Problem1_writeup_Hill

September 21, 2016

L. Drew Hill CE263 Problem # 1 September 15, 2016

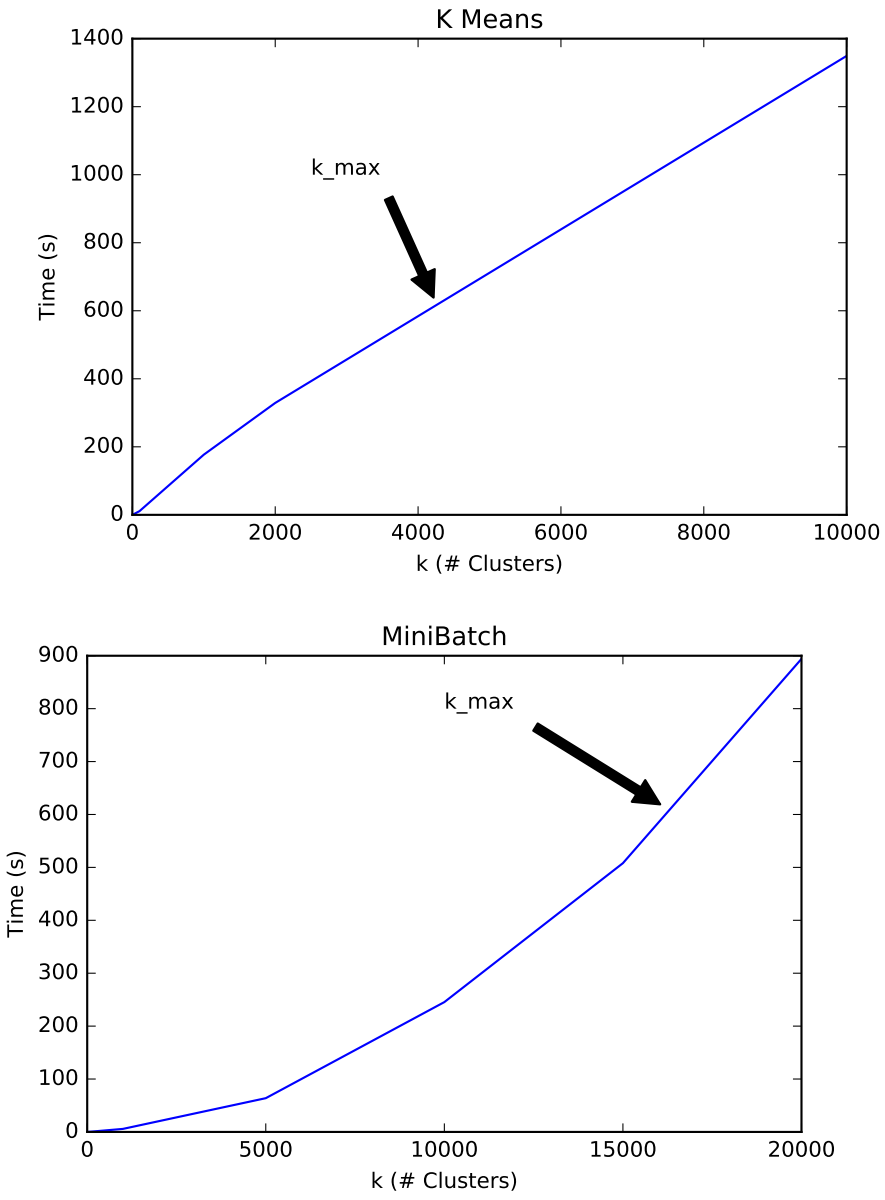
0.1 Part 1

A series of k-means and MiniBatch k-means processes were applied to the Latitude (km) and Longitude (km) data of 100,000 tweets randomly selected from the 1 million tweets provided to the class. Cooridnates were converted to relative KM value to allow for easier interpretation.

In [ ]:
```
## randomly sample 100k datapoints from the list of json files ('tweets')
# create index that randomly sample the data index 'n' times
k = 100000
random_index = random.sample(range(len(tweets)), k)

# reorder the index, then take each indexed object from the list
dt = [tweets[i] for i in sorted(random_index)]

# convert list of dictionaries to data frame
df = pd.DataFrame(dt)

# make time a datetime object
datetimer = lambda x: parse(time.strftime('%Y-%m-%d %H:%M:%S', time.strptim
df['timeStamp'] = df['timeStamp'].apply(datetimer)

# make numeric datetime
df['timeNum']=df['timeStamp']
numericizer = lambda x: time.mktime(x.timetuple())
df['timeNum'] = df['timeNum'].apply(numericizer)

for i in range(0,len(df)):
        df.at[i,'lat_km'] = df.at[i,'lat'] * 89.7
        df.at[i,'lng_km'] = df.at[i,'lng'] * 112.7
```

