Clustering of Documents using various techniques and performance evaluation

Department of Computer Science and Technology

Submitted by:

Mortha Sai Sriram	**Id No:**510517056
Sajal Soni	**Id No:**510517061
Bhanupratap Singh	**Id No:**510517074
Paras Jain	**Id No:**510517077
Shashwat Srivastava	**Id No:**510517083

Head of the Department

Prof. Sekhar Mandal

Mentor

Prof. Asit Kumar Das

Acknowledgement

We would like to express our gratitude towards our mentor, Prof. Asit Kumar Das, Department of Computer Science & Technology, for guiding us throughout this semester in making this mini project. Also, we thank our respected Head of the Department, Prof. Sekhar Mandal, and all other professors, for giving us this opportunity of learning something different through this mini project.

This work would surely help us in some way or the other in the future.

Date:

Mortha Sai Sriram:

Sajal Soni:

Bhanupratap Singh:

Paras Jain:

Shashwat Srivastava:

Contents:

OBJECTIVE:

Our motive was to process the text data available and manipulate it further so that we could categorize them into classes. This processed data can be further used for data retrieval and text mining. Natural language processing is one way to process and analyze large sets of documents. The task of retrieving data from a user-defined query has become so common and natural in recent years that some might not give it a second thought. Query retrieval can be described as the task of searching a collection of data, be that text documents, databases, networks, etc., for specific instances of that data.

In order to perform operation on the text data available to us, we first need to convert them into some numerical form. This could be done by the following methods:

- Bag of Words.
- TF-IDF.
- Vectorization of words.

We considered a group of 100 documents to perform the operations on them. The main aim is to build a model which is able to divide the documents into their respective classes.

We processed the data available to us by processes such as tokenization, removal of stopwords, lemmatization, Vectorization and clustering of the documents in order to achieve the target.

TEXT PREPROCESSING

Tokenization:

- **Tokenization** is the method of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols and other elements called tokens.
- We break the document into words. It involves removal of all the punctuation marks from the document which leaves us with only the words defined in the document.
- Further, all the words of the document are stored in a list.

Stopwords:

- **Stop words** are natural language **words** which have very little **meaning**, such as "and", "the", "a", "an", and similar words.
- We have to remove all the useless data from the documents. So, we remove stopwords from all the documents by using the library functions of python.

Lemmatization:

- This method makes use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word.
- We, in our project have also used lemmatization for obtaining the root words. The library function used is **WordNetLemmatizer.**

Term Frequency:

- Term frequency is defined as the frequency of all the words in a particular document.
- Term frequency($tf_{i,j}$)= $n_{i,j}/\sum n_j$ for i-th word in j-th document.

Inverse Document Frequency:

Inverse document frequency(idf) is a method to calculate the weight of the words across all the document taken into consideration.

Idf for any word is mathematically defined as, idf (w) = log (N/df_w)

N=Total number of documents taken into consideration.

df_w =The number of documents in which the word is present.

Tf-Idf:

- In information retrieval, tf–idf or TFIDF, short for term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.
- Tf-idf for each word is calculated by multiplying the tf of a word and its corresponding idf.

We calculated the tf-idf of each of the words in the documents and stored the result in a database.

INTRODUCTION

Text classification is the problem of automatically categorizing a set of documents into various classes based on their content. The increased volume of information available in digital form and the need of organizing it has generated and progressively intensified the interest in automatic text categorization.

This is achieved by designing a model which could divide the documents into various classes based on their contents with the help of algorithms and various tools available.

We worked on python programming language and thus were able to use Natural language tool kit (NLTK), which provides various libraries which helped us in this project.

Representation of Document

A document can be represented in various ways such as unigrams, bigrams, trigrams ,4-grams and so on. To teach a machine how to perform natural language processing we need provide our model with different patterns of the words from the document. One way of doing so is by the use of N-grams.

N-gram is basically a set of occurring words within a given window so when:

- n=1 it is Unigram – forming a list of all the words present in the document.
- n=2 it is bigram – forming a list of two consecutive words in the document.
- n=3 it is trigram – a list of three consecutive words taken together.

