COMP41450 Machine Learning

Assignment: Text Classification

Name: Aman Dhingra

Student ID: 16200307

Flow of The Report:

1. Introduction of kNN and its Properties
2. Explanation of my Implementation of kNN from Scratch in Python
3. Going through the Weighted and Unweighted kNN Classifiers
4. Reporting of Average Accuracy Scores obtained Over 5 iterations of kNN from k= [1..10] and Observations for the given Dataset.

Introduction of kNN and its Properties:

The k-Nearest Neighbour algorithm is one of the Classification Algorithms which is most commonly used for classification.

The Dataset provided is basically a .matrix file containing 1-1839 Documents containing Words and their frequency in that particular Document. The second file is a class label file with Document numbers and their Correct Labels ['Business', 'Politics', 'Sport', 'Technology'].

The kNN is one of the Supervised Machine Learning Techniques which follows the Lazy Learning Approach.

- This means the Training Set is well labelled to one Class and the Goal is to Approximate a mapping Function so that when we have new unseen data (X), the Function can predict class (Y) for the data.
- The Lazy Learning approach means that there is very little work done offline, the System trains using each Query from the Test Set. The Function is developed for each Query separately and the Query is classified based on some function (here, Similarity) based on the Training Set.

The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples.

The Classification phase would include taking the value of k from the user so as to consider k nearest instances from the Query instance based on a similarity measure and predict class of the Query instance based on a Voting system or a Weight Based consideration.

Explanation of my Implementation of kNN from Scratch in Python

The Source Code zip and README file have been attached along in the submission. The README has simple instructions to run the File.

I have used 5 Total Functions in the Code. Statistics about the Code are given below. These Statistics have been obtained from http://www.lizard.ws/ .

Function Name	NLOC	Complexity	Token #	Parameter #
loadDataFile	25	8	264	2
cosineSim	17	6	199	5
weightedClass	13	4	138	4
unweightedClass	10	3	106	4
main	11	2	131	0

main()

The *main()* function is the one which calls all other functions directly or indirectly and computes Accuracy by taking a user specified k value.

```
def main():
    matrix_file='../Data/news_articles.mtx'
    labelFile='../Data/news_articles.labels'
    k=int(input("Please Enter k Value: "))
    print("Loading Data Files...")
    myTrainSet,myTestSet,labels=loadDataFile(matrix_file,labelFile)
    print("Data Files Loaded.")
    print("Calculating Accuracy on basis of Voting as well as Weights...")
    w_acc=Counter()
    u_acc=Counter()
    for myTestDocNo in myTestSet.keys():
        unweighted,weighted=cosineSim(myTrainSet, myTestSet[myTestDocNo],myTestDocNo,labels,k)
        w_acc[weighted]+=1
        u_acc[unweighted]+=1
    print("For k= "+ str(k) +" Weighted: " +str(100*(float(w_acc[True])/float(len(myTestSet))))
```

- The file paths are stored in variables *matrix_file* and *labelFile*.
- The User input k is taken and the Splitting is performed by *loadDataFile*() function. It returns the Training Set, Test Set, and the Labels list.
- The *main*() function also performs Accuracy calculation based on the results obtained from *cosineSim*() function. Both the Weighted and Unweighted results are obtained and Counted for Accuracy measure using the *Counter*()
- There is no Error handling as such implemented for integer value of k because the focus is on getting a better accuracy in a stipulated amount of time more than anything else.

loadDataFIle():

- This function comprises of two parts.
    - One: Loading and splitting of the matrix file.
    - Two: Loading and splitting of the Labels file.

```
def loadDataFile(matrix_file,labelFile):
    matrix_file=open(matrix_file)
    i=0
    myTrainSet={}
    myTestSet={}
    rows=random.sample(range(1,1840),1839)
    for line in matrix_file.readlines():
        i=i+1
        if(i>2):
            docNo,word,freq=line.strip().split(' ')
            if(int(docNo) in rows[:int(len(rows)*0.7)]):
                if(int(docNo) not in myTrainSet.keys()):
                    myTrainSet[int(docNo)]={}
                myTrainSet[int(docNo)][int(word)]=int(freq)
            else:
                if(int(docNo) in rows[int(len(rows)*0.7):]):
                    if(int(docNo) not in myTestSet.keys()):
                        myTestSet[int(docNo)]={}
                    myTestSet[int(docNo)][int(word)]=int(freq)
```