A batch size of 1%, or 1000 tweets, was selected for the MiniBatch test. Processing for 100 clusters (k=100) required 10.9 seconds for the standard k-means routine, but only 0.5 seconds for a Mini Batch k-means. I experimented with the computational/processing load of each routine by running the k-means process at k = [2, 100, 1000, 2000, 10000] and the Mini Batch k-means at k = [2, 1000, 5000, 10000, 15000, and 20000] (note: the increased performance of the MiniBatch test at lower values of k allowed me to test a wider range within a reasonable amount of time). Neither

routine appeared to be constrained by Memory– both consistently occupied ~ 1gb of ram along all values of k. Processing time, however, quickly became unbearably slow (Figure 1).



** Figure 1 ** Time (s) plotted as a function of k (# of clusters) for k-means (top) and MiniBatch (bottom) algorithms.

```
In [ ]: ########################################
        ######## Clustering with k-means example

        # Subset dataframe to lat (km) and lng (km)
        X = df[[7,8]]
        # set cluster number
        n = 100
        ## 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
```

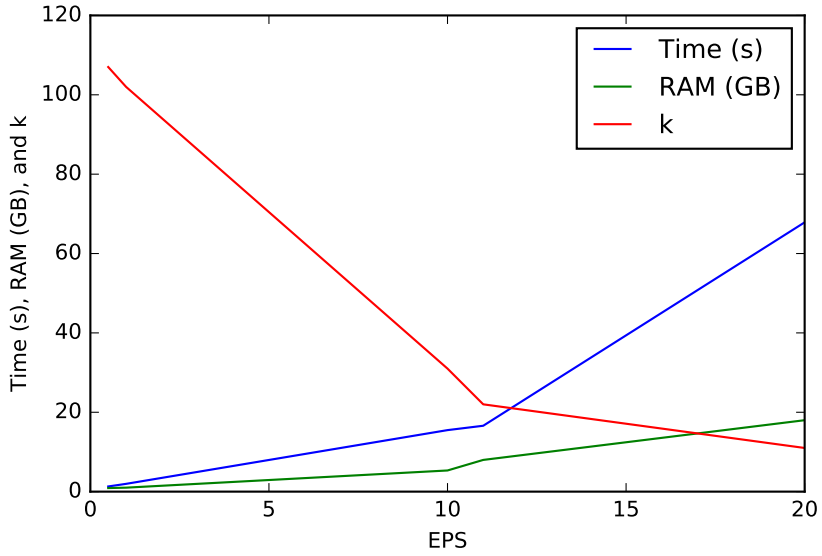
```
t_km = time.time()
# start clustering! using only lat and long (columns 2 and 3)
k_means.fit(X)
# results
k_means_labels = k_means.labels_
k_means_cluster_centers = k_means.cluster_centers_
k_means_labels_unique = np.unique(k_means_labels)
ft = (k_means_labels, k_means_cluster_centers, k_means_labels_unique)

# get the time to finish clustering
t_fin_km = time.time() - t_km
print(t_fin_km)

## Keep track of metrics as I iterate through values of "k"
df_k_n = np.array([2.0,100.0,1000.0,2000.0,10000.0])
df_k_time = np.array([.1,10.9,176.9,329,1349.3])
df_k_ram = [974, 980,975]

df_time = pd.DataFrame({'n':df_k_n, 'sec': df_k_time})

plt.plot(df_time['n'],df_time['sec'])
plt.ylabel('Time (s)')
plt.xlabel('k (# Clusters)')
plt.annotate('k_max', xy=(4300, 600), xytext=(2500, 1000),
             arrowprops=dict(facecolor='black', shrink=0.1),
             )
plt.title('K Means')

# save figure
pylab.savefig('kmeans_k_plot.pdf')
```

