# CS5785 Homework 1: Due by 12:59 PM EST - Thursday September 26, 2019

Eduardo Castillo (ec833) and Shobhna Jayaraman (sj747)

September 27, 2019

**Executive Summary**

This assignment introduces students to a real-world application of *Machine Learning* (ML) techniques such as K-nearest neighbor and logistic regression. These applications are explored from a fundamental implementation level, –where basic linear algebra and statistical methods are directly applied by students to create functional predictive models.

In addition to ML programming exercises, problems in the *Probability and Statistics* space are presented to further explore and reinforce the fundamental principles behind these techniques.

**Problem 1 - Digit Recognizer**

Solutions generated using Jupyter are found in Attachment 1 of this report.

**Problem 2 - Titanic Machine Learning Disaster**

**(a) Join the Titanic: Machine Learning From Disaster competition on Kaggle. Download the training and test data.**

Answer: We joined the Kaggle competition under the name “ec833sj747” and downloaded the training and testing data.

**(b) Using logistic regression, try to predict whether a passenger survived the disaster. You can choose the features (or combinations of features) you would like to use or ignore, provided you justify your reasoning.**

**SOLUTION:** Plots which helped predict which features to choose/drop.

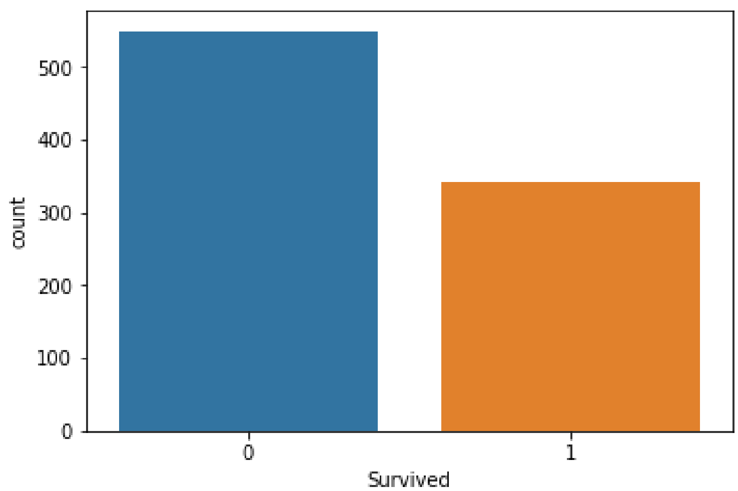
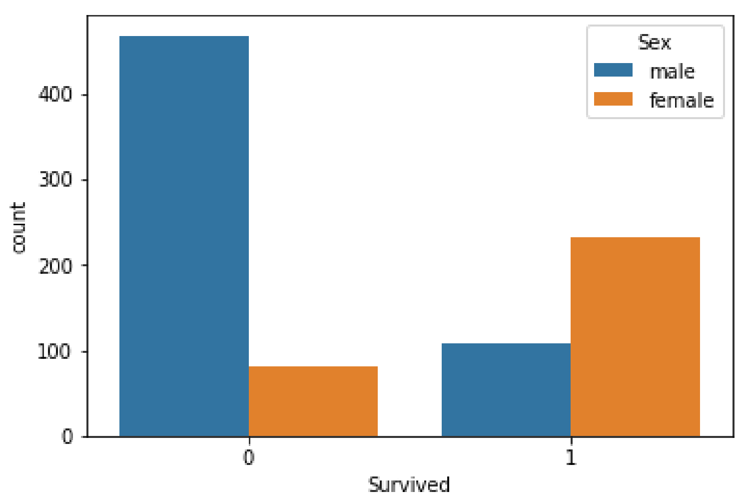
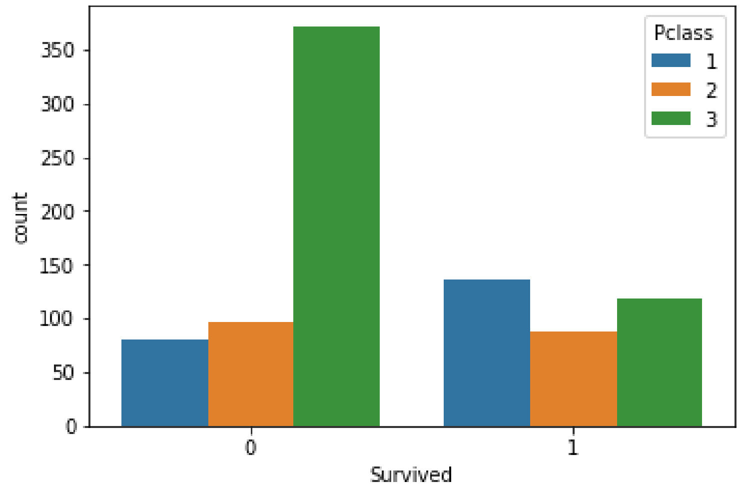



