Assignment 1

Yiyan Ge 9/17/2016

Part 1. Clustering: the baseline

Prepare data

Original data set is 1 million. The following code is used to extract 100K from the total sample.

```
with open('data/tweets_1M.json','r') as f:
    tweets = json.load(f)

X = np.array([[tweets[x]['lat'],tweets[x]['lng']] for x in range(0, len(tweets))])
sample = 100000
total = len(X)
X = X[0::int(total/sample)]
```

Reference time of clustering of 100K samples into k=100 clusters using k-means

Reference time is 141 seconds.

```
n = 100
k_means = KMeans(init='k-means++', n_clusters=n, n_init=10)
t_km = time.time()
k_means.fit(X)
t_fin_km = time.time() - t_km
print (t_fin_km)
```

Reference time of clustering of 100K samples into k=100 clusters with mini-batch k-means.

To select a propropriate batch_size, the following code is used to roughly look at the relationship between `batch size` and processing time.

```
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step

for perc in frange(0.01,0.11,0.01):
    batch_size=int(len(X)*perc)
    mbk = MiniBatchKMeans(init='k-means++', n_clusters=100, batch_size=batch_size,
```

```
                          n_init=10, max_no_improvement=10, verbose=0)
    t0 = time.time()
    mbk.fit(X)
    t_mini_batch = time.time() - t0
```

Simplified results follow:

Percentage	batch_size	Seconds to run 100 clusters
1%	1000	0.60
5%	5000	1.65
10%	10000	3.04

Maximum number of clusters k_max that the implementation of k-means can handle

Via some tests, n = 370 seems to be a reasonable starting number. The following code is implemented to get k_max. Processing time threshold is set to approximately 60 seconds. k_max = 36.

```
n = 35
t_fin_km = 0
while t_fin_km <= 60:
    print ('testing n equal to ' + str(n))
    ## initialize with K-means++, a good way of speeding up convergence
    k_means = KMeans(init='k-means++', n_clusters=n, n_init=10)
    ## record the current time
    t_km = time.time()
    # start clustering!
    k_means.fit(X)
    ## get the time to finish clustering
    t_fin_km = time.time() - t_km
    print (t_fin_km)
    n += 1
```

Maximum number of clusters k_max that the implementation of minibatch k-means can handle

Three batch_size (1000, 5000, 10000) were run to see how k_max changes. perc and n in the following code can vary as needed.

```
perc = 0.01
batch_size=int(len(X)*perc)
n = 2500
t_mini_batch = 0
while t_mini_batch <= 60:
```

```
    print ('testing n equal to ' + str(n))
    mbk = MiniBatchKMeans(init='k-means++', n_clusters=n, batch_size=batch_size,
                          n_init=10, max_no_improvement=10, verbose=0)
    t0 = time.time()
    mbk.fit(X)
    t_mini_batch = time.time() - t0
    n += 50
    print (t_mini_batch)
```

Results follow:

Percentage	batch_size	k_max
1%	1000	1350
5%	5000	365
10%	10000	175

One reason for the performance bottleneck caused by `k_max` is that cluster possibilities increase exponentially as number of clusters goes up.

Find eps_100 that resultes in 100 clusters with MinPts =100 as well as the corresponding processing time.

eps_100 = 0.001 (0.002 and 0.003 also produces 100 clusters)

```
eps = 0.05
n_clusters_ = 0
while n_clusters_ <= 100:

    print (eps)

    t_db = time.time()
    db = DBSCAN(eps=eps, min_samples=100).fit(X)
    t_fin_db = time.time() - t_db

    #array of numbers, one number represents one cluster
    db_labels = db.labels_
    # minus if there are unclustered noises
    n_clusters_ = len(set(db_labels)) - (1 if -1 in db_labels else 0)

    eps += 0.01
    print (t_fin_db, n_clusters_)
```

Part 2. Clustering: scalability

KMeans Scalability and Estimation

Essentially we are looking at how processing time increases as 1) number of clusters increases and 2) sample size increases.

The following (Figure 1) experiments with the relationship between number of clusters and processing time.

Given the 100K sample, computational time is estimated to be 160 seconds when number of clusters is 100, which is proved to be relatively accruate given the previous test (reference time around 141 seconds when number of clusters is 100).

The first figure in the four images (Figure 2) shows the relationship between processing time and sample size using k_means. Since `n_clusters` is fixed to 100 in this experiment. We don't have to scale up along this dimension. We only consider how sample size increases processing time. In order to roughly estimate how long 100 million samples will take to generate 100 clusters, three fitted lines were added shown in the rest three figures.

