Classifying Digits with kNN and Cross Validation

Kevin DelRosso

November 3, 2015

Setting up the workspace

```
rm(list = ls())
set.seed(377)
setwd("~/Desktop/STA_141/assignment_3/")

require(scales) # for changing alpha level in plots
require(png)
require(grid)
require(magrittr) # for updating row names (while returning object)
require(lattice)
```

Loading in the data.

```
# Download data from url and return as data.frame
get_data = function(url) {

    filename = "digitsTrain.csv"

    # if file doesn't exists, download using the url
    if( !file.exists(filename) ) download.file(url, filename)

    read.csv(filename)
}

link = "http://eeyore.ucdavis.edu/sta141/Data/digitsTrain.csv"
digits = get_data( link )

dim(digits)
```

```
## [1] 5000  785
```

```
# to avoid confusion with numeric indices and the name "1"
# index will now be i.ordiginal_index
rownames(digits) = sprintf( "i.%s", rownames(digits) )

# get the labels with their index name
get_y = function(df, y_index = 1) {
    setNames( df[, y_index], rownames(df) )
}

y = get_y(digits)
```

To save time, I saved distance matrices for 4 distance metrics.

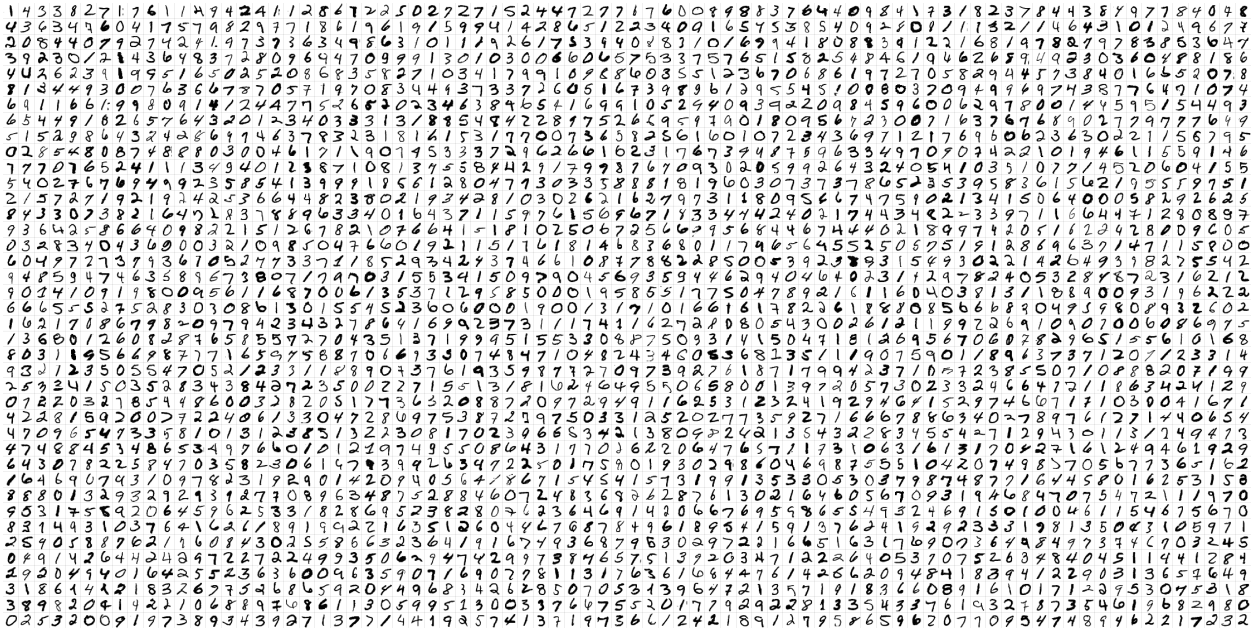
```
############################
# Saving distance matrices #
############################

dist_methods = c("euclidean", "maximum", "manhattan", "canberra")
filenames = sprintf("distance_matrix_%s.RData", dist_methods)

save_dist = function(df, filename, method)
{
    print( filename )

    # shuffle the rows
    df = df[ sample( nrow(df) ), ]
    dist_digits = as.matrix( dist( df[, -1], method = method) )

    print( sprintf("Saving %s....", filename) )
    cat("\n")
    save( dist_digits, file = filename )
}

# change to TRUE to create and save distance matrices for:
# "euclidean", "maximum", "manhattan", "canberra"
if(FALSE) invisible( mapply( function(file, method) {
    save_dist(digits, file, method)
    }, filenames, dist_methods ) )
```

Some provided functions for making plots.

```
######################
# Plotting functions #
######################

getImage =
function(vals)
{
    matrix(as.integer(vals), 28, 28, byrow = TRUE)
}

draw =
function(vals, colors = rgb((255:0)/255, (255:0)/255, (255:0)/255), ...)
{
    if(!is.matrix(vals))
        vals = getImage(vals)

    m = t(vals)  # transpose the image
    m = m[,nrow(m):1]  # turn up-side-down

    image(m, col = colors, ..., xaxt = "n", yaxt = "n")
}
```

Below is code for producing panel plots to explore the data and see many digits at once. The below image is more useful on a large monitor, but the small images still show a sample of the data and allows us to observe some of the peculiarities with the digit images.

```
save_plot = function (file_name, width = 1000, height = 600) {
    png(file_name, width = width, height = height, pointsize = 16)
    file_name
}

draw_digits =
    # draw digits by number of row by col
    # either randomly plot images, or plot images in rows_to_plot
function(row, col, df, row_given = FALSE, rows_to_plot = NULL)
{
    if(!row_given) {
        num = row * col
        # randomly select rows to plot
        rows_to_plot = sample( 1:nrow(df), num )
    }

    par(mfrow = c(row, col), mar = c(0, 0, 0, 0))
    invisible( sapply(rows_to_plot, function(r) draw(df[r, ]) ) )
}

row = 40
col = 80

image_1 = save_plot("./images/many_digits.png", col * 100, row * 100)

draw_digits(row, col, digits[, -1])

invisible( dev.off() )
grid.raster( readPNG(image_1) )
```



