README.md

Distributed K means

Kmeans is a popular clustering method where the objective is to divide n observations in k clusters in such a way that the cluster distortion is minimized.

$$\sum_{x \in S_j} (x - \mu_j)^2$$

Following is a distributed varient of same algorithm implemented using message passing interface in python.

1 Kmeans general algorithm

The standard kmeans algorithm starts with random initialization of k cluster centers from the dataset. This is followed by two step iterative process.

- Assignment step - Here each data point is assigned to the nearest cluster center. Different distance measures can be used to calculate the distance.
- Update step/ Recenter - In this step the cluster center is updated with mean of all cluster members.

2 Problem setting

2.1 Assumptions

There are multiple ways to solve distributed K means and all of them differ based on worker node's setup. For calculating distributed K means it is assumed that all workers have access to distributed file storage (DFS) and all the required data for training is present on this storage.
Alternatively it is also possible for one worker(master) to have access to the data and is later shared with other workers. But this results into added communication overhead.

2.2 Data description

This algorithm operates on top of TFIDF vectors (unigrams, bigrams and trigrams) calculated using Distributed TF-IDF algorithm for Twenty News Group dataset. The output of Distributed TFIDF vector was a directory containing multiple TFIDF vector(One for each document) in json format and a metaData.json having vocab length and dictionary mapping. This output will be consumed by distributed kmeans algorithm.
The output of distributed TFIDF includes :-

- TDIDF directory containing the TfIdf vectors as shown below

```
foo@bar:~$ ls
metadata.json  TFIDF
foo@bar:~$ ll TFIDF | head -4
total 179172
drwxrwxr-x 2 foo foo 376832 May  2 12:29 ./
drwxrwxr-x 3 foo foo   4096 May  2 12:30 ../
-rw-rw-r-- 1 foo foo   6070 May  2 12:29 10000
-rw-rw-r-- 1 foo foo   5187 May  2 12:29 10001
-rw-rw-r-- 1 foo foo   6277 May  2 12:29 10002
```

- Metadata json file named metadata.json

```
foo@bar:~$ less metadata.json
{
"vocabLength":213928
}
```

2.3 Expected results

Distributed kmeans should be able to calculate K cluster centroids with minimal distortion and store these centroids along with the cluster members.

A non distributed variant of Kmeans algorithm trained on the same dataset is available here. This variant took **1:56** minutes to train and converge and this duration will act as benchmark for distributed Kmeans algorithm. Kmeans algorithm with proper parallelization should be able reduce the duration for convergence.

3 Utility functions

These are some of the helper functions that will be used very frequently.

```
def buildInput(basePath):
        pathList = []
        for file in os.listdir(basePath):
                pathList.append(join(basePath, file))
        return pathList

def writeToFile(path, text):
        with open(path, 'w') as outputFile:
                outputFile.write(text)

def readJSONFile(path):
        outputJSON = None
        with open(path, 'r') as inputFile:
                outputJSON = json.load(inputFile)
        return outputJSON
```

The first procedure, buildInput builds the list of paths for all files present in the provided directory. Second procedure writeToFile stores the provided text to the file. Third procedure, readJSONFile reads json output generated from distributed TFIDF vectorizer.

4 Kmeans procedure

First we will look for non distributed variant of Kmeans and then we will scale it for distributed environment.

4.1 Data matrix creation

Usage of matrix speeds up the execution when compared to list or dictionary datastructure. This is true for Kmeans as well. To speed up the execution of Kmeans we need the TFIDF vectors in matrix format. These TFIDF vectors are mostly sparse and we need to preserve this sparsity. Also most of the operations that will be performed for calculation of kmeans will include slicing the rows of the matrix rather than slicing of columns making compressed row matrix a natural choice.

```
def jsonToCSRMatrix(pathList, dimensions):
        data = []
        rows = []
        columns = []
        for i in range(0, dimensions[0]):
                jsonDoc = readJSONFile(pathList[i])
                for key, value in jsonDoc.items():
                        """Convert the data to float16 to reduce memory footprint"""
                        data.append(np.float16(value))
                        columns.append(int(key))
                        rows.append(i)
        matrix = csr_matrix((data, (rows, columns)), shape=dimensions)
        return matrix
```

All the data from distributed TFIDF algorithm is in json format. The above method takes list of all the TFIDF vectors in json format and outputs a CSR matrix.

