https://drive.google.com/file/d/1xe8HZyz8XcC8-X5XYKk_s2_yF0RHrKq_/view?usp=sharing

Clustering:

1. K-means
2. Fuzzy C-means
3. GMM
4. Practical Example (repeat for all 3)

1. K-means clustering

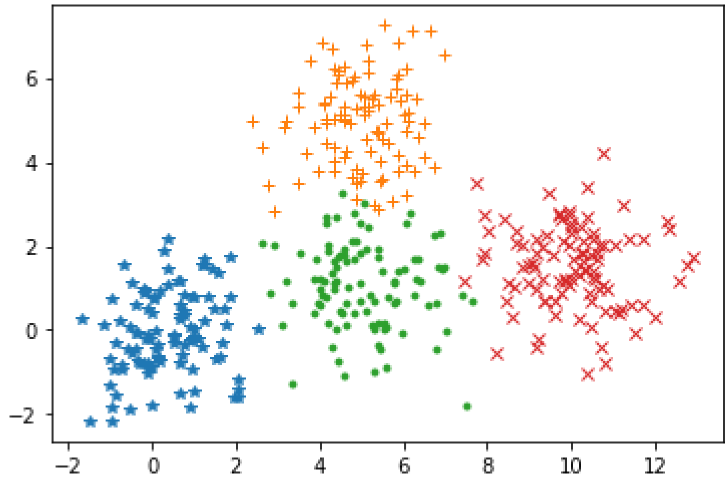
a) Data genearation

b) Generate 2D gaussian data of 4 types each having 100 points, by taking appropriate mean and varince (example: mean :(0.5 0) (5 5) (5 1) (10 1.5), variance : Identity matrix)

```
import numpy as np
import matplotlib.pyplot as plt
mean1 = [0.5, 0]
mean2 =[5, 5]
mean3 =[5, 1]
mean4 =[10, 1.5]
cov = [[1, 0], [0, 1]]
x1, y1 = np.random.multivariate_normal(mean1, cov, 100).T
plt.plot(x1, y1, '*')
x1.reshape(1,100)
y1.reshape(1,100)

x2, y2 = np.random.multivariate_normal(mean2, cov, 100).T
plt.plot(x2, y2, '+')
x2.reshape(1,100)
y2.reshape(1,100)

x3, y3 = np.random.multivariate_normal(mean3, cov, 100).T
plt.plot(x3, y3, '.')
x3.reshape(1,100)
y3.reshape(1,100)

x4, y4 = np.random.multivariate_normal(mean4, cov, 100).T
plt.plot(x4, y4, 'x')
x4.reshape(1,100)
y4.reshape(1,100)

x_tu=(x1,x2,x3,x4)
y_tu=(y1,y2,y3,y4)
x=np.concatenate(x_tu)
y=np.concatenate(y_tu)
print(x.shape)
plt.show()
```

```
plt.show()

## Data generation
# write your code here
```

```
(400,)
```

