Project 3
ESE 545, Data Mining: Learning from Massive Datasets
Team members: Shuairui Yao, Penn ID: 47673675; Zhefu Peng, Penn ID: 29419150

Introduction

The purpose of this project is to learn to process a massive dataset with programming techniques in practice, using Python to implement algorithms, K-means and K-means ++, with its improved version, such as mini-Batch, which could help us develop better clusters, getting better performance.

Problem1 Data Importing and Problem Formulating

As the problem mentions, we need to import data, and we need to formulate the problem by ourselves in order to avoid the trouble that dataset is too big. What's more, here is a tip, we are allowed to create new features from the data. Here are what we do:

i. Load dataset, using pandas to load csv file, and we can see the dimension of dataset is 1326100 * 21

```
import pandas as pd
import random
import numpy as np
import time
data = pd.read_csv('yelp.csv')
```

```
# data # 1326100 * 21
```

ii. Then, we start to process the data, discarding the useless feature and creating the feature we need. Our focus is on the popularity of each yelp user. The first one is the year, we just want to know how long ago did the comment make; And we create a feature called "ufcf_sum", aiming at calculating the number of feedback for each comment; And we create another feature called "compliment_sum", which is the sum of compliment features; At last we create a feature called "elite_count", collecting the whole information about elite.

```
# data perprocessing
data['yelping_year'] = pd.to_numeric(data['yelping_since'].str.split('-').str[0])
data['yelping_year'] = data['yelping_year'].map(lambda x: 2018 - x)
data['ufcf_sum'] = data['useful']+data['funny']+data['cool']+data['fans']
data['compliment_sum'] = data['compliment_more']+data['compliment_profile']+data['compliment_cute']+data['compliment_lis
data['elite_count'] = data['elite'].str.split(',')
data['elite_count'] = data['elite_count'].map(lambda x:len(x) if x != ['None'] else 0)
```

Then, we discard every original feature except "review_count", and getting the new features we created before.

```
data1 = data[['review_count','yelping_year','ufcf_sum','compliment_sum','elite_count']]
```

```
data1.head()
```

	review_count	yelping_year	ufcf_sum	compliment_sum	elite_count
0	10	5	0	0	0
1	1	1	0	0	0
2	6	3	0	0	0
3	3	2	0	0	0
4	11	6	4	1	0

iii. And changed the type from data frame to numpy array, which would be convenient for us in later questions.

```
dataset = np.array(data1).astype(np.float64)
```

Problem2 Implement mini-Batch version of K-means.

Since original K-means has its own defect, and the dataset we used is too big, we need to implement online version of K-means to accelerate the processing and improve the performance, here is the screen shot of mini-batch K-means algorithm that we depended on:

Algorithm 1 Mini-batch k-Means.

Given: k, mini-batch size b, iterations t, data set X
Initialize each $\mathbf{c} \in C$ with an $\mathbf{x}$ picked randomly from X
$\mathbf{v} \leftarrow 0$
for $i = 1$ to t **do**
  $M \leftarrow b$ examples picked randomly from X
  for $\mathbf{x} \in M$ **do**
    $\mathbf{d}[\mathbf{x}] \leftarrow f(C, \mathbf{x})$ // Cache the center nearest to $\mathbf{x}$
  end for
  for $\mathbf{x} \in M$ **do**
    $\mathbf{c} \leftarrow \mathbf{d}[\mathbf{x}]$ // Get cached center for this $\mathbf{x}$
    $\mathbf{v}[\mathbf{c}] \leftarrow \mathbf{v}[\mathbf{c}] + 1$ // Update per-center counts
    $\eta \leftarrow \frac{1}{\mathbf{v}[\mathbf{c}]}$ // Get per-center learning rate
    $\mathbf{c} \leftarrow (1 - \eta)\mathbf{c} + \eta\mathbf{x}$ // Take gradient step
  end for
end for