4.2 Cluster center generation

The first task for Kmeans algorithm is to assign K data points as initial cluster centroids. These can be generated by selecting random k indices from the generated CSR matrix. But taking all cluster centers randomly might result in slower convergence. This is because it is possible for two data points that are closer to each other assigned as different cluster centers during random initialization. To speed up the process of convergence, only one cluster center is selected randomly and rest K-1 cluster points are selected as farthest data points from existing cluster centers.

```
"""Will return distance from all clusters"""
def getDistanceFromCluster(clusterCenters, dataMatrix):
        distanceList = []

        """Calculate each cluster centroid
        distance from all data points in data matrix"""
        for clusterCenter in clusterCenters:
                distanceList.append(euclidean_distances(clusterCenter, dataMatrix).flatten())
        return np.array(distanceList)

def getFarthestPoint(clusterCenters, dataMatrix):
        clusterDistance = getDistanceFromCluster(clusterCenters, dataMatrix)
        clusterDistance = clusterDistance.sum(axis = 0)
        """Find the index with max distance"""
```

```
        index = np.where(clusterDistance == max(clusterDistance))[0][0]
        return dataMatrix.getrow(index), max(clusterDistance)
```

The above methods helps in finding the farthest point and its distance from existing clusters(Cluster centers are added cumulatively). This method expects a CSR matrix and clusterCenters as list of csr vectors.

4.3 Cluster assignment

Assigning the data points to clusters is the first step of the standard two step kmeans process.

```
def assignClusters(clusterCenters, dataMatrix):
        distortion = 0
        clusterMembership = {}
        """Get distance of all points from all clusters."""
        clusterDistance = getDistanceFromCluster(clusterCenters, dataMatrix)
        """Now each row is array having list of distances from cluster centers"""
        clusterDistance = clusterDistance.T

        for dataPointIndex, dataPointDistance in enumerate(clusterDistance):
                minDistance = min(dataPointDistance)
                clusterIndex = np.where(dataPointDistance == minDistance)[0][0]
                distortion = distortion + minDistance
                if clusterIndex not in clusterMembership:
                        clusterMembership[clusterIndex] = []
                """The datapoint index is local to the processor"""
                clusterMembership[clusterIndex].append(dataPointIndex)
        return clusterMembership, distortion
```

The above method assigns each point in the data matrix to a cluster from provided cluster centers. Additionally this method also returns the distortion for the calculated cluster configuration. This method calculates the Euclidian distance between data point and cluster centers and assigns the data point to the closest cluster. The cluster membership is a dictionary containing the cluster index and list of indices of data point belonging to this cluster.

4.4 Cluster mean calculation

Once the cluster membership is assigned, the second step is to update the centroids for all clusters with updated cluster membership configuration.

```
def recenter(clusterMembership, dataMatrix):
        newCenters = {}
        for clusterIndex in clusterMembership:
                subMatrix = dataMatrix[clusterMembership[clusterIndex]]
                """Tuple of cumulative sum of
                vectors of cluster members and total count of cluster members"""
                newCenters[clusterIndex] = [ csr_matrix(csr_matrix.sum(subMatrix, axis = 0)),
                                                        len(clusterMembership[clusterIndex]) ]
        return newCenters
```

The above procedure calculates new cluster centroids. However the average is not yet calculated. Instead a tuple containing the summed up cluster centroid and total count of data points is returned. For non distributed setting the division of cumulative sum by count will result into average i.e the new cluster centers.

4.5 Data point lookup

```
def localClusterLookup(clusterMembership, dataPointLookup):
        """For Index to data point mapping"""
        clusterMembers = {}
        for clusterIndex in clusterMembership:
                clusterMembers[clusterIndex] = [ dataPointLookup[dataPointIndex]
                                for dataPointIndex in clusterMembership[clusterIndex] ]
        return clusterMembers
```

There is more method required for kmeans. The output of cluster assignment is not the actual data point but an index to the data point in the data matrix. This is done for efficiency as the two steps of cluster membership assignment and calculating the cluster centroid are done again and again until the k means algorithm converges. Having index instead of value enables slicing and speeds up entire process. The lookup of converting indices to actual data point (name) is only required once the algorithm converges so that the data cluster membership can be persisted. For this a dataPoint lookup list is maintained where each index points to the name of file on the file system. The method localClusterLookup will replace the indices in membership dictionary with actual datapoint based upon the dataPointLookup list. It is imperative that both the list index and matrix row index point to same data point.

5 Wrapping all methods for parallel execution

Now the methods shown above are targeted to solve kmeans for non distributed environment. However it is possible to extend same kmeans algorithm for distributed setting. Following section will make use of above kmeans procedures and scale them for distributed kmeans algorithm.

