Sta 141a Assignment 6

Rohan Malhotra
Statistics Department
University of California, Davis
December 11, 2018

Technology, in the past years has become very advanced, and the best way to live is to use and adapt to these changes in technology. The U.S. Postal Service sorts its mail by the zip code number. In the past, it all used to be sorted by hand, but due to the advancements in technology for the past forty years they switched to an automated mail sorting system. This assignment, tests to see the accuracy of their sorting system by fitting a model to classify handwritten digits and then examine the effectiveness of the model.

1 The data and models

1.1 Data information

The data set is a collection of grayscale images of scanned zip code digits. Each image shows one digit. There are two text files, the training data and the testing data. Both were originally text files, but were read in and made into data frames to make the data easily accessible. The training data consists of 7,291 observations, whereas the testing data consists of 2007 observations and both have 257 columns.

1.2 K-nearest neighbors algorithm

This algorithm, takes in prediction points and then uses those prediction points to determine a class or label for those points. In this case, the algorithm will take in prediction points from the test data, and classifies that observation based on the class labels of the k nearest labels from the training data set. The classification power of the algorithm depends on what the user chooses as their k (how many neighbors) and the distance metric used to measure the distance between the testing point and all the training points.

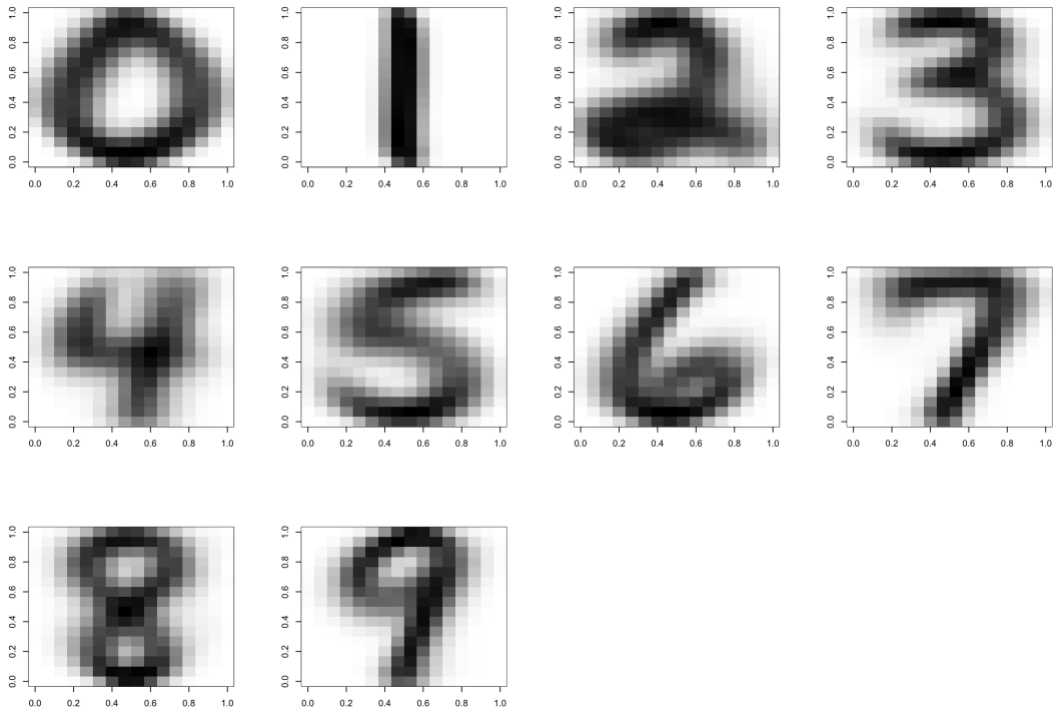
1.3 Cross-validation algorithm

This algorithm will estimate the error rate of our k-nearest neighbors algorithm by fitting the model multiple times with part of the training set left out to validate the model. So, in this case it will take the training data set and create m different partitions or folds of the data. Then the knn algorithm will be tested, by using one of the m folds as the testing data and the remaining m-1 folds will be used as the training data set, also known as the validation set. This process will repeat m times so that each subset is used once as the validation set. It will then return m error rates and average them up.

