1] "322.496 seconds elapsed"
```

```r
# convert accuracy to dataframe to be plotted
colnames(knnAcc) <- c("K","Accuracy")
knnAcc <- data.frame(knnAcc)

# plot KNN accuracy and determine the correct number for K
ggplot(knnAcc, aes(K,Accuracy)) + geom_line(color="red") + geom_point(color="red") +
  ggtitle("KNN - Accuracy by K-value")

# best prediction is at k=5
ptm <- proc.time()

best_prediction = knn(trainFinal,testFinal,train_labels,k=5)

elapsed <- proc.time() - ptm

print(paste0(elapsed[3] , " seconds elapsed"))

#calculate the accuracy for k=5
hit = 0
for (i in 1:10){
  hit = hit + table(best_prediction,test_labels)[i,i]
}
table_prediction <- table(prediction,test_labels)
accuracy = hit/sum(table_prediction)
print(accuracy)

## [1] 0.9754

library(gridExtra)
library(grid)
grid.table(table_prediction)
```

G.

```r
# Run KNN with entire dataset parameters (all 784) NO PCA
# Use k=27 since sqrt(784) = 28 and rounded to nearest odd number
ptm <- proc.time()

prediction = knn(train_scaled,test_scaled,train_labels,k=27)

# print accuracy
hit = 0
for (i in 1:10){
  hit = hit + table(prediction,test_labels)[i,i]
}
table_prediction <- table(prediction,test_labels)
accuracy = hit/sum(table_prediction)
print(accuracy)

## [1] 0.9606

elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))

## [1] "2718.268 seconds elapsed"
```

H.

```r
########################################################
# Random Forest
########################################################
library(randomForest)

library(readr)
library(caret)

set.seed(111)
numTrees <- 60

# run random forest with PCA reduced parameters
ptm <- proc.time()
rf <- randomForest(trainFinal,train_labels,ntree=numTrees)
plot(rf)

pred <- predict(rf,testFinal)
acc <- confusionMatrix(pred,test_labels)
acc$overall[1]

## Accuracy
##   0.9521
```

```
elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))
```

```
## [1] "31.8679999999999 seconds elapsed"
```

```
# Bagging
set.seed(112)
rfAcc <- matrix(nrow=9,ncol=2)
```

```
ptm <- proc.time()
r = 1
c = 1
for (i in 2:10){
  bag <- randomForest(trainFinal,train_labels,ntree=numTrees,mtry=i)
  pred <- predict(bag,testFinal)
  acc <- confusionMatrix(pred,test_labels)
  rfAcc[r,] <- c(i,acc$overall[1])


  r = r+1
  c = c+1
}
elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))
```

```
## [1] "298.131 seconds elapsed"
```

```
# determine optimal number of variables available for splitting at each tree node
plot(rfAcc,main="Optimal number of variables for splitting at each node",xlab="Number of variables",ylab="Accuracy")
```

```
# random forest for mtry = 4
optimal_bag <- randomForest(trainFinal,train_labels,ntree=300,mtry=4)
summary(optimal_bag)
```

```
pred <- predict(optimal_bag,testFinal)
acc <- confusionMatrix(pred,test_labels)
acc$overall[1]
```

```
## Accuracy
##   0.9559
```

```
# plot bag statistics
barplot(optimal_bag$err.rate[60,2:11]*100,main= "Error Rate By Class",xlab="Class",ylab = "Error Rate (%)")
```

I.

```
# Boosting
library(adabag)

ptm <- proc.time()
adaboost_df <- data.frame(trainFinal,train_labels)
ab <- boosting(train_labels~.,adaboost_df,boos=TRUE,mfinal=60)
elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))

## [1] "1066.628 seconds elapsed"

pred <- predict(ab,testFinal)
acc <- confusionMatrix(factor(pred$class),test_labels)
acc$overall[1]

## Accuracy
##   0.7165

elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))

# [1] "1068.329 seconds elapsed"
```

J.

```
# run Random Forest NO PCA
ptm <- proc.time()
# 27 chosen as sqrt of 784
rf <- randomForest(train_scaled,train_labels,ntree=numTrees)
plot(rf)
```

```r
pred <- predict(rf,test_scaled)
acc <- confusionMatrix(pred,test_labels)
acc$overall[1]
```

```
## Accuracy
##   0.9693
```

```r
elapsed <- proc.time() - ptm
print(paste0(elapsed[3] , " seconds elapsed"))
```

```
## [1] "1212.269 seconds elapsed"
```