Distributed Data Analytics Lab

Exercise 3

Sebastián Pineda Arango Mtr. Nr. 246098

In this lab, we want to show how to perform K-Means over 20 news data set [1] in a distributed way and how the performance improvement looks like.

Traditionally, the K-Means algorithm has the following stages, given an initial number of centers K:

- Initializing centers
- Iterate until convergence:
    - Assign data points to one of the centers **(assignation**)
    - Compute the new centers as the mean of all point assigned to a given center **(center computation)**

Now, we want to extend this procedure to a distributed setting. To do so, we implement following new distributed stages:

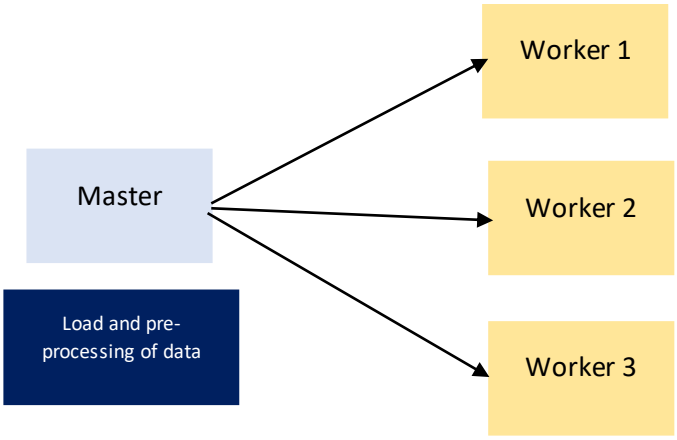
1) **Load and preprocessing of the data**
2) **Data distribution and centers initialization**
3) **Centers and stop flag sharing**
4) **Stop flag checking. Local centers, local count and local MSE calculation**
5) **Center aggregation and convergence checking**
6) **New center and stop flags distribution**
7) **If not all workers stopped, go back to step 4**

The distribution of data is don among different processes and the synchronization is performed through MPI point-to-point communication using MPI4PY library. In the following part, we explain step by step and show the python code to do.

Step-by-step explanation

1) **Load and preprocessing of the data**

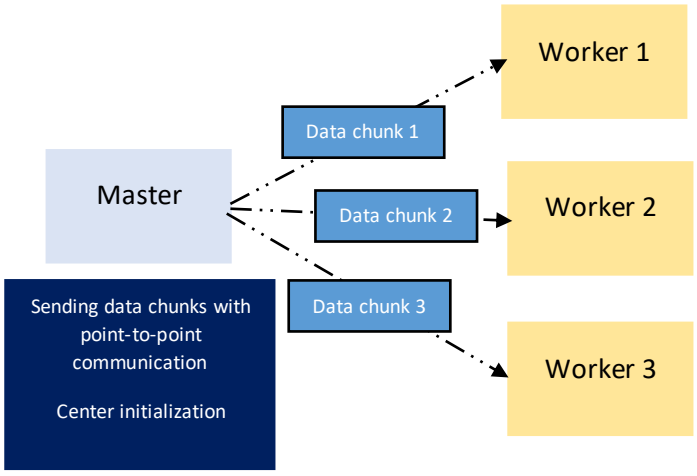
The **master** read the data, using scikit-learn function [1] and assuming that an external program has already preprocessed the data. Then, as the dimensionality is big (more than 170.000 columns), we reduce the dimensionality by using TruncatedSVD [2] to finally get 100 columns.



```
init = MPI.Wtime()

#Initializing importnate variables for control
stop_worker = [False]*num_workers
MSE_list = [0]*num_workers
MSE_diff = [1]*num_workers
MSE_history = []

#readint the dataset
newsgroups = fetch_20newsgroups(subset='all')

#transforming the dataset
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(newsgroups.data)

#reducing the dimensionality of the dataset
svd = TruncatedSVD(n_components=100, n_iter=10, random_state=42)
data = svd.fit_transform(X)
```