2 Images of the zip code digits

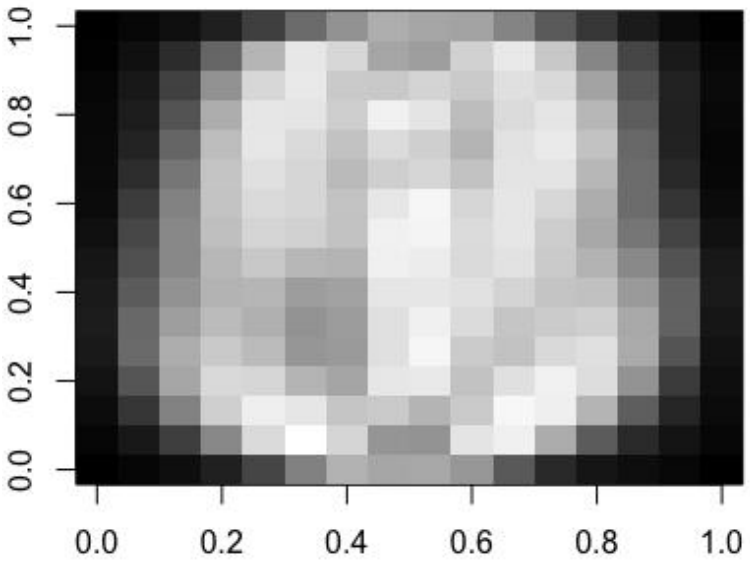
2.1 How do they look?

First, we were interested in how each image of each digit (0-9) looks. The average pixel value for all 256 pixels, of each digit (0-9) was computed. This was done both for the testing and training data sets. Since, the training data set has more observations, the images for the digits will look clearer. Below shows how each digit looks, based off the data from the training data set.



2.2 Classification of the pixels

The next point of interest was which pixels are the most useful for classification of the images above. To do this, the variance of each pixel was computed from the training data set. If the variance is low that means that those pixels are not that useful for classification as their value remains roughly the same for each digit, but if they have a high variance they are the most useful for classification because their value differs for each digit. On the next page, an image of the variance of these pixels can be displayed.



From the graphic above it is seen the pixels, that are dark on the borders mean they have low variance and pixels with high variance are in the middle of the image. So the pixels in the inside, and in the middle are the most useful pixels, whereas the pixels on the outside and towards the edges which are mainly shaded black are the least useful for classification.