Figure 1: Barplot showing number of survivors



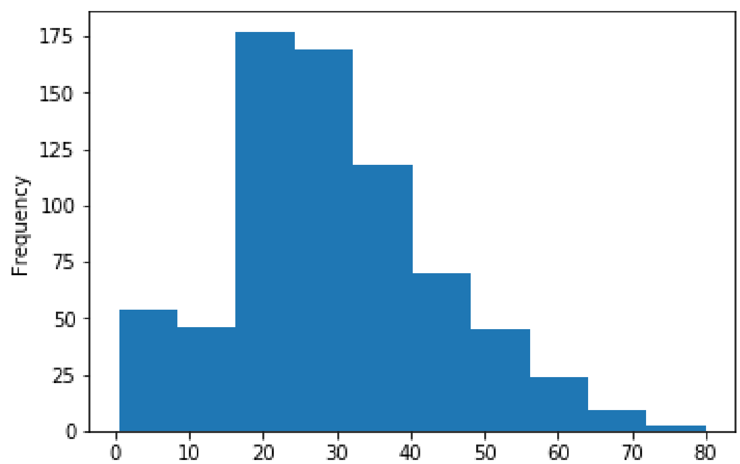

(1).png

Figure 2: Barplot showing relationship between gender and survival



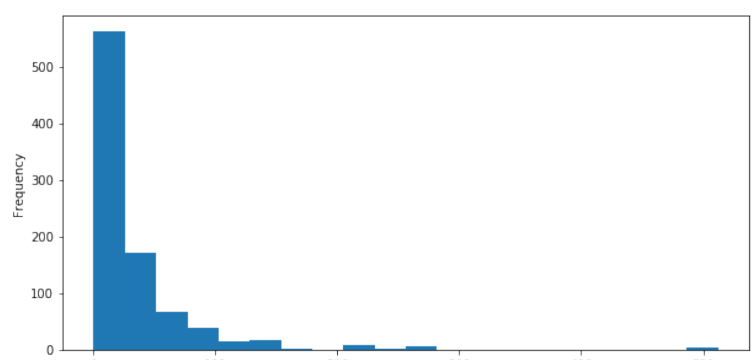

(2).png

Figure 3: Barplot showing relationshiop between passenger class and survival



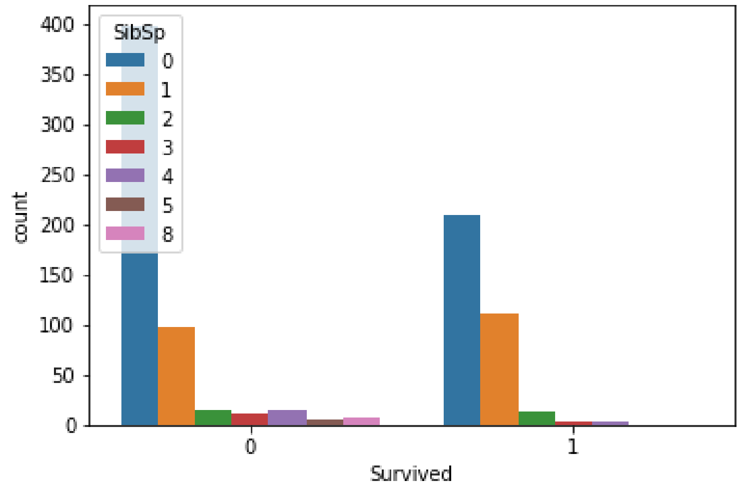

(3).png

Figure 4: Barplot showing relationship between Age and survival



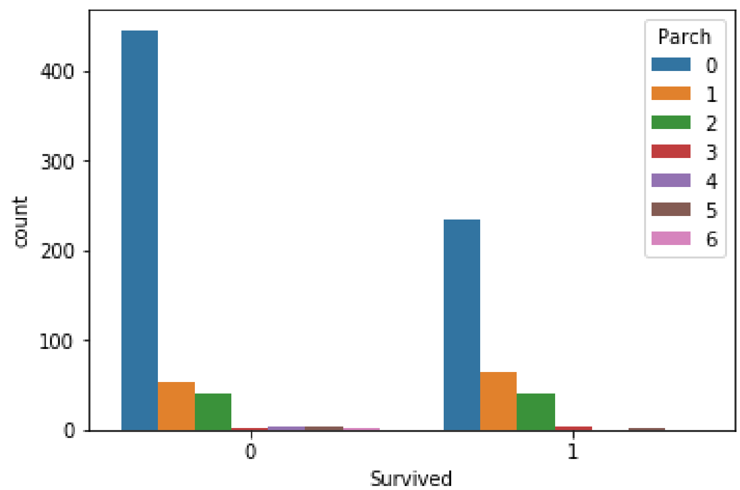

(4).png

Figure 5: Barplot showing relationship between Fare and survival



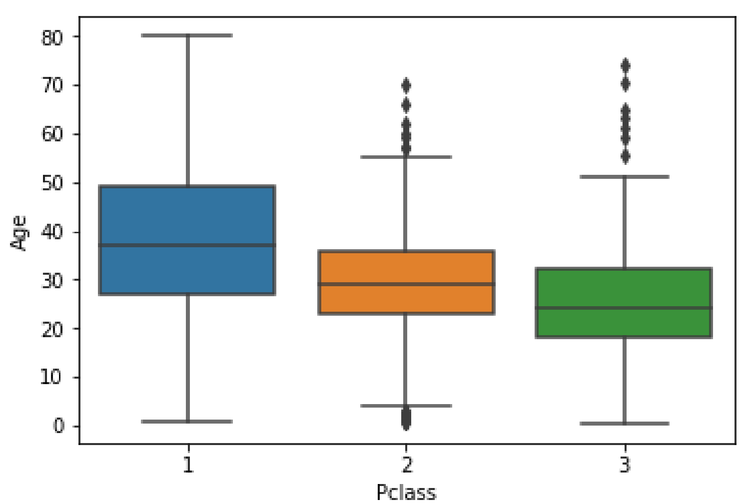

(5).png

Figure 6: Barplot showing relationship between Siblings and survival




(6).png

Figure 7: Barplot showing relationship between Parents and survival




(7).png

Figure 8: Relationship between Passenger Class and Age (we can see that people in first class are older generally)

**(c) Train your classifier using all of the training data, and test it using the testing data. Submit your results to Kaggle.**

**SOLUTION:** Submitted on Kaggle under the team name "ec833sj747". Please see attached notebook (.ipynb) file for code.

**EXERCISE 1. Variance of a sum. Show that the variance of a sum is var [X-Y ] = var [X] + var [Y]-2 cov [X,Y ], where cov [X,Y] is the covariance between random variables X and Y.**

**SOLUTION:**

Proof:

var[X-Y] = E $((X-Y)-E(X-Y)^2$

We know,

var [X] = E $(X-E(X))^2$

var [Y] = E $(Y-E(Y))^2$

cov[X,Y] = (X - E (X) ) ( Y - E (Y) )

Since E (X-Y) = E (X) - E (Y)

var (X-Y) = E $((X-Y)-(E(X)-E(Y)))^2$

=E $((X-E(X))-(Y-E(Y)))^2$

=E $((X-E(X))-(Y-E(Y)))^2$

=E ( $(X-E(X))^2+(Y-E(Y))^2$ - E ( 2 ( X - E (X) (Y - E (Y) ) )

=E $(X-E(X))^2$ + E $(Y-E(Y))^2$ - 2E ( (X - E (X) (Y - E (Y) ) )

=var [X] + var[Y] - 2 cov[X,Y]

Therefore, var [X-Y] = var [X]+ var [Y] 2 cov [X,Y]

**EXERCISE 2. Bayes rule for quality control. You're the foreman at a factorymaking ten million widgets per year. As a quality control step before shipment, you create a detector that tests for defective widgets before sending them to customers. The test is uniformly 95 % accurate, meaning that the probability of testing positive given that the widget is defective is 0.95, as is the probability of testing negative given that the widget is not defective. Further, only one in 100,000 widgets is actually defective. (a) Suppose the test shows that a widget is defective. What are the chances that it's actually defective given the test result?**

**SOLUTION:**

According to Bayes Theorem: **P(A|B) = P(B|A) P(A) / P(B)**

$$\frac{P(A|B) * P(B)}{P(A) * P(A|B) + P(B) * P(A|B)}$$

$$= \frac{0.95*0.00001}{0.95*0.00001 + 0.05*0.9999}$$

= 0.000189

(8).png

Figure 9: Bayes rule applied

**(b) If we throw out all widgets that are test defective, how many good widgets are thrown away per year? How many bad widgets are still shipped to customers each year?**

**SOLUTION:**

$Let$B$_{Waste}$ be the number of good widgets discarded per year. Then:
$B_{Waste} = 10^7 * (P(T_X = Negative|X = Positive) * P(X = Positive))$.
$B_{Waste} = 10^7 * 0.99999 * 0.05$ $B_{Waste} = 499995$ (Good Units Discarded)

Let $B_{Bad.Shipped}$ be the number of bad widgets shipped out per year after bad test discarded.Then:
$B_{Bad.Shipped} = 10^7 * (P(X = Positive|T_X = Negative) * P(T_X = Negative))$
$B_{Bad.Shipped} = 10^7 * 0.05 * P(T_X = Negative))$
$P(T_X = Negative)) = (P(T_X = Negative)|X = Negative)*P(X = Negative)+(P(T_X = Negative)|X = Positive) * P(X = Positive)$
$P(T_X = Negative)) = 0.99999 * 0.95 + 0.00001 * 0.05$
$P(T_X = Negative)) = 0.949991$
$B_{Bad.Shipped} = 10^7 * 0.05 * 0.949991$
$B_{Bad.Shipped} = 474995.5$

**3. In k-nearest neighbors, the classification is achieved by majority vote in the vicinity of data. Suppose our training data comprises n data points with two classes, each comprising exactly half of the training data, with some overlap between the two classes.**

**(a) Describe what happens to the average 0-1 prediction error on the training data when the neighbor count k varies from n to 1. (In this case, the prediction for training data point xi includes (xi , yi ) as part of the example training data used by kNN.)**

**SOLUTION:** Average 0-1 prediction error when k=1, is 0. This is because under the assumption that the training data will not contain samples belonging to different classes. A smaller value of k will have low bias and high variance. A higher K averages more voters in each prediction, and hence is more resilient to outliers. Larger values of K will have smoother decision boundaries, i.e, low variance and increased bias.

**(b) We randomly choose half of the data to be removed from the training data, train on the remaining half, and test on the held-out half. Predict and explain with a sketch how the average 0-1 prediction error on the held-out validation set might change when k varies? Explain your reasoning.**

**SOLUTION:** When we randomly choose half of the data to be removed from the training data, train on the remaining half, and test on the held-out half, the prediction will change as k varies - intuitively, the answer would be that as the value of 'k' increases, generally the number of false positives would increase, the average prediction error would hence be more. (Corner cases not in consideration here)

**(c) We wish to choose k by cross-validation and are considering how many folds to use. Compare both the computational requirements and the validation accuracy of using different numbers of folds for kNN and recommend an appropriate number.**

**SOLUTION:**

**Choosing k by means of cross-validation:** The number of folds is determined by the number of instances in a dataset. If in an 8-fold cross validation, with only 8 instances, there would only be 1 instance in the testing bucket. This does not truly represent the underlying distribution. Usually, k=5 is chosen, i.e a 5 fold cross validation , meaning 20 percent of the data is used as testing data. As the data size, or n increases, the value of k can increase. This is because we need the training sample to properly represent the underlying distribution of the dataset.

**(d) In kNN, once k is determined, all of the k-nearest neighbors are weighted equally in deciding the class label. This may be inappropriate when k is large. Suggest a modification to the algorithm that avoids this caveat.**

**SOLUTION:** Weighing the classes unequally is a way to fix the skewed deviations/ class distributions, if any. The class of each of the K neighbours, is multiplied by a weight which is proportional to the inverse of the distance from that point to the given testing point. cation This makes sure that the nearer neighbours contribute more to the final class distribution than the farther ones. So, ultimately if we weigh all neighbours equally, even the farthest point would contribute the same as the nearer one making the knn model inaccurate.

