Introduction

The MNIST dataset contains grey-scale images of digits which are hand drawn/written ranging from 0 to 9. It is a multi-Class classification as the images are classified into any one of the 0 to 9 category class Every image has a height of 28 pixels and a width of 28 pixels, for a total of 784 pixels. Every pixel is associated with unique pixel-value that represents its level of opacity; larger values correspond to darker pixels. This pixel-value is an inclusive integer ranging from 0 to 255.

Classification problem: The task is to correctly identify and classify the handwritten digits ranging from 0 to 9 by taking the pixel values into consideration which represents the visuals. These are divided into 10 classes.

For this purpose need to separate the dataset into training and test dataset so as to train the machine using the train data and test our machine on the test data which will be unknown to the machine (unseen data). This will help us evaluate the performance and accuracy of our model and to check if the model has learned meaningful patterns to predict the digit correctly.

Exploratory Data Analysis (EDA)

1. Loading the dataset

The MNIST data files are loaded in R and then converted into a specific format.

2. Visualizing first 25 cases of the dataset

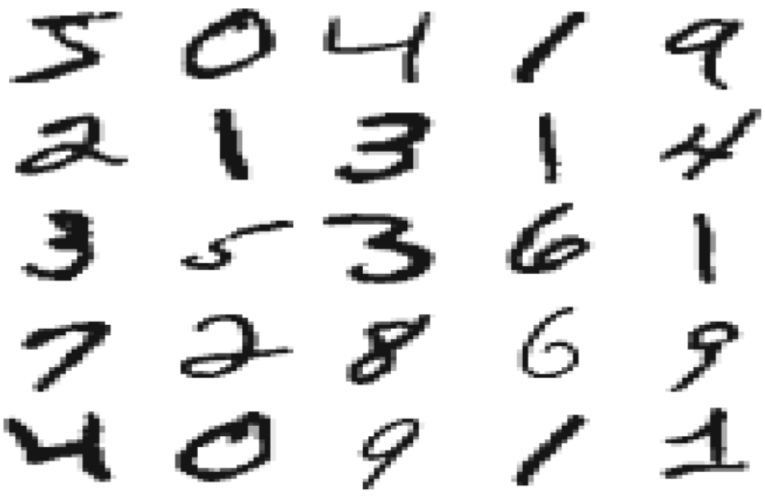
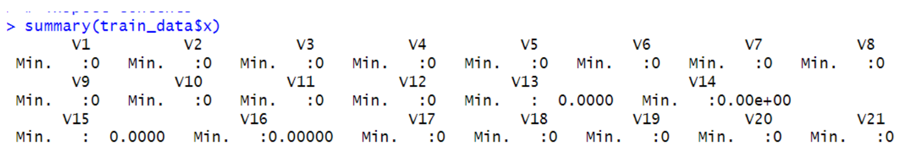


Fig 1: First 25 digits

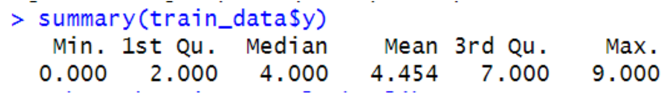
Fig 1 shows first 25 sample hand-written digits of the MNIST dataset and how they look. As seen different versions of hand written images for 0 to 9 digits.

The dataset contains 70000 rows which have been divided into training and test dataset. Training dataset contains 60000 rows and test dataset will have 10000 rows.

3. <u>Summary of train dataset x and y columns</u>



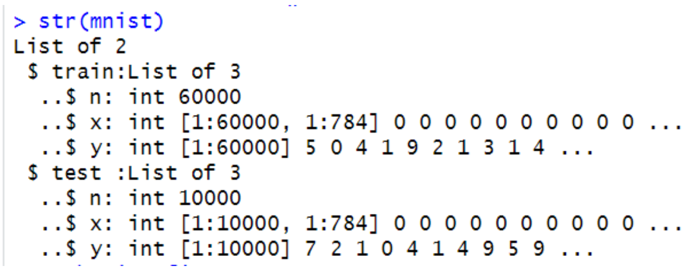
```
> summary(train_data$x)
       V1            V2             V3             V4             V5             V6             V7             V8
 Min.   :0    Min.   :0     Min.   :0     Min.   :0     Min.   :0     Min.   :0     Min.   :0     Min.   :0
       V9           V10            V11            V12            V13                           V14
 Min.   :0    Min.   :0     Min.   :0     Min.   :0     Min.   :  0.0000   Min.   :0.00e+00
      V15                 V16               V17            V18            V19            V20            V21
 Min.   :  0.0000   Min.   :0.00000    Min.   :0     Min.   :0     Min.   :0     Min.   :0     Min.   :0
```



```
> summary(train_data$y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  0.000   2.000   4.000   4.454   7.000   9.000 
```

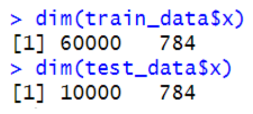
The summary shows the min, median and max values of the following numerical columns of our dataset.

4. <u>Structure of the dataset</u>



```
> str(mnist)
List of 2
 $ train:List of 3
  ..$ n: int 60000
  ..$ x: int [1:60000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ y: int [1:60000] 5 0 4 1 9 2 1 3 1 4 ...
 $ test :List of 3
  ..$ n: int 10000
  ..$ x: int [1:10000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ y: int [1:10000] 7 2 1 0 4 1 4 9 5 9 ...
```

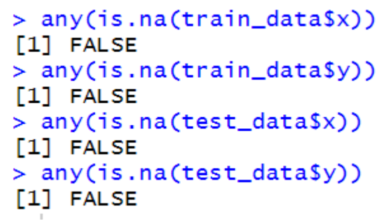
Using dim() to check if the dataset is split correctly into train and test. There are total of 60000 rows for train data into 784 dimensions and 10000 rows for test data into 784 dimensions.



```
> dim(train_data$x)
[1] 60000   784
> dim(test_data$x)
[1] 10000   784
```

Data pre-processing/cleaning

1. <u>Checking missing values.</u>



```
> any(is.na(train_data$x))
[1] FALSE
> any(is.na(train_data$y))
[1] FALSE
> any(is.na(test_data$x))
[1] FALSE
> any(is.na(test_data$y))
[1] FALSE
```

There are no missing values in the data in any of the column of our interest which says that the data is ready for further analysis.

