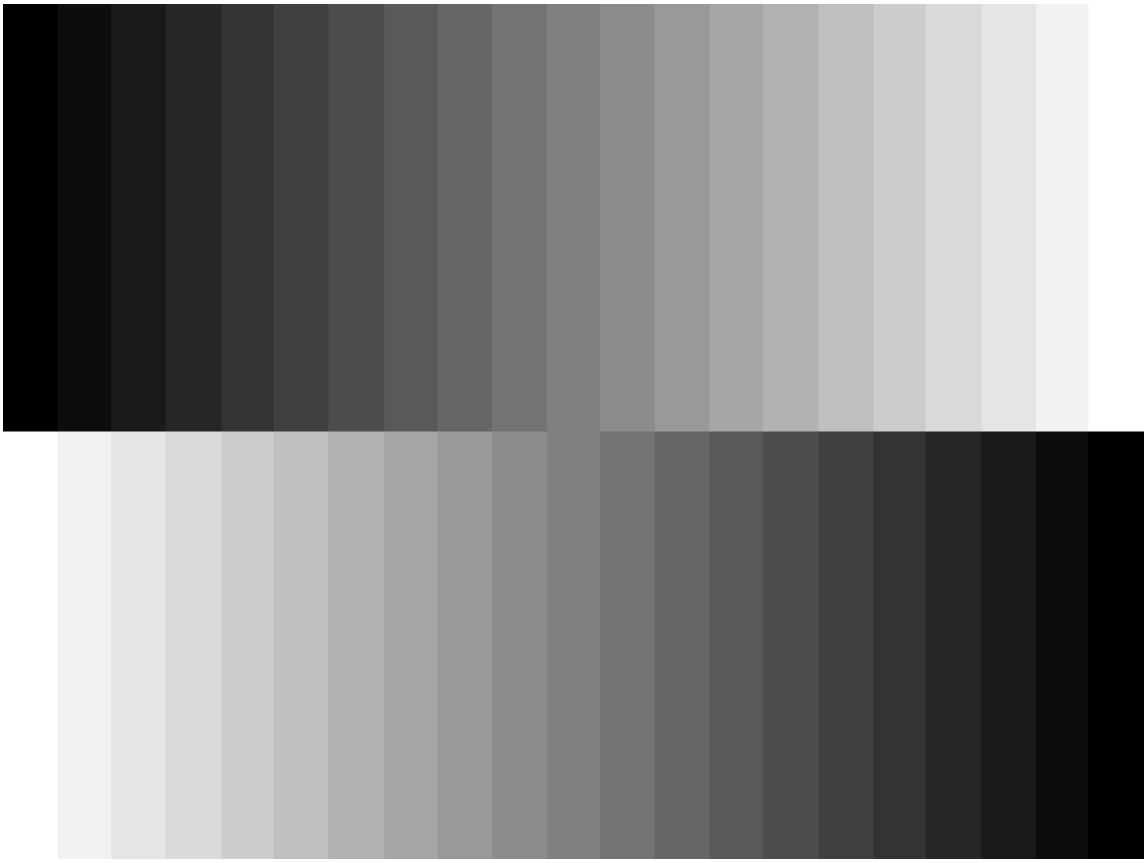


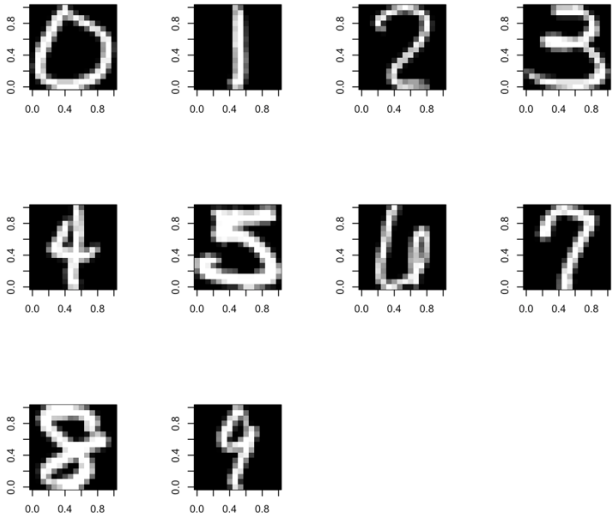
ANALYSIS OF GRAYSCALE IMAGES

Yiting Kuang | STA141A | 10/11/2019

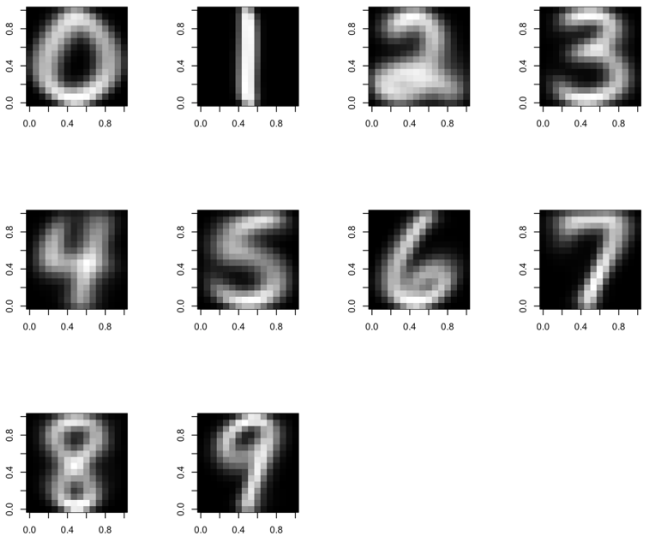
I. Introduction

In this project, I am going to examine the "digits" data set, a collection of grayscale images of scanned zip code digits. Each image shows one digit. In each file, each row is one digit image, and all the entrees in a row is the pixels of that digit image. There are 257 entrees, which forms a 16x16 matrix; the first entrée is the digit. There are 7291 observations in the training file and 2007 observations in the test file. In this report, I'll fit a model to classify handwritten digits and then examine the effectiveness of the model. Specifically, I'll implement a k-nearest neighbors' classifier and use cross-validation to estimate the error rate.

II. Explore the Digits Data



(Left: randomly selected digits)



(Right: Digits with pixels on average)

To compare the digits with varies grayscale in each pixel and the digits with pixels on average, first, I randomly selected every digit from 0 to 9 in the observations and got their images, which are left images above. Then I computed the mean of every pixels of each digit and got the right images above. As the images shown, the images with average pixels are blurrier than the images with randomly selected digits; one possible reason is that the average pixels have larger variations. However, the writing of the digits with pixels on average seem to be more standard and less cursive than the digits in the observations.

a pixel is most useful for classification if it tends to appear in one digit and not others. That is, if the variance of a pixel is large, it means it has large varying grayscale for different digits. For example, if the pixel1 varies from -1 to 1, which is from black to white, and pixel2 varies from 0 to 0.5, which is dark gray to gray; their variances are 2 and 0.5 respectively. Obviously, pixel1 is more useful for classification because it's easier to identity color black and white, and it also means that some digits appear in this pixel and some do not.

	1th	2nd	3rd	4th	5th	6th	7th	8th	9th	10th
Most Useful Pixels	230	213	219	220	120	105	121	185	204	235
Least Useful Pixels	1	241	16	256	80	17	65	96	33	49

According to the idea of identifying the most or least useful pixels based on their variance, I found the most useful and least useful pixels in the training file. The table above contains the top 10 most useful and least useful pixels.

