KNN

September 17, 2018

1 K-Nearest-Neighbor

1.1 Introduction

In this document we will be exploring the creation of a K-Nearest-Neighbor algorithm for digit classification on the MNIST dataset. We will take the 60,000 images run 10-fold cross validation on them to find the optimal value of k nearest neighbors, then we will run the algorithm on the test data provided and determine the accuracy, confidence interval for 95% and create a confusion matrix for the results. Then we will repeat the experiment with a sliding window technique in the KNN and run a significance test on the algorithm to determine which one preformed better.

1.2 Import Required Libraries

```
In [ ]: from __future__ import division
        import struct
        import gzip
        import numpy as np
        import pandas as pd
        from math import sqrt
        import seaborn as sn
        import matplotlib.pyplot as plt
        from scipy.spatial.distance import cdist
        from scipy.spatial.distance import euclidean
        from tqdm import tqdm_notebook as tqdm
```

1.3 A function to read the MNIST data set as a numpy array

```
In [ ]: def read_idx(filename):
            with gzip.open(filename) as f:
                zero, data_type, dims = struct.unpack('>HBB', f.read(4))
                shape = tuple(struct.unpack('>I', f.read(4))[0] for d in range(dims))
                return np.frombuffer(f.read(), dtype=np.uint8).reshape(shape)
```

1.4 Declare our dataset arrays

```
In [ ]: # Create a numpy array for the training data from the mnist dataset
        raw_train = read_idx('data/train-images-idx3-ubyte.gz')
        ## Flatten the training array
        train_data = np.reshape(raw_train, (60000, 28 * 28))
        train_label = read_idx('data/train-labels-idx1-ubyte.gz')

        # Create a numpy array for the test data from the mnist dataset
        raw_test = read_idx('data/t10k-images-idx3-ubyte.gz')
        ## Flatten the test array
        test_data = np.reshape(raw_test, (10000, 28 * 28))
        test_label = read_idx('data/t10k-labels-idx1-ubyte.gz')
```

```
In [ ]: ## Format new data into 30x30 for sliding window technique

        raw_test_data_30x30 = []

        for i in range(len(raw_test)):
            raw_test_data_30x30.append(np.pad(raw_test[i], ((1,1), (1,1)), 'constant', constant_

        raw_test_data_30x30 = np.array(raw_test_data_30x30)
        test_data_30x30 = np.reshape(raw_test_data_30x30, (10000, 30 * 30))
```

1.5 Euclidean Distance

```
In [ ]: def euclideanDistance(X, Y):
            return np.linalg.norm((X - Y));
```

1.6 Find the Nearest Neighbors

This function finds the nearest neighbors by creating an array the same size as the training data and populates that array with the image that we are trying to find the distances for compared to the other training data. Then the array is sorted based on the first k elements and the indexes are recorded and returned.

In the sliding windows approach we loop over the 30x30 image with boxes of 28x28 and calculate the euclidean distance for each box, then store the results in an array. After the image has been cropped 9 different times we take the smallest distance and use that as the distance for that particular piece of test data.

We had to define a special function for finding the optimal k value because of how the distance matrix works. This required us to separate the loop into separate pieces based on which training data it is currently evaluating to get the proper indices.

```
In [ ]: def nearestNeighbors(train_data, i, k, slidingWindow):
            distances = []
            n = train_data.shape[0]
            # Sliding window loops over train data and the crops array and finds the distances b
            # the train data at that j.
            if slidingWindow:
                for j in range(len(train_data)):
                    cropDistances = []
                    for k in range(9):
                        crop = crops[i][k]
                        cropDistances.append(euclideanDistance(crop, train_data[j]))
                    distances.append(np.sort(cropDistances)[0])
            else:
                for j in range(len(train_data)):
                    distances.append(distance_array[i][j])


            neighbors_idxs = np.argsort(distances)[:k]

            return neighbors_idxs
```

```
In [9]: def nearestNeighborsOptimal(i, k, k_fold, subset_size):
            # Special function for determing the optimal k value
            distances = []

            for j in range(k_fold * subset_size):
                distances.append(distance_array[i][j])

            for j in range(((k_fold + 1) * subset_size), len(train_data)):
                distances.append(distance_array[i][j])

            neighbors_idxs = np.argsort(distances)[:k]

            return neighbors_idxs
```

1.7 K Nearest Neighbor

The K Nearest Neighbor algorithm works by looping through each value in the test_data, determines the nearest neighbor, then predicts the result based on the point that it's closest to.