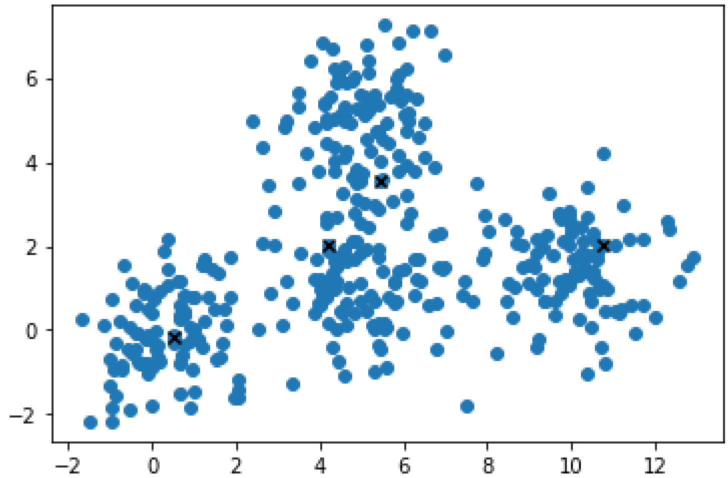


Cluster Initialization

a) Randomly initialize the cluster centers (any k- number of data points from the generated data)

```
import random
K=4
centroid = np.random.rand(4,2);
for i in range(4):
  r=random.randint(0,399)
  centroid[i][0] = x[r]
  centroid[i][1] = y[r]
print(centroid)
plt.scatter(x,y)
plt.scatter(centroid[:,0],centroid[:,1],marker='x',color='k')
plt.show()
```

```
[[ 5.46383432  3.55592432]
 [ 0.53847849 -0.1793623 ]
 [ 4.22282138  2.0112682 ]
 [10.7777018   2.04492435]]
```



Cluster assignment and re-estimation Stage

a) Using initial/estimated cluster centers (mean μ_i) perform cluster assignment.

b) Assigned cluster for each feature vector (X_j) can be written as:

$$arg\min_{i} ||C_i - X_j||_2,\ 1 \leq i \leq K,\ 1 \leq j \leq N$$

c) Re-estimation: After cluster assignment, the mean vector is recomputed as,

$$\mu_i = \frac{1}{N_i} \sum_{j \in i^{th} cluster} X_j$$

where N_i represents the number of datapoints in the i^{th} cluster.

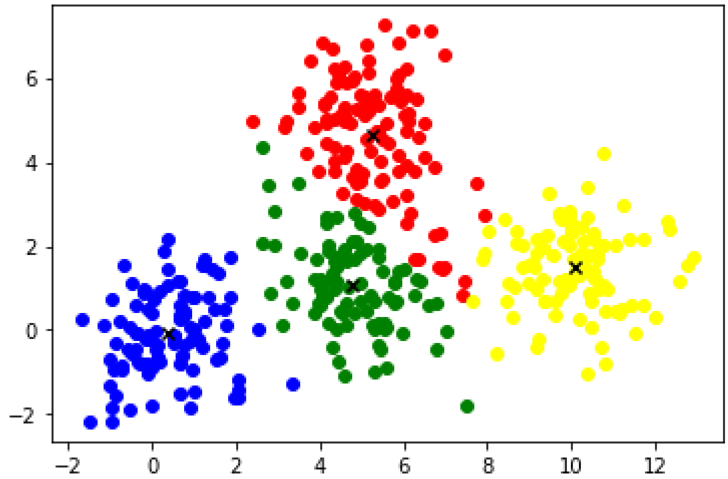
d) Objective function (to be minimized):

$$Error(\mu) = \frac{1}{N} \sum_{i=1}^{K} \sum_{j \in i^{th} cluster} ||C_i - X_j||_2$$