- The function considers the matrix file as a normal Text File with multiple Rows and data separated by spaces. Each Row is split into three parts: The *docNo*, *word* and the *freq*.
- Multi-Dimensional Python Dictionary is used to store data obtained by the splitting. Also, the Documents are shuffled and split into Training Set and Testing Set in the Ratio 70:30.
- Shuffling of documents gives randomness to the Training Set which also in some ways helps in Predicting the classes better.

```
    labels=[]
    labelFile=open(labelFile)
    for line in labelFile.readlines():
        n,label=line.strip().split(',')
        labels.append([int(n),label])
    return myTrainSet,myTestSet,labels
```

- The Second part is for the label file. The data here is again split using commas. The data thus obtained is stored in a List. The Function returns *myTrainingSet*, *myTestSet* and labels to the *main*() function.

```
def cosineSim(myTrainSet,myTestDoc, myTestDocNo,labels,k):
    cosine_values={}
    for doc in myTrainSet.keys():
        totalFrequency=0;
        totalFreqOfTrainDoc=0;
        totalFreqOfTestDoc=0
        CommonWords=(set(myTrainSet[doc].keys()) & set(myTestDoc.keys()))
        for word in CommonWords:
            totalFrequency+=myTestDoc[word]*myTrainSet[doc][word]
        for word in myTrainSet[doc].keys():
            totalFreqOfTrainDoc+=(myTrainSet[doc][word])**2
        for word in myTestDoc.keys():
            totalFreqOfTestDoc+=(myTestDoc[word])**2
        cosine_values[doc]=float((totalFrequency/(((totalFreqOfTrainDoc)**0.5)*((totalFreqOfTestDoc)**0.5))))
        if (cosine_values[doc]==1.0):
            cosine_values[doc]=0.999
    return unweightedClass(cosine_values,labels,myTestDocNo,k),weightedClass(cosine_values, labels,myTestDocNo,k)
```

- The similarity Measure used in my Program is the **Cosine Similarity**.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

- In terms of Text, A and B are the two Documents that are being compared. I am computing the frequencies of common words between A and B and then multiplying the Frequencies for each word and summing it all up for the Numerator.
- The Denominator considers all the Words from each participant and the Sum of their Squares of Frequencies are multiplied after Square-rooting.
- The Cosine Similarity Value ranges between 0 and 1, 0 meaning absolute no Similarity between the two and 1 meaning the Participants being completely Similar.
- There are some cases in this very dataset where 2 documents are absolutely similar i.e. the Cosine Similarity score between them is 1. To avoid a *DivideByZero* Error while calculating Weights in the further explained functions, I have changed the similarity score of these special cases to 0.999. This would give a non-zero value for Distance=1-Similairty.

Going through the Weighted and Unweighted kNN Classifiers:

weightedClass*(cosine_values, labels,myTestDocNo,k)*

```
def weightedClass(cosine_values, labels,myTestDocNo,k):
    votingGroup=Counter()
    for key in cosine_values.keys():
        distance=(1-cosine_values[key])
        cosine_values[key]=float(1/(distance))
    cosine_distance_sort=sorted(cosine_values, key=cosine_values.get, reverse=True)
    for element in range(0,k):
        votingGroup[labels[cosine_distance_sort[element]-1][1]]+=1
    topClass=sorted(votingGroup,key=votingGroup.get, reverse=True)
    if (topClass[0]==labels[int(myTestDocNo)-1][1]):
        return(True)
    else:
        return(False)
```

- For Weighted kNN Classification, I have converted the Cosine Similarity to Distance first and then Inverted Distance to get the Weighted scores between each of the Training Set instances and the Test Instance passed.
- Since, Distance = 1-Similarity, the same has been used in my function. These weights are then sorted in descending order in the *cosine_distance_sort*. Then, by setting up a *votingGroup* Counter I have gathered the class labels of each of the k Train Set documents only. Based on the Voting for highest k values of weights, the class is predicted and then compared to the actual class to get accuracy. If the prediction is accurate, True is returned back to the *cosineSim*() function and False otherwise.
- For instances with Equal Predictions of class (Mostly for Even k values), the method to resolve this clash that I have used is simply pick either of the two. I chose this method because the complexity would get affected by adding a function for perhaps another tie breaker. It is a trade-off for Complexity Vs Accuracy. Having a high complexity solution would give a better accuracy than I have achieved but the solution would take a lot of more time to execute.