In [ ]:
```
######################################
######## Clustering with MiniBatch example

# Subset dataframe to lat (km) and lng (km)
X = df[[7,8]]

# number of clusters
n= 2
batch_size = 1000

mbk = MiniBatchKMeans(init='k-means++', n_clusters = n, batch_size=batch_si
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
mbk.fit(X)
t_mini_batch = time.time() - t0
mbk_means_labels = mbk.labels_
mbk_means_cluster_centers = mbk.cluster_centers_
```

```
mbk_means_labels_unique = np.unique(mbk_means_labels)

print(t_mini_batch)

## Keep track of metrics as I iterate through values of "k"
df_mb_n = [2,100,1000,5000,10000,15000,20000]
df_mb_time = [0.05697,0.5,5.74,63.8,245.4,507.98,894.1]
df_mb_ram = [975,974, 990,990,990,1000,1000]

df_time_mb = pd.DataFrame({'n':df_mb_n, 'sec': df_mb_time, 'mb': df_mb_ram}

## Plot metrics
plt.plot(df_time_mb['n'],df_time_mb['sec'])
plt.ylabel('Time (s)')
plt.xlabel('k (# Clusters)')
plt.title('MiniBatch')
plt.annotate('k_max', xy=(16500, 600), xytext=(10000, 800),
             arrowprops=dict(facecolor='black', shrink=0.1),)
## Save plot
pylab.savefig('MiniBatch_k_plot.pdf')
```

For the purposes of this assignment, I set a k_max for each routined as determined by my computer's performance of each routine. Based on my own impatience, I set a time limitation of 600 seconds, which resulted in a k-means k_max of ~ 4500 and a MiniBatch k-means k_max of about 16500. This was determined graphically by plotting the results of the aforementioned expirment (Figure 1). I believe this performance bottleneck is related to the number of mathematical computations required to calculate the distance from each point to each centroid. As the number of clusters increase, so too do the number of centroids. This causes a linear increase in computation time in the k-means test, which calculates this distance for every point until optimal cluster centroids are established and, ultimately, increases computational time until it is unbearable (> 600 s). However, because the Mini Batch test takes random cluster samples (batches), the number of distance calculations is substantially smaller at lower values of k. As the number of clusters increases, I imagine the permutations of 'closest cluster' increase exponentially, causing a non-linear increase in distance calculations as k increases. I believe this may be why MiniBatch becomes increasingly bottle-necked by performance as k increases. I believe ram is barely affected by either of these routines, because distances can be stored in very efficient arrays or tables.

Unlike the other two routines, DBSCAN was a major memory hog. When running DBSCAN on 100,000 randomly selected tweets with a minimum sample number of 100 and ε of 10 took about 15.5 seconds but consumed over 5GB of my system's 16GB of ram according to Activity Monitor. With a bit of experimentation at ε = [0.5, 1, 10, 11, 20], I determined the ε that would produce 100 clusters is approximately ε = 1 (Figure 2), with a corresponding processing time of 1.8 seconds.



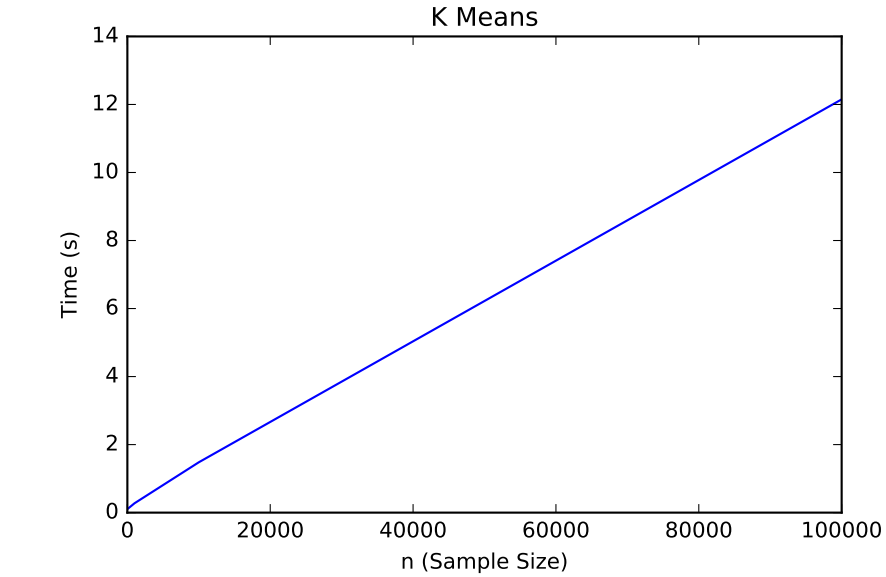
** Figure 2 ** Time (s), RAM use (GB), and k (# of clusters) plotted as a function of ε for several DBSCAN runs.

