KNN Implementation - Digits

Summary

In this k-NN implementation I have written a program that runs through a pre-defined list of distance metrics, calculating distances and returning k neighbors. The program then outputs some visual data, as well as the highest accuracy obtained by each of the distance metrics and their corresponding k values.

Using this information, I can quickly obtain the optimal distance metric to use as well as the optimal k value and then evaluate the test set with that particular model.

In this instance, I have three distance metrics: euclidean, manhattan, and chebyshev. I found the three to be roughly equivalent in accuracy, though Chebyshev did manage to ultimately win with an accuracy of almost 99% at k=18.

Implementation Overview

Imports

Begin with importing numpy and the relevant sklearn libraries. Numpy will be used for numerical operations, the other imports are the dataset, model_selection, and metrics, respectively. We will also be importing matplotlib to visualize these results.

```
# Imports
import numpy as np
from sklearn.datasets import load_digits
# from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
```

Load / Split Data

Here we want to load the data and target into X and y variables. We will then use np.random.shuffle(data) to randomize the rows prior to making a training split. We could also have used train_test_split since we imported it, but building the shuffler from scratch means we don't rely on libraries as much and we gain a deeper understanding of how things work.

```
# Load Data
digits = load_digits()
X = digits.data
y = digits.target

# Do this or you'll get ValueError: all input arrays must have same number of
dimensions...
y = np.expand_dims(y, 1)
data = np.append(X, y, 1)
```

Visualize Your Data

Print out some of the rows so you see what we are working with.

```
print(f"X is of type {type(X)}")
print(X[:10])
print()
print(f"y is of type {type(y)}")
print(y[:10])
print()
print("Shape:")
print(data.shape)
```

Shuffle the Data

Here, we want to shuffle the data *prior* to separating it into training/dev/test sets.

```
# Shuffle and Create Split Training/Dev/Test 70/15/15%
np.random.shuffle(data)

total_sample = len(data)
train = data[:int(total_sample*0.7)]
dev = data[int(total_sample*0.7):int(total_sample*0.85)]
test = data[int(total_sample*0.85):]
```

Check your work

Verify we only shuffled the rows. This can be done by storing some data in a temporary variable and shuffling it:

```
# See shuffling in action:
temp = data[95:99]
print("Initial Data:")
print(temp)
print()
print("Shuffled:")
np.random.shuffle(temp)
print(temp)
```

Check for Data Leaks

```
# Check for Data Leaks
def data_leaking_check(data1, data2):
  data_leaking = False
  for d1 in data1:
    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.")
data_leaking_check(train, dev)
```

Get Features

We need to grab our features and labels.

```
def get_features_and_labels(data):
  features = data[:, :-1]
  labels = data[:, -1]
  return features, labels

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)
```

Define Distances

Here we want to make some definitions. Given there are several different ways to calculate distances, we may want to play around with several different ones. To that end, I am implementing it with a way of choosing which calculation is used.

In the end, I used a for loop to iterate over a list of the metrics I implemented over the course of this project.

```
# Define Distance Metrics
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)

def manhattan_distance(row1, row2):
  return np.sum(np.abs(row1-row2))

def chebyshev_distance(row1, row2):
  return np.max(np.abs(row1 - row2))

def get_distance(distance_type, row1, row2):
  distance_functions = {
      'euclidean': euclidean_distance,
      'manhattan': manhattan_distance,
      'chebyshev': chebyshev_distance,
  }

  if distance_type in distance_functions:
    result = distance_functions[distance_type](row1, row2)
    return result
  else:
    raise ValueError(f"Unsupported distance type: {distance_type}.")
```

Get Neighbors

Here we are just getting *k* neighbors and sorting them by distance. We then return the neighbors, labels, and distances.

```
# Get Neighbors
def get_neighbors(distance_type, train_x, train_y, test_row, num_neighbors):
  distances = []
  for index in range(len(train_x)):
    train_row = train_x[index]
    train_label = train_y[index]
    distance = get_distance(distance_type, train_row, test_row)
    distances.append((train_row, train_label, distance))

  # Sort them by distance
  distances.sort(key=lambda i: i[2])

  # Get the k nearest neighbors
  output_neighbors = []
  output_labels = []
  output_distances = []
  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
```

Classification Prediction

Here we are going to call get_neighbors() to get the k-nearest neighbors for the test_row. We then use NumPy's bincount function to count the labels. We are then predicting the class by simply calling argmax to get the label that occurs most frequently among the k-nearest neighbors.

```
# Classification Prediction
def prediction_classify(distance_type, train_x, train_y, test_row, num_neighbors):
  output_neighbors, output_labels, output_distances = get_neighbors(distance_type,
train_x, train_y, test_row, num_neighbors)

  label_counts = np.bincount(output_labels)
  prediction = np.argmax(label_counts)
  return prediction
```

