Michael Dubravski
Professor Skalak
COMP 364
2 December 2020

k-Nearest Neighbors

Introduction

For my final project I have implemented k-Nearest Neighbors algorithm in Python using the Iris Flower Species Dataset provided by Kaggle. For this implementation of kNN I will be measuring distance of elements in the dataset by implementing a Euclidean Distance function. After the nearest neighbors are found a classification prediction can be made. The Iris Flower Dataset consists of elements that describe different properties about various iris flowers species including sepal length, sepal width, petal length, petal width, and their associated species classification. The algorithm is then evaluated using k-folds cross-validation with 5 folds to achieve a mean classification accuracy of approximately 96.667%. The main goal is that by using the kNN algorithm and the Iris Dataset one could predict the species classification of a flower given values for the other elements of the dataset.

Methods

Dataset

The Iris Flowers Dataset consists of a total of 150 instances. This breaks down further as 3 classes with 50 instances each. Each class corresponds to a species of iris flower (iris-setosa, iris-versicolor, iris-virginica). Each instance consists of 4 numerical predictive attributes and the class/species attribute. Below Figure 1 shows further information about the attributes and Figure 2 shows some examples of instances in the dataset.

Attribute Information	
Sepal length in cm	Numerical Value
Sepal width in cm	Numerical Value
Petal length in cm	Numerical Value
Petal width in cm	Numerical Value
Class/Species	Iris Setosa, Iris Versicolor, Iris Virginica

Figure 1. Information about Attributes

Instance Examples
`4.8,3.4,1.6,0.2,Iris-setosa`
`7.0,3.2,4.7,1.4,Iris-versicolor`
`6.4,3.1,5.5,1.8,Iris-virginica`

Figure 2. Three examples of instances in the dataset. First is an instance with class Iris Setosa, second is an instance with class Iris Versicolor, third is an instance with class Iris Virginica

kNN

This implementation of the k-Nearest Neighbors algorithm can be broken down into three parts, calculating the Euclidean Distance, finding the near neighbors of an element in the dataset, then making a prediction based off the closest neighbors. For calculating the Euclidean Distance, we find the straight-line distance between two vectors (rows) in the dataset, this is calculated as the square root of the sum of the squared differences between two vectors.
More specifically let $p = first\ row$ and $q = second\ row$, then Euclidean Distance function d can be defined as,

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

This was then implemented in Python as,

```
1. # Euclidean Distance function
2. def euclideanDistance(self, row1, row2):
3.         distance = 0.0
4.         for i in range(len(row1)-1):
5.             distance = distance + (row1[i] - row2[i])**2
6.         return sqrt(distance)
```

The next step for implementing the kNN algorithm is the write a function that gets the k nearest neighbors to a new instance of data. To do this we utilize the Euclidean Distance function to calculate the distance from each row in the dataset to the new instance of data. Then the records are sorted by their distance to the new instance and the closest k instances are returned. This is implemented in Python by making a list of distances formatted as tuples where the first element is instance and the second element is the calculated distance, then sorting the list of tuples by the second element (distance) and then returning the k closest instances (first element).

```
1. # Locate the nearest neighbors
2. def getNeighbors(self, train, testRow, numNeighbors):
3.     distances = list()
4.     # get Euc distance between each testRow and trainRow
5.     for trainRow in train:
6.         dist = self.euclideanDistance(testRow, trainRow)
7.         distances.append((trainRow,dist))
8.     # Sort using the second element of the tuple
9.     distances.sort(key=lambda x: x[1])
10.      # Build list of most similar neighbors to testRow
11.      neighbors = list()
12.      for i in range(numNeighbors):
13.          neighbors.append(distances[i][0])
14.      return neighbors
```

