CS5785 HW1

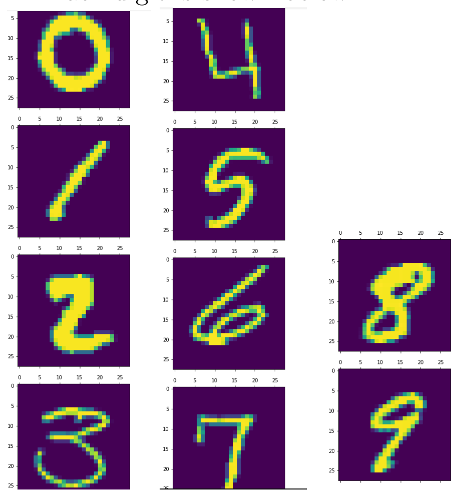
Andrew Palmer, Kushal Singh

26 September 2019

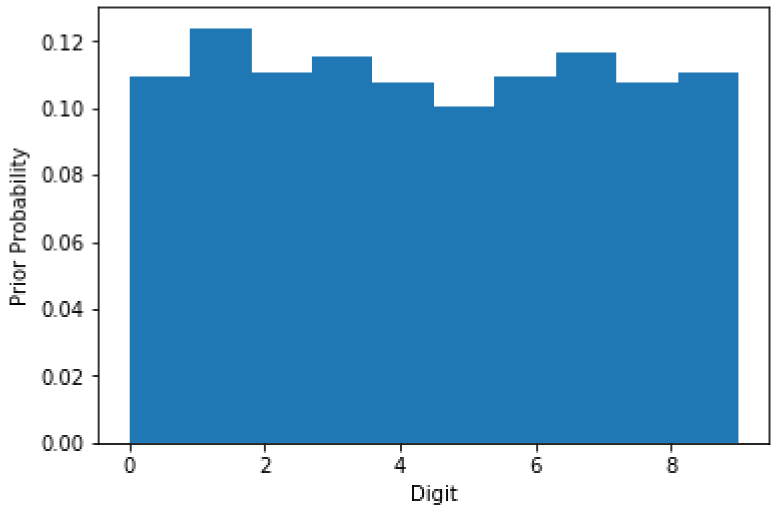
1 Digit Recognizer

(b) To show each specific digit, we filtered the label/pixels dataframe by the specific digit label we wanted and took the first occurrence of a digit. Once we had the desired digit's pixel values, we made use of matshow[1], which generates an image from a matrix, to visualize the digit. Since we know each image is a 28x28 pixel matrix, we reshaped the 784 pixel brightness values in the row vector into a 2D matrix before passing into matshow.

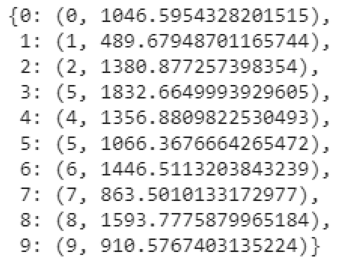
Each digit is shown below:



(c) To examine the prior probability of the digits in the training data, we generated a normalized histogram of digit counts. As can be seen in the histogram, the prior probability is not uniform across all digit classes. But, for the most part, each digit happens to appear between roughly 10-12 percent of the time, which is fairly balanced, considering there are nearly 42000 entries in the training set.

