Student Name	Nitin Yadav
Student Number	16212304
Email	Nitin.yadav2@mail.dcu.ie
Programme	MCM
Project Title	Text Classification
Module Code	CA-684
Lecturer	Dr. Qun Liu
Project Due	23 April 2017

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying is a grave and serious offence in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion, or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.
I have not copied or paraphrased an extract of any length from any source without identifying the source and using quotation marks as appropriate. Any images, audio recordings, video or other materials have likewise been originated and produced by me or are fully acknowledged and identified. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. I have read and understood the referencing guidelines found at http://www.library.dcu.ie/citing&refguide08.pdf and/or recommended in the assignment guidelines.
I understand that I may be required to discuss with the module lecturer/s the contents of this submission.

Date – 20 April 2017

Text Classification

Abstract

This paper proposes to use the unweighted and weighted features KNN algorithm for text classification. Text classification has an increasing importance since it allows automatic text organisation, and k-nearest neighbour classification is an easy method to understand it, as illustrated in the report.

1. Introduction

Text classification is the process of identifying the class to which a text document belongs. As the volume of information available on the internet and intranets continues to increase, there is a growing need for creating ways to help people in finding, filtering and managing resources. Text classification process get succeeded by comparing the content of the document with some predefined categories. Classification is used to organise data automatically where manual organisation of the data is impossible. It has become a method that helps is solving the real-world problems.

I have chosen to go on with my project using the KNN algorithm. I have come to this conclusion after understanding all the different approaches to text classification problem. In the 2 section, I have given the review of different approaches to text classification problems.

In the 3 section, I have talked about the Experiment implementations, which consists the description of the evaluation method, results and their analysis.
In the 4 section, I have concluded about the learning, which I have gained by doing this project.

2. Different Approaches

Some key methods, which are commonly used for text classification are Decision Tree methods, Probabilistic and Naive Bayes classifiers, SVM classifiers and KNN classification.

- **Decision Tree Classifiers**

A decision tree is essentially a hierarchical decomposition of the training data space, in which a predicate or a condition on the attribute value is used in order to divide the data space hierarchically. In reference to the text data, such predicates are commonly the conditions on the presence or absence of one or more words in the document. The division of the data space is performed recursively in the decision tree, until a leaf nodes contain certain minimum number of records. The majority class label in the leaf is used for the purpose of classification.

- **Probabilistic and Naive Bayes Classifiers**

Probabilistic classifiers are designed to use an implicit mixture model for generation of the underlying documents. This mixture model typically assume that each class is a component of the mixture. Every single component of a mixture is critically a generative model. Which provides the probability of sampling a specific term for the class or component. This is the reason why such kind of classifiers are also known as generative classifier. Perhaps, Naive Bayes classifier is the simplest and commonly used generative classifiers. It models the distribution of every single document of the class by using a probabilistic model with independence assumption about the distributions of different terms.

- **SVM Classifiers**

The main principle of the SVM classifier is to determine separators in the search space which can best separate the different classes. Support vector machines may be independent of the dimension of the feature space, reducing computational complexity is an essential issue to efficiently handle the large number of terms in practical application of text classification.

- **KNN Classification**

The K-Nearest Neighbour is one of the easiest and simplest lazy machine learning algorithms. Algorithm objective is to classify objects into one of the predefined classes of a sample group that was created by machine learning. The algorithm does not require the use of training data to perform classification, training data can be used during the testing phase. KNN is based on finding the most similar objects from sample groups about the mutual Euclidean distance.
I have used this method in the project because of the various reasons, which I have found during the research phase. Which makes it a better choice than the other methods. Some of the main reasons are:

- The KNN classifier is based on the assumption that the classification of an instance is most similar to the classification of other instances that are nearby in the vector space. If I compare it to the other text classification methods such as Bayesian classifier, KNN does not rely on prior probabilities, and it is computationally efficient.
- Unlike the Decision tree method which is a greedy algorithm and is not favourable outcome. KNN was a good choice as it is easier than other methods to understand.