i. Writing down a function called “minibatchKmeans”, and the parameter of the function is the batch size, the number of iterations, dataset and the number of centroids we want.

```
def minibatchKmeans(k, b, data, T):
```

ii. Firstly, we randomly select the k points from whole dataset; then we need to create a vector “v”, to record how many points for each cluster it has, then we need to update the centers according to the number of points they have. The more points they have, the variance and update of the center would be smaller, that is called “gradient step”.

```
def minibatchKmeans(k, b, data, T):
#     First, selecting the initialization of centroids from dataset
    centers = data[list(random.sample(range(0,len(data)),k))]
    # Then, we setting the vector v to record the number of points for each centroid
    v = np.zeros([k,1], dtype = float)
    # Starting the iteration:
    for i in range(T):
        # Selecting the sample data
        sample_data = data[list(random.sample(range(0,len(data)),b))]
        # For each sample data, finding the nearest center
        for j in range(b):
            # distance =  np.sqrt(np.sum(np.power(sample_data[j] - centers, 2),axis = 1))
            distance = np.linalg.norm((sample_data[j]-centers),axis = 1)
            # Finding the center index
            min_index = np.argmin(distance)
            v[min_index] += 1
            ita = 1/v[min_index]
            centers[min_index]  = (1-ita)*centers[min_index] + ita*sample_data[j]
    return centers
```

Problem3 Implement K-means++ Method to Initialize Centers

The purpose we implement K-means++ method, is to choose more appropriate centers to get better performance, since the performance of K-means algorithm depends much on the initialization centroids.

i. First, we need to write down a function which could calculate the distance between two vectors (or matrix).
Write a function called "disSqToCloseC" which could return the minimum square root of distance from data points to their corresponding centroid. The principle of this part code is Square difference formula.

```
def disSqToCloseC(X,C):
    '''
    input: X, m*n; C, k*n
    '''
    if(X.shape[0]==1):
        X.reshape(1,-1)
    if(C.shape[0]==1):
        C.reshape(1,-1)
    dis = -2*np.dot(X,C.T) # m*k
    dis += np.sum(X**2,axis=1)[:,np.newaxis] # axis = 1, row
    dis += np.sum(C**2,axis=1)[np.newaxis,:]
    min_ds_sq = np.min(dis, axis = 1) # m*k distance to centers
    return min_ds_sq
```

ii. Now we need to write down a code that implement the K-means++ method to choose initialization centroids. In sum, it means that we need to choose the center according to their probability, and the probability increases while distance becomes larger:

```
def kmeanspp(data,k):
    center_pos = []
    total = 0
    first = random.choice(range(len(data)))
    center_pos.append(first)
    for i in range(1,k):
        weights = disSqToCloseC(data,data[center_pos])
        rand = random.random()
        cdf = 0
        idx = -1
        while cdf < rand:
            idx += 1
            cdf += weights[idx]
        center_pos.append(idx)
    return data[center_pos]
```

iii. Then, we still use the same function as the problem 2 we wrote before, to update the centroids, which will help us guarantee the speed and performance at the same time.

```
def minibatchKmeanspp(k, b, data, T):
#     First, selecting the initialization of centroids from dataset
    centers = kmeanspp(data,k)
    # Then, we setting the vector v to record the number of points for each centroid
    v = np.zeros([k,1], dtype = float)
    # Starting the iteration:
    for i in range(T):
        # Selecting the sample data
        sample_data = data[list(random.sample(range(0,len(data)),b))]
        # For each sample data, finding the nearest center
        for j in range(b):
            distance = np.linalg.norm((sample_data[j]-centers),axis = 1)
            # Finding the center index
            min_index = np.argmin(distance)
            v[min_index] += 1
            ita = 1/v[min_index]
            centers[min_index]  = (1-ita)*centers[min_index] + ita*sample_data[j]
    return centers
```

Problem4 Design Our Own Selecting Initial Centroids Algorithm

As the problem mentions, we are asked to design our own algorithm to choose initial centroids. The requirement is running time should less than 5 minutes, and we can't set the algorithm to choose the furthest points, since it may cause the noise increases.

i. The function for this algorithm called "getinitC". First we need to find the column (or feature) that has most different value, which means, the feature is the most varied. To find it, we calculate each column's standard deviation.
Then, we pick the column we need, resetting its order (from minimum to maximum, top to bottom), and make the dataset in new order as the column we need.

```
def getinitC(k,data,b,T):
    ## First, finding the column which has highest standard deviation
    dev_list = data.std(axis=0)
    dev_list = dev_list.tolist() ## Converting to list to find the index
    col_idx = dev_list.index(max(dev_list))
    ## Then, we changed the order of the matrix accoring to the column order
    order= np.argsort(data[:,col_idx]) # Finding the order that the elements of column in order (min -> max)
    data neworder = data[order] ## Getting the dataset which is in new order
```

ii. Then, we need to split the whole dataset (new order) into k parts. Creating an empty list, called "dataset_list", then using a for loop to split the dataset, and append it to the list we created.

```
    ## After that, we need to split the data into k parts.
    dataset_list = []
    for i in range(k):
#         datasetlist.append(data_neworder[i*(len(data_neworder)//k):(i+1)*(len(dataset)//k)]) ## The list should have k
        dataset_list.append(data_neworder[i*(len(data_neworder)//k):(i+1)*(len(data_neworder)//k)])## The list should ha
    if (len(data_neworder)%k) == 0:
        pass
#         return dataset_list
    else:
        ## If the number of points is not integer multiple of k, remaining points will be added to the last one cluster
        dataset_list[-1] = np.r_[dataset_list[-1], data_neworder[-(len(data_neworder)-k*(len(data_neworder)//k)):]]
#     print(dataset list)
```

iii. Finally, we need to create an empty list called center_list, and in the for loop in the dataset_list, for every single part of dataset, we use the mini-Batch function we wrote before, for every single part, we find one center, as the initial centroid.

```
    ## Now, we have a dataset_list, which have k items(matrixs),
    ## We need to find only one center in each matrix
    center_list = []
    for matrix in dataset_list:
        center_list.append(minibatchKmeans(1,b,matrix,T)) ## The list should have k items, each is 1*5.
    return center_list
```

iv. But we have to return a matrix, not the list. So we need to convert the list into matrix. The size of matrix is k*n. And the matrix called "centers_final" is the initial centroids matrix we need.

```
#     return center_list
    ## If wanna return a matrix
    centers_final = np.zeros([k,5])
    for center in center_list:
        centers_final = np.r_[centers_final,center]
    centers_final = centers_final[k:,]
    return centers_final
```

Problem5 Evaluate the performance of Three Algorithms

We should write a function “evaluate” which returns the mean distance, minimum distance and maximum distance between a set of data points and k centers, helping us to view the performance directly.

i. Writing down the code to create a function called “evaluate”. And for the method how to calculate the distance we still use the same method in problem 3.

```
def evaluate(data, centers):
    dis = -2*np.dot(data,centers.T) # m*k, all the distances between datas X and centers C
    dis += np.sum(data**2,axis=1)[:,np.newaxis] # axis = 1, row
    dis += np.sum(centers**2,axis=1)[np.newaxis,:] # axis = 0, col
    dis = np.sqrt(np.abs(dis))
    min_d = np.min(dis,axis = 1)
    mean = np.mean(min_d)
    minn = np.min(min_d)
    maxx = np.max(min_d)
    return mean, minn, maxx
```