k-nearest neighbors and Cross Validation

```
############################
# Main functions for part 1 #
############################

get_cv_splits =
    # INPUT:
    #   - n: the number of rows in the data
    #   - folds: number of cross validation folds to create
    #
    # OUTPUT: list of length folds with indices for each CV fold
function(n, folds = 5)
{
    # get near equally sized folds, depending on n and folds
    # size differences will be at most 1
    cv_splits = sort( rep( 1:folds, ceiling( n/folds ) )[1:n] )

    split(1:n, cv_splits)
}


vote =
    # INPUT:
    #   - closest_points: a vector of labels
    #   - distances: vector of distances for corresponding labels
    #   - voting method: "simple" or "weight"
    #
    # OUTPUT: list with elements: k, prediction, probability, closest_distance, mean_distance
    #
    # DOC:
    #   - this function returns the predicted label given the inputs
    #   - ties are broken by selecting points based on probability proportional
    #   to the inverse of the distance
function(closest_points, distances, method = "weight")
{
    # use small epsilon to avoid division by zero
    epsilon = 10^(-10)

    counts =
    if( method == "weight" ) {
        # scale distances so total is length of distances
        scaled_dist = 1 / (distances + epsilon)
        scaled_dist = scaled_dist / sum(scaled_dist) * length(scaled_dist)
        scaled_dist

        # get total distances for each label in closest_points
        sapply( split( scaled_dist, closest_points ), sum )
    } else {
        # count the labels
        table( closest_points )
    }

    max_count = max( counts )
```

```
    # get all the points that have this max count
    most_frequent = names( counts[ counts == max_count ] )

    # get all the points and distances which occur most frequently
    mask = closest_points %in% most_frequent
    most_frequent = closest_points[mask]
    most_freq_dist = distances[mask]

    # compute a probability as the inverse of the distance
    prob_dist = 1 / (most_freq_dist + epsilon)
    prob_dist = prob_dist / sum(prob_dist) # make prob_dist sum to 1

    # to break tie, sample using the inverse distance probabilities
    # change to character so 3 is different from "3"
    winner = sample( as.character(most_frequent), 1, prob = prob_dist )

    # return list with the predicted value, probability, and distances
    list_names = c("k", "prediction", "probability", "closest_distance", "mean_distance")
    setNames( list( length(closest_points), winner, max_count / sum(counts),
                    min(distances), mean(distances) ), list_names )
}


nearest_neighbors =
    # INPUT:
    #   - row: a row of the distance matrix
    #   - labels: the label for each column of row
    #   - k_values: the values of k to use for prediction
    #   - vote_method: see vote()
    #
    # OUTPUT: data frame with prediction, probability, closest_distance, mean_distance
    # for each k in k_values
function(row, labels, k_values, vote_method = "simple")
{
    # order the distances in row, get the corresponding labels in correct order
    ordering = order(row)
    row = row[ordering]
    labels = labels[ordering]

    # for each k in k_values, get the predicted value (as well as other metrics)
    predictions = lapply( k_values, function(k) vote( labels[1:k], row[1:k], vote_method ) )

    # predictions is a list, combine the data into a matrix
    pred = do.call(rbind, predictions)

    if( length(k_values) == 1 )
        return( list( pred = pred, lab = labels[1:k_values] ) )

    pred

}

save_misclass =
```

```
    # save nearest neighbors to .csv file when all labels incorrect
function( df, labels, digit_name, label_true, filename = 'save_misclass' )
{
    if( df[,'probability'] != 1 ) return( NULL )

    pred = df[,'prediction']

    filename = paste0( filename, '_', length(labels), '.csv' )
    # create file and write header if file doesn't already exist
    if( !file.exists( 'save_misclass_3.csv' ) ) {
        tmp = paste( paste( "nearest", 1:length(labels), sep = "_" ), collapse = "," )
        header = sprintf( "digit,truth,%s,predicted", tmp )
        write( header, filename, append = TRUE )
    }

    if( pred != label_true ) {
        to_save = paste( c(digit_name, label_true, names(labels), pred), collapse = "," )
        write( to_save, filename, append = TRUE )
    }

    return( NULL )
}

cv_fold =
    # INPUT:
    #   - distance_mat: a distance matrix
    #   - labels: true labels for each row
    #   - hold_out: indices to hold out for test set
    #   - k_values: values to use for k
    #   - new_data / tts: only used with cv_fold_new_data wrapper function
    #
    # OUTPUT: a data frame
    # DOC: predicts the test set using the training set
function( distance_mat, labels, hold_out, k_values = 1:3, new_data = FALSE, tts = NULL )
{
    # if new_data is TRUE, we pass in fold, label_fold, label_true directly
    if(new_data) {
        fold = distance_mat
        label_fold = tts$y_train
        label_true = tts$y_test
    } else {
        # subset distance matrix and labels for a single cv fold
        fold = distance_mat[hold_out, -hold_out]
        label_fold = labels[ -hold_out ]
        label_true = labels[ hold_out ]
    }

    # making sure distance_mat colnames and label names match up
    stopifnot( all.equal( colnames(fold), names(label_fold) ) )

    # get predicted nearest neighbors
    predictions = apply( fold, 1, function(row) nearest_neighbors(row, label_fold, k_values) )
```

```
    if( length(k_values) == 1 ) {
        digit_name = names( predictions )

        # update predictions and save misclassifications to file
        predictions = mapply( function(p, n, label_t) {
            save_misclass( p$pred, p$lab, n, label_t )
            p$pred
        }, predictions, digit_name, label_true, SIMPLIFY = FALSE)
    }

    # making sure prediction names and true label names match up
    stopifnot( all.equal( names(predictions), names(label_true) ) )

    # stack all prediction data frames, adding columns for true label and image index
    predictions = as.data.frame( do.call(rbind, predictions) )
    predictions$y_true = rep( label_true, each = length(k_values) )
    predictions$image_index = rep( names( label_true ), each = length(k_values) )

    # several columns are lists, we'll unlist and convert to numeric vector
    cols_to_fix = c("k", "prediction", "probability", "closest_distance", "mean_distance")
    predictions[ ,cols_to_fix] = sapply( predictions[ ,cols_to_fix], function(col) {
        as.numeric( unlist(col) )
        })

    predictions
}


misclassification_rate = function(y_true, y_pred) {
    # return percent that don't match between y_true and y_pred
    mean( as.character(y_true) != as.character(y_pred) )
}


kNN =
    # INPUT:
    #   - filename: of a saved distance matrix
    #   - y_true: the labels for the columns of the distance matrix
    #   - cv_function: function to use for cross validation
    #   - k: range of k values to use
    #   - new_data/df/method: used with kNN_new_data wrapper function
    #
    # OUTPUT:
    #   - df: data frame of predictions (and other metrics like probabilies and distances)
    #   - nrow(df) = nrow(original data) * length(k); i.e. 5000 * 20 = 10000
    #
    # DOC: main function for finding kNN
function(filename, y_true, cv_function = cv_fold, k = 1:20, cv_folds = 5,
         new_data = FALSE, df = NULL, method = NULL)
{
    # pred is list of data frames (with length of cv_folds)
    pred =
    if(new_data) {
```

```
        # shuffle the rows, just in case
        df = df[ sample( nrow(df) ), ]

        # get cross validation train/test splits and perform predictions
        cv_splits = get_cv_splits( nrow(df), cv_folds )
        lapply( cv_splits, function(rows) cv_function( df, rows, method, k ) )
    } else {
        print( filename )

        # loads a distance matrix object called dist_digits
        load( filename )

        # arrange labels in the same order as columns from shuffled distance matrix
        y_true = y_true[ colnames(dist_digits) ]
        stopifnot( all.equal( names(y_true), colnames(dist_digits) ) )

        # get cross validation train/test splits and perform predictions
        cv_splits = get_cv_splits( nrow(dist_digits), cv_folds )
        lapply( cv_splits, function(rows) cv_function( dist_digits, y_true, rows, k ) )
    }

    # combine into a single data frame and add column for a correct prediction
    pred = do.call(rbind, pred)
    pred$correct = as.numeric( pred$prediction == pred$y_true )

    pred
}
```

