Parallel K-Means Data Clustering In Python

Review 2 Report

Submitted by

Prateek Arora [15BCE0058]

Abhinav Batra [15BCE0576]

Amar Nath Dutt Sharma [15BCE0953]

Prepared For:

Parallel and Distributed Computing

Slot : D2

Submitted To

MANOOV R

VIT®

UNIVERSITY

(Estd. u/s 3 of UGC Act 1956)

VELLORE ■ CHENNAI

www.vit.ac.in

1. Abstract:

Clustering is one of the most popular methods for data analysis, which is prevalent in many disciplines such as image segmentation, bioinformatics, pattern recognition and statistics etc. The most popular and simplest clustering algorithm is K-means because of its easy implementation, simplicity, efficiency and empirical success. However, the real-world applications produce huge volumes of data, thus, how to efficiently handle of these data in an important mining task has been a challenging and significant issue. In our project we tried to implement Parallel K- means clustering in python.

2. Introduction :

Clustering is the process of partitioning or grouping a given set of patterns into disjoint clusters. This is one such that patterns in the same cluster are alike and patterns belonging to two different clusters are different. Clustering is the grouping of similar objects and a clustering of a set is a partition of its elements that is chosen to minimize some measure of dissimilarity. Clustering algorithms are often useful in applications in various fields such as visualization, pattern recognition, learning theory, computer graphics, neural networks, AI,

and statistics. Practical applications of clustering include unsupervised classification and taxonomy generation, nearest neighbour searching, time series analysis, text analysis and navigation. Data clustering is a frequently used and is a useful tool in the data mining. There are a variety of data clustering algorithms, which are generally classified, into two categories: hierarchical algorithms and partitioned algorithms. A hierarchical algorithm creates a hierarchical decomposition of the given data set forming a tree, which splits the data set recursively into smaller sub-trees. A partitioned clustering algorithm obtains a single partition of the data instead of a clustering structure like a tree produced by the hierarchical technique.

3. Parallel K Means :

Parallel k-means has been studied by [Dhillon], [Xu], [Stoffle] previously for very large databases. There has been some study for studying the advantages of the parallelism in the k means procedure. The speedup and scale-up variation with respect to number of documents (vectors), the K of the k-means and the dimension of each of the documents has been studied [Dhillon]. Parallelization is basically studied for performance advantages mainly the data partitioning and parallel processing. Inherent to classical k-means algorithm is intrinsic parallelism. The most intensive calculation to

occur is the calculation of distances, and it would require a total of k distance computations where d is the number of documents and k is the number of clusters being created. On the parallel system, the main idea is that of splitting the dataset among processes for faster computation of vector-cluster membership for each partition generated. However it is important to note that the communication cost between the processors is significant (for small datasets communication computation). So, for an efficient parallel algorithm, it is necessary to minimize the communication between nodes. The centroid lists consists of k vectors of length d each and in practical applications, it will be much smaller compared to the actual data set, which consists of n vectors each of length d. Upon beginning the clustering, each process will receive the entire centroid list, but only the process with id as the “root” will compute the initial centroids and then broadcast this selected k initial centroids to all other processes. Each process takes care of only the part of the dataset. Thus each process is responsible for (n / number of processes) vectors rather than the entire set. Each process will compute the distances for only part of vectors to each of the centroids, which are all locally stored on each process. A series of assignments are generated mapping vectors (documents) to clusters. Each process then gathers the sum of all vectors allocated to a given cluster and computes the mean of the vectors assigned to a particular cluster. This is repeated for every cluster and a new set of centroids is available on every process, and the vectors can be reassigned with respect to the newly calculated centroids.

To compute the quality of the given partition, each process can compute the mean squared error for the portion of dataset over which it is operating. As these values are calculated, the process will sum its local mean squared error values. A simple reduction of these values among all processes will determine the overall quality of the cluster.

- Root calculates the initial means
- Replicate the k centroids
- Each processor computes distance of each local document vector to the centroids
- Assign points to closest centroid and compute local MSE (Mean Squared Error)
- Perform reduction for global centroids and global MSE value.