3. Code Implementation and Review

```
def load_dataset():
    dirname = "data"
    News_Document = {}
    News_Labels = {}
    for fn in sorted(os.listdir(dirname)):
        if fn.endswith("news_articles.mtx"):
            with open(os.path.join(dirname, fn), 'r') as f:
                News_Document = f.readlines()
        if fn.endswith("news_articles.labels"):
            with open(os.path.join(dirname, fn), 'r') as f:
                News_Labels = f.readlines()
    return News_Document, News_Labels
```

Here we have created function load_dataset() to read the datasets provided using os.listdir to return list containing the names of the entries in the directory given by path dirname = data. We are opening the files in read mode (r) and using f.readlines() to read the dataset. After reading the is returned to News_Document and News_Labels.

```
def termfreq(News_Document,News_Labels):

    width,height = 4882,1840
    documents = [[0 for x in range(width)] for y in range(height)]
    for x in range(len(News_Document)):
        if(x not in (0,1)):
            count=0
            for y in (News_Document[x].split()):
                if(count==0):
                    doc_id = int(y)
                if(count==1):
                    term = int(y)
                if(count==2):
                    documents[doc_id-1][term-1]= int(y)
                    documents[doc_id-1][-1]=doc_id-1
                count +=1

     width = 1839

    lables=[0 for x in range(width)]
    for x in range(len(news_lables)):
        count=0
        for y in (news_lables[x].split(",")):
            if(count==0):
                doc_id= int(y)
            if(count==1):
                lables[doc_id-1]=y.rstrip()
            count+=1

    return documents,lables
```

Here in the function termfreq() have defined the function to read In the document correctly. As the first two lines if the document are not the data points, so we are ignoring it here.

Also array indexing in python starts from zero whereas document data starts from one .To incorporate this into the array and create no future issues, we have subtracted one from the doc_id.

The same process is for retrieved class labels from the news_articles.labels file. After line one, we read in and strip the data.

```
def load_dataset(split,documents=[],train_set=[],test_set=[]):
    for x in range(len(documents)-1):
        if random.random() < split:
            train_set.append(documents[x])
        else:
            test_set.append(documents[x])
    return train_set, test_set
```

We have created a load_dataset() function to split the data set into training set and test set by using a split parameter ratio, which partitions the data.Any randomyly generated data point that is less than the split value is added to the train_set[ ] and the rest to the test_set[ ].

```
def cosine_distance(instance_one, instance_two, length):
    sumxx,sumxy,sumyy = 0, 0, 0
    for i in range(length):
        x=instance_one[i]; y=instance_two[i]
        sumxx += x*x
        sumyy += y*y
        sumxy += x*y
    return (1- (sumxy/math.sqrt(sumxx*sumyy)))
```

The cosine_distance() function here generates a metric that says how related are two documents by looking at the angle instead of magnitude. The distance is measured by subtracting the similarity measure from one. Ref: http://stackoverflow.com/questions/18424228/cosine-similarity-between-2-number-lists

```
def getNeighbors(train_set, testInstance, k):
    distances = []
    length = len(testInstance)-1
    for x in range(len(train_set)):
        dist = cosine_distance(testInstance, train_set[x], length)
        distances.append((train_set[x],x, dist))

    distances.sort(key=operator.itemgetter(2))

    neighbors = []
    for x in range(k):
        neighbors.append(distances[x][1])

        return neighbors
```

Reference:http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

We now need to find k most similar instances for a given unseen test case. We calculate the distance for all instances and selecting a subset with the smallest distance values. Above the getNeighbors() function returns k most similar neighbours from the training set for a given test instance, using the previously defined cosine_distance() function.

```
def getResponse(neighbors,labels):
    classVotes = {}
    for x in neighbors:
        response = labels[x]
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1

    sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True)

    return sortedVotes[0][0]
```

Reference:http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

Above the getResponse() function gets the majority voted response from a number of neighbours. It assumes the class is the last attribute for each neighbour. Each neighbour votes for their class attribute, and take the majority vote as the prediction (Sorted votes).It returns one response in the case of a draw, else return no response or select an unbiased random response.

```
def getAccuracy(test_set, predictions, labels):
    correct = 0
    for x in range(len(test_set)):
        if labels[test_set[x][-1]] == predictions[x]:
            correct +=1

    return (correct/float(len(test_set))) * 100.0
```

Reference:http://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

Above the getAccuracy() function it calculates the ratio of the total correct predictions out of all predictions and sums the total correct predictions and returns the accuracy as a percentage of correct classifications.

```
def main(split_val,k_val):
    train_set=[]
    test_set=[]
    documents=[]
    labels=[]
    split = split_val
```

The main function prepares the data and all the variables and lists are initialized here in the main body which are called out throughout the program body.