We'll need several other similar functions, which perform the same steps as above without first computing an entire distance matrix. These will be useful for computing the cosine similarity (which isn't included with the dist function), for any function that requires making changed based on the training data for each fold (i.e. for step 2), and for avoiding unnecessary distance calculation between vectors we don't need (again for step 2). We'll use the *proxy* package which also has a dist function. It's slightly slower than the original dist function, but we overcome this speed difference by computing fewer unused distance calculations.

```
###################################
# Additional functions for part 1 #
###################################

train_test_split =
    # INPUT: train and test data frames
    # OUTPUT: 4 element list: X_train, X_test, y_train, y_test
    # DOC: function assumes y is first column, can specify with y_index
function(train, test, y_index = 1)
{
    # gets y_train and y_test while retaining the row name
    tts = list( train[, -y_index], test[, -y_index], get_y( train ), get_y( test ) )
    setNames(tts, c("X_train", "X_test", "y_train", "y_test"))
}


cv_fold_new_data =
    # INPUT: a data frame and a distance metric
    # DOC: calls cv_fold but takes a data frame instead of a distance matrix
function(df, hold_out, method, k_values = 1:3)
{
    tts = train_test_split( df[ -hold_out, ], df[ hold_out, ] )

    fold =
    if( method == "cosine" ) {
        # subtract 1 since being near 1 indicates vectors are near each other
        # we can still find min distance as done with other distance metrics
        1 - proxy::simil( tts$X_test, tts$X_train, method = method)
    } else if ( method %in% c("euclidean", "manhattan", "canberra") ) {
        proxy::dist( tts$X_test, tts$X_train, method = method)
    } else {
        stop( "No method specified, please select: cosine, euclidean, manhattan, or canberra" )
    }

    cv_fold( fold, NULL, hold_out, k_values, new_data = TRUE, tts = tts )
}


kNN_new_data =
    # wrapper for kNN, we can pass a data frame and distance metric rather than a distance matrix
function(df, method, cv_function, k = 1:20, cv_folds = 5)
{
    kNN("", NULL, cv_function, k, cv_folds, new_data = TRUE, df = df, method = method)
}
```

1. Report the best model, i.e., value of k and metric.

First we'll search the distance metrics: "euclidean", "maximum", "manhattan", "canberra", and "cosine" using k from 1 to 20. Each distance metric returns a data frame with predictions, probabilities for those predictions, and distances to the nearest training image.

```
############################
# 1. Report the best model #
############################
```

```r
k = 1:20
pred_list = lapply( filenames, function(file) kNN(file, y, cv_fold, k) )
```

```
## [1] "distance_matrix_euclidean.RData"
## [1] "distance_matrix_maximum.RData"
## [1] "distance_matrix_manhattan.RData"
## [1] "distance_matrix_canberra.RData"
```

```r
names(pred_list) = dist_methods

# adding cosine similarity
cosine_df = kNN_new_data(digits, "cosine", cv_fold_new_data)
pred_list$cosine = cosine_df

# looking at the overall results
sapply(pred_list, dim)
```

```
##      euclidean maximum manhattan canberra cosine
## [1,]    100000  100000    100000   100000 100000
## [2,]         8       8         8        8      8
```

```r
sapply(pred_list, function(df) 1 - mean(df$correct))
```

```
## euclidean   maximum manhattan  canberra    cosine
##   0.07601   0.37791   0.08846   0.07498   0.06105
```

Looking at the overall misclassifcation rate (that is the mean using all values of k for each distance metric) we see that cosine similarity is marginally the best, while maximum is not good at all. For this problem, it makes sense that maximum would perform much worse, since all the values are scaled to fall between 0 and 255, and we're more likely to have many similar maximum difference and hence unable to differentiate between different images. For this reason, we'll remove maximum from the remainder of this analysis.

