MINJUNG_PARK_FINAL_PROJECT

April 30, 2015

```
In [3]: import os
        import sys
        import glob
        import matplotlib.pyplot as plt
        import matplotlib.cm as cm
        import numpy as np
        import pandas as pd
        %matplotlib inline
        %precision 4
        plt.style.use('ggplot')
```

1 STA 663 Project Outline

1.1 *Background Clustering*

Clustering is to assign a set of data points into clusters. It can be applied into the real life example. For instance, imagine you go to the IKEA. Items in the IKEA are usually well organized to be easily found. There are several sections we can choose; living room, Bedroom, Bathroom and so on. Also, a Bedroom part has some aspects like bedding, storage, etc. If we can quantify this qualitative data, these aspects for the Bedroom can be clustered into one categories. Clustering method can be useful in this case.

1.2 *Scalable K-means++*

Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar and Sergei Vassilvitskii

Clustering is one of the most important issues in data mining. Especially, K-means is considered as the most popular clustering method. K-means has an advantage on its simplicity, just starting with randomly chosen initial centers, repeatedly assigning each input point to its nearest center, and then recalculating the centers given the assigned points. However, K-means also has a disadvantage on its time efficiency and quality. In this paper, they describe a pallel version of K-means++ initialization algorithm and prove its efficiency. Also, the main idea of their paper is that they want to implement sampling O(k) points in each round and repeat the process for O(log n) rounds rather than just sampling a single point in each pass of the k-mean++ algorithm. At the end of the algorithm, they are left with O(klogn) points that form a solution that is within a constant factor away from the optimum and then recluster these points into k initial centers for the Lloyd's iteration. They called this initialization algorithm k-means||

k-means|| has several advantages. First, O(log n) iterations are not necessary. k-means|| can find better solution than any other meathod. Second, k-means|| is much faster than existing parallel algorithms for k-means. Third, the number of interations can be smallest.

1.3 *Outline of algorithm*

1.3.1 Algorithm 1, KMeans

Basically, the input of k-means algorithm is a dataset X (vectorized) of N points with a parameter K indicating how many clusters are there. The output of it is a set of K cluster centroids and a labeling of X

that assigns each of the points in X to a unique cluster. All points within a cluster are closer in distance to their centroid than other centroids.

In [2]:
```
def KMeans(data,centroids,k):

    converged = False
    cluster_values = []
    iterations = 0

    while (not converged) and (iterations < 1000):
        data_points = data[:, np.newaxis, :]
        # data_points n x k x m by broadcasting
        # To introduce the third dimension into data

        ## calculate the Euclidean distance between a centroid and a data point
        euclidean_dist = (data_points - centroids) ** 2
        sum_up_dist = np.sum(euclidean_dist, axis=2) # total distance over the 3rd axis (n x k)

        ## clustering, which cluster each data point belongs
        min_dist = np.zeros(sum_up_dist.shape) # = n x k
        min_dist[range(sum_up_dist.shape[0]), np.argmin(sum_up_dist, axis=1)] = 1
        # [i,j] = 1 in matrix (n x k) if the ith data point belongs to cluster j

        ## clusters
        cluster_val = np.sum(sum_up_dist[min_dist == True])
        cluster_values.append(cluster_val)

        # new centroids
        new_centroids = np.empty(centroids.shape)
        for j in range(0, k):
            if data[min_dist[:,j] == True,:].shape[0] == 0:
                new_centroids[j] = centroids[j]
            else:
                new_centroids[j] = np.mean(data[min_dist[:,j] == True, :], axis=0)
        # comparing centroids
        if np.array_equal(centroids,new_centroids):
            converged = True
        else:
            centroids = new_centroids

        iterations += 1
    print (iterations, 'iterations are required to converge.')
    return iterations, cluster_values, centroids, min_dist
```

In [14]:
```
## Unit testing

k = 3
# if all of inputs is correct, it works well.
data1 = np.random.randn(1000,2)
centroids1 = data1[np.random.choice(range(data1.shape[0]), k, replace=False),:]
print KMeans(data1,centroids1,k)

# if the input is an empty vector, it gives an error.
data2 = []
centroids2 = data1[np.random.choice(range(data1.shape[0]), k, replace=False),:]
```

```
        KMeans(data2,centroids2,k)

(11, 'iterations are required to converge.')
(11, [1362.1419180869309, 1118.6938971038817, 990.53102606943298, 940.17957292690141, 928.64624932636059
      [-0.7074,  0.7313],
      [ 1.0151,  0.1881]]), array([[ 1.,  0.,  0.],
      [ 0.,  0.,  1.],
      [ 1.,  0.,  0.],
      ...,
      [ 0.,  1.,  0.],
      [ 0.,  1.,  0.],
      [ 1.,  0.,  0.]]))


       ---------------------------------------------------------------------------
   TypeError                                 Traceback (most recent call last)

       <ipython-input-14-fcf3d672d7b9> in <module>()
         9 data2 = []
        10 centroids2 = data1[np.random.choice(range(data1.shape[0]), k, replace=False),:]
   ---> 11 KMeans(data2,centroids2,k)


       <ipython-input-2-25d79bca730b> in KMeans(data, centroids, k)
         6
         7     while (not converged) and (iterations < 1000):
   ----> 8         data_points = data[:, np.newaxis, :]
         9         # data_points n x k x m by broadcasting
        10         # To introduce the third dimension into data


       TypeError: list indices must be integers, not tuple
```

1.3.2 Algorithm 2, KMeans++

This is the algorithm to get initial centroids points. Firstly, choose 1 centroid randomly from the data. Then, use it to generate the next centroid until getting k number of centroids with the probablility distribution. Finally, this algorithm gives k centroids which will be used as the initial centroids for KMeans.

```
In [16]: def KMeansPlusPlus(data, k):
             # choose 1 centroid randomly from the data
             centroids = data[np.random.choice(range(data.shape[0]),1), :]
             data_points = data[:, np.newaxis, :]

             ## run k-1 passes
             while centroids.shape[0] < k :
                 # the process is the same as Kmeans
                 euclidean_dist = (data_points - centroids) ** 2
                 sum_up_dist = np.sum(euclidean_dist, axis=2)
                 min_dist = np.zeros(sum_up_dist.shape)
                 min_dist[range(sum_up_dist.shape[0]), np.argmin(sum_up_dist, axis=1)] = 1
                 cluster_val = np.sum(sum_up_dist[min_dist == True])

                 ## probability distribution
```

```
            prob_distribution = np.min(sum_up_dist, axis=1)/cluster_val

            ## choose the next centroid by using prob_distribution
            centroids = np.vstack([centroids, data[np.random.choice(range(data.shape[0]),1,p=prob_
        return centroids
```

In [18]:
```
## Unit testing

k = 3
# if all of inputs is correct, it works well.
data1 = np.random.randn(1000,2)
print KMeansPlusPlus(data1,k)

# if the input is an empty vector, it gives an error.
data2 = []
KMeansPlusPlus(data2,k)
```

```
[[-0.4407 -0.3963]
 [ 0.966   1.1351]
 [ 0.4302 -0.0434]]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-18-74447a50088c> in <module>()
  8 # if the input is an empty vector, it gives an error.
  9 data2 = []
---> 10 KMeansPlusPlus(data2,k)


<ipython-input-16-fca45e9de50b> in KMeansPlusPlus(data, k)
  1 def KMeansPlusPlus(data, k):
  2     # choose 1 centroid randomly from the data
----> 3     centroids = data[np.random.choice(range(data.shape[0]),1), :]
  4     data_points = data[:, np.newaxis, :]
  5


AttributeError: 'list' object has no attribute 'shape'
```

1.3.3 Algorithm 3, Scalable KMeans++

This is also the algorithm to get initial centroids points. Firstly, choose 1 centroid randomly from the data. Then, use it to generate the next centroid until getting a set of centroids which is more than k numbers with the probablility distribution made by the weights. After that, reduce this set of centroids which is higher than k into k numbers of centroids by using KMeans++. Finally, this algorithm gives k centroids which will be used as the initial centroids for KMeans.