The first one assumes the simplest linear relation ship with polynomial degree of 1 and the computational time is estimated to be 143 seconds as sample size approaches 100 million. It is underestimated because, as shown in previously, 100K samples with 100 clusters require 141 seconds processing time already.

The bottom left figure uses linear relationship with polynomial degree of 2 and the predicted processing time is 446 seconds, which seems more reasonable.

The last figure shows how exponential relationship fits with the data. It looks similar on the graph but the estimate time is close to 3018 seconds.

MiniBatch KMeans Scalability and Estimation

We also consider how processing time increases as 1) number of clusters increases and 2) sample size increases using MiniBatchKMeans. However, MiniBatchK-Means also has another variable, `batch_size` which can also influence processing time.

Figure 3 shows how compuational time increases as `n_clusters` increases by different `batch_size`. It actually reveals similar information we see before: given the same `n_clusters`, higher `batch_size` increases processing time. Nonetheless, lower `batch_size` allows larger `k_max` (largest number of clusters produced within 60 seconds).

The follwoing adopts even lower `batch_size` to speed up process. Figure on the left (Figure 4) shows roughly how `batch_size` has an impact on processing time. One noticeable pattern is that the linear relationship between processing time and sample size is much less stable as `batch_size` decreases. Also, with smaller sample sizes, smaller `batch_size` takes longer processing time. In order to make estimation, `batch_size` is fixed to be 10000. Given the fitted line in

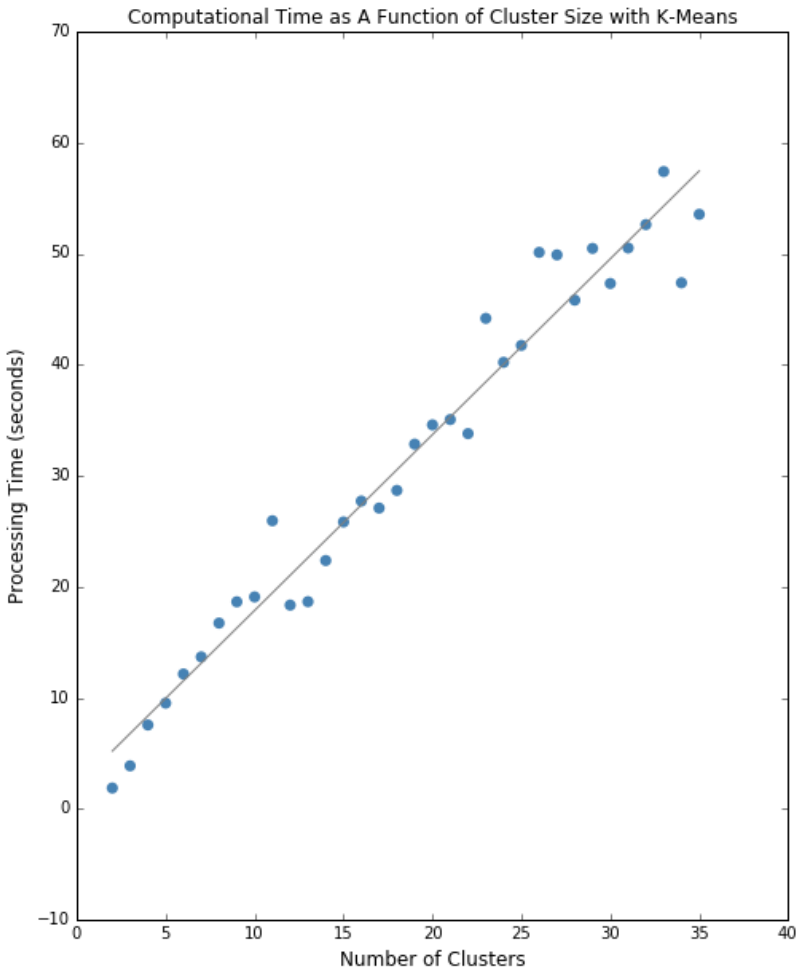

Figure 1: Computational time as a function of `n_clusters` (consider the range of 2 to the `k_max`) with k_means