From each data frame, we first extract the misclassifcation rate for each value of k for each distance metric. We then find the minimum to determine which distance metric and k value is best. We find the cosine similarity with $k = 3$ is best, with euclidean with $k = 3$ as second best (and the best minkowski distance metric).

```r
pred_list$maximum = NULL

misclass =
    # misclassification rate for a data frame and a vector of k values
function(df, k)
{
    missed = sapply(k, function(k_val) 1 - mean( df[ df$k == k_val,  ]$correct ) )
    setNames(missed, k)
}

# misclassification matrix, rows are k values, columns are distance metrics
misclass_mat = sapply( pred_list, function(df) misclass(df, k) )

# smallest misclassifcation rates by distance metric
```

```
smallest_misclass = apply(misclass_mat, 2, function(col) c( min(col), which.min(col) ) )
smallest_misclass = smallest_misclass[ ,order(smallest_misclass[1, ]) ]
rownames(smallest_misclass) = c("misclassification", "k")
smallest_misclass
```

```
##                   cosine euclidean canberra manhattan
## misclassification 0.0538    0.0646   0.0692    0.0732
## k                 3.0000    3.0000   6.0000    3.0000
```

Another method for achieving the "best" model is to refuse to classify images we aren't confident in. The kNN output data frame includes "probabilities." For example if the labels were "3", "3", "1", the probability would be 2/3 as a "3". If we take the simplest approach and only predict with a probability greater than 0.5, we could achieve a misclassification rate of ~4.6% by skipping ~60 images (cosine with k = 3), or even better a misclassification rate of ~2.2% by skipping ~430 images (cosine with k = 2). The choice at this point would depend on how we plan to use the image classifications. Note, the table shows probabilities as rows and 0/1 (incorrect / correct) prediction as columns.

```
best_misclass =
    # for computing misclassifcation for k = 2 or 3
    # while leaving out uncertain predictions
function(df, k)
{
    # get the rows where k == (input k)
    df = df[ df$k == k, ]
    mat = as.matrix( table( df$probability, df$correct ) )

    # we'll keep rows with probability greater than 0.5
    rows_to_drop = as.numeric( rownames(mat) ) <= 0.5
    not_classified = sum( mat[rows_to_drop, ] )

    # compute new misclassification rate
    mat_2 = mat[-1, ,drop = FALSE]
    misclass = 1 - sum(mat_2[ ,"1"]) / sum(mat_2)

    list_names = c("table", "not_classified", "misclassification_rate")
    setNames( list(mat, not_classified, misclass), list_names )
}

best_misclass( pred_list$cosine, 3 )
```

```
## $table
##
##                      0    1
##   0.333333333333333   42   19
##   0.666666666666667  165  465
##   1                   62 4247
##
## $not_classified
## [1] 61
##
## $misclassification_rate
## [1] 0.04596072
```

```
best_misclass( pred_list$cosine, 2 )
```

```
## $table
##
##          0    1
##   0.5  209  219
##   1    100 4472
##
## $not_classified
## [1] 428
##
## $misclassification_rate
## [1] 0.02187227
```

Let's also compare incorrect vs correct classifications in terms of average distance from the test points. We see that the misclassified images are on average always further away from the correctly classified images. This is a great sign, as it follows the theory behind the k-nearest neighbor classifier.

```
average_distance = function(df, k) {
    df = df[ df$k == k, ]

    # split by correct 0/1 and then compute of the mean of the mean_distance column
    ave_dist = by(df, df$correct, function(df) mean( df$mean_distance ) )
    setNames( as.numeric(ave_dist), c("0", "1") )
}

sapply( pred_list, function(df) average_distance(df, 3) )
```

```
##   euclidean manhattan canberra    cosine
## 0  1588.516  15710.50 400.3070 0.2347710
## 1  1336.084  12572.26 344.3477 0.1609859
```

Finally, we could create a quick ensemble classifier using our two best models from above. Simply, we'll only classify images in which both models agree, otherwise we'll skip. Using this method gives a misclassification rate of ~3% while skipping ~270 images.

```
get_predictions =
    # retreives the specified distance metric and k value from the list of data frames
function(df_list, method, k, all_columns = FALSE)
{
    df = df_list[[method]]
    df = df[ df$k == k, ]

    if(all_columns) {
        return( df )
    } else {
        return( df[, c("prediction", "y_true", "image_index")] )
    }
}

# combining the two best models
model_1 = get_predictions(pred_list, "cosine", 3)
```

```
model_2 = get_predictions(pred_list, "euclidean", 3)

# merge on "image_index", "y_true"
model_merge = merge(model_1, model_2, by = c("image_index", "y_true"))

# if both models predict the same value that is our prediction, otherwise skip
same_pred = model_merge$prediction.x == model_merge$prediction.y
model_merge$ensemble = ifelse( same_pred, model_merge$prediction.x, NA )
to_pred = !is.na(model_merge$ensemble)

# misclassification rate of the ensemble classifier
misclassification_rate( model_merge$y_true[to_pred], model_merge$ensemble[to_pred] )
```

```
## [1] 0.03043754
```

```
sum(!to_pred)
```

