CS5785: Applied Machine Learning **(Due: 09/26/19)**

Homework 1

Instructor: Serge Belongie *Names:* Eva Esteban Velasco, Simran Rajpal*, Netids:* epe26, sr2258

1 Programming Exercises

1.1 Digit Recognizer

(a,b) For this question, we first joined the "Digit Recognizer" competition on Kaggle and downloaded the training and testing datasets. We loaded the .csv files containing the training and testing data, and reshaped the data into 28x28 images of the digits. We wrote a function which searches for the first instance of every MNIST digit class from 0 to 9 and displays it. The results of this can be found on Figure 1.

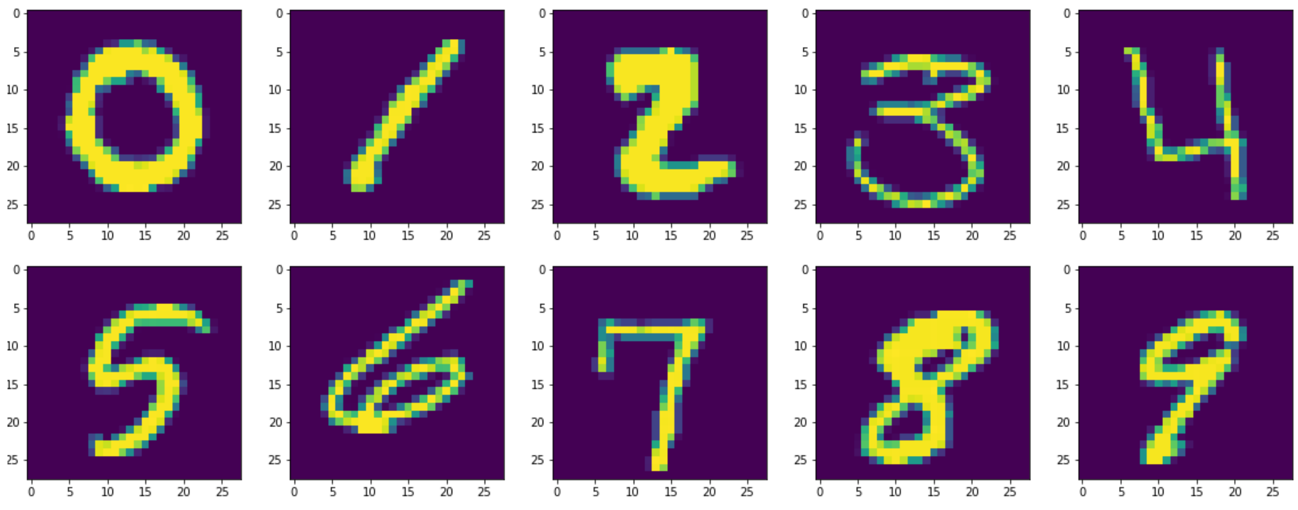

Figure 1: Display of one instance of each MNIST digits.

(c) Then, we wrote a function to calculate and print the prior probability of each digit. The results output by the function were:

- Prior probability for 0 : 0.09838095238095237
- Prior probability for 1 : 0.11152380952380953
- Prior probability for 2 : 0.09945238095238096
- Prior probability for 3 : 0.1035952380952381
- Prior probability for 4 : 0.09695238095238096
- Prior probability for 5 : 0.09035714285714286
- Prior probability for 6 : 0.0985
- Prior probability for 7 : 0.10478571428571429
- Prior probability for 8 : 0.09673809523809523
- Prior probability for 9 : 0.09971428571428571

These prior probabilities range from a minimum of 0.090 for digit '5' to a maximum of 0.111 for digit '1'. From this, we can see that the prior probability is almost uniform across all digits. Once this analysis was completed, we wrote a function to create a histogram of normalized counts. The histogram is shown on Figure 2. See appendix A1 for the code that generated this graph and attached code for the complete code. It is relatively very even, although there is a slight variation between the counts for each digit.