The frequency distribution of every bigram in a string is commonly used for simple statistical analysis of text in many applications, including in computational linguistics, cryptography, speech recognition, and so on.

Example: If input is “Wireless speakers for television”

N=1 Unigram- Output - “Wireless”, “speakers”, “for”, “television”

N=2 Bigram- Output - “Wireless speakers”, “speakers for”, “for television”

N=3 Trigram – Output - “Wireless speakers for”, “speakers for television”

We have used Bigram to perform further operations with our model. We formed a list of the bigrams for each of the documents and stored them together in a dictionary.

bigram_dict = { ($n_{(1,1)}$,$n_{(1,2)}$,.....$n_{(1,k1)}$) , ($n_{(2,1)}$,$n_{(2,2)}$,.....$n_{(2,k2)}$) , ...($n_{(j,1)}$,$n_{(j,2)}$.....$n_{(j,kj)}$) }

Here, j = Number of documents and (k1,k2,k3,……..kj) are the corresponding number of bigrams present in each of the document.

Code snippet of how to form the dictionary of bigrams for all the documents:

```
def newBigramList(lem_words):
    length=len(lem_words)
    new_list=[]
    for i in range(0,length-1):
        new_list.append(lem_words[i]+lem_words[i+1])
    return new_list
```

After forming bigrams, we used two ways to cluster data. They are

1. Vectorization
2. TFIDF

First, we will discuss about Vectorization

1.Vectorization

Processing of the bigrams requires the conversion of these into some mathematical form so that their relation with other bigrams can be analyzed with ease. So we converted the bigrams into a four dimensional vector. By using vast amounts of data, it is possible to have a neural network learn good vector representations of words that have some desirable properties like being ***able to do math*** with them. For this we used Google's Word2Vec tool which provides an efficient implementation for computing vector representations of words. The **word2vec** tool takes a text corpus as input and produces the word vectors as output. It first constructs a vocabulary from the training text data and then learns vector representation of words. The resulting word vector file can be used as features in many natural language processing and machine learning applications.

Word2vec basically places the word in the feature space is such a way that their location is determined by their meaning i.e. words having similar meaning are clustered together and the distance between two words also have same meaning.

Code for vectorization of bigrams using Word2Vec:

```
import gensim
model=gensim.models.Word2Vec(docwords,min_count=1,size=4)
print(choice)
X= model[model.wv.vocab]
print(X)
```

After having converted all the bigrams into vectors, the problem now was to device a way in which we could relate these to one another. Also, we were working on a hundred documents and each document and thus the total number of bigrams present in our dataset would be large. This, may increase the time of execution of our model when the dataset may be increased further.

The above two problems can be solved by clustering of the dataset. We used the distance between the vectors as the parameter for clustering of all the vectors into a hundred clusters.

By the use of Word2Vec, the words having similar meaning are placed closer to each other i.e. higher the similarity of the words, lesser will be the distance between the two vectors. This property is used by us and we finally formed hundred different clusters of bigrams using the K-Means clustering algorithm.

K-Means Clustering

K-means is one of the simplest unsupervised learning algorithms. The procedure follows a simple and easy way to classify a given data set through a certain number of clusters (assume k clusters) fixed apriori. The main idea is to define k centers, one for each cluster. These centers should be placed in a cunning way because of different location causes different result. So, the better choice is to place them as much as possible far away from each other.

The next step is to take each point belonging to a given data set and associate it to the nearest center. When no point is pending, the first step is completed and an early group age is done. At this point we need to re-calculate k new centroids as barycenter of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new center. A loop has been generated and as a result of this loop we may notice that the k centers change their location step by step until no more changes are done or in other words centers do not move any more.

We have implemented this algorithm with k-value of hundred.

Clusters to Vector:

We now have hundred different clusters containing the bigrams with some similarities. We now aimed at converting every document into a hundred-dimensional vector. The main objective behind this:

- Every document can now be worked upon as a similar entity.
- Relating the clusters having bigrams to the document from which it has been taken.

Each dimension of the vector represents the number of bigrams present in that cluster for the corresponding document.