2) Data distribution and centers initialization

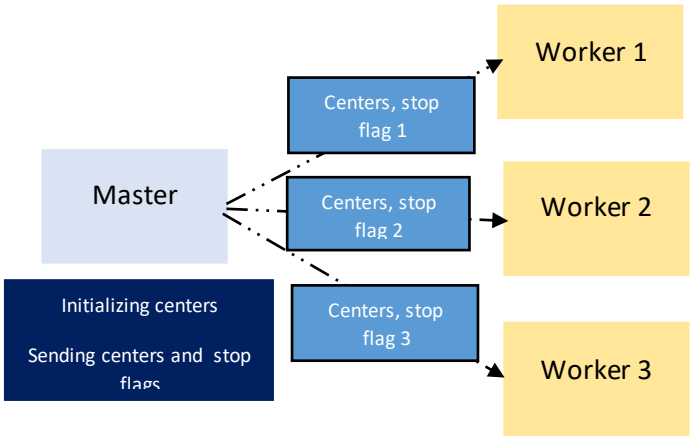
Once the dimensionality is reduce, the **master** distributes the data among all the **workers**. The **workers** hold every chunk of data for the rest K-Means computation. In this stage, the **master** also initializes the centers and some other important variables that are used in further steps. The following image and snippet code implement the above mentioned procedure.



```
#extracting information from the dataset
n_samples = data.shape[0] #number of samples
d = data.shape[1] #dimensionality of the problem

#random centers generation
idx = np.random.randint(0, n_samples, K)
C= data[idx, ]

#fraction of data for workers
frac = int(n_samples/(num_workers-1))
chunk_size = []

print("Time preprocessing: ", MPI.Wtime()-init)

i=0

init =  MPI.Wtime()
#sending data to workers
for i in range(1, num_workers-1):
    comm.send(data[((i-1)*frac):(i*frac)], dest=i)
    chunk_size.append(frac)
comm.send(data[(i*frac):], dest=(i+1))
chunk_size.append(data[(i*frac):].shape[0])
```

3) Centers and stop flag sharing

The **master** sends the centers and stop flags (all initialized to *false* in the first iteration) to every **worker**, as a dictionary. In further iterations, the **master** also checks if the all the **workers** have *locally converged*. A worker is considered to have *locally converged* if the *local distortion* difference between successive iterations is lower than a given threshold.



```
        #evaluating if all workers have converged
        if(sum(stop_worker)==(num_workers-1)):
            stop=True
            np.save("C",C)


        #sending center and "stop" flag to workers
        for w in range(1, num_workers):
            buf = {'centers':C,
                'stop': stop}
            comm.send(buf, dest=w)
```

4) Stop flag checking and computing local information.

Every worker receives the centers and the stop flag. If the stop flag is equal to *true*, it means that all the workers converged and therefore, the worker should stop iterating. In this step, every worker also calculates the *local information*: *local centers, local count* and *local MSE (or local distortion).*

To calculate the local centers, first we have to perform the assignation, it is, to assign every sample in the local chunk of data to one of the received clusters (in total K centers). Once the assignation step is completed, each worker computes the *local center* as:

$$\boldsymbol{C}_{i,j} = \sum_{x_n \in C_{ij}} x_n$$

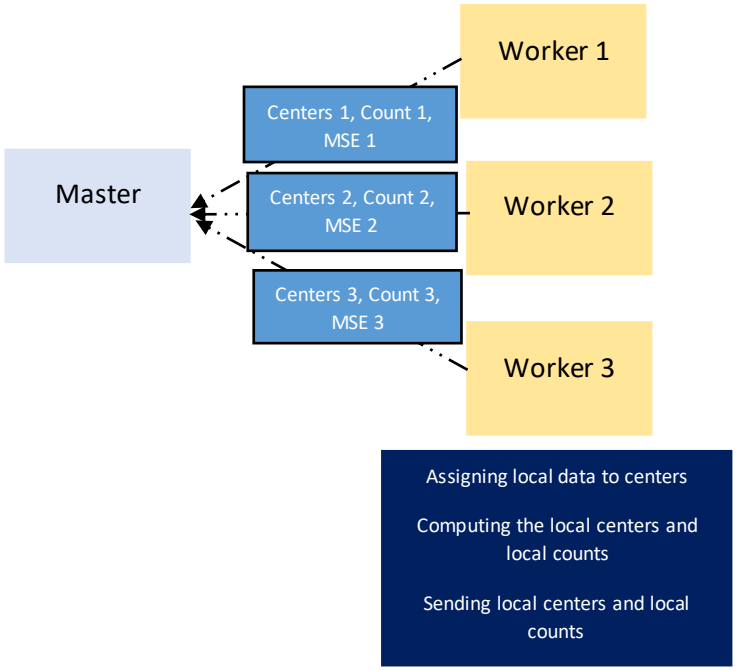
The local center $\boldsymbol{C}_{ij}$ corresponds to the local center$i$, $i \in \{1, \dots K\}$, and to the worker$j$. We calculate the partial sums, since the real centers is computed in the master using also the local count, where local count refers to the size of each local cluster. The *local count* is computed as:

$$c_{i,j} = \sum_{x_n \in C_{ij}} 1$$

The worker also computes the distortion of the cluster (sometimes worngly called MSE). The local distortion *(local MSE)* of the worker *j* can be computed as:

$$MSE_j = \sum_{i}^{K} \sum_{x_n \in C_{ij}} \left(x_n - C_{ij}\right)^2$$

After computing the local information, it is sent back to master.