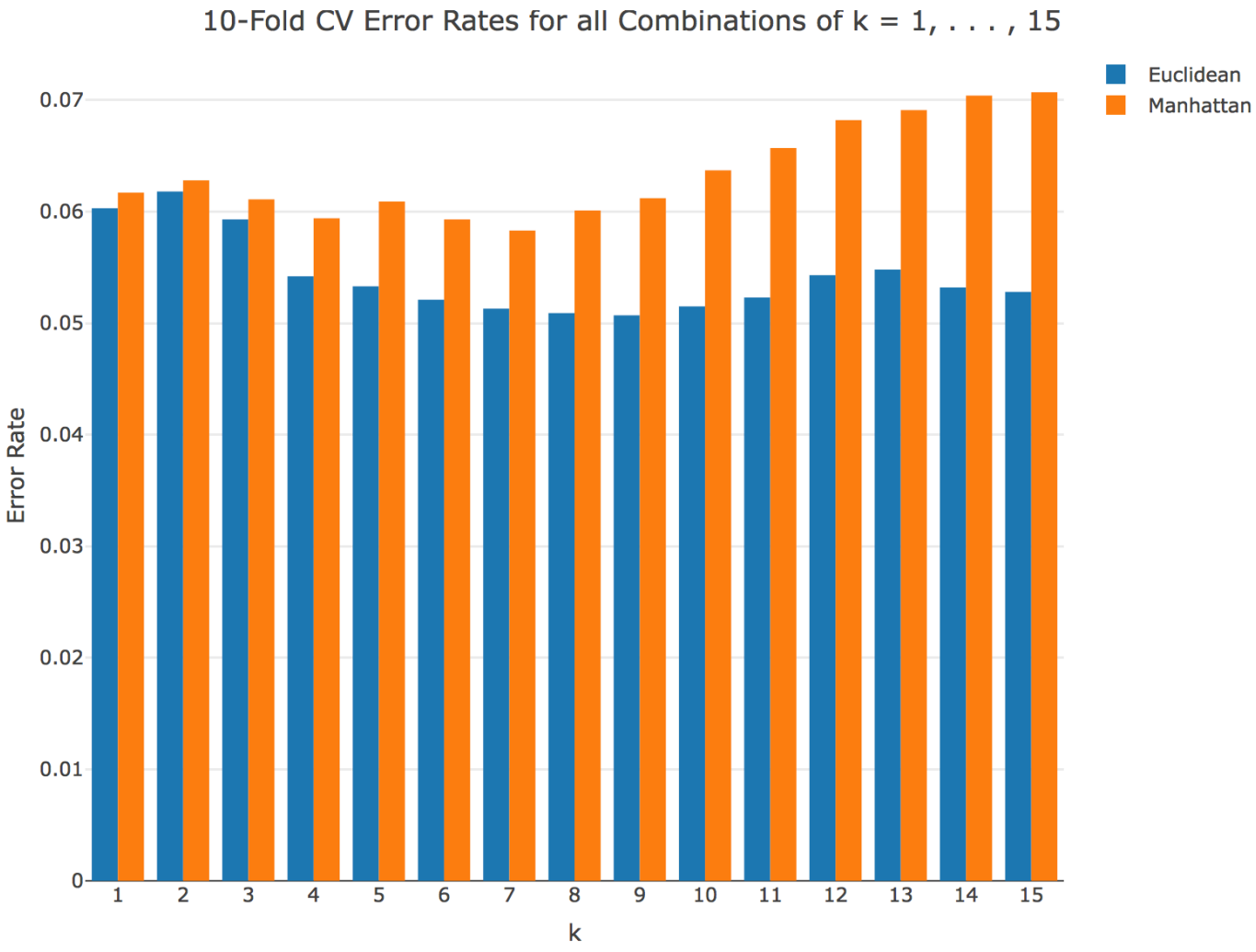
III. Cross-Validation - cv_error_knn( )

The function cv_error_knn( ) uses 10-fold cross-validation to estimate the error rate for k-nearest neighbors. Since our model is k-nearest neighbors, fitting the model would be calling k-NN function with the folds as parameters. I compared the label on the row I left out to the label the k-NN function gave me. The accuracy rate of the model is the number of labels the model predicted correctly divided by the number of labels it needs to predict for the current fold. The error rate is 1 – accuracy rate. For the cv_erroer_knn function, I looped over the folds for cross validation and looped over prediction points for k-NN. One strategy I used to increase the efficiency of my function is to order the distance matrix outside

the loop; in other words, I put the distance calculation outside the for loop, and it decreased the running time of the code. Another strategy I used, referred to the professor, is that rather than splitting entire observations into subsets for cross-validation, it is easier and more efficient to split their indexes (row numbers) into subsets.

IV. 10-Fold CV Error Rates

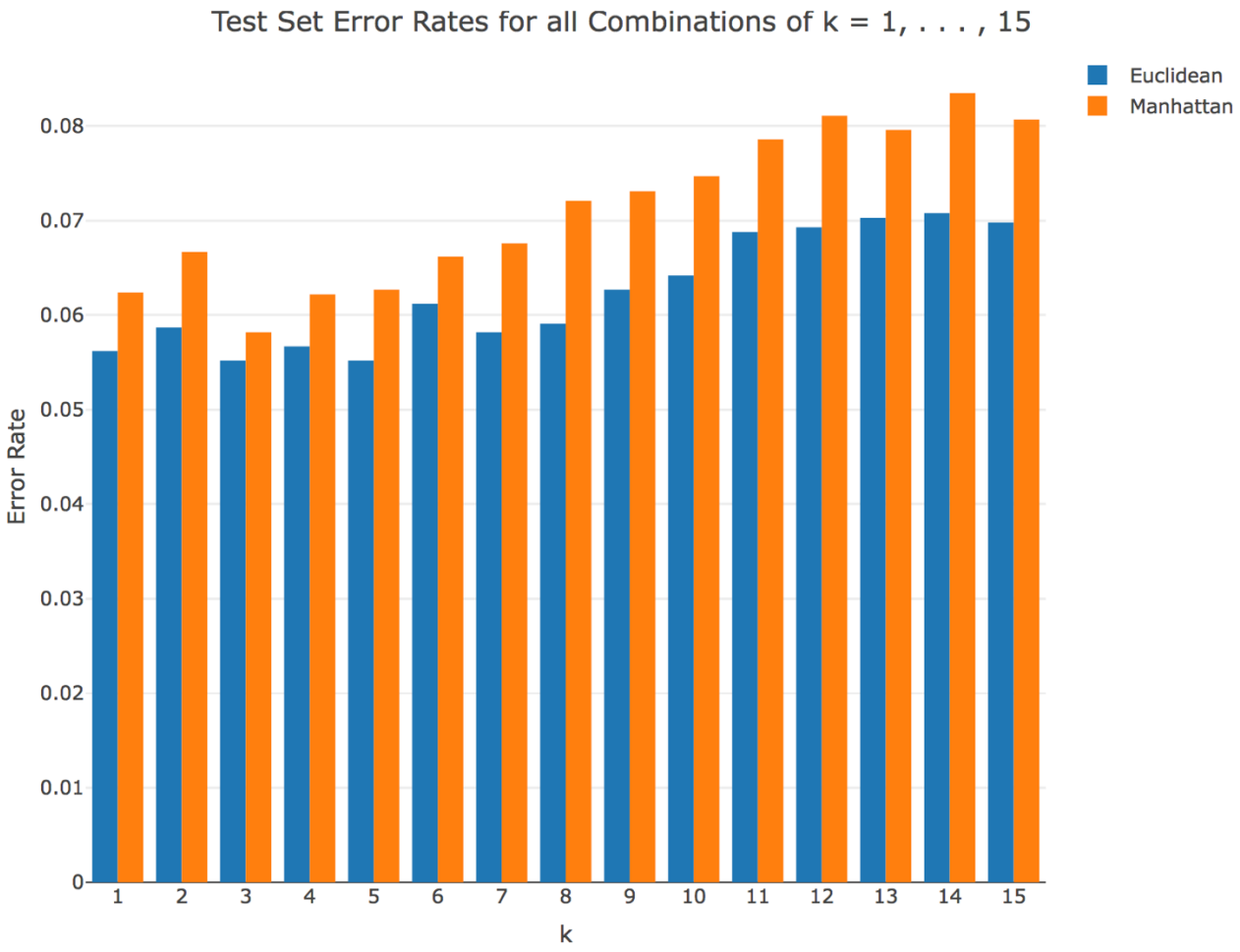
I created a bar graph comparing the 10-fold CV error rates for all combinations of k = 1, . . . , 15 using Euclidean and Manhattan distance metric method.



As the graph shown, the 10-fold CV has overall smaller error rates using Euclidean method compared to Manhattan method. When k < 5, Euclidean and Manhattan method have similar error rates; however, when k > 4, the difference between Euclidean and Manhattan in terms of 10-fold CV error rates became larger. The lowest 10-fold CV error rate with the combination of k = 9 and Euclidean method works best for the data set. It would be useful to consider additional values of the k because the trend of the graph seems to be downward, which means that the error rates tend to get smaller when k increases.

V. Test Set Error Rates

I created a bar graph comparing the test set error rates for all combinations of k = 1, . . . , 15 using Euclidean and Manhattan distance metric method.



Similar to the 10-fold CV, the test set has overall smaller error rates using Euclidean method compared to Manhattan method. When $k < 6$, Euclidean and Manhattan method have similar error rates; however, when $k > 5$, the difference between Euclidean and Manhattan in terms of the test set error rates became larger. The lowest 10-fold CV error rate with the combination of $k = 5$, which is different from the 10-fold CV, and Euclidean method also works best for the data set. Another difference between the 10-fold CV and the test set is that unlike the 10-fold CV, it would not be useful to consider additional values of the k because the trend of the graph seems to fo upward, which means that the error rates tend to get bigger when k increases.

Appendix

```
library(stringr)
library(ggplot2)
library(splus2R)
library(fBasics)
library("plotly")

## 1. Write a function read_digits() --------------------------------
file = list.files(full.names = T)

read_digits = function(file){
  # Input: A digits file (i.e., "test.txt" or "train.txt")
  # Output: the digits in the file
  txt = readLines(file, encoding = "UTF-8") # Load the file.
  df = read.table(file, header = F, sep = "") # creating a data frame
  return(df)
}
train_df = read_digits("train.txt")
test_df = read_digits("test.txt")

## 2. Explore the digits data ---------------------------------------
#Reference: https://stackoverflow.com/questions/32443250/matrix-to-image-in-r
train_matrix = as.matrix(train_df)

# Split the matrix into a list of all the properly aligned images
images = lapply(split(train_matrix, row(train_matrix)),
                function(x) t(apply(matrix(x[-1], 16, 16, byrow=T), 2, rev)))

# Plot 49 of them
img = do.call(rbind, lapply(split(images[1:49], 1:7), function(x) do.call(cbind, x)))
image(img, col=grey(seq(0,1,length=100)))

# take a close look on one digit - get each image of digits from 0-9 ramdomly
par(mfrow = c(3:4))
zero = matrix(train_matrix[9, 2:257], 16, 16, byrow = T)
zero = image(t(apply(zero, 2, rev)), col=grey(seq(0,1,length=256)))

one = matrix(train_matrix[8, 2:257], 16, 16, byrow = T)
one = image(t(apply(one, 2, rev)), col=grey(seq(0,1,length=256)))

two = matrix(train_matrix[53, 2:257], 16, 16, byrow = T)
two = image(t(apply(two, 2, rev)), col=grey(seq(0,1,length=256)))

three = matrix(train_matrix[27, 2:257], 16, 16, byrow = T)
three = image(t(apply(three, 2, rev)), col=grey(seq(0,1,length=256)))

four = matrix(train_matrix[21, 2:ncol(train_matrix)], 16, 16, byrow = T)
four = image(t(apply(four, 2, rev)), col=grey(seq(0,1,length=256)))

five = matrix(train_matrix[2, 2:257], 16, 16, byrow = T)
five = image(t(apply(five, 2, rev)), col=grey(seq(0,1,length=256)))
```

```
six = matrix(train_matrix[6, 2:257], 16, 16, byrow = T)
six = image(t(apply(six, 2, rev)), col=grey(seq(0,1,length=256)))

seven = matrix(train_matrix[4, 2:257], 16, 16, byrow = T)
seven = image(t(apply(seven, 2, rev)), col=grey(seq(0,1,length=256)))

eight = matrix(train_matrix[70, 2:257], 16, 16, byrow = T)
eight = image(t(apply(eight, 2, rev)), col=grey(seq(0,1,length=256)))

nine = matrix(train_matrix[65, 2:257], 16, 16, byrow = T)
nine = image(t(apply(nine, 2, rev)), col=grey(seq(0,1,length=256)))

# Get the average
par(mfrow = c(3:4))

all_0 = subset(train_df, V1 == 0)
avg_0 = sapply(all_0, mean)
avg_zero = matrix(avg_0[2:257], 16, 16, byrow = T)
image(t(apply(avg_zero, 2, rev)), col=grey(seq(0,1,length=256)))

all_1 = subset(train_df, V1 == 1)
avg_1 = sapply(all_1, mean)
avg_one = matrix(avg_1[2:257], 16, 16, byrow = T)
image(t(apply(avg_one, 2, rev)), col=grey(seq(0,1,length=256)))

all_2 = subset(train_df, V1 == 2)
avg_2 = sapply(all_2, mean)
avg_two = matrix(avg_2[2:257], 16, 16, byrow = T)
image(t(apply(avg_two, 2, rev)), col=grey(seq(0,1,length=256)))

all_3 = subset(train_df, V1 == 3)
avg_3 = sapply(all_3, mean)
avg_three = matrix(avg_3[2:257], 16, 16, byrow = T)
image(t(apply(avg_three, 2, rev)), col=grey(seq(0,1,length=256)))

all_4 = subset(train_df, V1 == 4)
avg_4 = sapply(all_4, mean)
avg_four = matrix(avg_4[2:257], 16, 16, byrow = T)
image(t(apply(avg_four, 2, rev)), col=grey(seq(0,1,length=256)))

all_5 = subset(train_df, V1 == 5)
avg_5 = sapply(all_5, mean)
avg_five = matrix(avg_5[2:257], 16, 16, byrow = T)
image(t(apply(avg_five, 2, rev)), col=grey(seq(0,1,length=256)))

all_6 = subset(train_df, V1 == 6)
avg_6 = sapply(all_6, mean)
avg_six = matrix(avg_6[2:257], 16, 16, byrow = T)
image(t(apply(avg_six, 2, rev)), col=grey(seq(0,1,length=256)))

all_7 = subset(train_df, V1 == 7)
avg_7 = sapply(all_7, mean)
```

```
avg_seven = matrix(avg_7[2:257], 16, 16, byrow = T)
image(t(apply(avg_seven, 2, rev)), col=grey(seq(0,1,length=256)))

all_8 = subset(train_df, V1 == 8)
avg_8 = sapply(all_8, mean)
avg_eight = matrix(avg_8[2:257], 16, 16, byrow = T)
image(t(apply(avg_eight, 2, rev)), col=grey(seq(0,1,length=256)))

all_9 = subset(train_df, V1 == 9)
avg_9 = sapply(all_9, mean)
avg_nine = matrix(avg_9[2:257], 16, 16, byrow = T)
image(t(apply(avg_nine, 2, rev)), col=grey(seq(0,1,length=256)))

# Which pixels seem the most likely to be useful for classification?
combine = cbind(avg_0, avg_1, avg_2, avg_3, avg_4,
                avg_5, avg_6, avg_7, avg_8, avg_9)