Tabular Representation of the vector:

Cluster → Document	Cluster1	Cluster2	…….	Cluster 100
Doc1	F_{11}	F_{12}		F_{1100}
Doc2	F_{21}	F_{22}		F_{2100}
Doc x	F_{x1}	F_{x2}		F_{x100}

Categorizing into classes:

After having the vector of hundred dimensions for each document, our task is to classify these into a set of classes, each of which will consist of similar documents. The similarity of documents is computed with the help of different clustering algorithms based on which the documents are divided into various classes.

We use different clustering algorithms to do this and also evaluate these algorithms. The clustering for which the evaluation efficiency is maximum is considered by us as the final result.

2.TFIDF

After forming the TFIDF dictionary, we formed a dictionary of document number as key and value containing list of words with non-zero TFIDF values for that document.

For each pair of documents, we found the ratio of cardinal number of intersection and union of list of words with non-zero TFIDF values for that pair of documents and formed an adjacency matrix.

If newdict is the dictionary formed from step-1 and M is the adjacency matrix formed from step-2, then M[i][j]=M[j][i]=n(intersection(newdict[i],newdict[j]))/n(union(newdict[i],newdict[j]))

Then we found the average of all the values of adjacency matrix M, and considered it as the threshold value.

Let threshold value= thr

We form the graph from the adjacency matrix in such a way that if in the adjacency matrix M, the value of M[i][j] is greater than the threshold value (thr) then, we consider an edge with document i and j with weight equal to M[i][j].

Then we apply different clustering algorithms on the graph formed, and calculated DB index and Dunn index.

Clustering algorithms:

Clustering is the process of grouping together objects based on their similarity to each other. In the field of Natural Language Processing (NLP), there are a variety of applications for clustering. The most popular ones are document clustering in applications related to retrieval and word clustering for finding sets of similar words or concept hierarchies. Traditionally, language objects are characterized by a feature vector. These feature vectors can be interpreted as points in a multidimensional space. For dividing the vectors of documents into their respective classes we considered a number of clustering algorithms. The efficiency of all these were examined using the applicable methods and then the best result was chosen as the final output.

There were mainly four different clustering algorithms on which we processed the hundred-dimensional vectors. These were:

- Infomap clustering
- K-means clustering
- Louvain clustering
- Fast-greedy clustering

The algorithms for all of these methods have been explained below.

Infomap Clustering

- Infomap is graph based clustering algorithm capable of achieving high quality communities.
- The Infomap algorithm is based on the principles of information theory.
- Infomap characterizes the problem of finding the optimal clustering of a graph as the problem of finding a description of minimum information of a random walk on the graph.
- The algorithm maximizes an objective function called the Minimum Description Length, and in practice an acceptable approximation to the optimal solution can be found quickly.
- First, we convert the vectors to a graph and then apply the clustering algorithm to it.

The code snippet for implementation of Infomap clustering algorithm:

```
from igraph import *
g=Graph.Read("Finaldata.gml")
d=g.community_infomap()
print("Infomap Clustering Algorithm")
print (d)
```

K-Means Clustering:

As already discussed earlier, k-means clustering is a method of vector quantization, that is popular for cluster analysis in data mining.

K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster.

The code for executing the k-means algorithm in our project:

```
from nltk.cluster import KMeansClusterer

NUM_CLUSTERS=100

kclusterer = KMeansClusterer(NUM_CLUSTERS, distance=nltk.cluster.util.euclidean_distance,
repeats=25)

assigned_clusters = kclusterer.cluster(X, assign_clusters=True)

print (assigned_clusters)
```

The time complexity of K-means clustering algorithm is of the order of $O(N^2)$.

Louvain Clustering:

- Each dataset, i.e. the document vector is treated as a node and connected to the other similar nodes to form communities using the dataset.
- Once this step is over the communities formed are treated as nodes and the same operation is iterated through till we reach a saturation point.
- If there is no significant change in the node configuration after the algorithm being implemented on it, no further iterations take place.

The time complexity of this algorithm is estimated to be of the order O(N*log(N)).

