CMPE 255-02 Data Mining Program 2

Submitted By: Kashika Jain **SID: 012505649** **Email: kashika.jain@sjsu.edu**

Rank – 9 **NMI – 0.6953**

PR2: Text Clustering - Objective

The objective of the project is to perform text clustering by implementing the Bisecting K-Means algorithm and evaluate the cluster using best metrics.

Methodology -

Import data and libraries

1. Import relevant libraries required for the text clustering task and import data from file.

```
#import libraries
import numpy as np
from scipy.sparse import *
from math import sqrt
from collections import defaultdict
from sklearn.metrics import calinski_harabaz_score
from sklearn.utils import shuffle
from sklearn.decomposition import TruncatedSVD
```

```
#create sparse matrix
sparse_matrix = get_sparse("train.dat")
```

Data Cleaning and Preprocessing

2. We have a full sparse matrix as input and we will create a matrix to take into account the extra space used by the zeroes in the matrix by creating a csr_matrix. This is implemented by reading the data and checking for non-zero rows and removing the empty rows for creating a compressed matrix

3. To implement this, we use a module in **Scipy,** which has many operations inbuilt for sparse matrices. Here we will use the **csr_matrix** which is short for Compressed Sparse Row. Using this library, we achieve a much smaller matrix which occupies much less space. The CSR format is generally efficient for fast solutions of vector products. The CSR column indices can/cannot be sorted using the **sorted.indices()** function depending on it sorted indices is what is required for our case. Here we are sorting the indices after creating the csr matrix and returning the compressed matrix back for further operations.

```
    sparse_matrix = csr_matrix((x, y, z), shape=(nrows, ncols), dtype=np.float)
    sparse_matrix.sort_indices()
    return sparse_matrix
```

4. Vectorize the data using Inverse document frequency technique.

```
#perform idf
idf_mat = get_idf(sparse_matrix, check=True)
```

5. The calculation of inverse document frequency for the sparse matrix is done by using the indices, data and index pointer values of the sparse matrix. Based on what values are retrieved from the sparse matrix, a document frequency matrix is constructed for data and non-zero rows. The presence value is added for a value in the row based on how many times it appears in the row.

```
#inverse document frequency
def get_idf(sparse_matrix, check=False, **kargs):
    if check is True:
        sparse_matrix = sparse_matrix.copy()
    nrows = sparse_matrix.shape[0]
    nnz = sparse_matrix.nnz
    y, x, z = sparse_matrix.indices, sparse_matrix.data, sparse_matrix.indptr
    dict_ = defaultdict(int)
    for index in y:
        dict_[index] = dict_[index]+ 1
    for key,value in dict_.items():
        dict_[key] = np.log(nrows / float(value))
    for index in range(0, nnz):
        x[index] *= dict [y[index]]
    return dict_ if check is False else sparse_matrix
```

6. After the vectorization of the matrix, I have performed normalization.

```
#perform normalization
normalized_matrix = get_normalized(idf_mat, check=True)
```

7. The next operation performed is normalization so that we have a standard scale for performing all the operations on the sparse matrix and all terms are given relevant importance in the document. Here we are normalizing the data by dividing 1 by the sum of squares of the row values. This is a standard way of normalizing the data to make it standardized for scaling purpose.

```
#normalization
def get_normalized(sparse_matrix, check=False, **kargs):
    if check is True:
        sparse_matrix = sparse_matrix.copy()
    nrows = sparse_matrix.shape[0]
    nnz = sparse_matrix.nnz
    y, x, z = sparse_matrix.indices, sparse_matrix.data, sparse_matrix.indptr
    for index in range(nrows):
        sum_rows = 0.0
        for j_index in range(z[index], z[index+1]):
            sum_rows = sum_rows + x[j_index]**2
        if sum_rows == 0.0:
            continue
        sum_rows = float(1.0/np.sqrt(sum_rows))
        for j_index in range(z[index], z[index+1]):
            x[j_index] *= sum_rows
    if check is True:
        return sparse_matrix
```

8. The next step is to perform dimensionality reduction as we have a high dimensional data and processing this data for text clustering will be an extremely difficult task because of the curse of dimensionality. For the purpose of this assignment, I am using **Truncated SVD** which is imported from the **Sklearn's** decomposition library. I have used the library to reduce the dimensionality by passing the parameters – number of components as 150, number of iterations as 50, random state as 101 and reduction algorithm – **'ARPACK'** which is used for solving the large-scale eigenvalue problems in the matrix-free fashion.

```
#perform dimensionality reduction
reduced_dimensions = TruncatedSVD(n_components=150, n_iter=50, random_state=101,algorithm='arpack')
trunc_sparse=reduced_dimensions.fit_transform(normalized_matrix)
trunc_sparse= csr_matrix(trunc_sparse)
```

Clustering Using Bisection K-Means

9. In the next step, I implement Bisecting K-Means algorithm. Pseudo Code:

   A) Initially, all the points are considered in one centroid(cluster). For this text clustering assignment, we use the total number of clusters as 7(k=7). To reach the desired clustering solution,

   B) Loop until the desired condition for 7 clusters is reached.

   C) For each cluster, calculate the sum of squared error for the parent cluster.

D) Apply K-means for k=2 and calculate the SSE error for the generated new clusters and compare it with the parent cluster.