For weighted KNN , only two functions change in the program body. That are :

```
def getNeighbors(train_set, testInstance, k): #returns ks most similar
    distances = []
    length = len(testInstance)-1
    for x in range(len(train_set)):
        dist = cosine_distance(testInstance, train_set[x], length)
        distances.append((train_set[x],x, dist))

    distances.sort(key=operator.itemgetter(2))

    neighbors = []
    weights = []
    for x in range(k):
        neighbors.append(distances[x][1])
        try:
            exec(repr(1/distances[x][2]))
            weights.append(1/distances[x][2])
        except ZeroDivisionError:
            weights.append(0)

    return neighbors,weights
```

The main logic of the getNeigbour() functions remains the same as the un weighted knn function above to returns k most similar neighbours from the training set for a given test instance, using the previously defined cosine_distance() function. The only difference is that while computing this we assign weights to each of them. Here we are weighing the contribution of each of the k neighbours according to their distance to the query point , giving greater weight to closer neighbours.

```
def getResponse(neighbors,labels,weights): #returns the majority voted response from a
    classVotes = {}
    for x in range(len(neighbors)):
        response = labels[neighbors[x]]

        if response in classVotes:
            classVotes[response] += weights[x]
        else:
            classVotes[response] = weights[x]

    sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reverse=True)

    return sortedVotes[0][0]
```

Here to the main logic for the getResponse() function remains the same as un weighted knn to return the majority voted response from a number of neighbours. But instead of adding the votes numerically, we add the weights too that had been calculated and assigned from getNeighbours () function.

And finally we make some changes to the Main() part by assigning a list for weights, i.e. weights = [].

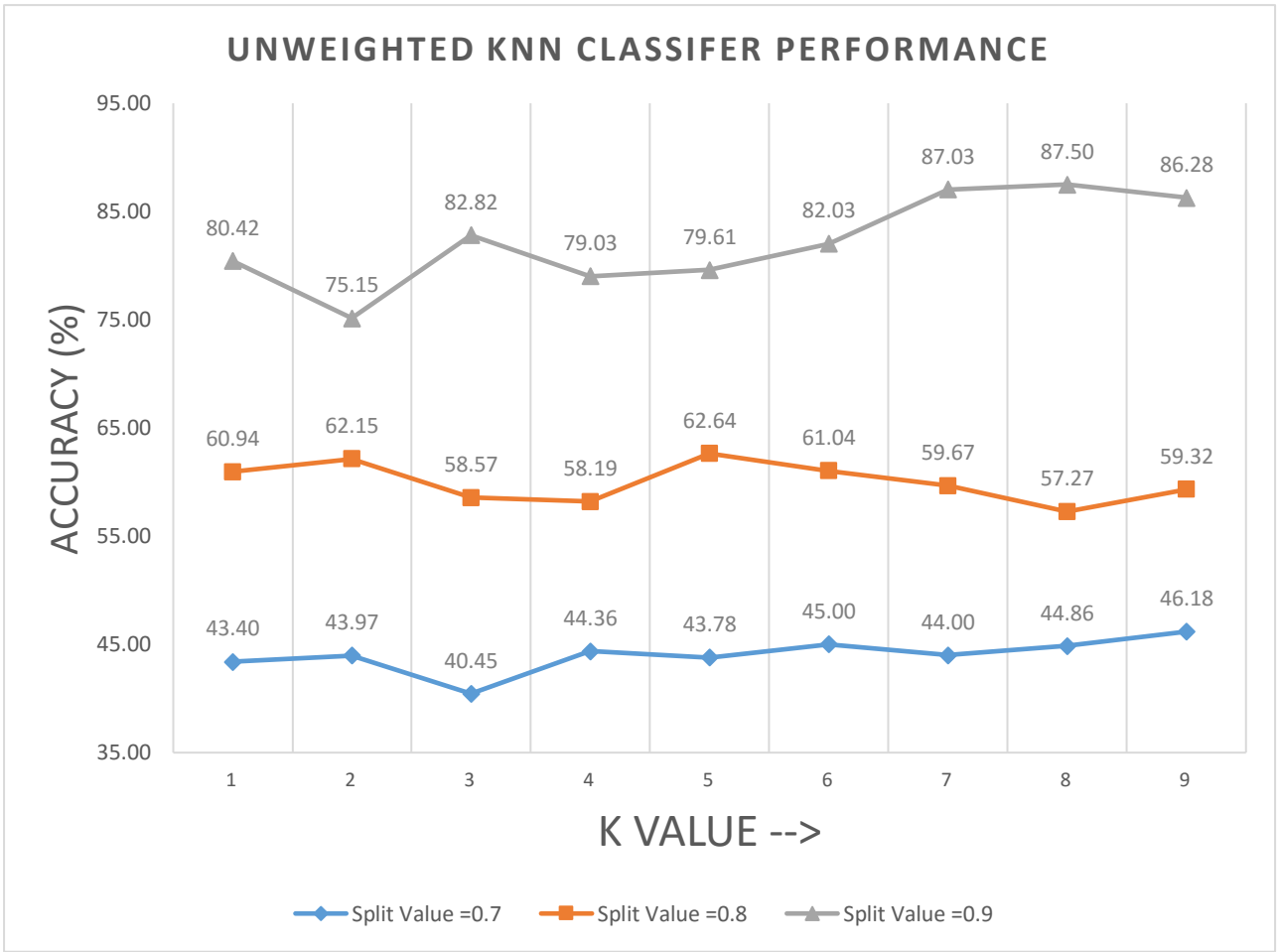
- **Analysis of KNN Classifier Performance with Varying K and Split Value**