3 Cross-validation algorithm design

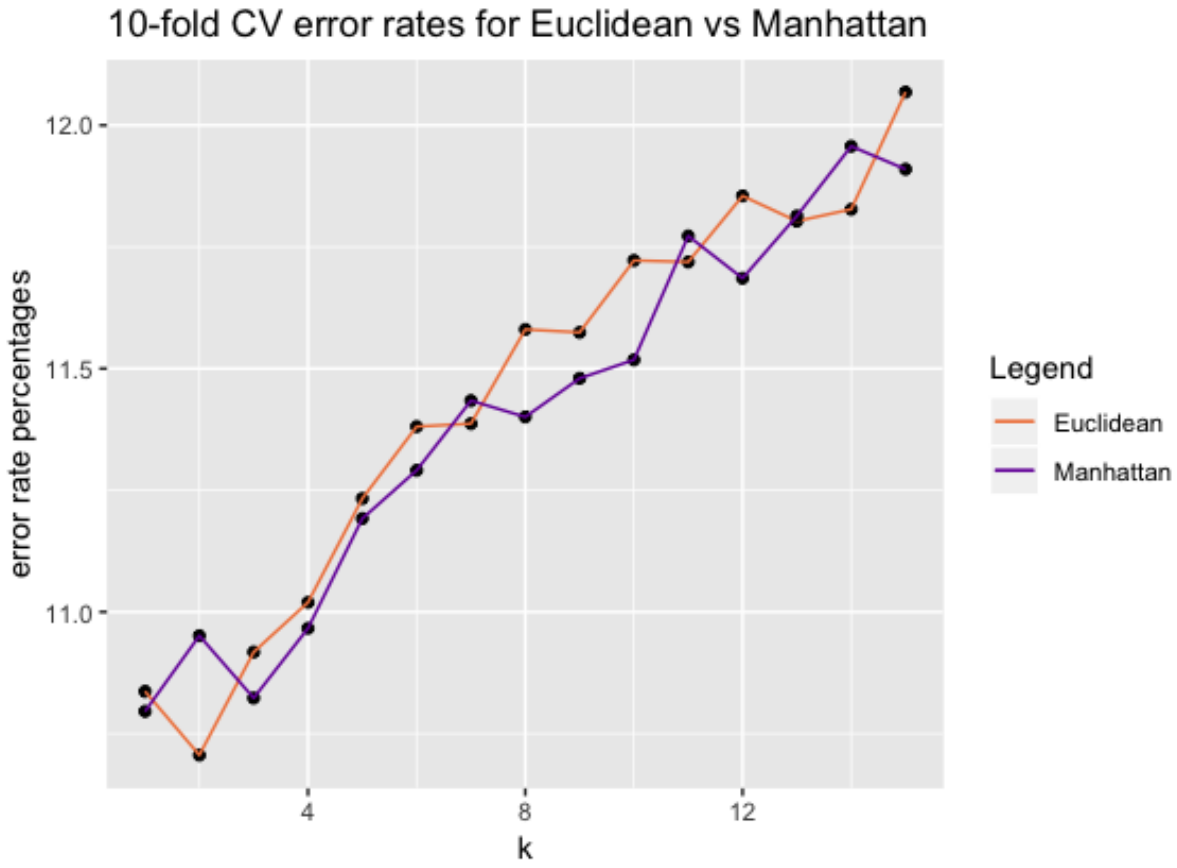
This function was made in the most efficient way possible. To make this function it was important to realize what to include inside the function itself. One of the biggest ways it was made efficient was having the distance matrix calculated outside the function instead of the inside as it significantly reduces the run time for the function. Also, instead of using a loop, a lapply function was used to compute the error rate for each of the m folds, which also helps with the efficiency of the code. Finally, this code also runs efficiently because the predict knn algorithm, which is used in the cross-validation algorithm, was made efficiently by also calculating the distance matrix outside the function and using alternatives to looping. Both functions run swiftly and correctly. When testing this cross-validation algorithm for 10 folds, the average error rate returned is 10% which is perfect because that means our knn algorithm has an accuracy of 90%.

4 Comparison between different distance metric

As mentioned, before the power of the knn algorithm depends on the distance metric to compute the distance matrix. Therefore, I will compare the Euclidean and the Manhattan distance metrics.