2. <u>Multivariate Analysis</u>

The points are clustered around a line which show corelation among the variables. Some pixels are corelated very strongly while others don't corelate. We can now proceed with training of our model. For Var1 and Var5 the scatter plots show no corelation while for var2 and var3, var3 and var2 and var4 and var5 the plots show positive corelation.

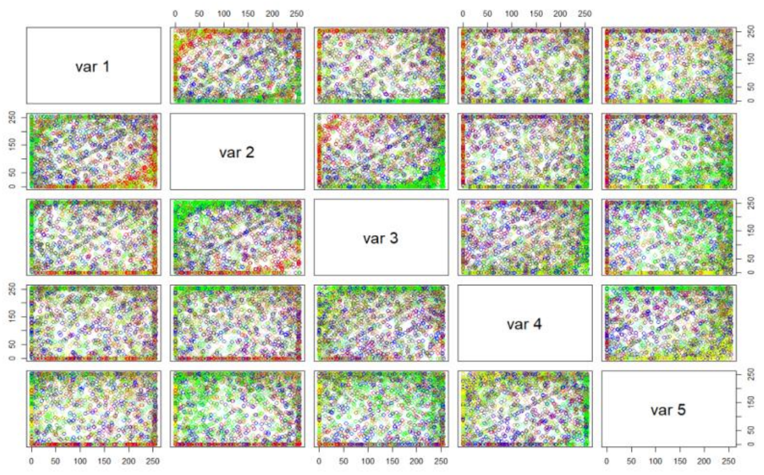
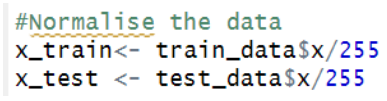


Fig 2: Corelation

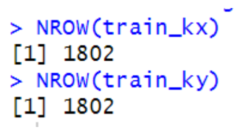
3. Normalisation of data



```
#Normalise the data
x_train<- train_data$x/255
x_test <- test_data$x/255
```

The training data is normalised by dividing it by 255. Since pixel values range from 0 to 255 in grayscale images (where 0 is black and 255 is white), dividing by 255 scales these values to a range between 0 and 1. Similarly we will normalise the pixel values in the testing data as well by dividing each value by 255. This ensures consistency between the training and testing data preprocessing steps, which is crucial for model generalization and performance evaluation.

4. Reducing the train dataset to 3% of the original train data



```
> NROW(train_kx)
[1] 1802
> NROW(train_ky)
[1] 1802
```

The dataset has been reduced to 3% of the original dataset to increasing the computational power of knn. After reducing the dataset we get 1802 rows for train dataset to run machine learning algorithm on.

5. Range of k

To find the range of k, square-root of the reduced dataset is taken which comes out as 42. Sqrt(1802) = 42.44

Hence the range of k will be 0 to 42.

K-Nearest-Neighbour Classification and Error Rate Evaluation

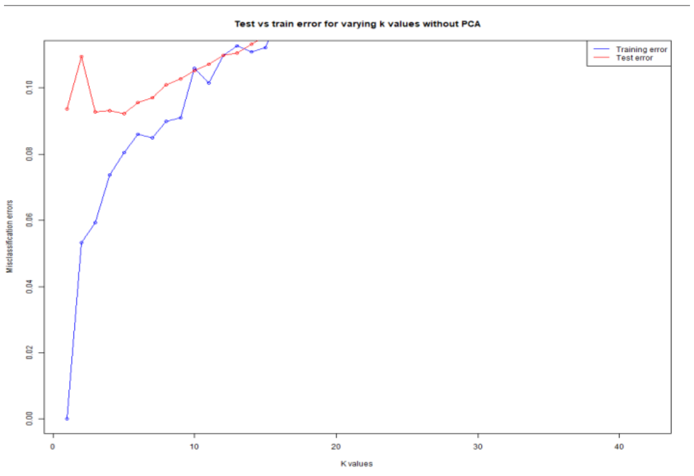
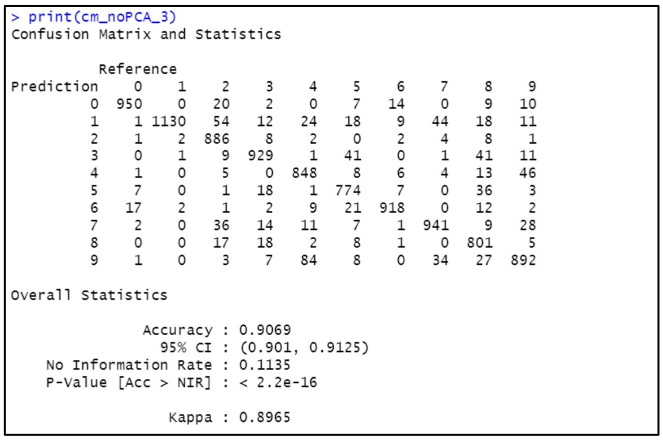


Fig 3: Error rates for different values of k

The graph shows how the KNN algorithm's performance varies with different k values in the range 0 to 42. The blue line represents the training error for different k values while the red line represents the test error for different k values. After visualizing and analysing the training and test errors for various values of k, it was concluded that K values of 3 or 5(approximately) is the optimal choice as at these values the test error is minimized and follows a stable pattern. In this case, the points 3 or 5 on the graph creates a balance between model complexity and performance, indicating that the kNN algorithm with k=3 or 5 will be well-suited for the dataset. The error rate goes on increasing after k value of 17-18 which is not seen in the graph. Odd values of k will be chosen to avoid ties.

i. KNN model when k = 3

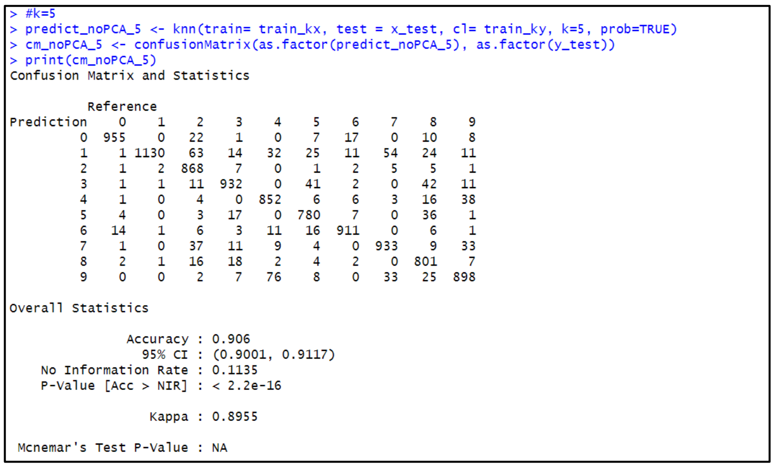


```
> print(cm_noPCA_3)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  950    0   20    2    0    7   14    0    9   10
         1    1 1130   54   12   24   18    9   44   18   11
         2    1    2  886    8    2    0    2    4    8    1
         3    0    1    9  929    1   41    0    1   41   11
         4    1    0    5    0  848    8    6    4   13   46
         5    7    0    1   18    1  774    7    0   36    3
         6   17    2    1    2    9   21  918    0   12    2
         7    2    0   36   14   11    7    1  941    9   28
         8    0    0   17   18    2    8    1    0  801    5
         9    1    0    3    7   84    8    0   34   27  892

Overall Statistics

               Accuracy : 0.9069
                 95% CI : (0.901, 0.9125)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8965
```

