Final Project: Image Recognition

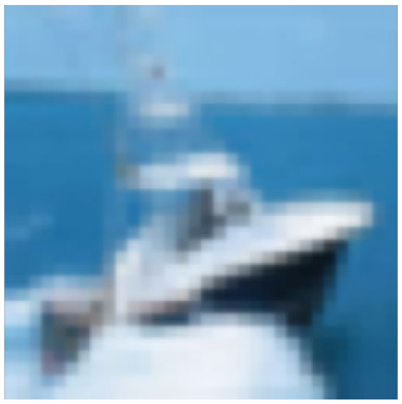
Jared Yu, Tiffany Chen, Emily Watkins
STA 141A - Spring 2018
Lecturer S. Gupta

3. Explore the image data. In addition to your own explorations:
Display graphically what each class (airplane, bird,cat...) looks like. You can randomly choose one image per class. Which pixels at which color channels seem the most likely to be useful for classification? Which pixels at which color channels seem the least likely to be useful for classification? Why?

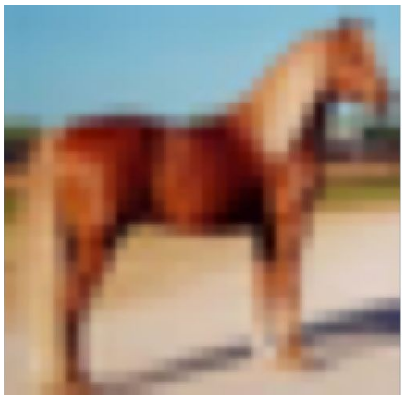
Frog

Airplane

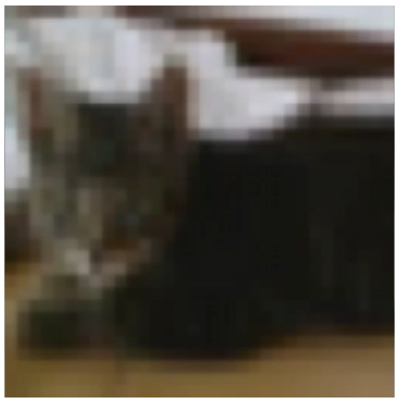
Ship



Horse



Cat



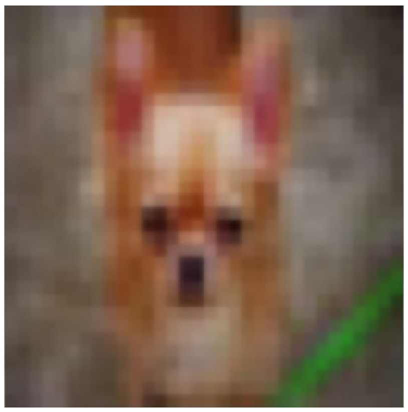
Automobile

Bird

Deer

Truck

Dog



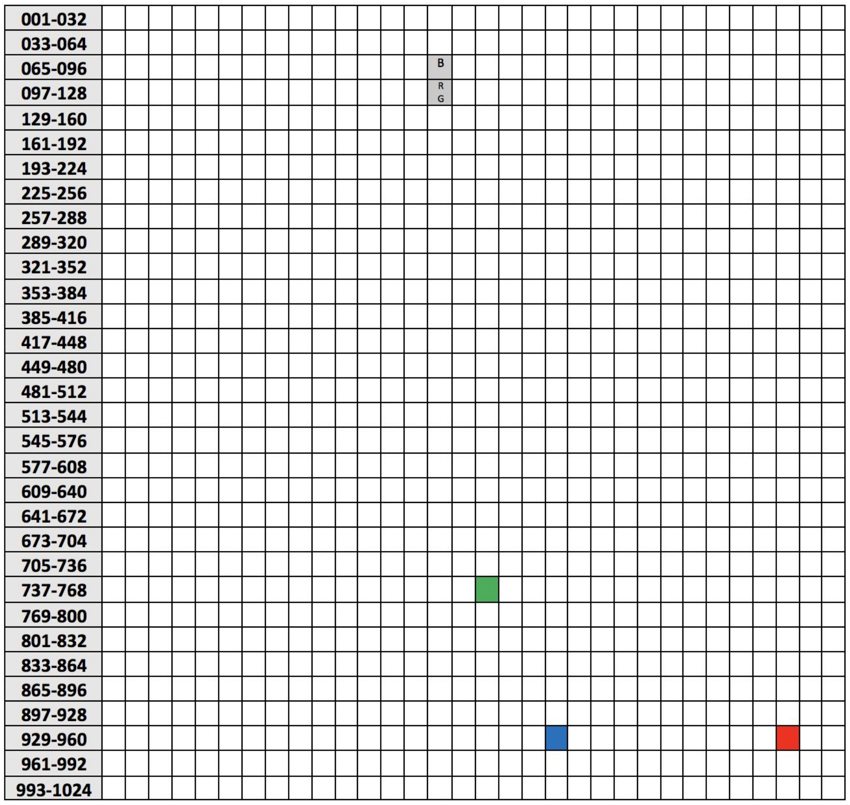
In order to find which pixels at which color channels seem the most likely to be useful for classification and which pixels at which color channels seem the least likely to be useful for classification, we observed the variance for each pixel within each color channel, for both the testing and training data. The pixels with the highest variance in a given color channel were determined to be the most useful for classification, and the pixels with the lowest variance in a given color channel were determined to be the least useful for classification. The pixels with the lowest variance varied the least in terms of color variation, therefore implying that they were used the least in terms of classifying the image (e.g. airplane, dog, cat, etc.). The pixels with the highest variance varied the most in terms of color variation, therefore implying that they were used the most when classifying images.

We find the following: for the training data, the *red* and *green* channels share an equally least useful pixel, pixel **111**, while the least useful pixel for *blue* was **79**. In the training data, the most useful pixels were **958** for *red*, **753** for *green*, and **948** for *blue*. For the testing data, all three color channels share an equally least-useful pixel, pixel **416**. The most useful pixels were **719** for *red*, **450** for *green*, **522** for *blue*.

