STA 141A

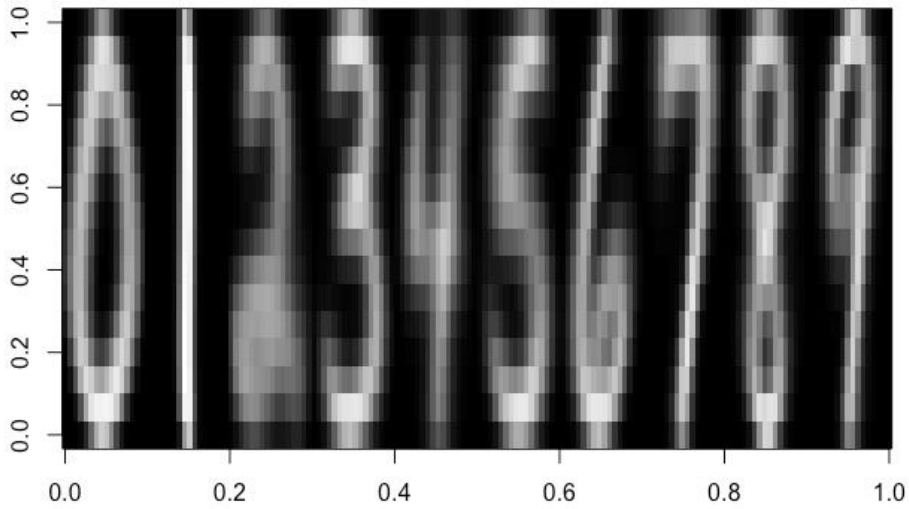
Franco Gonzales
914953764

Final Project (6)
12/12/2018

2) Explore the digits data:

• Display graphically what each digit (0 through 9) looks like on average.

After some manipulation to the data, I ended up being able to calculate the means of every column for the class. In the case of this project, I'm referring to the numbers as class due to most KNN models seeming to do so as well. Anyways, if we take the means and then plot the images side by side we get the following:



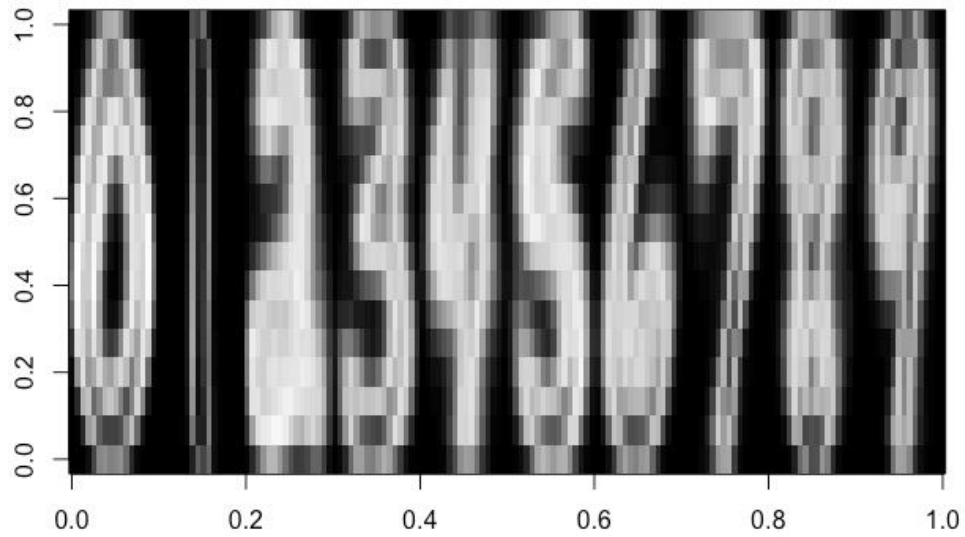
While it might not seem like much, there are some things to take note of here. First, the most obvious would be that the class "1" has the sharpest image out of all the other numbers. Looking at it logistically, this would does make sense as it's literally one straight line up and down. The other classes though are blurry, and my guess is that single images pulled from our set would have a more distinct image.

