Yue_Mu_Final_Project

April 30, 2015

1 STA663 Final Project

1.1 Background

- K-means clustering method remains the most popular clustering method and one of the top 10 algorithms in data mining. However, it is not a good clustering method in terms of efficiency or quality. Recent work has focused on improving the initialization procedure, which leads to k-means++ and kmeans|| algorithm.
- Basic K-means clustering uses the Lloyd algorithm, which starts with k arbitrary centers and performs an EM-type local search till convergence. However, it is very sensitive to initialization and takes many iterations to converge.
- K-means++ solves this problem by first choosing a center at random, and then sampling one point each iteration based on their probability distribution. It does spread out the centers pretty well, but it needs k passes over the data, which can be a problem when dataset is large.
- K-means||, also called scalable K-means++, oversamples by sampling each point independently with a larger probability, which is intuitively equivalent to updating the distribution much less frequently. It turns out to be more efficient than K-means++.

1.2 Algorithm

```
In [1]: from __future__ import division
        import os
        import sys
        import glob
        import matplotlib.pyplot as plt
        import matplotlib.cm as cm
        import numpy as np
        import pandas as pd
        import time
        %matplotlib inline
        %precision 4
        plt.style.use('ggplot')
```

1.2.1 K-means Algorithm (Simple)

```
In [2]: def initialization(data,k):
            """Return the initial set of cluster centers."""

            #Generate k random indices between 0 and the number of rows in the dataset
            centers_index = np.random.choice(range(data.shape[0]), k, replace=False)
            return data[centers_index, :]

        def kmeans1(centers,data,k):
```

```
        """Implement K-means clustering algorithm. Return converged cluster centers."""

        converge = False
        phi = []
        iterations = 0
        while (not converge) and (iterations < 1000):
            #Find out which cluster each data point is the closest to
            min_dist = np.zeros(data.shape[0])
            min_index = np.zeros(data.shape[0])
            for i,p in enumerate(data):
                result = minimum_distance(p,centers)
                min_dist[i] = result[0]
                min_index[i] = result[1]
            #Calculate phi
            phi_val = np.sum(min_dist)
            phi.append(phi_val)

            #Calculate the new centers
            new_centers = np.empty(centers.shape)
            for i in range(0, k):
                if data[min_index == i,:].shape[0] == 0:
                    new_centers[i] = centers[i]
                else:
                    new_centers[i] = np.mean(data[min_index == i, :], axis=0)

            #Compare old centers with new centers to see if the algorithm has converged
            if np.array_equal(centers, new_centers):
                converge = True
            else:
                centers = new_centers

            iterations += 1

        return (iterations, phi, centers, min_index)


    def minimum_distance(p, centers):
        """Return the distance to the closest cluster center"""

        min_index = 0
        min_dist = sys.float_info.max
        for i, cc in enumerate(centers):
            d = np.sum((p - cc) ** 2)
            if min_dist > d:
                min_dist = d
                min_index = i
        return (min_dist,min_index)
```

1.2.2 K-means Algorithm (Vectorized)

```
In [3]: def kmeans2(centers,data,k):
            """Implement K-means clustering algorithm. Return converged cluster centers."""

            converge = False
```

```
        phi = []
        iterations = 0
        while (not converge) and (iterations < 1000):
            #Find the Euclidean distance between a center and a data point
            data2 = data[:, np.newaxis, :]
            d2 = (data2 - centers) ** 2
            #Calculate the total distance to each center for each data point.
            distance = np.sum(d2, axis=2)

            #Find out which cluster each data point belongs to.
            min_index = np.zeros(distance.shape)
            min_index[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1

            #Calculate phi
            phi_val = np.sum(distance[min_index == 1])
            phi.append(phi_val)

            #Calculate the new centers
            new_centers = np.empty(centers.shape)
            for i in range(0, k):
                if data[min_index[:, i] == 1,:].shape[0] == 0:
                    new_centers[i] = centers[i]
                else:
                    new_centers[i] = np.mean(data[min_index[:, i] == True, :], axis=0)

            #Compare old centers with new centers to see if the algorithm has converged
            if np.array_equal(centers, new_centers):
                converge = True
            else:
                centers = new_centers

            iterations += 1

        return (iterations, phi, centers, min_index)
```