**(e) Give two reasons why kNN may be undesirable when the input dimension is high.**

**SOLUTION:**

1. Whenever the dimensionality of input features/feature set increases, the volume of the space increases rapidly, meaning the available data would become sparse. The distance metric used in kNN becomes meaningless at high input dimensions. This is because we use Euclidean distances for kNN. The accuracy of KNN can be severely degraded with high dimension data because there is little difference between the nearest and the farthest neighbour.

2. In high dimensional spaces, data points that are similar would have very large distances. Our simple intuition of euclidean distances for 2 and 3 dimensional space would breakdown. The distribution of the indegree vertices of the k-NN graph skew with a peak on the right because of the appearance of a disproportionate number of data-points that appear in many more kNN lists of other data-points than the average.

**REFERENCES:**


(a) `https://www.youtube.com/watch?v=k9do8J-w9hYt=334s`

(b) `https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.boxplot.html`

(c) `https://seaborn.pydata.org/generated/seaborn.heatmap.html`

(d) `https://seaborn.pydata.org/generated/seaborn.countplot.html`

(e) `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

(f) `https://machinelearningmastery.com/k-nearest-neighbors-for-machine-learning/`

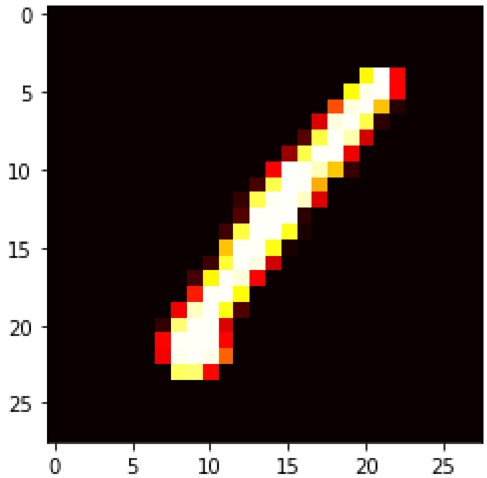
# Attachment 1 - Solutions for Exercise 1

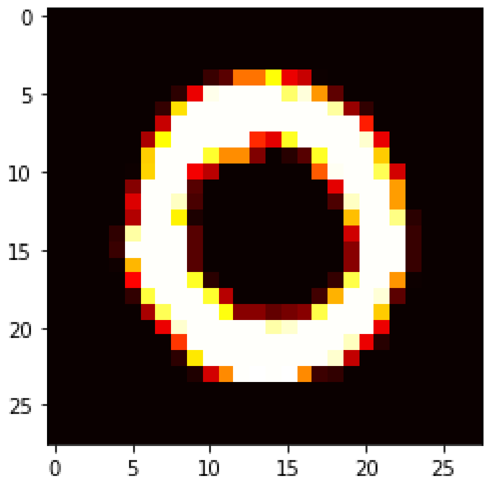
### (a) Join the Digit Recognizer competition on Kaggle. Download the training and test data. The competition page describes how these files are formatted: Answer: The Kaggle Data is read into Python objets below.

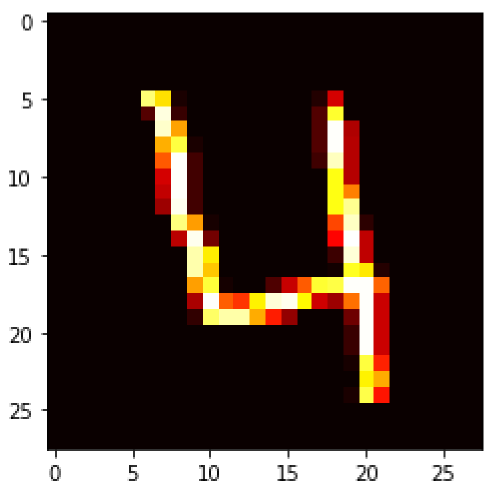
In [10]:
```
import numpy as np
import matplotlib.pyplot as plt

m = int(28)            # This is the width and height of the square image, in
 pixels. The area of image is m*m pixels.

# Part 1.a

# Open data (labels and features as "train")
with open ('train.csv') as train:

            train = np.array (train.readlines()[:])

            # These three lines are condensed below comma separated values
            labels_tr_list = [i.split(',') for i in train]

            # Take all datapoints (labels AND features) for which the label is
either a zero or a one (to solve part e of Problem 1)
            binary_data = np.array([i for i in labels_tr_list[1:] if i[0]=='0'
or i[0]=='1'],dtype=np.float)

            # picks the labels/classes and put to a list
            labels_tr_list = [int(i[0]) for i in labels_tr_list[1:]]


            features_train = np.array([i.split(',') for i in train])
            # m x m reshapeing th e feature set
            features_train = [np.reshape(i,(m,m)) for i in features_train[1:,1
:]]

            # make sure every element is a floating point
            features_train = np.array(features_train,dtype=np.float)


            labels_train = features_train[1:,0]
```

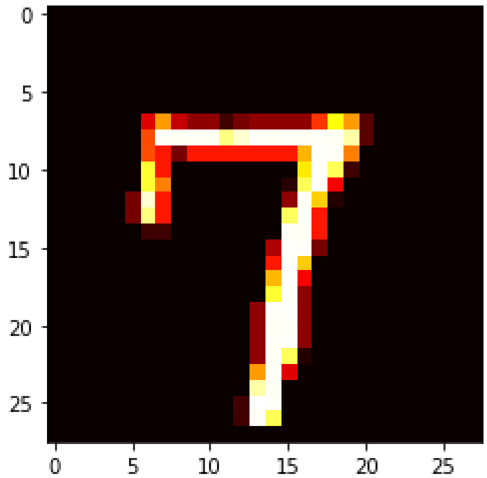
### (b) Write a function to display an MNIST digit. Display one of each digit. Answer: The code below prints each digit from zero to nine, using a heatmap plot representation for each pixel.

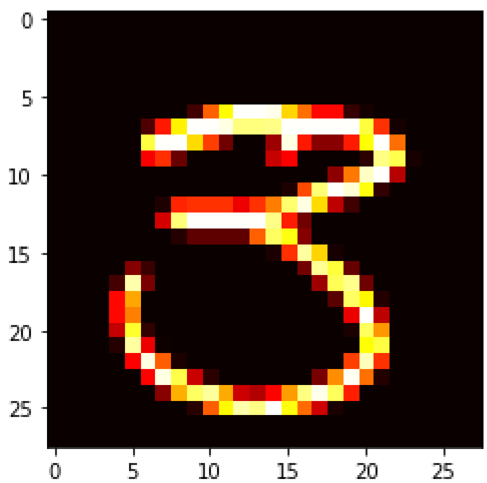
In [11]:
```
#The three lines above, condensed:
#a = np.array([np.reshape(i,(28,28)) for i in (np.array([i.split(',') for i in train]))[1:,1:]],dtype=np.float)

# # Part 1.b

found = []
found_ind = []
for i,j in enumerate(labels_tr_list):
    if len(found_ind)==10:
        break
    if j not in found:
        plt.imshow(features_train[i], cmap='hot', interpolation='nearest')
        plt.show()
        found.append(j)
        found_ind.append(i)
```

