KNN Handwritten Digits Classification Algorithm

Kathryn White

Implementation of the KNN Algorithm

In this dataset, I have used a K Nearest Neighbors algorithm in order to process images of handwritten digits and predict their true numerical value. This dataset has been established and imported into Python through SciKit Learn. The original dataset is an object that includes attributes of such as: description, data, images, target, and target names. The data attribute is an N-dimensional array that houses 1797 records each consisting of an 8x8 matrix. Each value in the matrix corresponds to a pixel – the number is the amount of black in a pixel. These matrices could be plotted using the imported library matplotlib in order to see what they actually look like.

In this instance, I transformed the data from a 3D array into a 2D array by transforming the 8x8 matrix into a 1x64 array. In a separate N-dimensional array, we have the attribute 'target' which houses a value from 1-9. This value corresponds to the actual number of the handwritten digit. In order to tie this value in with the values of the 'data' attribute, I created a new array that had each row of the data and included the target as the last column.

After the data set was loaded and transformed, I needed to shuffle the new array so that the information was assigned randomly. I did this simply with the *numpy.random* function.

Now that the dataset is shuffled, it is important to ‘train’ the dataset. This essentially means assigning sections of the data to be used for different purposes. For this algorithm, 70% of the data was allotted for training or teaching the algorithm, 15% was used for development of the algorithm, and 15% was used for testing. When we assign portions of the dataset, its important that we check for data leaks. Data leaks are rows or portions of the array that are duplicated between different sections of the dataset. I was able to check and verify that there were no data leaks by implementing a for loop that checks a given row against another in a separate portion of the dataset.

Each portion of the dataset then has two subsequent categories to consider: features and labels. In my dataset, the features are all the columns of the array except the last one, and the label is the last column. With this example, the features are the values of the image and the label is the actual number for that image. It’s important that we separate features and labels into their own variables for the training, development, and testing sets. This is done so that we can feed the feature information to the algorithm and it predict the label.

The most important part of this code is the KNN (K Nearest Neighbors) algorithm that is then implemented. KNN works by taking a given piece of information (in this code it is a row of data) as well as other information adjacent to it, and calculating which piece of information is closest to the original. From there it will make predictions on the given information based on its closet neighbor.

The KNN algorithm is able to gauge how close one piece of information is to another by calculating the distance between the two. There are many different types of distance calculations that can be done, but usually Euclidean is best for KNN due to its direct nature. I created a function to test the distance via Euclidean distance between to rows. From there the distance is fed to another function that uses the X and y coordinates from the training set and finds the K Nearest Neighbors and stores them in output arrays.

Now that the algorithm knows how to find the nearest neighbors from the dataset, we can feed it a row from an 'image' and let it predict what numerical value an image might have based off its neighbors. It was important to test the K value (the value that is used to determine how close the other neighbors are that we check) in order to determine the best value for the dataset. The accuracy rate was highest with a K value of 10 for the handwritten digit dataset. The development set is used here in order to test the best K value.

The last step of the algorithm is to test the entire model. This is done with the testing set. We compare the predictions made by the algorithm to the actual data. Accuracy is gauged by a range of 0.0 to 1.0, with 1.0 being the most accurate. The accuracy of this algorithm consistently rated 0.98 or higher which is considered quite accurate.

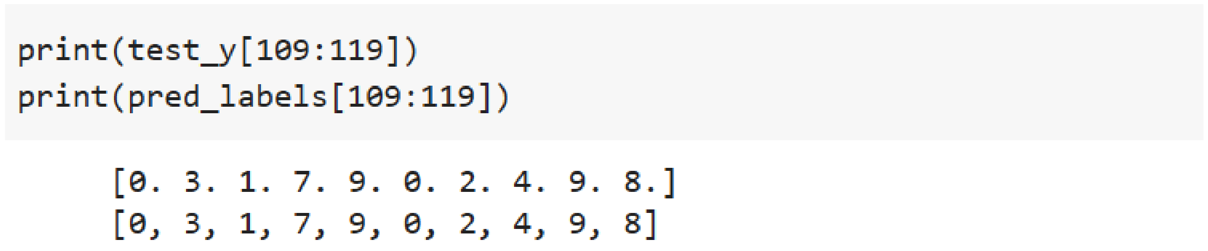
Performance of the Test Set

The best performance of this particular algorithm was an accuracy rating of 0.996. I believe this was achieved by the large dataset as well as the significant training sample that was passed to the algorithm in order to make predictions on the test set. The K value used to achieve this high rating was 10 and the distance calculation method was the Euclidean distance method.