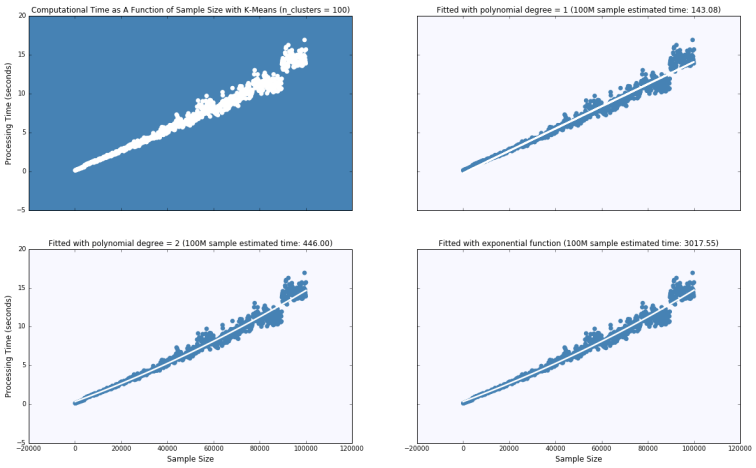

Figure 2: Computational time as a function of sample size for a fixed k=100 with k_means

the figure on the right, processing time is estimated to be around 3.62 seconds if sample size hits 100 million.

DBSCAN Scalability and Estimation

Original data were converted into unit of meter so that eps is easily interpreted. Given eps=100m, processing time as a function of sample size are plotted with two types of fitted line (Figure 5). Figure below indicates that darker blue fitted line seems to better represent the relationship between processing time and sample size. Given this, processing time to produce at least 100 clusters is estimated to be around 15.45 seconds when sample size is 100 million.

Part 3. Clustering: 1 million samples problem

Two variables, `batch_size` and `n_clusters` are adjusted in the problem to find minimal processing time to detect clusters of tweets that correspond to important locations in California (defined by `min_samples`=100 and `eps`=100).

1. Given previous experiments, `n_clusters` is set to be lower than 100 and two `batch_size` are tested. Results are shown in Figure 6. We can see that lower processing time occurs in the middle range of `n_clusters`.
2. `n_clusters` is constrained to be between 50 and 80 as shown below. One more larger `batch_size` is added also to understand better how different