In [20]:
```
def ScalableKMeansPlusPlus(data, k, l, r):
    centroids = data[np.random.choice(range(data.shape[0]),1), :]
    data_points = data[:, np.newaxis, :]
    passes = 0
```

```
            while passes < r:
                euclidean_dist = (data_points - centroids) ** 2
                sum_up_dist = np.sum(euclidean_dist, axis=2)
                # the minimum distance
                min = np.min(sum_up_dist, axis=1).reshape(-1,1)
                # random matrix with the same size as min
                random_matrix = np.random.rand(min.shape[0],min.shape[1])
                # replace zeros in min with the lowest positive float
                min[np.where(min==0)] = np.nextafter(0,1)
                # (1.0/min)th root of random matrix
                random_matrix = random_matrix ** (1.0/min)
                # choose the highest l
                center = data[np.argsort(random_matrix, axis=0)[:, 0]][::-1][:l, :]
                # combine the new centroids with the old one
                centroids = np.vstack((centroids, center))
                passes += 1
                # Finally we get a set of centroids which is higher than k
            ## reduce this to k using KMeans++
            euclidean_dist = (data_points - centroids) ** 2
            sum_up_dist = np.sum(euclidean_dist, axis=2)
            min_dist = np.zeros(sum_up_dist.shape)
            min_dist[range(sum_up_dist.shape[0]), np.argmin(sum_up_dist, axis=1)] = 1
            weights = np.array([np.count_nonzero(min_dist[:, i]) for i in range(centroids.shape[0])],d
            prob_distribution = weights/np.sum(weights)
            centroids = data[np.random.choice(range(weights.shape[0]),k,p=prob_distribution.ravel()),:]
            return centroids
```

In [24]:
```
## Unit testing

k = 3
l = 2
r = 2
# if all of inputs is correct, it works well.
data1 = np.random.randn(1000,2)
print ScalableKMeansPlusPlus(data1,k,l,r)

# if the input is an empty vector, it gives an error.
data2 = []
ScalableKMeansPlusPlus(data2,k,l,r)
```

```
[[-0.4768 -0.9257]
 [-0.3029  1.0657]
 [ 0.767   0.0622]]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

<ipython-input-24-2b56ba1db349> in <module>()
 10 # if the input is an empty vector, it gives an error.
 11 data2 = []
---> 12 ScalableKMeansPlusPlus(data2,k,l,r)


<ipython-input-20-16078a7eb03c> in ScalableKMeansPlusPlus(data, k, l, r)
```

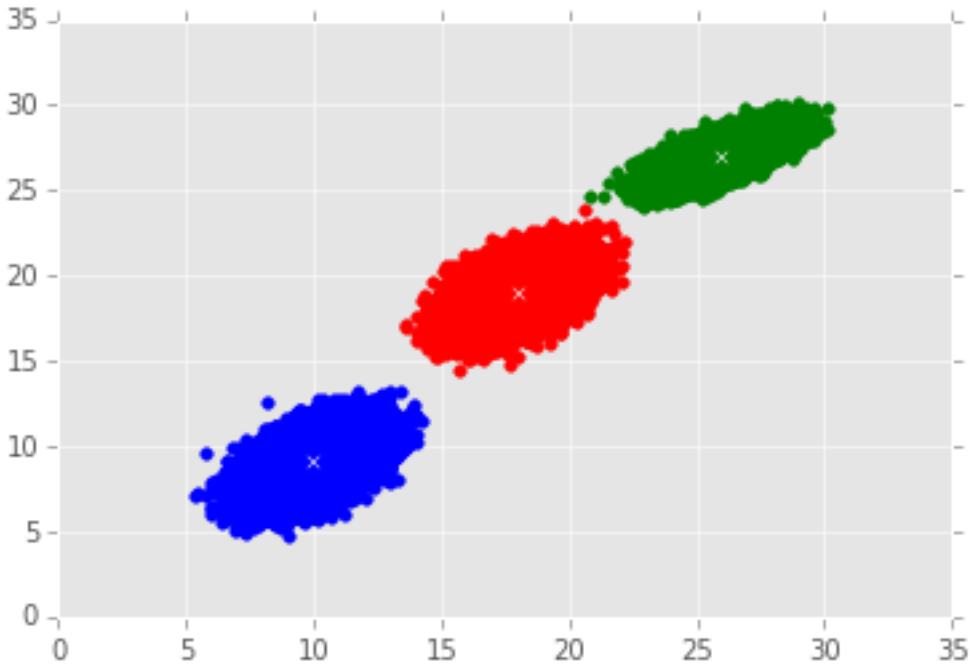
```
      1 def ScalableKMeansPlusPlus(data, k, l, r):
----> 2     centroids = data[np.random.choice(range(data.shape[0]),1), :]
      3     data_points = data[:, np.newaxis, :]
      4     passes = 0
      5


AttributeError: 'list' object has no attribute 'shape'
```