Prediction Visualization

In order to visualize what the algorithm is doing, we can look at the actual labels (y) of each row/image at any point in the testing set, and compare them to the predicted values. The predicted values are those that the algorithm as stored as the numerical value they believe the dataset is based off the prediction function. If we look at just 10, it will appear as if there is a 1.00 or 100% accuracy rating.

Of course, if we looked at an even larger data set we might see a few machine errors. This ultimately explains why our accuracy was .996 and not a perfect 1.0. Below is an example of the actual versus predicted values.



```
print(test_y[109:119])
print(pred_labels[109:119])
```

```
[0. 3. 1. 7. 9. 0. 2. 4. 9. 8.]
[0, 3, 1, 7, 9, 0, 2, 4, 9, 8]
```

KNN Algorithm Source Code: (see attached)

```
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt
```

Load Data

```
# loading data
digitsData = load_digits()

X = digitsData.data
y = digitsData.target

# Converts y to 2d array to be appended to X
y = np.expand_dims(y, 1)

#X = digitsData.images.reshape((len(digitsData.images)), -1)
```

Shuffle Dataset

```
# Creates a new dataset with target appended at end for further classification
data = np.append(X, y, 1)

# Shuffles rows only
np.random.shuffle(data)
```

Train dataset

```
# Stores the total number of samples
total_sample = len(data)

# Sets our train/dev/test splits
train = data[:int(total_sample*0.70)]
dev = data[int(total_sample*0.70):int(total_sample*0.85)]
test = data[int(total_sample*0.85):]
```

Check Data Leaks

```
# Checks duplicate data between train/dev/test sets
def data_leaking_check(data1, data2):
```

✓ 14s completed at 5:55 PM

```
    for d2 in data2:
      if(np.array_equal(d1, d2)):
        data_leaking = True
        print("Find same sample: ")
        print(d1)
  if(not data_leaking):
    print("No Data Leaking!")

# Calls function to test data set:
# data_leaking_check(train,test)
```

Assigns Features and Labels

```
# Assigns specific colums to features and labels (last column of the dataset)
def get_features_and_labels(data):
  features = data[:, :-1]
  labels = data[:, -1]
  return features, labels

# Sets train/dev/test X and y to the features and labels of each set
train_X, train_y = get_features_and_labels(train)
dev_X, dev_y = get_features_and_labels(dev)
test_X, test_y = get_features_and_labels(test)
```

Euclidean Distance

```
# Calculates Euclidean distance between two rows given to the function
def euclidean_distance(row1, row2):
  distance = 0.0
  for i in range(len(row1)-1):
    distance += (row1[i] - row2[i]) ** 2
  return np.sqrt(distance)
```

KNearestNeighbors

```
# Finds the nearest neighbors in a given set to the specific data
def get_neighbors(train_X, train_y, test_row, num_neighbors):
  distances = []

  # Gets all distances
  for index in range(len(train_X)):
    train_row = train_X[index]
    train_label = train_y[index]
    dist = euclidean_distance(train_row, test_row)
    distances.append((train_row, train_label, dist))

  # Sorts the distance list
  distances.sort(key=lambda i: i[2])

  output_neighbors = [] # features
  output_labels = [] #labels
  output_distances = [] #distances

  # Gets the K Nearest Neighbor and returns it
  for index in range(num_neighbors):
    output_neighbors.append(distances[index][0])
    output_labels.append(distances[index][1])
    output_distances.append(distances[index][2])

  return output_neighbors, output_labels, output_distances
```

Predictions

```
# Predicts the classification (hand written digit) by receiving training data
def prediction(train_X, train_y, test_row, num_neighbors):
  output_neighbors, output_labels, output_distances = get_neighbors(train_X, train_y, test_
  label_cnt = np.bincount(output_labels)
  prediction = np.argmax(label_cnt)
  return prediction
```

K Value Testing

```
k_list = list(range(1, 100, 2))
performance = []

for k in k_list:
  pred_labels = []
  for dev_data in dev_X:
    pred = prediction(train_X, train_y, dev_data, k)
    pred_labels.append(pred)
  accuracy = accuracy_score(dev_y, pred_labels)
  performance.append(accuracy)

print(performance)
```

K Value Testing Plot

```
# Plots figure
plt.figure(figsize=(20, 6))
plt.plot(k_list, performance)
plt.plot(k_list, performance, "o")
plt.xlabel("K Values")
plt.ylabel("Accuracy")
plt.title("Performance on Dev Set")
```

Accuracy Rating

```
k = 10
pred_labels = []
for test_data in test_X:
  pred = prediction(train_X, train_y, test_data, k)
  pred_labels.append(pred)
accuracy = accuracy_score(test_y, pred_labels)
print(accuracy)
```

```
0.9962962962962963
```