1.2.3 K-means++ Algorithm (Simple)

```
In [4]: def kmeanspp1(data, k):
        """Implement the K-means++ algorithm. Return k initial cluster centers."""
        #Sample a point uniformly at random from the data
        centers = data[np.random.choice(range(data.shape[0]),1), :]

        #Iterate k-1 times through the dataset to select the initial cluster centers
        while centers.shape[0] < k:
            min_dist = np.zeros(data.shape[0])
            for i,p in enumerate(data):
                min_dist[i] = minimum_distance(p,centers)[0]
            #Calculate cost of the data
            phi = np.sum(min_dist)
            #Calculate the probability distribution
            prob = min_dist/phi
            #Select the next centroid using the probability distribution calculated
            centers = np.vstack([centers, data[np.random.choice(range(data.shape[0]),1,
                                                                  p = prob), :]])
        return centers
```

1.2.4 K-means++ Algorithm (Vectorized)

```
In [5]: def kmeanspp2(data, k):
            """Implement the K-means++ algorithm. Return k initial cluster centers."""
            #Sample a point uniformly ar random from the data
            centers = data[np.random.choice(range(data.shape[0]),1), :]
            data2 = data[:, np.newaxis, :]

            #Iterate k-1 times through the dataset to select the initial cluster centers
            while centers.shape[0] < k:
                d2 = (data2 - centers) ** 2
                distance = np.sum(d2, axis=2)
                min_index = np.zeros(distance.shape)
                min_index[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1
                min_dist=distance[min_index == 1]
                phi = np.sum(min_dist)
                #Calculate the probability distribution
                prob = min_dist/phi
                #Select the next centroid using the probability distribution calculated
                centers = np.vstack([centers, data[np.random.choice(range(data.shape[0]),1,
                                                                     p = prob), :]])
            return centers
```

1.2.5 Scalable K-means++ Algorithm

```
In [6]: def scalablekmeanspp(data,k,l):
            """Implement the K-means++ algorithm. Return k initial cluster centers."""
            #Sample a point uniformly at random from the data
            centers = data[np.random.choice(range(data.shape[0]),1), :]
            data2 = data[:, np.newaxis, :]
            min_dist = np.zeros(data.shape[0])
            for i,p in enumerate(data):
                min_dist[i] = minimum_distance(p,centers)[0]
            phi = np.sum(min_dist)

            for i in range(int(round(np.log(phi)))):
                d2 = (data2 - centers) ** 2
                distance = np.sum(d2, axis=2)
                min_index = np.zeros(distance.shape)
                min_index[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1
                min_dist=distance[min_index == 1]
                phi = np.sum(min_dist)
                for j,p in enumerate(data):
                    prob = l*min_dist[j]/phi
                    u=np.random.uniform(0,1)
                    if prob >= u:
                        centers = np.vstack([centers, p])
            #Now we have an initial set of cluster centers that is greater than k
            #Find number of points in dataset closer to each center than any other centers
            d2 = (data2 - centers) ** 2
            distance = np.sum(d2, axis=2)
            min_index = np.zeros(distance.shape)
            min_index[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1
```

```
        weights = np.array([np.count_nonzero(min_index[:, j]) for j in
                            range(centers.shape[0])]).reshape(-1,1)
        #Recluster the weighted points into k clusters using kmeans++
        new_centers = centers[np.random.choice(range(centers.shape[0]),1), :]
        centers2 = centers
        index = np.where(centers2==new_centers)[0]
        centers2 = np.delete(centers2,index[0],axis=0)
        weights = np.delete(weights,index[0])
        while new_centers.shape[0] < k:
            #Calculate the probability distribution based on weights
            prob = weights/np.sum(weights)
            #Select the next centroid using the probability distribution calculated
            c = centers2[np.random.choice(range(centers2.shape[0]),1, p = prob.ravel()), :]
            index = np.where(centers2==c)[0]
            new_centers = np.vstack([new_centers, c])
            #Remove the selected center and its corresponding weight
            centers2 = np.delete(centers2,index[0],axis=0)
            weights = np.delete(weights,index[0])


        #Implement kmeans clustering to obtain the new centers
        new_centers2 = kmeans2(new_centers,centers,k)[2]
        return new_centers2
```