Fig 4: Output kNN k = 3

The accuracy of the model when k = 3 is 0.9069 which is 90.69% which indicates that the model is 90% correctly predicting the class label and hand written images based on test data.

ii. KNN model when k = 5

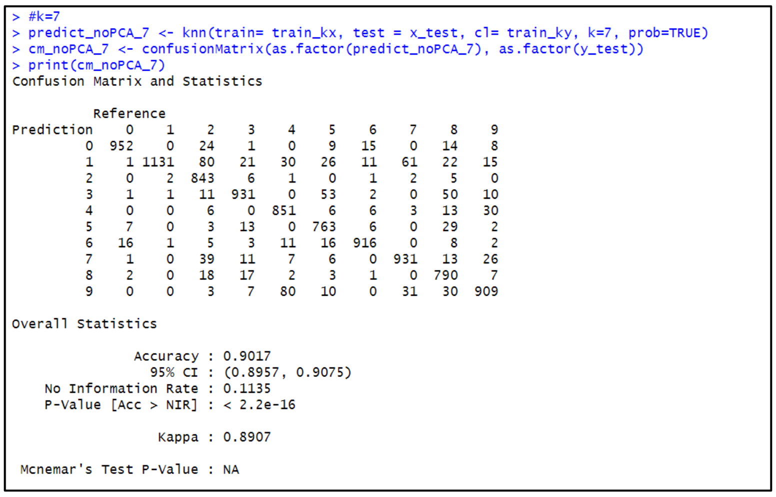


```
> #k=5
> predict_noPCA_5 <- knn(train= train_kx, test = x_test, cl= train_ky, k=5, prob=TRUE)
> cm_noPCA_5 <- confusionMatrix(as.factor(predict_noPCA_5), as.factor(y_test))
> print(cm_noPCA_5)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  955    0   22    1    0    7   17    0   10    8
         1    1 1130   63   14   32   25   11   54   24   11
         2    1    2  868    7    0    1    2    5    5    1
         3    1    1   11  932    0   41    2    0   42   11
         4    1    0    4    0  852    6    6    3   16   38
         5    4    0    3   17    0  780    7    0   36    1
         6   14    1    6    3   11   16  911    0    6    1
         7    1    0   37   11    9    4    0  933    9   33
         8    2    1   16   18    2    4    2    0  801    7
         9    0    0    2    7   76    8    0   33   25  898

Overall Statistics

               Accuracy : 0.906
                 95% CI : (0.9001, 0.9117)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8955

 Mcnemar's Test P-Value : NA
```

Fig 5: Output kNN k = 5

The overall accuracy of the model when k = 5 is 0.906 which is 90.60% which indicates that the model is correctly predicting the class label and hand written images based on test data for most of the instances.

iii. KNN model when k = 7

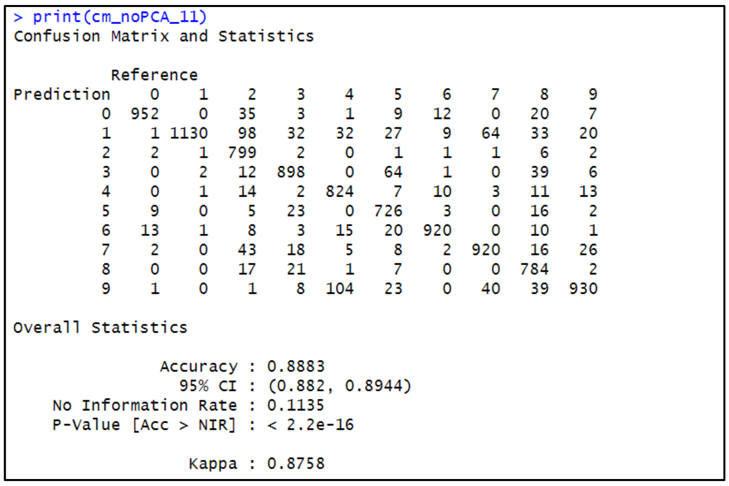


```
> #k=7
> predict_noPCA_7 <- knn(train= train_kx, test = x_test, cl= train_ky, k=7, prob=TRUE)
> cm_noPCA_7 <- confusionMatrix(as.factor(predict_noPCA_7), as.factor(y_test))
> print(cm_noPCA_7)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  952    0   24    1    0    9   15    0   14    8
         1    1 1131   80   21   30   26   11   61   22   15
         2    0    2  843    6    1    0    1    2    5    0
         3    1    1   11  931    0   53    2    0   50   10
         4    0    0    6    0  851    6    6    3   13   30
         5    7    0    3   13    0  763    6    0   29    2
         6   16    1    5    3   11   16  916    0    8    2
         7    1    0   39   11    7    6    0  931   13   26
         8    2    0   18   17    2    3    1    0  790    7
         9    0    0    3    7   80   10    0   31   30  909

Overall Statistics

               Accuracy : 0.9017
                 95% CI : (0.8957, 0.9075)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8907

 Mcnemar's Test P-Value : NA
```

Fig 6: Output kNN k = 7

With the value of k = 7, less accuracy was observed than with k = 5 and 3 with accuracy of 90.17%.

iv. KNN model when k = 11



```
> print(cm_noPCA_11)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  952    0   35    3    1    9   12    0   20    7
         1    1 1130   98   32   32   27    9   64   33   20
         2    2    1  799    2    0    1    1    1    6    2
         3    0    2   12  898    0   64    1    0   39    6
         4    0    1   14    2  824    7   10    3   11   13
         5    9    0    5   23    0  726    3    0   16    2
         6   13    1    8    3   15   20  920    0   10    1
         7    2    0   43   18    5    8    2  920   16   26
         8    0    0   17   21    1    7    0    0  784    2
         9    1    0    1    8  104   23    0   40   39  930

Overall Statistics

               Accuracy : 0.8883
                 95% CI : (0.882, 0.8944)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.8758
```

Fig 7: Output kNN k = 11

With the value of k = 11 we get accuracy of 88.83% which is the least among above. Hence, we can consider 3 as the optimal value for k for our model which has the highest accuracy.