After looking at it from this perspective, now it's easier to visualize how a KNN algorithm would be applied to our data. The fuzziness in the images can be caused for a lot of reasons, but my educated (but actually really not) guess is due to the fact that we took the mean value for all the pixels. Since each class has essentially their own unique values for the pixels (columns), taking the means would just lead to a generalization of the images.

• Which pixels seem the most likely to be useful for classification? Which pixels seem the least likely to be useful for classification? Why?

So to see which pixels would be most useful, I decided to find the variance for each column per class. So looking at the pixels in its purest form, in the data frame, it was difficult to make an

inference with just what was given to me. There are 256 columns per class, so going through each column would simply take too much time. Thankfully, taking images of the variances is an option.



If you take the plots and put them side by side, you'll notice that the means and variances seem to be inverses of each other. On to the analysis, it seems that the "darker" pixels in this set would be the most useful for classification. Sifting through the data provided, it seems that those pixels can be associated with the absolutes (-1 and 1). If we were to test a pixel, those with low variances can be "anchors" for those classes that contain them. For example, 1. With it being the simplest out of all the other classes, there isn't that much of a difference if at all with some of the pixels. With how KNN calculates differences between points, if a class has low variance in specific pixels, we'll only get higher accuracy in our predictions. So in contrast, those with high variances wouldn't be very useful in terms of predicting classes. The higher the variance means that the values can be anything, which can lead to higher errors.

4) Write a function cv_error_knn() that uses 10-fold cross-validation to estimate the error rate for k-nearest neighbors. Briefly discuss the strategies you used to make your function run efficiently.

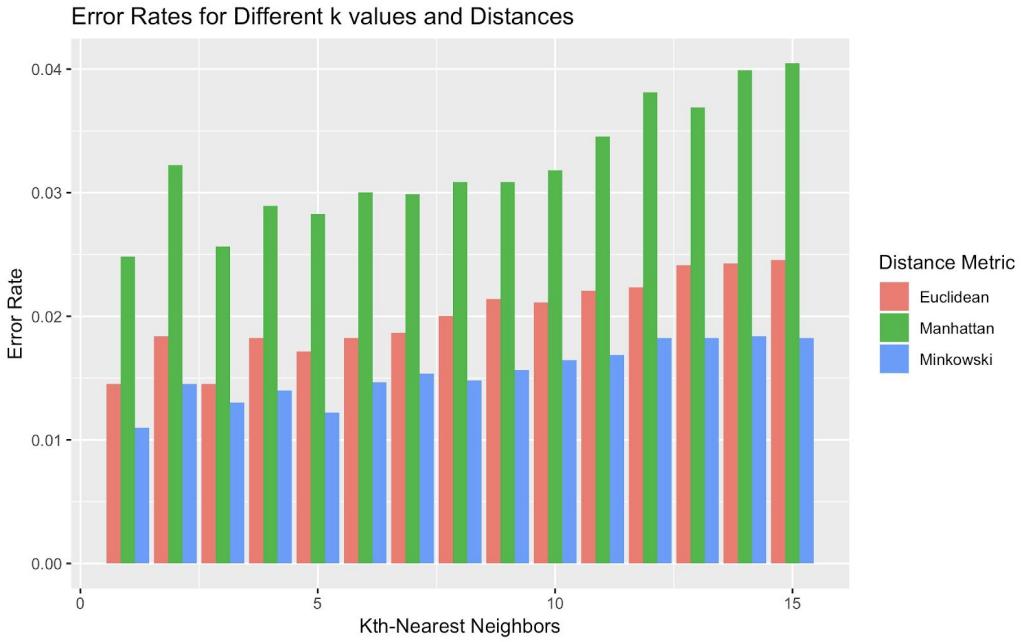
This function gave me the most trouble due to how inefficient I was at first when calculating the errors. I had left the dist() to constantly measure for each loop in the function, which lead to it taking an enormous amount of time to simulate. I had actually left it running while I got my haircut, and it still hadn't finished! Thinking of ways to move it out of the function, I realized I just

had to measure the distances *and then* manipulate the positions of the distance matrix in the loop. Originally having to find a new distance for each value of i in the for loop took around 15 minutes. After taking it out, it was able to finish simulating the function in under a few minutes for *all* distances.