1.3 Simulate Data

```
In [7]: def generate_data(n):
            """Return mixed data with cluster labels."""
            mean0 = [0, 1]
            cov0 = [[1, 0.5], [0.5, 1]]
            data0 = np.random.multivariate_normal(mean0, cov0, n)
            data0 = np.hstack((data0, np.ones((data0.shape[0],1))))

            mean1 = [6, 7]
            cov1 = [[1, 0.5], [0.5, 1]]
            data1 = np.random.multivariate_normal(mean1, cov1, n)
            data1 = np.hstack((data1, np.ones((data1.shape[0],1)) * 2))

            mean2 = [11, 12]
            cov2 = [[1, 0.5], [0.5, 1]]
            data2 = np.random.multivariate_normal(mean2, cov2, n)
            data2 = np.hstack((data2, np.ones((data2.shape[0],1)) * 3))

            data = np.vstack((data0, data1, data2))
            np.random.shuffle(data)
            print (data.shape)
            return data

        data1=generate_data(1000)[:,0:2]
```

```
(3000, 3)
```

```
In [8]: def generate_data(n):
            """Return mixed data with cluster labels."""
            mean0 = [0, 1]
            cov0 = [[1, 0.5], [0.5, 1]]
            data0 = np.random.multivariate_normal(mean0, cov0, n)
            data0 = np.hstack((data0, np.ones((data0.shape[0],1))))

            mean1 = [5, 6]
            cov1 = [[1, 0.5], [0.5, 1]]
            data1 = np.random.multivariate_normal(mean1, cov1, n)
            data1 = np.hstack((data1, np.ones((data1.shape[0],1)) * 2))

            mean2 = [10, 11]
            cov2 = [[1, 0.5], [0.5, 1]]
            data2 = np.random.multivariate_normal(mean2, cov2, n)
            data2 = np.hstack((data2, np.ones((data2.shape[0],1)) * 3))

            mean3 = [15, 16]
            cov3 = [[1, 0.5], [0.5, 1]]
            data3 = np.random.multivariate_normal(mean2, cov2, n)
            data3 = np.hstack((data3, np.ones((data3.shape[0],1)) * 4))

            mean4 = [20, 21]
            cov4 = [[1, 0.5], [0.5, 1]]
            data4 = np.random.multivariate_normal(mean2, cov2, n)
            data4 = np.hstack((data4, np.ones((data4.shape[0],1)) * 5))

            data = np.vstack((data0, data1, data2, data3, data4))
            np.random.shuffle(data)
            print (data.shape)
            return data

        data2=generate_data(5000)[:,0:2]
```