PCA Dimensionality Reduction

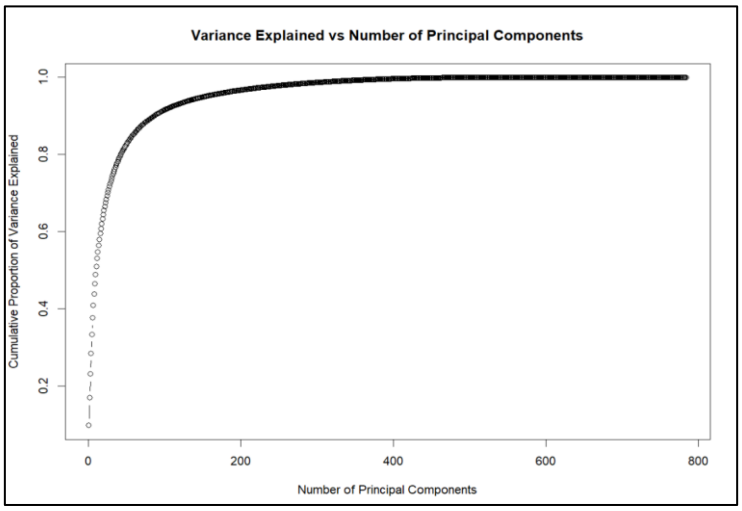
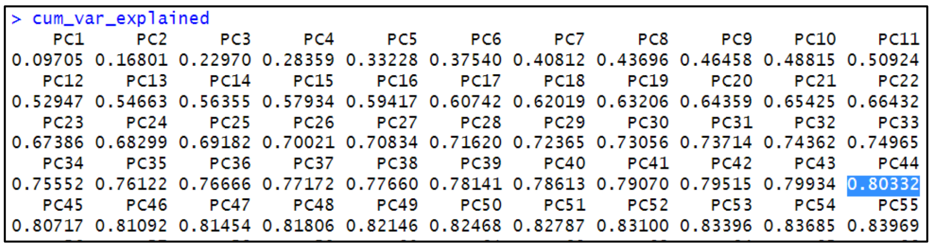


Fig 8: PCA variance explained vs PC

In Fig 8 above we have plotted the cumulative sum of variances vs the number of components. After 200 components the variance stays stable and approximately 100 components show 90% of the variance.



```
> cum_var_explained
    PC1     PC2     PC3     PC4     PC5     PC6     PC7     PC8     PC9    PC10    PC11 
0.09705 0.16801 0.22970 0.28359 0.33228 0.37540 0.40812 0.43696 0.46458 0.48815 0.50924 
   PC12    PC13    PC14    PC15    PC16    PC17    PC18    PC19    PC20    PC21    PC22 
0.52947 0.54663 0.56355 0.57934 0.59417 0.60742 0.62019 0.63206 0.64359 0.65425 0.66432 
   PC23    PC24    PC25    PC26    PC27    PC28    PC29    PC30    PC31    PC32    PC33 
0.67386 0.68299 0.69182 0.70021 0.70834 0.71620 0.72365 0.73056 0.73714 0.74362 0.74965 
   PC34    PC35    PC36    PC37    PC38    PC39    PC40    PC41    PC42    PC43    PC44 
0.75552 0.76122 0.76666 0.77172 0.77660 0.78141 0.78613 0.79070 0.79515 0.79934 0.80332 
   PC45    PC46    PC47    PC48    PC49    PC50    PC51    PC52    PC53    PC54    PC55 
0.80717 0.81092 0.81454 0.81806 0.82146 0.82468 0.82787 0.83100 0.83396 0.83685 0.83969 
```

Fig 9: Variance for PC

As seen Fig 9, the variance is mostly stable after 44 components which explain almost 80% of the variance as per Heuristics rule. This model needs at least 44 components to explain 80% of the variance.

Thus, can be implied that minimum 44 components are needed to encode the dataset and predict the digit with maximum. In this case considering 90% of the variance which is explained by 90 components, 90 components are chosen as principal component.

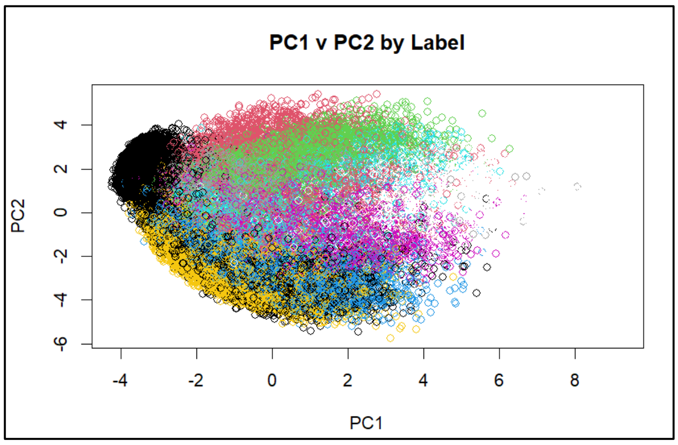


Fig 10: Scatter plot of PC1 vs PC2

The above plot Fig 10 illustrates the scatter plot of first two principal variance which shows the most variance.

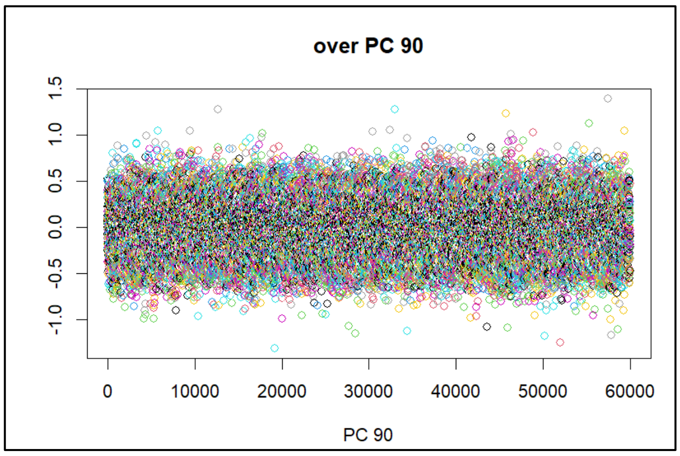


Fig 11: PC 90 scatter plot

Fig 11 shows the scatter plot of data projected onto the 90th principal component which was chosen for the analysis. The values are tightly clustered around a range of zero with most of the values approximately ranging between -0.2 to +0.2. This tells us that 90 PC shows relatively less amount of variance.