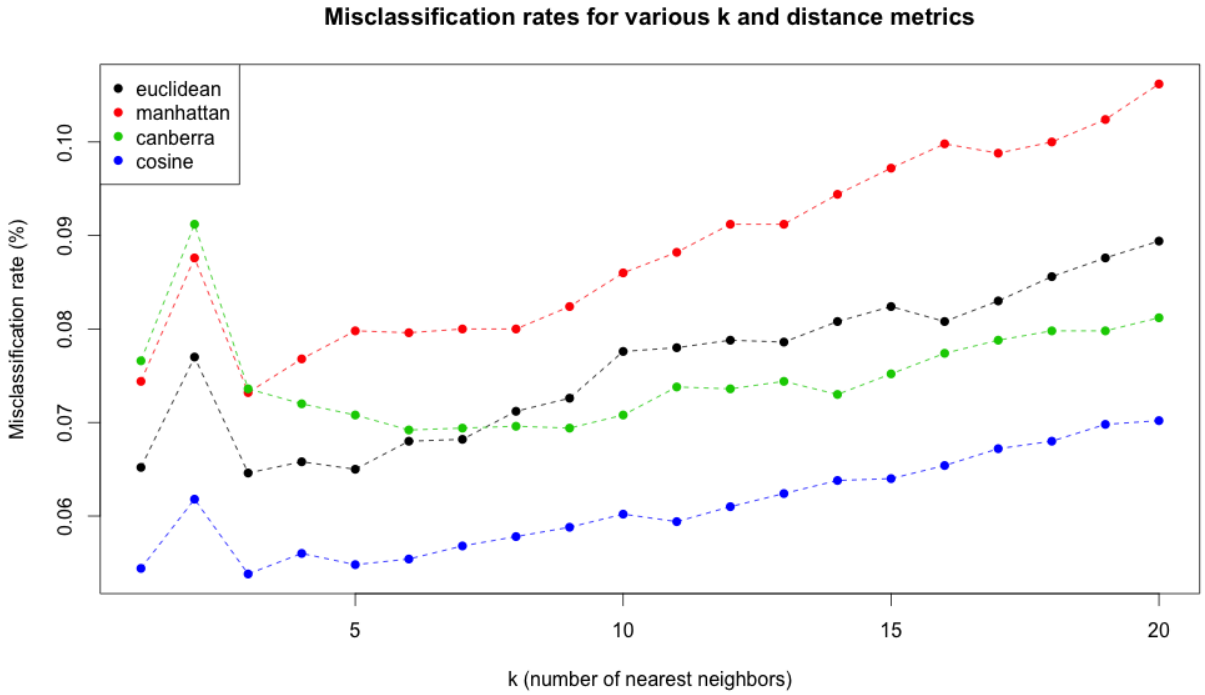
```
## [1] 269
```

2. Draw a plot showing the overall cross-validation misclassification rate versus k and the distance metrics.

We can see from the plot that cosine similarity and euclidean are the two best distance metrics, as well as the values of k that minimize the misclassification rates. There is a strange looking local maximum at k = 2 for each of the distance metric. By looking back at one of the tables from number 1 (recreated below), we see that if both points are the same label, then k = 2 does very well. However, about 10% of the time the labels differ, meaning that we'll have to use our tie-breaker which is essentially a coin flip.

```
####################################
# 2. Draw a misclassification plot #
####################################

image_2 = save_plot("./images/misclassification.png")

plot( k, seq_along(k), type = "n", ylim = range(misclass_mat),
      xlab = "k (number of nearest neighbors)", ylab = "Misclassification rate (%)",
      main = "Misclassification rates for various k and distance metrics")

plot_misclass = function(df, k) {
    # in case we get a matrix
    df = as.data.frame(df)

    # colors for the plot
    colors = 1:ncol(df)

    mapply( function(values, color) {
        points( k, values, pch = 16, col = color)
        points( k, values, type = "l", lwd = 1, lty = 2, col = color)
        }, df, colors )

    legend("topleft", legend = colnames(df), col = colors, pch = 16)
}
```

```
plot_misclass(misclass_mat, k)

invisible( dev.off() )
grid.raster( readPNG(image_2) )
```




```
# table used for explaining peak at k = 2
best_misclass( pred_list$cosine, 2 )$table
```

```
##
##          0    1
##   0.5  209  219
##   1    100 4472
```

3. Calculate the confusion matrix for the training set using the chosen value of k and metric.

The confusion matrix and misclassification by digit is shown below. Note, the misclassification by digit should be read: given the true digit is i, the misclassifcation rate of i is shown

```
########################
# 3. Confusion matrix #
########################

confusion_matrix =
    # INPUT: a list of data frames, distance metric, and k
    # OUTPUT: 2 element list: confusion_matrix and misclassification_by_digit
    # DOC: wrapper for confusion_matrix_helper
function(df_list, method, k)
{
```

```
    # get the df for method and rows where k == (input k)
    df = df_list[[method]]
    df = df[ df$k == k, ]

    confusion_matrix_helper(df$y_true, df$prediction)
}


confusion_matrix_helper =
    # INPUT: vector of the true classification and the predicted values
    # OUTPUT: 2 element list: confusion_matrix and misclassification_by_digit
function(y_true, y_pred, values = 0:9)
{
    # create factors with all digits as levels, in case of a missing digit in y_true/y_pred
    y_true = factor( y_true, levels = values )
    y_pred = factor( y_pred, levels = values )
    confusion_mat = table( y_true, y_pred )

    # find misclassification by true digit
    correct = diag(confusion_mat)
    row_sums = apply(confusion_mat, 1, sum)
    misclass_by_digit = sort( 1 - (correct / row_sums) )

    list_names = c("confusion_matrix", "misclassification_by_digit")
    setNames( list( confusion_mat, round(misclass_by_digit, 4) ),  list_names )
}

cc = confusion_matrix(pred_list, "cosine", 3); cc
```