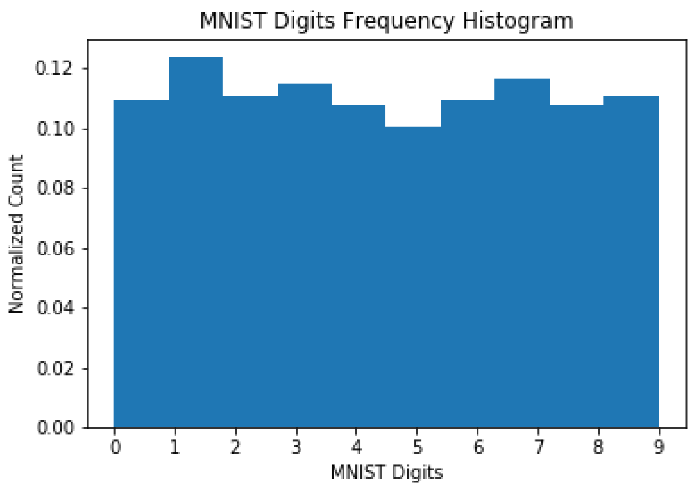

Figure 2: Histogram showing the normalized count number for each MNIST digit. (Code: appendix A1)

(d) We chose an instance of each digit and calculated its L2 norm distance or Euclidean distance to all of the other digits to find its best match or nearest neighbor. Initially we created our own function to calculate the Euclidean distance, but since the running time was considerable, we then changed it to use numpy's linear algebra function to calculate the norm of the difference between the two image matrices in order to have the best computation time [1]. The output of this function was:

- The nearest neighbor for the image selected for 0 is at index 12950
- The nearest neighbor for the image selected for 1 is at index 29704
- The nearest neighbor for the image selected for 2 is at index 9536
- * The nearest neighbor for the image selected for 3 is at index 8981
- The nearest neighbor for the image selected for 4 is at index 14787
- The nearest neighbor for the image selected for 5 is at index 30073
- The nearest neighbor for the image selected for 6 is at index 16240
- The nearest neighbor for the image selected for 7 is at index 15275
- The nearest neighbor for the image selected for 8 is at index 32586
- The nearest neighbor for the image selected for 9 is at index 35742

As we can see, the matches were all correct except for digit '3', for which the algorithm returned an image of digit '5'. This is expected as the algorithm is only trying to find the nearest neighbor to one example of the digit '3', and this makes it less robust. Figure 3 provides a visual representation of the matches for each digit previously shown in Figure 1

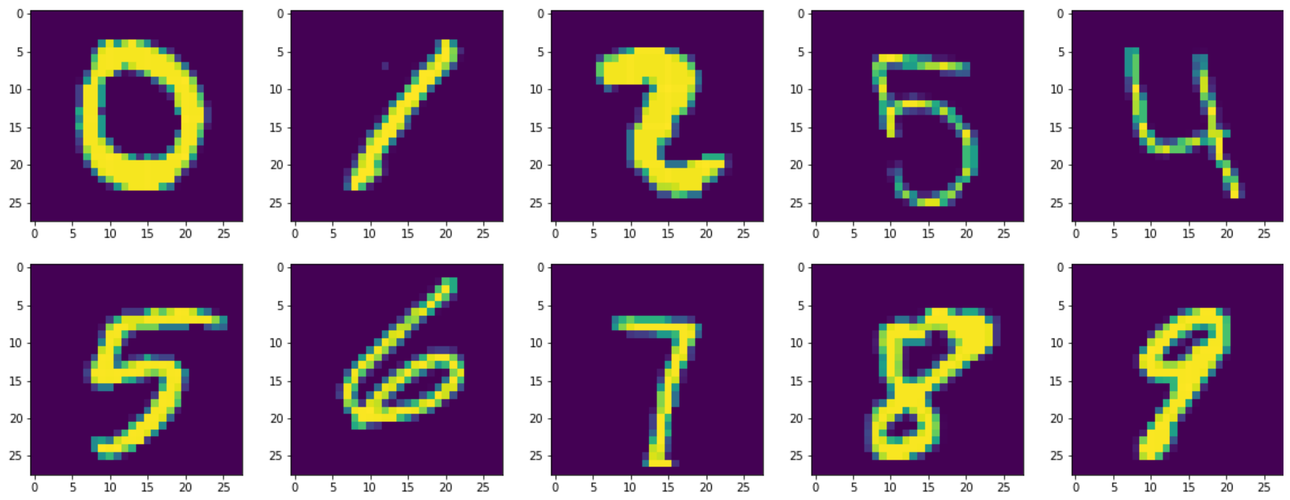

Figure 3: Display of each MNIST digit best match.

