Lab Course - Distributed Data Analytics
Exercise 3

1.1 Implement K-means:

The k-means algorithm clusters the data instances into k clusters by using Euclidean distance between data instances and the centroids of the clusters. The detail description of the algorithm is listed on slides 1-10 https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf. However, in this exercise sheet you will implement a distributed version of the K-means. Figure below explains a strategy to implement a distributed K-means. You have to implement distributed K-means clustering using MPI framework. Your solution should be generic and should be able to run for arbitrary number of clusters. It should run in parallel i.e. not just two workers working in parallel but all should participate in the actual work.

Function name : read_data
Parameter : No parameter

This function reads the extracts the documents from the library environment, reads the text from the document, stems the words and finds the TFIDF of each word in the documents of the entire corpus. Once it is obtained, it converts the TFIDF to a matrix of all possible words vs document number with the TFIDF values using vectorizer. This fuction returns a sparse matrix of size 11014x94875 and a dataframe with text and corresponding stemmed text.

```
def read_data():
    """Read, Preprocess, TFIDF and Vectorizing of Data"""
    newsgroup_train = fetch_20newsgroups(subset = "train", remove = ('headers','footers','quotes'))
    text_df = pd.DataFrame([str(dat) for dat in newsgroup_train.data])
    text_df.columns = ["Text"]
    stemmer = PorterStemmer()
    text_df['stemmed'] = text_df["Text"].apply(lambda x: " ".join([stemmer.stem(str(y)) for y in x.split()]))
    vectorizer = TfidfVectorizer(stop_words = 'english')
    text_vector = vectorizer.fit_transform(text_df.stemmed.dropna())
    return text_vector,text_df
```

Function name : initial_centroid
Parameter : sparse matrix and value of k

This function checks makes a list of k random numbers within the size of the sparse matrix and the same is stored in a variable and is returned for further k-means implementation. Here, a 20 random numbers were chosen and corresponding data from the sparse matrix were used for k-means implementation. This function is accessed before parallelising the process.

```
def  initial_centroid(text_vector,k):
    centroid = []
    centroid_pos = list(np.random.randint(text_vector.shape[0],size=k))
    centroid = text_vector[centroid_pos[0],:].todense()
    for i in centroid_pos[1:]:
        temp = np.array(text_vector[i,:].todense())
        centroid = np.concatenate((centroid,temp))
    return(centroid)
```

Function name : euclidean_distance
Parameter : sparse matrix and centroid matrix

This function calculates the Euclidean distance between two instances. As it is the case, it can also be achieved by doing a dot product of two matrices and finding the minimum of the 20 values in a row to get the corresponding cluster of the data. This returns an array of 11014x1 containing the new updated cluster numbers of the text.

```
def euclidean_distance(text_vector,centroid):
    dist = np.sqrt(text_vector.dot(centroid.T))
    cluster = np.argmin(dist, axis=1)
    return(np.array(cluster))
```

Function name : centroid_mean
Parameter : part of clusters from workers, text matrix

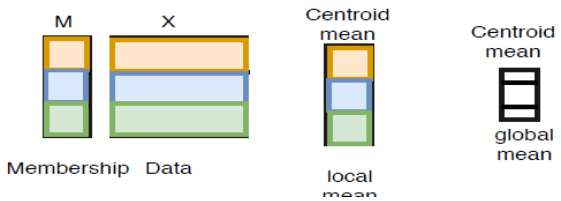
This function uses the list of split clusters of the documents and the sparse vectorised matrix to find the local mean of data with workers. Using the clusters, it uses the corresponding data from the data, sums them up and finds a local mean of the same and sends it to master which is further stacked to be sent to find the global mean.

```
def centroid_mean(cluster,text_vector):
    group_sum = np.array([])
    clusters = np.unique(cluster)
    for clust in clusters:
        group = scipy.sparse.vstack([text_vector[j] for j in range(text_vector.shape[0]) if cluster[j] == clust])
        group_sum = scipy.sparse.vstack((group_sum,np.array(group.sum(axis = 0))))
    group_sum = (group_sum.tocsr())[1:,]
    return group_sum
```

Function name : update_centroid
Parameter : concatenated means and clusters

This function uses the the concatenated local means and uses them to find the global mean. This global mean will now act as the centroid for the next iteration. This process is handled by the master and the result is shared to the workers again as the loop continues till convergence.

```
def updated_centroid(cluster,text_vector):
    new_centroid = np.array([])
    clusters = np.unique(cluster)
    for clust in clusters:
        group = scipy.sparse.vstack([text_vector[j] for j in range(text_vector.shape[0]) if cluster[j] == clust])
        cnt = list(cluster).count(clust)
        new_centroid = scipy.sparse.vstack((new_centroid,np.array(group.sum(axis = 0)))) / cnt
    new_centroid = (new_centroid.tocsr())[1:,]
    return new_centroid
```