Code for the algorithm is given below:

```
from igraph import *
g=Graph.Read_GML("Finaldata.gml")
p=g.community_multilevel()
q=g.modularity(p)
print("Louvain Clustering Algorithm")
print(p)
```

Fast Greedy Clustering:

- Fast Greedy Sparse Subspace Clustering (FGSSC) algorithm providing an efficient method for clustering data belonging to a few low-dimensional linear or affine subspaces.
- The main difference of our algorithm from predecessors is its ability to work with noisy data having a high rate of erasures (missed entries with the known coordinates) and errors (corrupted entries with unknown coordinates).

Code for the algorithm is given below:

```
from igraph import *
g=Graph.Read("Finaldata.gml")
d=g.community_fastgreedy()
print("Fast Greedy Clustering ALgorithm\n")
print(d.as_clustering())
p=d.as_clustering()
```

After having implemented the various clustering algorithms we verified the results obtained from them through the validity indices.

The validity indices are used to identify the set of clusters, that are compact, with a small variance between members of the cluster (*intracluster distance*), and well separated (*intercluster distance*), where the means of different clusters are sufficiently far apart compared to intracluster distance.

Validity Indices:

Validity index can be considered as a measure of the accuracy of the partitioning produced by clustering algorithms.

We first find the representative of all the classes of documents. This is done by calculating the average of the intra-cluster distance between the pair of documents for all the classes.

Also, we calculate the inter-cluster distance of all the classes.

Inter-Cluster distance – The distance between a pair of representatives of the classes is defined as the Inter-cluster distance. This signifies that how different are the two classes from one another. Higher the value of inter-cluster distance, greater is the difference in the contents of the two classes i.e. higher is the accuracy of the clustering algorithm implemented.

Intra-Cluster Distance – It is defined as the average of the distance between all pairs of document vectors of the same class. Higher value of this denotes that the documents in the same class varies a lot in their content which denotes that the accuracy of the clustering algorithm implemented is low. Thus the value of inter-cluster distance should be as low as possible.

There are two different validity indices which we considered for our model.

1. Davies Bouldin Index.
2. Dunn Index.

Davies Bouldin (DB) Index:

The Davies–Bouldin index (DBI) (introduced by David L. Davies and Donald W. Bouldin in 1979) is a metric for evaluating clustering algorithms. This is an internal evaluation scheme, where the validation of how well the clustering has been done is made using quantities and features inherent to the dataset. This has a drawback that a good value reported by this method does not imply the best information retrieval.

We calculated the Intra-Cluster distance (S_i), for all the clusters and also the Inter-Cluster distance ($M_{i,j}$) for all pairs of clusters.

Let $R_{i,j}$ be a measure of how good the clustering scheme is. This measure, by definition has to account for $M_{i,j}$ the separation between the i^{th} and the j^{th} cluster, which ideally has to be as large as possible, and S_i, the within cluster scatter for cluster i, which has to be as low as possible.

$$R_{i,j} = (S_i + S_j) / M_{i,j}$$

And, $D_i = \max(R_{i,j})$

If N is the number of clusters,

$DB = (\sum D_i) / N$, $1 <= i <= N$.

DB is called the Davies–Bouldin index.

Lower the value of D_i., better is the clustering implementation.

Dunn Index

The Dunn index (DI) (introduced by J. C. Dunn in 1974) is a metric for evaluating clustering algorithms. This is part of a group of validity indices including the Davies–Bouldin index or Silhouette index, in that it is an internal evaluation scheme, where the result is based on the clustered data itself. As do all other such indices, the aim is to identify sets of clusters that are compact, with a small variance between members of the cluster, and well separated, where the means of different clusters are sufficiently far apart, as compared to the within cluster variance.

There are many ways to define the size or diameter of a cluster. It could be the distance between the farthest two points inside a cluster, it could be the mean of all the pairwise distances between data points inside the cluster, or it could as well be the distance of each data point from the cluster centroid. Each of these formulations are mathematically shown below:

Let C_i be a cluster of vectors. Let *x* and *y* be any two n dimensional feature vectors assigned to the same cluster C_i.

$\Delta_i = \max_{x,y \in C_i} d(x,y)$, which calculates the maximum distance.