The final step for implementing the kNN algorithm is the write a function that can be used to make predictions based off of the k nearest neighbors. In this case we are looking to classify what species of flower a given instance is so the function should return the most represented class/species that are among the k closet neighbors. This is implemented in Python by utilizing the previously defined $getNeighbors()$ function to get the k closest neighbors, then we make a list of all the classes/species from the neighbors and use the $max()$ to find the most represented class/species among them.

```
# Make a classification prediction with the nearest neighbors
def predictClassification(self, train, testRow, numNeighbors):
    # get the nearest neighbors
    neighbors = self.getNeighbors(train, testRow, numNeighbors)
    # get the class of each instance in neighbors
    values = list()
    for row in neighbors:
        values.append(row[-1])
    # return the most represented class/species
    prediction = max(set(values), key=values.count)
    return prediction
```

Loading iris.csv and Formatting Data

Next, we must load the dataset and convert and format the loaded-in data. To do this we create helper methods of *loadCSV*() to load in the iris.csv dataset as a list of lists of instances (represented as strings), *srtingColToFloat*() to convert instance values from strings to floats, and *stringColToInt*() to convert the last column of class values from strings to integers. These functions were adapted from "*How to Load Machine Learning Data From Scratch In Python*",

```
# Load a CSV file ignoring empty rows, returns a list of lists of str
def loadCSV(self, filename):
    dataset = list()
    with open(filename, 'r') as file:
        cReader = reader(file)
        for row in cReader:
            if not row:
                continue
          dataset.append(row)
   return dataset

# Convert String column to float
def stringColToFloat(self, dataset, col):
    for row in dataset:
        # convert string to float value
        # use .strip() to remove preceding or trailing whitespace
        row[col] = float(row[col].strip())
# Convert String class value column to int
def stringColToInt(self, dataset, col):
    classes = list()
    for row in dataset:
        classes.append(row[col])
    # get unique class values
    unique = set(classes)
    lookup = dict()
    # assign integer to each class value
    for i, value in enumerate(unique):
        lookup[value] = i
        # print the mapping of class names to their associated integer
        print("[%s] -> %d" % (value, i))
        # replace class values with their associated integer values
        for row in dataset:
            row[col] = lookup[row[col]]
        return lookup
```

Algorithm Analysis and Evaluation

In order to evaluate and analyze the kNN algorithm we will implement k-fold cross-validation using 5 folds along with some other helper functions. As there are 150 total instances there will be 30 instances in each fold. To implement this in Python we will define four helper functions of; $crossValidationSplit()$ to split the dataset into k folds, $evaluateAlgorithm()$ to evaluate a given algorithm using a cross-validation split, $kNearestNeighbors()$ to manage the application of the kNN implementation by learning the statistics from a given training dataset and using them to making predictions for a test dataset, and $accuracyMetric()$ to calculate the accuracy percentage of each prediction. These functions were adapted from "*How to Implement Resampling Methods From Scratch In Python*",

```
1. # Split dataset into k folds
2. def crossValidationSplit(self, dataset, n):
3.     datasetSplit = list()
4.     datasetCopy = list(dataset)
5.     foldSize = int(len(dataset) / n)
6.     for _ in range(n):
7.         fold = list()
8.         while len(fold) < foldSize:
9.             i = randrange(len(datasetCopy))
10.              fold.append(datasetCopy.pop(i))
11.          datasetSplit.append(fold)
12.     return datasetSplit
13.
14. # Find accuracy percentage
15. def accuracyMetric(self, actual, predicted):
16.     numCorrect = 0
17.     for i in range(len(actual)):
18.         if actual[i] == predicted[i]:
19.             numCorrect += 1
20.     return numCorrect / float(len(actual)) * 100.0
21.
22. # Evaluate an algorithm using a cross validation split
23. def evaluateAlgorithm(self, dataset, algo, nFolds, *args):
24.     folds = self.crossValidationSplit(dataset, nFolds)
25.     scores = list()
26.     for fold in folds:
27.         trainSet = list(folds)
28.         trainSet.remove(fold)
29.         trainSet = sum(trainSet, [])
30.         testSet = list()
31.         for row in fold:
32.             rowCopy = list(row)
33.             testSet.append(rowCopy)
34.             rowCopy[-1] = None
35.         predicted = algo(trainSet, testSet, *args)
36.         actual = [row[-1] for row in fold]
37.         accuracy = self.accuracyMetric(actual, predicted)
38.         scores.append(accuracy)
39.     return scores
40.
41. # kNN Algorithm for use in evaluateAlgorithm()
42. def kNearestNeighbors(self, train, test, numNeighbors):
43.     predictions = list()
```

```
44.      for row in test:
45.           value = self.predictClassification(train,row,numNeighbors)
46.          predictions.append(value)
47.      return predictions
```