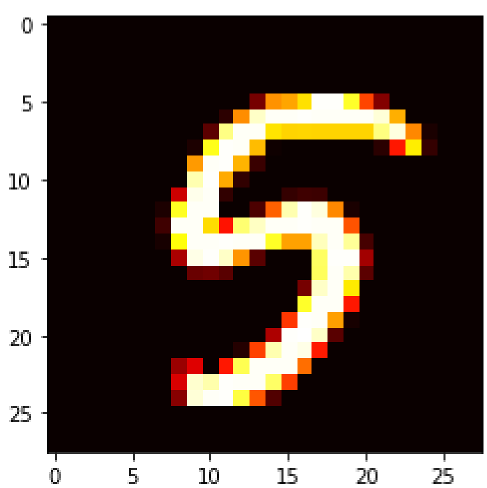


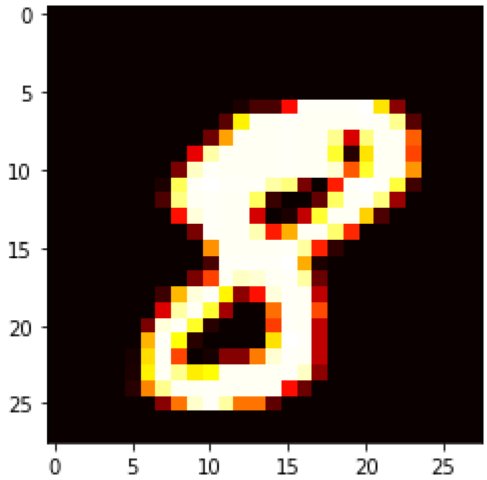


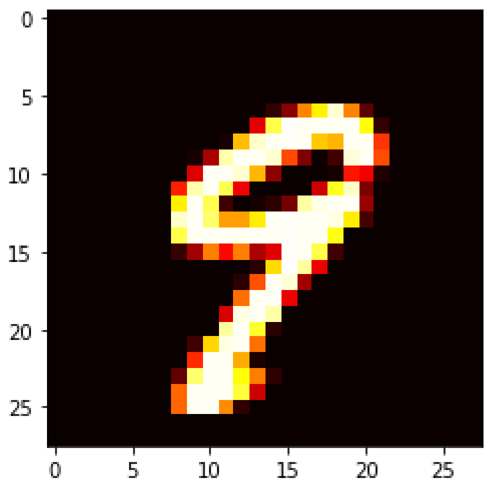


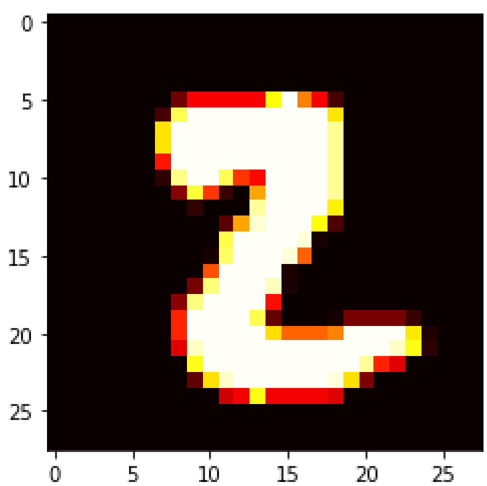

0
5
10
15
20
25
0
5
10
15
20
25

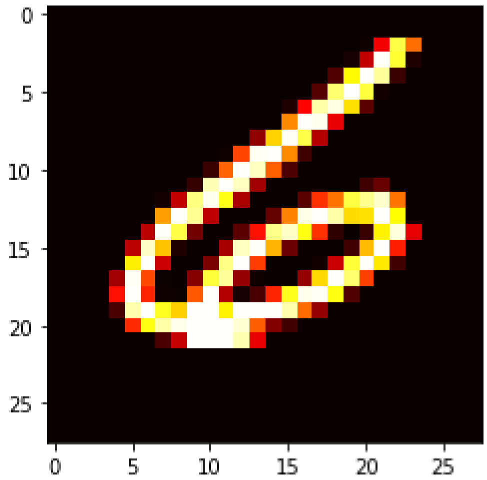

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0 5 10 15 20 25

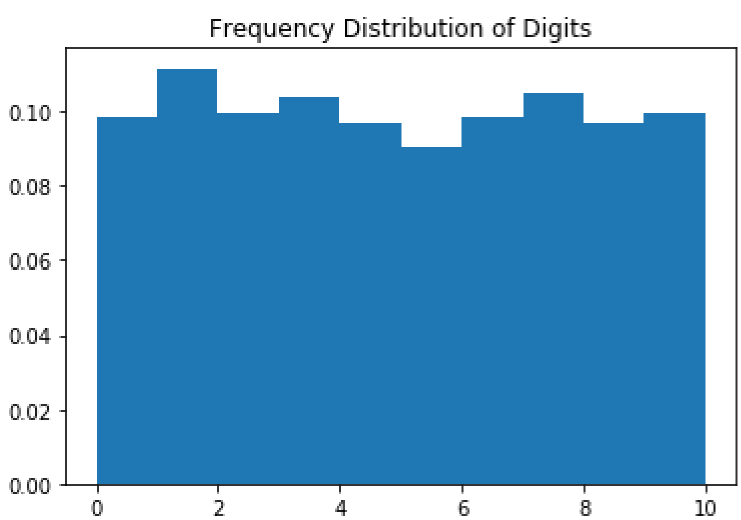

0
5
10
15
20
25
0 5 10 15 20 25


0
5
10
15
20
25
0 5 10 15 20 25



**(c) Examine the prior probability of the classes in the training data. Is it uniform across the digits? Display a normalized histogram of digit counts. Is it even?**
**Answer: The prior probabilities across the classes is not perfectly uniform across all classes, as shown below. As shown on the normalized frequency distribution shown below, it is fairly even, with slightly higher counts of the digit 1.**

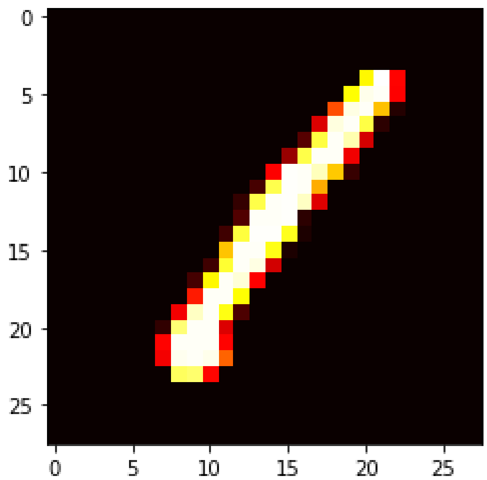
In [12]:
```
# Creates histogram object based on the labels contained in the array "labels_
tr_list". desity argument is used
# to normalize de histogram.
plt.hist(labels_tr_list,bins = np.linspace(0,10,11),density=True)

# Add title to histogram
plt.title("Frequency Distribution of Digits")

# Show histogram
plt.show()
```

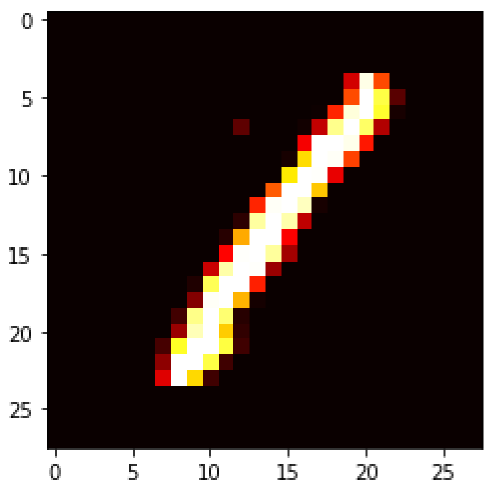


**(d) Pick one example of each digit from your training data. Then, for each sample digit, compute and show the best match (nearest neighbor) between your chosen sample and the rest of the training data. Use L2 distance between the two images' pixel values as the metric. This probably won't be perfect, so add an asterisk next to the erroneous examples (if any). Answer: The code below extracts the closest neighbor for each digit and plots a heatmap of these neighbors.**

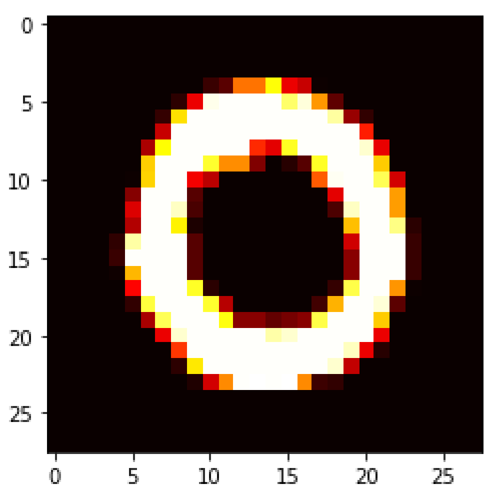
In [18]:
```
# Part 1.d
eucl_dists = np.empty([len(found), len(features_train),1], dtype=int)
for j,p in enumerate(found_ind):
    for i,k in enumerate(features_train):
        eucl_dists[j,i]= (np.dot(np.reshape(features_train[p] - features_train
[i],(1,m*m))  ,  np.transpose(np.reshape(features_train[p] - features_train[i
],(1,m*m)))))**.5
        if eucl_dists[j,i] == 0:
            eucl_dists[j,i] = 1000000000
    min_index = np.argmin(eucl_dists[j])
    if labels_tr_list[p] != labels_tr_list[min_index]:
        print("The Nearest Neighbor (L2) Distance is",str(eucl_dists[j,i]**.5)
.lstrip('[').rstrip(']'),"*")
    else:
        print("The Nearest Neighbor (L2) Distance is",str(eucl_dists[j,i]**.5)
.lstrip('[').rstrip(']'))
    plt.imshow(features_train[p], cmap='hot', interpolation='nearest')
    plt.show()
    plt.imshow(features_train[np.argmin(eucl_dists[j])], cmap='hot', interpola
tion='nearest')
    plt.show()
```