$\Delta_i = \frac{1}{|C_i|(|C_i|-1)} \sum_{x,y \in C_i, x \neq y} d(x,y)$, which calculates the mean distance between all pairs.

$\Delta_i = \frac{\sum_{x \in C_i} d(x,\mu)}{|C_i|}, \mu = \frac{\sum_{x \in C_i} x}{|C_i|}$, calculates distance of all the points from the mean.

This can also be said about the intercluster distance, where similar formulations can be made, using either the closest two data points, one in each cluster, or the farthest two, or the distance between the centroids and so on. The definition of the index includes any such formulation, and the family of indices so formed are called Dunn-like Indices. Let

$\delta(C_i, C_j)$ be this intercluster distance metric, between clusters C_i and C_j.

With the above notation, if there are m clusters, then the Dunn Index for the set is defined as:

$$DI_m = \frac{\min\limits_{1 \leqslant i < j \leqslant m} \delta(C_i, C_j)}{\max\limits_{1 \leqslant k \leqslant m} \Delta_k}$$

For a given assignment of clusters, a higher Dunn index indicates better clustering. One of the drawbacks of using this is the computational cost as the number of clusters and dimensionality of the data increase.

Use of Validity Indices in K-Means Clustering:

The validity indices, namely the DB and DI index are used for the determination of best possible value of ‘ K ‘ in K-Means clustering.

The value of k determines the number of clusters to be formed by the clustering algorithm. Thus, we can also calculate the best possible number of classes into which the documents can be divided using the same method.

We calculated the DB and Dunn index for different values of ‘K’ and plot separate graphs for each of them. The local maxima in the DB index graph or the local minima in the graph of Dunn index gives the best value of ‘K’ for K-Means clustering.

RESULTS:

WORD2VEC:

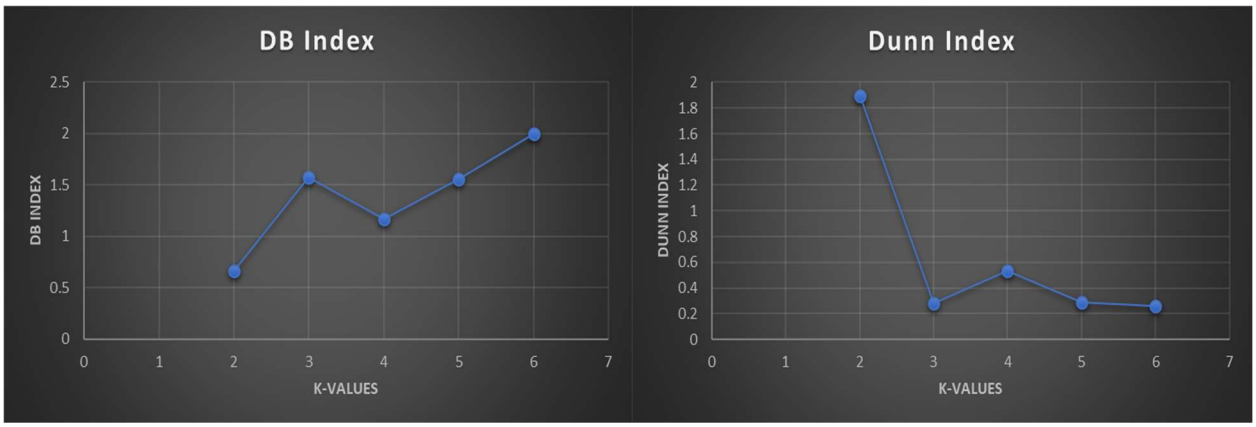
```
root@kali:~/Desktop/Desktop/mini/word2vec# python3 fastgreedy.py
Fast Greedy Clustering ALgorithm

Clustering with 100 elements and 2 clusters
[0] 0, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 17, 18, 19, 20, 21, 22, 25,
    26, 27, 28, 29, 30, 32, 34, 35, 36, 37, 39, 40, 41, 42, 43, 44, 45, 46,
    48, 49, 50, 51, 52, 54, 55, 56, 60, 61, 62, 63, 66, 67, 68, 73, 74, 75,
    76, 77, 79, 81, 82, 83, 84, 85, 88, 89, 90, 92, 93, 96, 97, 98, 99
[1] 2, 13, 16, 23, 24, 31, 33, 38, 47, 53, 57, 58, 59, 64, 65, 69, 70, 71, 72,
    78, 80, 86, 87, 91, 94, 95
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0]
[2.832610076163029, 0.5537125210560263]
```