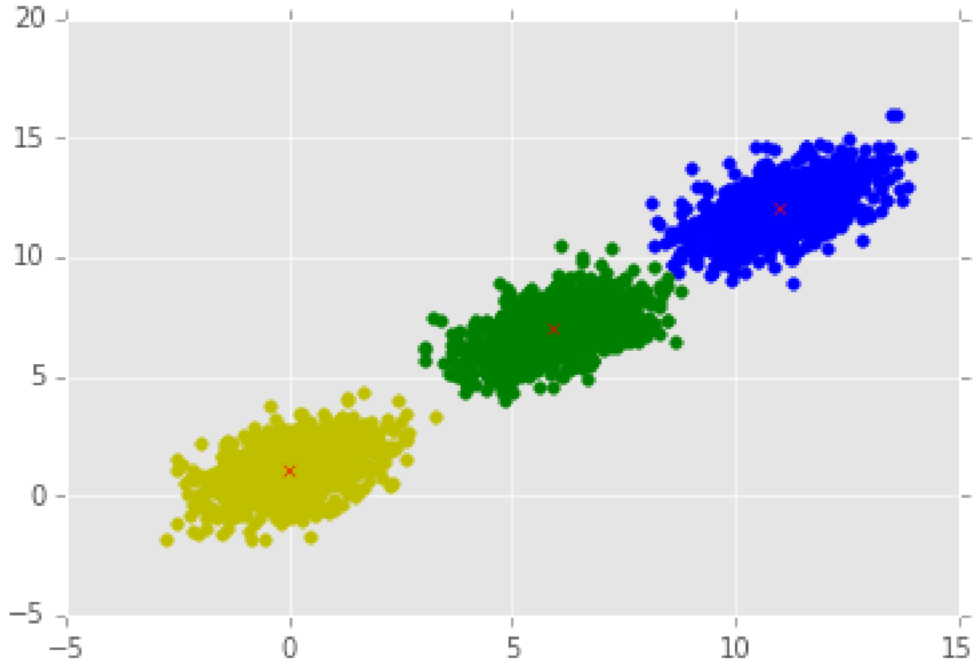
```
(25000, 3)
```

1.4 Test

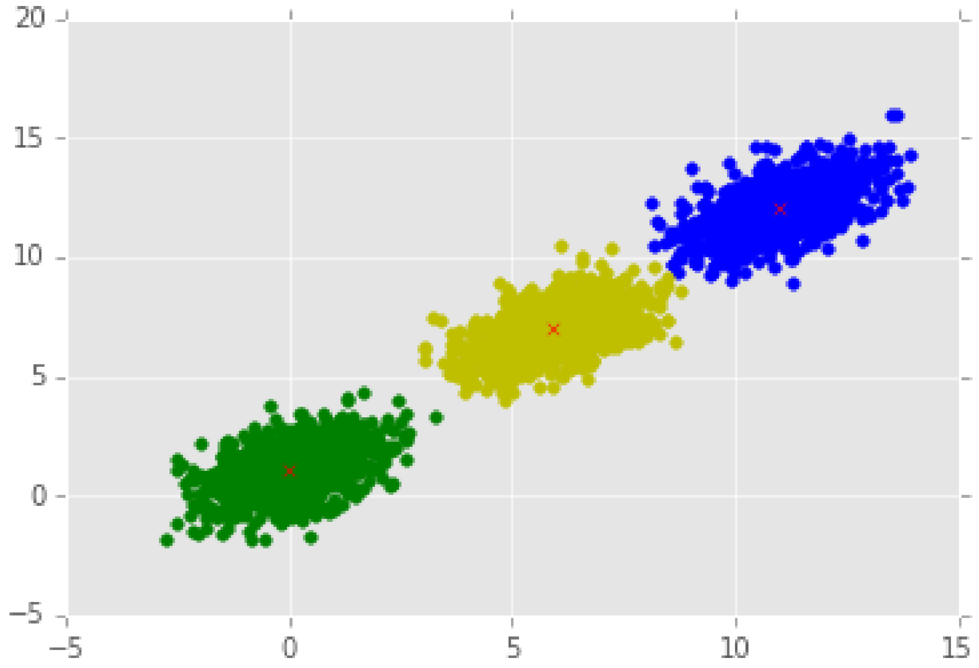
```
In [229]: k=3
          result = kmeans2(initialization(data1,k),data1,k)
          centers = result[2]
          min_index = result[3]
          #Plot clusters
          cols=iter(['b','g','y'])
          for i in range (k):
              plt.scatter(data1[min_index[:,i] == 1, :][:,0],
                          data1[min_index[:,i] == 1, :][:,1], color=next(cols))
          plt.scatter(centers[:,0], centers[:,1], color='r', marker='x')
```

```
('Initial Centers:', array([[ 8.2972,  8.0749],
       [ 6.5101,  6.9435],
       [-1.2449, -0.4135]]))
('Required ', 4, ' iterations to converge.')
```

```
Out[229]: <matplotlib.collections.PathCollection at 0x7fe759310e10>
```