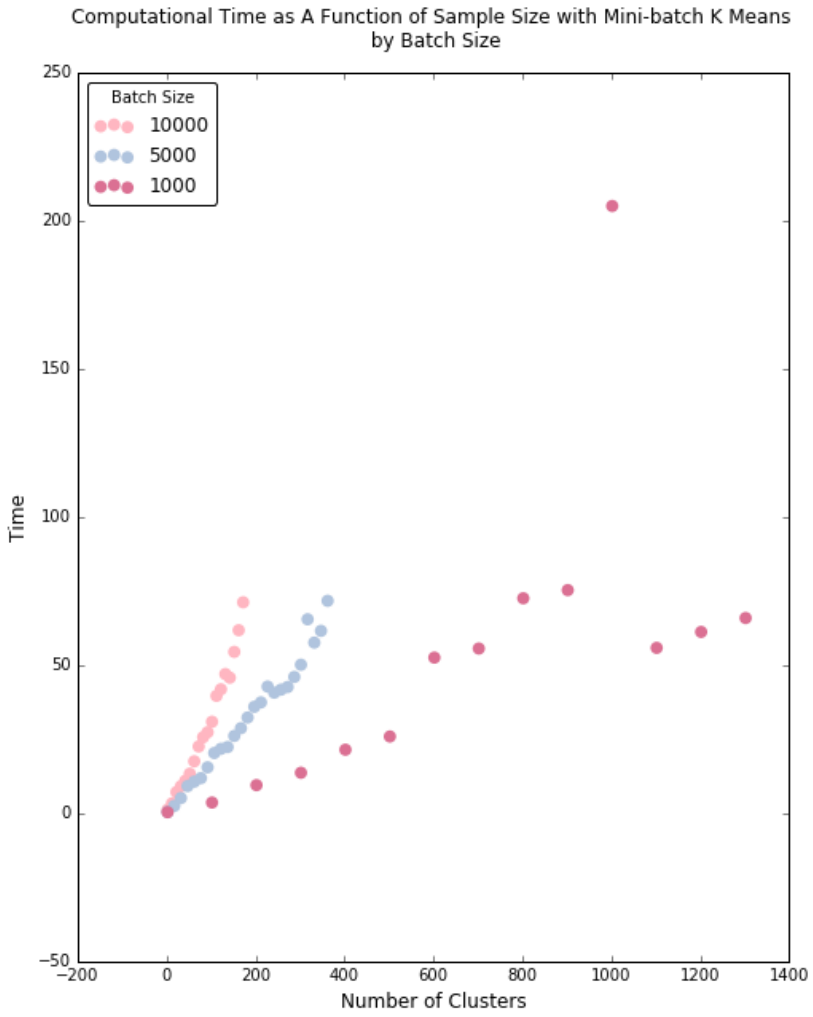

Figure 3: Computational time as a function of `n_clusters` with MiniBatchK-Means

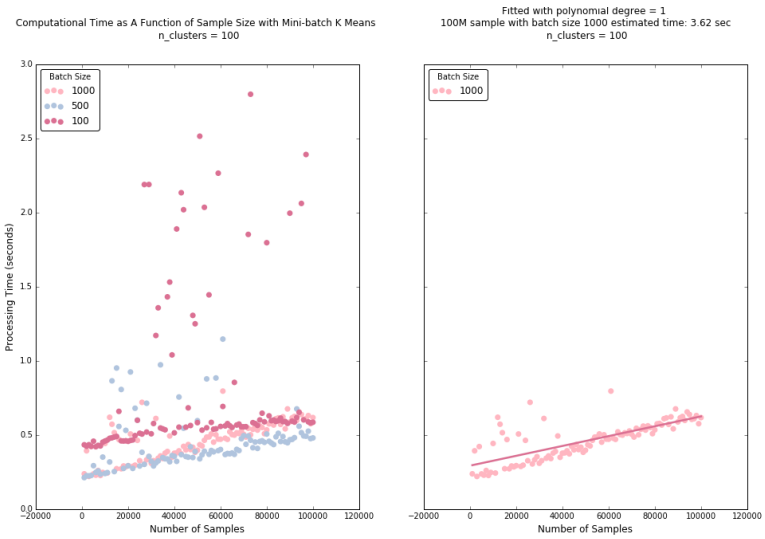

Figure 4: Computational time as a function of sample size for a fixed k=100 with MiniBatchKMeans

`batch_size` performs. Results (Figure7) show that larger `batch_size` seems to perform better. One potential reason is that better results (caused by larger and more representative `batch_size`) can improve DBSCAN later. Finer level `n_clusters` needs to be run to see how processing time varies.

3. Four `batch_size` are run in this step and we see that having even larger `batch_size` starts to negatively affect processing time. The best performing set of variables is `n_clusters`=64 and `batch_size`=2000. Processing time is around 16.14 seconds.

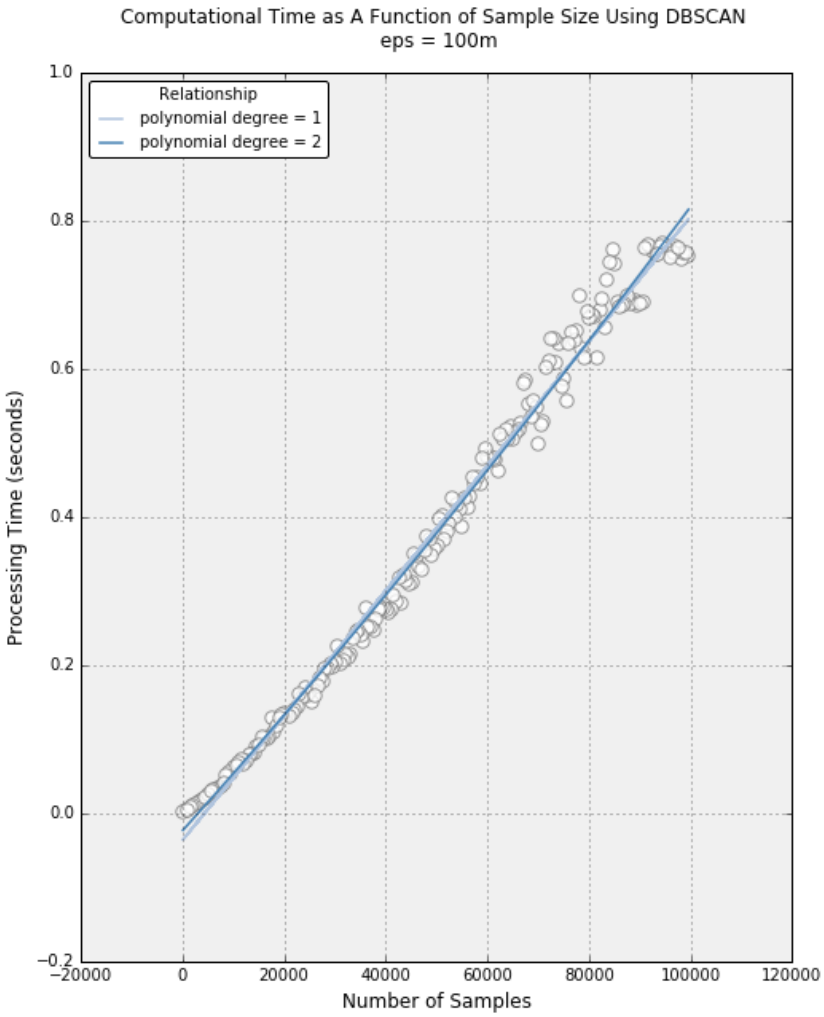

Figure 5: Computational time as a function of sample size for a fixed eps=0.01m with DBSCAN