1.4 *Simulating data*

```
In [128]: def data_generation(n):
              mean1 = [10, 9]
              cov1 = [[1, 0.5], [0.5, 1]]
              data1 = np.random.multivariate_normal(mean1, cov1, n)

              mean2 = [18, 19]
              cov2 = [[1,0.5], [0.5, 1]]
              data2 = np.random.multivariate_normal(mean2, cov2, n)

              mean3 = [26, 27]
              cov3 = [[1, 0.5], [0.5, 0.5]]
              data3 = np.random.multivariate_normal(mean3, cov3, n)

              data = np.vstack((data1, data2, data3))
              np.random.shuffle(data)
              print (data.shape)
              return data
```

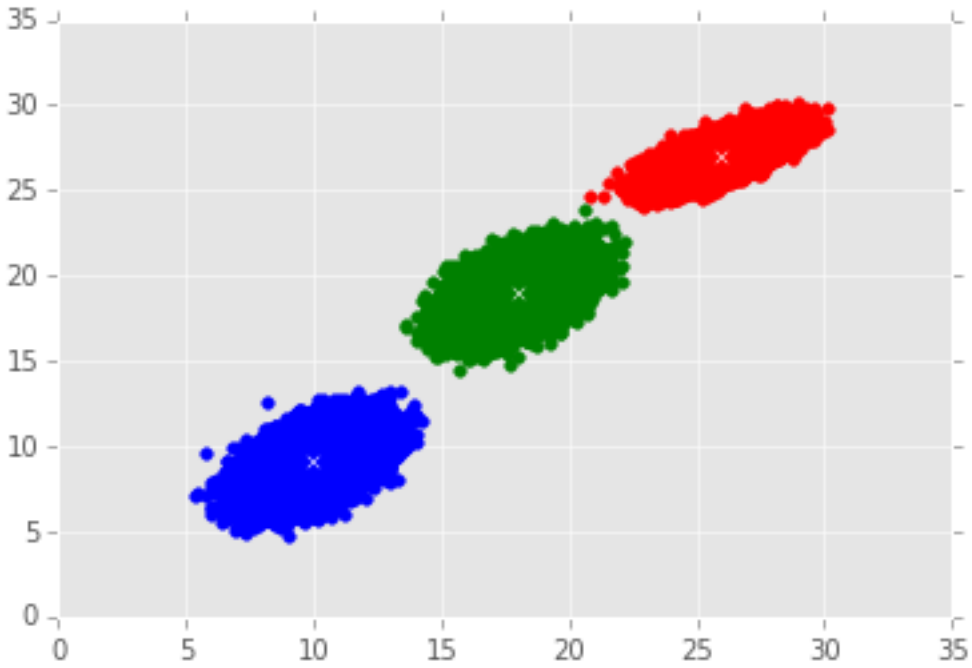
```
In [175]: ### KMeans

          np.random.seed(190)
          n = 100000
          k = 3
          data = data_generation(n)

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

          colors = iter(['r','b','g'])
          result1 = KMeans(data,centroids, k)
          centroids1 = result1[2]
          min_dist1 = result1[3]

          for i in range (k):
              plt.scatter(data[min_dist1[:,i] == True, :][:,0], data[min_dist1[:,i] == True, :][:,1], c
          for j in range(k):
              plt.scatter(centroids1[j,0],centroids1[j,1],color='w',marker='x')
```

```
(300000, 2)
(5, 'iterations are required to converge.')
```