3.Algorithm:

N: *number of data objects*
K: *number of clusters*

objects[N]: *array of data objects*
clusters[K]: *array of cluster centers*
membership[N]: *array of object memberships*

```
kmeans_clustering( )
1   while δ/N > threshold
2      δ ← 0
3      for i ← 0 to N-1
4         for j ← 0 to K-1
5            distance ← | objects[i] - clusters[j] |
6            if distance < dmin
7               dmin ← distance
8               n ← j
9         if membership[i] ≠ n
10           δ ← δ + 1
11           membership[i] ← n
12        new_clusters[n] ← new_clusters[n] + objects[i]
13        new_cluster_size[n] ← new_cluster_size[n] + 1
14     for j ← 0 to K-1
15        clusters[j][*] ← new_clusters[j][*] / new_cluster_size[j]
16        new_clusters[j][*] ← 0
17        new_cluster_size[j] ← 0
```

4.Code [In Python]:

(Any line written after # is comment)

```
import numpy as np
from IPython import parallel
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import timeit
import warnings



class KMeans:
    def __init__(self, data, k):
        self.data = data
        self.k = k

    def cluster(self):
        return self._lloyds_iterations()

    def _initial_centroids(self):

        # get the initial set of centroids
        # get k random numbers between 0 and the number of rows in the data set

        centroid_indexes = np.random.choice(range(self.data.shape[0]), self.k,
replace=False)

        # get the corresponding data points
        return self.data[centroid_indexes, :]

    def _lloyds_iterations(self):
        #warnings.simplefilter("error")
        centroids = self._initial_centroids()
        #print('Initial Centroids:', centroids)

        stabilized = False

        j_values = []
        iterations = 0
```

```
        while (not stabilized) and (iterations < 1000):
            print ('iteration counter: ', iterations)
            try:
                # find the Euclidean distance between a center and a data point
                # now as a result of broadcasting, both array sizes will be n x k x m


                data_ex = self.data[:, np.newaxis, :]
                euclidean_dist = (data_ex - centroids) ** 2


                # now take the summation of all distances along the 3rd axis(length of the dimension is m).

                distance_arr = np.sum(euclidean_dist, axis=2)

                # now we need to find out to which cluster each data point belongs.
                # Use a matrix of n x k where [i,j] = 1 if the ith data point belongs
                # to cluster j.


                min_location = np.zeros(distance_arr.shape)
                min_location[range(distance_arr.shape[0]), np.argmin(distance_arr, axis=1)] = 1

                # calculate J

                j_val = np.sum(distance_arr[min_location == True])
                j_values.append(j_val)

                # calculates the new centroids
                new_centroids = np.empty(centroids.shape)
                for col in range(0, self.k):
                    if self.data[min_location[:, col] == True,:].shape[0] == 0:
                        new_centroids[col] = centroids[col]
                    else:
                        new_centroids[col] = np.mean(self.data[min_location[:, col] == True, :], axis=0)

                # compare centroids to see if they are equal or not

                if self._compare_centroids(centroids, new_centroids):
                    # it has resulted in the same centroids.
                    stabilized = True
                else:
                    centroids = new_centroids
```

```
            except:
                print ('exception!')
                continue
            else:
                iterations += 1

        print ('Required ', iterations, ' iterations to stabilize.')
        return iterations, j_values, centroids, min_location

    def _compare_centroids(self, old_centroids, new_centroids, precision=-1):
        if precision == -1:
            return np.array_equal(old_centroids, new_centroids)
        else:
            diff = np.sum((new_centroids - old_centroids)**2, axis=1)
            if np.max(diff) <= precision:
                return True
            else:
                return False

    def initCost(self):
        t = timeit.Timer(lambda: self._initial_centroids())
        return t.timeit(number=10)




class KMeansP:
    def __init__(self, data, k):
        KMeansBase.__init__(self, data, k)

    def _initial_centroids(self):

        # pick the initial centroid randomly

        centroids = self.data[np.random.choice(range(self.data.shape[0]),1), :]
        data_ex = self.data[:, np.newaxis, :]