We also want to evaluate 10 random samples from each of the distance metrics we use:

```
def evaluate_random_samples(distance_type, train_x, train_y, test_x, test_y,
num_neighbors, num_samples=10):
    for _ in range(num_samples):

        # Randomly selects index from the test set
        random_index = np.random.choice(len(test_x))

        test_row = test_x[random_index]
        true_label = test_y[random_index]

        output_neighbors, output_labels, output_distances =
get_neighbors(distance_type, train_x, train_y, test_row, num_neighbors)
        label_counts = np.bincount(output_labels)
        predicted_label = np.argmax(label_counts)

        print(f"Sample: True Label = {true_label}, Predicted Label =
{predicted_label}")
```

Now we will synthesize all of the code into a single, simple function that will perform all of the functions we have defined so far, as well as grab a random sampling of size=10 from each distance metric so we can visualize some of the predictions.

After each model runs the highest accuracy and the corresponding k for that model are printed for comparison.

```
# Get predictions and visualize results
def predict_and_visualize(distance_type, num_of_k):
  print(f"Making Predictions with {distance_type} distance.")
  data = load_data()
  #visualize_data()
  #visualize_shuffle()
  total_sample, train, dev, test = shuffle_and_split(data)
  data_leaking_check(train, dev)

  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)
```

```
  k_list = list(range(3, num_of_k, 3))
  performances = []

  print(f"10 Randomly Selected Test Data, k=15, metric={distance_type}.")
  evaluate_random_samples(distance_type, train_x, train_y, test_x, test_y, 15, 10)
  print()

  for k in k_list:
    predicted_labels = []
    for dev_data in dev_x:
      prediction = prediction_classify(distance_type, train_x, train_y, dev_data,
k)
      predicted_labels.append(prediction)
    accuracy = accuracy_score(dev_y, predicted_labels)
    performances.append(accuracy)

  # Find the best model and k value
  max_accuracy_index = np.argmax(performances)
  best_k = k_list[max_accuracy_index]
  best_accuracy = performances[max_accuracy_index]

  print(f"Best performance: distance metric: {distance_type}, k={best_k}, accuracy=
{best_accuracy}")
  print()

  plt.figure(figsize=(20,6))
  plt.plot(k_list, performances)
  plt.plot(k_list, performances, 'o')
  plt.xticks(k_list)
  plt.xlabel("k values")
  plt.ylabel("accuracy")
  plt.title(f"{distance_type} Performance on Dev Set")
```

Running the Program

Here we are pulling everything together and having the program run through all three of the distance measurements we used (Euclidean, Manhattan, Chebyshev).

```
# Set number of k values to test prior to running the program
# The program will start at 3 and increment by 3 each time
```

```
num_of_k = 99

distance_types = [
    'euclidean',
    'manhattan',
    'chebyshev',
]

for metric in distance_types:
  predict_and_visualize(metric, num_of_k)
```

Results

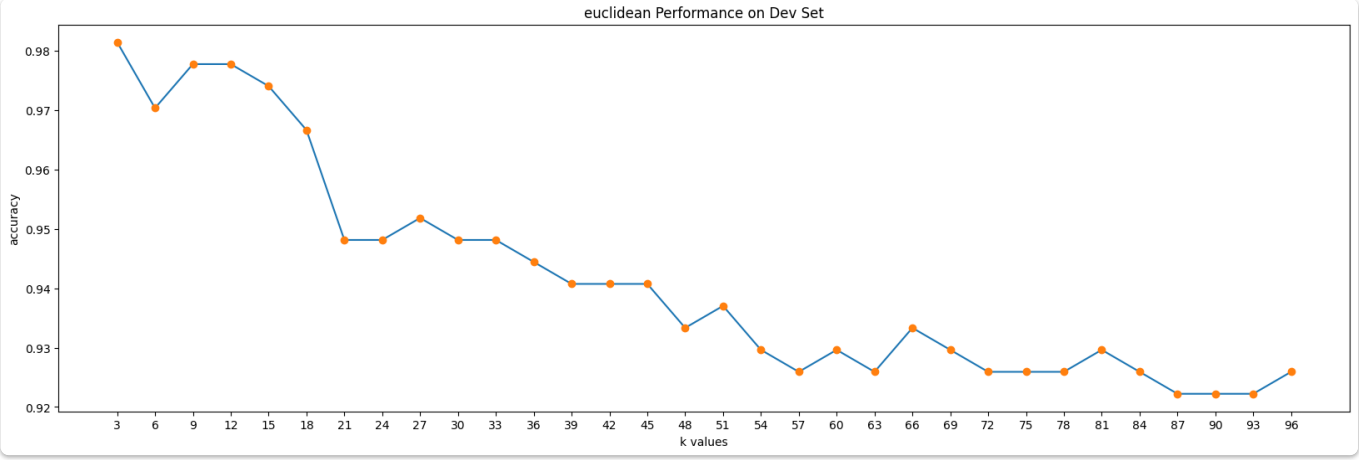
First, we get the output of the random sampling. Here I have k=15, which is arbitrarily chosen for this demonstration. We can easily tweak the function to take the highest k from the testing process and then use that k to get our randomly sampled predictions. Given the high accuracy k=15 was seeing, I opted for quickly iterating on new ideas rather than exploring quality of life improvements.