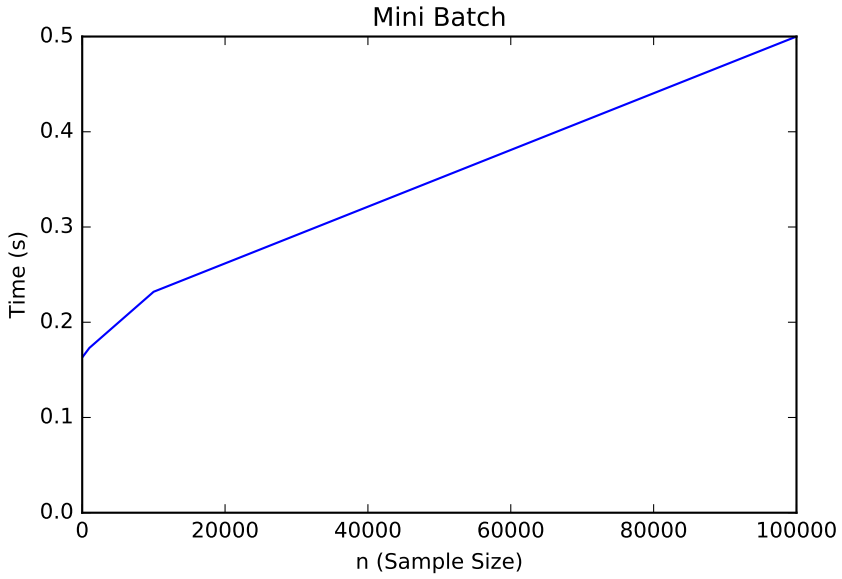
```
In [ ]: ########################################
        ########  DBSCAN example

        t_db = time.time()

        db = DBSCAN(eps=1, min_samples=100).fit(X)

        t_fin_db = time.time() - t_db

        ## results
        db_labels = db.labels_
        core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
        core_samples_mask[db.core_sample_indices_] = True
        db_labels_unique = np.unique(db_labels)
        print(t_fin_db)
        print(len(db_labels_unique))

        ## Keep track of metrics as I iterate through EPS values
        df_db_eps = [0.5,1.0,10.0,11.0,20.0]
        df_db_time = [1.3,1.96,15.5,16.6,67.86]
        df_db_ram = [.9 ,1.000,5.330,7.990,18.000]
        df_db_k = [107.0,102.0,31.0,22.0,11.0]

        df_time_db = pd.DataFrame({'eps':df_db_eps, 'sec': df_db_time, 'gb': df_db_

        ## Plot metrics
        plt.plot(df_time_db['eps'],df_time_db['sec'], label = 'Time (s)')
        plt.plot(df_time_db['eps'],df_time_db['gb'], label = 'RAM (GB)')
        plt.plot(df_time_db['eps'],df_time_db['k'], label = 'k')
```

```
plt.ylabel('Time (s), RAM (GB), and k')
plt.xlabel('EPS')
pylab.legend(loc='upper right')

# save plot
pylab.savefig('DBSCAN_eps_plot.pdf')
```