(e) Next, to consider the case of binary comparison between the digits '0' and '1', we used the L2 norm or Euclidean distance to compute the pairwise distances between all genuine matches and all impostor matches. Genuine matches are the distances between each '0' image and every other '0' image, and the distances between each '1' image and every other '1' image. Imposter matches are the distances between each '0' image and all '1' images, and the distance between each '1' image and all '0' images. We plotted a histogram of the results, which can be found on Figure 4. See appendix A2 for the code that generated this graph and attached code for the complete code.

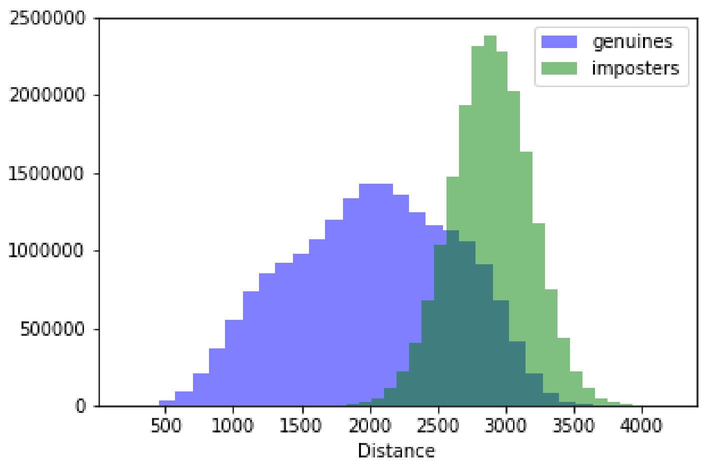

Figure 4: Histogram of genuine and impostor matches. (Code: appendix A2)

(f) To analyze the performance of this technique, we plotted a Receiver Operating Characteristic (ROC) curve, which can be found on Figure 5. See appendix A3 for the code that generated this graph and attached code for the complete code. Our ROC curve determines if we are correctly classifying into either the genuine or imposter distribution. We plotted the ROC curve with relation to the genuine and imposter matches by determining the False Positive Rate (FPR) and True Positive Pate (TPR) at different threshold distances.

$FPR = \frac{FP}{FP+TN}$ $TPR = \frac{TP}{TP+FN}$

False positives (FP) occur when the distance is greater than the threshold but the value is in the genuine distribution because these are items that have a high distance and should be classified as the other value, not itself. True negatives (TN) occur when the distance is less than the threshold and the value is in the genuine distribution because these are items that have a low distance and so are correctly classified as themselves. False negatives (FN) occur when the distance is less than the threshold and the value is in the imposter distribution because these are items that have a low distance and so are incorrectly classified as something other than themselves. True positives (TP) occur when the distance is greater than the threshold and the value is in the imposter distribution because these are items that have a high distance and so are correctly classified as something other than themselves.

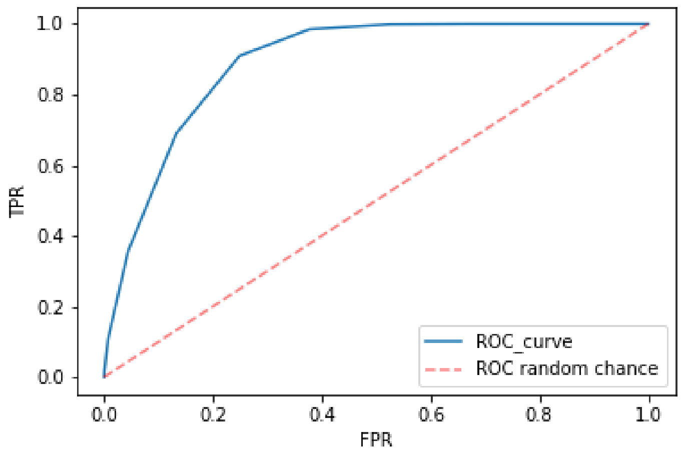

Figure 5: ROC curve of the performance of the genuine and impostor matches separation technique. (Code: appendix A3)

The equal error rate (EER) for this ROC curve is 0.19 [2]. This was calculated by determining the intersection of the ROC curve with the curve y=-x+1. This is the FPR for which the FPR is equal to the FNR. For comparison, the equal error rate of random chance is 0.5. Thus, our equal error rate is smaller and therefore our classifier works better than random chance.