5.1 Distributed K means helper methods

For scaling the k means algorithm in distributed enviornment some additional helper methods are required.

```
def recenterGlobalClusterCenters(globalClusterMembership):
        """This method will calculate global cluster centroids
        using previously calculated local cluster centroid"""
        cummulativeCenters = {}
        cummulativeCount = {}
        clusterCenters = []

        for clusterMembership in globalClusterMembership:
                for clusterIndex in clusterMembership:
                        if clusterIndex in newCenters:
                                cummulativeCenters[clusterIndex] = cummulativeCenters[clusterIndex]
                                                + clusterMembership[clusterIndex][0]
                                cummulativeCount[clusterIndex] = cummulativeCount[clusterIndex]
                                                + clusterMembership[clusterIndex][1]
                        else:
                                cummulativeCount[clusterIndex] = clusterMembership[clusterIndex][1]
                                cummulativeCenters[clusterIndex] = clusterMembership[clusterIndex][0]
        """Calculate global cluster mean and
        store it in new list"""
        for i in range(0, len(cummulativeCenters)):
                clusterCenters.append(cummulativeCenters[i]/cummulativeCount[i])
        return clusterCenters
```

Each worker will execute recenter procedure and will have a cumulative sum as cluster centroid. This cumulative sum needs to be aggregated for all workers in order to calculate actual cluster centers. This is done using above procedure. The above method is calculated by root node once it has accumulated the results from recenter procedure from all the worker nodes. The output of this is updated clusters which is later shared with all the other workers.

```
def globalClusterLookup(localClusterMembers, clusterCenters):
        globalClusterMembers = {}
        for localClusterMember in localClusterMembers:
                for clusterIndex in localClusterMember:
                        if str(clusterIndex) not in globalClusterMembers:
                                globalClusterMembers[str(clusterIndex)] = []
                        globalClusterMembers[str(clusterIndex)].extend(localClusterMember[clusterIndex])
        return globalClusterMembers
```

Second method globalClusterLookup wraps together output of localClusterLookup of all clusters on root. This method provides final cluster membership dictionary which can be persisted on disk. This method is called only once the kmeans has converged.

5.2 Data distribution

As stated before it is assumed that the data is available on DFS and can be accessed by all the workers. First task to run distributed kmeans is to divide this data among all workers.

```
"""Command line arguments"""
inputPath = sys.argv[1]
outputPath = sys.argv[2]
k = int(sys.argv[3])
maxEpochs = int(sys.argv[4])

"""MPI variables"""
comm = MPI.COMM_WORLD
processorCount = comm.Get_size()
currentRank = comm.Get_rank()
root = 0


processorTasks = None
dataPoints = {}
clusterCenters = []

vocabSize = None

"""Build the list of inputs"""
if currentRank == root:
        startTime = MPI.Wtime()
        pathList = buildInput(join(inputPath, "TFIDF"))
```

```
        if not os.path.exists(outputPath):
                os.makedirs(outputPath)

        shuffle(pathList)
        metaData = readJSONFile(join(inputPath, "metadata.json"))
        vocabSize = metaData["vocabLength"]
        vocabSize = int(vocabSize)
        processorTasks = np.array_split(pathList, processorCount)

vocabSize = comm.bcast(vocabSize, root)
processorTasks = comm.scatter(processorTasks, root)
```

First the root node will call buildInput method. This will generate the list of all input file paths. Also root processor will create any additional folders required. This list will be shuffled and broken in n parts where n is total no of processors. Additionally the total vocabulary count is read from metadata.json. Root worker will broadcast the vocabulary count to all other workers and distribute the output of build input equally or nearly equally. This will distribute the required data among all the workers.

5.3 Parallel execution

Following is the code that will be executed in parallel on all the workers.

5.3.1 Building data matrix

For reducing the time taken by kmeans it is necessary to build sparse csr matrix. Each processor will build the sparse csr matrix based upon the processor task assigned to them.

```
"""For faster computation all processors will build a CSR matrix"""
dataMatrix = jsonToCSRMatrix(processorTasks, (len(processorTasks), vocabSize))

"""Used for storing the name of file. Useful for identifying which item belongs to which cluster"""
dataPointLookup = []
for processorTask in processorTasks:
        dataPointLookup.append(basename(processorTask))
```