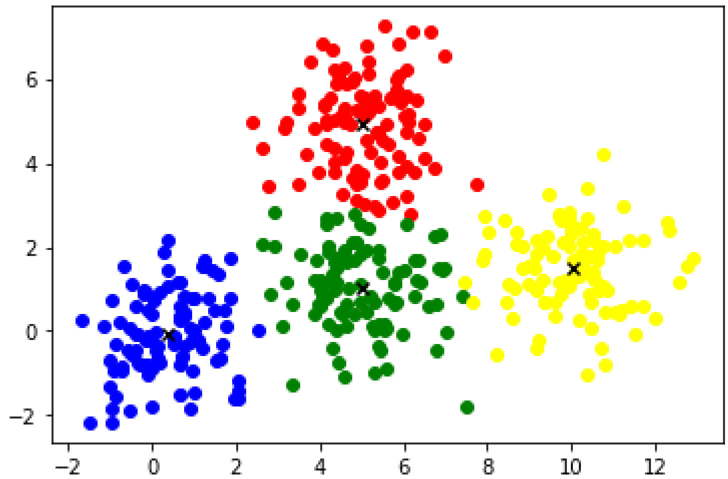
```
# write Your code here

i=0
error=-1
distances = np.zeros((len(x),K))
clusters = np.zeros(len(x))
t=0
K=4
err=[]
ite=[]
colours = ["red","blue","green","yellow"]
while error!=0:
  for i in range(K):
    xar = np.array(centroid[i][0]*np.ones([400,]))
    yar = np.array(centroid[i][1]*np.ones([400,]))
    distances[:,i] = np.sqrt(np.square(x-xar)+np.square(y-yar))
  for i in range(len(x)):
    clusters[i]=np.argmin(distances[i])+1
  for i in range(K):
    centroid[i][0]=np.mean(x[clusters==i+1])
    centroid[i][1]=np.mean(y[clusters==i+1])
  up_error =0
  ite.append(t)
  t=t+1
  for j in range(K):
    plt.scatter(x[clusters==j+1],y[clusters==j+1],c=colours[j])
    plt.scatter(centroid[j][0],centroid[j][1],marker='x',color='k')
    plt.title('iteration:%d'%(t))

    x_tem = np.array(centroid[j][0]*np.ones([len(x[clusters==j+1]),]))
    y_tem = np.array(centroid[j][1]*np.ones([len(y[clusters==j+1]),]))
    up_error = up_error+ np.sum (np.square(x_tem-x[clusters==j+1])+np.square(y_tem-y[clust
  plt.show()
  up_error = up_error/400
```

```
      up_error = round(up_error,4)
      err.append(up_error)
      if (error==up_error):
        break
      else:
        error = up_error

    plt.plot(ite,err)
    plt.title("error")
    plt.show()
```