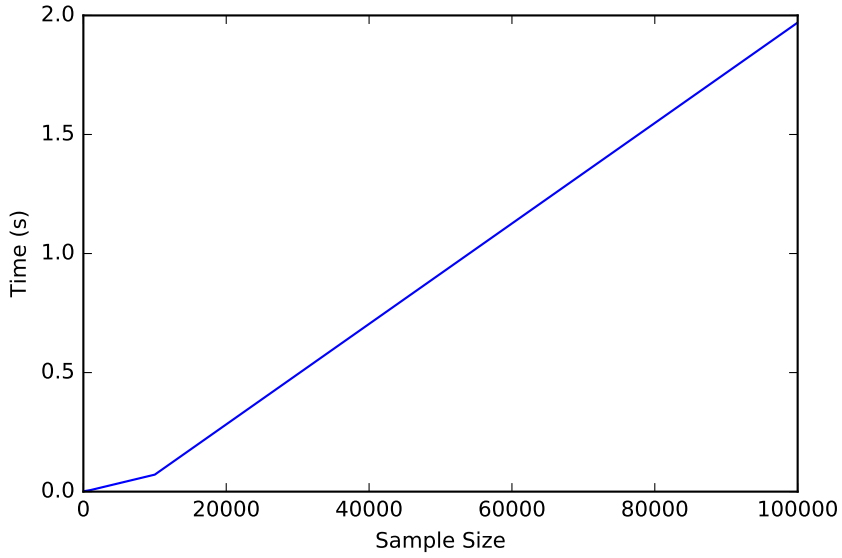
0.2 Part 2

As shown in Figure 1, processing time was plotted as a function of k for both the k-means (k = [2, 100, 1000, 2000, 10000]) and MiniBatch (k = [2, 1000, 5000, 10000, 15000, and 20000]) routines. Both produced a relatively linear relationship between time and k. The relationship between time and sample size was also examined where n = [100, 1000, 10000, 100000] with a fixed k of 100 for the k-means and MiniBatch routines, a fixed batch size of 1% of sample size for the MiniBatch routine, and a fixed ε of 1 (corresponding to production of about 100 clusters under a sample size of 100000 as previously demonstrated) and a fixed MinPts of 100 for the DBSCAN routine. The results are shown in Figure 3. By far, the fastest of all routines under their specific setups was MiniBatch, though DBSCAN came in at a close second. K-means was considerably slower (roughly ~ 10 times). Linear curves were fit to the time ~ n relationship of each routine, and then extrapolated to estimate the amount of processing time required at n = 1 million under the previously stated conditions. * K-Means: Time ~ 0.00011982 * n (r2 = 1.00) * Time ~ 0.00011982 * 1000000 * Time ~ 120 seconds * MiniBatch: Time ~ 3.256e-06 * n (r2 = 0.98) * Time ~ 3.3 seconds * DBSCAN: Time ~ 2.0069 e-05 * n (r2 = 1.00) * Time ~ 20.1 seconds





** Figure 3 ** Time (s) plotted as a function of sample size for several k-means and Mini Batch k-means runs.



** Figure 4 ** Time (s) plotted as a function of sample size at ε=1 for several DBSCAN runs.

```
In [ ]: ## Function to create random samples of number "i" and convert lat to km
        for i in [100, 1000, 10000, 50000, 100000]:
            random_index = random.sample(range(len(tweets)), i)
            globals()['df_%s' % i] = [tweets[i] for i in sorted(random_index)]
            for x in range(0,i):
                globals()['df_%s' % i][x]['lat_km'] = globals()['df_%s' % i][x]['la
                globals()['df_%s' % i][x]['lng_km'] = globals()['df_%s' % i][x]['ln
            globals()['df_%s' % i] = pd.DataFrame(globals()['df_%s' % i])


        ################################
        ###### k-means example

        # set cluster number
        k = 100

        # set dataset to df_n and subset to include only lat (km) and lon (km)
        X = df_100000[[2,4]]

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

        print(t_fin_km)

        df_k_n = np.array([100, 1000, 10000, 100000])
        df_k_time = np.array([ 0.12 , 0.275 , 1.48 , 12.15])
```

```
df_time = pd.DataFrame({'n':df_k_n, 'sec': df_k_time})

plt.plot(df_time['n'],df_time['sec'])
plt.ylabel('Time (s)')
plt.xlabel('n (Sample Size)')
plt.title('K Means')


pylab.savefig('kmeans_n_plot.pdf')

# fit curve to data
regr = linear_model.LinearRegression()
regr.fit(df_k_n.reshape(-1,1) ,df_k_time.reshape(-1,1))
print('Coefficients: \n', regr.coef_)
print('Vriance score: %.2f' % regr.score(df_k_n.reshape(-1,1) ,df_k_time.re


##############################
###### MiniBatch example

# set dataset to df_n and subset to include only lat (km) and lon (km)
X = df_100000[[2,4]]

batch_size = 1000

# number of clusters
k= 100

mbk = MiniBatchKMeans(init='k-means++', n_clusters = k, batch_size=batch_si
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
mbk.fit(X)
t_mini_batch = time.time() - t0
mbk_means_labels = mbk.labels_
mbk_means_cluster_centers = mbk.cluster_centers_
mbk_means_labels_unique = np.unique(mbk_means_labels)

print(t_mini_batch)

df_mb_n = np.array([100, 1000, 10000 ,100000])
df_mb_time = np.array([ 0.164, 0.173, 0.232 ,0.50])

df_time_mb = pd.DataFrame({'n':df_mb_n, 'sec': df_mb_time})

plt.plot(df_time_mb['n'],df_time_mb['sec'])
plt.ylabel('Time (s)')
plt.xlabel('n (Sample Size)')
pylab.ylim([0,.50])
```

```
pylab.xlim([0,100000])

plt.title('Mini Batch')
pylab.savefig('minibatch_n_plot.pdf')

# fit curve to data
regr = linear_model.LinearRegression()
regr.fit(df_mb_n.reshape(-1,1) ,df_mb_time.reshape(-1,1))
print('Coefficients: \n', regr.coef_)
print('Vriance score: %.2f' % regr.score(df_mb_n.reshape(-1,1) ,df_mb_time.


################################
###### DBSCAN example

# set dataset to df_n and subset to include only lat (km) and lon (km)
X = df_100000[[2,4]]

## DBSCAN
t_db = time.time()
db = DBSCAN(eps=1, min_samples=100).fit(X)
t_fin_db = time.time() - t_db

## DBSCAN Results
db_labels = db.labels_
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True

db_labels_unique = np.unique(db_labels)

print(t_fin_db)

# keep track of metrics
df_db_n = np.array([100, 1000, 10000, 100000])
df_db_time = np.array([0.0019,0.0065,0.0714,1.97])

df_time_db = pd.DataFrame({'n':df_db_n, 'sec': df_db_time})

# plot metrics
plt.plot(df_time_db['n'],df_time_db['sec'], label = 'Time (s)')
plt.ylabel('Time (s)')
plt.xlabel('Sample Size')
pylab.savefig('DBSCAN_n_plot.pdf')

# fit curve to data
regr = linear_model.LinearRegression()
regr.fit(df_db_n.reshape(-1,1) ,df_db_time.reshape(-1,1))
print('Coefficients: \n', regr.coef_)
```

```
print('Vriance score: %.2f' % regr.score(df_db_n.reshape(-1,1) ,df_db_time.
```