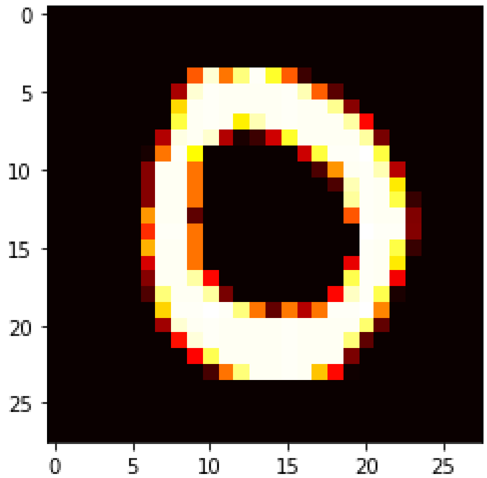
```
The Nearest Neighbor (L2) Distance is 46.65833259
```



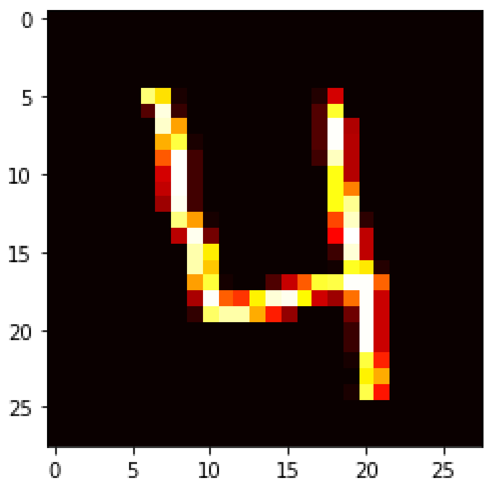



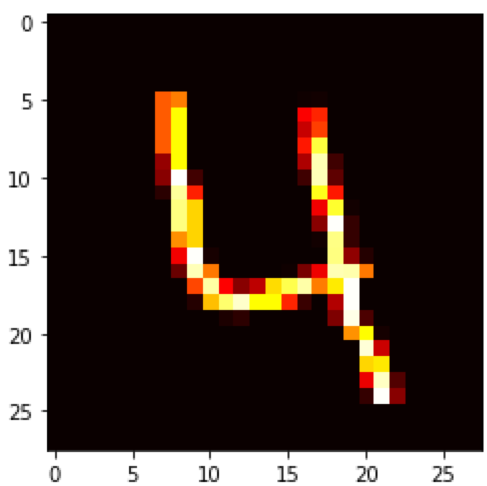

```
The Nearest Neighbor (L2) Distance is 55.62373594
```



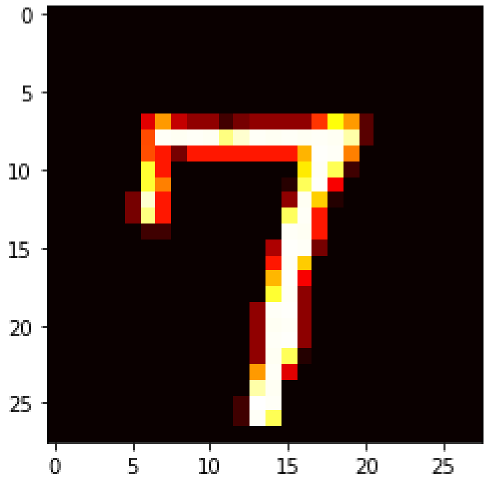


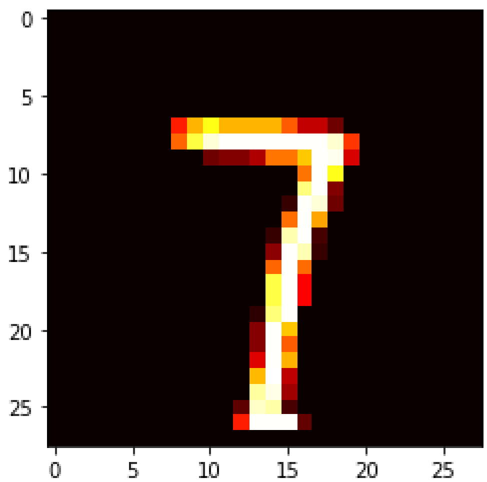

The Nearest Neighbor (L2) Distance is 46.66904756



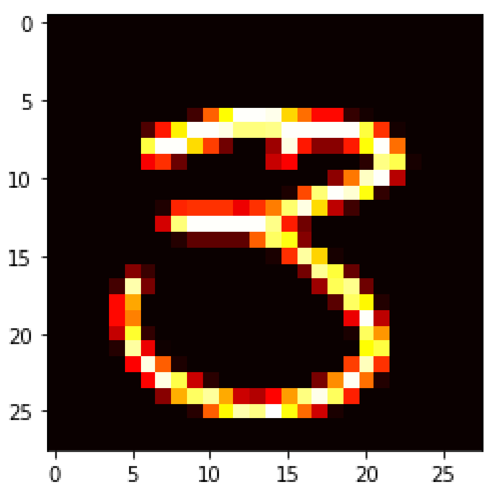



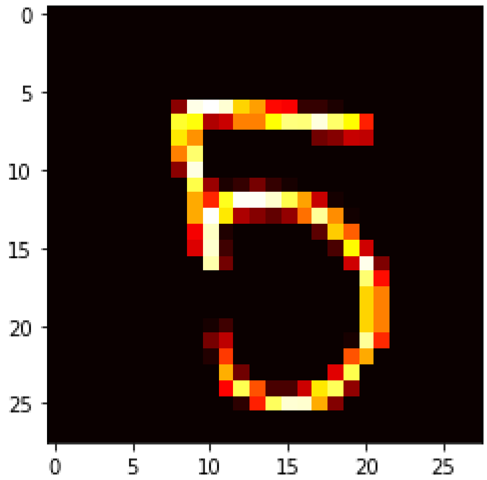

The Nearest Neighbor (L2) Distance is 44.58699362



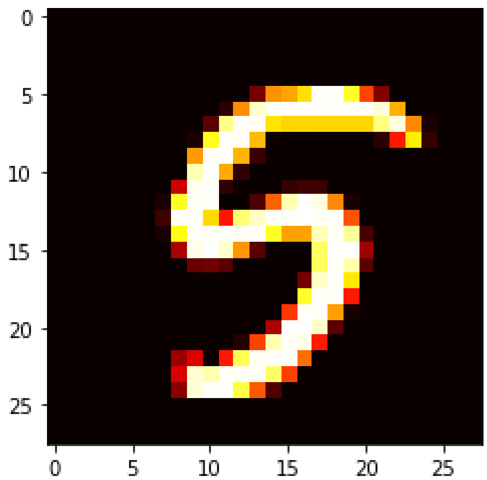


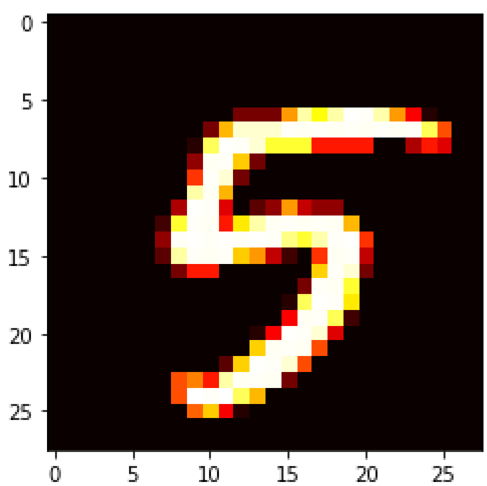
```
The Nearest Neighbor (L2) Distance is 48.82622246 *
```



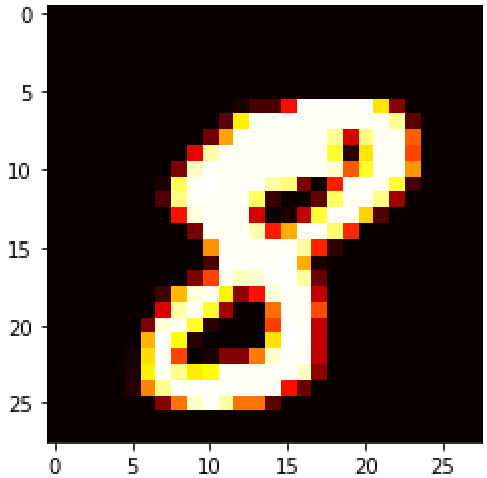


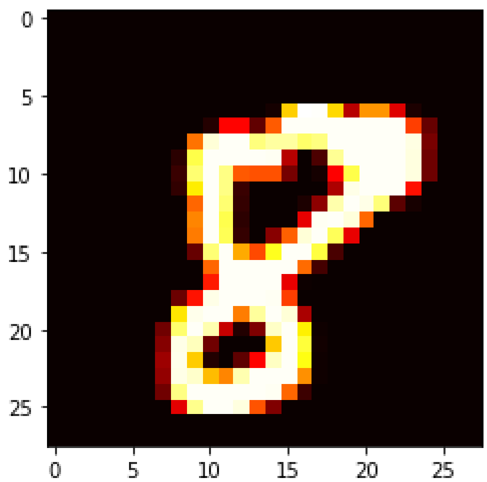
The Nearest Neighbor (L2) Distance is 49.689033



