k Nearest Neighbors Python Implementation

Damon Nutter
Cal State San Bernardino
CSE 5160 Machine Learning
nutter.damon.a@gmail.com

Abstract

This report is on a k Nearest Neighbors Python implementation without the use of any external python libraries for CSE 5160 Machine Learning. The data set used for this machine learning program was the seeds Data Set from the UCI Machine Learning repository. Many different test splits and k values were chosen for this implementation in order to show the variance in accuracy and to come up with an average overall testing error. Overall the testing accuracy of the program was 89% and the total tests ran were 308.

1 Implementation Process

The implementation process consisted of a function by function implementation of the overall system. It was initially expected that the overall system would contain a four step process: read the input file, calculate the euclidean distance, get the nearest neighbors, and make the classification prediction.

This 4 step process was the goal that was set initially, but it was soon realized that the process would be a 6 step process.

The first function that was written was the readData function which simply read the seeds data and wrote it into a dictionary with the key as the classifier value. The second function that was implemented was the normalizeData function which served as a min-max normalization of the data. The data needed to be normalized because there were 7 different attribute values and in order for the data to all be weighted the same, the normalization was necessary. Next, the data was split into a training set and a test set. Then the classifier function was created in order to calculate the euclidean distances of the normalized data and classify each test instance. Lastly, an extra function was added called runClassifier in order to test the accuracy of the program. The main function was created in order to contain a series of runs of different test splits and different k values to test all possible outcomes of the system. Each test split (from 3% to 14%) and k value (from 3 to 25) was tested and the total of classification hits and misses were tallied in order to access the overall accuracy.

2 Code

2.1 readData Function

```
#Read data into a dictionary with number of keys = number of different
#classifications
def readData(filename):
    dataSet ={1:[],2:[],3:[]}
    with open(filename, 'r') as reader:
        line = reader.readline() #read each line of data
        while line != '':
            attributes = line.split() #split each attribute
            for i in range(0, len(attributes)):
                attributes[i] = float(attributes[i])
            if attributes[len(attributes)-1] == 1:
                dataSet[1].append(attributes)
            elif attributes[len(attributes)-1] == 2:
                dataSet[2].append(attributes)
            elif attributes[len(attributes)-1] == 3:
                dataSet[3].append(attributes)
            else:
                print("error, classification value should be 1, 2, or 3")
            line = reader.readline()
    return dataSet
```

2.2 normalizeData Function

```
#function to min-max normalize the dataset
def normalizeData(dataSet):
    #find max and minimum values
    maxVals = [0,0,0,0,0,0,0]
    minVals = [0,0,0,0,0,0,0]
    for classification in dataSet:
        for attributes in dataSet[classification]:
            i = 0
            while i < len(attributes) - 1:
                if maxVals[i] < attributes[i]:
                    maxVals[i] = attributes[i]
                if minVals[i] > attributes[i]:
                    minVals[i] = attributes[i]
                i += 1
    #normalize the values
    for classification in dataSet:
        for attributes in dataSet[classification]:
            i = 0
            while i < len(attributes) - 1:
                attributes[i] = (attributes[i] - minVals[i])/(maxVals[i] -
                    minVals[i])
                i += 1
    return dataSet
```

2.3 splitSet Function

```
#Splits the data into a training set and a test set. Test set has 1/m % of the data.
#training set has the remaining 1-1/m % of the data.
def splitSet(dataSet, m):
    count = 0
    trainingSet = {1:[],2:[],3:[]}
    testSet = {1:[],2:[],3:[]}
    percent = round(1/m,2)* 100
    for classification in dataSet:
        for attributes in dataSet[classification]:
            count += 1
            if count%m != 0:
                trainingSet[classification].append(attributes)
            else:
                testSet[classification].append(attributes)
    return [trainingSet, testSet];
```

2.4 classifier Function

```
#Function that finds the classification of a test instance.
#k is the number of nearest neighbors to check.
def classifier(trainingSet, test, k):
    distance = []
    for classification in trainingSet:
        for attributes in trainingSet[classification]:
            #calc euclidean distance of 7 attributes.
            i = 0
            values = []
            while i < len(attributes) - 1:
                values.append((attributes[i] - test[i])**2)
                i += 1
            e_distance = math.sqrt(sum(values))
            distance.append((e_distance,classification))
    distance = sorted(distance)[:k] #sort distance and select the first k distances

    count1 = 0; #how many match classification 1
    count2 = 0; #how many match classification 2
    count3 = 0; #how many match classification 3
    #loop over i = k distances to see which classification fits.
    for i in distance:
        if i[1] == 1:
            count1 += 1
        elif i[1] == 2:
            count2 += 1
        elif i[1] == 3:
            count3 += 1
    if count1>count2 and count1>count3:
        return 1
    elif count2>count1 and count2>count3:
        return 2
    else:
        return 3
```