We have run the KNN classifier program [refer to knn.py] by varying the value of K from 1 to 10 over different split size for training and testing data set. We have used three different split sizes here: Seventy-Thirty, Eighty-Twenty and Ninety-Ten.

The following accuracies has been obtained after running the classifiers using different parameters mentioned above (unweighted KNN classification results):

Size of K	**ACCURACY (%)[Unweighted KNN]**		
	Split Value =0.7	**Split Value =0.8**	**Split Value =0.9**
1	43.40	60.94	80.42
2	43.97	62.15	75.15
3	40.45	58.57	82.82
4	44.36	58.19	79.03
5	43.78	62.64	79.61
6	45.00	61.04	82.03
7	44.00	59.67	87.03
8	44.86	57.27	87.50
9	46.18	59.32	86.28

Graph for performance of the classifier over different K Values and Splits:



General Observations:

a. Varying Split Value:

- We observe that among the three split values for the given data set, split value of Ninety-Ten gives the best accuracy over varying values of K with an average of 82.21% accuracy.
- The Eighty-twenty split gives us a lower average of 59.98 % accuracy for varying K values but is better than seventy-thirty split.
- The Seventy-thirty split on an average gives the lowest accuracy of 44% amongst all three.

b. Varying K Value:

- For Split Value Ninety-Ten: From K=1 to K=2 there is drop of six percent in accuracy and then it increases for K=3 and again dips for K=4. From K=5 till K=10 there is a gradual increase in accuracy. For K=8, the accuracy is the highest. I.e. 87.5%.

- For Split Value Eight-Twenty: From k=1-2, the accuracy is marginally more but then drops for K=3 and stabilizes till k=4. After that for k=5, the overall prediction accuracy is the highest (62.64%) amongst all K values. After that for the remaining values of K there is a gradual decrease in accuracy.