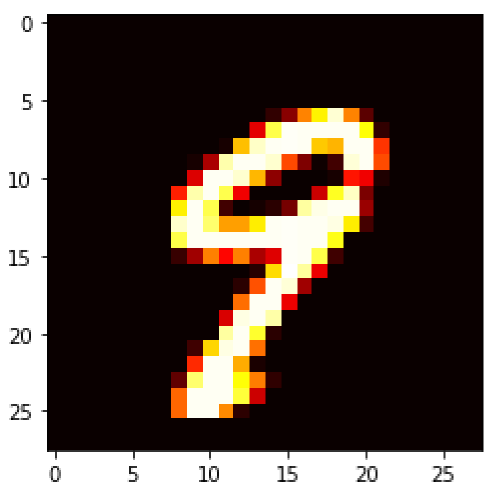


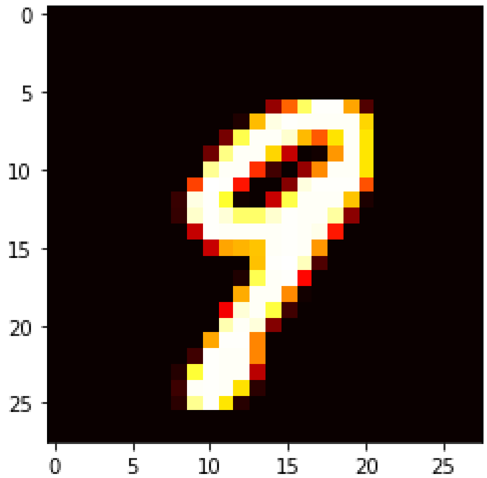
The Nearest Neighbor (L2) Distance is 50.82322304



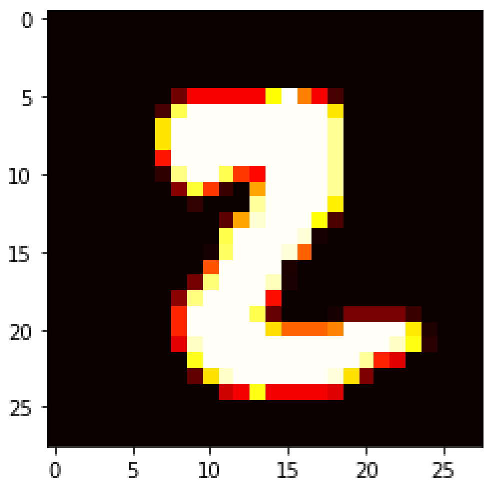



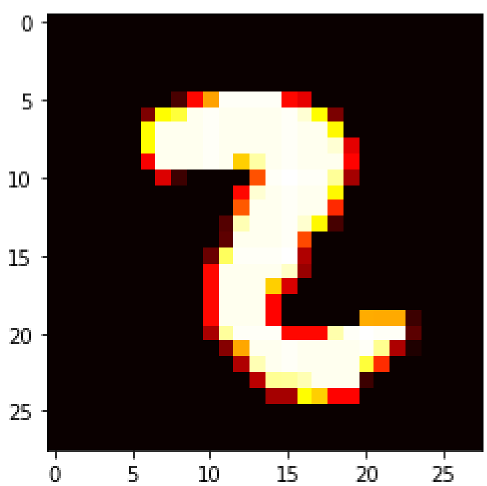

```
The Nearest Neighbor (L2) Distance is 46.06517123
```



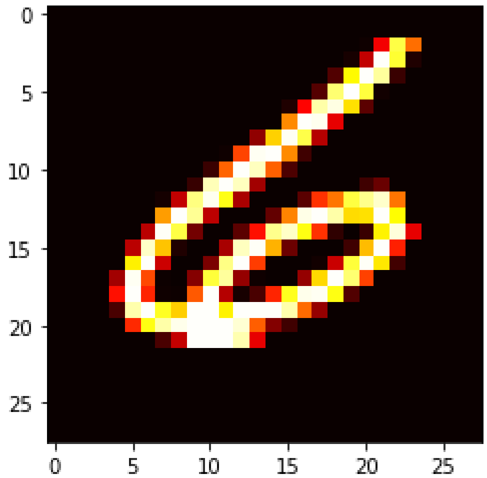


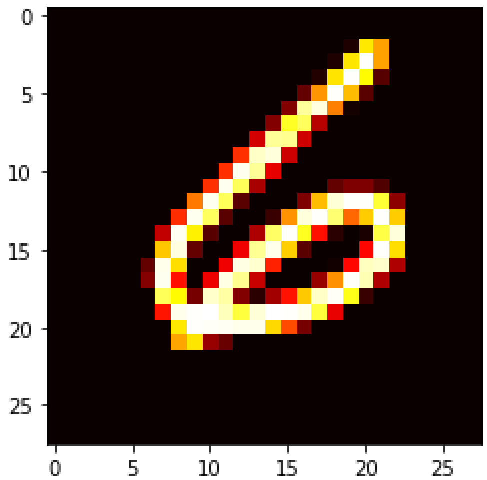
The Nearest Neighbor (L2) Distance is 51.8748494





The Nearest Neighbor (L2) Distance is 49.1934955



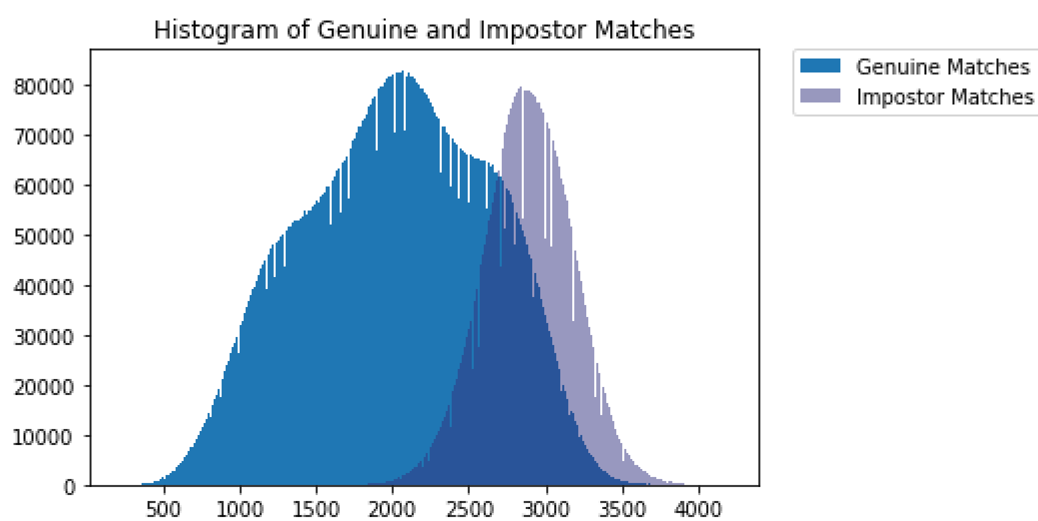


**e) Consider the case of binary comparison between the digits 0 and 1. Ignoring all the other digits, compute the pairwise distances for all genuine matches and all impostor matches, again using the L2 norm. Plot histograms of the genuine and impostor distances on the same set of axes. Answer: The solution is computed below. A boolean test is completed to verify that all zeros and ones in the dataset are included in the computation.**