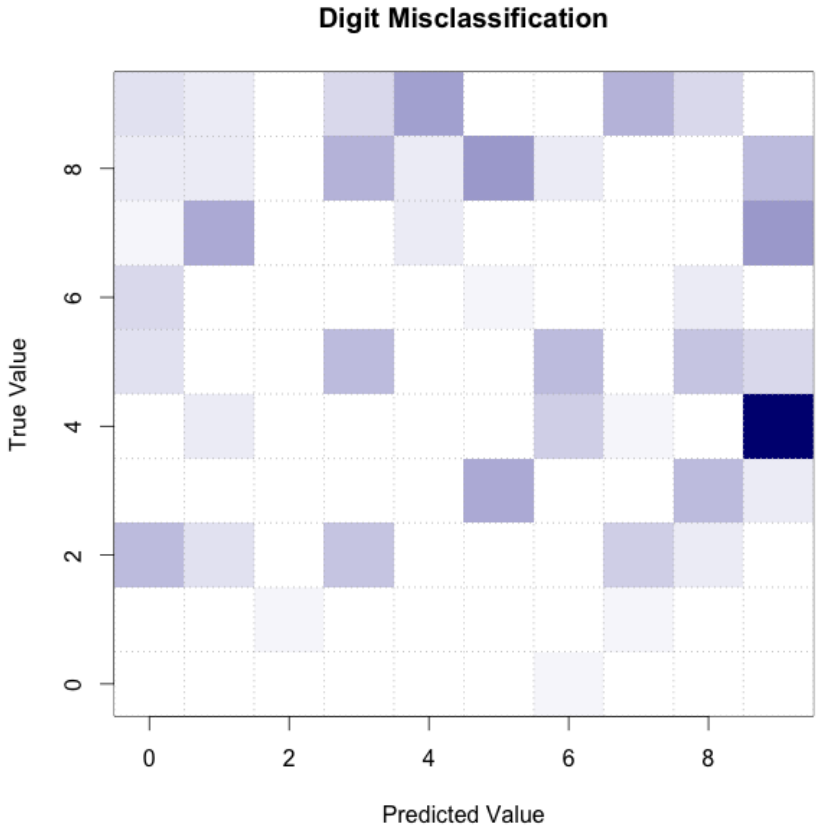
```
## $confusion_matrix
##       y_pred
## y_true   0   1   2   3   4   5   6   7   8   9
##      0 488   0   0   0   0   0   2   0   0   0
##      1   1 587   2   1   1   0   0   2   0   0
##      2   8   4 483   7   1   0   1   6   3   1
##      3   1   0   1 498   0  10   1   1   8   3
##      4   1   3   0   0 429   0   6   2   1  34
##      5   4   1   0   8   0 415   8   0   7   5
##      6   5   1   1   0   1   2 483   0   3   0
##      7   2  10   1   0   3   1   0 525   0  12
##      8   3   3   1   9   3  12   3   1 383   8
##      9   4   3   1   5  11   1   0   9   5 440
##
## $misclassification_by_digit
##      0      1      6      3      7      2      5      9      4      8
## 0.0041 0.0118 0.0262 0.0478 0.0523 0.0603 0.0737 0.0814 0.0987 0.1009
```

```
confusion_matrix(pred_list, "euclidean", 3)
```

```
## $confusion_matrix
##       y_pred
## y_true   0   1   2   3   4   5   6   7   8   9
```

```
##      0 487   0   0   0   0   0   3   0   0   0
##      1   0 588   1   1   1   0   1   2   0   0
##      2   6  15 461   5   3   0   4  13   3   4
##      3   1   2   5 491   2   8   0   1   6   7
##      4   1   9   0   0 430   0   5   3   0  28
##      5   1   3   0  15   1 411   8   0   4   5
##      6   6   3   1   0   1   5 478   0   2   0
##      7   0  13   0   0   3   1   0 528   0   9
##      8   3  12   1  13   2  16   2   4 360  13
##      9   3   2   1   6   9   0   0  13   2 443
##
## $misclassification_by_digit
##      0      1      6      7      3      9      5      4      2      8
## 0.0061 0.0101 0.0363 0.0469 0.0612 0.0752 0.0826 0.0966 0.1031 0.1549
```

```
confusion = cc$confusion_matrix

image = save_plot("./images/digit_misclass_color.png", width = 600, height = 600)

diag(confusion) = 0L
par( mfrow = c(1, 1), mar = c(4, 5, 4, 1) )
color_fun = colorRampPalette( c("white", "navy") )
image(0:9, 0:9, t(confusion), col = color_fun( max(confusion) ), ylab = "True Value",
      xlab = "Predicted Value", main = "Digit Misclassification")
p = seq(-.5, by = 1, length = 11)
abline(v = p, h = p,  col = "gray", lty = 3)

invisible( dev.off() )
grid.raster( readPNG(image) )
```



4. Which digits were generally classified best? worst?

Looking at the confusion matrix for cosine from part 3, we see that the digits 4, 8, 5, and 9 were mis-classified worse, while the digits 0, 1, and 6 were classified best. Interestingly, cosine and euclidean were better/worse at classifying different digits. Cosine was better with 8s and 2s, while euclidean was better with 9s and 7s.

5. Which digits were generally confused with others?

Again looking at the confusion matrix for cosine from part 3, when the true image was a 4, we often confused the image with a 9. Some of the other common error were 8s being confused for 3s (and vice versa), 7s being confused with 9s, and 5s being confused with 6s. These confusions all follow our intuition, as the digits which appear most similar to one another.

6. Show some of the digits that were misclassified that were difficult for a human to classify. Suggest why these were misclassified.

We'll identify digits in which the three nearest neighbors are all the same and we have a misclassification. We found 73 such digits and they are shown in the first plot below. In the next plot, we'll highlight the original digit with a red box and show the three nearest neighbors (plotted as the next three images to the right of each red box). For identification purposes, the upper left corner of each image shows the true digit label. These residuals give insight into why some of these digits were misclassified using this algorithm.

```
########################
# 6. Show some digits #
########################

euclid_df = kNN( "distance_matrix_euclidean.RData", y, cv_fold, k = 3 )
```

```
## [1] "distance_matrix_euclidean.RData"
```