5.3.2 Initializing cluster centers.

One of the fastest way to initialize the k cluster centroids is to take k random points from dataset. However as discussed before this will result in slower convergence. For faster convergence first point should be selected randomly while others should be selected based upon their distance from cluster centers.

```
"""First center will be selected from random from data available with root"""
if currentRank == root:
        randomIndex = random.randint(0, dataMatrix.get_shape()[0])
        clusterCenters.append(dataMatrix.getrow(randomIndex))
        print("Cluster center-1 calculated")

clusterCenters = comm.bcast(clusterCenters, root)

"""Next clusters will be selected as one having farthest distance"""
for i in range(1, k):
        farthestPoint, farthestDistance = getFarthestPoint(clusterCenters, dataMatrix)
        clusterCenterCandidates = comm.gather([farthestDistance, farthestPoint], root)

        if currentRank == root:
                maxDistance = 0
                maxDistanceDataPoint = None

                for clusterCenterCandidate in clusterCenterCandidates:
                        if maxDistance < clusterCenterCandidate[0]:
                                maxDistance = clusterCenterCandidate[0]
                                maxDistanceDataPoint = clusterCenterCandidate[1]
                clusterCenters.append(maxDistanceDataPoint)
                print("Cluster center-"+str(i+1)+" calculated")

        """BroadCast the updated cluster centers to all"""
        clusterCenters = comm.bcast(clusterCenters, root)
```

The above code selects the first cluster centroid randomly from the root node and then it is sent to all worker using broadcast method of MPI. Then the next cluster centers are selected cumulative as farthest point from existing cluster centers. Each worker will nominate a candidate and root will select cluster center among these nominated centroid candidates. Iteratively newer candidates will be added in this clusters using getFarthestPoint procedure seen in section 4.

5.3.3 Iterative kmeans procedure

The following code block describes the iterative two step procedure of kmeans

```
"""Start clustering process"""
currentEpoch = 0
prevClusterMembership = {}
distortions = []
while(currentEpoch < maxEpochs):

        if currentRank == root:
                print("Epoch-"+str(currentEpoch+1))

        """Step-1 Assign data point to cluster centers"""
        clusterMembership, distortion = assignClusters(clusterCenters, dataMatrix)
        totalDistortion = comm.gather(distortion, root)
        if currentRank == root:
                totalDistortion = np.sum(totalDistortion)
                print("Distortion is "+str(totalDistortion))
                distortions.append(totalDistortion)

        """Check if clusters changed locally"""
        if prevClusterMembership == clusterMembership:
                clusterMembershipChanged = False
        else:
                clusterMembershipChanged = True

        prevClusterMembership = clusterMembership

        """Check if clusters are changed globally"""
        comm.barrier()
        globalClusterMembershipChanged = comm.allgather(clusterMembershipChanged)
        convergenceStatus = True
        for globalUpdateStatus in globalClusterMembershipChanged:
                if globalUpdateStatus == True:
                        """Atleast one processor cluster members changed"""
                        convergenceStatus = False


        if convergenceStatus == True:
                """No change in cluster membership on any worker.
                Kmeans has converged"""
                if currentRank == root:
                        print("Kmeans converged")
                break

        """No convergence calculate new cluster center"""
        """Step-2 Calculate new cluster centroids"""
        newCenters = recenter(clusterMembership, dataMatrix)
        newCenters = comm.gather(newCenters, root)
        if currentRank == root:
                clusterCenters = recenterGlobalClusterCenters(newCenters)

        comm.barrier()
        clusterCenters = comm.bcast(clusterCenters, root)
        currentEpoch+=1

"""End of while loop"""
```

The above code block is executed until maximum provided epochs are performed or the kmeans algorithm converges.

First the cluster assignment is calculated for available cluster centers using assignClusters procedure seen in section 4. Then test for convergence is performed. If kmeans has converged, no need to recalculate the cluster centers as they will be unchanged.

```
        """Check if clusters changed locally"""
        if prevClusterMembership == clusterMembership:
                clusterMembershipChanged = False
        else:
                clusterMembershipChanged = True

        prevClusterMembership = clusterMembership

        """Check if clusters are changed globally"""
        comm.barrier()
        globalClusterMembershipChanged = comm.allgather(clusterMembershipChanged)
        convergenceStatus = True
        for globalUpdateStatus in globalClusterMembershipChanged:
                if globalUpdateStatus == True:
                        """Atleast one processor cluster members changed"""
                        convergenceStatus = False

        if convergenceStatus == True:
                if currentRank == root:
                        print("Kmeans converged")
                break
```

As shown in above snippet first we check if the cluster membership has changed. We assign appropriate value to clusterMembershipChanged variable. However there is a catch, this variable only describes the current worker's convergence. Kmeans will converge when all the workers cluster membership remains unchanged.