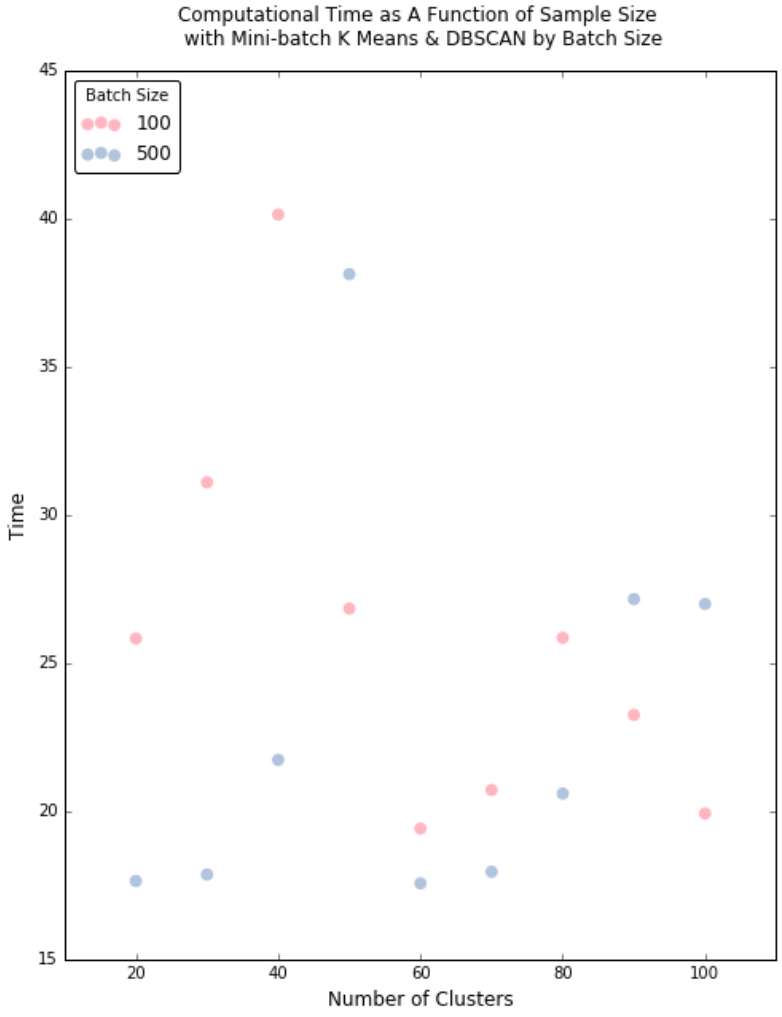

Figure 6: Step1

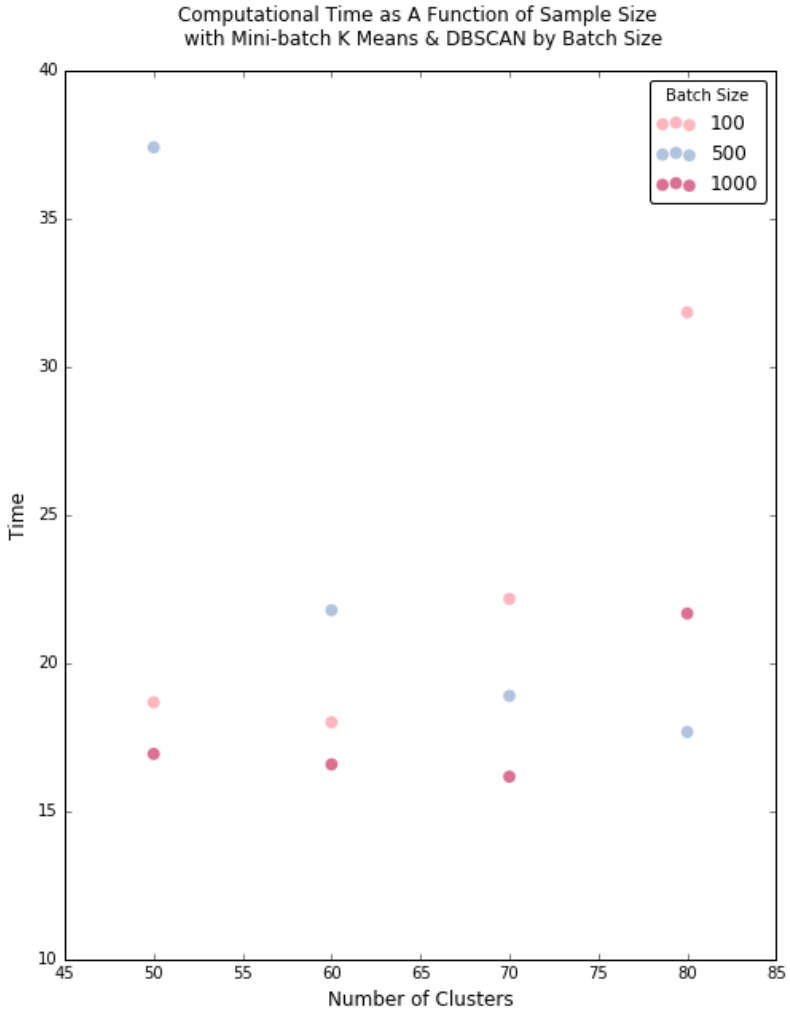

Figure 7: Step2

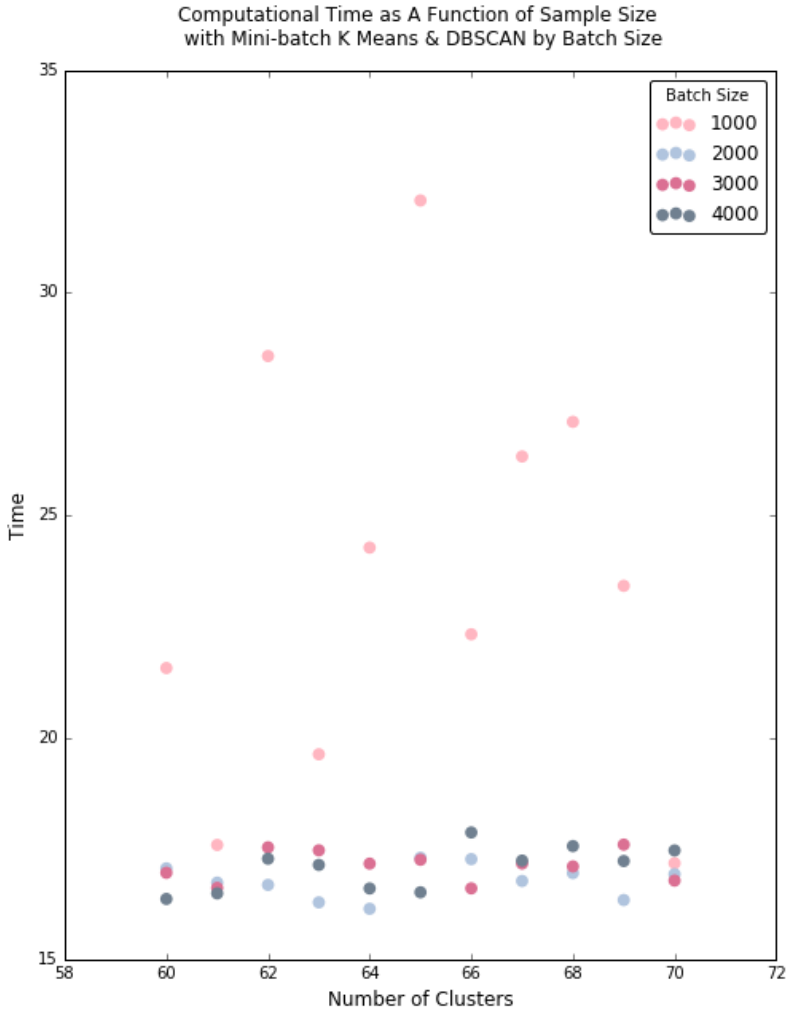

Figure 8: Step3