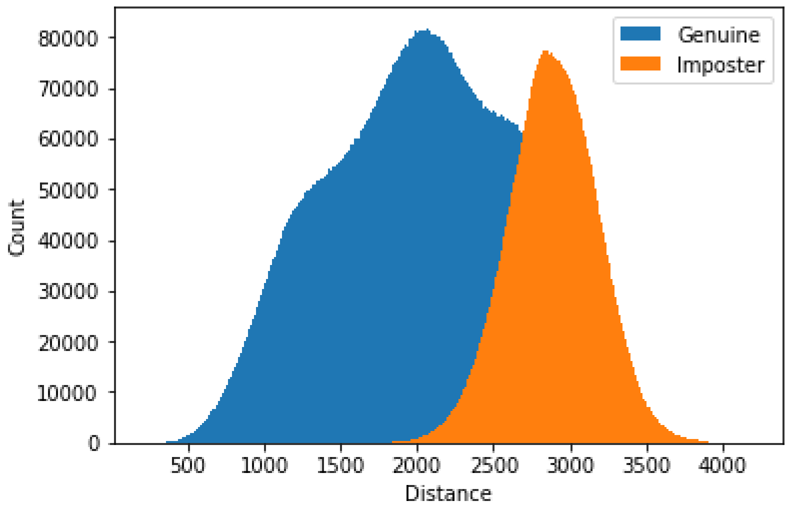


(d) To generate one example of each digit from our training data, we called the unique() function on the 'label' column, thereby generating an array that contains ten different label values, from 0 to 9, inclusive. In order to store the best match (nearest neighbor) between the chosen sample and the rest of the training data, we used a dictionary, where the true digit was the key, and the value was a tuple that contained the label of the nearest neighbor and the distance to that nearest neighbor. The metric we use to find the closest neighbor was L2 distance (aka euclidean distance), which we calculated via the np.lingalg.norm function. Our nearest neighbor values for each digit are shown below.

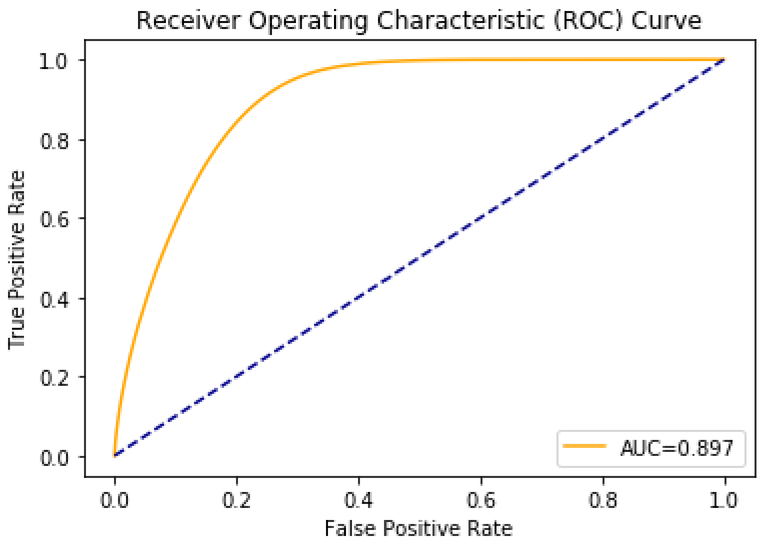


```
{0: (0, 1046.5954328201515),
 1: (1, 489.67948701165744),
 2: (2, 1380.877257398354),
 3: (5, 1832.6649993929605),
 4: (4, 1356.8809822530493),
 5: (5, 1066.3676664265472),
 6: (6, 1446.5113203843239),
 7: (7, 863.5010133172977),
 8: (8, 1593.7775879965184),
 9: (9, 910.5767403135224)}
```

(e) The histogram of genuine (0's/0's and 1's/1's) distances and imposter (0's/1's) distances is shown below. Distances were computed pairwise using pdist[2] for genuine and cdist[3] for imposter. From the histogram, there are clear distribution of distances between the genuine and imposter distances. However, one can also see there is non-trivial overlap between the two distributions.



(f) Shown below is the ROC curve from our genuine and imposter distances. To generate this, we needed the False Positive Rates and True Positive Rates at different thresholds. We used sklearn.metrics.roc_curve[4] to generate the required values and then plotted these values on an ROC curve.
The Equal Error Rate is determined by the intersection of a our curve and a straight line joining (1,0) and (0,1). On our graph, the EER came out to be 0.18557486137996923. The Equal Error Rate of a classifier that is randomly guessing is simply where the lines (0,0) (1,1) and (1,0) (0,1) intersect, which is at 0.5



(g) To implement the KNN classifier, we defined three different functions: a euclidean distance function, a getNeighbors function, and a predictKNNClass function. The euclidean distance function uses np.power(), np.sum(), and np.sqrt() to calculate the distance between two digits' pixel values. The getNeighbors function takes in the training data set, a query point, and a value for k, and returns the indices of the k closest neighbors to the query point, using the euclidean distance function defined above. The predictKNNClass function takes in the result from the getNeighbors function, and takes in a vector of labels, and counts up the total number of votes for each label, returning the label with the most votes.