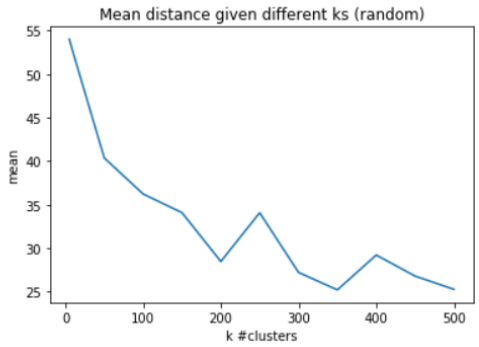
ii. For the problem2 performance, we set the k = [5,10,50,100,150,200,250,300,350,400,450,500], batch size = 100, iteration number = 200.

```
# random
start = time.clock()

ks = [5,10,50,100,150,200,250,300,350,400,450,500] # time: 17min
# k varies, b=1000, data, T=200
centers = []
means = []
minns = []
maxxs = []
for k in ks:
    print("k",k)
    center = minibatchKmeans(k, 1000, dataset, 200)
    meann, minn, maxx = evaluate(dataset,center)
    centers.append(center)
    means.append(meann)
    minns.append(minn)
    maxxs.append(maxx)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

In [83]:
```
from matplotlib import pyplot as plt
# plot mean min & max
plt.plot(ks,means)
plt.xlabel('k #clusters')
plt.ylabel('mean')
plt.title("Mean distance given different ks (random)")
plt.show()
```



```
# mean min max of random method
print("meanspp",means)
print("minnspp",minns)
print("maxxspp",maxxs)
```

```
meanspp [53.99022454535128, 52.42370319530689, 40.35938393978693, 36.228390159984414, 34.08412557472313, 28.481991420
518014, 34.0760859953918, 27.225440522198753, 25.22557177231526, 29.221671454068055, 26.808265574755143, 25.285604197
974806]
minnspp [0.43435908459693356, 0.3276618865364444, 0.022691269078636805, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
maxxspp [666405.5850189615, 666644.7085146975, 656671.0289872445, 640606.1448198278, 639642.4571144939, 603579.951078
7335, 658154.319256516, 593587.6129568394, 553629.3229722008, 643191.8221670089, 589589.0291315503, 516887.5446114259
]
```

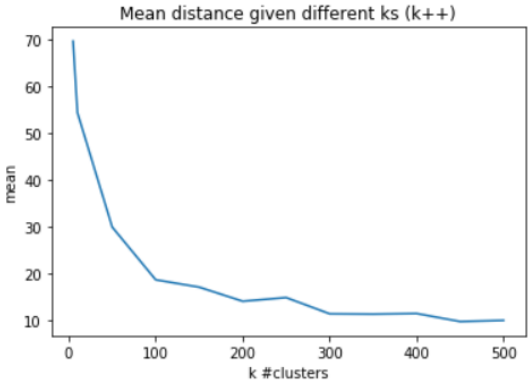
iii. For the problem3, that we need to use K-means++, the values of k and batch size remain, reduce the number of iterations to 80.

```
# kmeans++
start = time.clock()

ks = [5,10,50,100,150,200,250,300,350,400,450,500]
# k varies, b = 1000, data, T = 80
centerspp = []
meanspp = []
minnspp = []
maxxspp = []
for k in ks:
    print("k",k)
    center = minibatchKmeanspp(k, 1000, dataset, 100)
    meann, minn, maxx = evaluate(dataset,center)
    centerspp.append(center)
    meanspp.append(meann)
    minnspp.append(minn)
    maxxspp.append(maxx)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

```
from matplotlib import pyplot as plt
# plot mean min & max
plt.plot(ks,meanspp)
plt.xlabel('k #clusters')
plt.ylabel('mean')
plt.title("Mean distance given different ks (k++)")
plt.show()
```




```
# mean min max of kmeans++ method
print("meanspp",meanspp)
print("minnspp",minnspp)
print("maxxspp",maxxspp)
```

```
meanspp [69.76085770489725, 54.486812630305735, 29.97007279654489, 18.683918369660287, 17.10802206252377, 14.09148636
744423, 14.872889674042971, 11.401857222979526, 11.342638350318014, 11.465080391869629, 9.745774674100659, 9.99203140
2104137]
minnspp [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
maxxspp [373779.5352383541, 210606.04878777819, 50537.76079329198, 21687.302921294755, 17806.947604797402, 11408.7241
17972175, 12409.55659965335, 8563.360482310669, 7648.389111440395, 7938.987529905813, 9771.678207964074, 6564.3055230
542095]
```

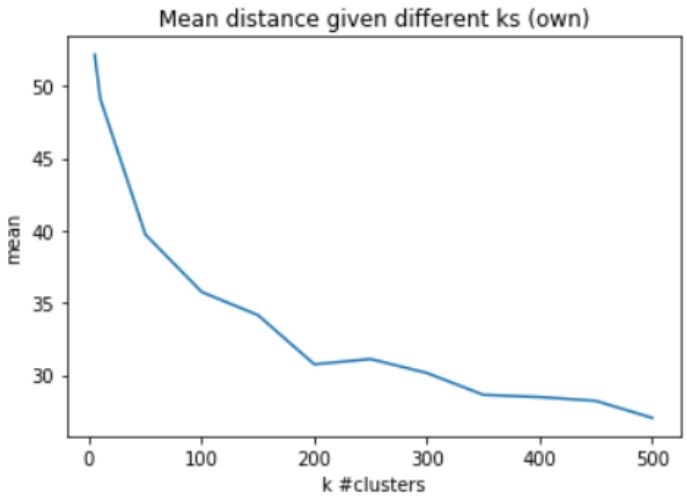
iv. For problem4, that our designed algorithm to choose initial centroids, the values of k and batch size remain, increasing the number of iterations to 1000.

```
# own method
start = time.clock()

ks = [5,10,50,100,150,200,250,300,350,400,450,500] # time: 17min
# k varies, b=1000, data, T=200
centersown = []
meansown = []
minnsown = []
maxxsown = []
for k in ks:
    print("k",k)
    center = minibatchKmeansOwn(k, 1000, dataset, 200)
    meann, minn, maxx = evaluate(dataset,center)
    centersown.append(center)
    meansown.append(meann)
    minnsown.append(minn)
    maxxsown.append(maxx)

elapsed = (time.clock()-start)/60
print("minute used: ",elapsed)
```

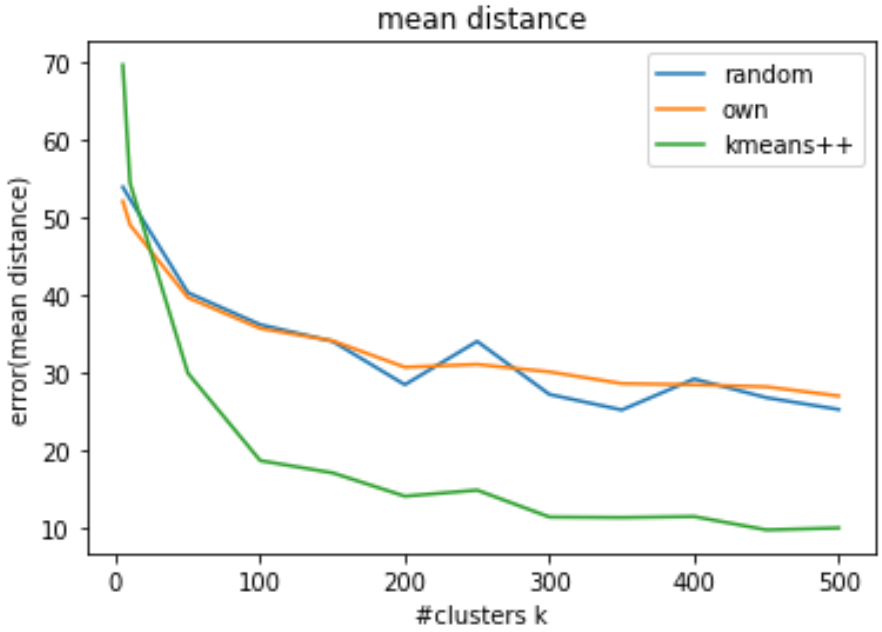
```
from matplotlib import pyplot as plt
# plot mean min & max
plt.plot(ks,meansown)
plt.xlabel('k #clusters')
plt.ylabel('mean')
plt.title("Mean distance given different ks (own)")
plt.show()
```



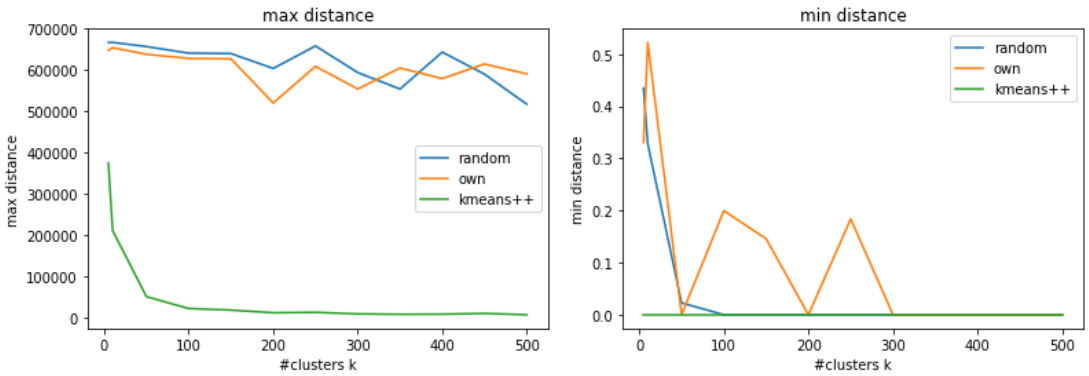

```
# mean min max of own method
print("meansown",meansown)
print("minnsown",minnsown)
print("maxxsown",maxxsown)
```

```
meansown [52.162532229770406, 49.094326979289185, 39.71986468766933, 35.74792616235197, 34.14484180169499, 30.7417221
33910724, 31.107312262988245, 30.14014908375965, 28.63336032702109, 28.477000236802457, 28.212532566316714, 27.034751
29855162]
minnsown [0.3305630847189503, 0.5223509617100421, 1.1920928955078125e-07, 0.19952352590828554, 0.14562118126270954, 0
.0, 0.1837191934279376, 0.0, 1.1920928955078125e-07, 0.0, 0.0, 0.0]
maxxsown [647426.7012010026, 653856.5593648385, 637828.3993235184, 627803.9302922419, 626884.6638306847, 519799.95134
42313, 608622.4849194715, 553804.5681877761, 604451.4001840391, 578938.6943853357, 614211.0488500809, 590151.53678244
64]
```

v. Now we need to compare the performance of three methods in just one chart.



vi.



Obviously, the method that have the best performance is using K-means++ to choose initial centroids.

vii. The error (mean distance) under different initialization scheme.

#k	5	10	50	100	150	200	250	300	350	400	450	500
Rand	54.0	52.4	40.4	36.2	34.1	28.5	34.1	27.2	25.2	29.2	26.8	25.3
Own	52.2	49.1	39.7	35.7	34.1	30.7	31.1	30.1	28.6	28.5	28.2	27.0
Kpp	69.8	54.5	29.9	18.7	17.1	14.1	14.9	11.4	11.3	11.5	9.74	9.99

The lowest error achieved by each initialization method is labeled red in the above figure. For random initialization, the minimum error is achieved when k is 350; for our own method, k is 500; for k-means++, k is 450. We could tell that k is quite large when minimum error is achieved.

In conclusion, from the figure and the data we get, the k-means++ combined with mini-batch method is the best method in this project, since the converging process is completed quickest and the error is lowest. And my partner thinks this is the reason that sklearn use K-means++.

For the completely random method (in problem2), the error curve is not stable and has many zig-zags, maybe the reason is K-means cluster algorithm depends much on the initial centroids, so the random initialization will be unstable. However, the K-means++ method to initial centroids is slow, maybe we need to upgrade the speed in the future.
.