0.3 Part 3

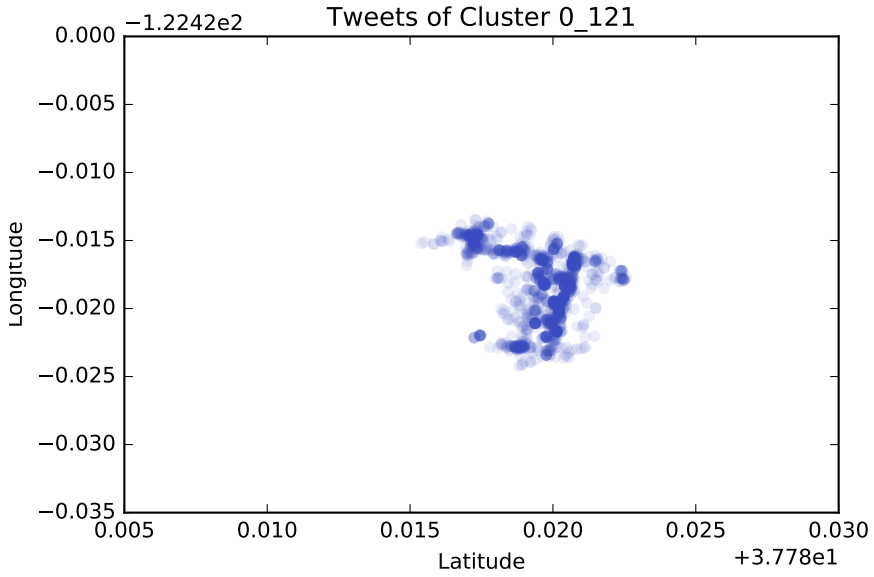
I approached this problem using the hierarchical method suggested by Alexei. In general, the approach followed these basic steps: 1. Full dataset is run through MiniBatch processing 2. Each MiniBatch cluster is run through DBSCAN with eps = 100m, and min_samples = 100. This was intended to simulate a cluster core of at least 100 samples within a radius of 100m.

Considering the computational processing and memory limitations analyzed in Part 2, I iterated through the MiniBatch step with k = [10, 25, 35, 50, 75, 100, 200] and a batch size of 1% of the sample (10000). These tests went very quickly, and so I pushed the batch size to 10% (100000) to produce the following computational times at k = [10, 25, 35, 50, 75, 100, 200], t = [2.51, 5.96, 8.40, 12.35, 28.00, 61.80] in seconds. A tolerable sweet spot seemed to be less than a minute, and so I limited my k options (at a batch size of 100000) to 10-100. This also seemed to produce a reasonable sample size for DBSCAN, with a minimum of 1000 datapoints at k= 100 clusters, if divided evenly among the samples. The MiniBatch output produced by the k range of 10-100 was then processed through DBSCAN at EPS_100 = 0.1 and min_samples = 100. For timing considerations, a truncated set was used, k = [10, 25, 50, 100]. At each k, the following was observed: * 10 MiniBatch: 1607 assigned clusters, 22.82 sec * 25 MiniBatch: 1604 assigned clusters, 28.27 sec * 50 MiniBatch: 1605 assigned clusters, 37.89 sec * 100 MiniBatch: 1606 assigned clusters, 58.60 sec * 200 MiniBatch: 1614 assigned clusters, 104.67 sec (included for fun/comparison, even though out of my pre-defined range of k)