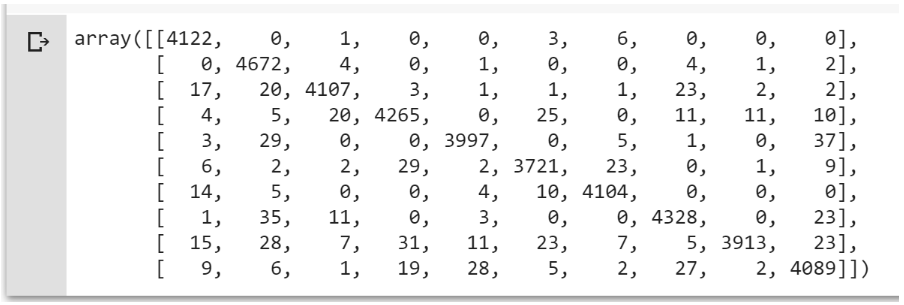
(h) To generate the average accuracy from 3-fold cross validation, we used sklearn's KNeighborsClassifier with $n = 5$ nearest neighbors, and sklearn's $cross\_val\_score$ function, which gave us the following results: [0.96458155, 0.96328047, 0.96478068]. These results show us that during the first iteration of cross-validation, where the first fold was used as the testing set and the second and third folds were used as the training set, our success rate was 96.4 percent.

Similarly, during the second iteration, where the second fold was used as the testing set and the first and third folds were used as the training set, our success rate was 96.3 percent.

Finally, during the third iteration, where the third fold was used as the testing set and the first and second folds were used as the training set, our success rate was 96.4 percent.

Combining these errors, we see that the average error rate is equal to 0.9642142333, which is roughly equal to 96.4 percent. These results make sense. Namely, since kNN is a lazy learning algorithm, every time it is presented with new test data, it has to run the entire algorithm on the training set, which means that each test fold is roughly independent of the other test folds.

(i) To generate the confusion matrix, we made a call to the $confusion\_matrix$ function from sklearn. These are the results that we generated.



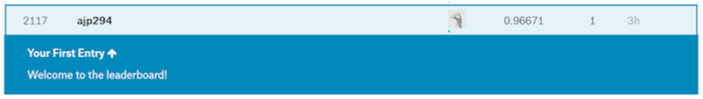
```
array([[4122,    0,    1,    0,    0,    3,    6,    0,    0,    0],
       [   0, 4672,    4,    0,    1,    0,    0,    4,    1,    2],
       [  17,   20, 4107,    3,    1,    1,    1,   23,    2,    2],
       [   4,    5,   20, 4265,    0,   25,    0,   11,   11,   10],
       [   3,   29,    0,    0, 3997,    0,    5,    1,    0,   37],
       [   6,    2,    2,   29,    2, 3721,   23,    0,    1,    9],
       [  14,    5,    0,    0,    4,   10, 4104,    0,    0,    0],
       [   1,   35,   11,    0,    3,    0,    0, 4328,    0,   23],
       [  15,   28,    7,   31,   11,   23,    7,    5, 3913,   23],
       [   9,    6,    1,   19,   28,    5,    2,   27,    2, 4089]])
```

From this confusion matrix, we can see that the digits 4 and 9 were particularly trickiest to classify, as 4 was correctly classified 3997 times as the digit four, while incorrectly classified 37 times as the digit nine. Similarly, 9 was correctly classified 4089 times as the digit nine, while incorrectly classified 28 times as the digit four. This makes sense because depending on how four is written, it can, at times, be made to look like a nine, and vice-versa.

Additionally, one was classified as a seven 35 times. Depending on how the 1 digit is written, it can often have an additional piece on the upper left, making it look like a seven. Interestingly, the 7 was not often classified as a 1. While a 1 may have the extra piece, like a 7, 7 will generally not be missing the upper left part.

Another hard to classify digit is the 3 versus the 8. This can also be analyzed intuitively as if you connect the lower and upper hemispheres of the 3, it becomes an 8. Depending on the individual's handwriting, the 8 and 3 could look similar in terms of euclidean distance between pixels.

(j) For this part, we used sklearn's KNeighborsClassifier to train the MNIST digit data set. We ended up getting a 96.6 percent on sklearn's classifier.



```
2117   ajp294                                  0.96671   1   3h

Your First Entry ↑
Welcome to the leaderboard!
```

2 The Titanic Disaster

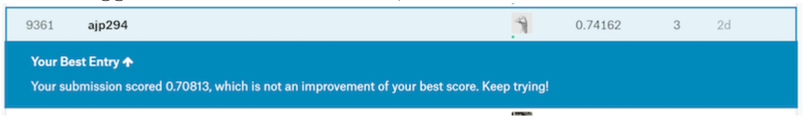
(b) Features Used:
- Age
- Sex
- Fare
- PClass

The above features were chosen to train our logistic regression model to best determine the outcome of a passenger surviving the Titanic disaster. Having the benefit of historical knowledge, we know that women and children were generally evacuated into lifeboats first. Thus, we determined that age and sex would be strong indicator features for survival rate. Additionally, we postulated that the upper class were loaded into lifeboats before those of lower classes. The PClass feature enumerates a passenger's class. Being a proxy for class, Fare was also used to train our model based on socio-economic status.