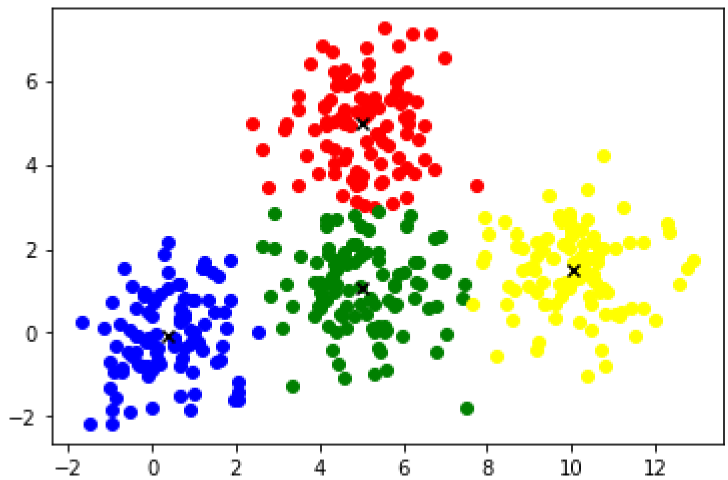
iteration:1



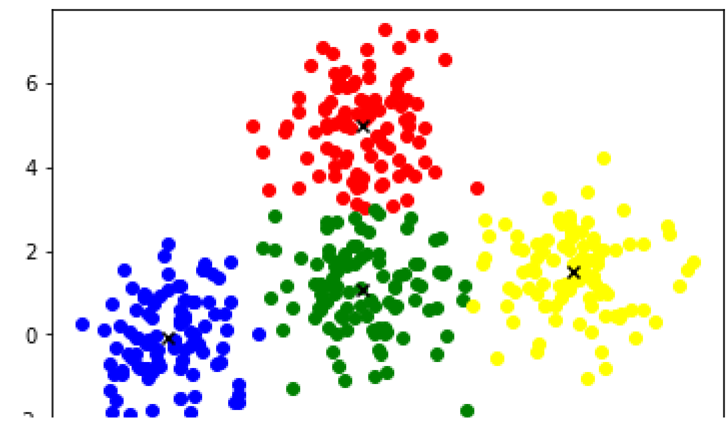
iteration:2



iteration:3



iteration:4



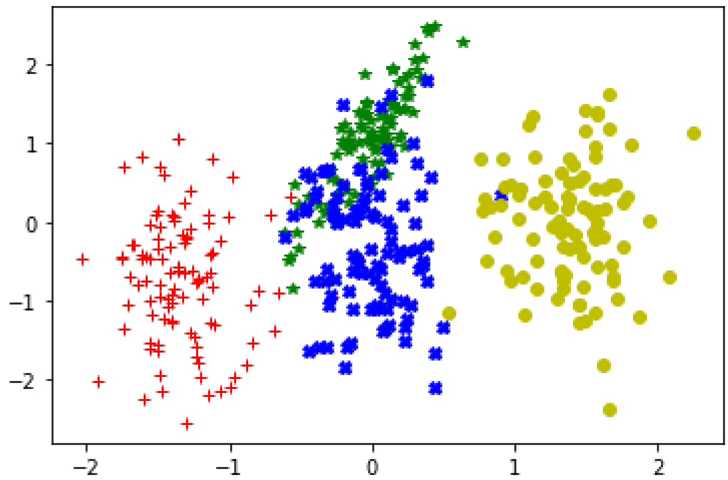
2. GMM Clustering

1. Data generation

a) Use the same data that you generated for K-means

```
import numpy as np
import matplotlib.pyplot as plt
```

```
(400,)
```



2. Initialization

a) Mean vector (randomly any from the given data points) (μ_k)

b) Coveriance (initialize with (identity matrix)*max(data)) (Σ_k)

c) Weights (uniformly) (w_k), with constraint: $\sum_{k=1}^{K} w_k = 1$

```
#%% Initialisations

def initialization(data,K):

  # write your code here
  return theta
```

3. Expectation stage

$$\gamma_{ik} = \frac{w_k P(x_i|\Phi_k)}{\sum_{k=1}^{K} w_k P(x_i|\Phi_k)}$$

where,

$$\Phi_k = \{\mu_k, \Sigma_k\}$$
$$\theta_k = \{\Phi_k, w_k\}$$
$$w_k = \frac{N_k}{N}$$
$$N_k = \sum_{i=1}^{N} \gamma_{ik}$$
$$P(x_i|\Phi_k) = \frac{1}{\quad} e^{-(x_i-\mu_k)^T \Sigma_k^{-1}(x_i-\mu_k)}$$

```
# Expectation stage

#%% E-Step GMM
from scipy.stats import multivariate_normal
def E_Step_GMM(data,K,theta):

    # write your code here

    return responsibility
```

3. Maximization stage

a) $w_k = \frac{N_k}{N}$, where $N_k = \sum_{i=1}^{N} \gamma_{ik}$

b) $\mu_k = \frac{\sum_{i=1}^{N} \gamma_{ik} x_i}{N_k}$

c) $\Sigma_k = \frac{\sum_{i=1}^{N} \gamma_{ik}(x_i-\mu_k)(x_i-\mu_k)^T}{N_k}$

Objective function(maximized through iteration):

$$L(\theta) = \sum_{i=1}^{N} log \sum_{k=1}^{K} w_k P(x_i|\Phi_k)$$

```
# Maximization stage

#%% M-STEP GMM
def M_Step_GMM(data,responsibility):
    # write your code here

    return theta, log_likelihood
```