     # run k - 1 passes through the data set to select the initial centroids
        while centroids.shape[0] < self.k :

            #print (centroids)

            euclidean_dist = (data_ex - centroids) ** 2
```

```
            distance_arr = np.sum(euclidean_dist, axis=2)
            min_location = np.zeros(distance_arr.shape)
            min_location[range(distance_arr.shape[0]), np.argmin(distance_arr, axis=1)] =
1


            # calculate J

            j_val = np.sum(distance_arr[min_location == True])

            # calculate the probability distribution

            prob_dist = np.min(distance_arr, axis=1)/j_val

            # select the next centroid using the probability distribution calculated before
            centroids = np.vstack([centroids,
self.data[np.random.choice(range(self.data.shape[0]),1, p = prob_dist), :]])
        return centroids



rc = parallel.Client()

# load balanced view

d_view = rc[:]

# do synchronized processing

d_view.block=True

# import Numpy
with d_view.sync_imports():
    import numpy


def initial_centroids(data, k):
    '" select k random data points from the data"'
    return data[numpy.random.choice(range(data.shape[0]), k, replace=False)]

def compare_centroids(old_centroids, new_centroids, precision=-1):
    if precision == -1:
        return numpy.array_equal(old_centroids, new_centroids)
    else:
        diff = numpy.sum((new_centroids - old_centroids)**2, axis=1)
        if numpy.max(diff) <= precision:
```

```
            return True
        else:
            return False

def lloyds_iteration(data, centroids):
    # find the Euclidean distance between a center and a data point

    # now as a result of broadcasting, both array sizes will be n x k x m
    data_ex = data[:, numpy.newaxis, :]
    euclidean_dist = (data_ex - centroids) ** 2
    # now take the summation of all distances along the 3rd axis(length of the dimension is m).
    # This will be the total distance from each centroid for each data point.
    # resulting vector will be of size n x k
    distance_arr = numpy.sum(euclidean_dist, axis=2)

    # now we need to find out to which cluster each data point belongs.
    # Use a matrix of n x k where [i,j] = 1 if the ith data point belongs
    # to cluster j.
    min_location = numpy.zeros(distance_arr.shape)
    min_location[range(distance_arr.shape[0]), numpy.argmin(distance_arr, axis=1)] = 1

    # calculate J
    j_val = numpy.sum(distance_arr[min_location == True])

    # calculates the new centroids
    new_centroids = numpy.empty(centroids.shape)
    membership_count = numpy.empty(centroids.shape[0])
    for col in range(0, centroids.shape[0]):
        new_centroids[col] = numpy.mean(data[min_location[:, col] == True, :], axis=0)
        membership_count[col] = numpy.count_nonzero(min_location[:, col])

    return {'j-value':j_val, 'centroids':new_centroids, 'element-count':membership_count}

def ScalableKMeansPP(data, l, centroids):

    data_ex = data[:, numpy.newaxis, :]

    euclidean_dist = (data_ex - centroids) ** 2

    distance_arr = numpy.sum(euclidean_dist, axis=2)

    # find the minimum distance, this will be the weight
```

```
    min = numpy.min(distance_arr, axis=1).reshape(-1, 1)

    # let's use weighted reservoir sampling algorithm to select l centroid

    random_numbers = numpy.random.rand(min.shape[0], min.shape[1])

    # replace zeros in min if available with the lowest positive float in Python

    min[numpy.where(min==0)] = numpy.nextafter(0,1)

    # take the n^th root of random numbers where n is the weights
    with numpy.errstate(all='ignore'):

        random_numbers = random_numbers ** (1.0/min)
    # pick the highest l

    cent = data[numpy.argsort(random_numbers, axis=0)[:, 0]][::-1][:l, :]

    # combine the new set of centroids with the previous set

    centroids = numpy.vstack((centroids, cent))

    # now we have the initial set of centroids which is higher than k.

    euclidean_dist = (data_ex - centroids) ** 2

    distance_arr = numpy.sum(euclidean_dist, axis=2)


    min_location = numpy.zeros(distance_arr.shape)

    min_location[range(distance_arr.shape[0]), numpy.argmin(distance_arr, axis=1)] =
1

    weights = numpy.array([numpy.count_nonzero(min_location[:, col]) for col in
range(centroids.shape[0])]).reshape(-1,1)
    return {'centroids': centroids, 'weights': weights}



data = numpy.random.randn(1000000,2)
# distribute the data among the engines
d_view.scatter('data', data)

# first pick a random centroid. Ask one engine to pick the first random centroid
centroids = rc[0].apply(initial_centroids, parallel.Reference('data'),1).get()
```