Results

Results on Toy Datasets

To evaluate and test the implementation of the kNN algorithm and its various methods, a number of experiments have been made. To start, I tested the three helper functions of the kNN algorithm on 2 toy datasets that consists of 10 instances each with 3 attributes (2 numerical attributes and 1 class attribute).

```
1. # Example dataset for testing
2. dataset = [[ 0.54857876,  5.25697175,0],
3.           [ 7.81764176,  9.17441769,1],
4.           [ 3.74230831,  2.06165119,0],
5.           [ 8.39840247,  9.46184753,1],
6.           [ 4.16672282,  4.14349935,1],
7.           [ 5.16342338,  3.32907878,1],
8.           [ 0.27843358,  7.39775071,0],
9.           [ 1.8701541 ,  4.98218665,0],
10.         [2.23288871,  7.38405505,1],
11.         [ 8.94828588,  7.82797826,1]]
```

To test the *euclideanDistance*() function I ran a for loop that prints the distance between the first row in the dataset and all other rows starting with the first row. This should result in the first line of output being 0 as the distance from the first row and itself should be 0.

```
1. # Experiment for testing euclideanDistance()
2. print("Testing euclideanDistance(): ")
3. row0 = dataset[0]
4. for row in dataset:
5.     distance = knn.euclideanDistance(row0,row)
6.     print(distance)
7. print("\n")
```

```
1. OUTPUT:
2.     Testing euclideanDistance():
3.     0.0
4.     8.257460844036228
5.     4.517740798197913
6.     8.905094755436844
7.     3.7856026236882085
8.     5.001356033169241
9.     2.157756467689511
10.    1.3498400758897793
11.    2.7131869402619295
12.    8.784369879293624
```

To test $getNeighbors()$ I assigned the output of $getNeighbors()$ to a list then iterated through the list printing the three most similar elements in the dataset to the first element in the dataset in order of most similar. The first element of the dataset should be printed first as it is most similar to itself.

```
1. # Experiment for testing getNeighbors()
2. print("Testing getNeighbors(): ")
3. neighbors = knn.getNeighbors(dataset, dataset[0], 3)
4. for n in neighbors:
5.     print(n)
6. print("\n")
```

```
1.  OUTPUT:
2.     Testing getNeighbors():
3.     [0.54857876, 5.25697175, 0]
4.     [1.8701541, 4.98218665, 0]
5.     [0.27843358, 7.39775071, 0]
```

To test $predictClassification()$ I ran $predictClassification()$, which in this case makes it prediction based upon the 3 most similar instances in the dataset, to predict the classification for the first row in the dataset and compared its output prediction classification to the actual classification. In this case the actual classification is 0.

```
1. # Experiment for testing predictClassification()
2. print("Testing predictClassification(): ")
3. prediction = knn.predictClassification(dataset, dataset[0], 3)
4. print("Expected Classification: %d \nActual Classification: %d \n" %
   (dataset[0][-1], prediction))
```

```
1. OUTPUT:
2.     Testing predictClassification():
3.     Expected Classification: 0
4.     Actual Classification: 0
```