4. Final run (EM algorithem)

a) initialization

b)Itterate E-M untill $L(\theta_n) - L(\theta_{n-1}) \leq th$

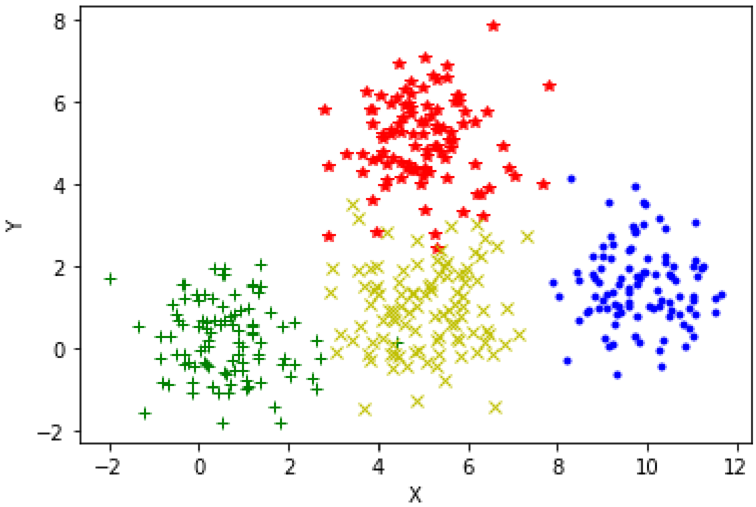
c) Plot and see the cluster allocation at each itteration

```
log_l=[]
Itr=50
eps=10**(-14)  # for threshold
clr=['r','g','b','y','k','m','c']
mrk=['+','*','X','o','.','<','p']


K=4   # no. of clusters

theta=initialization(data,K)

for n in range(Itr):

  responsibility=E_Step_GMM(data,K,theta)

  cluster_label=np.argmax(responsibility,axis=1) #Label Points

  theta,log_likhd=M_Step_GMM(data,responsibility)

  log_l.append(log_likhd)

  plt.figure()
  for l in range(K):
    id=np.where(cluster_label==l)
    plt.plot(data[id,0],data[id,1],'.',color=clr[l],marker=mrk[l])
  Cents=theta[0].T
  plt.plot(Cents[:,0],Cents[:,1],'X',color='k')
  plt.title('Iteration= %d' % (n))

  if n>2:
    if abs(log_l[n]-log_l[n-1])<eps:
      break


plt.figure()
plt.plot(log_l)
```

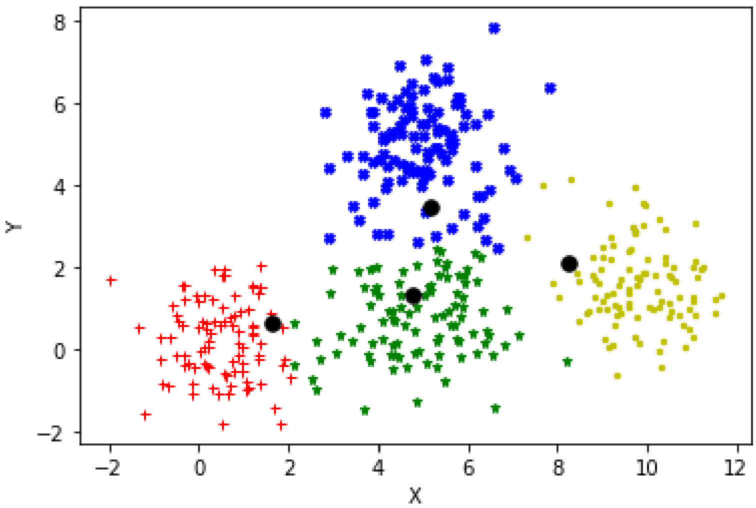
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random

a = int(''.join(format(ord(i), 'b') for i in 'b')[:6])
np.random.seed(a),random.seed(a)

cor = np.identity(2, dtype = float)
samples1 = np.random.multivariate_normal([0.5, 0], cor, 100)
samples2 = np.random.multivariate_normal([5, 5], cor, 100)
samples3 = np.random.multivariate_normal([5, 1], cor, 100)
```

```
samples3 = np.random.multivariate_normal([5, 1], cor, 100)
samples4 = np.random.multivariate_normal([10, 1.5], cor, 100)
data = np.concatenate((samples1, samples2, samples3, samples4))