Reconstruction

The reconstruction is done using principal components 1, 90, 150 and 784.

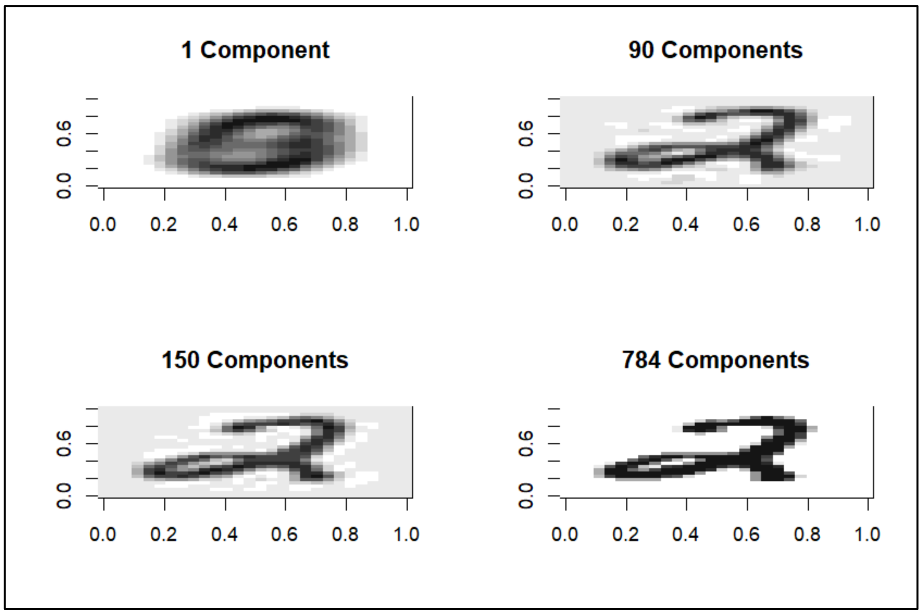


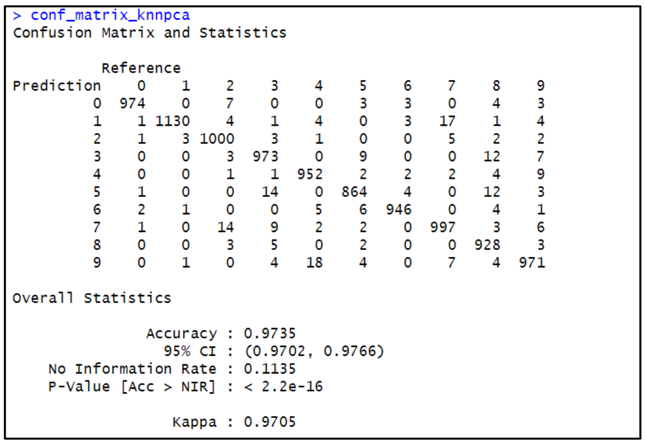
Fig 12: Reconstruction of images using different PC's

- It can be seen from the Fig 12 that as the number of principal components increase the reconstructions i.e. the digits start to appear more and more like the original image.
- As seen, that except for Component 1 all of the 2's look like 2 with 784th component as a perfect digit followed by 150th component.
- 90th component is the lowest among them which can predict the occurrence of the digit 2 as it explains 90% of the variance.

KNN with PCA

After reducing the model with PCA by selecting the number of principal components i.e. 90 in this case the KNN classification is executed on the model.

Please find the results below.



```
> conf_matrix_knnpca
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  974    0    7    0    0    3    3    0    4    3
         1    1 1130    4    1    4    0    3   17    1    4
         2    1    3 1000    3    1    0    0    5    2    2
         3    0    0    3  973    0    9    0    0   12    7
         4    0    0    1    1  952    2    2    2    4    9
         5    1    0    0   14    0  864    4    0   12    3
         6    2    1    0    0    5    6  946    0    4    1
         7    1    0   14    9    2    2    0  997    3    6
         8    0    0    3    5    0    2    0    0  928    3
         9    0    1    0    4   18    4    0    7    4  971

Overall Statistics

               Accuracy : 0.9735
                 95% CI : (0.9702, 0.9766)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9705
```

Fig 13: KNN with PCA

The output shows that the accuracy is increased as compared to kNN model without PCA. Previously the accuracy was 90.77% for k =3 on the original dataset, which is now increased to 97.35% after dimensionality reduction thus reducing the error rate from 9.23% to 2.65% which is a significant decrease.

Thus, we can say that the impact of PCA on kNN has helped the model to generate more accurate results resulting in more accuracy of the model. The PCA's impact suggests that the reduction of the dataset has been efficient without compromising the essential features of the dataset. All the impactful features were considered which led to higher accuracy of 97.35%.

Second ML Technique and Error Rate Evaluation

RANDOM FOREST

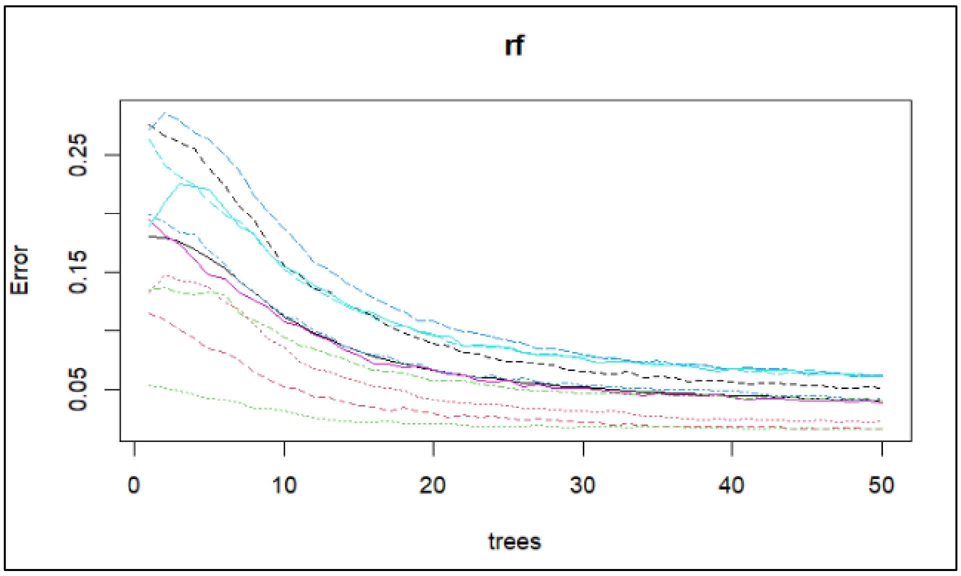
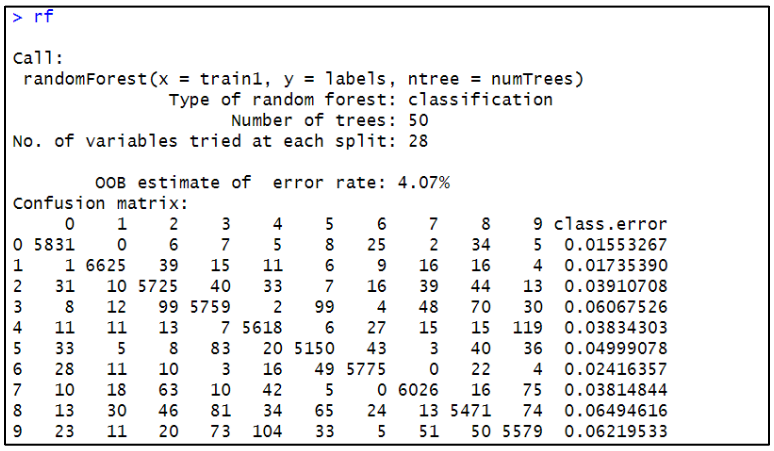


Fig 14: Trees vs Error rates

Random forest classifier has been executed for 50 trees. As seen in Fig 14, the error decreases as the number of trees increases. The error rate gradually decreases and from 40 tress onwards it becomes stable at around 0.05% which is quite low.



```
> rf

Call:
 randomForest(x = train1, y = labels, ntree = numTrees)
               Type of random forest: classification
                     Number of trees: 50
No. of variables tried at each split: 28

        OOB estimate of  error rate: 4.07%
Confusion matrix:
     0    1    2    3    4    5    6    7    8    9 class.error
0 5831    0    6    7    5    8   25    2   34    5  0.01553267
1    1 6625   39   15   11    6    9   16   16    4  0.01735390
2   31   10 5725   40   33    7   16   39   44   13  0.03910708
3    8   12   99 5759    2   99    4   48   70   30  0.06067526
4   11   11   13    7 5618    6   27   15   15  119  0.03834303
5   33    5    8   83   20 5150   43    3   40   36  0.04999078
6   28   11   10    3   16   49 5775    0   22    4  0.02416357
7   10   18   63   10   42    5    0 6026   16   75  0.03814844
8   13   30   46   81   34   65   24   13 5471   74  0.06494616
9   23   11   20   73  104   33    5   51   50 5579  0.06219533
```

Fig 14: Output random forest

The OOB estimation of error rate of 4.07% states that the model's accuracy is 95.93% (Calculated by Accuracy = 1 – Error rate). The number of trees used were 50 because by looking at the plot Fig where after 40-50 the error rates flatten. And as its Random Forest there is no overfitting with the increase in number of trees.

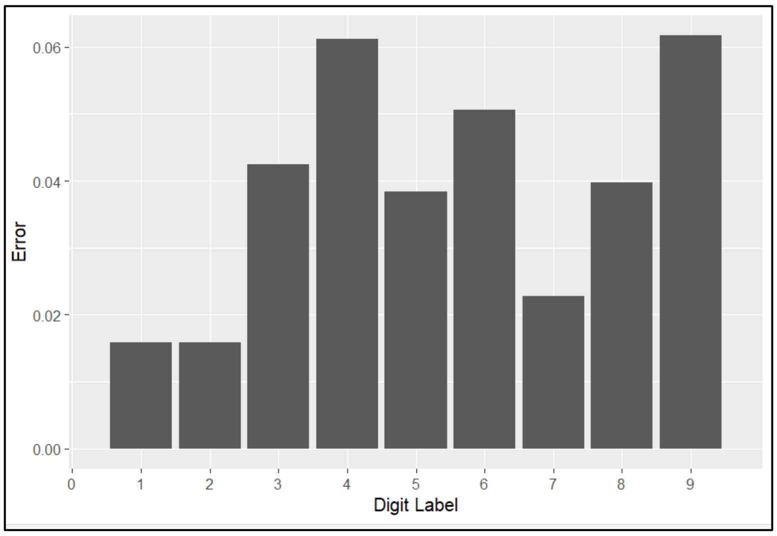
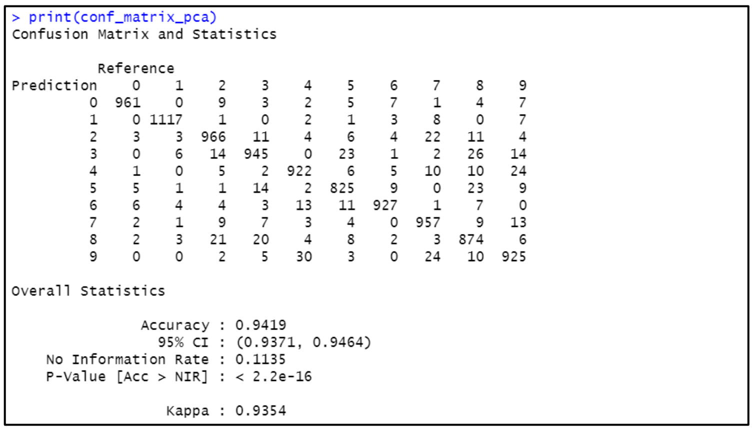


Fig 15: Error by label after 50 trees

As seen in the Fig 15 while predicting the digits 4,6 and 9 the error rate is maximum. Comparatively the error rates are minimum for digits 1 and 2.