(g) Once we had an understanding of how k-Nearest-Neighbors (KNN) works, we developed an algorithm to perform KNN on a dataset. This algorithm takes a number called 'mynum' and attempts to classify it into a number 0-9 based on a training dataset called 'nums' with labels. It performs this classification using the majority vote of 'k' nearest neighbors. To perform this classification, we iterated through all numbers, 'num', in 'nums' and found the distance between 'num' and 'mynum'. While doing this we maintained a sorted list that was k values long to store tuples of (distance, label) of the values with the shortest distance. Every time we can across a 'num' that had a shorter distance than the maximum samples in the list, we replaced this item with the new sample. We ended up with the k nearest neighbors. Then we took the majority vote of these k nearest neighbors to determine where to classify 'mynum'.

We tested this model with k=3 by splitting our training data into train and test sets, where the test set was 25% of the data and determined the accuracy to be 96.6%.

(h) We then performed 3-fold crossvalidation on our model with k=3 using sklearn.model_selection's KFold method. While performing crossvalidation we stored the accuracy, to allow us to determine the mean accuracy after all folds were completed. We also stored the test labels and the predictions to develop a confusion matrix later. Our accuracy was 96.5%.

(i) Then, we created a confusion matrix from all of the true label values and the predicted values found during crossvalidation. This is shown in Figure 6. See appendix A4 for the code that generated this graph and attached code for the complete code. The digit that was the most tricky to classify was '5'. This digit had misclassifications accross all numbers 0-9 but there were more misclassifications into '3' and '9' than others. Other numbers that were most difficult to classify were '9', '4', '2', and '10', in that order. It was interesting that '4' and '10' were confused a lot. The other numbers were able to get over 4000 correct classifications.

```
[[4110    0    4    0    0    5   10    0    2    1]
 [   0 4658    8    1    2    1    4    7    1    2]
 [  31   48 3988   13    5    3    4   74    7    4]
 [   6   12   34 4202    0   39    2   21   18   17]
 [   3   45    1    0 3899    0   15    4    1  104]
 [  15    4    2   75    4 3618   43    2    5   27]
 [  26    7    0    0    5   17 4080    0    2    0]
 [   2   54   10    3   12    0    0 4268    0   52]
 [  24   50   25   68   19   76   17   11 3730   43]
 [  17   11    6   34   44   13    2   59    7 3995]]
```

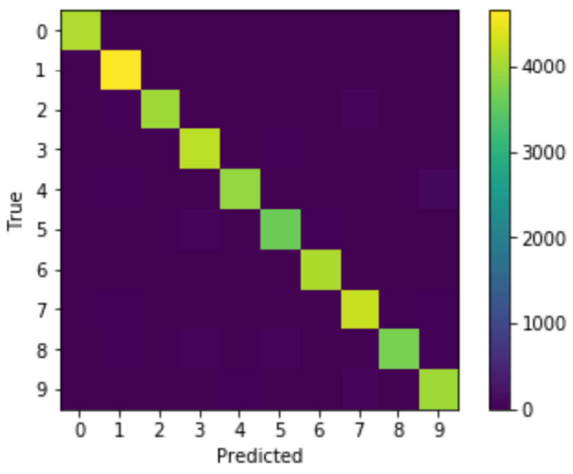

Figure 6: Confusion Matrix after 3-fold Crossvalidation of a 3-NN Classifier. (Code: appendix A4)

(j) We trained our classifier with all of the training data and determined guesses for all of the test data and submitted the results to Kaggle under the name “Simran Rajpal”. We were unable to create a team for this challenge. Our predictions gave an accuracy of 96.857%. We were ranked at #2055.