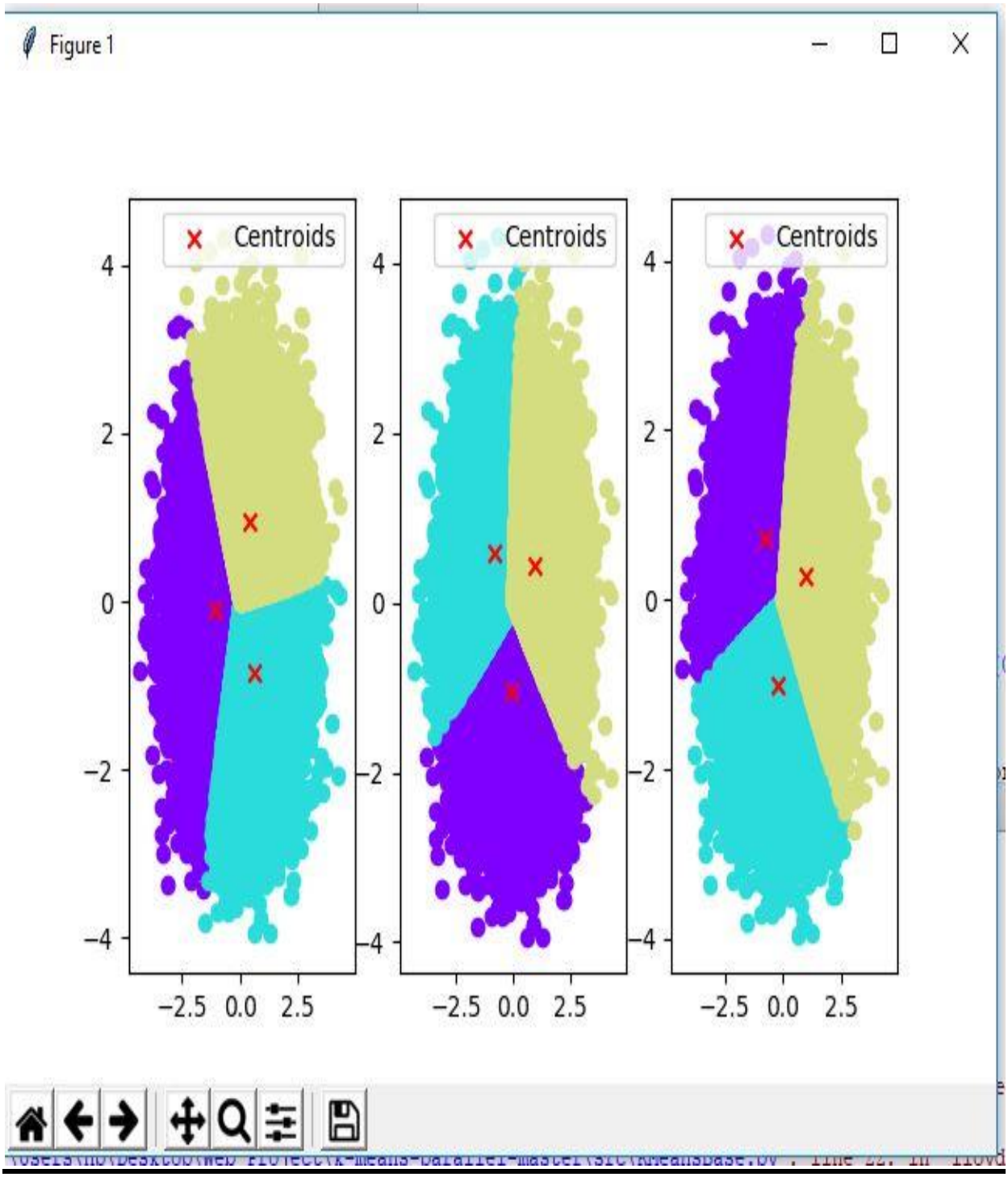
```
r = 3
l = 2
k = 4
passes = 0
while passes < r:
    result = d_view.apply(ScalableKMeansPP, parallel.Reference('data'), l, centroids)
    print('centroids from one engine: ', result[0]['centroids'].shape)
    # combine the centroids for the next iteration
    centroids = numpy.vstack(r['centroids'] for r in result)
    passes += 1
# next step is to calculate k centroids out of the centroids returned by each engine
# for this we use KMeans++
weights = numpy.vstack(r['weights'] for r in result)
kmeans_pp = KMeansPP.KMeansPP(weights, k)

_, _, _, min_locations = kmeans_pp.cluster()

# calculates the new centroids

new_centroids = numpy.empty((k, data.shape[1]))
for col in range(0, k):
    new_centroids[col] = numpy.mean(centroids[min_locations[:, col] == True, :],
axis=0)

centroids = new_centroids
# now do the lloyd's iterations
stabilized = False
iterations = 0
while not stabilized:
    iterations += 1
    ret_vals = d_view.apply(lloyds_iteration, parallel.Reference('data'), centroids)
    member_count = numpy.sum(numpy.array([r['element-count'] for r in
ret_vals]).reshape(len(ret_vals),-1),axis=0)
    local_sum = numpy.array([r['centroids'] * r['element-count'].reshape(-1,1) for r in
ret_vals])
    new_centroids = numpy.sum(local_sum, axis=0)/member_count.reshape(-1,1)
    if compare_centroids(centroids, new_centroids):
        stabilized = True
    else:
        centroids = new_centroids

print('Iterations:', iterations)
```

```
centroids from one engine:  (3, 2)
centroids from one engine:  (14, 2)
centroids from one engine:
```

```
if __name__ == '__main__':
    k = 3
    data = np.random.randn(100000,2)
    #data = np.array([[1.1,2],[1,2],[0.9,1.9],[1,2.1],[4,4],[4,4.1],[4.2,4.3],[4.3,4],[9,9],[8.9,9],[8.7,9.2],[9.1,9]])
    kmeans = KMeansPP(data, k)