Fast Greedy Clustering Algorithm

```
root@kali:~/Desktop/Desktop/mini/word2vec# python3 louvain.py
Louvain Clustering Algorithm
Clustering with 100 elements and 3 clusters
[0] 2, 3, 4, 5, 6, 10, 13, 17, 19, 21, 23, 25, 27, 31, 32, 37, 39, 40, 43, 46,
    52, 53, 57, 58, 65, 69, 71, 72, 79, 86, 90, 91, 95, 96, 98, 99
[1] 0, 1, 7, 8, 9, 11, 12, 14, 15, 18, 20, 22, 28, 29, 30, 34, 35, 36, 38, 41,
    42, 44, 45, 48, 49, 50, 51, 54, 55, 56, 60, 61, 62, 63, 66, 67, 68, 73,
    74, 75, 76, 77, 81, 82, 83, 84, 85, 88, 89, 92, 93, 97
[2] 16, 24, 26, 33, 47, 59, 64, 70, 78, 80, 87, 94
[1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 2, 0, 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 1, 1, 1, 0, 0, 2, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 2, 1, 1,
 1, 1, 0, 0, 1, 1, 1, 0, 0, 2, 1, 1, 1, 1, 2, 0, 1, 1, 1, 0, 2, 0, 0, 1, 1, 1, 1, 1, 2, 0, 2, 1, 1, 1, 1, 1, 0, 2, 1, 1, 0, 0, 1, 1, 2, 0, 0, 1, 0, 0]
[7.767866071979509, 0.17659272862794687]
```

Louvain Clustering Algorithm



K-Means Clustering Algorithm for different values of K

At K=2, DB Index is minimum and Dunn Index is Maximum.

K=2 is the best way of Clustering the data.

TFIDF:

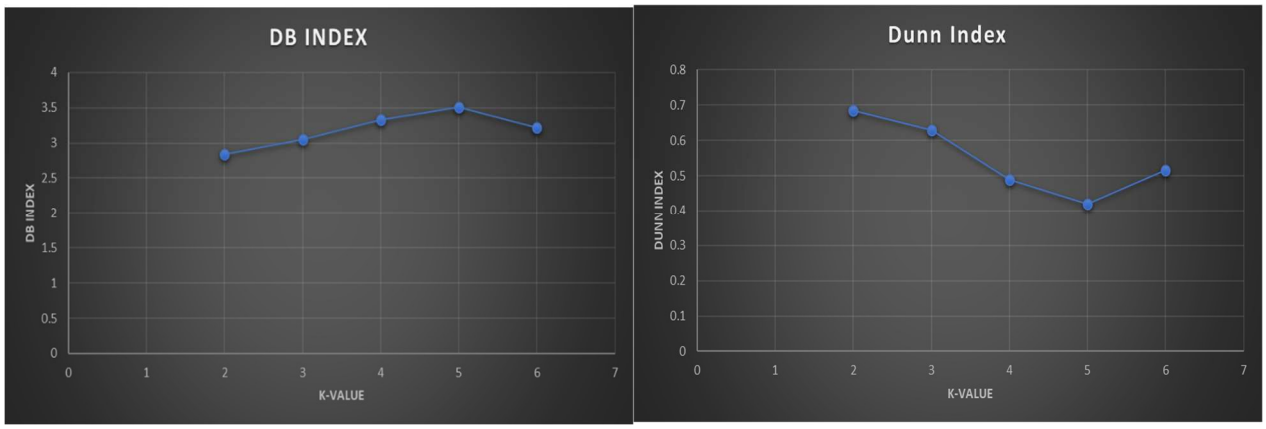
```
root@kali:~/Desktop/Desktop/mini/tfidf# python3 louvain.py
Louvain Clustering Algorithm
Clustering with 100 elements and 9 clusters
[0] 0
[1] 1
[2] 13
[3] 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 20, 25, 26, 28, 33, 35,
    36, 47, 48, 49, 50, 51, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77,
    78, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 92, 93, 95, 96, 97, 98, 99
[4] 34
[5] 32, 37, 38, 55
[6] 31, 52
[7] 17, 18, 19, 21, 22, 23, 24, 27, 29, 30, 39, 40, 41, 42, 43, 44, 45, 46,
    53, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 79
[8] 91, 94
[0, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 2, 3, 3, 3, 7, 7, 7, 3, 7, 7, 7, 7, 7, 3, 3, 7, 3, 7, 7, 6, 5, 3, 4, 3, 3, 5, 5, 7, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3,
 3, 3, 6, 7, 7, 5, 7, 7, 7, 7, 7, 7, 7, 7, 7, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 7, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 8, 3, 3, 8, 3, 3, 3, 3, 3]
[1.982115610844578, 0.5096374798521219]
root@kali:~/Desktop/Desktop/mini/tfidf#
```