In [20]:
```
# Part 1.e Consider the case of binary comparison between the digits 0 and 1.
 Ignoring all the other
# digits, compute the pairwise distances for all genuine matches and all impos
tor matches,
# again using the L2 norm. Plot histograms of the genuine and impostor distanc
es on the same
# set of axes.

# Create empty array to hold eucledian distances for observations with labels
 that are either
# a zero (0) or one (1).


# This represents the number of unique distances. It can be computed as the nu
mber of unique
# pair combinations given n objets = n!/((n-2)!*2!). However, that just simpli
fies to the expression
# shown below for simplicity.
dist_count = int((len(binary_data))*(len(binary_data)-1)  /  2)

# Create empty array of size dist_count (the total number of unique distances)
by 2.
bin_eucl_dists = np.empty([dist_count,2], dtype=int)

# Dummy variable to index what element of bin_eucl_dists we write to (see nest
ed loops below).
a = 0


# Nested for loops to populate "bin_eucl_dists" array (created above) with two
values:

# Value bin_eucl_dists[i,0] is the Eucledian Distance between two feature vect
ors. Note that we take the square root
# of the inner product to get the Eucledian Distance.

# Value bin_eucl_dists[i,1] is a Boolean value (0,1) representing whether thes
e two elements are in fact matches.

#print(binary_data)
for i1,j1 in enumerate(binary_data):

    # If you have n distances, the nth distance will be compared iteratively w
ith distances d_0 through distance
    # d_n-1. Hence, we say below that i1 must remain < dist_count, allowing di
st_count - 1 iterations.
    if i1 < dist_count:
        for i2,j2 in enumerate(binary_data[i1+1:]):
            # Here, we index vector j1 and j2 from column 1 to : because colum
n 0 is just the label,
            # not to be included in the vector difference computation.

            bin_eucl_dists[a,0],bin_eucl_dists[a,1] = (np.dot(j1[1:]-j2[1:],np
.transpose(j1[1:]-j2[1:])))**.5, j1[0]==j2[0]
            a +=1
```

```
# Count the number of zero and one labels in binary_data
len_ones = len([i for i in binary_data if i[0]==1])
len_zeros = len([i for i in binary_data if i[0]==0])

# Boolean operation to test that all elements in binary_data (zero and one lab
els) have been captured.
print (len_ones + len_zeros == len(binary_data))

# Extract genuine matches from bin_eucl_dists based in matching of labels in n
ested loops above this line.
genuine_match = [j[0] for j in bin_eucl_dists if j[1]==1]
false_match = [j[0] for j in bin_eucl_dists if j[1]==0]


#Create histograms for genuine and impostor matches
_ = plt.hist(genuine_match, bins="auto",label="Genuine Matches")  # arguments
 are passed to np.histogram
_ = plt.hist(false_match, bins="auto", label="Impostor Matches", fc =(.2,.2,.5
), alpha=.5)  # arguments are passed to np.histogram
plt.title("Histogram of Genuine and Impostor Matches")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.show()

g_hist = np.histogram(genuine_match, bins="auto")
i_hist = np.histogram(false_match, bins="auto")
```

```
True
```



**Part 1.(f) Generate an ROC curve from the above sets of distances. What is the equal error rate? What is the error rate of a classifier that simply guesses randomly? Answer: The ROC of True Matches (TM, a.k.a.: True Positives) versus False Matches (FM, a.k.a.: False Positives) is generated below. For this curve, the equal error rate occurs when the FM (or false positive) rate equals 0.1857. This point is found by the intersection with the "1-x" line and identifies the value at which the rate of false positive predictions equals the rate of false negative predictions.**

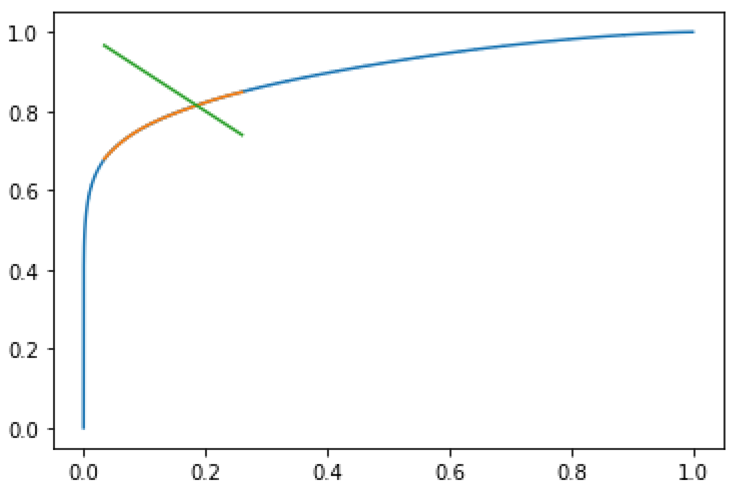
**What is the error rate of a classifier that simply guesses randomly? Answer: The error rate of a binary classifier that simply guesses randomly is 0.5. Rationale: E_random = P(Match)*P(Prediction = Match) + P(Not_Match)*P(Prediction = Not_Match) {Equation 1} ---> P(Prediction = Match) = P(Prediction = Not_Match) = 0.5 = 1/num_of_classes (i.e. we're naively guessing)---> Also, P(Not_Match) = 1 - P(Match) ---> Hence, Equation 1 becomes: E_random = (P(Match) + 1 - P(Match))*0.5 = 1*0.5 = 0.5**

In [21]:
```
# Part 1.f Generate an ROC curve from the above sets of distances. What is the
equal error rate? What
# is the error rate of a classifier that simply guesses randomly?

ROC_x = []
ROC_y = []

sort_bin_eucl_dists = bin_eucl_dists[bin_eucl_dists[:, 0].argsort()]

for i,j in enumerate(sort_bin_eucl_dists):
    if j[1]==1:
        if i == 0:
            ROC_x.append(0)
            x_last = 0
            ROC_y.append(1/len(genuine_match))
            y_last = 1/len(genuine_match)
        else:
            ROC_x.append(x_last)
            ROC_y.append(y_last + 1/len(genuine_match))
            y_last = y_last + 1/len(genuine_match)
    else:
        if i == 0:
            ROC_y.append(0)
            y_last = 0
            ROC_x.append(1/len(false_match))
            x_last = 1/len(false_match)
        else:
            ROC_y.append(y_last)
            ROC_x.append(x_last+ 1/len(false_match))
            x_last = x_last+ 1/len(false_match)

#print(ROC_y)
# Create two dimensional empty array to hold X and Y values of ROC Curve
ROC_comb = np.empty([len(ROC_y),2])

# Store X and Y values of ROC Curve in Array
for i,j in enumerate(ROC_comb):
    j[0], j[1] = ROC_x[i],ROC_y[i]

# Extract points of the ROC curve where slope passes through Equal Error Rate.
# I just looked at the ROC graph and saw that it happens somewhere prior to Y
 = 0.8
# which corresponds to these indexes: Start at the 33th percentile (100/3), en
d at 57th
# percentile (100/1.75).


low_ind = int(len(ROC_comb)/2.8)
high_ind = int(len(ROC_comb)/1.8)
ROC_for_EER = ROC_comb[low_ind:high_ind]

# Define degre of polynomial fit
n1 = 4
# Fit ROC_for_EER using "n1" degree polynomial (Array Form)
z1 = np.polyfit(ROC_for_EER[:,0],ROC_for_EER[:,1],n1)
# Take Array Form polynomial and turn into function
```

```
p1 = np.poly1d(z1)
print(p1)

z2 = np.polyfit([0,1],[1,0],1)
z2 = [0,0,0,z2[0],z2[1]]
p2 = np.poly1d(z2)
print(p2)
#print(z1-z2)
EER = np.roots(z1-z2)

print("The false positive rate equals the false negative rate when the rates e
qual", np.real(EER[3]))


# Print the index of the element of ROC_for_EER that results in a slope closes
t to 1/2**.5
#print(ROC_for_EER[delta_der_p2.argmin(0),1])

fig2, ax2 = plt.subplots()
#ax2.set_aspect('equal', 'box')
ax2.plot(ROC_x,ROC_y)

ax2.plot(ROC_for_EER[:,0],p1(ROC_for_EER[:,0]))
ax2.plot(ROC_for_EER[:,0],p2(ROC_for_EER[:,0]))
```

```
        4         3         2
-73.44 x + 54.5 x - 15.85 x + 2.66 x + 0.6052

-1 x + 1
The false positive rate equals the false negative rate when the rates equal
0.18572097396707798
```

Out[21]: `[<matplotlib.lines.Line2D at 0x2a2a419cbe0>]`




**(g) Implement a K-NN classifier. (You cannot use external libraries for this question; it should be your own implementation.) Answer: The code below implements the K-NN Classifier. It is very memory-intensive so it was not run on Jupyter. Instead, Spyder or a similar IDE can be used to execute it.**