    _, _, centroids, min_location = kmeans.cluster()
    # plotting code
    plt.figure()
    plt.subplot(1,3,1)
    colors = iter(cm.rainbow(np.linspace(0, 1, k + 1)))
    for col in range (0,k):
        plt.scatter(data[min_location[:,col] == True, :][:,0], data[min_location[:,col] == True, :][:,1], color=next(colors))


    centroid_leg = plt.scatter(centroids[:,0], centroids[:,1], color=next(colors), marker='x')
    plt.legend([centroid_leg], ['Centroids'], scatterpoints=1, loc='best')

    kmeans = KMeansBase(data, k)
    _, _, centroids, min_location = kmeans.cluster()
    plt.subplot(1,3,2)
    colors = iter(cm.rainbow(np.linspace(0, 1, k + 1)))
    for col in range (0,k):
        plt.scatter(data[min_location[:,col] == True, :][:,0], data[min_location[:,col] == True, :][:,1], color=next(colors))

    centroid_leg = plt.scatter(centroids[:,0], centroids[:,1], color=next(colors), marker='x')
    plt.legend([centroid_leg], ['Centroids'], scatterpoints=1, loc='best')


    kmeans = ScalableKMeansPP(data, k, 2, 2)
    _, _, centroids, min_location = kmeans.cluster()
    plt.subplot(1,3,3)
```

```
    colors = iter(cm.rainbow(np.linspace(0, 1, k + 1)))
    for col in range (0,k):
        plt.scatter(data[min_location[:,col] == True, :][:,0], data[min_location[:,col] == True, :][:,1], color=next(colors))

    centroid_leg = plt.scatter(centroids[:,0], centroids[:,1], color=next(colors), marker='x')
    plt.legend([centroid_leg], ['Centroids'], scatterpoints=1, loc='best')

    plt.show()
```

4.Output



5.References :

1. ieeexplore.ieee.org/document/6128477
2. https://www.cse.buffalo.edu//faculty/miller/Courses/CSE633/Chandramohan-Fall-2012-CSE633.pdf
3. http://bobweigel.net/wiki/images/Tboggs_project_report_csi702.pdf
4. http://ai2pdfs.s3.amazonaws.com/23db/2d9c5a97f36f1b63ea249402b4be0919ebc9.pdf