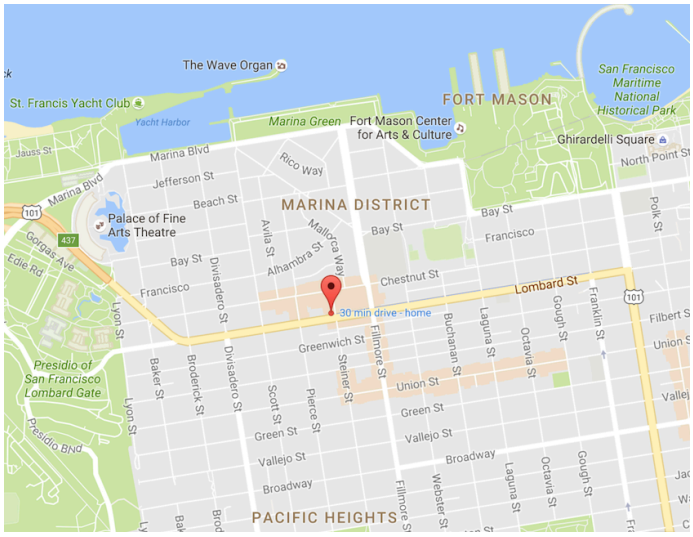
Loading on RAM did not exceed approximately 2.3 GB, and so was not a major consideration at this range of parameters. With these considerations in mind, I believe the following hierarchical clustering parameters to be optimal: a MiniBatch with k = 10 and a batch size of 100,000, followed by DBSCAN on each resulting cluster with EPS_100 = 0.1 and min_samples = 100. This set produced almost exactly the same number of assigned clusters (1607) as all larger k values examined at this batch size. (This excludes any unassigned clusters labelled "-1" during the DBSCAN routine.)

0.3.1 Extra Credit Visualization

I quasi-randomly decided to examine a cluster named 0_121 – which means it was sub-cluster number 121 as determined by DBSCAN of the first MiniBatch cluster (0). This cluster falls within the latitudes of 37.795 to 37.803 (median: 37.7998) and the longitudes of -122.444 to -122.433 (median: -122.437939). The region it covers spans about 1.21 km across and about 0.64 km up and down the map (using the rough coordinate conversions suggested in the assignment). Tweet coordinates ar eplotted in Figure 5. Based on these coordinates, the cluster is likely a representation of the Marina District in San Francisco. Based on the shape of the cluster, I would posit these tweets come from the Lombard St / Chestnut St @ Filmore Street shopping and dining district in the Marina (Figure 6). Examining the text of the tweets in this cluster, one major theme emerges: celebration, with subtopics in drinking and eating. This must be a dining region. Throughout the range of dates/times these tweets cover (Sep 12 - Oct 5), a good number of sports events appear to be commented upon as well. I believe the linguistic content of the tweets confirms that this cluster is, indeed, home to many Marina Bros.



** Figure 5 ** Cluster "0_121" plotted in terms of latitude and longitude.



** Figure 6 **
The central Lat and Long coordinates (the red pin) of cluster "0_121" as shown on Google Maps.

** Figure 7 ** Some of the most commonly used words tweeted at this location. Size indicates relative frequency. Created at Wordle.