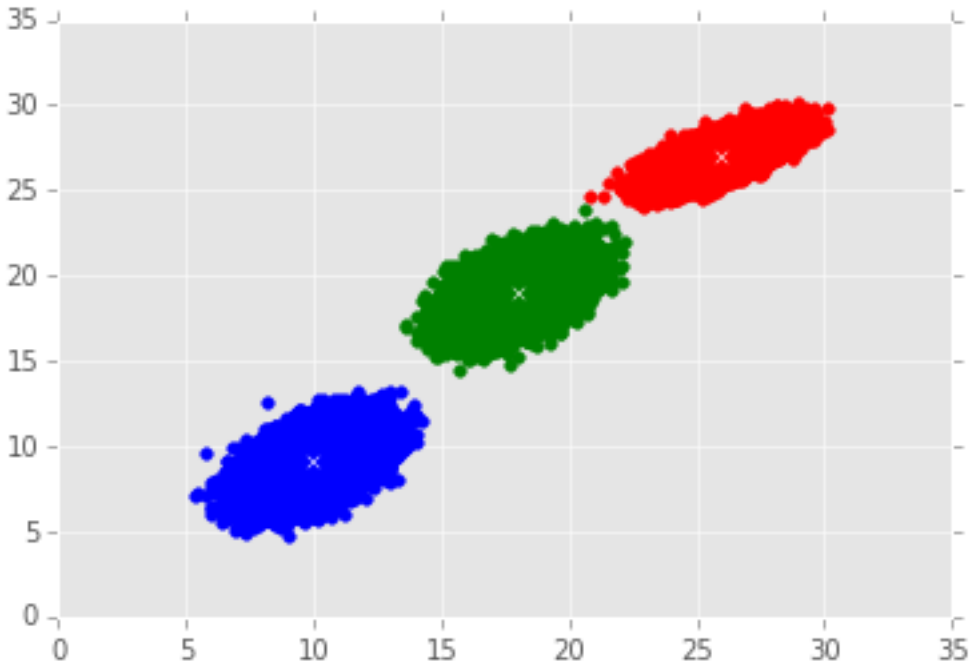
In [174]:
```
### KMeansPlusPlus
np.random.seed(190)
n = 100000
k = 3
data = data_generation(n)

colors = iter(['r','b','g'])
result2 = KMeans(data,KMeansPlusPlus(data,k),k)
centroids2 = result2[2]
min_dist2 = result2[3]

for i in range (k):
    plt.scatter(data[min_dist2[:,i] == True, :][:,0], data[min_dist2[:,i] == True, :][:,1], c
for j in range(k):
    plt.scatter(centroids2[j,0],centroids2[j,1],color='w',marker='x')
```

```
(300000, 2)
(3, 'iterations are required to converge.')
```



In [173]:
```
# Scalable KMeansPlusPlus
np.random.seed(190)
n = 100000
k = 3
data = data_generation(n)

colors = iter(['r','b','g'])
result3 = KMeans(data,ScalableKMeansPlusPlus(data, k, 2, 2),k)
centroids3 = result3[2]
min_dist3 = result3[3]
for i in range (k):
    plt.scatter(data[min_dist3[:,i] == True, :][:,0], data[min_dist3[:,i] == True, :][:,1], c
for j in range(k):
    plt.scatter(centroids3[j,0],centroids3[j,1],color='w',marker='x')
```

```
(300000, 2)
(2, 'iterations are required to converge.')
```



1.5 *Comparison*

```
In [176]: iteration = result1[0]
          print('iteration = ', iteration)
          result1[1][0]

('iteration = ', 5)

Out[176]: 16444739.6287

In [177]: iteration = result2[0]
          print('iteration = ', iteration)
          result2[1][0]

('iteration = ', 3)

Out[177]: 918210.2055

In [178]: iteration = result3[0]
          print('iteration = ', iteration)
          result3[1][0]

('iteration = ', 2)

Out[178]: 1194579.1451
```

According to the above three results, we can conclude that the number of iterations in ScalableK-MeansPlusPlus algorithm is the smallest one. It means that algorithm3 is much more efficient than other algorithms. Also, the first cluster_values of ScalableKMeansPlusPlus is the highest one. It means that initial centroids chosen by ScalableKMeansPlusPlus is more efficient than others.

1.6 *Profiling*

```
In [179]: np.random.seed(190)
          n = 100000
          k = 3
          data = data_generation(n)
```

```
(300000, 2)
```

```
In [180]: ! pip install --pre line-profiler &> /dev/null
          ! pip install psutil &> /dev/null
          ! pip install memory_profiler &> /dev/null
```

```
In [181]: %load_ext line_profiler
```

```
The line_profiler extension is already loaded. To reload it, use:
  %reload_ext line_profiler
```

```
In [182]: %lprun -f KMeans KMeans(data,KMeansPlusPlus(data,3),3)
```

```
(3, 'iterations are required to converge.')
```

Looking at the result of profiling for KMeansPlusPlus, this algorithm spent most time on the clustering part.

```
In [183]: %lprun -f KMeans KMeans(data,ScalableKMeansPlusPlus(data, 3, 2, 2),3)
```

```
(10, 'iterations are required to converge.')
```

ScalableMeansPlusPlus algorithm also spent most time on the clustering part.