In [ ]:
```
import numpy as np
import csv

m = int(28)            # This is the width and height of the square image, in
 pixels. The area of image is m*m pixels.

# Part 1.a

# Open data (labels and features as "train")
with open ('train.csv') as train:

    #create array from lines in "train" object
    train_raw = np.array (train.readlines()[:])

# create unprocessed array (i.e.: labels, column names and features are still
 included)
train_raw = np.array([i.split(',') for i in train_raw])

# process array of features (i.e.: remove features, remove column names and ke
ep labels)
labels_train = np.array([int(i[0]) for i in train_raw[1:]],dtype=np.int)

# process array of features (i.e.: remove labels, remove column names and appl
y m x m reshapeing the feature set)
# data type for features is floating
features_train = np.array([np.reshape(i,(m,m)) for i in train_raw[1:,1:]],dtyp
e=np.float)

with open ('test.csv') as test:

    #create array from lines in "test" object
    test_raw = np.array (test.readlines()[:])

# create unprocessed array (i.e.: labels, column names and features are still
 included)
test_raw = np.array([i.split(',') for i in test_raw])
# process array of features (i.e.: remove features, remove column names and ke
ep labels)

# process array of features (i.e.: remove column names and apply m x m reshape
ing the feature set)
# data type for features is floating
features_test = np.array([np.reshape(i,(m,m)) for i in test_raw[1:,:]],dtype=n
p.float)
class_pred = []

k_par = 9
dist_array = np.zeros([len(features_test),len(features_train),2],dtype=np.int)
for j,p in enumerate(features_test):
    for i,k in enumerate(features_train):
        dist_array[j,i][0] = (np.dot(np.reshape(features_test[j] - features_tr
ain[i],(1,m*m))  ,  np.transpose(np.reshape(features_test[j] - features_train[
i],(1,m*m)))))**.5
        dist_array[j,i][1] = labels_train[i]
    sort_dist_array=labels_train[np.argsort(dist_array[j,:,0])]
    a = np.array(sort_dist_array[0:k_par])
```

```
#     print(a)
    counts = np.bincount(a)
#     print (np.argmax(counts))
    class_pred.append([j,int(np.argmax(counts))])
#     plt.imshow(features_test[j], cmap='hot', interpolation='nearest')
#     plt.show()

with open("submittion.csv",'w', newline='') as resultFile:
    wr = csv.writer(resultFile)
    wr.writerows([class_pred])

print("Done")
```

**(h) Using the training data for all digits, perform 3 fold cross-validation on your K-NN classifier and report your average accuracy. Answer: The 3-fold cross validation model is implemented using the code below. The accuracy per bucket was 0.963, 0.962 and 0.964, for an average prediction accuracy across buckets of 0.961.**

In [ ]:
```
import numpy as np
import csv

m = int(28)            # This is the width and height of the square image, in
 pixels. The area of image is m*m pixels.

#Number of observations to read for run. (n_in+ 1) MUST BE MULTIPLE OF 3.
n_per_bucket = int(42000/3)
buckets = int(3)

with open ('train.csv') as train:
    #create array from lines in "train" object
    train_raw = np.array (train.readlines()[0:buckets*n_per_bucket+1])

# create unprocessed array (i.e.: labels, column names and features are still
 included)
train_raw = np.array([i.split(',') for i in train_raw])

# process array of features (i.e.: remove features, remove column names and ke
ep labels)
labels_train = np.array([int(i[0]) for i in train_raw[1:]],dtype=np.int)

# process array of features (i.e.: remove labels, remove column names and appl
y m x m reshapeing the feature set)
# data type for features is floating
features_train = np.array([np.reshape(i,(m,m)) for i in train_raw[1:,1:]],dtyp
e=np.float)


bucket_features_train = []
bucket_features_test = []
bucket_labels_train = []
bucket_labels_test = []
for i in range(buckets):

    bucket_features_train.append(list(features_train))
    del bucket_features_train[i] [i*n_per_bucket:(i+1)*n_per_bucket]

    bucket_labels_train.append(list(labels_train))
    del bucket_labels_train[i] [i*n_per_bucket:(i+1)*n_per_bucket]

    bucket_features_test.append(list(features_train[i*n_per_bucket:(i+1)*n_per
_bucket]))
    bucket_labels_test.append(list(labels_train[i*n_per_bucket:(i+1)*n_per_buc
ket]))


bucket_features_train = np.array(bucket_features_train)
bucket_features_test = np.array(bucket_features_test)
bucket_labels_train = np.array(bucket_labels_train)
bucket_labels_test = np.array(bucket_labels_test)


k_par = 9

bucket_dist_array = np.zeros([buckets, n_per_bucket,int((buckets-1)*n_per_buck
```

```
et),2],dtype=np.int)
class_predi = []

for k1 in range(buckets):
    global class_pred
    match_count = 0
    for j,p in enumerate(bucket_features_test[k1]):

        for i,k in enumerate(bucket_features_train[k1]):
            bucket_dist_array[k1,j,i][0] = (np.dot(np.reshape(bucket_features_
test[k1,j] - bucket_features_train[k1,i],(1,m*m))  ,  np.transpose(np.reshape(
bucket_features_test[k1,j] - bucket_features_train[k1,i],(1,m*m)))))**.5
            bucket_dist_array[k1,j,i][1] = bucket_labels_train[k1,i]
        bucket_sort_dist_array=bucket_labels_train[k1,np.argsort(bucket_dist_a
rray[k1,j,:,0])]
        a = np.array(bucket_sort_dist_array[0:k_par])
#
        counts = np.bincount(a)
        class_pred = (np.argmax(counts))
        class_predi.append([int(np.argmax(counts)),int(bucket_labels_test[k1,j
])])

        if class_pred == bucket_labels_test[k1,j]:
            match_count +=1
#             print("IT'S A MATCH!!!! :) ")
#         plt.imshow(bucket_features_test[k1][j], cmap='hot', interpolation='n
earest')
#         plt.show()


    print("The accuracy for this split is", match_count/n_per_bucket)
```

**(i) Generate a confusion matrix (of size 10 x 10) from your results. Which digits are particularly tricky to classify? Answer: The confusion matrix is generated through the code below. Rows are predictions and columns are actual labels. The average accuracy of the cross validation was 96.1%. The digits four and five were particularly tricky to classify.**

| Predicted\Actual | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 4104 | 0 | 42 | 6 | 3 | 11 | 25 | 3 | 19 | 16 |
| 1 | 2 | 4656 | 75 | 22 | 54 | 14 | 8 | 72 | 62 | 15 |
| 2 | 3 | 9 | 3927 | 26 | 0 | 0 | 2 | 11 | 13 | 4 |
| 3 | 0 | 2 | 17 | 4170 | 0 | 58 | 0 | 1 | 78 | 36 |
| 4 | 0 | 3 | 4 | 5 | 3887 | 6 | 7 | 11 | 21 | 40 |
| 5 | 6 | 0 | 6 | 44 | 0 | 3624 | 16 | 0 | 81 | 10 |
| 6 | 14 | 2 | 7 | 3 | 13 | 44 | 4077 | 0 | 22 | 1 |
| 7 | 2 | 8 | 86 | 29 | 10 | 3 | 0 | 4262 | 13 | 74 |
| 8 | 0 | 1 | 7 | 24 | 1 | 6 | 2 | 0 | 3699 | 10 |
| 9 | 1 | 3 | 6 | 22 | 104 | 29 | 0 | 41 | 55 | 3982 |

In [ ]:

```
class_predi = np.array(class_predi)

#print(len(class_predi))
num_confu = [0,1,2,3,4,5,6,7,8,9]
confu_mat = np.zeros([11,11], dtype=int)

for i1,j1 in enumerate(num_confu):
    confu_mat[0,i1+1]=i1
    confu_mat[i1+1,0]=i1

for i1,j1 in enumerate(class_predi):

    # index "class_predi[i1,0]" determines what row you +1 to. It is extractin
g the predicted value from KNN
    # index "class_predi[i1,1]" determines what column you +1 to. It is extrac
ting the actual value from KNN
    confu_mat[class_predi[i1,0]+1,class_predi[i1,1]+1] +=1

print("\n")
print("Below is the Confusion Matrix.")
print("Rows Represent Predicted Labels for Digits 0 thru 9.")
print("Columns Represent Actual Labels for Digits 0 thru 9.")
print("Matrix Element 0,0 is superflous.\n")

print(confu_mat)

with open('class_predi31.csv', 'w', newline='') as f0:
    writer = csv.writer(f0)
    writer.writerow(confu_mat)

with open('class_predi32.csv', 'w', newline='') as f0:
    writer = csv.writer(f0)
    writer.writerow(class_predi)
```

### (j) Train your classifier with all of the training data, and test your classifier with the test data. Submit your results to Kaggle. Answer: The classifier was submitted to Kaggle under Team Name ec833sj747. An accuracy of 96.7% was achieved.

In [ ]: