Parallel and Distributed Computing

Implementation of Scalable K means ++

using Hadoop

TEAM MEMBERS-

Adeep Biswas - 16BCI0169
Belal Ahmed- 16BCE0281
Mudit Agarwal - 16BCE0663
Abhishek C - 16BCE0564
Sourav Agarwalla – 16BCE0608

FACULTY – DR. ANTO S

VIT®
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)
VELLORE ■ CHENNAI
www.vit.ac.in

SCOPE

Abstract:-

Our aim in this project is to implement the scalable K-means++ algorithm. This algorithm allows k-means 0 clustering for larger number of data points efficiently. It is an efficient way to initialize centres Implementation of this paper. Now about the K-means algo which is one of the widely used data processing algorithms. However, the initialization in the algorithm is crucial for obtaining a good solution. K-means++ caters to this, by obtaining an initial set of centers that is close to the optimum solution. But k-means++ is a sequential algorithm and is inefficient over large data-set, k-passes need to be made over the entire data to find a set of good initial centres. It is important to reduce the number of passes made to get good initialization.

Deliverables:-

At the end of this project, we have:

- Implementation of the algorithm in sequential format.
- Implementation of the algorithm in parallel format.

Implementation:-

The implementation language will be Python. Core parts of the work will be done in Cython.

Unit tests will be added to validate against the existing clustering algorithms. Benchmark analysis will be done using the make_blobs dataset. Documentation and examples will be added to the user guide and the API.

- The initialization function will be added in cluster/k_means_.py

Description:-

Currently, k-means clustering algorithm does not perform very well on really large datasets. The scalable-kmeans++ algorithm aims to uses a parallel implementation which makes choosing the centers faster. Also as more than one candidate centers can be chosen at a point, it makes this type of initialization less sensitive to outliers.

Intuitive explanation of the algorithm:-

kmeans++ needs k number of passes over the data, for an application with large amount of data, k can value to 100 or even 1000.

We want to pick each point independently so we do not want to choose points from a joint distribution we pick each point independently This is very useful for distributed implementation because each point independently gets to decide lif it should be in the set of centres or not. This intuitively corresponds to updating your distribution much more infrequently which is a much more coarser sampling.

It works as: when we have a distribution we pick some points from it and then update the distribution. More points are taken from the distribution we get after the above step. This is k means parallel a.k.a Scalable k-means++.

Methodology:-

In this work we acquire a parallel variant of the k-means++ introduction calculation and exactly exhibit its viable .The primary thought is that as opposed to testing a solitary point in each go of the k-means++ calculation, we test O(k) focuses in each round and rehash the procedure for around O(log n) rounds. Toward the finish of the calculation we are left with O(k log n) focuses that shape an answer that is inside a steady factor far from the ideal. We at that point recluster these O(k log n) focuses into k introductory places for the Lloyd's cycle. This introduction calculation, which we call k-means||, is very basic and fits simple parallel usage.

Lloyd iterations:-

In simple terms Lloyd iterations are nothing but randomly sampling k initial cluster centroids from our dataset.

The process starts when some number k of point sites are initially placed in the input domain. These would be the vertices that are to be smoothed, as said above in some use cases they may be placed at random or by intersecting a triangular mesh which would be with the input domain

Our Implementation:–

```
def randCent(dataSet, K):
    n =shape(dataSet)[0]
    #initialize 2-d mat with first center selected arbitrarily
    elected_centers = [dataSet[0,:].tolist()[0]]

    for k in range(1,K):
        Dx = array([min([distEclud(c,x) for c in elected_centers]) for x in dataSet])
        r = random.rand()
        probs = Dx/sum(Dx)
        cumsprobs = cumsum(probs)

        for j,p in enumerate(cumsprobs):
            if r < p:
                i = j
                break
        elected_centers.append(dataSet[i].tolist()[0])

    return matrix(elected_centers)
```

Limitations:-

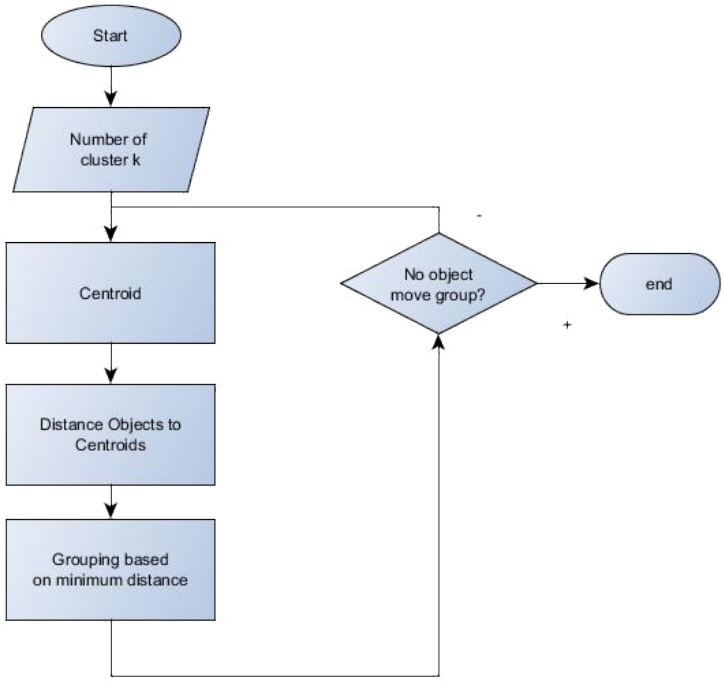
- Random initializations although fast would give rise to more number of iterations until convergence.
- Heavily susceptible to convergence at local minima

Kmeans++:-

K means is an algorithm used to cluster or gather together n number of objects based similar feature and attributes into k partitions. [k<n]

It has some aspects that are similar to the expectation-maximisation algorithm. The similar aspect is that both the algorithms try to find a natural cluster’s centre in the data.

How kmeans++:-



Step A: Let k = number of clusters

Step B: The partitions that classify the data are to be put into the k clusters. The initial centroid selection is done using a distance cumulative probability distribution function.

Step C: Get all the samples in a sequence and then calculate their distance from the centroid of each of their clusters. If the selected sample in not present in the cluster with the closest centroid, switch the sample to that cluster and update the centroid of the cluster.

Step D: Repeat step C until the centroids stop moving or convergence happens.

Limitations:-

- Asymptotic time taken is very long.
- This algorithm does no scale with data.

Code:-

[Sequential execution of scalable Kmeans++]

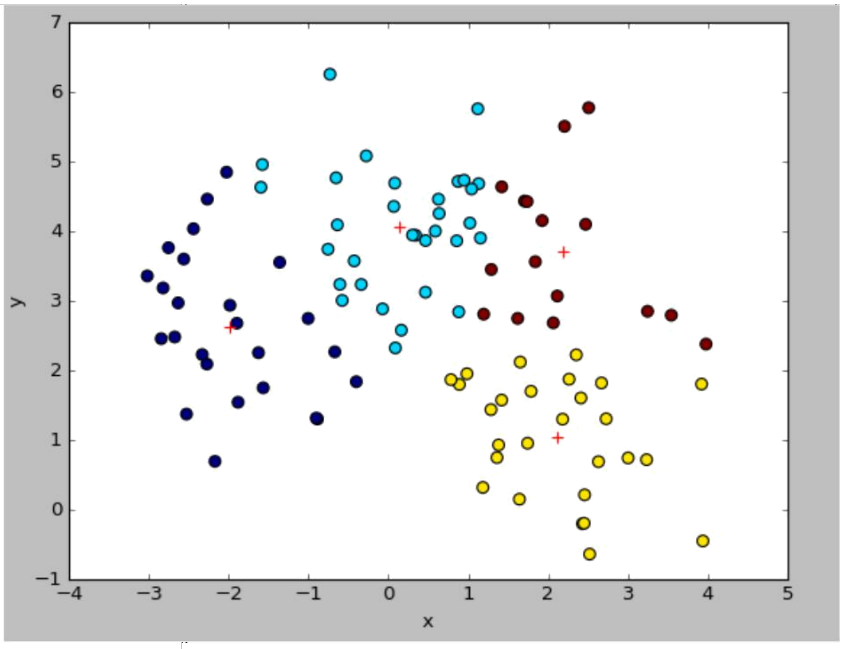
```
from math import log,ceil
from numpy import *


def loadDataSet(fileName):
    dataMat = []
    fr = open(fileName)
    for line in fr.readlines():
        curLine = line.strip().split('\t')
        fltLine = map(float,curLine)
        dataMat.append(fltLine)
    return dataMat


def distEclud(vecA, vecB):
    return sqrt(sum(power(vecA-vecB,2)))
def createCent(dataSet, K,sampling_factor=2):
    '''
        Parameters:
        ----------
        dataSet: numpy.matrix
        K: integer, number of clusters
        sampling_factor: integer
      #let dataSet be an array of arrays
    elected_centers = []
    candidate_centers = []
    if type(dataSet) == mat:
        dataSet = dataSet.A
    else:
```

```
    dataSet = array(dataSet)
#initialize 2-d mat with first center selected arbitrarily
candidate_centers.append(dataSet[0, :].tolist())

#find the nearest distance to the closest centers in candidate_centers
Dx = array([min([distEclud(c, array(x)) for c in candidate_centers]) for x in dataSet])
psi = sum(Dx)
l = int(ceil(log(psi)))

#This is to avoid recalculation of Dx and psi one time
count = True
for k in range(l):
    if count:
        Dx = array([min([distEclud(c, array(x)) for c in candidate_centers]) for x in dataSet])
        psi = sum(Dx)
    else:
        count = False
    r_points = random.random_sample((sampling_factor, ))
    probs = (sampling_factor * Dx)/psi
    cumsumprobs = cumsum(probs)
    # parallel job start
    for r in r_points:
        for j,p in enumerate(cumsumprobs):
            if r < p:
                i = j
                break
        candidate_centers.append(dataSet[i, :].tolist())
        dataSet = delete(dataSet, i, axis = 0)
    #parallel job stop
w = [0 for _ in range(len(candidate_centers))]
```

```
for i in range(len(dataSet)):
    minDist = inf
    for j,c in enumerate(candidate_centers):
            dist = distEclud(c, array(dataSet[i]))
            if dist < minDist:
                minDist = dist
                index = j
    w[index]+=1
#select k-clusters according to kmeans++
w = array(w)
probs = w/float(sum(w))
cumsprobs = cumsum(probs)
for k in range(K):
    r = random.rand()
    for j,p in enumerate(cumsprobs):
        if r < p and candidate_centers[j] not in elected_centers:
            index = j
            elected_centers.append(candidate_centers[index])
            break
        return elected_centers
```

Output:-

```
centroids are: [[-2.26646701  4.46089686]
 [ 0.4666179   3.86571303]
 [-0.40026809  1.83795075]
 [ 2.20656076  5.50616718]]
[[-1.9801693   2.62890834]
 [ 0.14423231  4.06904335]
 [ 2.12273919  1.05026024]
 [ 2.17928197  3.71035532]]
cluster: [[ 2.]
 [ 1.]
 [ 2.]
 [ 1.]
 [ 1.]
 [ 1.]
 [ 0.]
 [ 1.]
 [ 2.]
 [ 1.]
 [ 3.]
 [ 3.]
 [ 2.]
 [ 3.]
 [ 0.]
 [ 2.]
 [ 0.]
 [ 1.]
 [ 1.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 1.]
 [ 2.]
 [ 2.]
 [ 2.]
 [ 2.]
 [ 0.]
```



Code:-

[Parallel execution of scalable Kmeans++]

MAPPER CLASS (mapper.py)

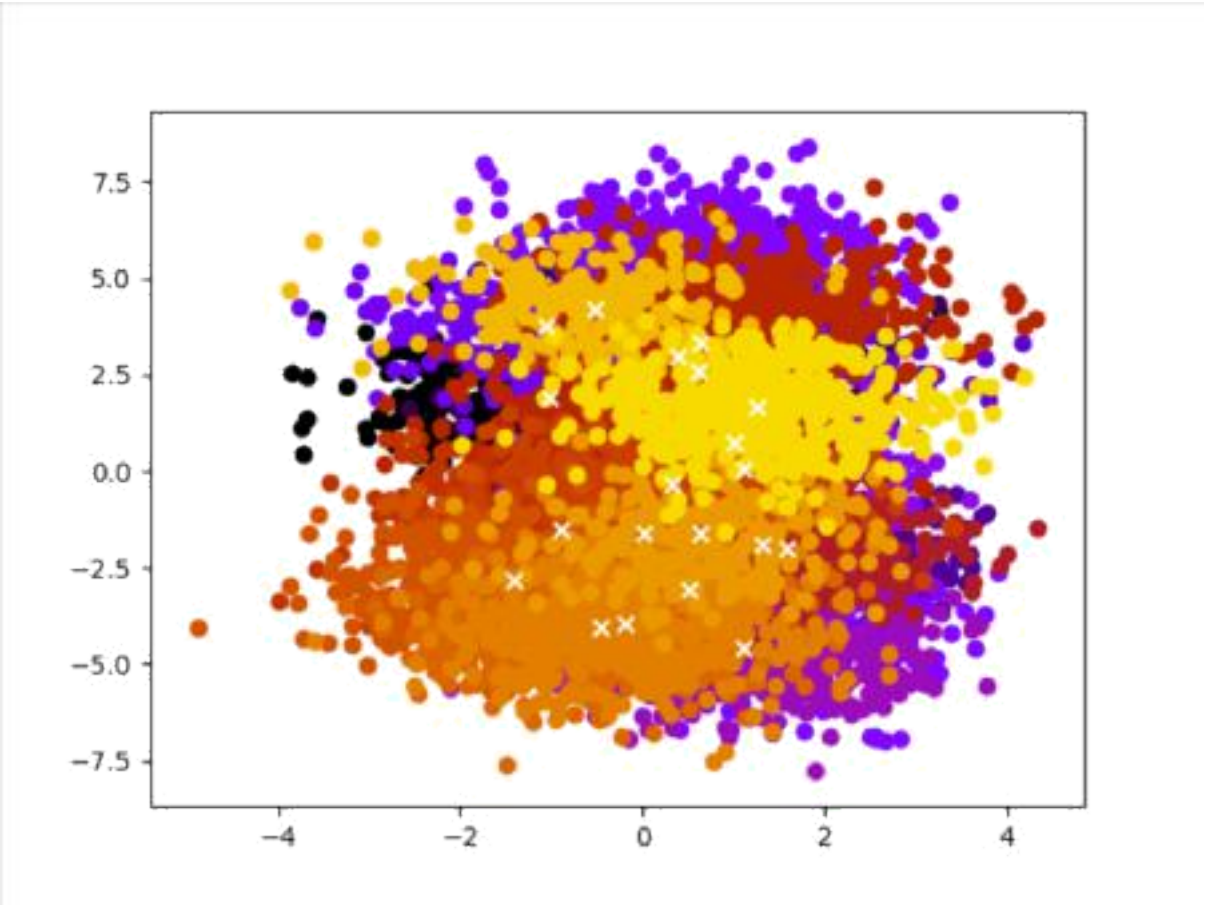
```
def ScalableKMeansPlusPlus_cy(data, k, l, weighted=False, iter=5):


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


    for i in range(iter):
        #Get the distance between data and centroids
        dist = distance_cy(data, centroids)

        #Calculate the cost of data with respect to the
        centroids norm_const = cost_cy(dist)

        #Calculate the distribution for sampling l new
        centers p = distribution_cy(dist,norm_const)

        #Sample the l new centers and append them to the original
        ones centroids = np.r_[centroids, sample_new_cy(data,p,l)]


    #   reduce k*l to k using KMeans++ dist
    = distance_cy(data, centroids) weights =
    get_weight_cy(dist,centroids) if
    weighted:
        initial = centroids[np.random.choice(range(len(weights)),k,replace=False),:]
        centers= weightedKMeans(centroids, k, weights, initial)
    else:
        centers= centroids[np.random.choice(len(weights), k, replace= False, p = weights),:]
    return centers
```

REDUCER CLASS (reducer.py)

```
import numpy as np
from kmeans_func import KMeans
from kmeanspp_func import KMeansPlusPlus
from sklearn.datasets.samples_generator import make_blobs
import matplotlib.pyplot as plt
#   Simulate data
k = 20
n = 10000
d = 15
```

```
#   simulate k centers from 15-dimensional spherical Gaussian
distribution mean = np.hstack(np.zeros((d,1)))
cov = np.diag(np.array([1,10,100]*5))
centers = np.random.multivariate_normal(mean, cov, k)

#   Simulate n data
for i in range(k):
   mean = centers[i]
   if i == 0:
      data = np.random.multivariate_normal(mean, np.diag(np.ones(d)), int(n/k+n%k))
      trueLabels = np.repeat(i,int(n/k+n%k))
   else:
      data = np.append(data, np.random.multivariate_normal(mean,
np.diag(np.ones(d)) , int(n/k)), axis = 0)
      trueLabels = np.append(trueLabels,np.repeat(i,int(n/k)))

centroids_initial = KMeansPlusPlus(data, 20)
output_kpp = KMeans(data, k, centroids_initial)

cmap = plt.get_cmap('gnuplot')
colors = [cmap(i) for i in np.linspace(0, 1, k)]

centroids1 =output_kpp["Centroids"]
labels1 = output_kpp["Labels"]

for i,color in enumerate(colors,start =1):
   plt.scatter(data[labels1==i, :][:,0], data[labels1==i, :][:,1], color=color)

for j in range(k):
   plt.scatter(centroids1[j,0],centroids1[j,1],color = 'w',marker='x')

plt.show()
```

Output:-



Result:-

Compare Clustering Cost

In [31]:
```
from distance_func import distance
from kmeanspp_func import cost

def clusterCost(data,predict):
    dist = distance(data,predict["Centroids"])
    return cost(dist)/(10**4)
```

In [32]:
```
print("Clustering Cost:")
print("Random:", clusterCost(data, output_k)) # Random
print("KMeans++:",clusterCost(data, output_kpp)) # KMeans++
print("Scalable KMeans++:", clusterCost(data, output_spp)) # Scalable KMeans++
```

```
Clustering Cost:
Random: 47.8283387768
KMeans++: 19.7374814982
Scalable KMeans++: 52.9138532121
```

The clustering cost of the final results using KMeans ++ and Scalable KMeans++ are similar and much better than the clustering cost of random initial centroids.

Compare runtime

In [33]:
```
%%time
a = KMeansPlusPlus(data, k)  # KMeans++
b = KMeans(data,k,a)
```

```
Converge! after: 22 iterations
CPU times: user 10.2 s, sys: 20 ms, total: 10.2 s
Wall time: 10.1 s
```

In [34]:
```
%%time
a = ScalableKMeansPlusPlus(data, k, l) # Scalable KMeans++
b = KMeans(data,k,a)
```

```
Converge! after: 27 iterations
CPU times: user 12.5 s, sys: 28 ms, total: 12.5 s
Wall time: 12.5 s
```

In [35]:
```
%%time
a = ScalableKMeansPlusPlus_cy(data, k, l) # Scalable KMeans++ with cython
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 8.62 s, sys: 32 ms, total: 8.65 s
Wall time: 8.64 s
```

In [36]:
```
%%time
a = ScalableKMeansPlusPlus_p(data, k, l) # Scalable KMeans++ with parallel
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 9.54 s, sys: 300 ms, total: 9.84 s
Wall time: 12 s
```

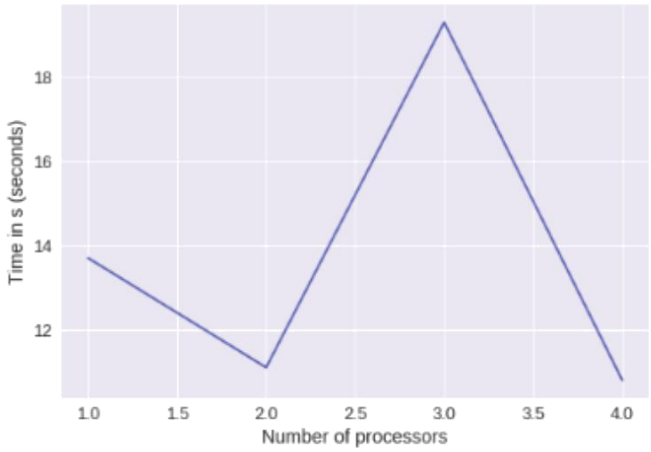
Conclusion:-

As we can see in the results. Scalable Kmeans++ works much better than random initialization. Here the parallel version is computed with a relatively slow pace, but if more data is added it would work faster and more efficiently than the kmeans++ version.

Scalable Kmeans ++ with multiple processors:

As we increase the number of processors used, the speed increases.

In [36]:
```
y = np.array([13.7, 11.1, 19.3, 10.8])
x = np.array([1,2,3,4])
#plt.plot(x, y)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_xlabel("Number of processors")
ax.set_ylabel("Time in s (seconds)")
ax.plot(x, y)
plt.show()
```



The runtime of KMeans++ is slow, since it sample k initial centroids one by one. The runtime of `Scalable KMeans++` without speeding up is even slower than KMeans++, since it sample more than k centroids and then clustering them again. After speeding up with `Cython` and `multiprocessing`, the `Scalable KMeans` works better in most case (not so stable).

In [29]:
```
%%time
a = ScalableKMeansPlusPlus_p(data, k, l , cpu_count=1) # Scalable KMeans++ with parallel
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 10.7 s, sys: 204 ms, total: 10.9 s
Wall time: 13.7 s
```

In [30]:
```
%%time
a = ScalableKMeansPlusPlus_p(data, k, l , cpu_count=2) # Scalable KMeans++ with parallel
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 8.49 s, sys: 324 ms, total: 8.82 s
Wall time: 11.1 s
```

In [31]:
```
%%time
a = ScalableKMeansPlusPlus_p(data, k, l , cpu_count=3) # Scalable KMeans++ with parallel
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 16.6 s, sys: 416 ms, total: 17.1 s
Wall time: 19.3 s
```

In [32]:
```
%%time
a = ScalableKMeansPlusPlus_p(data, k, l , cpu_count=4) # Scalable KMeans++ with parallel
b = KMeans_cy(data,k,a)
```

```
Converge
CPU times: user 8.02 s, sys: 512 ms, total: 8.53 s
Wall time: 10.8 s
```

In [36]:
```
y = np.array([13.7, 11.1, 19.3, 10.8])
x = np.array([1,2,3,4])
```

References:

[1] M. R. Ackermann, C. Lammersen, M. M•artens, C. Raupach, C. Sohler, and K. Swierkot. StreamKM++: A clustering algorithm for data streams. In ALENEX, pages 173{187, 2010.

[2] N. Ailon, R. Jaiswal, and C. Monteleoni. Streaming k-means approximation. In NIPS, pages 10{18, 2009.

[3] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of Euclidean sum-of-squares clustering. Machine Learning, 75(2):245{248, 2009.

[4] D. Arthur and S. Vassilvitskii. How slow is the k-means method? In SOCG, pages 144{153, 2006.

[5] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In SODA, pages 1027{1035, 2007.

[6] B. Bahmani, K. Chakrabarti, and D. Xin. Fast personalized PageRank on MapReduce. In SIGMOD, pages 973{984, 011.

[7] B. Bahmani, R. Kumar, and S. Vassilvitskii. Densest subgraph in streaming and mapreduce. Proc. VLDB Endow., 5(5):454{465, 2012.

[8] P. Berkhin. Survey of clustering data mining techniques. In J. Kogan, C. K. Nicholas, and M. Teboulle, editors, Grouping Multidimensional Data: Recent Advances in Clustering. Springer, 2006.

[9] P. S. Bradley, U. M. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. In KDD, pages 9{15, 1998.

[10] E. Chandra and V. P. Anuradha. A survey on clustering algorithms for data in spatial database management systems. International Journal of Computer Applications, 24(9):19{26, 2011.