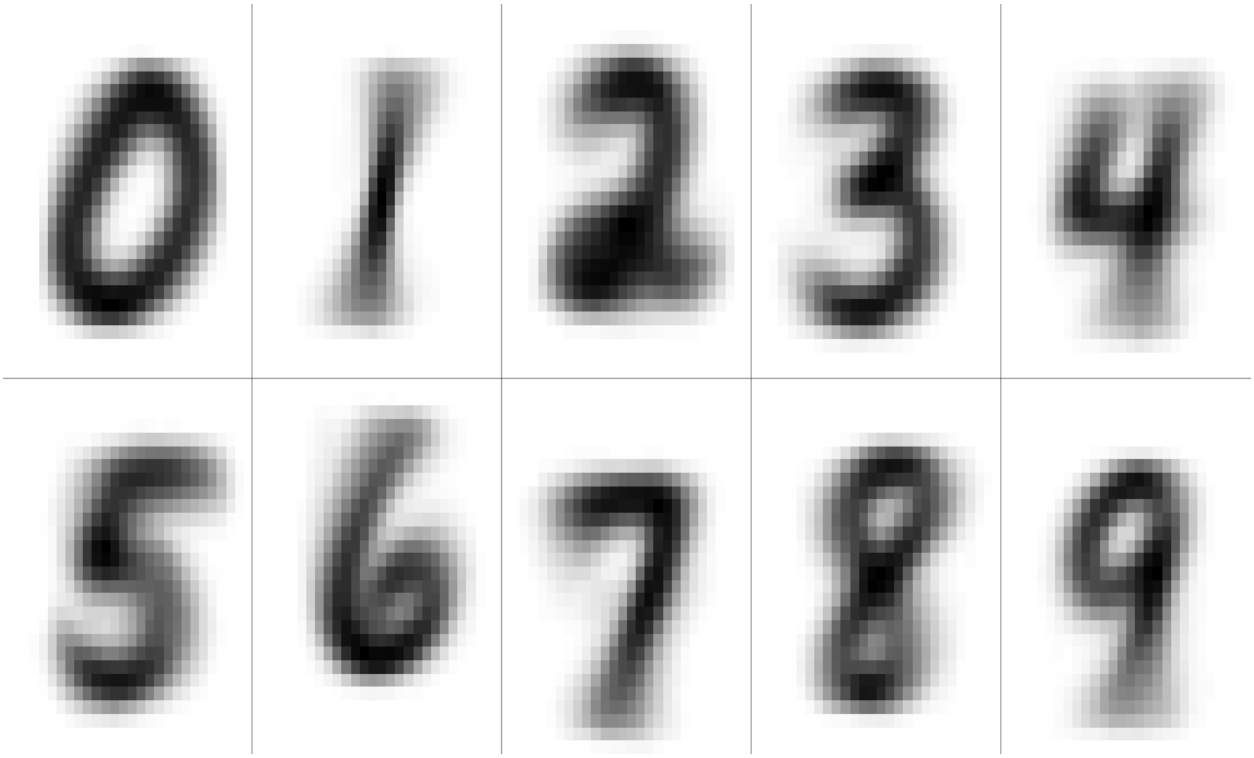
```
# read in digits for residual plot, removing duplicates
misclass_df = read.csv( 'save_misclass_3.csv', stringsAsFactors = FALSE )
misclass_df = misclass_df[ !duplicated( misclass_df$digit  ), ]

# order by truth digits
ordering = order( misclass_df$truth )
misclass_df = misclass_df[ordering,]

# parameters for label locations
u = par()$usr
x0 = u[1] + .1*(u[2] - u[1])
y0 = u[3] + .9*(u[4] - u[3])

image = save_plot("./images/misclass_digits.png")

# plot mis-classified digits
par(mfrow = c(10, 12), mar = rep(0, 4))
invisible( apply( digits[misclass_df$digit,], 1, function(row) {
    label = row[1]
    draw( row[-1] ) # remove prediction column
    u = par()$usr
    x0 = u[1] + .1*(u[2] - u[1])
    y0 = u[3] + .9*(u[4] - u[3])
    text(x0, y0, label, col = "red", cex = .75, adj = .1)
}) )

invisible( dev.off() )
grid.raster( readPNG(image) )
```

```
image = save_plot("./images/misclass_digits_res.png", width = 1000, height = 1000)

# plot mis-classified digits showing residuals
par(mfrow = c(18, 16), mar = rep(0, 4), pty = "s")
rows = sort( sample( 118, size = 72 ) )
invisible( apply( misclass_df[rows,], 1, function(row) {

    x = digits[ row['digit'], ]

    # plot red box around original image
    draw( x[-1] )
    box( col = "red", lwd = 2 )
    text( x0, y0, x[1], cex = .75, adj = .1 )

    # plot nearest neighbors
    ids = row[ grep( '^nearest', names(row) ) ]
    mapply( function(i, label ) {
        draw( digits[i, -1] )
        text( x0, y0, label, cex = .75, adj = .1 )
    }, ids, row['predicted'] )
} ) )

invisible( dev.off() )
grid.raster( readPNG(image) )
```

Distance to Average

First we'll find and graph all the average digits.

```
######################
# Distance to Average #
######################

average_by_digit = function(df) {
    # split by label and compute column means for each split
    ave_digit = as.list( by(df, df[ ,1], colMeans) )

    # combine results as a matrix, 10 by 785
    do.call(rbind, ave_digit)
}

ave_digit = average_by_digit( digits )

image_4 = save_plot("./images/average_digits.png")

draw_digits(row = 2, col = 5, ave_digit[, -1], TRUE, 1:10)

invisible( dev.off() )
grid.raster( readPNG(image_4) )
```



Then we'll write a replacement cv_fold_new_data function, which will compute the average digits on the training set. We can then find the distance from these average digits to each test point in order to make predictions, using k = 1 only. The misclassification rate is considerably higher than before at ~18% for cosine and ~19% for euclidean. This misclassification rate is still very good considering we've reduced our training set from 4000 rows to 10.

```
cv_fold_new_data =
    # INPUT:
    #   - a data frame and a distance metric
    #   - fold_type is either "new_data", "average", or "variance"
    #
    # DOC: calls cv_fold but takes a data frame instead of a distance matrix
function(df, hold_out, method, k_values = 1:3, fold_type = "")
{
    tts = train_test_split( df[ -hold_out, ], df[ hold_out, ] )

    # we make update to tts if "average" or "variance"" is specified
    if(fold_type == "average") {
        # get the average by digit
        training_df = average_by_digit( df[ -hold_out, ] )
        tts = train_test_split( training_df, df[ hold_out, ] )
    } else if(fold_type == "variance") {
        # remove columns with zero variance
        col_to_keep = !apply( tts$X_train, 2, function(col) var(col) == 0 )
        tts$X_train = tts$X_train[ ,col_to_keep]
        tts$X_test = tts$X_test[ ,col_to_keep]
    }

    fold =
    if( method == "cosine" ) {
        # subtract 1 since being near 1 indicated vectors "near" each other
        # we can still find min distance as done with other distance metrics
        1 - proxy::simil( tts$X_test, tts$X_train, method = method)
    } else if ( method %in% c("euclidean", "manhattan", "canberra") ) {
        proxy::dist( tts$X_test, tts$X_train, method = method)
    } else {
        stop( "No method specified, please select: cosine, euclidean, manhattan, or canberra" )
    }

    cv_fold( fold, NULL, hold_out, k_values, new_data = TRUE, tts = tts )
}


# wrapper for cv_fold_new_data
cv_fold_average = function(df, hold_out, method, k_values = 1:3) {
    cv_fold_new_data(df, hold_out, method, k_values, "average")
}


get_average_df = function(df, method) {
    kNN_new_data(df, method, cv_fold_average, k = 1)
}

dist_methods = c("cosine", "euclidean", "manhattan", "canberra")

ave_df_list = lapply( dist_methods, function(method) get_average_df(digits, method) )
names(ave_df_list) = dist_methods

sapply(ave_df_list, function(df) misclass(df, 1))
```

```
##    cosine.1 euclidean.1 manhattan.1  canberra.1
##      0.1896      0.1898      0.3416      0.4418
```