Random Forest with PCA



```
> print(conf_matrix_pca)
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  961    0    9    3    2    5    7    1    4    7
         1    0 1117    1    0    2    1    3    8    0    7
         2    3    3  966   11    4    6    4   22   11    4
         3    0    6   14  945    0   23    1    2   26   14
         4    1    0    5    2  922    6    5   10   10   24
         5    5    1    1   14    2  825    9    0   23    9
         6    6    4    4    3   13   11  927    1    7    0
         7    2    1    9    7    3    4    0  957    9   13
         8    2    3   21   20    4    8    2    3  874    6
         9    0    0    2    5   30    3    0   24   10  925

Overall Statistics

               Accuracy : 0.9419
                 95% CI : (0.9371, 0.9464)
    No Information Rate : 0.1135
    P-Value [Acc > NIR] : < 2.2e-16

                  Kappa : 0.9354
```

Fig 16: Random Forest with PCA

After applying PCA on Random Forest the accuracy has reduced approximately by 1% which is not a major change. This might be because applying PCA reduces the dimensionality of the dataset by creating new variables (principal components) which capture most of the variance in the data. However, the components that capture the most variance aren't always the ones that are the most predictive. Important features might be lost during the dimensionality reduction process, if not enough principal components are retained. Random Forest, which is capable to capture non-linear and complex pattern possibly can perform better with the original features.

Overall Observations and findings

Model Type	PCA Used	Accuracy	Error rate
Random Forest	No	95.93%	4.07%
Random Forest	Yes	94.19%	5.81%
KNN	No	90.70%	9.30%
KNN	Yes	97.35%	2.65%

Table 1: Stats Comparison

As seen in Table 1, for KNN, PCA with KNN clearly enhance the performance, as evidenced by the improved accuracy from 0.9070 without PCA to 0.9735 with PCA.

For Random Forest, the performance degrades a bit with PCA up to 1%, and depending on the dataset it may even improve if PCA removes the unwanted features and just keep the significant features which will effectively reduce overfitting and complexity of the model.

Current Limitations and Further Strategies

As discussed, PCA might remove features that are significant useful for classification despite not being the most variant ones which leads to the loss of important information.

For both KNN and Random Forest, tuning of parameters such as the no. of neighbours (in KNN) and the number of trees (in Random Forest) is crucial.

Beyond basic random forests, techniques like boosting and stacking could provide improvements by combining multiple models for better performance.

Conclusion

Based on the above observations and results, KNN performance is enhanced by PCA by addressing its high dimensionality. For Random Forest, the impact of PCA depends more on the characteristics of the dataset and feature. When selecting an approach, it's important to consider not only the accuracy but also the training efficiency, model simplicity. Further explorations with advanced ensemble techniques and alternative dimensionality reduction methods could also help with significant enhancements.