My biggest obstacle was pulling the correct rows and columns from the distancematrices. Originally, I had left the whole distance matrix and tried to subset that, but that was unsuccessful. After hours of frustration and a hint on piazza, I realized that instead of making the whole distance matrix a parameter, we need to input the already subsetted rows and columns of the distance matrix. After that It was smooth sailing. The function didn't kill my computer and ran everything smoothly in the allotted time that it originally should have ran.

5) In one plot, display 10-fold CV error rates for all combinations of k = 1,...,15 and two different distance metrics. Which combination of k and distance metric works best for this data set? Would it be useful to consider additional values of k ?

First things first let's discuss what distance metric I used to calculate the different distances. I used: euclidean, manhattan, and minkowski distance with a power of 3. The error estimates were calculated by taking the number of predicted digits and actual digits for the test set, and then dividing by the number of rows in the test data. After that I subtracted by one to see what proportion of the predictions don't don't match up with the actual. If we plot the error rates per value of k for each method we get the following:



On first glance, Minkowski distance has the lowest of all error rates when it comes to the two other distances. Manhattan distance has significantly higher error compared to the other two distances. This makes sense with the way that Manhattan distances are calculated. To take a deeper look, I singled out the Minkowski distances. To see which k is the best for determining

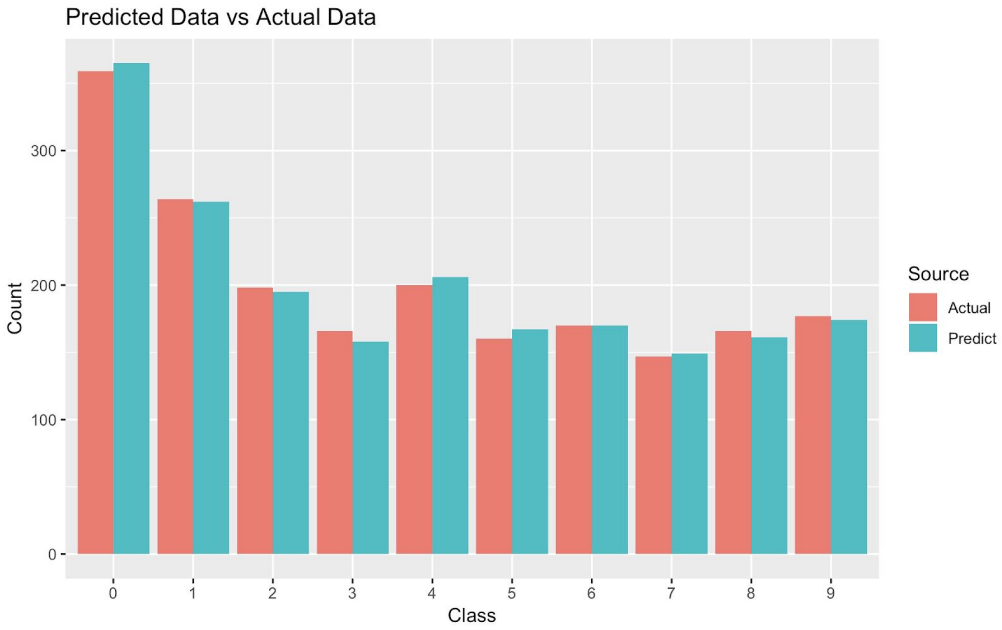
our nearest neighbors. The result that we get is k = 1 has the lowest error rate. My only explanation for this would be that as we increase k, the higher the number of neighbors. As a result, we get more variability. This leads to greater variance in our predictions, which in turn will lead to error up to a certain point. So with the data given, k = 1 using the Minkowski distance (which had a power of 3) leads to the lowest error rate.

6) In one plot, display the test set error rates for all combinations of k = 1,...,15 and two different distance metrics. How does the result compare to the 10-fold CV error rates? What kinds of digits does the "best" model tend to get wrong?