The function returns an array of predictions.

Special function for finding the optimal k requires it to call the its own special nearest neighbor function.

```
In [ ]: def kNearestNeighbor(train_data, train_labels, test_data, k, slidingWindow=False):
            predictions = []

            for i in tqdm(range( len( test_data ) )):
                neighbors = nearestNeighbors(train_data, i, k, slidingWindow)
                predictions.append(predict(neighbors, train_labels))

            return np.array(predictions)
```

```
In [ ]: def kNearestNeighborOptimal(k, k_fold, subset_size, training_labels):
            predictions = []
            for i in tqdm( range((k_fold * subset_size), ((k_fold  + 1) * subset_size)) ):
                neighbors = nearestNeighborsOptimal(i, k, k_fold, subset_size)
                predictions.append(predict(neighbors, training_labels))

            return np.array(predictions)
```

1.8 Cross Validation

This function runs the cross validation on the specified number of folds. We are using 10-fold cross validation so that is the only value that will be passed later on in our code.

This works by looping through the array 10 times and uses each subset that is 1/10 the size as testing data, at the same time using the rest as training data. The accuracies are then recorded and averaged to get the mean accuracy for that particular test. We run this function for every value of k to find the optimal amount of nearest neighbors.

The usefulness of running cross validation is the fact that it gives us an idea of the performance on the entire dataset rather than a specific subset in every case. This essentially gives us a more diverse group of testing data and a more accurate accuracy with the same data set.

```
In [ ]: def crossValidation(num_folds, k):
            subset_size = len(train_data) // num_folds
            accuracyList = []

            for i in range(num_folds):
                testing_labels = train_label[i*subset_size:(i + 1)*subset_size]
                training_labels = np.concatenate((train_label[:i*subset_size], train_label[(i +
                predictions = kNearestNeighborOptimal(k, i, subset_size, training_labels)
                accuracyList.append(accuracy(predictions, testing_labels))

            meanAccuracy = sum(accuracyList) / float(len(accuracyList))
            print("The accuracy when k=" + str(k) + " is: "+ str(meanAccuracy), flush=True)
            return meanAccuracy
```

1.9 Predict

This function takes in an array of neighbors and the training labels then adds the label of the neighbor to the results array for each nearest neighbor.

The function then returns the prediction based off what label appears with the greatest frequency in the results array. This is done with the numpy np.bincount function that finds the greatest frequencies in an array format, then uses the np.argmax function to find the index of the max labels that appears and returns the result.

```
In [ ]: def predict(neighbors, train_labels):
            results = []
            for idx in neighbors:
                results.append(train_labels[idx])
            results = np.array(results)
            return np.argmax(np.bincount(results));
```

1.10 Accuracy

```
In [ ]: def accuracy(predictions, test_labels):
            return np.sum(predictions == test_labels) / len(test_labels)
```

1.11 Find the Optimal K

Finds the optimal value of k by looping through k values 1-10 and determines the optimal k by saving the k with the highest classfication accuracy.

```
In [ ]: def findOptimalK():
            optimal_k = 1
            optimal_accuracy = -1
            k_min = 1
            k_max = 10
            k_accuracy = -1;

            for i in range(6, k_max + 1):
                k_accuracy = crossValidation(10, i)

                if k_accuracy > optimal_accuracy:
                    optimal_k = i
                    optimal_accuracy = k_accuracy

            #optimal_k = k_accuracy.index(max(k_accuracy)) + 1
            print("The optimal value of k is: ",optimal_k," with an accuracy of ",(100 * optimal
            return optimal_k
```