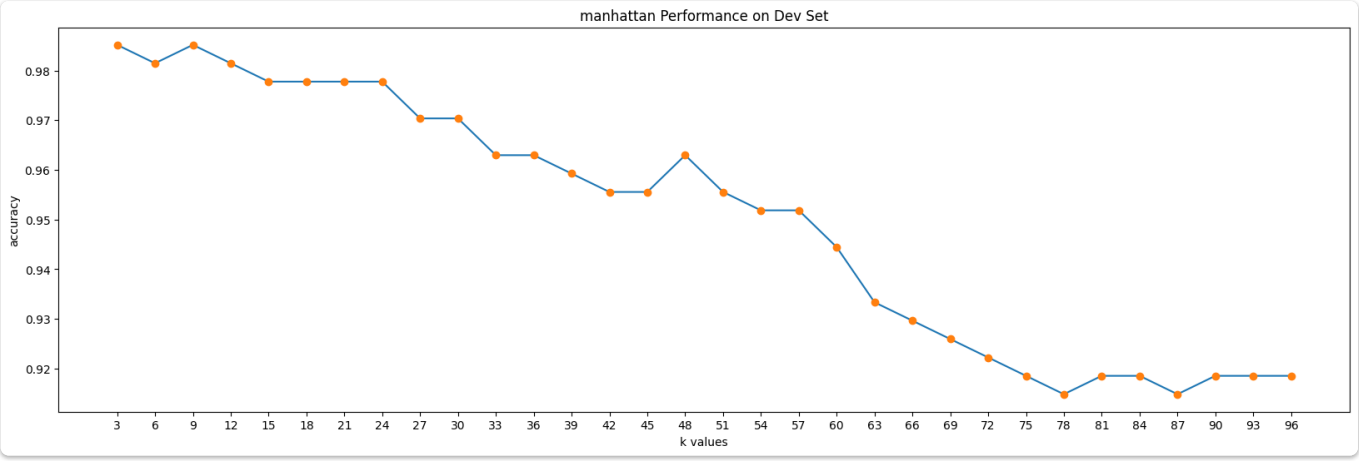
This visualization could be improved. For example, we might only choose sample data where the prediction was wrong. Another option is to show the actual drawn digit, to better elucidate some of the errors in a more visual manner.