APPENDIX

```
library(class)
library(caret)
library(factoextra)
library(tidytable)
library(ggplot2)
library(randomForest)
library(readr)


setwd("C:/Users/Ashish Khatavkar/Downloads/MNIST-data")

#code to load the MNIST digit recognition dataset into R
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_data <<- load_image_file('train-images-idx3-ubyte')
 test_data <<- load_image_file('t10k-images-idx3-ubyte')

 train_data$y <<- load_label_file('train-labels-idx1-ubyte')
 test_data$y <<- load_label_file('t10k-labels-idx1-ubyte')

 return(
  list(
   train = train_data,
   test = test_data
  )
 )
}

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

#load data
```

```
mnist <- load_mnist()
str(mnist)

#code showing first 25 cases
labels <- paste(mnist$train$y[1:25],collapse = ", ")
par(mfrow=c(5,5), mar=c(0.1,0.1,0.1,0.1))
for(i in 1:25) show_digit(mnist$train$x[i,], axes=F)

#code ot get summary of the data
summary(train_data$x)
summary(train_data$y)

#code to get dimensions of the data
dim(train_data$x)
dim(test_data$x)

#how the pictures looks like
train_data$x[1,]
show_digit(train_data$x[1,])

#having a look at the individual features (pixels)
pairs(test_data$x[,404:408],
col=c("red","green","blue","aquamarine","burlywood","darkmagenta","chartreuse","yellow"
,"chocolate","darkolivegreen")
    [test_data$y+1])

correlation_matrix <- cor(test_data$x[, 404:408])

C <- cov(train_data$x)
image(C)

#################################################################
####################

#Find the number of observation
NROW(train_data$x)
NROW(test_data$x)

#code to check if there are any missing values on the data
any(is.na(train_data$x))
any(is.na(train_data$y))
any(is.na(test_data$x))
any(is.na(test_data$y))

#Normalise the data to transforming them to a range of 0 - 1;
x_train<- train_data$x/255
x_test <- test_data$x/255
y_train <- train_data$y
y_test <- test_data$y

#code for creating partition/subset of the original training dataset to 3% of it
Find_k = createDataPartition(y_train, p=0.03, list=FALSE, times=1)
train_kx = x_train[Find_k,]
train_ky = y_train[Find_k]
```

```
#after reducing the dataset to check the no of rows
NROW(train_kx)
NROW(train_ky)

#range of k 0-42 by taking sqrt of the training no of observations
error_train_full <- replicate(0,42)

for(k in 1:42){
  predictions <-knn(train=train_kx, test=train_kx, cl=train_ky,k)
  error_train_full[k] <- 1-mean(predictions==train_ky)
}
error_train_full <- unlist(error_train_full, use.names=FALSE)

error_test_full <- replicate(0,42)

for(k in 1:42){
  predictions <- knn(train=train_kx, test=x_test, cl=train_ky, k)
  error_test_full[k] <- 1-mean(predictions==y_test)
}
error_test_full <- unlist(error_test_full, use.names=FALSE)

#plots the misclassification of errors to find the optimal value of k from range of k
png("k_values_knn_no_pca.png", height=800, width=1000)
plot(error_train_full, type="o", ylim=c(0,0.11), col="blue", xlab="K values",
ylab="Misclassification errors", main="Test vs train error for varying k values without PCA")
lines(error_test_full, type="o", col="red")
legend("topright",legend=c("Training error", "Test error"), col=c("blue","red"), lty=1:1)
dev.off()

#code knn without pca
#for k = 3
predict_noPCA_3 <- knn(train= train_kx, test = x_test, cl= train_ky, k=3, prob=TRUE)
cm_noPCA_3 <- confusionMatrix(as.factor(predict_noPCA_3), as.factor(y_test))
print(cm_noPCA_3)

#k=5
predict_noPCA_5 <- knn(train= train_kx, test = x_test, cl= train_ky, k=5, prob=TRUE)
cm_noPCA_5 <- confusionMatrix(as.factor(predict_noPCA_5), as.factor(y_test))
print(cm_noPCA_5)

#k=7
predict_noPCA_7 <- knn(train= train_kx, test = x_test, cl= train_ky, k=7, prob=TRUE)
cm_noPCA_7 <- confusionMatrix(as.factor(predict_noPCA_7), as.factor(y_test))
print(cm_noPCA_7)

#k=11
predict_noPCA_11 <- knn(train= train_kx, test = x_test, cl= train_ky, k=11, prob=TRUE)
cm_noPCA_11 <- confusionMatrix(as.factor(predict_noPCA_11), as.factor(y_test))
print(cm_noPCA_11)

#######################################PCA####################
##############
```

```
#covariance matrix
covariance_train <- cov(x_train)
str(covariance_train)
dim(covariance_train)

#code for PCA
pca_train <- prcomp(x_train, center=TRUE, scale.=FALSE)

# Visualizing the explained variance
var_explained <- summary(pca_train)$importance[2, ]
plot(var_explained, type = 'b', main = "Explained Variance by Principal Components", xlab =
"Principal Component", ylab = "Proportion of Variance Explained")

#code to determine the number of components to retain based on cumulative variance
USING Heuristics
cum_var_explained <- cumsum(var_explained)
num_components <- which(cum_var_explained >= 0.80)[1]   # at least 80% variance
explained

# Plot Variance explained vs Number of Principal Components "Scree Plot"
plot(cum_var_explained, type = "b", xlab = "Number of Principal Components",
    ylab = "Cumulative Proportion of Variance Explained",
    main = "Variance Explained vs Number of Principal Components")

str(pca_train)
View(pca_train$x)

####### Reconstruction of MNIST digits with different number of PC #############
reconstruction_1PC = t(t(pca_train$x[,1:1] %*%
                    t(pca_train$rotation[,1:1])) +
                   pca_train$center)
reconstruction_90PC = t(t(pca_train$x[,1:90] %*%
                    t(pca_train$rotation[,1:90])) +
                   pca_train$center)
reconstruction_150PC = t(t(pca_train$x[,1:150] %*%
                    t(pca_train$rotation[,1:150])) +
                   pca_train$center)
reconstruction_784PC = t(t(pca_train$x[,1:784] %*%
                    t(pca_train$rotation[,1:784])) +
                   pca_train$center)
# png("pca_reconstructions.png",height=800,width=1400)
par(mfrow=c(2,2))
show_digit(reconstruction_1PC[340,], main="1 Component")
show_digit(reconstruction_90PC[340,], main="90 Components")
show_digit(reconstruction_150PC[340,], main="150 Components")
show_digit(reconstruction_784PC[340,], main="784 Components")

################################          KNN          ON          PCA
####################################
tr_label <- y_train
plot(pca_train$x, col=tr_label, main = 'PC1 v PC2 by Label')
# Spread of PC1 only
plot(pca_train$x[,1], col = tr_label, main = 'Variance of Pixels in the Sample for PC 1', ylab =
'', xlab
= 'PC 1')
#Spread of PC2 only
plot(pca_train$x[,2],col = tr_label, main = 'over PC 2', ylab = '', xlab = 'PC 2')
#Spread of PC44 only
plot(pca_train$x[,44],col = tr_label, main = 'over PC 44', ylab = '', xlab = 'PC 44')

#Insert your number of Principal components in code below
trainFinal = as.matrix(x_train) %*% pca_train$rotation[,1:90]
head(trainFinal)

#Select number of principal components
num_components <- 90

#code to transform the data using PC 90
train_final <- as.matrix(x_train) %*% pca_train$rotation[, 1:num_components]

#code to train kNN classifier with k = 3
k <- 3
predict_pca_on_knn <- knn(train = train_final, test = x_test %*% pca_train$rotation[,
1:num_components], cl = y_train, k = k)

#code to calculate confusion matrix
conf_matrix_knnpca <- confusionMatrix(factor(predict_pca_on_knn, levels =
levels(factor(y_train))), factor(y_test))
conf_matrix_knnpca

# Plot scree plot
fviz_eig(pca_train, addlabels = TRUE)

##################second classifier


set.seed(132)

# Specify the number of training samples to use and no of trees
numTrain <- 40000
numTrees <- 50

#code to randomly select row indices to create a training dataset of the specified size
rows <- sample(1:nrow(x_train), numTrain)

labels <- as.factor(y_train)
train1 <- x_train

#code to train the Random Forest model with the specified number of trees
rf <- randomForest(train1[rows, ], labels[rows], ntree = numTrees)
plot(rf)
summary(rf)

#code to xtract the out-of-bag (OOB) error rate from the final row of the error rate matrix
oob_error_rate <- rf$err.rate[nrow(rf$err.rate), "OOB"]

#code to alculate the overall accuracy from the OOB error rate
accuracy <- 1 - oob_error_rate
```

```
#extract the error rate by class (for each digit) after 50 trees
err <- rf$err.rate
errbydigit <- data.frame(Label = 1:9, Error = err[50, 2:10])

#codelot the error by digit
errbydigitplot <- ggplot(data = errbydigit, aes(x = Label, y = Error)) +
  geom_bar(stat = "identity")

errbydigitplot + scale_x_discrete(limits = 0:9) + xlab("Digit Label") +
  ylab("Error")

#code to predict using the trained Random Forest model (using OOB samples) and capture
the predicted classes
oob_predicted <- predict(rf, type = "response")

#code to enerate the confusion matrix using the OOB predictions
conf_matrix <- confusionMatrix(as.factor(oob_predicted), labels)

print(conf_matrix)

pca_train_data <- x_train %*% pca_train$rotation[, 1:90]
pca_test_data <- x_test %*% pca_train$rotation[, 1:90]

dim(pca_train_data)
dim(pca_test_data)

#random forest with PCA

set.seed(132)
numTrainPCA <- min(40000, nrow(pca_train_data))
numTreesPCA <- 50

rowsPCA <- sample(1:nrow(pca_train_data), numTrainPCA)

#code to training the Random Forest model on PCA-transformed data
rf_pca <- randomForest(x = pca_train_data[rowsPCA, ], y = as.factor(y_train[rowsPCA]),
ntree = numTreesPCA)
plot(rf_pca)

#code to predict using the PCA-based model on PCA-transformed test data
predictions_pca <- predict(rf_pca, pca_test_data)

conf_matrix_pca <- confusionMatrix(predictions_pca, as.factor(y_test))
print(conf_matrix_pca)
```