For this value of clusterMembershipChanged is gathered on all workers. If all workers cluster membership is unchanged, then the kmeans has converged and the outer loops stops execution.

However if there is at least one worker with non converged cluster membership then new cluster means are calculated, accumulated on root. Root will calculate new centroids and broadcast the data to all workers. For calculating the new centroids recenter and recenterGlobalClusterCenters seen in section 4 and 5.1 are used.

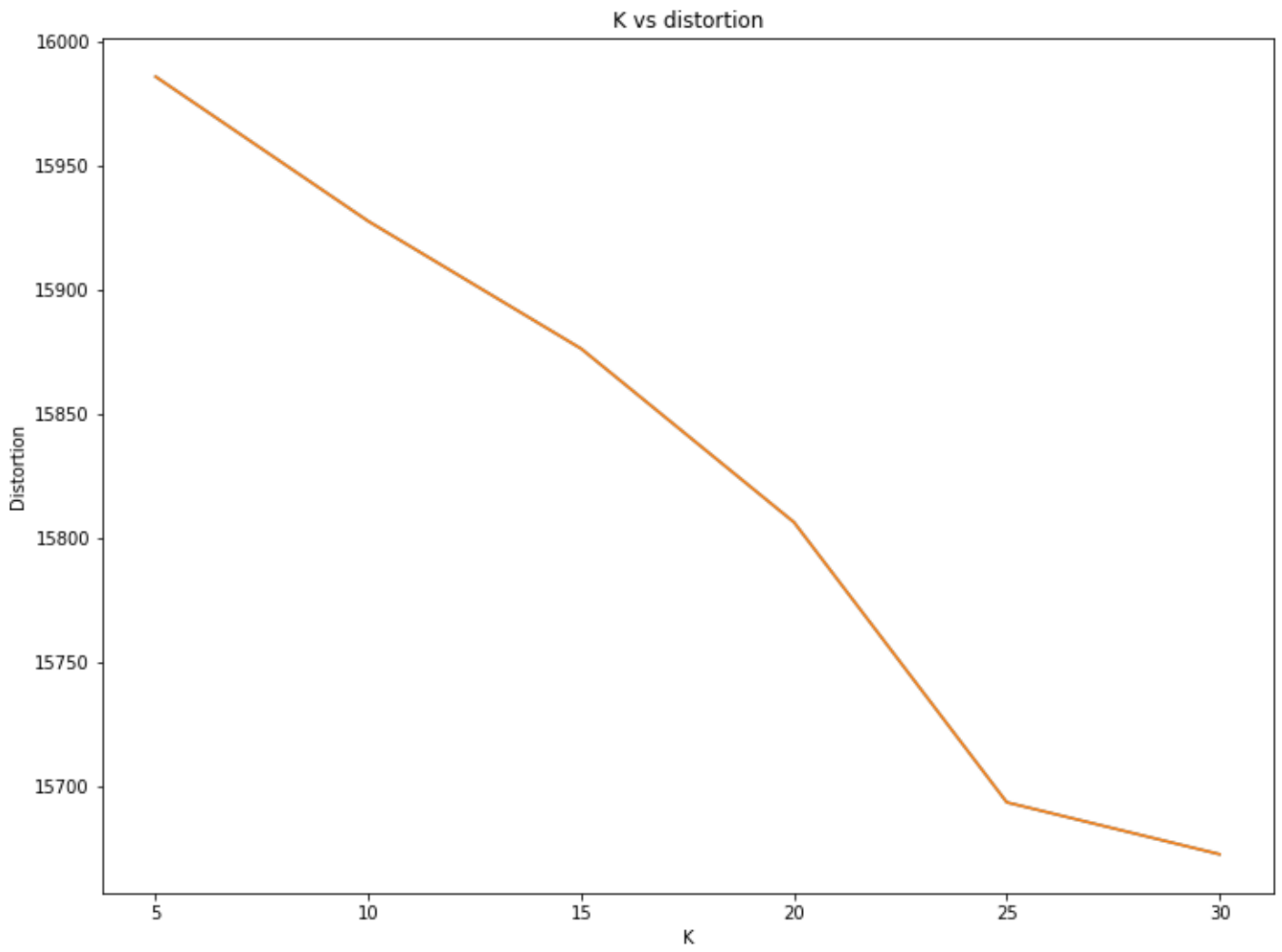
```
"""No convergence calculate new cluster center"""
   newCenters = recenter(clusterMembership, dataMatrix)
   newCenters = comm.gather(newCenters, root)
   if currentRank == root:
       clusterCenters = recenterGlobalClusterCenters(newCenters)

   comm.barrier()
   clusterCenters = comm.bcast(clusterCenters, root)
   currentEpoch+=1
```