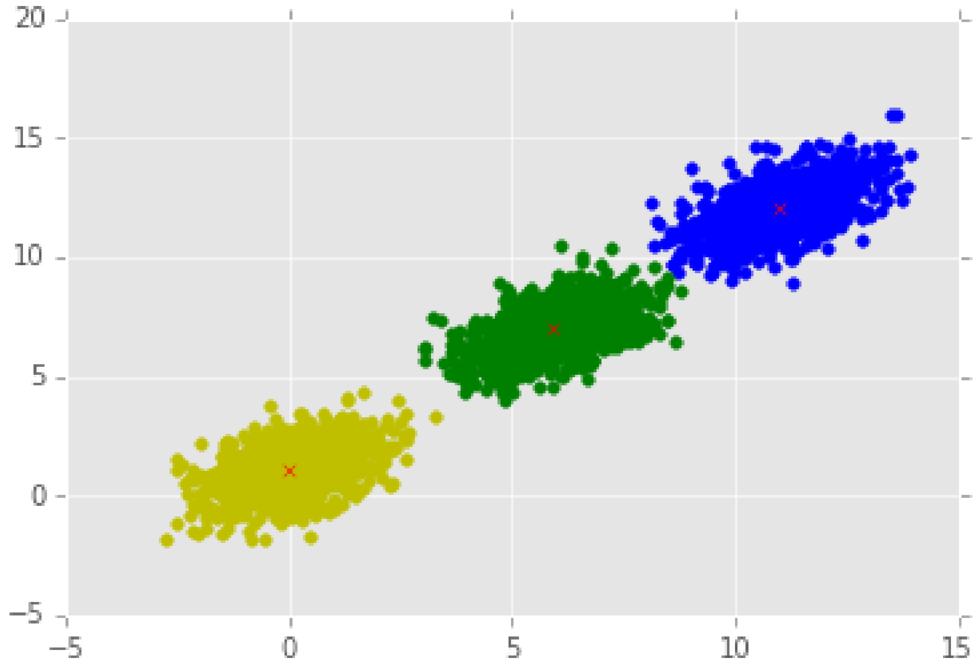
```
In [230]: result = kmeans2(kmeanspp2(data1,k),data1,k)
          centers = result[2]
          min_index = result[3]
          #Plot clusters
          cols=iter(['b','g','y'])
          for i in range (k):
              plt.scatter(data1[min_index[:,i] == 1, :][:,0],
                          data1[min_index[:,i] == 1, :][:,1], color=next(cols))
          plt.scatter(centers[:,0], centers[:,1], color='r', marker='x')

('Initial Centers:', array([[ 10.4522,  14.5413],
       [  1.6204,   2.3635],
       [ 11.7767,  10.5528]]))
('Required ', 6, ' iterations to converge.')

Out[230]: <matplotlib.collections.PathCollection at 0x7fe7592a2e50>
```



```
In [232]: result = kmeans2(scalablekmeanspp(data1,k,2),data1,k)
          centers = result[2]
          min_index = result[3]
          #Plot clusters
          cols=iter(['b','g','y'])
          for i in range (k):
              plt.scatter(data1[min_index[:,i] == 1, :][:,0],
                          data1[min_index[:,i] == 1, :][:,1], color=next(cols))
          plt.scatter(centers[:,0], centers[:,1], color='r', marker='x')
```

```
('Initial Centers:', array([[ 12.5395,  12.3895],
       [ 10.388 ,  12.0946],
       [  6.6019,   7.9372]]))
('Required ', 7, ' iterations to converge.')
('Initial Centers:', array([[ 11.1654,  12.1567],
       [  5.877 ,   6.9726],
       [  0.4497,   1.459 ]]))
('Required ', 3, ' iterations to converge.')
```

```
Out[232]: <matplotlib.collections.PathCollection at 0x7fe75923bf90>
```



From the output, we can see that all three algorithms are working as expected.