E) Choose the cluster split that gives the lowest error.

```
#number of clusters, k =7
k=7
#testing for number of iterations = 20
n_iter=20
#number of epochs =2
epoch=2
```

10. In the below snapshot of Bisecting K-means algorithm, I don't need to initialize an K Number of centroids before the algorithm starts as we start with one cluster considering all the points. However, a fixed number of clusters (7 in our case) must be declared on before the main algorithm is started, which will serve as the stopping condition for the algorithm. Splits are decided based on SSE.

```
#bisecting k means algorithm
def bisecting_kmeans(k, sparse_matrix, matrix_index, n_iter=10, epoch=10):

    sp_mat = sparse_matrix[matrix_index,:]
    minimum_index=[None]* (k+1)
    err2= err_min = np.inf

    for epoch in range(1,epoch):
        rand_data = shuffle(sp_mat, random_state=101)
        list_centroid= rand_data[:k,:]
        err1 = err2 = diff =np.inf
        mat_indx=[None]* (k+1)
        for n_iter in range(n_iter):
            list_index = list()
            similarity = sp_mat.dot(list_centroid.T)
            for val in range(similarity.shape[0]):
                t_ind = obj.argsort()[-1]
                t_val = obj[obj.argsort()[-1]]
                list_index.append(t_ind + 1)

            list_centroid = []
            for index in range(1,k+1):
                ind_val = [var for var, lists in enumerate(list_index) if lists == index]
                if (lists1.shape[0] > 1):
                    list_centroid.append(lists1.toarray().mean(0))
            cent1=None
            if(len(list_centroid)>0):
                cent1 = csr_matrix(list_centroid)
            list_centroid = cent1
```

```
            if(list_centroid==None):
                break
            for index in range(1,k+1):
                ind_val = [var for var, lists in enumerate(list_index) if lists == index]
                new_val =[]
                for new_ind in ind_val:
                    new_val.append(matrix_index[new_ind])
                    mat_indx[index]=new_val
            err1=err2
            err_list=[]
            for i in range(1,(len(mat_indx))):
                ind=mat_indx[i]
                if(ind is not None and len(ind)>0):
                    err_list.append(calculate_error(sparse_matrix,ind))
            err2= np.sum(err_list)
            diff = err1 - err2
            if(err2<err_min):
                err_min=err2
                minimum_index=mat_indx
            if(diff < 0.01):
                break;
    return list_index, minimum_index
```

```
#calculate error
def calculate_error(sp_mat,list_index):
    list_centroid = []
    val = sp_mat[list_index,:]
    if (val.shape[0] > 1):
        list_centroid=val.toarray().mean(0)
    if(len(list_centroid)>0):
        return np.sum(np.linalg.norm(val - list_centroid, 2, 1))
    else:
        return 0
```

11. Create the list of final clustering formed by the matrix

```
index=[None]*normalized_matrix.shape[0]
for i in range(1,8):
    for j in bisect_mat[i]:
        index[j]=i
```

Clustering Evaluation

12. Save the file and evaluate the cluster by calculating the calinski harbaz score for the normalized matrix. Here the score is used to evaluate the optimal number of clusters using the Calinski-Harabaz clustering evaluation criterion.

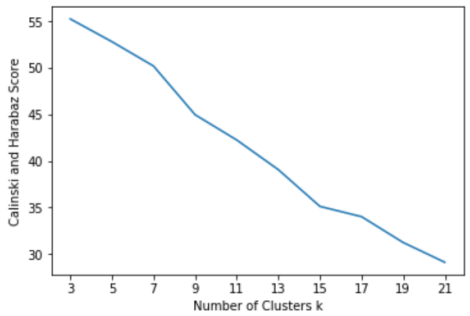
```
#Save to file
def save_output(index):
    file = open("output.dat", "w")
    for j_index in index:
        file.write(str(j_index) +'\n')
    file.close()
```

```
#Evaluation
save_output(index)
print("Clustering Evaluation: ")
pred = calinski_harabaz_score(normalized_matrix.toarray(), index)
print(pred)
```

```
Clustering Evaluation:
50.21417179221697
```

13. Generate the Calinski-Harabaz score for K=1 -21 with 2 step size and plot graph for it.

```
K= 3 Calinski Harabaz Score is 55.279393
K= 5 Calinski Harabaz Score is 52.835868
K= 7 Calinski Harabaz Score is 50.214172
K= 9 Calinski Harabaz Score is 44.976567
K= 11 Calinski Harabaz Score is 42.261867
K= 13 Calinski Harabaz Score is 39.068552
K= 15 Calinski Harabaz Score is 35.118516
K= 17 Calinski Harabaz Score is 34.013958
K= 19 Calinski Harabaz Score is 31.236542
K= 21 Calinski Harabaz Score is 29.110033
```



Conclusion- The Bisecting K-means algorithm is performed successfully, and text is classified into 7 clusters. The cluster evaluation score for the predicted clustering is achieved as 50.2141

References-

1. https://www.mathworks.com/help/stats/clustering.evaluation.calinskiharabaszevaluation-class.html
2. https://www.linkedin.com/pulse/initial-investigation-k-means-bisecting-algorithms-dave-blodgett/
3. https://github.com/rashmishrm/bisect-clustering/blob/master/bisect_clustering.py