5.4 Storing output

```
if currentRank == root:
        """Merge local processors lookup """
        globalClusterMembers = globalClusterLookup(globalClusterMembers, clusterCenters)
        writeToFile(join(outputPath, "clusterMembers.json"), json.dumps(globalClusterMembers, indent=2))

        for index, clusterCenter in enumerate(clusterCenters):
                pickle.dump(clusterCenter, open(join(outputPath, str(index)+".pickle"), 'wb'))

        """Save plots"""
        plt.plot(range(0, len(distortions)), distortions)
        plt.plot(range(0, len(distortions)), distortions, "rx")
        plt.xticks(range(0, len(distortions)))
        plt.title("Iteration vs distortion")
        plt.xlabel("Iteration")
        plt.ylabel("Distortion")
        plt.savefig(join(outputPath, "iterationsVSdistortion.png"))
        plt.clf()
        distortions = distortions[1:]
        plt.plot(range(0, len(distortions)), distortions)
        plt.plot(range(0, len(distortions)), distortions, "rx")
        plt.xticks(range(0, len(distortions)))
        plt.title("Iteration vs distortion")
        plt.xlabel("Iteration")
        plt.ylabel("Distortion")
        plt.savefig(join(outputPath, "iterationsVSdistortion2.png"))

        endTime = MPI.Wtime()
```

Finally all the output with cluster membership dictionary and cluster centers are stored to DFS.

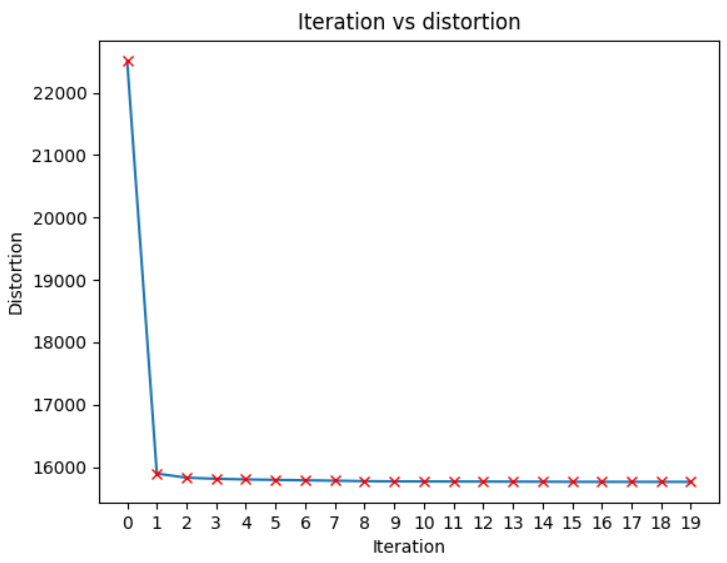
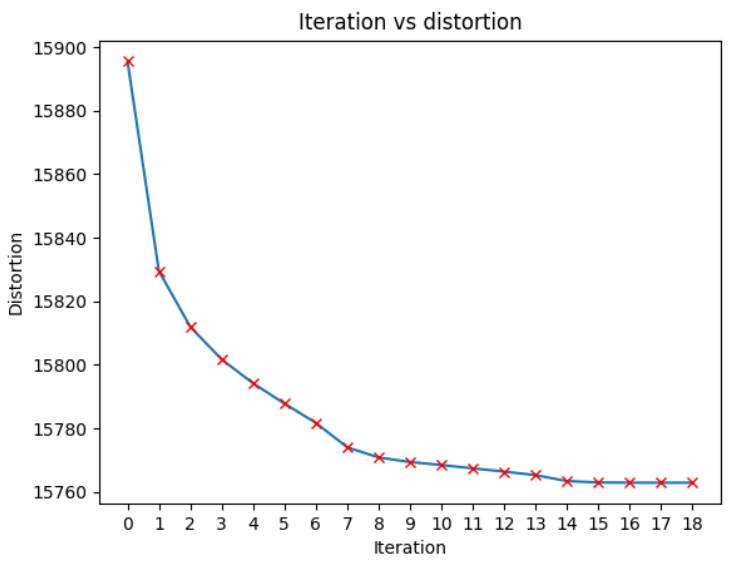
6 Optimal K Selection



The above graph shows the cluster distortion vs K. Using elbow method we can say that optimal no of clusters are somewhere between k = 20 and k = 25. We will select k = 20 for all further calculations.