The following code shows the implementation of this step. Some of the used functions are given as annex at the end of this document.

```
        buff = comm.recv( source=0) #recigin information of current centers
and stop flag
        C = buff["centers"]
        stop = buff["stop"]

        #computing local centers
        assignation = find_assignation(data,C) #finding the clusters of
every data point
        local_centers = find_center(data, assignation, K) #finding local
centers using local assignation
        local_MSE = find_MSE(data,C,assignation) #finding local MSE
        local_count = [assignation.count(a) for a in range(K)] #finging
local count

        #creating dictionary to send to the master
        buf = {
            "local_centers": local_centers,
            "local_MSE": local_MSE,
            "local_count": local_count
        }

        #sending dictionary to the master
        comm.send(buf, dest=0)
```

5) Center aggregation and convergence checking

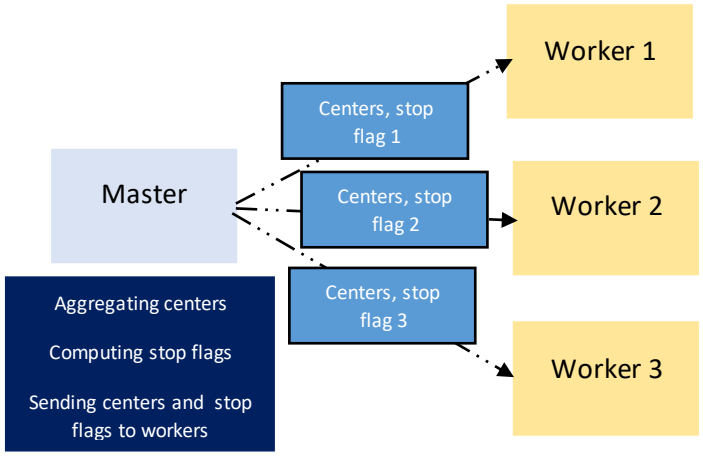
The master receives all the local information of the workers. Firstly, it uses the local centers and counts to compute the actual centroids as:

$$\boldsymbol{C}_i = \frac{\sum_j^{|W|} \boldsymbol{C}_{ij}}{\sum_j^{|W|} c_{ij}}$$

Where $|W|$ is the number of workers and $\boldsymbol{C}_i$ is the final cluster center$i \in \{1, \dots, K\}$.

As the master also receives the MSE of every worker, it is able to determine which worker has locally converged, by comparing the MSE of the last iteration. If the difference between the current MSE and the last MSE for a given worker falls below a specific threshold (*tolerance)*, then the master considers this worker has locally converged. The steps 3, 4 and 5 are repeated until all workers converge locally.

Once all the workers converge locally, the master sets the stop flag to *true* so that all the workers stop iterating.



The following code shows how to implement this idea.

```
#receiving partial centers and aggregating to final resutls
        centers = np.zeros(C.shape)
        count = np.zeros(K)
        for w in range(1, num_workers):

            #checking if the worker has converged
            if(abs(MSE_diff[w])<tol):
                stop_worker[w]=True

            #receiving information from worker
            buf = comm.recv(source=w)
```

```
            #reading MSE variables
            MSE_local = buf["local_MSE"]
            MSE_diff[w] = MSE_list[w]-MSE_local
            MSE_list[w] = MSE_local

            #agreggating centers sum and counts
            centers += buf["local_centers"]
            count += np.array(buf["local_count"])

        #calculating current MSE
        current_MSE = sum(MSE_list)/n_samples
        MSE_history.append(current_MSE)
        print("MSE at iter ",i, ": ", current_MSE)

        #computing centroids
        count[count==0] = 1 #to avoid zero division
        C = np.divide(centers.T, count).T #mean of centers: centroids
```

Speed-up tests

To analyze how the performance increases by using several processes we implement a strategy considering the following:

- We do not take into account the preprocessing time (which is in average around 30 secs.), since this is the same regardless the number of processors.
- We do not take into account the time spent by sending the initial data to all the workers. This time is showed in the following table, where we see that the time slightly increase as the number of processes increases (W refers to the number of workers).