1.5 Comparison of Algorithms

```
In [53]: np.random.seed(200)
         result = kmeans2(initialization(data1,3),data1,3)
         #Print number of iterations till convergence
         print result[0]
         #Print the clustering cost after initialization
         print result[1][0]
```

```
7
46372.4191656
```

```
In [55]: np.random.seed(200)
         result = kmeans2(kmeanspp2(data1,3),data1,3)
         print result[0]
         print result[1][0]
```

```
4
13765.1574859
```

```
In [54]: np.random.seed(200)
         result = kmeans2(scalablekmeanspp(data1,3,1.5),data1,3)
         print result[0]
         print result[1][0]
```

```
3
6396.64504607
```

```
In [66]: np.random.seed(140)
         result = kmeans2(initialization(data2,5),data2,5)
         print result[0]
         print result[1][0]

36
179399.609486

In [65]: np.random.seed(140)
         result = kmeans2(kmeanspp2(data2,5),data2,5)
         print result[0]
         print result[1][0]

27
81570.8032694

In [64]: np.random.seed(140)
         result = kmeans2(scalablekmeanspp(data2,5,2.5),data2,5)
         print result[0]
         print result[1][0]

14
34903.9182741
```

Based on the results above, we can see that K-means++ reduces number of Lloyd iterations by a great amount, and scalable K-means++ does that even more than K-means++, especially when the dataset is bigger and has more clusters. The efficiency of scalable K-means++ can also be reflected in the initial clustering cost. The smaller the initial clustering cost is, the better the intial centroids are, meaning that they are closer to the actual cluster centers. The initial clustering cost of scalable K-means++ is significantly lower than the other two algorithms.

1.6 Code Efficiency

```
In [67]: ! pip install --pre line-profiler &> /dev/null
         ! pip install psutil &> /dev/null
         ! pip install memory_profiler &> /dev/null

In [68]: %load_ext line_profiler

In [92]: %lprun -f kmeans1 kmeans1(kmeanspp1(data1,3),data1,3)

In []: Timer unit: 1e-06 s

       Total time: 0.417552 s
       File: <ipython-input-2-eb941895ad31>
       Function: kmeans1 at line 8

       Line #      Hits         Time  Per Hit   % Time  Line Contents
       ==============================================================
            8                                           def kmeans1(centers,data,k):
            9                                               """Implement K-means clustering algorithm. 
           10
           11         1            2      2.0      0.0      converge = False
           12         1            2      2.0      0.0      phi = []
           13         1            0      0.0      0.0      iterations = 0
           14         4           11      2.8      0.0      while (not converge) and (iterations < 1000
           15                                                   #Find out which cluster each data point
           16         3           30     10.0      0.0          min_dist = np.zeros(data.shape[0])
           17         3           10      3.3      0.0          min_index = np.zeros(data.shape[0])
           18      9003        10552      1.2      2.5          for i,p in enumerate(data):
           19      9000       381187     42.4     91.3              result = minimum_distance(p,centers
           20      9000        14599      1.6      3.5              min_dist[i] = result[0]
           21      9000         9386      1.0      2.2              min_index[i] = result[1]
           22                                                   #Calculate phi
           23         3           40     13.3      0.0          phi_val = np.sum(min_dist)
           24         3            7      2.3      0.0          phi.append(phi_val)
           25
           26                                                   #Calculate the new centers
           27         3           15      5.0      0.0          new_centers = np.empty(centers.shape)
           28        12           32      2.7      0.0          for i in range(0, k):
           29         9          514     57.1      0.1              if data[min_index == i,:].shape[0] 
           30                                                           new_centers[i] = centers[i]
           31                                                       else:
           32         9         1051    116.8      0.3                  new_centers[i] = np.mean(data[m
           33
           34                                                   #Compare old centers with new centers t
           35         3          109     36.3      0.0          if np.array_equal(centers, new_centers)
           36         1            1      1.0      0.0              converge = True
           37                                                   else:
           38         2            1      0.5      0.0              centers = new_centers
           39
           40         3            3      1.0      0.0          iterations += 1
           41
           42         1            0      0.0      0.0      return (iterations, phi, centers, min_index
```