To further explore the reason for a higher misclassification rate, we can again look at the average distance between test images and the average training images. Below are the distances we found before in step 1, and those for the digit averages. We see that the digit averages are much further away than those found in step 1.

```
# new average distances
new_ave_dist = sapply( ave_df_list, function(df) average_distance(df, 1) )
new_ave_dist
```

```
##      cosine euclidean manhattan canberra
## 0 0.3330581  1726.100  23740.74 636.0543
## 1 0.2536501  1573.387  22962.02 623.7055
```

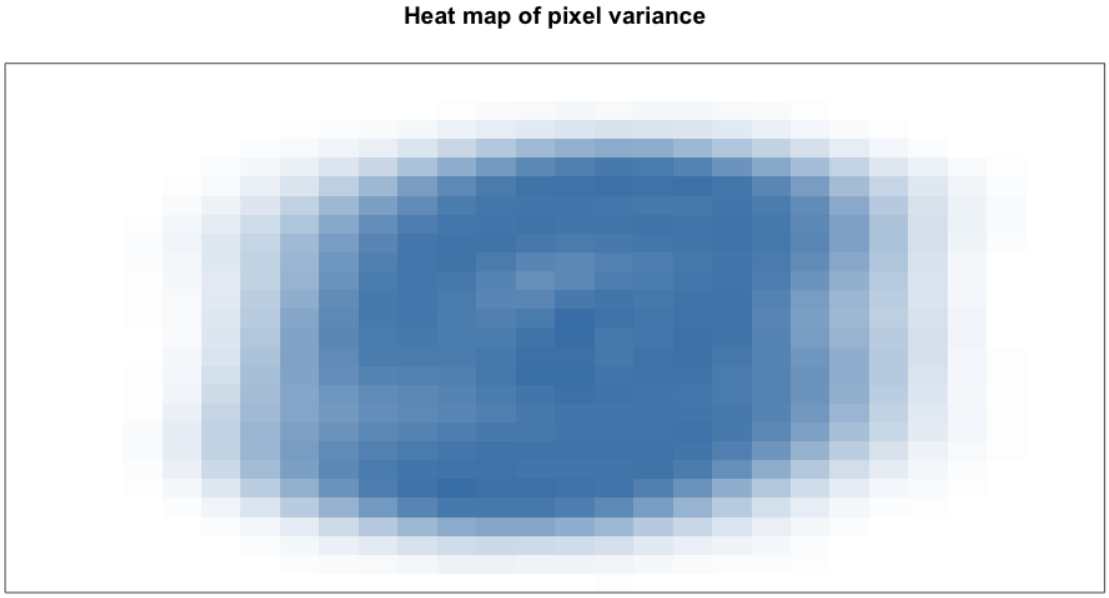
```
# original average distances
sapply( pred_list, function(df) average_distance(df, 1) )[, colnames(new_ave_dist)]
```

```
##      cosine euclidean manhattan canberra
## 0 0.2149992  1539.514  14994.00 387.3822
## 1 0.1435026  1254.497  11568.39 326.8577
```

Further Exploration - Dimensionality Reduction

As a last step, we'll explore the idea of remove pixels with zero variance in hopes of avoiding the curse of dimensionality problem. Let's first create a heat-map of the variance of the pixels.

```
############################
# Dimensionality Reduction #
############################

variance = apply(digits[, -1], 2, var)

image_5 = save_plot("./images/variance_heat_map.png")

draw(variance, colors = alpha("steelblue", seq(0, 1, 0.01)))
title(main = "Heat map of pixel variance")

invisible( dev.off() )
grid.raster( readPNG(image_5) )
```




```
sum( variance == 0 )
```

```
## [1] 131
```

Based on this plot, there are likely many pixels that don't contribute to the classification predication. With the whole data set, we find 131 pixels with zero variance. We'll create a final cv_fold wrapper function in order to remove columns from the training set which have zero variance, and remove those same columns from the test set. For simplicity, we'll only fit the two best models from before, cosine and euclidean both with k = 3. The results show that we can actually slightly improve our misclassifcation rate by removing unused pixels.

```
# wrapper for cv_fold_new_data
cv_fold_variance = function(df, hold_out, method, k_values = 1:3) {
    cv_fold_new_data(df, hold_out, method, k_values, "variance")
}

k = 3
method_names = c("cosine", "euclidean")

var_df_list = lapply( method_names, function(method) {
    kNN_new_data(digits, method, cv_fold_variance, k = k)
    })

# # misclassification rates with pixels removed
setNames( sapply(var_df_list, function(df) misclass(df, k)), method_names )
```

```
##    cosine euclidean
##    0.0558    0.0646
```

```r
# original misclassification rates
smallest_misclass["misclassification", c("cosine", "euclidean")]
```

```
##    cosine euclidean
##    0.0538    0.0646
```