	K=2	K=4	K=8	K=20
W=1	0,0371	0,0396	0,0421	0,0376
W=2	0,0422	0,0412	0,0409	0,0403
W=3	0,0418	0,0423	0,0471	0,0455
W=4	0,0421	0,0424	0,0421	0,0471
W=5	0,0455	0,0496	0,0427	0,0494

- We measure the average time per iteration (steps 3, 4, 5) for different number of processes and different values of K. The results are shown in the following table:

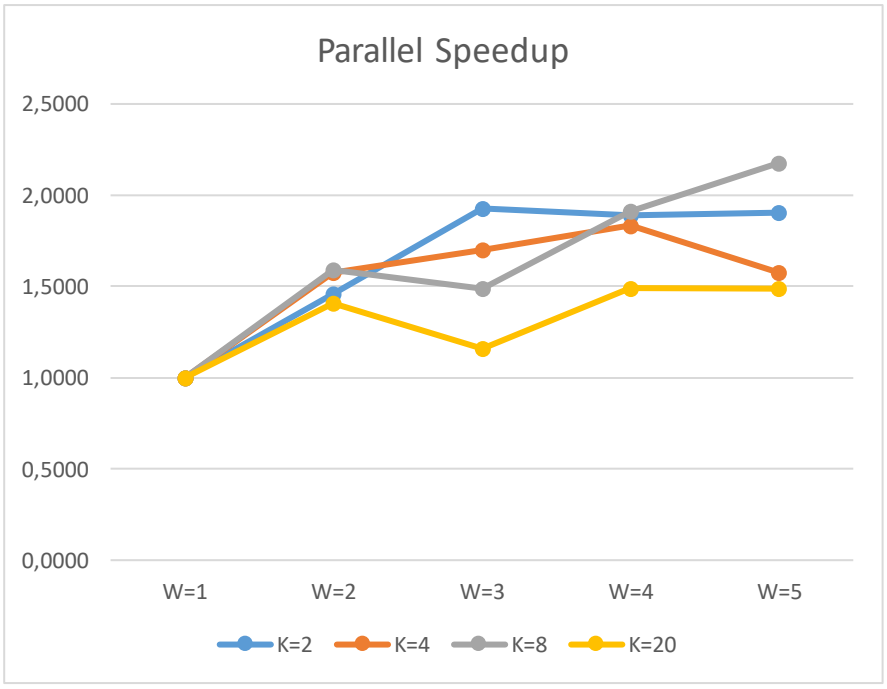
	K=2	K=4	K=8	K=20
W=1	1,0000	1,0000	1,0000	1,0000
W=2	1,4579	1,5734	1,5935	1,4073
W=3	1,9274	1,6978	1,4858	1,1611
W=4	1,8916	1,8345	1,9139	1,4911
W=5	1,9072	1,5782	2,1767	1,4864

- In order to compare the results, we plot the *speed-up* curves, which are explained in [3]. For that, we take the time spent by only one worker (parameter = 2 processes for master and worker) as the best sequential time (T_s). Then, we apply the following equation:

$$S_p = \frac{T_s}{T_p}$$

To obtain the speed given by a given number of processes. T_p, in this case, is the average time measured when running with p processes. Therefore, after applying the last equation to the last table, we obtain the following values (also plotted).

	K=2	K=4	K=8	K=20
W=1	1,0000	1,0000	1,0000	1,0000
W=2	1,4579	1,5734	1,5935	1,4073
W=3	1,9274	1,6978	1,4858	1,1611
W=4	1,8916	1,8345	1,9139	1,4911
W=5	1,9072	1,5782	2,1767	1,4864



In general, we can see a sublinear parallel speed-up. In some cases, there is a decrease in the speed-up curve, which can be due to some overload in the communication between master and nodes. We should remember, that although we measure the iteration time, there is indeed a

communication in every iteration between them, since they are exchanging local information and aggregated centers.

References

[1] Loading 20 news dataset from scikit-learn: https://scikitlearn.org/0.19/datasets/twenty_newsgroups.html

[2] Truncated SVD: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

[3] Parallel speedup: https://www.ismll.uni-hildesheim.de/lehre/bd-16s/exercises/bd-02-lec.pdf

Anex

In this annex, we add some functions used to calculate the centers.

```
def find_assignation(X,C):

    '''Finds the cluster represented by centers in C to which each
       sample of X belongs. Returns the number of the cluster for
       each sample.'''

    assignation=[]
    for i in range(X.shape[0]):
        #Look for the closes center for each data point

assignation.append(np.argmin(np.linalg.norm(np.subtract(X[i,].reshape(1,-
1),C),axis=1)))

    return assignation

def find_center(X, assignation, k):

    '''Returns the new centers according to the assignation. '''

    d = X.shape[1]
    C = np.zeros([k, d])

    for i in list(set(assignation)):

        #calculate the new center with assgination index given by i
        C[i,:] = X[np.array(assignation)==i,:].sum(axis=0)
```

```
    return C

def find_MSE(X,C,assignation):

    '''Returns the RMSE for a given clustering configuration: (data and
centers)'''

    q = []

    for i, a in enumerate(assignation):
        q.append(np.linalg.norm(X[i,]-C[a,])**2)

    return np.sum(q)
```