We also thought about using the sibsp and parch features, as there could be a higher likelihood of survival if a passenger had family members aboard. However, when we included these in our model, we received a score of 0.70813 compared to 0.74162 without the features. Looking closer at the data description, a feature like SibSp and Parch describes relationships among children and adults in the same feature. Thus, although a child may be more likely to survive if the child's parents are aboard, the same cannot necessarily be said for a father who had many children aboard. Because we know women and children were given priority, including these features for a male may reduce the effectiveness of our model.

Based on our findings about SibSp and Parch features, it could be beneficial to feature engineer using the given data. For instance, we believe that the SibSp and Parch features would be stronger indicators of survival for children who have siblings and parents aboard, but not for parents, especially male, who have a spouse or children aboard. Therefore, we could create our own feature to increase the granularity of the SibSp and Parch features to separate between children and parents and also mother and father.

(c) After analysis completed in part b for determining our classifier's desired features. We submitted our results to Kaggle. Although we submitted multiple times to Kaggle with various features, our best result was 0.74162.


9361 ajp294 0.74162 3 2d

Your Best Entry

Your submission scored 0.70813, which is not an improvement of your best score. Keep trying!

3 Written Exercises

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 co-variance between random variables X and Y.

Solution

$Var(X) = E[X^2] - E[X]^2$ (1)

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

$Var(X - Y) = E[X^2 - 2XY + Y^2] - E[X - Y]^2$ (3)

$Var(X - Y) = E[X^2] + E[Y^2] - 2E[XY]$ and $E[X - Y] = E[X] - E[Y]$ (By linearity of expectation) (4)

So, replacing equation (3) above, we get:

$Var(X - Y) = E[X^2] + E[Y^2] - 2E[XY] - (E[X] - E[Y])(E[X] - E[Y])$

$Var(X - Y) = E[X^2] + E[Y^2] - 2E[XY] - (E[X]^2 - 2E[X]E[Y] + E[Y]^2)$

$Var(X - Y) = E[X^2] - E[X]^2 + E[Y^2] - E[Y]^2 - 2(E[XY] - E[X]E[Y])$

$Var(X - Y) = Var(X) + Var(Y) - 2 * cov[X, Y]$

2. Bayes rule for quality control. You're the foreman at a factory making 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
Given Bayes:
$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$

Bayes for our problem:
$P(Def|PosTest) = \frac{P(Def)P(PosTest|Def)}{P(Def)P(PosTest|Def)+P(PosTest|NotDef)P(NotDef)}$

$P(Def) = \frac{1}{100{,}000} = 0.00001$ given from the problem statement.

$P(PosTest|Def) = 95\% = .95$ also given from the problem statement.

$P(PosTest|NotDef) = 5\% = .05$, which is determined by $1 - .95 = .05$.

$P(NotDef) = 99.999\% = .99999$, which is determined from $1 - P(Def) = 1 - .00001$

Plugging the values in to Bayes:

$P(Def|PosTest) = \frac{0.00001*0.95}{0.00001*0.95+0.05*0.00009}$

$P(Def|PosTest) = 0.0001897$

(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
Good widgets thrown away per year:

$good\_thrown\_out = P(PosTest|NotDef) * P(NotDef) * total\_units$
$good\_thrown\_out = 0.05 * 0.99999 * 10,000,000$
$good\_thrown\_out = 499,995$ units per year

Bad widgets shipped per year:

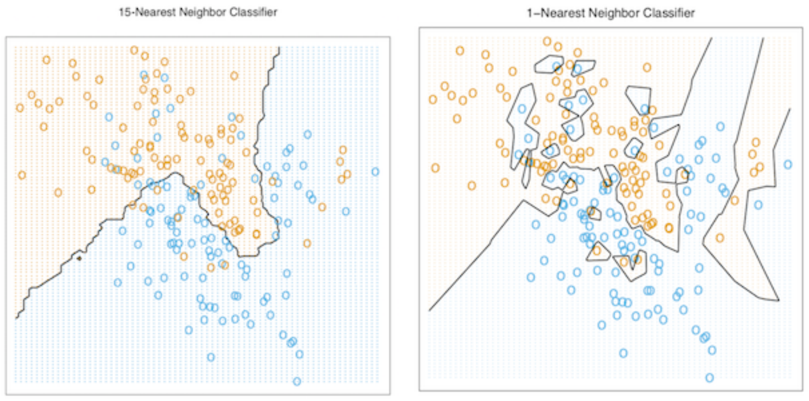
$bad\_shipped = P(NegTest|Def) * P(Def) * total\_units$
$bad\_shipped = 0.05 * 0.00001 * 10,000,000$
$bad\_shipped = 5$ units per year

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 x_i includes (x_i, y_i) as part of the example training data used by kNN.)