4.1 Comparison of the cross-validation error rates

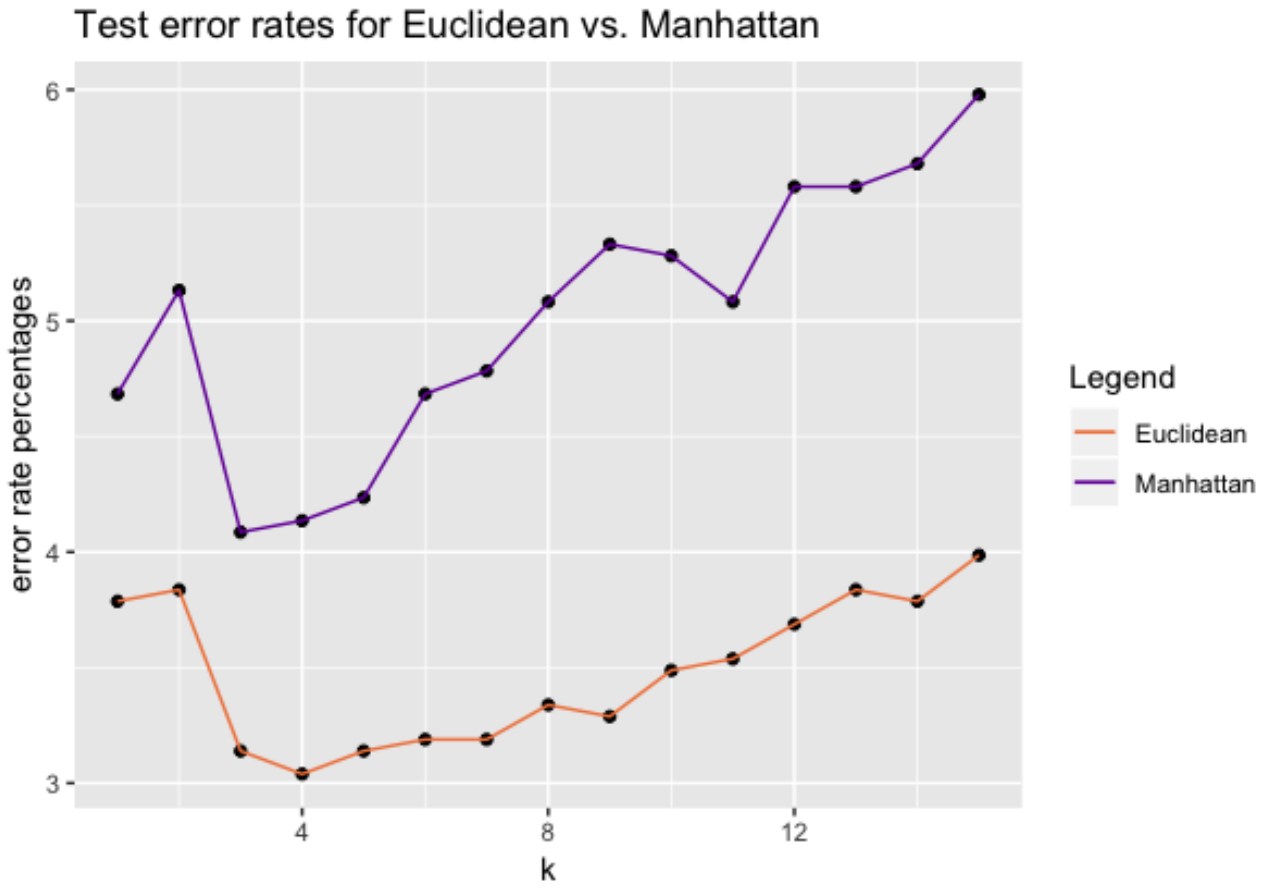
First, let's take compare the cross-validation error rates between the Euclidean distance metric and the Manhattan distance metric. For both cases, 10 folds will be used and will be examined for "k"s: 1-15. The figure below represents this relationship.



Looking at the graph above it is clear that the lowest error rate is seen when k is 2 and the Euclidean distance metric is used. Looking at additional k's will not be too useful because overall there is no clear relationship on which method is better as one may be better for a specific k and also increasing k will lead to more bias as more labels will be considered, this is often regarded as the bias-variance tradeoff. Also, for "k"s 1 through 15 it is seen that as k increases so does the error rate, which may suggest there's some linear relationship among the "k"s and the cross validation error rates.

4.2 Comparison of the test error rates

Now let's compare the test error rates between the Euclidean and Manhattan distance metric systems. Again only "k"s 1-15 are being observed. The figure is depicted below.



It seen from the figure above, when comparing test error rates the best distance metric is Euclidean. Also looking at the comparison between test error rates and cross validation error rates it is seen that the test error rates are significantly lower than the cross validation error rates. This is possibly because the test error rates consider only one test case. Finally, the most common digits that best model, meaning model using the Euclidean distance metric, for "k" 1-15 were 8, 3, and 2, which can be seen from the table below.

Digit	0	1	2	3	4	5	6	7	8	9
Count of wrong predictions	45	48	191	167	128	115	65	81	138	84