combine = combine[-1,]
var = rowVars(combine)
sort(var, decreasing = T)

## 3. Write a function predict_knn() ----------------------------------
#Reference: Haolin Li

# Compute the distance between two vectors
get_distance = function(d1, d2, dist){
  di = rbind(as.numeric(d1), as.numeric(d2))
  di = dist(di, dist)
  return(di)
}


get_label = function(prediction, train, dist_metric, k){
  #INPUT: prediction points, training points, distance metric method, and k
  train_pixels = train_df[,-1]
  predict_pixels = prediction[2:257]
  train_df[, 257] = apply(train_pixels, 1, get_distance, d2 = predict_pixels, dist = dist_metric)
  train_sort = train_df[sort.list(train_df[,257], decreasing = FALSE),]
  train_label = as.factor(c(train_sort[1:k,1]))
  train_label = names(sort(summary(train_label), decreasing=T)[1])
  train_label = as.numeric(train_label)
  return(train_label)
}


predict_knn = function(pred, train_points, distance, k){
  #OUTPUT: the index number and the label
  digits = apply(pred, 1, get_label, train = train_points, dist_metric = distance, k = k)
  return(digits)
}


# 4 10-fold CV error rates for all combinations of k = 1, . . . , 15 ----------------------
```

```
k = c("1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15")
euclidean_cv = c(0.0603, 0.0618, 0.0593, 0.0542, 0.0533, 0.0521, 0.0513, 0.0509, 0.0507,
                 0.0515, 0.0523, 0.0543, 0.0548, 0.0532, 0.0528)

manhattan_cv = c(0.0617, 0.0628, 0.0611, 0.0594, 0.0609, 0.0593, 0.0583, 0.0601, 0.0612,
                 0.0637, 0.0657, 0.0682, 0.0691, 0.0704, 0.0707)

data = data.frame(k, euclidean_cv, manhattan_cv)
data$k <- factor(data$k, levels = c("1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12",
"13", "14", "15"))
p = plot_ly(data, x = ~k, y = ~euclidean_cv, type = 'bar', name = 'Euclidean') %>%
  add_trace(y = ~manhattan_cv, name = 'Manhattan') %>%
  layout(title = "10-Fold CV Error Rates for all Combinations of k = 1, . . . , 15",
         xaxis = list(tile = "k"),
         yaxis = list(title = 'Error Rate'), barmode = 'group')


# 6 test set error rates for all combinations of k = 1, . . . , 15  ----------------------
#Reference: Haolin Li

euclidean_test = c(0.0562, 0.0587, 0.0552, 0.0567, 0.0552, 0.0612, 0.0582, 0.0591, 0.0627, 0.0642,
                 0.0688, 0.0693, 0.0703, 0.0708, 0.0698)

manhattan_test = c(0.0624, 0.0667, 0.0582, 0.0622, 0.0627, 0.0662, 0.0676, 0.0721,
                 0.0731, 0.0747, 0.0786, 0.0811, 0.0796, 0.0835, 0.0807)

data = data.frame(k, euclidean_test, manhattan_test)
data$k = factor(data$k, levels = c("1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12",
"13", "14", "15"))
p = plot_ly(data, x = ~k, y = ~euclidean_test, type = 'bar', name = 'Euclidean') %>%
  add_trace(y = ~manhattan_test, name = 'Manhattan') %>%
  layout(title = "Test Set Error Rates for all Combinations of k = 1, . . . , 15",
         xaxis = list(tile = "k"),
         yaxis = list(title = 'Error Rate'), barmode = 'group')
p
```