#def data_plot():
col = ['+g','*r','xy','.b']
plt.plot(samples1[:, 0], samples1[:, 1], col[0],label='Custer 1')
plt.plot(samples2[:, 0], samples2[:, 1], col[1],label='Custer 2')
plt.plot(samples3[:, 0], samples3[:, 1], col[2],label='Custer 3')
plt.plot(samples4[:, 0], samples4[:, 1], col[3],label='Custer 4')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

from scipy.spatial.distance import euclidean as distance
from scipy.stats import multivariate_normal
class GMM:
  def __init__(self, k: int, n_iters: int, tol: float):
    self.n_components, self.n_iters,self.tol  = k, n_iters, tol
    np.random.seed(a),random.seed(a)

  def _do_estep(self, X):

    for k in range(self.n_components):
      prior = self.weights[k]
      likelihood = multivariate_normal(self.means[k], self.covs[k]).pdf(X)
      self.resp[:, k] = prior * likelihood

    log_likelihood = np.sum(np.log(np.sum(self.resp, axis = 1)))

        # normalize over all possible cluster assignments
    self.resp = self.resp / self.resp.sum(axis = 1, keepdims = 1)
    return log_likelihood

  def _do_mstep(self, X):
        # total responsibility assigned to each cluster, N^{soft}
    resp_weights = self.resp.sum(axis = 0)
        # weights
    self.weights = resp_weights / X.shape[0]
        # means
    weighted_sum = np.dot(self.resp.T, X)
    self.means = weighted_sum / resp_weights.reshape(-1, 1)
        # covariance
    for k in range(self.n_components):
      diff = (X - self.means[k]).T
      weighted_sum = np.dot(self.resp[:, k] * diff, diff.T)
      self.covs[k] = weighted_sum / resp_weights[k]

  def fit(self, X):
        # data's responsibility vector
    self.resp = np.zeros((X.shape[0], self.n_components))

        # initialize parameters #self.covs = np.full(shape, np.cov(X.T))
    self.means = X[np.random.choice(X.shape[0], self.n_components)]
    self.weights = np.full(self.n_components, 1 / self.n_components)
    self.covs = np.full((self.n_components, X.shape[1], X.shape[1]), cor*np.max(np.asarray
```

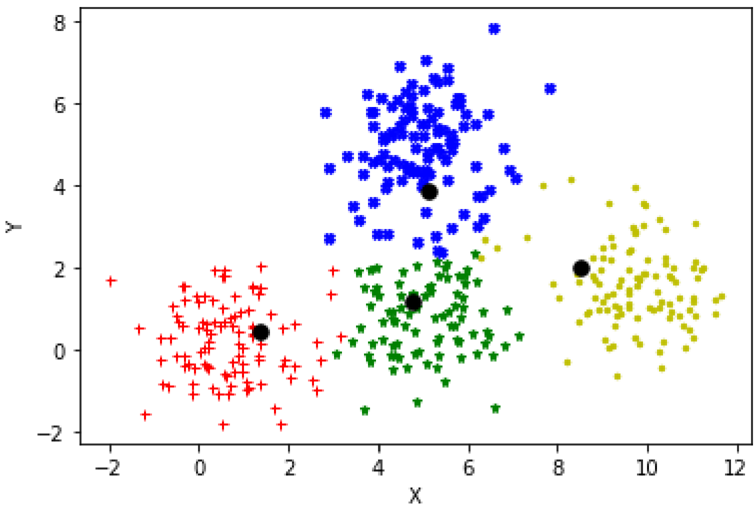
```
    self.covs = np.full((self.n_components, X.shape[1], X.shape[1]), cov*np.max(np.asarray
    log_likelihood, self.log_likelihood_trace, clr, mrk = 0, [], ['g','b','y','r'], ['*','