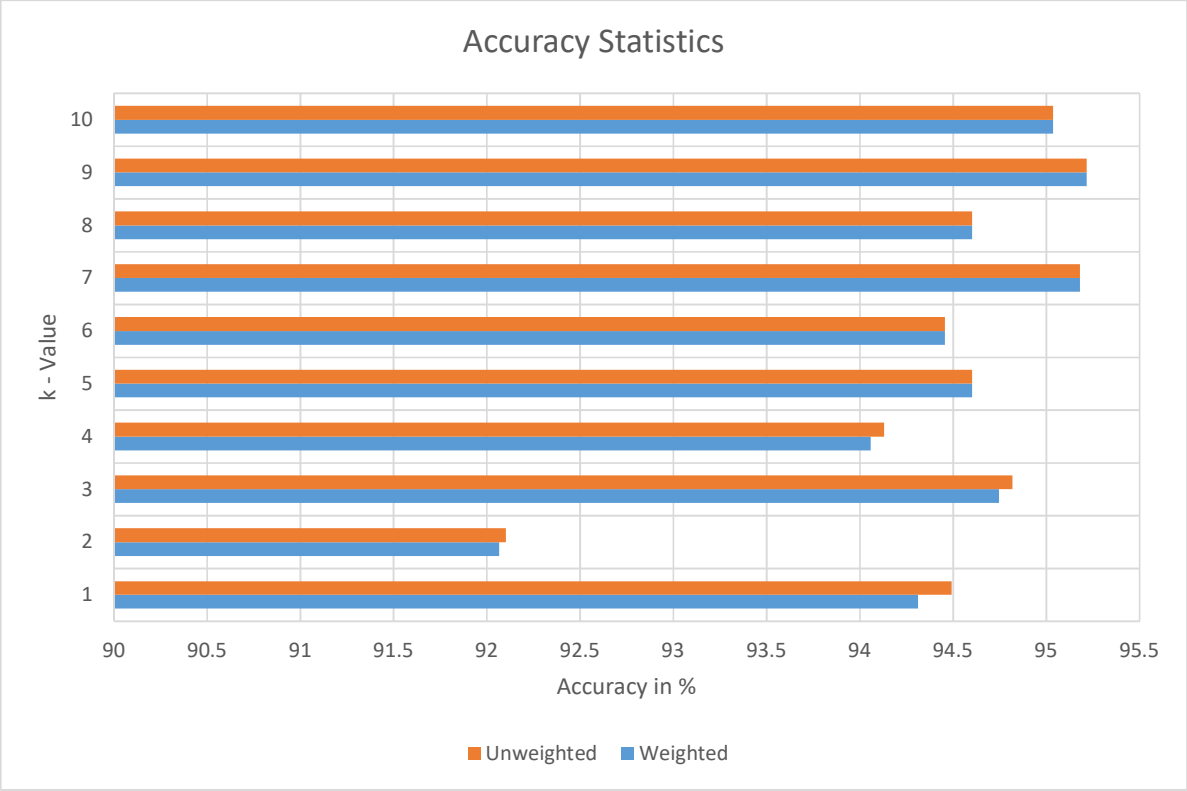
unweightedClass*(cosine_values, labels,myTestDocNo,k)*

```
def unweightedClass(cosine_values, labels,myTestDocNo,k):
    cosine_values_sort=sorted(cosine_values,key=cosine_values.get, reverse=True)
    votingGroup=Counter()
    for element in range(0,k):
        votingGroup[labels[cosine_values_sort[element]-1][1]]+=1
    topClass=sorted(votingGroup,key=votingGroup.get, reverse=True)
    if (topClass[0]==labels[int(myTestDocNo)-1][1]):
        return(True)
    else:
        return(False)
```

- The Unweighted uses the same voting method and sorting methods but in this case, the values considered are Simple Cosine Values i.e. the Similarity measure itself.
- The cosine values are sorted in descending order to obtain k most similar documents to the Query document and then its class is predicted. The highest counted class label in the *votingGroup* is matched with the actual class of the Query document. If Accurate, True is returned, else False.

Reporting of Average Accuracy Scores obtained Over 5 iterations of kNN from k= [1..10] and Observations for the given Dataset:

- I have Run the Code for 5 times over k [1-10] and averaged results. This is because every time the Classifier is iterated, it generates a random Training Set and Test Set but in the ratio 70:30. Averaging results over 5 iterations means given 5 combinations of Training Set and Test Set, the k has been changed from 1 to 10 and the results have been gathered.



Observations:

- From the graph above, it can be seen that the Unweighted Classifier performs better than the Weighted Classifier for some cases of k which makes sense as the Unweighted Classifier calculates directly based on the Similarity and not some derived parameter as Weight or Distance.
- Both the Classifier perform with same accuracy in some of the cases. It really depends on the random Training Set and Test Set created.
- The Accuracy achieved in any of the cases is above 92% in my program. The highest Accuracy achieved is about 95.18%.
- The accuracy however, could be increased in some cases by implementing a different Tie-Breaker method which would trade-off for an enormous amount of Time. On my machine the code would take just under 100 Seconds for any value of k.