Standalone Script for Part 3

```
In [ ]: # Load packages
        import time,datetime
        import pylab
        import json
        from sklearn.datasets import load_iris
        from sklearn.cluster import KMeans, DBSCAN, MiniBatchKMeans
        from scipy.cluster.hierarchy import dendrogram, linkage
        from sklearn.metrics.pairwise import pairwise_distances_argmin
        from sklearn.datasets.samples_generator import make_blobs
        from sklearn import linear_model
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy
        import scipy.io
        import pandas as pd
        import math
        import random
        from mpl_toolkits.mplot3d import Axes3D
        from dateutil.parser import parse
        from sklearn.metrics import silhouette_samples, silhouette_score

        # setting to show the plots in line in the notebook
        %matplotlib inline

        # ignore warnings
        import warnings
        warnings.filterwarnings('ignore')

        # Load Tweet Dataset
        with open('tweets_1M.json','r') as f:
```

```
    tweets=json.load(f)

# Convert tweets to data frame
twits = pd.DataFrame(tweets)

# convert lat long
for i in range(0,len(twits)):
        twits.at[i,'lat_km'] = twits.at[i,'lat'] * 89.7
        twits.at[i,'lng_km'] = twits.at[i,'lng'] * 112.7

# confirm column values
twits.columns.values

# subset to include only lat (km) and long (km) columns
X = twits[[6,7]]

################################
##### Inputs for experimentation
################################
# change with each iteration
n= 100
batch_size = 100000

# start timer
t_total_0 = time.time()

############
## MiniBatch K Means
mbk = MiniBatchKMeans(init='k-means++', n_clusters = n, batch_size=batch_si
                      n_init=10, max_no_improvement=10, verbose=0)
t0 = time.time()
mbk.fit(X)
t_mini_batch = time.time() - t0
mbk_means_labels = mbk.labels_
mbk_means_cluster_centers = mbk.cluster_centers_
mbk_means_labels_unique = np.unique(mbk_means_labels)

print(t_mini_batch)

## Add labels back to the original dataset
twits['label']= mbk_means_labels
# twits for each MiniBatch cluster, then remove label
for i in range(1,n):
    globals()['t%s' % i] = twits[['lat_km','lng_km','label']].query('label

# keep track of metrics
MB_hier_k = np.array([10, 25, 35, 50, 75, 100, 200])
MB_hier_sec = np.array([2.51, 5.96, 8.40, 12.35, 20.15, 28.00, 61.80])
```

```
###############
## DBSCAN

# function to run dbscan on twits by 'label' value; store new dataframes ir
df_list = {}
for i in range(0, n ):
    # split/subset by MiniBatch label
    df_interim = twits[['lat_km','lng_km','label']].query('label == %s' % i
    # remove label column
    df_interim = df_interim[['lat_km','lng_km']]
    # run DBSCAN
    db = DBSCAN(eps=0.1, min_samples=100).fit(df_interim)
    # create column for label and DBSCAN group
    df_interim['label_db'] = db.labels_
    df_interim['db_group'] = '%s' % i
    # create a list of all dataframes, clusters 0 - (n-1)
    df_list['df_db%s' % i] = df_interim

# recombine by concatenating the df_list into one final dataframe
df_final = pd.concat(df_list)

# Paste Label and DBSCAN Group values together to create unique
# cluster label for each row
import functools
def reduce_concat(x, sep=""):
    return functools.reduce(lambda x, y: str(x) + sep + str(y), x)

def paste(*lists, sep=" ", collapse=None):
    result = map(lambda x: reduce_concat(x, sep=sep), zip(*lists))
    if collapse is not None:
        return reduce_concat(result, sep=collapse)
    return list(result)

df_final['cluster'] = paste(df_final.db_group,df_final.label_db, sep = "_")

## Count all labels (a.k.a. unassigned clusters)
print("Total # of Clusters:", df_final.cluster.nunique())

## Count labels, excluding "-1" (a.k.a. unassigned clusters)
print("Total # of assigned Clusters:" , df_final.query('label_db != -1').cl


# End time
t_total = time.time() - t_total_0
print("Total runtime for MiniBatch of ",n, "samples:" , t_total)
```

Extra credit code

```
In [ ]: ### Convert lat/lng back
        df_final['lat'] = df_final['lat_km']/89.7
        df_final['lng'] = df_final['lng_km']/112.7

        ## Chose a cluster
        grouped = df_final[['cluster','lat']].groupby('cluster')
        grouped.count()

        ## I chose 0_121, as it has a fair number of samples (~1650)
        df_clust = df_final.query('cluster == "0_121"')

        ### Describe lat and long
        df_clust.describe()


        ### Plot it
        uniq = list(set(df_clust['cluster']))
        uniq
        # df_final['lat'].describe()
        import matplotlib.colors as colors
        import matplotlib.cm as cmx
        # determine number of unique clusters in Mill Valley area (10)
        print('Number of Clusters:',df_clust.cluster.nunique())
        uniq = list(set(df_clust['cluster']))
        # Set the color map to match the number of clusters
        z = range(1,len(uniq))
        cols = plt.get_cmap('coolwarm')
        cNorm  = colors.Normalize(vmin=0, vmax=len(uniq))
        scalarMap = cmx.ScalarMappable(norm=cNorm, cmap=cols)

        # plot each cluster
        for i in range(len(uniq)):
            indx = df_clust['cluster'] == uniq[i]
            plt.scatter(df_clust['lat'], df_clust['lng'], s=20, color=scalarMap.to_

        plt.xlabel('Latitude')
        plt.ylabel('Longitude')
        plt.title('Tweets of Cluster 0_121')
        # plt.legend(loc='center left', fancybox=True, shadow=True, bbox_to_anchor=
        plt.show()

        # save plot
        pylab.savefig('Cluster_plot.pdf')


        ### Manually examine tweets in this cluster
```

```
## subset full tweets dataset by geographic region
    # range:
        # lat 37.795 to 37.803
        # lng -122.444 to -122.433

df_marina = df.query('lat < 37.803 and lat > 37.795 and lng < -122.433 and
df_marina[['text','timeStamp']]
df_marina['cluster'] = str('0_121')

### Look at every bit of text in this cluster
# concatenate all text (row by row), separated by a ' '
df_wordle= df_marina.groupby(['cluster'])['text'].apply(lambda x: ' '.join

# output that concatenated text to clipboard for pasting into Wordle
df_wordle.to_clipboard()
```