Code and Sources:

```
# Libraries --------
library(ggplot2)
install.packages("viridis")
library(viridis)
library(nnet)
library(hexbin)
library(ggrepel)
# 1 --------------

# Read digits will take in a text file, in our case the user can input their file path of the test and
training files.
# The function will then make a dataframe and convert everything to a double
test_dir = "~/Desktop/Sta 141a Assignment 6/digits/test.txt"
train_dir = "~/Desktop/Sta 141a Assignment 6/digits/train.txt"

read_digits = function (file){
  table = read.table(file, header = FALSE, sep = "")
  table = apply(table,2,as.numeric)
  table = as.data.frame(table)
  return(table)
}

# Get the train and testing data
test_dat = read_digits(test_dir)
train_dat = read_digits(train_dir)

# Inspect the data sets
dim(test_dat) #  2007 rows, 257 columns
dim(train_dat) #  7291 rows, 257 columns

# 2 ----------------------------------

# Get the average number of each pixel for each number (0 to 9) of both data sets
pixel_by_number_training = aggregate(.~ V1, train_dat, mean)
pixel_by_number_training = pixel_by_number_training[,-1]

pixel_by_number_test = aggregate(.~ V1, test_dat, mean)
pixel_by_number_test = pixel_by_number_test[,-1]

# Function will make a 3 by 4 grid then it will produce an image of how each number graphically
looks and add it to that grid
get_image = function(dataset){
  par(mfrow = c(3,4)) # Reference: Piazza Post 582
```

```
  for (num in 1:10){
    my_num = as.numeric(dataset[num,])
    num_matrix = matrix(my_num, nrow = 16, ncol = 16)
    num_matrix = t(apply(num_matrix, 1 ,rev)) # Reference: Stack Overflow post on how to fix
the numbers coming upside down (Full link at the end)
    image(num_matrix,col = grey(seq(1,0, length = 256)))
  }
}

# Display graphically how each number will look, in both the training and testing data set
get_image(pixel_by_number_training)
get_image(pixel_by_number_test)

# Calculate the variance of each pixel and display graphically the variances of the pixels
graphics.off()
train2 = train_dat[,-1]
pixel_variance = as.matrix(apply(train2, 2, var))
var_mat = matrix(pixel_variance[,1:ncol(pixel_variance)],16,16)
image(t(apply(var_mat,1,rev)), col=grey(seq(0,1,length=256)))

# 3 ---------------------------------

# First get the distance matrix
combined_data = rbind(test_dat,train_dat)
distance_matrix = as.matrix(dist(combined_data))

# Function will take in the testing data, training data, number of neighbors and the distance
matrix calculated above
# It will then return the label for each prediction point.
predict_knn = function(test,train, k, dm) {
  combined_df = rbind(test,train)
  labels = combined_df[,1] # Have to use labels of the combined data so the rows match, once we
subset the matrix
  test_row = nrow(test)
  train_row = nrow(train)
  sub_dm = dm[1:test_row,(1+test_row):(test_row+train_row),drop = FALSE] # Subset the
matrix to only get comparision between testing and training points

  # The following below will order the subsetted matrix. That ordered matrix will give us the
column the minimum distance is in
  # for each prediction point. Since those columns correspond to the rows of the combined data
frame, which we then use to find the label
  prediction_labels = unlist(apply(sub_dm,1,function(x){
    column_index = order(x)[1:k] + test_row
    prediction_labels_freq = table(labels[column_index])
```

```
    winning_position = which.is.max(prediction_labels_freq) # Will settle a tie-breaker between
labels, reference: rdocumentation
    winner = as.numeric(names(prediction_labels_freq[winning_position]))
    return(winner)
  }))
  return(unname(prediction_labels)) # Will return the prediction labels for each point in the
training data
}
x = predict_knn(test_dat,train_dat,4,distance_matrix)

# 4 ------------------------

# First step is to obtain a distance matrix of just the training data, but randomize the rows before
finding the distance matrix to remove bias.
randomized_rows = sample(nrow(train_dat))
training_distance_matrix = as.matrix(dist(train_dat[randomized_rows,]))

# Function will make m folds compare the folds to eachother, and return m error rates and then
will calculate average error rate
cv_error_knn = function(m, train, model, k, dm){