1.2 The Titanic Disaster

(a) For this question, we joined the "Titanic: Machine Learning From Disaster" competition on Kaggle and downloaded the training and testing datasets.

(b) We read the data into a Pandas DataFrame [3] to determine which features need to be modified or dropped.

We determined that 77% of 'Cabin' values were NaN, so we decided to drop this feature, since there was not enough data. We also decided to drop 'PassengerId', 'Name', and 'Ticket', because these features are uniquely specific to the person. We found that most of the survivers were male and young by creating some plots that can be seen in our iPython Notebook, so we found it to be important to include 'Age' and 'Sex' in our model.

Then we cleaned up the data. We transformed the 'Sex' and 'Embarked' data into binary values using Pandas's 'get_dummies' function. We also, cleaned up the 'Age' data because about 20% of the age values were NaN. To clean this up we replaced the NaN ages with the mean of all ages so that it did not impact our model. Finally, we removed the 'Survived' data from the features and set it as the labels, so that the model did not train based on the labels.

We used the LogisticRegression model from sklearn.linear_model. We randomly divided our training data into a train and test set using train_test_split, where 25% of the data went into the test set. Then, we fit the model to the train set and tested it using this test set. We were able to get 84.75% accuracy. There results are reflected on the confusion matrix in Figure 7.

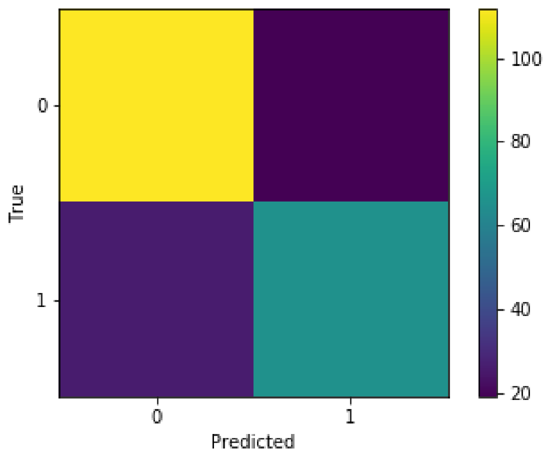

Figure 7: Confusion matrix for Logistic Regression classifier. (Code: appendix A5)

(c) In order to submit our model to Kaggle we needed to clean the testing data set we were given. We cleaned the data in the same way that we cleaned the training data. However, we found that the testing data set also had some NaN values for 'Fare'. We handeled this the same way we handled 'Age', by filling these values with the mean of all the Fares. We then ran the logistic regression on the full set of training data and tested it on the full set of testing data and saved the results alongside the passenger Ids to submit the results to Kaggle. We submitted the prediction results to Kaggle and got 75.598% accuracy with rank #8824. We did attempt to remove features one by one from the training data, after splitting it into a train and test set, to see if there was any affect on training accuracy. However, there was not a particular feature removal that increased the training accuracy. Thus, using all of the features that were available (minus some that we removed based on logic as given in the code) the logistic regression model is not very robust for this data.

2 Written Exercises

2.1 Exercise 1

The variance of a random variable X is given by $Var(X) = E[(X - E[X])^2]$, where E denotes the expected value [4]. This expression can be expanded in the following way:
$= E[X^2 + E[X]^2 - 2XE(X)] = E[X^2] + E[X]^2 - 2E[X]E[X] = E[X^2] - E[X]^2$.

Using this expression for X-Y we get
$Var(X - Y) = E[(X - Y)^2] - E[X - Y]^2$.
We can expand this as follows:
$= E[X^2 + Y^2 - 2XY] - E[X - Y]^2$
$= E[X^2] + E[Y^2] - 2E[XY] - E[X - Y]^2$
$= E[X^2] + E[Y^2] - 2E[XY] - (E[X] - E[Y])^2$
$= E[X^2] + E[Y^2] - 2E[XY] - E[X]^2 - E[Y]^2 + 2E[X]E[Y]$
$= E[X^2] - E[X]^2 + E[Y^2] - E[Y]^2 + 2E[X]E[Y] - 2E[XY]$
$= Var(X) - Var(Y) + 2E[X]E[Y] - 2E[XY]$
$= Var(X) - Var(Y) - 2(2E[XY] - E[X]E[Y])$.
We have:
$Var(X - Y) = Var(X) - Var(Y) - 2(2E[XY] - E[X]E[Y])$ (Eq. 1)