7 Performance

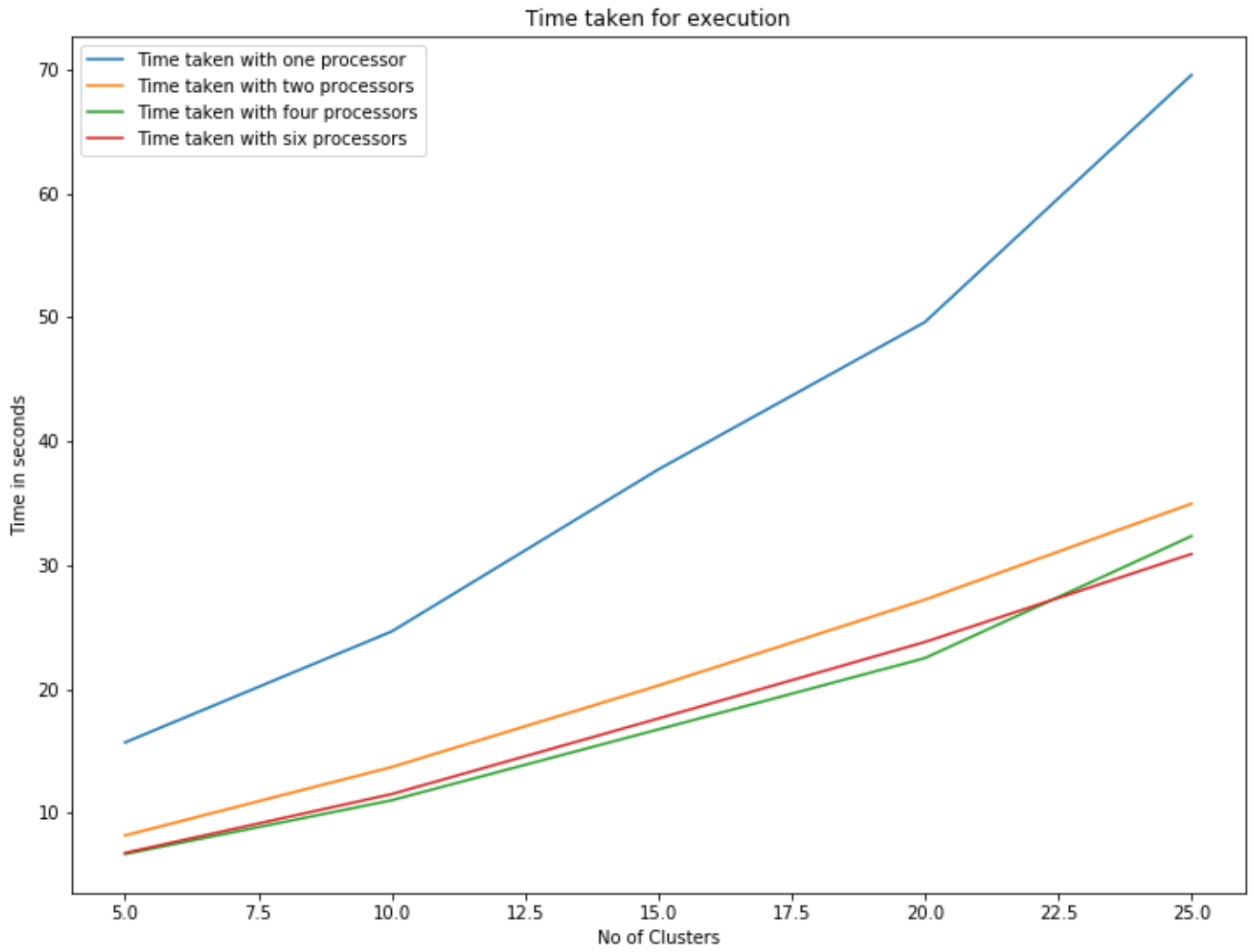
7.1 Epochs vs Distortion

Iterations (1-20)	Iterations (2-20)
	

The above figures show the distortion with K = 20. The first figure captures all 20 iterations, while the second figure captures iteration 2 to 20. There is a significant drop in cluster distortion from first to second iteration. This is expected. In second figure it can be seen that the distortion becomes constant after 16th iteration. Essentially we can say that kmeans converges in 16 iterations.

7.2 Execution time

Following section describes time taken to execute the distributed kmeans for 10 epochs.