The most and least useful pixels for the testing data set and training data set are shown below in the graphs titled **Training Data** and **Testing Data**. Both graphs are 32x32, with the leftmost column indicating pixel numbers for a given row, for reference. The grey-shaded pixels indicate the least useful pixels, and the *red*/*green*/*blue*-shaded pixels indicated the most useful pixels for that corresponding color channel.

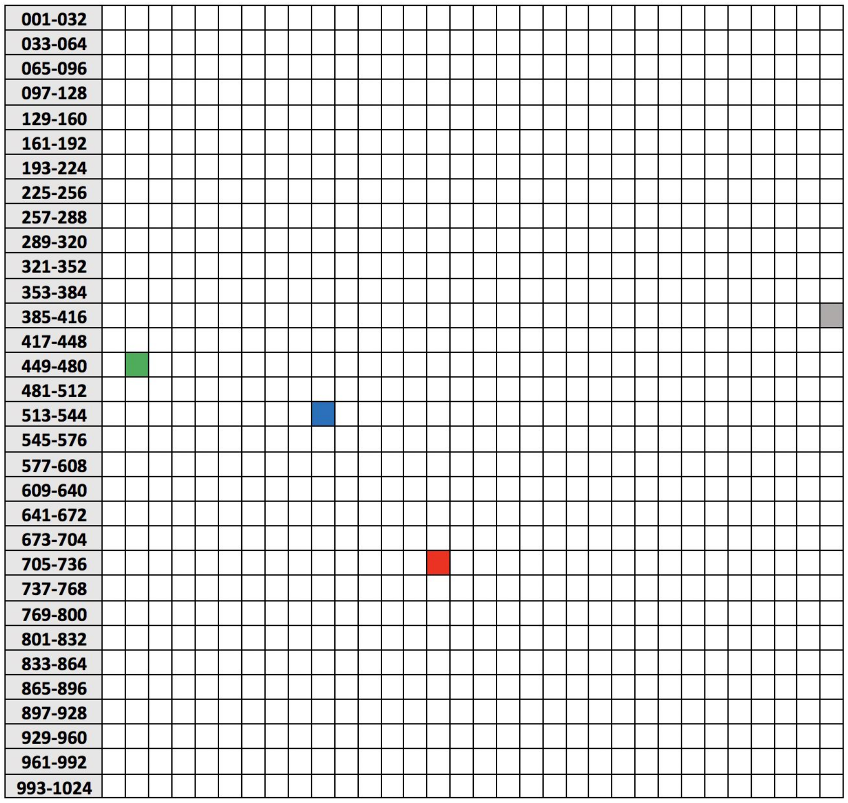
Surprisingly, the most useful pixels in the Training Data are closer towards the southern border of the image, but the most useful pixels in the Testing Data are closer towards the center of the image. We suspect that border pixels vary the most because they give context to an image — be it the forest, the ground, the sky, the ocean. But, the least useful pixels are also border pixels in both Training Data and Testing Data, and this may be due to the fact that the borders of images are usually empty space. Empty space can be black, gray, or brown — usually some darker color. The variation between those colors is small, making it "least useful."

Training Data



Least and Most Useful Pixels in R, G, B Color Channels in Training Dataset

Testing Data



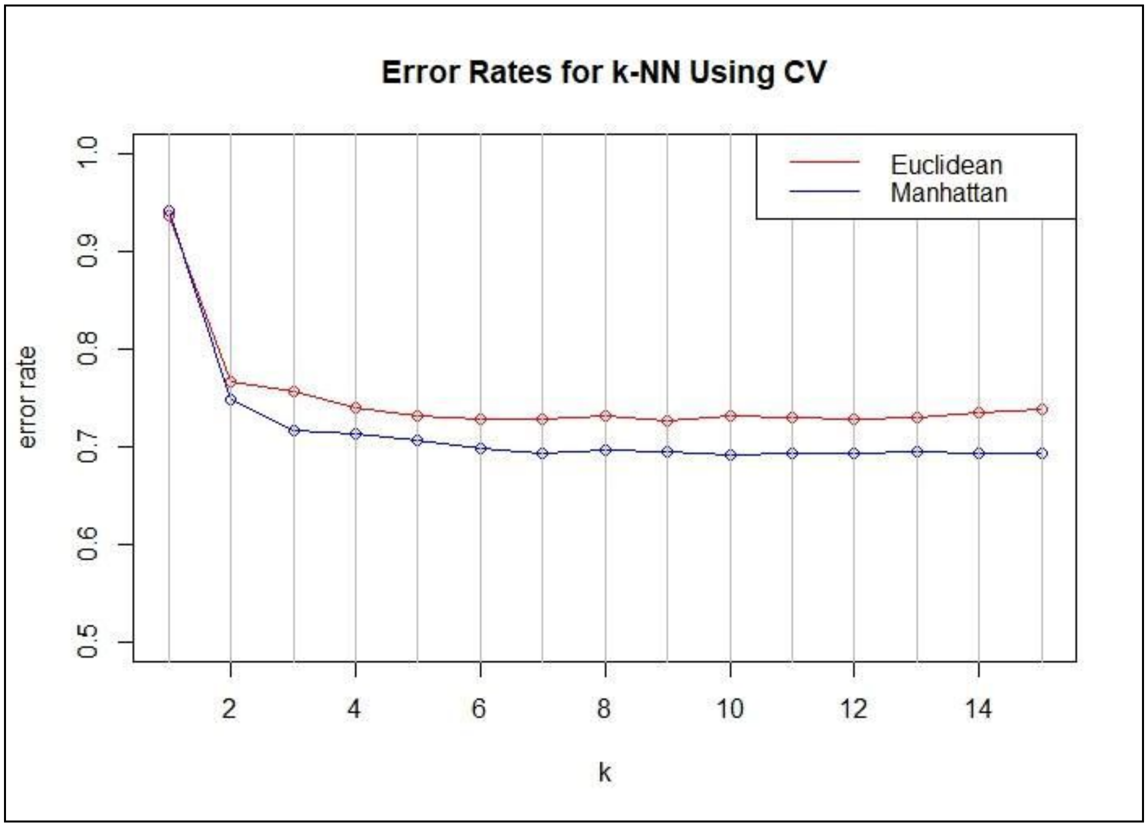
Least and Most Useful Pixels in R, G, B Color Channels in Testing Dataset

5. 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.