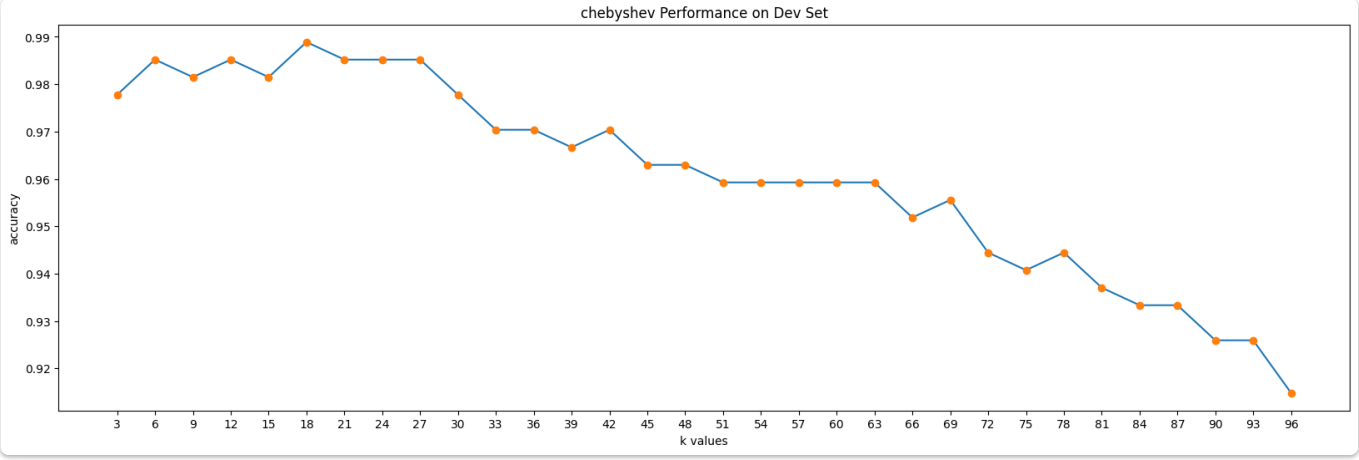
```
Making Predictions with euclidean distance.
No Data Leaking.
10 Randomly Selected Test Data, k=15, metric=euclidean.
Sample: True Label = 2.0, Predicted Label = 2
Sample: True Label = 5.0, Predicted Label = 5
Sample: True Label = 9.0, Predicted Label = 9
Sample: True Label = 5.0, Predicted Label = 5
Sample: True Label = 9.0, Predicted Label = 9
Sample: True Label = 0.0, Predicted Label = 0
Sample: True Label = 0.0, Predicted Label = 0
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 8.0, Predicted Label = 8
Sample: True Label = 9.0, Predicted Label = 1

Best performance: distance metric: euclidean, k=3, accuracy=0.9814814814814815

Plotting euclidean accuracy report.
```

```
Making Predictions with manhattan distance.
No Data Leaking.
10 Randomly Selected Test Data, k=15, metric=manhattan.
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 5.0, Predicted Label = 5
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 5.0, Predicted Label = 5
Sample: True Label = 7.0, Predicted Label = 7
Sample: True Label = 2.0, Predicted Label = 2
Sample: True Label = 8.0, Predicted Label = 8
Sample: True Label = 4.0, Predicted Label = 4
Sample: True Label = 3.0, Predicted Label = 3

Best performance: distance metric: manhattan, k=3, accuracy=0.9851851851851852

Plotting manhattan accuracy report.

Making Predictions with chebyshev distance.
No Data Leaking.
10 Randomly Selected Test Data, k=15, metric=chebyshev.
Sample: True Label = 0.0, Predicted Label = 0
Sample: True Label = 3.0, Predicted Label = 5
Sample: True Label = 4.0, Predicted Label = 4
Sample: True Label = 0.0, Predicted Label = 0
Sample: True Label = 7.0, Predicted Label = 7
Sample: True Label = 4.0, Predicted Label = 4
Sample: True Label = 8.0, Predicted Label = 8
Sample: True Label = 8.0, Predicted Label = 8
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 6.0, Predicted Label = 6

Best performance: distance metric: chebyshev, k=18, accuracy=0.9888888888888889

Plotting chebyshev accuracy report.
```

Chebyshev wins, with an accuracy of almost 99% with k=18.
Next, we have the graphs:







Surprisingly, all of the models performed well with accuracies >98%. You can also clearly see how the accuracy decreases as the k increases, as a result of over fitting the data.

Running the Test Data

Here we simply altar some code from above to run the test data instead of dev data. We will use the Chebyshev metric with a k=18 as that was the best model, as seen above.

```
# Run the Test Dataset with results from above
# TODO: automate this step
```

```
distance_type = 'chebyshev'
k = 18

def final_test(distance_type, k):
  print(f"Making Predictions with {distance_type}, k={k}.")

  # Load Data
  data = load_data()
  total_sample, train, dev, test = shuffle_and_split(data)
  data_leaking_check(train, dev)




  # Get Labels
  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)

  performances = []

  print(f"10 Randomly Selected Test Data, k=15{k}, metric={distance_type}.")
  evaluate_random_samples(distance_type, train_x, train_y, test_x, test_y, k, 10)
  print()

  predicted_labels = []
  for test_data in test_x:
    prediction = prediction_classify(distance_type, train_x, train_y, test_data, k)
    predicted_labels.append(prediction)
  accuracy = accuracy_score(test_y, predicted_labels)

  print(f"{distance_type}'s accuracy at k={k} is {accuracy}")

final_test(distance_type, k)
```

The output includes our 10 randomly selected data for visualizing and a print out of the model's accuracy:

```
Making Predictions with chebyshev, k=18.
No Data Leaking.
10 Randomly Selected Test Data, k=1518, metric=chebyshev.
```

```
Sample: True Label = 3.0, Predicted Label = 3
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 7.0, Predicted Label = 7
Sample: True Label = 9.0, Predicted Label = 9
Sample: True Label = 6.0, Predicted Label = 6
Sample: True Label = 5.0, Predicted Label = 5
Sample: True Label = 4.0, Predicted Label = 4
Sample: True Label = 1.0, Predicted Label = 1
Sample: True Label = 3.0, Predicted Label = 3
Sample: True Label = 5.0, Predicted Label = 5

chebyshev's accuracy at k=18 is 0.9814814814814815
```