  # creates m folds each with randomized row numbers of training data set
  df_shuffled = train[randomized_rows,]
  m_folds = split(randomized_rows, 1:m)

  # Will take 1 fold and find prediction labels, by looking through the other m - 1 folds. Then it
will compute how many predictions were right and wrong
  # and return an error rate
  m_error_rates = lapply(1:m, function(x){
    fold_labels = as.numeric(model(train = df_shuffled[-m_folds[[x]],], test =
df_shuffled[m_folds[[x]],], k, dm)) # Reference: Patrick's Office Hours
    num_true = table(fold_labels == df_shuffled[-m_folds[[x]],][,1])[2]
    total_num = sum(table(fold_labels == df_shuffled[-m_folds[[x]],][,1]))
    error_rate = num_true/total_num
  })
  # Gets the average error rate of m folds
  all_errors = unlist(m_error_rates)
  avg_error_rate = sum(all_errors) / m
  return(avg_error_rate)
}
y = cv_error_knn(10,train_dat,predict_knn,4,training_distance_matrix)

# 5 ---------------------

# Get two distance maticies one for each distance metric, euclidean and minkowski
euclidean_matrix_random = as.matrix(dist(train_dat[randomized_rows,]))
```

```
manhattan_matrix_random =  as.matrix(dist(train_dat[randomized_rows,],method =
"manhattan"))

# Calculate the 10 fold cv for k: 1-15  for both matricies
euclidean_cv = sapply(1:15, function(x){
  cv_error_knn(10,train_dat,predict_knn,x,euclidean_matrix_random)
})

manhattan_cv = sapply(1:15, function(x){
  cv_error_knn(10,train_dat,predict_knn,x,manhattan_matrix_random)
})

# Plot the errors
manhattan_cv = as.data.frame(manhattan_cv)
manhattan_cv$k = c(1:15)
euclidean_cv = as.data.frame(euclidean_cv)
euclidean_cv$k = c(1:15)
final_dat = cbind(manhattan_cv,euclidean_cv)
final_dat = final_dat[,-4]
colnames(final_dat) = c("manhattan","k","euclidean")

ggplot(final_dat) + geom_point(aes(x = k, y = 100*euclidean)) + geom_line(aes(x = k, y =
100*euclidean, color = "Euclidean")) +
  geom_point(aes(x = k, y = 100*manhattan)) + geom_line(aes(x = k, y = 100*manhattan,color =
"Manhattan")) +
  scale_color_viridis(discrete = TRUE,option = "plasma",begin = 0.7,end = 0.23,name =
"Legend") +
  labs(title = "10-fold CV error rates for Euclidean vs Manhattan", y = "error rate percentages")
# Best model: K = 2, Euclidean distance metric

# 6 ----------------------

# Get the matrices for both euclidean and manhattan distance metrics
euclidean_matrix_unrandom = as.matrix(dist(combined_data))
manhattan_matrix_unrandom =  as.matrix(dist(combined_data,method = "manhattan"))

# Compute the error rates
euclidean_error_rate = sapply(1:15, function(x){
  eu_labels = predict_knn(test_dat,train_dat,x,euclidean_matrix_unrandom)
  wrong = table(eu_labels == test_dat[,1]) [1]
  total = sum(table(eu_labels == test_dat[,1]))
  error_rate = wrong/total
})

manhattan_error_rate = sapply(1:15, function(x){
  eu_labels = predict_knn(test_dat,train_dat,x,manhattan_matrix_unrandom)
```

```
  wrong = table(eu_labels == test_dat[,1]) [1]
  total = sum(table(eu_labels == test_dat[,1]))
  error_rate = wrong/total
})

# Convert to Data frame to graph
df_euclidean = as.data.frame(euclidean_error_rate)
df_manhattan = as.data.frame(manhattan_error_rate)
final_dat_2 = cbind(df_manhattan,df_euclidean)
final_dat_2$k = c(1:15)
colnames(final_dat_2) = c("manhattan","euclidean","k")

ggplot(final_dat_2) + geom_point(aes(x = k, y = 100*euclidean)) + geom_line(aes(x = k, y =
100*euclidean, color = "Euclidean")) +
  geom_point(aes(x = k, y = 100*manhattan)) + geom_line(aes(x = k, y = 100*manhattan,color =
"Manhattan")) +
  scale_color_viridis(discrete = TRUE,option = "plasma",begin = 0.7,end = 0.23,name =
"Legend") +
  labs(title = "Test error rates for Euclidean vs. Manhattan", y = "error rate percentages")

# Finding the most common digits predicted wrong from Euclidean distance metric
sort(table(unlist(sapply(1:15, function(x){
  predictions = predict_knn(test_dat,train_dat,x,euclidean_matrix_unrandom)
  wrong = unlist(test_dat$V1[predictions != test_dat$V1])
})))


# References ----------

# Worked with Jonathan Fernandez, Kelly Chan, Jiemin Huang, Fenglan Jiang
# https://stackoverflow.com/questions/32443250/matrix-to-image-in-
r?fbclid=IwAR2dymyb_gLFrOxD-cqvsJhIG_NzJ0F58ZOYANoJaPrvVXdz3kxfIWzJcTA
# https://www.rdocumentation.org/packages/nnet/versions/7.3-9/topics/which.is.max
```