We can see that majority of the time was spent on running the minimum_distance function because it requires running through the entire dataset. We can try to optimize this part of the algorithm using Cython.

```
In [9]: %load_ext cythonmagic

In [10]: %%cython -a
         import numpy as np
         cimport numpy as np

         cimport cython
         @cython.boundscheck(False)
         @cython.wraparound(False)

         def cminimum_distance(np.ndarray[np.float64_t, ndim=1] p,
                               np.ndarray[np.float64_t, ndim=2] centers):

             cdef int min_index = 0
             cdef float min_dist = 1.79769313486e+308
             cdef int i
             cdef np.ndarray[np.float64_t, ndim=1] cc
             for i, cc in enumerate(centers):
                 d = np.sum((p - cc) ** 2)
                 if min_dist > d:
                     min_dist = d
                     min_index = i
             return (min_dist,min_index)
```

```
Out[10]: <IPython.core.display.HTML at 0x7fa432851b90>
```

1.6.1 K-means Algorithm (Cython)

```
In [11]: def kmeans3(centers,data,k):
             """Implement K-means clustering algorithm. Return converged cluster centers."""

             converge = False
             phi = []
             iterations = 0
             while (not converge) and (iterations < 1000):
                 #Find out which cluster each data point is the closest to
                 min_dist = np.zeros(data.shape[0])
                 min_index = np.zeros(data.shape[0])
                 for i,p in enumerate(data):
                     #Here use the cython version of the minimum_distance function
                     result = cminimum_distance(p,centers)
                     min_dist[i] = result[0]
                     min_index[i] = result[1]
                 #Calculate phi
                 phi_val = np.sum(min_dist)
                 phi.append(phi_val)

                 #Calculate the new centers
                 new_centers = np.empty(centers.shape)
                 for i in range(0, k):
                     if data[min_index == i,:].shape[0] == 0:
                         new_centers[i] = centers[i]
                     else:
                         new_centers[i] = np.mean(data[min_index == i, :], axis=0)

                 #Compare old centers with new centers to see if the algorithm has converged
                 if np.array_equal(centers, new_centers):
                     converge = True
                 else:
                     centers = new_centers

                 iterations += 1

             return (iterations, phi, centers, min_index)
```

1.6.2 K-means++ (Cython)

```
In [12]: def kmeanspp3(data, k):
             """Implement the K-means++ algorithm. Return k initial cluster centers."""
             #Sample a point uniformly at random from the data
             centers = data[np.random.choice(range(data.shape[0]),1), :]

             #Iterate k-1 times through the dataset to select the initial cluster centers
             while centers.shape[0] < k:
                 min_dist = np.zeros(data.shape[0])
                 for i,p in enumerate(data):
                     #Here use the cython version of the minimum_distance function
                     min_dist[i] = cminimum_distance(p,centers)[0]
                 #Calculate cost of the data
```

```
                phi = np.sum(min_dist)
                #Calculate the probability distribution
                prob = min_dist/phi
                #Select the next centroid using the probability distribution calculated
                centers = np.vstack([centers, data[np.random.choice(range(data.shape[0]),1,
                                                                    p = prob), :]])
            return centers
```

```
In [13]: %timeit -n 5 kmeans1(kmeanspp1(data1,3),data1,3)
         %timeit -n 5 kmeans2(kmeanspp2(data1,3),data1,3)
         %timeit -n 5 kmeans3(kmeanspp3(data1,3),data1,3)
```

```
5 loops, best of 3: 469 ms per loop
5 loops, best of 3: 5.36 ms per loop
5 loops, best of 3: 531 ms per loop
```

It seems that in this case vectorization increases the efficiency the most.