Louvain Clustering Algorithm

```
root@kali:~/Desktop/Desktop/mini/tfidf# python3 fastgreedy.py
Fast Greedy Clustering ALgorithm

Clustering with 100 elements and 6 clusters
[0] 0
[1] 1, 2, 5, 7, 8, 10, 11, 12, 14, 15, 16, 19, 20, 25, 26, 33, 36, 39, 45, 46,
    47, 48, 52, 60, 62, 66, 67, 68, 69, 78, 79, 85, 88, 91, 93, 94, 95, 97
[2] 3, 4, 6, 28, 32, 35, 49, 50, 70, 71, 72, 73, 74, 76, 77, 80, 81, 82, 83,
    84, 86, 87, 89, 90, 96, 98, 99
[3] 9, 17, 18, 21, 22, 23, 24, 27, 29, 30, 31, 37, 38, 40, 41, 42, 43, 44, 51,
    53, 54, 55, 56, 57, 58, 59, 61, 63, 64, 65, 75, 92
[4] 13
[5] 34
[0, 1, 1, 2, 2, 1, 2, 1, 1, 3, 1, 1, 1, 4, 1, 1, 1, 3, 3, 1, 1, 3, 3, 3, 3, 1, 1, 3, 2, 3, 3, 3, 2, 1, 5, 2, 1, 3, 3, 1, 3, 3, 3, 3, 3, 1, 1, 1, 1, 2,
 2, 3, 1, 3, 3, 3, 3, 3, 3, 3, 3, 1, 3, 1, 3, 3, 3, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 2, 2, 1, 1, 2, 2, 2, 2, 2, 1, 2, 2, 1, 2, 2, 1, 3, 1, 1, 1, 2, 1, 2, 2]
[2.6014614242531615, 0.4194985066120432]
root@kali:~/Desktop/Desktop/mini/tfidf#
```

Fast Greedy Clustering Algorithm



K-Means Clustering Algorithm for different values of K

At K=2, DB Index is minimum and Dunn Index is Maximum.

K=2 is the best way of Clustering the data.

CONCLUSION:

Using all the methods detailed above, we were able to cluster all the hundred different documents into classes where each class contained documents having related contents.

Our program reads all the documents and the model programmed by us, distributes them into classes. The program output contains the class number indexed from zero and the indexes of the documents contained in each of the classes.

The output is not highly accurate every time the program is run on different datasets but it can be improved by training the model more on more and also using some more tools of Natural Language Processing.

Thus, we conclude this project of Text Processing in Python.

REFERENCES:

- ➔ https://en.wikipedia.org
- ➔ https://medium.freecodecamp.org/how-to-process-textual-data-using-tf-idf-in-python-cd2bbc0a94a3
- ➔ https://www.geeksforgeeks.org
- ➔ https://medium.com/@paritosh_30025/natural-language-processing-text-data-vectorization-af2520529cf7
- ➔ https://www.researchgate.net/publication/320028295_A_Greedy_Clustering_Algorithm_Based_on_Interval_Pattern_Concepts_and_the_Problem_of_Optimal_Box_Positioning
- ➔ https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4938516/