MPI Parallelisation Flow:

```
if __name__ == "__main__":
    text_vector = read_datacsv()
    k = 20
    centroid = initial_centroid(text_vector,k)

    cluster = []
    time_list = []
    starttime = time.time()
    for epoch in range(10):
        if rank == 0:
            for i in range(1,size):
                centroid_sum = np.array([])
                old_cluster = cluster
                rows = text_vector.shape[0]
                start = int((i-1)*(rows/(size-1)))
                end = int(((rows/(size-1))*i))
                comm.send(text_vector[start:end,:],dest = i,tag = 1)
                comm.send(centroid,dest = i,tag = 2)
                centroid_sum = scipy.sparse.vstack((centroid_sum,comm.recv(source = i, tag = 3)))
                cluster.extend(comm.recv(source = i,tag = 4))
            centroid = updated_centroid(cluster,text_vector)
            if (cluster == old_cluster).all():
                print('Converged')
                break

        else:
            text_vector = comm.recv(source = 0, tag = 1)
            centroid = comm.recv(source = 0, tag = 2)
            cluster = euclidean_distance(text_vector,centroid)
            centroid_sum = centroid_mean(cluster,text_vector)
            comm.send(centroid_sum,dest = 0,tag = 3)
            comm.send(cluster,dest = 0,tag = 4)
    time_list.append(time.time() - starttime)
```

Initially in the implementation, the data is read using the function *read_data()*. The output of it and the Cluster size is sent as input to *initial_centroid()* function to calculate or randomly initialise the centroids for the k-means process. Once this is completed, the implementation of k-means with parallel processing starts.

In the parallel process, the master uses the initial centroid and sparse matrix, splits the sparse matrix depending on the number of workers and sends the same along with the centroid matrix to the workers. The workers receive the data from master and calculates the distance using *euclidean_distance()* and the uses the output of it to calculate the local mean with *centroid_sum()* function. Performing these steps, the workers sends back the updated centroids and clusters to the master. The master receives them, stacks the centroids from different workers and finds the global mean using *updated_centroid()* function. Once it is done, the master checks of the updated cluster and previous cluster are of similar values. If yes, the model is converged and if not, the iteration continues till convergence. The output with 10 epochs and k being 20 is as follows:

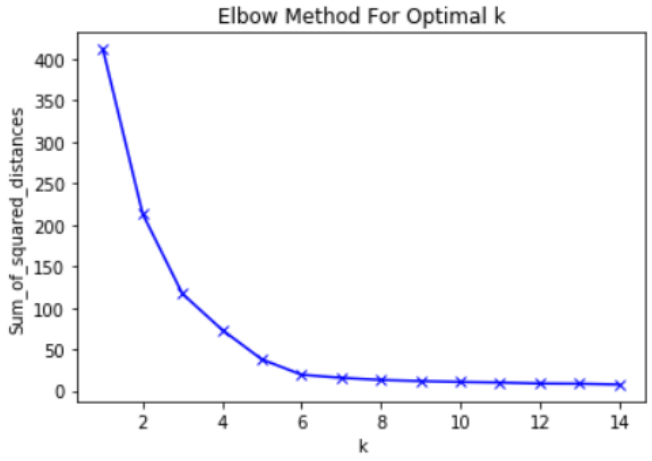
Process	Tp
1	124.37
2	118.68
4	125.22
6	130.57
8	132.07

It can be seen that for a 2 core machine, the process time is at its best when P is 2

1.2 Performance Analysis:

You have to do a performance analysis and plot a speedup graph. First you will run your experiments with varying number of clusters i.e. P =[1,2,4,6,8]. To plot the speedup graph please follow the lecture slides 15 https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf.

For varying the number of clusters, the optimal value of k can be found using the Elbow method. Knowing the value of k will be 20, the implementation was carried out. With elbow method, the point at the elbow is considered optimal k and a sample visualisation of the elbow method is as follows.



https://cdn-images-1.medium.com/max/1200/0*jWe7Ns_ubBpOaemM.png

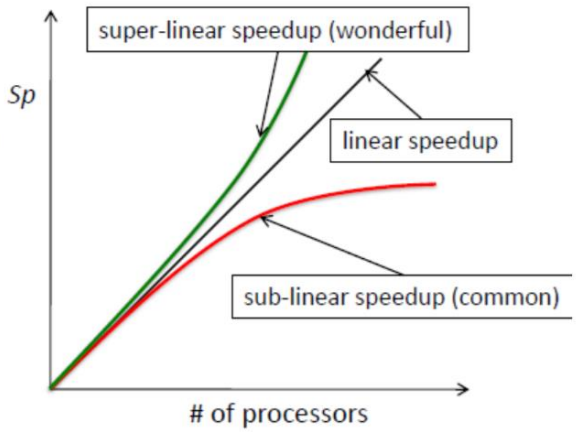
```
def read_data():
    """Read, Preprocess, TFIDF and Vectorizing of Data"""
    newsgroup_train = fetch_20newsgroups(subset = "train", remove = ('headers','footers','quotes'))
    text_df = pd.DataFrame([str(dat) for dat in newsgroup_train.data])
    text_df.columns = ["Text"]
    stemmer = PorterStemmer()
    text_df['stemmed'] = text_df["Text"].apply(lambda x: " ".join([stemmer.stem(str(y)) for y in x.split()]))
    vectorizer = TfidfVectorizer(stop_words = 'english')
    text_vector = vectorizer.fit_transform(text_df.stemmed.dropna())
    return text_vector,text_df

def  initial_centroid(text_vector,k):
    centroid = []
    centroid_pos = list(np.random.randint(text_vector.shape[0],size=k))
    centroid = text_vector[centroid_pos[0],:].todense()
    for i in centroid_pos[1:]:
        temp = np.array(text_vector[i,:].todense())
        centroid = np.concatenate((centroid,temp))
    return(centroid)

def euclidean_distance(text_vector,centroid):
    dist = np.sqrt(text_vector.dot(centroid.T))
    cluster = np.argmin(dist, axis=1)
    return(np.array(cluster))

def centroid_mean(cluster,text_vector):
    group_sum = np.array([])
    clusters = np.unique(cluster)
    for clust in clusters:
        group = scipy.sparse.vstack([text_vector[j] for j in range(text_vector.shape[0]) if cluster[j] == clust])
        group_sum = scipy.sparse.vstack((group_sum,np.array(group.sum(axis = 0))))
    group_sum = (group_sum.tocsr())[1:,]
    return group_sum
```

```
k_list = [20]
for k in k_list:
    text_vector = read_datacsv()
    centroid = initial_centroid(text_vector,k)
    centroid.shape
    cluster = list((np.zeros(text_vector.shape[0])))
    start = time.time()
    for i in range(10):
        old_cluster = cluster
        cluster = euclidean_distance(text_vector,centroid)
        summ = centroid_mean(cluster,text_vector)
        centroid = updated_centroid(cluster,text_vector)
        if (old_cluster == cluster).all():
            print("Converged")
            break
    print(k,time.time() - start)
```

Parallel Speedup & Efficiency

- Speedup $S_p = \frac{T_s}{T_p}$
  - P = # processes
  - Ts = Best serial execution time
  - Tp = execution time on P processes
  - Sp = Speedup on P processes
- Parallel Efficiency Ep

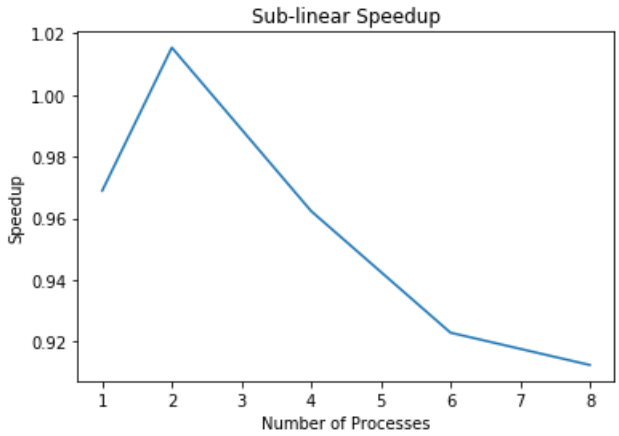
$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$



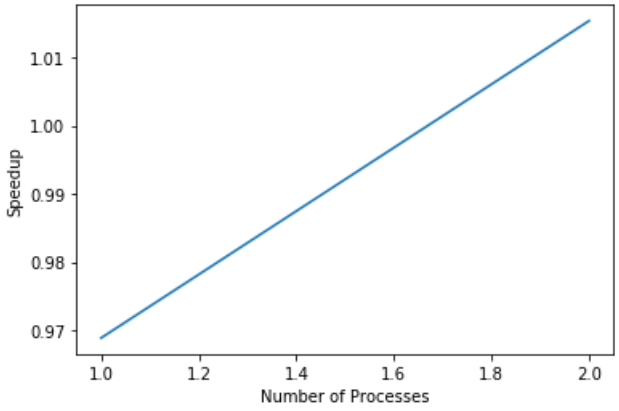
$$S_p = \frac{T_s}{T_p} \quad E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$

It can be seen that Efficiency depends on the speedup and the number of processes. Speedup is nothing but the best serial time divided by the corresponding processing time. The best serial timing is found to be 120.5 and the maximum run time of the workers are taken as Tp. Using this, we can calculate the speed up and efficiency for P = [1,2,4,6,8]. The results are tabulted as follows:

Process	Tp	Speedup	Efficiency
1	124.37	0.968	0.968
2	118.68	1.015	0.5075
4	125.22	0.962	0.241
6	130.57	0.923	0.154
8	132.07	0.912	0.11



For P = [1,2,4,6,8], it can be seen that the graph follows sub-linear speedup as shown in the lecture. This is because of the overriding of the processes. The algorithm was performed on a 2 core machine and hence after P being 2, the speedup curve starts to decrease showing that it is sub-linear speedup.



For P = [1,2], it can be seen that the process acheives a super-linear speed up because of non over-riding of the processes. Hence it is always good to look for the speed up graph and set the number of process to be used.