Solution
When $k = n$, the training error should have an expected value of 1/2. When $k = 1$, there will be no training error since every x_i will predict its own y_i label. As k varies, the average prediction error should lie between these values.

(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
At $k = 1$, the generalization error should be high, due to potential overfitting. As k increases, the model should be able to generalize a lot better, which means the error should be lower. I would assume that, at some point, as k increases a lot, the error rate won't change that much and may level off.



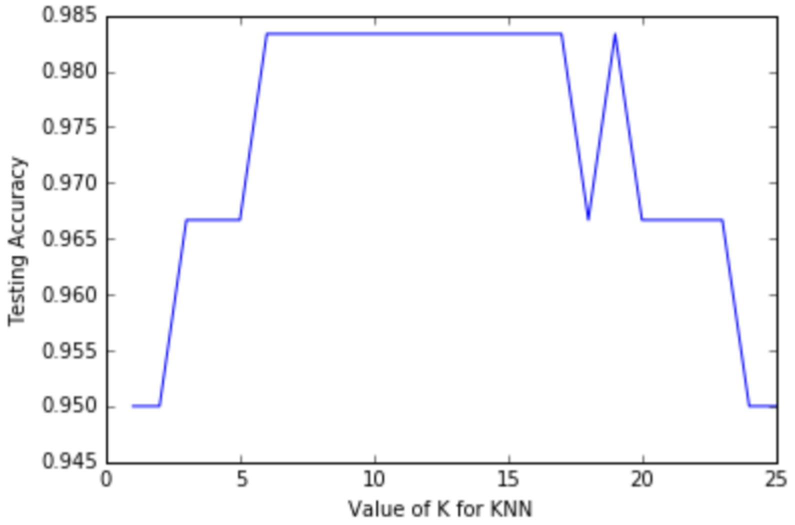
A large part of the prediction error also relies on how many of each digit we choose to train our data on. If we remove half of the data, which coincidentally contains a bunch of information on the digits 0-4, and test on the held-out half that consists predominantly of the digits 5-9, then our classifier will predict fairly poorly, since our training and testing sets contain different information.

Picture from The Elements of Statistical Learning (Data Mining, Inference, Prediction)

(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
Computational requirements in kNN, especially at higher dimensions, increasing dramatically as the number of k increases. At low values of k, our computational requirements are reduced; however, the accuracy can suffer as well. At high values of k, the compute resources greatly increase because of the feature vector's size. Computing pairwise Euclidean distance scales massively with increased number of features.

Additionally, as can be seen in the graph below[5], increasing values of k eventually have diminishing returns for the additional compute cost. Therefore, it's important to empirically determine an appropriate value of k for one's situation to balance compute requirements and accuracy. For the below graph, one could say that a k value of about 6 or 7 provides the highest accuracy per compute resources.



(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
When k is large, we inherently include more samples as neighbors in our voting of the nearest neighbor to choose. However, not all k values will be have the same numerical similarity to our sample. Therefore, we may want to weigh neighbors that are closer to our sample more heavily than neighbors that are far away. This is what happens in the Epanechnikov kernel, which we have discussed in lecture.

For example, if k were 20, we could give weight to each of the k's vote. The samples at k values 1-5 would have a higher weight to their votes than the samples at k values 16-20. In contrast to this bucketing of samples, there could be a function of each sample's distance measure to determine the weight of its vote.

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

Solution
1. One reason why kNN suffers when input dimension is high relates to the distance measure used. When the input dimension is high, the computational cost of calculating distances increases dramatically as dimensions increase.
2. Additionally, kNN benefits when data is highly clustered. When dimensions are increased, data becomes more sparse, which reduces the effectiveness of finding near-neighbors.

References

[1] https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.matshow.html

[2] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html

[4] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_curve.html

[5] https://www.ritchieng.com/machine-learning-k-nearest-neighbors-knn/