In this question, we chose to not use the *predict_knn()* function from **Question 4**. Instead, we simply took our distance matrices (Euclidean and Manhattan), subsetted the train-by-train portion of the matrix, and ran it through our function. We split the 5000-observation subset into 10 "folds" so that we can compare a single fold with nine other folds. In order to cross-validate, we called the single fold the "test set" and the rest of the folds the "training sets." We switched around the different folds so that each fold was considered a "test" once. From there, we calculated the average of our tests and got an accuracy rate of 25.44%, and an error rate of 74.56%.

The efficiency of our function lies in our strategy with the distance matrices. We referred to our previously computed distance matrices, which were computed for **Question 1**, so that we wouldn't have to recompute the distances for every fold in the *cv_error_knn()* function. Using the already-computed distance matrices contributed greatly to the overall efficiency of *cv_error_knn()*.

6. Display 10-fold cross-validation error rates for *k* = 1, …, 15 and at least 2 different distance metrics in one plot. Discuss your results. Which combination of *k* and *distance metric* is the best? Would it be useful to consider additional values of *k*?



The plot above illustrates the error rates of our 10-fold cross-validation using two *distance metrics*, Euclidean and Manhattan. We tried different combinations of *metrics* and *k* (*k* = 1...15), and we found that the combination with the lowest error rate is *k* = 10 and the Manhattan *distance metric*. The Manhattan *distance metric* was consistently associated with a lower error rate.

It would not be useful to consider additional values of *k* based on the downward trend in the plot. The trend starts to stagnate at *k* = 6, and there doesn't seem to be anymore dramatic decreases after that point.

7. For each of the 3 best *k* and *distance metric* combinations, use 10-fold cross-validation to estimate the confusion matrix. Discuss your results. Does this change which combination you would choose as the best?

Key for Classes in Confusion Matrices

Number	Class
0	Airplane
1	Automobile
2	Bird
3	Cat
4	Deer
5	Dog
6	Frog
7	Horse
8	Ship
9	Truck

Confusion Matrix (*Distance Metric* = Manhattan; *k* = 10)

		Predicted Class									
		0	1	2	3	4	5	6	7	8	9
Actual Class	0	253	3	63	3	28	1	18	6	123	2
	1	56	78	60	22	100	11	46	10	95	22
	2	63	1	229	25	114	8	26	9	24	1
	3	45	5	113	72	119	40	65	13	24	4
	4	46	3	134	13	235	3	32	14	17	3
	5	42	2	110	65	116	83	48	5	27	2
	6	25	2	151	26	140	7	142	2	4	1
	7	54	2	99	20	159	14	30	86	26	10
	8	118	6	29	6	38	8	8	3	275	9
	9	72	38	49	29	45	7	22	32	119	87

Confusion Matrix (*Distance Metric* = Manhattan; *k* = 15)

		Predicted Class									
Actual Class		0	1	2	3	4	5	6	7	8	9
	0	259	1	54	7	28	2	19	6	121	3
	1	43	59	66	21	115	8	44	10	100	34
	2	64	0	213	20	122	7	30	9	33	2
	3	43	4	121	68	112	34	75	13	25	5
	4	37	3	140	11	238	5	34	10	19	3
	5	34	5	113	51	125	83	51	7	28	3
	6	23	2	154	23	138	11	138	4	6	1
	7	54	0	112	19	163	12	18	29	23	10
	8	103	5	24	6	39	8	5	5	298	7
	9	53	31	57	16	44	5	30	31	142	91

Confusion Matrix (*Distance Metric* = Manhattan; *k* = 7)

		Predicted Class									
Actual Class		0	1	2	3	4	5	6	7	8	9
	0	259	2	73	8	26	2	8	7	113	2
	1	65	86	52	30	98	11	38	9	86	25
	2	62	2	222	23	126	7	23	7	23	5
	3	54	9	126	80	111	43	49	10	15	3
	4	48	3	141	8	236	3	34	14	12	1
	5	47	1	106	65	112	92	39	12	24	2
	6	29	5	167	35	120	7	131	2	4	0
	7	64	6	102	25	139	12	28	88	28	8
	8	125	9	37	4	31	11	4	8	263	8
	9	75	40	53	28	48	7	27	25	119	78

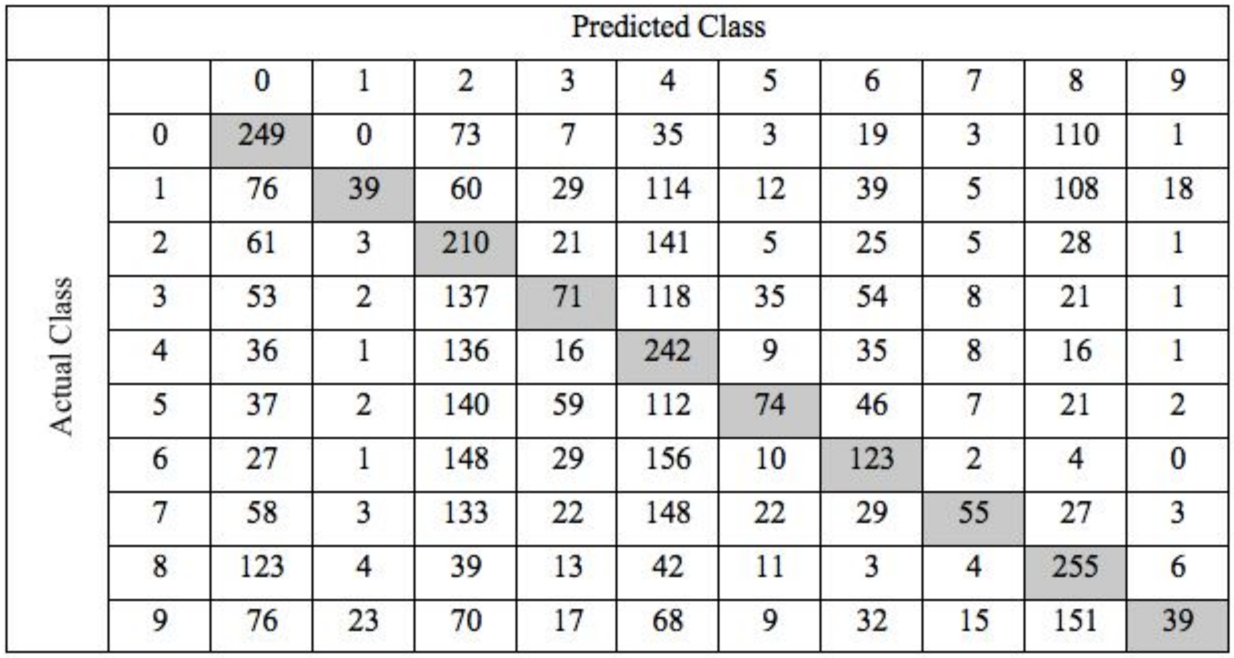
Confusion Matrix (*Distance Metric* = Euclidean; *k* = 9)