1.12 Calculate Confidence Interval

We find the confidence interval using the z-score of the 95% confidence interval in this case. This is found by computing the z square multiple by the square of the accuracy times the error divided by the length.

```
In [ ]: def calculateConfidenceInterval(accuracy, length, z):

            interval = z * sqrt( (accuracy * (1 - accuracy)) / length )

            return interval
```

1.13 Create Confusion Matrix

Find the confusion matrix by putting the data in a pandas series then create a confusion matrix with pandas crosstab method. This compares the predictions to the actual results and marks the areas in the matrix with what is wrong and what is right. The diagonal of the matrix is the correct predictions. A confusion matrix helps us visualize what are algorithm gets wrong most often.

```
In [ ]: def createConfusionMatrix(predictions, test_labels):

            y_actu = pd.Series(test_labels, name='Actual')
            y_pred = pd.Series(predictions, name='Predicted')
            df_confusion = pd.crosstab(y_actu, y_pred)

            df_cm = pd.DataFrame(df_confusion)
```

```
        sn.set(font_scale=1.4)
        plt.figure(figsize = (12,8))
        sn.heatmap(df_cm, annot=True,annot_kws={"size": 16}, fmt='g')
```

1.14 Image Cropper

Crops the image by taking the pixels within a specific range.

```
In [ ]: def imageCropper(img, xstart, ystart):
            return img[ystart: ystart + 28, xstart: xstart + 28]
```

2 K Nearest Neighbor k=1

Runs the nearest neighbor algorthim for k=1

```
In [ ]: distance_array = cdist(test_data, train_data, 'euclidean')
```

```
In [ ]: k1_predictions = kNearestNeighbor(train_data, train_label, test_data, 1)

        k1_accuracy = accuracy(k1_predictions, test_label)
```

```
In [ ]: print("Classifcation accuracy:", k1_accuracy * 100, "%")
```

```
Classifcation accuracy: 96.91 %
```

3 Find the Optimal K

```
In [ ]: ## Find the optimal value of k
        distance_array = cdist(train_data, train_data, 'euclidean')
        optimal_k = findOptimalK()
```

```
The accuracy when k=1 is: 0.9705

The accuracy when k=2 is: 0.9645000000000001

The accuracy when k=3 is: 0.9711666666666666

The accuracy when k=4 is: 0.9695666666666668

The accuracy when k=5 is: 0.9701166666666667

The accuracy when k=6 is: 0.9688666666666664

The accuracy when k=7 is: 0.96915

The accuracy when k=8 is: 0.9677166666666668

The accuracy when k=9 is: 0.9672166666666666

The accuracy when k=10 is: 0.9662333333333333
```

Results

Thus after running through all values from of k in the range of 1-10 we found that the optimal value of k is k=3. Which is not suprising as we get to larger values of k we have to worry about the problem over overfitting, which drops the accuracy slightly.

4 Standard K Nearest Neighbor Statistics

```
In [ ]: del distance_array
        ## Uses a distance array and calculates the euclidean distance for it, speeds up executi
        distance_array = cdist(test_data, train_data, 'euclidean')
```

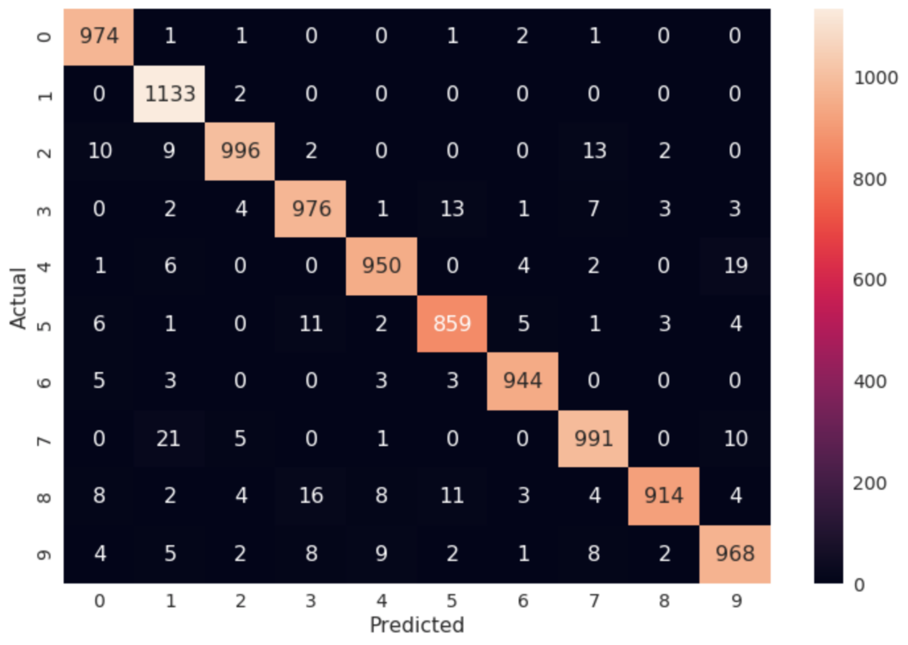
```
In [ ]: standard_predictions = kNearestNeighbor(train_data, train_label, test_data, optimal_k)

        standard_accuracy = accuracy(standard_predictions, test_label)
```

```
In [ ]: standard_interval = calculateConfidenceInterval(standard_accuracy, len(test_label), 1.96

        print("Classifcation accuracy:", standard_accuracy * 100, "%")
        print("Confidence interval for 95%: (", (standard_accuracy - standard_interval) * 100 ,"
        createConfusionMatrix(standard_predictions, test_label)
```

```
Classifcation accuracy: 97.05 %
Confidence interval for 95%: ( 96.7183615709843 %, 97.38163842901571 %)
```



5 Sliding Windows K Nearest Neighbor Statistics

```
In [8]: # This piece of codes finds all the crops of the images first to improve the speed of sl
        crops = np.zeros((10000, 9, 784))
        for k in tqdm(range(len(raw_test_data_30x30))):
```

```
        index = 0
        for i in range(3):
            for j in range(3):
                crop = imageCropper(raw_test_data_30x30[k], i, j)
                crop = np.reshape(crop, (28 * 28))
                crops[k][index] = crop
                index = index + 1
```

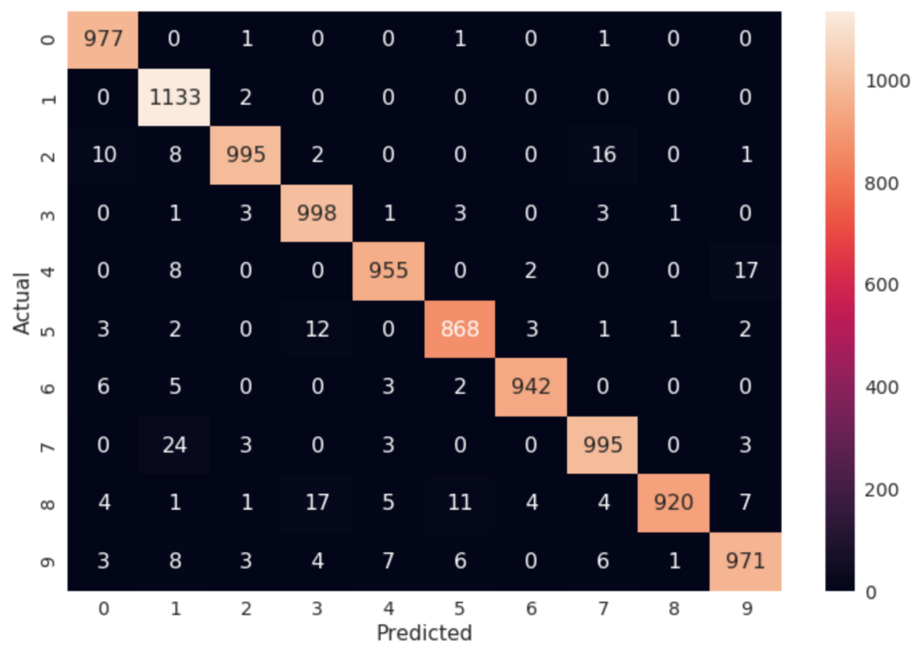
```
In [ ]: sliding_predictions = kNearestNeighbor(train_data, train_label,  raw_test_data_30x30, 3,

        sliding_accuracy = accuracy(sliding_predictions, test_label)
```

```
In [ ]: sliding_interval = calculateConfidenceInterval(sliding_accuracy, len(test_label), 1.96)

        print("Classifcation accuracy:", sliding_accuracy * 100, "%")
        print("Confidence interval for 95%: (", (sliding_accuracy - sliding_interval) * 100 ,"%,
        createConfusionMatrix(sliding_predictions, test_label)
```

```
Classifcation accuracy: 97.54 %
Confidence interval for 95%: ( 97.23639074891565 %, 97.84360925108435 %)
```



Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	977	0	1	0	0	1	0	1	0	0
1	0	1133	2	0	0	0	0	0	0	0
2	10	8	995	2	0	0	0	16	0	1
3	0	1	3	998	1	3	0	3	1	0
4	0	8	0	0	955	0	2	0	0	17
5	3	2	0	12	0	868	3	1	1	2
6	6	5	0	0	3	2	942	0	0	0
7	0	24	3	0	3	0	0	995	0	3
8	4	1	1	17	5	11	4	4	920	7
9	3	8	3	4	7	6	0	6	1	971

6 Significance Testing

6.1 Hypothesis Testing

First we assume that there are two possible hypothesis, that the values of the classification accuracies are the same and the alternative is that the classification accuracies for the two algorithms are different. $H_0 : p_0 = p_1$ and $H_a : p_0 \neq p_1$

Let's assume that p_0 is the classification accuracy of the standard k nearest neighbor algorithm and that p_1 is the classification accuracy of the sliding window k nearest neighbor algorithm.

Thus we need to find the confidence interval of $p_0 - p_1$. Therefore we can find the variance of the confidence interval by calculating that sum of the two variances represented by $\sigma_{p_0-p_1} = \frac{p_0(1-p_0)+p_1(1-p_1)}{20,000}$, where 20,000 is the sum of the two samples sizes.

We found that $p_0 - p_1 = -.0049$ or 0.49%. We need to make sure that this value has a 95% chance of falling within the confidence intervals standard deviation to make sure that this equation $p_0 - p_1$ is actually true. Our standard deviation $d = 0.00317932874$. Therefore we have a 95% confidence interval for $p_0 - p_1$ is (0.00172067126,0.00807932874).

To test the hypothesis H_0 and H_1 we are using a significance level p of .05 or 5%, so we need to find that probability of having the mean that is different than is less that of p to prove the null hypothesis wrong. We assume that H_0 is true. So we need to find the $P(p_0 - p_1 | H_0)$. So we assume that the probability of $p_0 - p_1 = 0$ based on the null hypothesis. We can calculate the z-score by calculating the standard deviations of the true proportion. We let $\hat{p} = \frac{p_0 * 10,000 - p_1 * 10,000}{20,000} = 0.97295$. After calculating the standard deviation it is 0.0022942666584336.

We can calculate the z-score with $\frac{.0049 - 0}{0.0022942666584336} = 2.1357587105177$. Thus assuming that we had a $p_0 - p_1$ we found that the actual difference lies approximately 2.13576 standard deviations away.

Therefore we can reject the null hypothesis and conclude that classifiers are significantly different because the $P(z = 2.13576 | H_0) < .05$ or 5%, because it lies outside of the standard deviation of the 95% confidence interval.

6.2 Difference Rule

The difference rule that states if a classifier is significantly better if the accuracy of classifier A is greater than .2% than classifier B's. This can be best explained by how the standard deviations match up and when a classifier with an accuracy greater than that of .2% of another is compared, that when we assume the accuracies are equal we find that the probability of that is close to about the 5% mark and the z-score lies very near 1.96.

Like in the test above we showed that when the z-score is greater than 1.96 when calculating at a significance level of 5%, it is said that the classifiers are significantly different. This is a similar rule that is being applied when the difference rule makes an assumption about the .2% mark.

The difference rule seems to apply in my classifiers as well, they had a difference of around .49% and were shown to be significantly different.