We know that the definition of Covariance between two random variables X and Y is:
$Cov(X, Y) = (E[X - E[X]])(E[Y - E[Y]])$.
Expanding this we obtain:
$= E[XY - XE[Y] - E[X]Y + E[X]E[Y]]$
$= E[XY] - 2E[X]E[Y] + E[X]E[Y]$
$= E[XY] - E[X]E[Y]$.
Applying this to Eq.1 we get:
$Var(X - Y) = Var(X) + Var(Y) - 2Cov(X, Y)$.

2.2 Exercise 2

a) We want to know the probability that the widget is defective given the test result. The equation for Bayes law for the probability of the widget being defective given a positive test result is as follows.

$P(defective|test+) = \frac{P(test+|defective)P(defective)}{P(test+)}$ (Eq. 2)

The equation for Bayes law for the probability of the widget being defective given a negative test result is as follows.

$P(defective|test-) = \frac{P(test-|defective)P(defective)}{P(test-)}$ (Eq. 3)

From the problem statement we know that the probability of testing positive, given that the widget is defective is 0.95 and the probability of testing negative given that the widget is not defective is 0.95. These probabilities are given as follows.
$P(test + |defective) = 0.95$
$P(test - |not\_defective) = 0.95$ (Eq. 5)

We know that only 1 in 100,000 items can be defective:
$P(defective) = \frac{1}{100000} = 0.00001$

$P(not\_defective) = \frac{99999}{100000} = 0.99999$

Now that we have all the values, we can calculate the probability that the test is positive and the probability that the test is negative:
$P(test+) = P(test + |defective)P(defective) + P(test + |not\_defective)P(not\_defective)$
$= 0.95 * 0.00001 + (1 - 0.95) * 0.99999$
$= 0.05$

$P(test-) = P(test - |defective)P(defective) + P(test - |not\_defective)P(not\_defective)$
$= (1 - 0.95) * 0.00001 + 0.95(0.99999)$
$= 0.9405$

Plugging in the probabilities that we found into Eq. 2:
$P(defective|test+) = \frac{0.95*0.00001}{0.05} = 0.00019$
Plugging in the probabilities that we found into Eq. 3:
$P(defective|test-) = \frac{(1-0.95)*0.00001}{0.9405} = 5e-7$

b) The factory makes 10^7 widgets a year, and 1 in 100,000 is defective. Therefore there are $\frac{10^7}{10^5} = 100$ defective widgets produced each year, and $10^7 - 100 = 9,999,900$ non defective widgets made in a year. The probability that a widget tests positive and is not defective is 0.05, and the probability that a widget tests negative when it is defective is 0.05. Thus, the amount of good widgets thrown in a year is $0.05*(10^7-100) = 499,995$ widgets. And the amount of defective widgets sold in a year is $0.05 * 100 = 5$ widgets.

2.3 Exercise 3

a) For smaller values of k the training prediction error will be smaller because the method chooses the best fit at each point, giving a jagged decision boundary. Here the classifier is overfitting. For larger values of k, although the decision boundary will be smoother, the training prediction error will be higher because the classifier takes into consideration many more points, including outliers which may cause the decision boundary to move away from the best fit.

b) For small values of k the classifier is overfitted to the training data because it does not consider enough neighbor points to be able to generalize to the test data. Therefore, the test prediction error will be large. As we increase k, more neighbors are considered and the classifier is able to generalize to test data. In this case, the test prediction error decreases. Somewhere in this part of the graph we will find the optimal k value. If we keep increasing k, the classifier will start underfitting the training data. At this point, the classifier becomes too general, even for the test data, and both the test prediction error and the training prediction error increase. This can be illustrated in the sketch shown in Figure 8.