	Predicted Class										
		0	1	2	3	4	5	6	7	8	9
Actual Class	0	242	0	74	6	39	5	20	4	109	1
	1	61	40	70	18	125	12	38	8	109	19
	2	77	1	204	20	141	5	25	4	22	1
	3	50	3	137	56	132	43	47	7	24	1
	4	35	1	142	15	244	4	29	10	19	1
	5	39	1	123	52	129	74	51	6	24	1
	6	20	1	159	19	163	9	123	1	5	0
	7	53	2	128	23	174	14	26	50	27	3
	8	120	4	39	8	41	13	5	5	260	5
	9	67	21	72	14	65	6	33	19	158	45

Confusion Matrix (*Distance Metric* = Euclidean; *k* = 6)

	Predicted Class										
		0	1	2	3	4	5	6	7	8	9
Actual Class	0	258	1	64	9	34	2	20	5	106	1
	1	80	47	67	26	119	7	32	7	100	15
	2	70	2	206	24	134	6	26	1	31	0
	3	50	3	131	73	124	52	41	6	19	1
	4	40	0	132	21	238	6	36	10	16	1
	5	38	2	114	66	125	81	41	10	19	4
	6	23	2	155	27	161	10	113	5	3	1
	7	68	1	130	22	153	13	28	50	30	5
	8	146	5	35	14	38	8	6	6	235	7
	9	84	24	71	19	62	8	31	14	140	47

Confusion Matrix (*Distance Metric* = Euclidean; *k* = 7)



		Predicted Class									
Actual Class		0	1	2	3	4	5	6	7	8	9
	0	249	0	73	7	35	3	19	3	110	1
	1	76	39	60	29	114	12	39	5	108	18
	2	61	3	210	21	141	5	25	5	28	1
	3	53	2	137	71	118	35	54	8	21	1
	4	36	1	136	16	242	9	35	8	16	1
	5	37	2	140	59	112	74	46	7	21	2
	6	27	1	148	29	156	10	123	2	4	0
	7	58	3	133	22	148	22	29	55	27	3
	8	123	4	39	13	42	11	3	4	255	6
	9	76	23	70	17	68	9	32	15	151	39

The 3 best *k* and *distance metric* combinations for the Manhattan metric are *k* = 10, *k* = 15, and *k* = 7. The 3 best *k* and *distance metric* combinations for the Euclidean metric are *k* = 9, *k* = 6, and *k* = 7.

The Manhattan metric consistently predicts Automobile (1) very well relative to other predictions. For all values of *k*, truck (9) is frequently mispredicted as ship (8). In general, the model most often predicts bird (2) and deer (4), although these are often misclassifications.

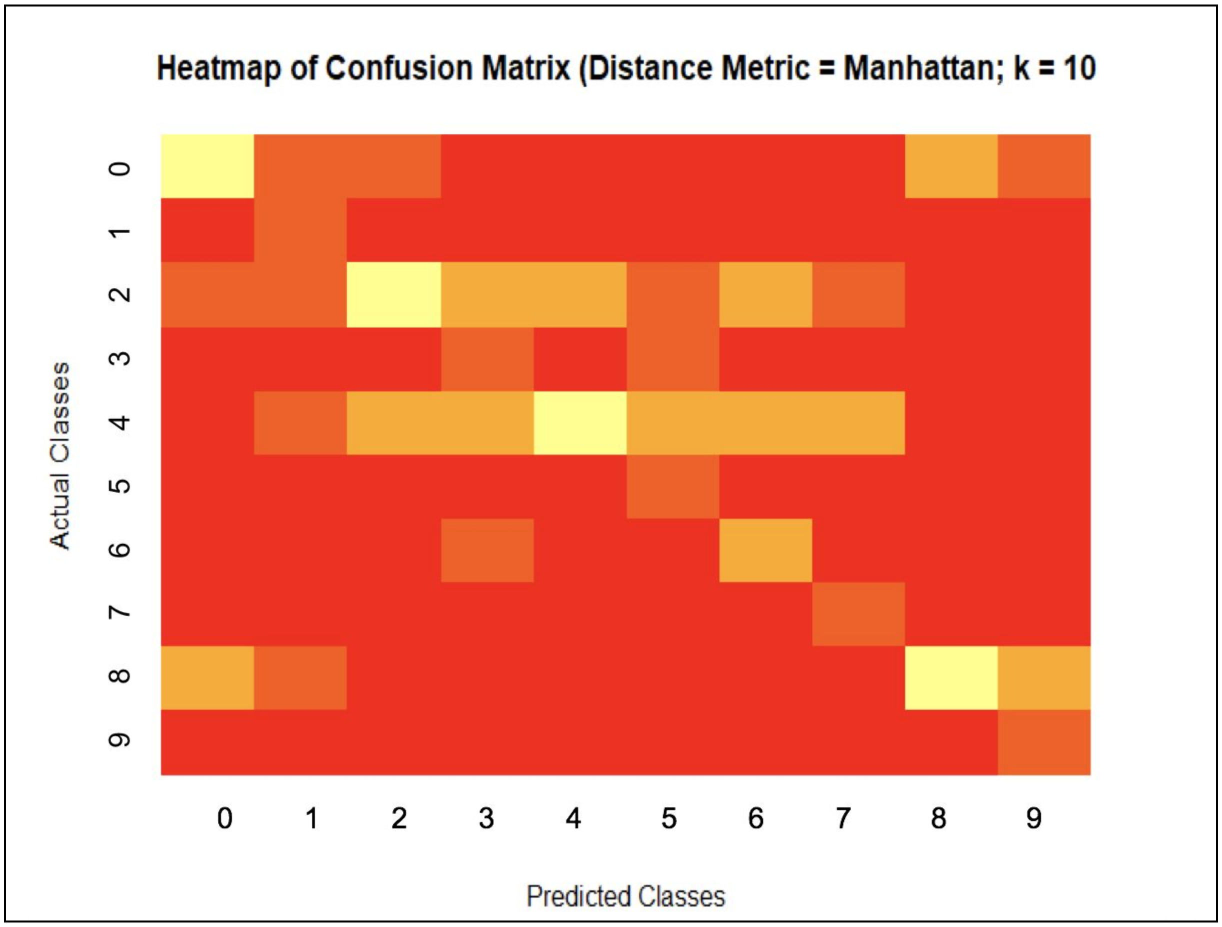
Meanwhile, the Euclidean metric best predicts Automobile (1), Dog (5), Frog (6), Horse (7), and Truck (9). But, it struggles in classifying Bird (2) and Deer (4). Something interesting to note is the classifier's struggle to differentiate between Airplane (0), Automobile (1), Ship (8), and Truck (9).