2.5 runClassifier Function

```
#Function that runs the test set against the training set.
#function also returns the classification error
def runClassifier(trainingSet, testSet, k):
    classifications=[]
    correct = 0
    wrong = 0
    for classification in testSet:
        for attributes in testSet[classification]:
            cl = classifier(trainingSet,attributes,k)
            classifications.append(cl)
            if cl == attributes[len(attributes)-1]:
                correct += 1
            else:
                wrong += 1
    total = correct + wrong
    percentCorrect = correct/total
    return percentCorrect
```

2.6 main Function

```
def main():
    file = "seeds_dataset.txt" #data file
    dataSet = readData(file) #data set from data file
    normData = normalizeData(dataSet) #normalize the dataset
    #Run the data starting with m = 7 and k = 3
    avgs = []
    percentCorrect = []
    m = 7
    total = 0
    while m < 35:
        percentCorrect.clear()
        k = 3
        while k < 25:
            split = splitSet(normData, m)
            #split the dataset into training and test set using
            #m = 1/m % of the data.
            correct = runClassifier(split[0], split[1], k)
            #run classifier on 2 data sets.
            percentCorrect.append(correct)
            k += 2
            total += 1
        if(len(percentCorrect)!= 0):
            avg = sum(percentCorrect)/len(percentCorrect)
            avgs.append(avg)
        else:
            print("div by zero")
        m += 1
    print("For k from 3 to 25.")
    m = 7
    i = 0
    while i < len(avgs):
        print("m = " + str(m) + ", average correct classification was: " +
             str(avgs[i]))
        m += 1
        k += 2
        i += 1
    print("The average accuracy for all testing sizes and k values: " +
        str(sum(avgs)/len(avgs)))
    print("Total tests ran: " + str(total))
```

3 Output and Results

```
For k from 3 to 25. Where test set size = 1/m of the total data.
m = 7, average correct classification was: 0.9212121212121214
m = 8, average correct classification was: 0.8531468531468532
m = 9, average correct classification was: 0.8695652173913043
m = 10, average correct classification was: 0.7619047619047618
m = 11, average correct classification was: 0.8995215311004785
m = 12, average correct classification was: 0.946524064171123
m = 13, average correct classification was: 0.9431818181818182
m = 14, average correct classification was: 0.9333333333333335
m = 15, average correct classification was: 0.9350649350649352
m = 16, average correct classification was: 0.9370629370629371
m = 17, average correct classification was: 0.8333333333333333
m = 18, average correct classification was: 0.9173553719008264
m = 19, average correct classification was: 0.909090909090909
m = 20, average correct classification was: 0.5999999999999999
m = 21, average correct classification was: 0.9090909090909091
m = 22, average correct classification was: 0.8787878787878789
m = 23, average correct classification was: 0.8787878787878789
m = 24, average correct classification was: 1.0
m = 25, average correct classification was: 0.75
m = 26, average correct classification was: 0.8863636363636364
m = 27, average correct classification was: 0.987012987012987
```

```
m = 28, average correct classification was: 1.0
m = 29, average correct classification was: 1.0
m = 30, average correct classification was: 0.8701298701298701
m = 31, average correct classification was: 0.9242424242424243
m = 32, average correct classification was: 0.8787878787878789
m = 33, average correct classification was: 0.9848484848484849
m = 34, average correct classification was: 0.8333333333333333
The average accuracy for all testing sizes and k values: 0.894345802438572
Total tests ran: 308
```

4 Strategy

The training set vs test set split function played a pivotal role in creating reliable sized sets for the k-NN classifier to run. The different training sizes varied widely so that there would be a larger test accuracy range and it would be easier to tell which test splits yielded better results. For the k-NN classifier, there was first a split of the set, and then a euclidean distance for a test set value was calculated against each training set value in order to find the k nearest neighbors to the test instance. During these calculations, hits and misses are recorded within the runClassifier function in order to calculate accuracy.

5 Conclusion

Overall, the python implementation of k-nearest neighbors algorithm yielded acceptable results for the seeds data. I am pleased overall with the performance and I think that perhaps on a larger data set, the program could reach higher accuracy numbers. Furthermore, the program is written such that it can import any data set and run the classifier. Additional test sets should be ran in the future in order to further test the accuracy of the program.