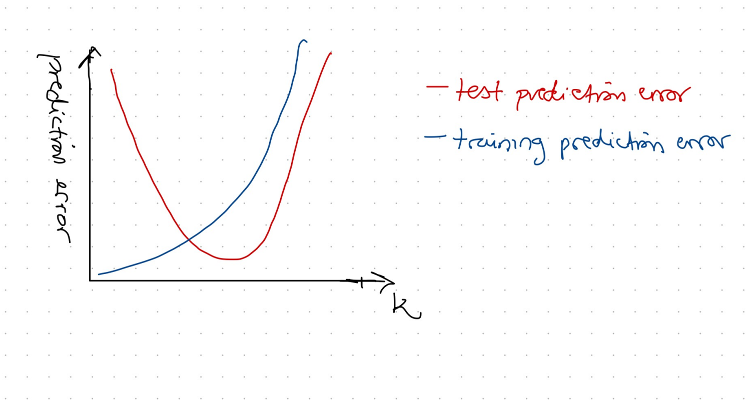

Figure 8: Sketch of Test and Train Prediction Error Changes as k in creases in KNN.

c) In theory the more folds you use for crossvalidation the better. The validation accuracy will increase as you increase the number of folds, as long as you ensure that the test and train data sets are large enough to be able to properly test and train the data. If the number of folds is too high, for example, you get to leave-one-out crossvalidation, the validation accuracy might begin to dwindle. Also, as you increase the number of folds the computational complexity will increase because you will be running the training and testing K times, where K is the number of folds. About 10 folds should be enough to allow for a good enough validation accuracy and a reasonable computation time.

d) A potential modification to the KNN algorithm, when k is large, could be to split the neighbors into two groups. The first group would be the first half of neighbors closest to the point, and the second group would be the half of the neighbors furthest from the point. If the number of points is odd, the mid point would be assigned to the second group. The points in the first group can then be weighted twice as much as the points in the second group.

e) KNN might be undesirable when the input dimension is high because as we increase the number of features, the amount of data points that we need for the classifier to generalize increases in an exponential fashion. Basically, the data becomes sparse relative to the space. This is often know as the curse of dimensionality. Secondly, the L2 norm or Euclidean distance calculates distance between all features, squares them and adds them up. When there are many dimensions the distance measures might be similar for samples that are not best matches because they may have had the right combination of differences and similarities with different features that caused them to match.

References

[1] T. E. Oliphant, *A guide to NumPy*. Trelgol Publishing USA, 2006, vol. 1.

[2] Quora. Eer. "https://www.quora.com/How-can-I-understand-the-EER-Equal-Error-Rate-and-why-we-use-it"[Accessed 2019-09-23].

[3] W. McKinney, "Data structures for statistical computing in python," *Proceedings of the 9th Python in Science Conference*, pp. 51–56, 2010.

[4] Wikipedia. Variance. "https://en.wikipedia.org/wiki/Variance"[Accessed 2019-09-23].

A Code to Generate Graphs

A.1 Code for Labels Histogram (1c)

```
training_data=pd.read_csv("Digit_Recognition_Data/train.csv")

labels=training_data['label'].values # extract label values

# function to calculate the prior probability of the class indicated by the label parameter
def calculatePrior(label):
    return np.count_nonzero(labels == label)/(42000)


# print the prior probability for each MNIST digit
for i in range(10):
    print ("Prior probability for ", i, ": ", calculatePrior(i))

# plot a histogram of the labels, density = True is for normalization
plt.hist(labels, bins = 10, density = True)
plt.xticks(np.arange(10), ['0','1','2', '3', '4', '5', '6', '7', '8', '9'])
plt.title('MNIST Digits Frequency Histogram')
plt.xlabel('MNIST Digits') # label x axis
plt.ylabel('Normalized Count') # label y axis
```

A.2 Code for Genuine and Imposter Distributions (1e)

```
# function to calculate the Euclidean or L2 norm distance
def euclideanDist(img1, img2):
    return np.linalg.norm(img2-img1)

zerosIndexes = [] # list to store the indexes of all the '0' digits
onesIndexes = [] # list to store the indexes of all the '1' digits

for i, label in enumerate(labels):
    if label == 0: # if it is a '0' store in list for zeros
        zerosIndexes.append(i)
    elif label == 1: # if it is a '1' store in list for ones
        onesIndexes.append(i)

distZeros = [] # list to store distances between all '0', or genuine '0', pairs

for indx, i in enumerate(zerosIndexes):
    for j in zerosIndexes[indx+1:]:
        distZeros.append(euclideanDist(data[i], data[j]))

distOnes = [] # list to store distances between all '1', or genuine '1', pairs

for indx,i in enumerate(onesIndexes):
    for j in onesIndexes[indx+1:]:
        distOnes.append(euclideanDist(data[i], data[j]))
distImposters = [] # list to store distances between all '0'-'1', or impostor, pairs

for indx,i in enumerate(zerosIndexes):
    for j in onesIndexes[indx+1:]:
        distImposters.append(euclideanDist(data[i], data[j]))

for indx,i in enumerate(onesIndexes):
    for j in zerosIndexes[indx+1:]:
        distImposters.append(euclideanDist(data[i], data[j]))

distGenuine = distZeros + distOnes # merge genuine distance lists for '0's and '1's

# plot histogram of genuine and impostor pairwise distances
plt.hist(distGenuine, color = 'b', label = 'genuines', bins = 32, alpha = 0.5)
plt.hist(distImposters, color = 'g', label = 'imposters', bins = 32, alpha = 0.5)
plt.xlabel('Distance') # label x axis
plt.ylabel('Counts') # label y axis
plt.legend() # add legend
```

A.3 Code for ROC Curve of Genuine and Imposter Distributions (1f)

```
ROCPointsX = [] # list to store the X coordinate of points in the ROC curve
ROCPointsY = [] # list to store the Y coordinate of points in the ROC curve
# merge genuine and impostor distances into a single list
distTotal = distGenuine + distImposters

# calculate each ROC curve point
for j in range(0, math.ceil(max(distTotal)), 250):

    FP = sum(i > j for i in distGenuine)

    TN = sum(i < j for i in distGenuine)

    FN = sum(i < j for i in distImposters)

    TP = sum(i > j for i in distImposters)

    TPR = TP / (TP + FN)
    FPR = FP / (FP + TN)

    ROCPointsY.append(TPR) # add TPR value to list
    ROCPointsX.append(FPR) # add FPR value to list

# plot ROC curve
plt.plot(ROCPointsX, ROCPointsY, label='ROC_curve')
x = np.linspace(0,1,100)
# plot random chance for comparison
plt.plot(x, x, color='r', linestyle='dashed', alpha = 0.5, label='ROC random chance')
plt.xlabel("FPR") # label x axis
plt.ylabel("TPR") # label y axis
plt.legend() # add legend
```

A.4 Code for 3-NN Classifier Confusion Matrix (1i)

```
# create confusion matrix with predictions and actual values for our KNN classifier
confusion_m = confusion_matrix(my_actuals, my_predictions)
print(confusion_m)

# display the confusion_matrix in an intuitive way
fig, ax = plt.subplots()
imshow(confusion_m)
plt.colorbar()
classes = [0,1,2,3,4,5,6,7,8,9]
ax.set(xticks=classes,yticks=classes,xticklabels=classes,yticklabels=classes,ylabel='True',
        xlabel='Predicted');
```

A.5 Code for Logistic Regression Confusion Matrix (2b)

```
# split data into test and train set for testing
x_train, x_test, y_train, y_test =
        train_test_split(training_data_t_clean.drop('Survived', axis=1),
           training_data_t_clean['Survived'], test_size=0.25)

# perform logistic regression
lr = LogisticRegression()
lr.fit(x_train, y_train)

Ps = lr.predict(x_test) # store predictions
```

```
# understand accuracy
conf_matrix = confusion_matrix(y_test, Ps)

print('The accuracy of our model is',((accuracy_score(y_test, Ps))*100))
print(conf_matrix)

# display the confusion matrix in an intuitive way
fig, ax = plt.subplots()
imshow(conf_matrix)
plt.colorbar()
classes = [0,1]
ax.set(xticks=classes,yticks=classes,xticklabels=classes,yticklabels=classes,ylabel='True',
        xlabel='Predicted');
```