8. For the best *k* and *distance metric* combination, explore the training data that were misclassified during cross-validation by displaying a confusion matrix. Discuss what you can conclude about the classifier.

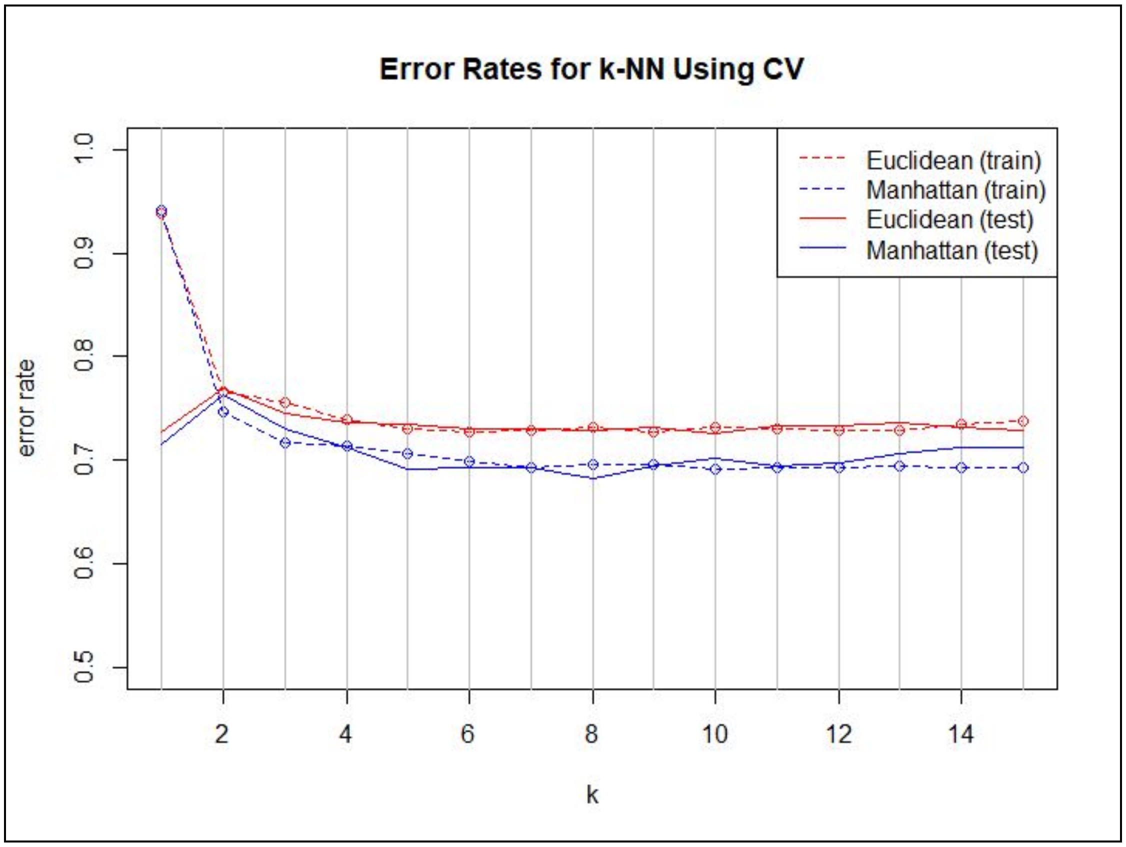
The single best *k* and *distance metric* combination is *k* = 10 and the Manhattan metric, according to our error rate plot in **Question 6**. Based on frequency alone (from the heatmap), the most frequently correctly predicted classes are Airplane (0), Bird (2), Deer (4), and Ship (8). In our opinion, the classes that are actually best predicted with this *k* and *distance metric* combination are Automobile (1) and Ship (8), the frequency of correct predictions is high relative to the frequency of mispredictions. Some patterns that we noticed are that animals are often mispredicted as other animals, and objects are often mispredicted as other objects. For example, the Bird (2) is often predicted as Cat (3), Dog (5), Frog (6), and Horse (7), while the Deer (4) is often mispredicted as Automobile (1), Bird (2), Cat (3), Dog (5), Frog (6), and Horse (7). The Airplane (0), Automobile (1), Ship (8), and Truck (9) are often mistaken for each other as well. Lastly, the one class that is most frequently predicted is Deer (4). It makes sense, because if you refer back to our image of a deer in **Question 3**, the colors present are brown, black, white, and silver. These colors are often used in objects and animals alike. In addition, deer are thicc and four-legged creatures that may appear to be four-wheeled objects or other four-legged creatures. Make note that horses are also thick and four-legged, but horses' leg-to-torso ratio is greater than a deer's leg-to-torso ratio. Therefore, it makes sense as to why the deer is often misclassified as every other class. (Side note: made a heatmap as suggested in Piazza)

Confusion Matrix (*Distance Metric* = Manhattan; *k* = 10)

		Predicted Class									
		0	1	2	3	4	5	6	7	8	9
Actual Class	0	253	3	63	3	28	1	18	6	123	2
	1	56	78	60	22	100	11	46	10	95	22
	2	63	1	229	25	114	8	26	9	24	1
	3	45	5	113	72	119	40	65	13	24	4
	4	46	3	134	13	235	3	32	14	17	3
	5	42	2	110	65	116	83	48	5	27	2
	6	25	2	151	26	140	7	142	2	4	1
	7	54	2	99	20	159	14	30	86	26	10
	8	118	6	29	6	38	8	8	3	275	9
	9	72	38	49	29	45	7	22	32	119	87


Heatmap of Confusion Matrix (Distance Metric = Manhattan; k = 10
Actual Classes
Predicted Classes
0
1
2
3
4
5
6
7
8
9

9. Display test set error rates for *k* = 1, …, 15 and at least 2 different *distance metrics* in one plot. Compare your results to the 10-fold cross-validation error rates.



The plot of 10-fold cross-validation error rates of the test dataset is strange in that the error rate has an upward trend from *k* = 1 to *k* = 2, whereas the training dataset has a steep downward trend for those values of *k*. The trend of the error rates of the test dataset are generally similar to the trend of the error rates of the training dataset. While the best combination for the lowest error rate of the training dataset is *k* = 10 and the Manhattan *distance metric*, the best combination of the test dataset is *k* = 8 and the Manhattan *distance metric.* It seems that the Manhattan *distance metric* is the best way to go about calculating distances and lowering error rates in both datasets.

10. Briefly summarize what each group member contributed to the group.

Jared wrote a large portion of the code with significant help from Tiffany and Emily. Emily and Tiffany wrote the report. Jared, Emily, and Tiffany all dedicated time to attending office hours in order to gain knowledge for the project.

Appendix

Sources:

https://l.facebook.com/l.php?u=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F7421503%2Fhow-to-plot-a-confusion-matrix-using-heatmaps-in-r&h=ATNY05vWTfDYc-4pF6iDRNb6_cOjvNh-jPJMCow0Co3k4bJnBLpgM0ryCAf0jg68vkxA4OUB5R86VJJSxFJwBll47_STQgs5krXxNabCBXb9E3otW2X8b95zcT6wcINoRLeRcQ
https://www.tutorialspoint.com/r/r_binary_files.htm
https://stats.idre.ucla.edu/r/faq/how-can-i-read-binary-data-into-r/
http://stat.ethz.ch/R-manual/R-devel/library/base/html/readBin.html
http://www.sthda.com/english/wiki/reading-data-from-txt-csv-files-r-base-functions
https://stat.ethz.ch/R-manual/R-devel/library/grid/html/grid.raster.html
https://stackoverflow.com/questions/5237557/extract-every-nth-element-of-a-vector
https://stackoverflow.com/questions/21807987/calculate-the-mean-for-each-column-of-a-matrix-in-r
https://stackoverflow.com/questions/5812493/adding-leading-zeros-using-r?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa
https://stat.ethz.ch/pipermail/r-help/2017-February/445089.html

Piazza: @672, @702, @688, @752, @751

Code:

Code:

```
library(grid); library(broman); library(ggplot2); library(reshape2) # Load li
braries
setwd("C:/Users/qizhe/Desktop/STA 141A/final") # Set working directory

### 1
list_to_matrix = function(bin_list) {
  bin_list = as.character(bin_list)
  bin_matrix = matrix(bin_list, nrow = 10000, byrow = TRUE)
  return(bin_matrix)
} # Converts a list to a matrix

load_training_images = function(input_dir, output_file) {
  bin_data = list.files(path = input_dir, # Extract binary files
                        pattern = "data_batch_[0-9].bin", full.names = TRUE)
  bin_list = lapply(bin_data, function(x) readBin(con = x, # Read the binary
data
                                                  what = "raw", n = 3073*1000
0))
  bin_matrices = lapply(bin_list, list_to_matrix) # Change vectors to matrice
s in list
  train = do.call("rbind", bin_matrices) # Combine the list into a single dat
a
  save(train, file = output_file) # Save to file
} # Loads the binary data and converts them to workable data (training)

load_testing_images = function(input_dir, output_file) {
  bin_data = list.files(path = input_dir, # Extract binary file
                        pattern = "test_batch.bin", full.names = TRUE)
  bin_vec = readBin(con = bin_data, what = "raw", n = 3073*10000) # Read the
binary data
  bin_vec = as.character(bin_vec) # Set to character
  test = matrix(bin_vec, nrow = 10000, byrow = TRUE) # Change to matrix
  save(test, file = output_file) # Save to file
} # Loads the binary data and converts them to workable data (test)

training_images = load_training_images(input_dir = "C:/Users/qizhe/Desktop/ST
A 141A/final",
                                       output_file = "C:/Users/qizhe/Desktop/
STA 141A/final/training_set.rds")

testing_images = load_testing_images(input_dir = "C:/Users/qizhe/Desktop/STA
141A/final",
                                     output_file = "C:/Users/qizhe/Desktop/ST
A 141A/final/test_set.rds")

load("C:/Users/qizhe/Desktop/STA 141A/final/training_set.rds") # Load the sav
ed data
```

```r
load("C:/Users/qizhe/Desktop/STA 141A/final/test_set.rds")

data_rescale <- function(labels, k = 500) {
  sort(as.vector(sapply(unique(labels),
                        function(i) which(labels == i))[1:k, ]))
} # TA code for rescaling

train2 <- train[data_rescale(train[,1], k = 500),] # Rescale the data
test2 <- test[data_rescale(test[,1], k = 100),]

### 2
view_images = function(image_data, observation, image_labels) {
  photo_label = as.numeric(image_data[observation,1]) # Get the integer of th
e observation label
  photo_label = image_labels[photo_label + 1,] # Get the corresponding name o
f the label

  red = sapply(image_data[observation, 2:1025], hex2dec) # Change colors to d
ecimal
  green = sapply(image_data[observation, 1026:2049], hex2dec)
  blue = sapply(image_data[observation, 2050:3073], hex2dec)

  rgb_mat = matrix(rgb(red, green, blue, # Create matrix of colors
                       maxColorValue = 255), nrow = 32, ncol = 32, byrow = T)
  grid.raster(rgb_mat) # Plot image
  return(photo_label)
} # Display the image of a particular data point

image_label = data_batch_1[seq(3073, length(data_batch_1), 3073)] # Extract t
he labels of each row

### 3
# Find an example of each class within the data
view_images(train2, 105, image_labels) # 1. Frog
view_images(train2, 31, image_labels) # 2. Airplane
view_images(train2, 9, image_labels) # 3. Ship
view_images(train2, 44, image_labels) # 4. Horse
view_images(train2, 10, image_labels) # 5. Cat
view_images(train2, 499, image_labels) # 6. Automobile
view_images(train2, 109, image_labels) # 7. Bird
view_images(train2, 300, image_labels) # 8. Deer
view_images(train2, 358, image_labels) # 9. Truck
view_images(train2, 785, image_labels) # 10. Dog

color_range = function(color) {
  if (color == "red") {
    color = 2:1025
  } else if (color == "green") {
    color = 1026:2049
  } else if (color == "blue") {
    color = 2050:3073
  }
  return(color)
} # Determine the range for a specific color

class_index = function(image_labels, class_name) {
  label_index = (which(image_labels == class_name) - 1)
  return(paste0("0",label_index))
} # Return the index of the class

pixel_variance = function(color, image_data) {
  color_subset = color_range(color = color) # Determine the range for given c
olor
  colored_data = image_data[1:nrow(image_data), color_subset] # Subset the co
lor from the data
  colored_data_mat = matrix(hex2dec(colored_data), # Transform data into matr
ix
                            nrow = nrow(colored_data), ncol = ncol(colored_da
ta), byrow = TRUE)
  colored_data_var = apply(colored_data_mat, 2, var)
  special_pixel = which(colored_data_var == max(colored_data_var))
  unspecial_pixel = which(colored_data_var == min(colored_data_var))
  pixels = c(special_pixel, unspecial_pixel)
  print("Special, then unspecial pixels:")
  return(pixels)
} # Find the pixel with greatest variance from a specific color

# Find variance for RGB within test and train data
pixel_variance(color = "red", image_data = test2)
pixel_variance(color = "green", image_data = test2)
pixel_variance(color = "blue", image_data = test2)

pixel_variance(color = "red", image_data = train2)
pixel_variance(color = "green", image_data = train2)
pixel_variance(color = "blue", image_data = train2)

### 4
top_k = function(dist_mat, k, test_data, training_data) {
  top_k_mat = label_indices = matrix(NA, nrow = nrow(dist_mat),
                                     ncol = k) # Create dummy matrix for orde
red labels

  for (i in 1:nrow(dist_mat)) {
    top_5 = names(head(sort(dist_mat[i,], # Retrieve the top k training obser
vations that match the test
                            decreasing = FALSE), k))
    top_5_index = as.integer(top_5) - nrow(test_data) # Convert to format of
training indices
```

```r
    top_k_mat[i,] = top_5_index # Fill the rows of the dummy matrix
  } # Fill a dummy matrix with the top k training observations

  for (i in 1:nrow(label_indices)) {
    label_indices[i,] = training_data[top_k_mat[i,], 1]
  } # Determine the label of each of the top k training observations

  label_indices2 = as.integer(label_indices) + 1 # Convert to the image_label
s indices
  label_indices2 = matrix(label_indices2, # Convert back to matrix
                          nrow = nrow(label_indices), ncol = ncol(label_indic
es))

  return(label_indices2)
} # Find the top k labels per observation from the distance matrix

vote_k = function(test_data, training_data, k_mat) {
  vote_label = rep(NA, nrow(k_mat)) # Dummy vector for vote labels

  for (i in 1:nrow(k_mat)) {
    vote_label[i] = sample(names(which(table(k_mat[i,]) == # Vote for the tes
t label
                                         sort(table(k_mat[i,]), decreasing =
TRUE)[1])), 1)
  }
  vote_label = as.integer(vote_label) # Convert back to integer

  return(vote_label)
} # Vote for the test label, and determine the accuracy

predict_knn = function(test_data, training_data, distance_metric, k) {
  test_data2 = test_data[,-1] # Remove the label column from both test and tr
aining data
  training_data2 = training_data[,-1]

  test_train_mat = rbind(test_data2, training_data2) # Combine both into a si
ngle matrix
  test_train_mat2 = matrix(hex2dec(test_train_mat), # Convert to integer pixe
ls, and keep as matrix
                           nrow = (nrow(test_data2) + nrow(training_data2)),
ncol = ncol(test_data2))

  dist_mat = as.matrix(dist(test_train_mat2, method = distance_metric)) # Tak
e the distance matrix of the combined matrix
  dist_mat2 = dist_mat[1:nrow(test_data2), # Subset the informative parts of
the distance matrix
                       (nrow(test_data2) + 1):(nrow(test_data2) + nrow(traini
ng_data2))]
```

```
  k_mat = top_k(dist_mat = dist_mat2, k = k, # Discern the k nearest labels
                test_data = test_data, training_data = training_data)

  k_votes = vote_k(test_data = test_data, # Vote for the test label
                   training_data = training_data, k_mat = k_mat)

  return(k_votes)
}

# predict_knn(test_data = test3, training_data = train3, distance_metric = "e
uclidean", k = 3)

### 5
### Create the entire distance matrix once for Euclidean and Manhattan distan
ces
test2c = test2[,-1]; test2d = apply(test2c, 2, function(x) strtoi(x, 16L)) #
Remove labels, set to integer
train2c = train2[,-1]; train2d = apply(train2c, 2, function(x) strtoi(x, 16L)
) # Same for train data
dist_euclidean = dist(rbind(test2d, train2d)); dist_euc = as.matrix(dist_eucl
idean) # Create distance matrix
save(dist_euc, file = "dist_euclidean") # Save file
dist_manhattan = dist(rbind(test2d, train2d), method = "manhattan"); dist_man
= as.matrix(dist_manhattan)
save(dist_man, file = "dist_manhattan") # Same for Manhattan distances

# read in dist_euc
load("dist_euclidean.rda"); load("dist_manhattan.rda"); load("test_set.rds");
load("training_set.rds")

cv_error_knn = function(train2, test2, k = 3, numOfFolds = 10, all_distance){
  n = nrow(train2)
  m = nrow(test2)
  real_labels = train2[,1] # Retrieve labels
  all_distance = as.matrix(all_distance) # Change to distance matrix
  all_distance = all_distance[-c(1:m), -c(1:m)] # Subset the correct data
  colnames(all_distance) = 1:nrow(all_distance)
  row.names(all_distance) = 1:nrow(all_distance)
  classes = sample(rep(1:10,500)) # Generate list of classes
  indexes = split(1:n, classes) # indexes[[1]] show the index of the images f
rom training set goes to first fold
  fold_distance = lapply(1:numOfFolds, function(x) all_distance[do.call("c",
indexes[-x]), indexes[[x]]]) # calculate distance of one fold vs the other fo
lds (9 folds)
  top_ks = lapply(fold_distance, function(x) apply(x, 2, function(y) real_lab
els[as.numeric(names(sort(y))[1:k])])) # select the top k
  if (k == 1) {
    top_classes = lapply(top_ks, function(x) names(x)[1])
  } else {
    top_classes = lapply(top_ks, function(x) apply(x, 2, function(y) names(so
rt(table(y), decreasing=TRUE))[1]))
  }
  return(list(true = as.numeric(real_labels[do.call("c", indexes)]), predict
= as.numeric(do.call("c", top_classes))))
}

### 6
err.euc = err.man = rep(0, 15) # Create empty vector for errors
err.euc.true = err.man.true = err.euc.predict = err.man.predict = vector("lis
t", 15) # Empty list for pred/true

set.seed(456) # Set seed
for (i in c(1:15)) {
  out_euc = cv_error_knn(train2, test2, k = i, numOfFolds = 10, all_distance
= dist_euc)
  err.euc[i] = mean(out_euc$true != out_euc$predict)
  err.euc.true[[i]] = out_euc$true
  err.euc.predict[[i]] = out_euc$predict

  out_man = cv_error_knn(train2, test2, k = i, numOfFolds = 10, all_distance
= dist_man)
  err.man[i] = mean(out_man$true != out_man$predict)
  err.man.true[[i]] = out_man$true
  err.man.predict[[i]] = out_man$predict
} # Produce output for k = 1,...,15 and 10 folds for Euclidean and Manhattan
distances

plot(1:15, err.euc, main = "Error Rates for k-NN Using CV", # Plot the error
rates
     xlab= "k", ylab= "error rate", ylim=c(0.5, 1), col="red", type = "l")
points(1:15, err.euc, col = "red"); lines(1:15, err.man, col="blue")
points(1:15, err.man, col = "blue"); abline(v=c(1:15), col="grey")
legend("topright", c("Euclidean", "Manhattan"), col = c('red', 'blue'), lty=c
(1,1))

### 7
# Determine the top 3 for Euclidean and Manhattan distances
order(err.euc)[1:3] # 9, 6, 7
order(err.man)[1:3] # 10, 15, 7

# Euclidean Confusion Matrix
table(data.frame(true = err.euc.true[[9]], predict = err.euc.predict[[9]]))
table(data.frame(true = err.euc.true[[6]], predict = err.euc.predict[[6]]))
table(data.frame(true = err.euc.true[[7]], predict = err.euc.predict[[7]]))
# Manhattan Confusion Matrix
table(data.frame(true = err.man.true[[10]], predict = err.man.predict[[10]]))
table(data.frame(true = err.man.true[[15]], predict = err.man.predict[[15]]))
table(data.frame(true = err.man.true[[7]], predict = err.man.predict[[7]]))

### 8
table(data.frame(true = err.man.true[[10]], predict = err.man.predict[[10]]))

# Heatmap
bestcombo.tab = table(data.frame(true = err.man.true[[10]], predict = err.man
.predict[[10]]))
bestcombo.df = matrix(bestcombo.tab, ncol = 10)

image(bestcombo.df[, ncol(bestcombo.df):1],
      xlab = "Predicted Classes", ylab = "Actual Classes",
      main = "Heatmap of Confusion Matrix (Distance Metric = Manhattan; k = 1
0",
      axes = F, col = heat.colors(5))

### 9
test_error_knn = function(train2, test2, k = 10, all_distance){
  n = nrow(train2) # Get rows for test and train data
  m = nrow(test2)
  real_labels = train2[,1] # Subset true labels from train
  all_distance = as.matrix(all_distance) # Convert to distance matrix
  all_distance = all_distance[-c(1:m), c(1:m)] # Subset the test x train data
  colnames(all_distance) = 1:m # Correct column and row names
  row.names(all_distance) = 1:nrow(all_distance)
  top_k = apply(all_distance, 2, function(y) real_labels[as.numeric(names(sor
t(y))[1:k])]) # Find top k
  if (k == 1) {
    predict_labels = as.numeric(top_k)
  } else {
    predict_labels = apply(top_k, 2, function(x) as.numeric(names(sort(table(
x), decreasing=TRUE))[1]))
  }
  return(list(true = as.numeric(test2[,1]), predict = predict_labels))
} # Find the knn error rate for the test data

euc_test_err = rep(0, 15); man_test_err = rep(0, 15) # Create empty vectors
set.seed(141)
for (i in c(1:15)) {
  predict1 = test_error_knn(train2, test2, k = i, dist_euc)
  euc_test_err[i] = mean(predict1$true != predict1$predict)

  predict1 = test_error_knn(train2, test2, k = i, dist_man)
  man_test_err[i] = mean(predict1$true != predict1$predict)
} # Retrieve the error rates for the test data

plot(1:15, err.euc, main = "Error Rates for k-NN Using CV", # Plot the error
rates
     xlab= "k", ylab= "error rate", ylim=c(0.5, 1), col="red", type = "l", lt
y = 2)
points(1:15, err.euc, col = "red"); lines(1:15, err.man, col="blue", lty = 2)
points(1:15, err.man, col = "blue"); abline(v=c(1:15), col="grey")
lines(1:15, euc_test_err, col = "red"); lines(1:15, man_test_err, col = "blue
")
legend("topright", c("Euclidean (train)", "Manhattan (train)", "Euclidean (te
st)", "Manhattan (test)"),
       col = c('red', 'blue', 'red', 'blue'), lty=c(2,2,1,1))

order(euc_test_err)[1:3] # 10, 1, 8
order(man_test_err)[1:3] # 8, 5, 6
```