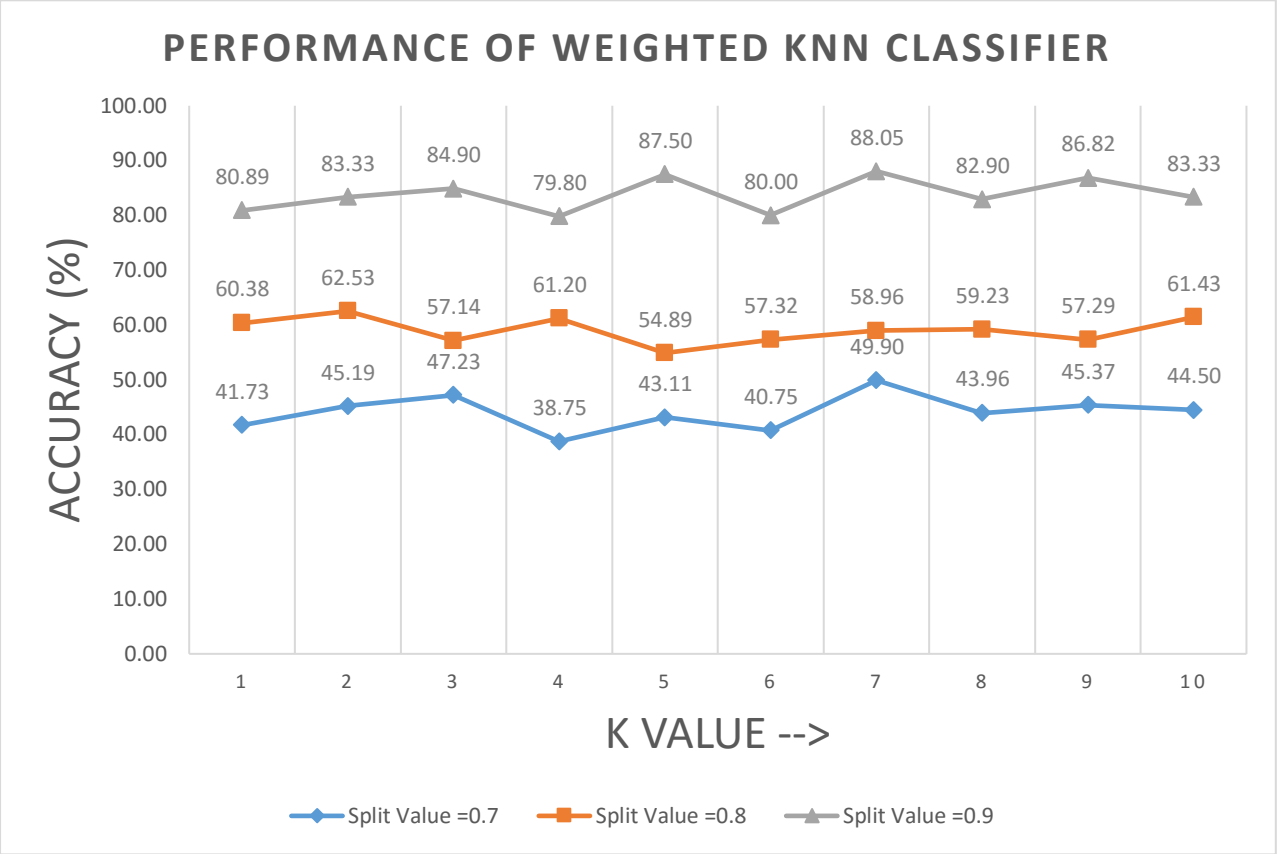
- For split value seventy-thirty: From k=1-2, the accuracy is marginally more but then drops for K=3 and stabilizes till k=4. After that for k=4 till K=10, the overall accuracy is stable and is the highest (46.18%) amongst all K=9.

- In general, we can see that for odd values of K (1,3,5,7,9) the accuracy is high and for most even values of K, the accuracy is lower than its predecessor odd K value.

The following accuracies has been obtained after running the classifiers using different parameters mentioned above (Weighted KNN classification results):

Size of K	**ACCURACY (%)**		
	Split Value =0.7	**Split Value =0.8**	**Split Value =0.9**
1	41.73	60.38	80.89
2	45.19	62.53	83.33
3	47.23	57.14	84.90
4	38.75	61.20	79.80
5	43.11	54.89	87.50
6	40.75	57.32	80.00
7	49.90	58.96	88.05
8	43.96	59.23	82.90
9	45.37	57.29	86.82
10	44.50	61.43	83.33

Graph for performance of the classifier over different K Values and Splits:



General Observations:

a. Varying Split Value:
   - We observe that among the three split values for the given data set, split value of Ninety-Ten gives the highest accuracy (K=7, Accuracy = 88.05%) over varying values of K with an average of 83.75% accuracy.
   - The Eighty-twenty split gives us a lower average accuracy of 59% for varying K values but is better than seventy-thirty split.
   - The Seventy-thirty split on an average gives the lowest accuracy of 44% amongst all three.

b. Varying K Value:
   - For Ninety-ten split, all the odd values of K give higher accuracy than its preceding even K values. For K value =7, it gives the highest accuracy of 88.05%.
   - For Eighty-Twenty split, all the even values of K give higher accuracy than its preceding odd K values. For K value =2, it gives the highest accuracy of 62.53%.
   - For Seventy-Thirty split, all the odd values of K give higher accuracy than its preceding even K values. For K value =7, it gives the highest accuracy of 49.9%.

General Inference for both weighted and Un-weighted KNN classification:

Varying Accuracies for Varying K values:

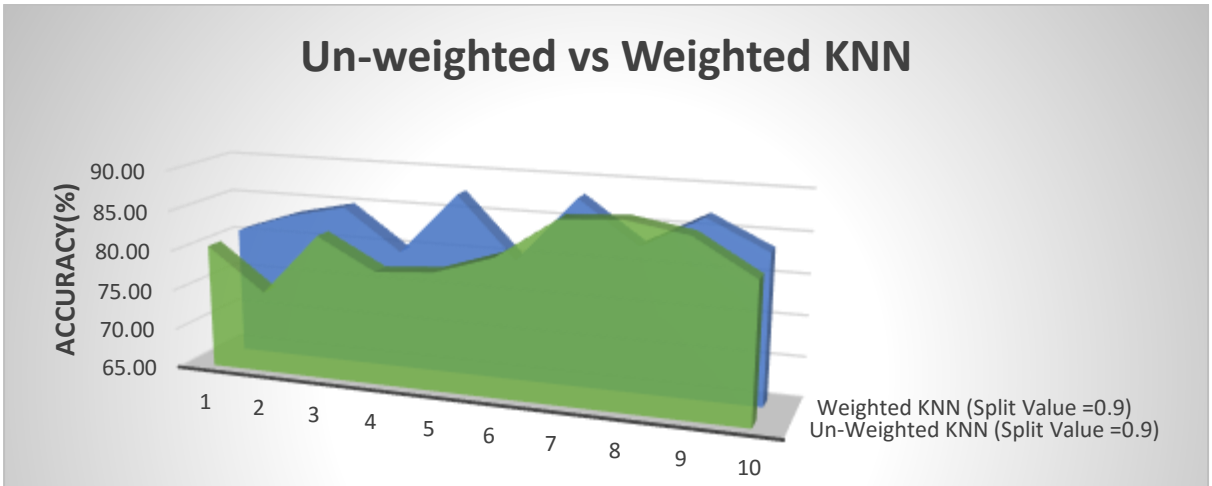
We see that the accuracy pattern for different K values is not some constant or specific pattern. The KNN method is a non-parametric statistical classification technique which supposes that no statistical distribution is fitted to the data of each class. Hence this method tries to predict the class of new data points based on the nearest neighbors. The value of K is extremely training-data dependent, changing the position of a few training data here leads to a

significant loss of performance. The run time increases as well. Therefore, we can say that for any value of K that gives the best accuracy is the optimal k-value for the classifier.

Effect of Varying Split Sizes:

With less training data, parameter estimates have greater variance. With less testing data, performance measure will have greater variance. In fact, data should be divided in such a manner that neither variance is too high, which is more to do with the absolute number of instances in each category rather than the percentage. Therefore, the greater the size of data set, feeding the classifier with more training examples will help classify the unseen data better and increase accuracy. From the above accuracy results we can see that 90-10 split on an average provides the best accuracy result for both weighted and un-weighted than 80-20 and 70-30 splits.

Weighted vs Un-weighted KNN:



	1	2	3	4	5	6	7	8	9	10
Un-Weighted KNN (Split Value =0.9)	80.42	75.15	82.82	79.03	79.61	82.03	87.03	87.50	86.28	82.00
Weighted KNN (Split Value =0.9)	80.89	83.33	84.90	79.80	87.50	80.00	88.05	82.90	86.82	83.33

Un-Weighted KNN (Split Value =0.9) Weighted KNN (Split Value =0.9)

Observation: On an average, we can see that the Weighted KNN classifier performs better at predictions and gives better accuracy results than un-weighted KNN classifier.

4. Conclusion

To improve voting and accuracy in KNN, one method is not to give 1 vote to all the neighbours. We use weighted KNN. Here each point has a weight which is typically calculated using its distance. We have used inverse distance weighting. Under inverse distance weighting, each point has a weight equal to the inverse of its distance to the point to be classified. This means that neighbouring points have a higher vote than the farther points. The accuracy increases when we increase k but the computation cost also increases.

NOTE: ALL THE ACCURACY OUTPUTS BY THE CONSOLE HAS BEEN SAVED. REFER TO OUTPUTS.PDF FOR REFERENCES.

References

1. Trstenjak, B., Mikac, S. and Donko, D. (2013). KNN with TF-IDF based Framework for Text Categorization. http://www.sciencedirect.com/science/article/pii/S1877705814003750
2. Liao, Y. (2002). Review of K-Nearest Neighbor Text Categorization Method. [online] Usenix.org. https://www.usenix.org/legacy/event/sec02/full_papers/liao/liao_html/node4.html
3. Soucy, P. and Mineau, G. (2001). A simple KNN algorithm for text categorization - IEEE Xplore Document. http://ieeexplore.ieee.org/document/989592/?reload=true
4. Kırmemiş, Ö. and Toker, G. Text Categorization and k-Nearest Neighbor Classification. http://user.ceng.metu.edu.tr/~e120321/survey.pdf
5. Mulani, N. and Kalshetty, Y. (2013). Discussion on Different Algorithm for Text Classification and Feature Selection.

Appendix

Github Repository: https://github.com/NitinYadav20/Machine-Learning

- This link contains all the source code, dataset and screenshots.