Similar to the problem above, I used a loop to calculate the the predict_knn digits for each value of k from one to fifteen. Again, I calculated the distance matrices for three different types of distances. The distances used were the same as question 5 with the Minkowski distance using three for the power. Using bar plots to compare the errors with the test, we get:

This plot has essentially the same patterns as the cross validation for the training set. It does have a higher error rate, which makes sense as we are inputting a whole different set of data. In fact, the error rates are almost double than those in our cross validation. Just like our plot for the cross validation the Manhattan distance has a significantly higher error rate than Euclidean and Minkowski.

Now comparing the predicted and class counts vs the actual class counts the plot looks like:



There honestly isn't that much of a difference between the two counts. Again, this is measure using the Minkowski data which also had the lowest error rates. If you look at the differences in height for each bar, the digits that have the lowest deltas are 1, 2, 9, and 6. 6 actually seems to have 0 difference in estimation, which is surprising. My prediction from the first few questions had it where those with the lowest variance, would have the most accurate predictions. The table below shows the mean variances of each class.

Class	0	1	2	3	4	5	6	7	8	9
Var	.4049	.0743	.4557	.3624	.3758	.4356	.3447	.2855	.3754	.3052

1 has the lowest variance by a mile, and the prediction was extremely close to what they actually were. Unfortunately, I see no relationship between variance and the prediction itself. There must be some other factor that determines accuracy of a model. For example, 6 has a variance of .3447, but seems to be perfect in terms of its prediction. While 1 has such a low mean variance the difference between predicted and actual isn't that much different from the other classes. Overall, my assumption that low variance leads to correct forecasting was wrong.

Sources:

- For the KNN Function:
  - https://piazza.com/class/jmf7qwk0sf03ya?cid=546
- For 10 Fold Cross Validation:
  - Jake Drew (https://stats.stackexchange.com/users/49529/jake-drew), How to split a data set to do 10-fold cross validation, URL (version: 2014-07-04): https://stats.stackexchange.com/q/105839

Code Appendix

```
# STA 141A
# Project 6
# Franco Gonzales
# 914953764
# 12/12/2018

setwd("/Users/franco/Desktop/R Files/141A/Project 6")

#Q1: Write a function read_digits()that reads a digits file into R. Your function must
allow
# users to specify the path to the file (training or test) that they want to read.


read_digits = function(path_file) { # Allow users to specify the path to the file
  file = read.table(path_file)
  names = names(file)
  names = tolower(names)
  names(file) = names
  colnames(file)[1] = "class"
  file
}


#
-------------------------------------------------------------------------------


#Q2: Explore the digits data:
#   * Display graphically what each digit (0 through 9) looks like on average.

train = read_digits("digits/train.txt")
test = read_digits("digits/test.txt")

# Let's see how the images look in general
train_matrix = as.matrix(test)
show_image = matrix(test2[1, 2:ncol(train)], 16, 16, byrow = T)
```

```
image(t(apply(show_image, 2, rev)), col=grey(seq(0,1,length=256)))

amount = table(train$class)
amount
# So the data has a ton of 0s and 1s with relatively equal amounts of everything else.

library(ggplot2)
train_plot = ggplot() + geom_density(aes(class), color = "red", data = train) +
  geom_density(aes(class), color = "blue", data = test) +
  ylim(0, .20) +
  scale_x_continuous(labels = as.character(0:9), breaks = (0:9))  +
  labs(x = "Class", title = "Distribution of Classes for Training Set")

train_separate = split(train, train$class)


means = lapply(train_separate, colMeans)
means = do.call(rbind, means)
means = as.matrix(means)


images = lapply(split(means, row(means)), function(x) t(apply(matrix(x[-1], 16, 16,
byrow=T), 2, rev)))
img = do.call(rbind, lapply(split(images[1:10], 1:10), function(x) do.call(cbind, x)))
image(img, col=grey(seq(0,1,length=100)))

library(matrixStats)

matrix_trainsep = lapply(train_separate, as.matrix)
vars = as.data.frame(t(as.data.frame(lapply(matrix_trainsep, colVars))))
names = tolower(names(vars))
names(vars) = vars

colnames(vars)[1] = "class"
vars$class = c(0:9)

vars_matrix = as.matrix(vars)
images2 = lapply(split(vars_matrix, row(vars_matrix)), function(x)
t(apply(matrix(x[-1], 16, 16, byrow=T), 2, rev)))
img2 = do.call(rbind, lapply(split(images2[1:10], 1:10), function(x) do.call(cbind,
x)))
image(img2, col=grey(seq(0,1,length=100)))

#
-------------------------------------------------------------------------------

# Q3: Write a function predict_knn() that uses k-nearest neighbors to predict the
label for
# a point or collection of points. At a minimum, your function must have parameters
for the
```

```
# prediction point(s), the training points, a distance metric, and k.

train = read_digits("digits/train.txt")
test = read_digits("digits/test.txt")

distance = function(x, type) {   # Let's make this the distance metric that we put
into
                                 # our KNN model. Let type = euclidean or mankowski or
etc.
  measure = dist(x, method = type, upper = TRUE, diag = TRUE)
  measure_mat = as.matrix(measure)
  return(measure_mat)

}

y = distance(rbind(test, train), "euclidean")
g = rownames(test)
distances = y[rownames(test), !colnames(y) %in% rownames(test)]

predict_knn = function(k, train, test, metric) {
  # Where metric is the type of distance
  # that we want to use.
  # colmatrix is there for the columns that we want in the matrix for our function to
predict.
  # Using cross-validation and comparing two different sets would have different
matrices.
  # It helps to choose the specific columns that we want.
  digits = train[ ,1]


  # Returns us a matrix with 2007 rows with columns that
  # correspond with the distances (total of 7291)

  min_distances2 = apply(metric, 1, function(x){ # Now find the K-nearest neighbor
    smallest = order(x)[1:k]                     # for each row in our test set.
    label = digits[smallest]
    votes = table(label)

    winner = which.max(votes)
    names = as.numeric(names(winner))
    digs =  return(names)
    digs
  })
  return(min_distances2) # Gives us the digits after calculating the k-nearest
neighbor.

}
```

```
#
-------------------------------------------------------------------------------------

# Q4: Write a function cv_error_knn() that uses 10-fold cross-validation to estimate
the
# error rate for k-nearest neighbors. Briefly discuss the strategies you used to make
your
# function run efficiently.

distance_train = distance(train, "euclidian")


cv_error_knn = function(b, k, train_set, metric) {
  set = train_set[sample(nrow(train_set)), ]
  folds = cut(seq(1,nrow(set)),breaks=b,labels=FALSE)

  error_estimates = 0

  for(i in 1:b){

    indexes = which(folds == i, arr.ind = TRUE)
    testset = set[indexes,]
    trainset = set[-indexes,]
    m = metric[rownames(testset), rownames(trainset)]

    labs = predict_knn(k, trainset, testset, m)
    error_estimates[i] = 1 -  (sum(labs == testset[, 1])/nrow(testset))
  }

  return(mean(error_estimates))
}



#
-------------------------------------------------------------------------------------
-

# Q5: In one plot, display 10-fold CV error rates for all combinations of k= 1,...,15
and
# two different distance metrics. Which combination of k and distance metric works
best for
# this data set? Would it be useful to consider additional values of k?

m1 = distance_train
m2 = distance(train, "manhattan")
m3 = as.matrix(dist(train, "minkowski", p = 3))
k = 15
```

```
euc_error = function(b, k, data, metric1, metric2, metric3){
  means_euc = 0
  for(i in 1:k){

    means_euc[i] = cv_error_knn(b, i, data, metric1)
  }

  df1  = data.frame(Errors = means_euc, k = 1:15, method = "Euclidean")

  means_man = 0
  for(i in 1:k) {

    means_man[i] = cv_error_knn(b, i, data, metric2)
  }

  df2 = data.frame(Errors = means_man, k = 1:15, method = "Manhattan")

  means_min = 0
  for(i in 1:k) {
    means_min[i] = cv_error_knn(b, i, data, metric3)
  }
  df3 = data.frame(Errors = means_min, k = 1:15, method = "Minkowski")

  final = rbind(df1,df2,df3)
  return(final)
}

# Writing this is actually 100x more work than making separate variables and then
binding
# but oh well haha.
library(ggplot2)

errors = euc_error(10, 15, train, m1, m2, m3)

ggplot(errors, aes(k, Errors, fill = method)) +
  geom_bar(stat = "identity", position = "dodge") +
  labs( x = "Kth-Nearest Neighbors", y = "Error Rate", title = "Error Rates for
Different k values and Distances", fill = "Distance Metric")

ggsave("error_train.jpeg", width = 8, height = 5)

# Find the best k by finding the k with the lowest error rate. Minkowski is the best
metric.

best_k = subset(errors, metric == "Minkowski")

# Best k value is 1.

#
--------------------------------------------------------------------------------
```

```
# Q6: In one plot, display the test set error rates for all combinations of k=
1,...,15
# and two different distance metrics. How does the result compare to the 10-fold CV
error
# rates? What kinds of digits does the “best” model tend to get wrong?

dist_traintest_euc = distance(rbind(test,train), "euclidean")
metric_euc = dist_traintest[rownames(test), !colnames(dist_traintest) %in%
rownames(test)]

dist_traintest_man = distance(rbind(test,train), "manhattan")
metric_man = dist_traintest_man[rownames(test), !colnames(dist_traintest_man) %in%
rownames(test)]

dist_traintest_min = dist(rbind(test,train), "minkowski", p = 3)
metric_min = dist_traintest_min[rownames(test), !colnames(dist_traintest_min) %in%
rownames(test)]

loop_test = function(k , train, test , metric, type) {
  error = 0
  for(i in 1:k) {
    digits = predict_knn(i, train, test, metric)
    error[i] =  1 - (sum(digits == test[,1])/nrow(test))


  }
  results = data.frame(Errors = error, k = 1:15, method = type)
  return(results)

}

errors_euc = loop_test(15, train, test, metric_euc, "Euclidean")
errors_man = loop_test(15, train, test, metric_man, "Manhattan")
errors_min = loop_test(15, train, test, metric_min, "Minkowski")
errors_all = rbind(errors_euc, errors_man, errors_min)

# Plot the data together to see which distance metric best suits our needs.

ggplot(errors_all, aes(k, Errors, fill = method)) + geom_bar(position = "dodge", stat
= "identity") +
  labs(x = "Values for k", y = "Error Rate", title = "Error Rate for Test Data", fill
= "Distance Metric")

ggsave("testerrors.jpeg", width = 8, height = 5)


cvv_results =  subset(errors, method == "Minkowski")
cvv_results$`Type` =  "Cross Validation"
```

```
test_results = subset(errors_all, method == "Minkowski")
test_results$`Type` = "Actual"

compare = rbind(test_results, cvv_results)

ggplot(compare, aes(k, Errors, fill = Type)) + geom_bar(position = "dodge", stat =
"identity") +
  labs(x = "Values for k", y = "Error Rate", title = "Error Rate CV vs Error Rate Test
Set")

# k = 3 had the lowest error rate.
test_digits = as.data.frame(table(test[,1]))
colnames(test_digits) = c("Class", "Count")
test_digits$`Source` = "Actual"

digits = predict_knn(3, train, test, metric_min)
predict_digits = as.data.frame(table(digits))
colnames(predict_digits) = c("Class", "Count" )
predict_digits$`Source` = "Predict"

compare2 = rbind(predict_digits, test_digits)


ggplot(compare2, aes(Class, Count, fill = Source)) +
  geom_bar(position = "dodge", stat = "identity") +
  labs(x = "Class", y = "Count", title = "Predicted Data vs Actual Data")

ggsave("predictvactual.jpeg", width = 8 , height = 5)

rowMeans(vars) # To see if lowest variances leads to accurate predictions for digits.
```