Results on Iris Dataset

First, I ran an experiment to evaluate the kNN implementation using the k-fold cross-validation methods implemented earlier. This experiment used 5 folds and outputs the mean classification accuracy scores for each cross-validation fold and the total mean accuracy score. This results in a mean accuracy of 96.667%.

```
1. print("Testing Mean Classification Accuracy Scores and Mean Accuracy
   Score(): ")
2. scores = knn.evaluateAlgorithm(dataset, knn.kNearestNeighbors, nFolds,
   numNeighbors)
3. print("Scores: %s" % scores)
4. print("Mean Accuracy: %.3f%%\n" % (sum(scores)/float(len(scores))))
```

```
1. OUTPUT:
2.     Testing Mean Classification Accuracy Scores/Mean Accuracy Score():
```

```
3.      Scores: [96.66666666666667, 96.66666666666667, 100.0, 90.0, 100.0]
4.      Mean Accuracy: 96.667%
```

Finally, I developed an experiment for making predictions with the implemented kNN algorithm. Three new rows are defined with the goal of predicting their classification/species. These instances' values differ by a factor of 0.1 from actual values in the dataset meaning that their classification/species is the same as the instances they referenced. Then I ran the $predictClassification()$ method three times once for each new instance and printed their predicted classification. Note that the predicted classifications are integers and not strings thus a key is also printed that must be used to verify the classifications.

```
1. # Define new instances
2. irisSetosaRow = [4.8,3.0,1.4,0.2]
3. irisVersicolorRow = [5.5,2.6,4.1,1.2]
4. irisVirginicaRow = [7.6,3.9,6.8,2.3]
5.
6. # Make class predictions
7. predictedClass =  knn.predictClassification(dataset, irisSetosaRow,
   numNeighbors)
8. print("Data: %s \nPredicted: %s\n" % (irisSetosaRow, predictedClass))
9.
10.  predictedClass =  knn.predictClassification(dataset,
   irisVersicolorRow, numNeighbors)
11.  print("Data: %s \nPredicted: %s\n" % (irisVersicolorRow,
   predictedClass))
12.
13.  predictedClass =  knn.predictClassification(dataset, irisVirginicaRow,
   numNeighbors)
14.
15.  print("Data: %s \nPredicted: %s\n" % (irisVirginicaRow,
   predictedClass))
```

```
1. OUTPUT:
2.      Testing kNN Algorithm on the Iris Flowers dataset:
3.      [Iris-virginica] -> 0
4.      [Iris-setosa] -> 1
5.      [Iris-versicolor] -> 2
6.
7.      Making a prediction using kNN:
8.      Data: [4.8, 3.0, 1.4, 0.2]
9.      Predicted: 1
10.
11.      Data: [5.5, 2.6, 4.1, 1.2]
12.      Predicted: 2
13.
14.      Data: [7.6, 3.9, 6.8, 2.3]
15.      Predicted: 0
```

Conclusion

In closing, I found that the kNN algorithm is very powerful for how simple its implementation can be. In this case it classifies the species for new instance of the Iris Flower Dataset with incredible accuracy. I found that adjusting the k number of neighbors some interesting performance changes happen. Some further work that can be done would be to try this algorithm out on other classification datasets. It would also be interesting to see how it performs on much larger datasets as well, I expect that it wouldn't perform as well.

Works Cited

Brownlee, Jason. *Develop k-Nearest Neighbors in Python From Scratch*. 23 Feb. 2020, machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/.

Brownlee, Jason. "How to Implement Resampling Methods From Scratch In Python." *Machine Learning Mastery*, 13 Aug. 2019, machinelearningmastery.com/implement-resampling-methods-scratch-python/.

Brownlee, Jason. "How to Load Machine Learning Data From Scratch In Python." *Machine Learning Mastery*, 11 Dec. 2019, machinelearningmastery.com/load-machine-learning-data-scratch-python/.

Brownlee, Jason. "A Gentle Introduction to k-Fold Cross-Validation." *Machine Learning Mastery*, 2 Aug. 2020, machinelearningmastery.com/k-fold-cross-validation/.

Fisher, Ronald. "Iris Flower Dataset." *Kaggle*, 22 Mar. 2018, www.kaggle.com/arshid/iris-flower-dataset.