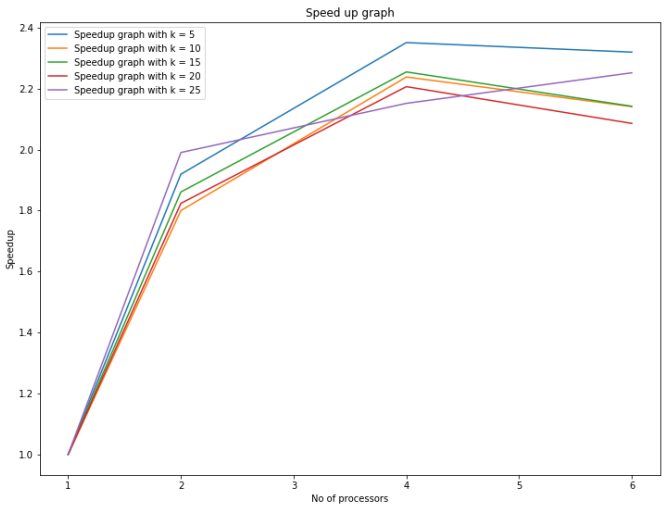
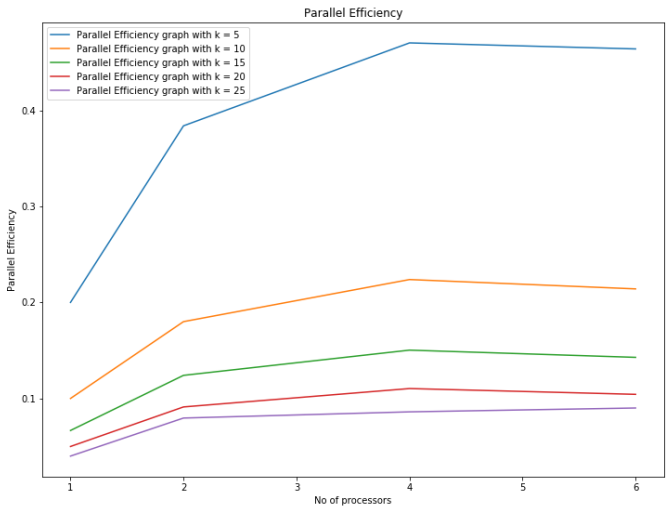


Cluster Count	Time taken in seconds			
	One processor	Two processors	Four processors	Six processors
5	15.68 +/-0.4	8.17 +/- 0.56	6.67 +/- 0.21	6.76 +/- 0.33
10	24.64 +/-0.79	13.69 +/- 0.38	11.01 +/- 0.11	11.51 +/-0.4
15	37.72 +/-0.17	20.27 +/- 0.77	16.73 +/- 0.87	17.61 +/-0.8
20	49.62 +/-0.67	27.21 +/- 0.45	22.49 +/- 0.20	23.79 +/-0.79
25	69.58 +/- 0.41	34.96 +/-0.56	32.34 +/- 0.47	30.90 +/-0.8

The above table describes the outcome of parallelizing the kmeans code.

The previous benchmark of non distributed kmeans was 1:56 minutes. Time taken for distributed kmeans to converge with 4 workers and K = 20 is 51.16 seconds. It converged in 26 iterations essentially performing better than previous baseline.

7.3 Speedup and parallel efficiency

Speedup	Parallel Efficiency
	

By looking at the speed up graph we can say that for all K (5, 10, 15, 20, 25) there is a super linear speedup from one worker to two workers. However on increasing processors from two to three and from three to four there is linear speedup for all values of k. Now increasing the no of processors from four to six there is decrease in performance for k = 5, 10, 15 and 20. But for K = 25 there is linear increase.

8 Execution

8.1 Execution script

To run the code call the following script

```
foo@bar:~$ sh run.sh {ProcessorCount} {InputPath} {OutputPath} {ClusterCount} {MaxIterations}
```

Here the input path is output path of distributed TFIDF algorithm.

The abouve shell script contains following snippet.

```
if test "$#" -ne 5; then
    echo "Illegal number of parameters"
else
    mpiexec -n $1 python3 kmeans_sparse.py $2 $3 $4 $5
fi
```

8.2 Output

The output of distributed kmeans is a json file containing cluster center id and associated data points.

```
{"0":[1, 2, 4],
"1":[3, 5, 6]
}
```

Furthermore there are k csr vectors stored in pickle format.

```
foo@bar:~$ ls *.pickle
0.pickle   11.pickle  13.pickle  15.pickle  17.pickle  19.pickle  20.pickle  22.pickle  24.pickle  26.pickl
10.pickle  12.pickle  14.pickle  16.pickle  18.pickle  1.pickle   21.pickle  23.pickle  25.pickle  27.pickl
```