    for i in range(self.n_iters):
      log_likelihood_new = self._do_estep(X)
      self._do_mstep(X)
      cluster_label=np.argmax(self.resp,axis=1) #Label Points
      for l in range(self.n_components):
        id=np.where(cluster_label==l)
        plt.plot(X[id,0],X[id,1],'.',color=clr[l],marker=mrk[l],markersize=5)
      plt.plot(self.means[:,0],self.means[:,1],'.',color='black',markersize=15,label="Mean
      plt.xlabel('X')
      plt.ylabel('Y')
      plt.title('Iteration: '+str(i+1))
      plt.show()
      if abs(log_likelihood_new - log_likelihood) <= self.tol:
        print("Converged")
        break
            #print("Difference in Log Likelihood: "+str(abs(log_likelihood_new - log_likel
      log_likelihood = log_likelihood_new
      self.log_likelihood_trace.append(log_likelihood)
    plt.plot(np.asarray(self.log_likelihood_trace))
    plt.xlabel('no.of iterations')
    plt.ylabel('Log Likelihood')
    plt.title('Log Likelihood Trace')
    plt.show()

gmm = GMM(k = 4, n_iters = 50, tol = 1e-4)
gmm.fit(data)
```



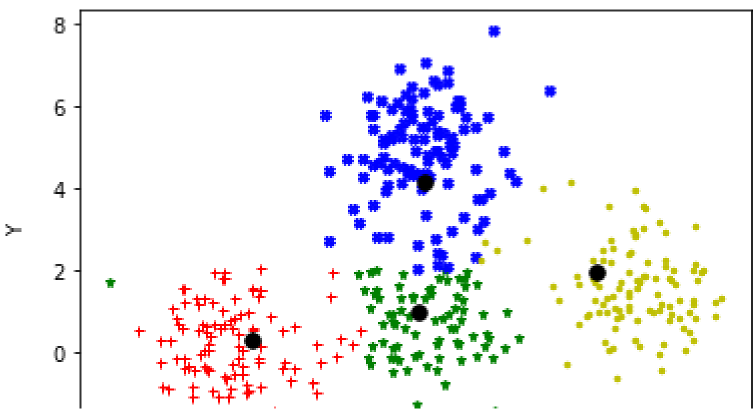
Iteration: 1

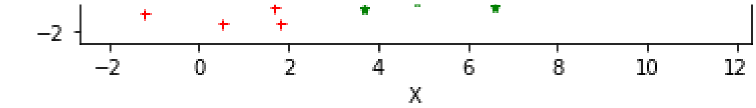


Iteration: 2

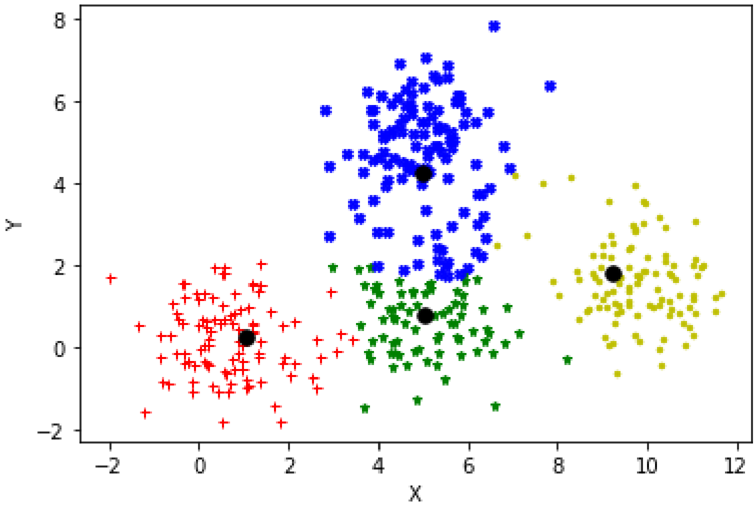


Iteration: 3

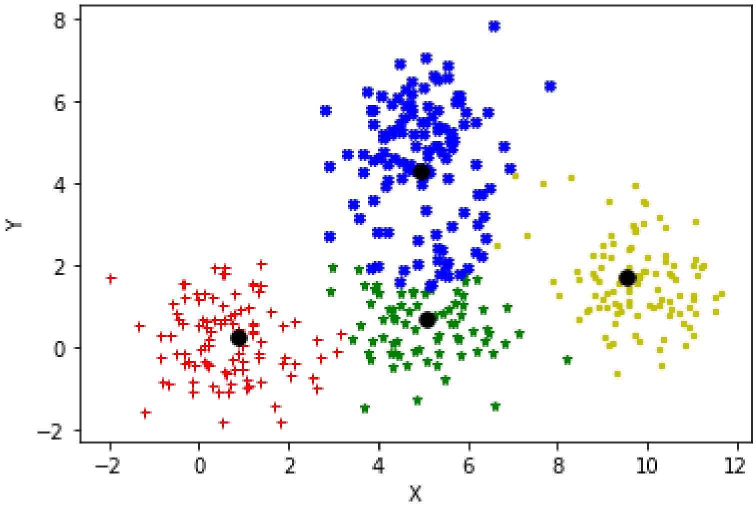




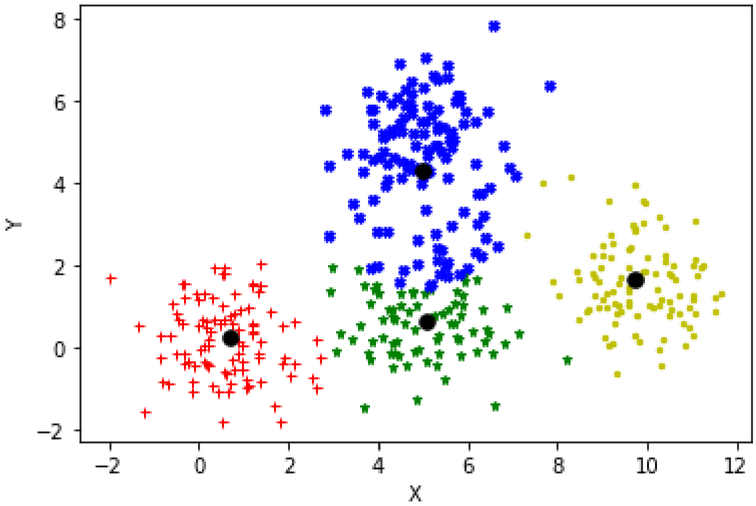
Iteration: 4



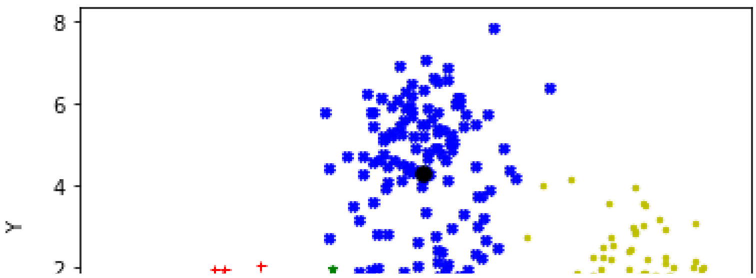
Iteration: 5

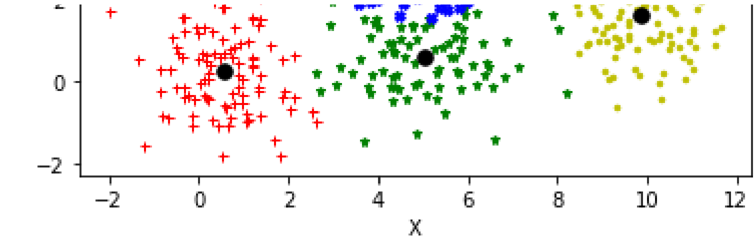


Iteration: 6

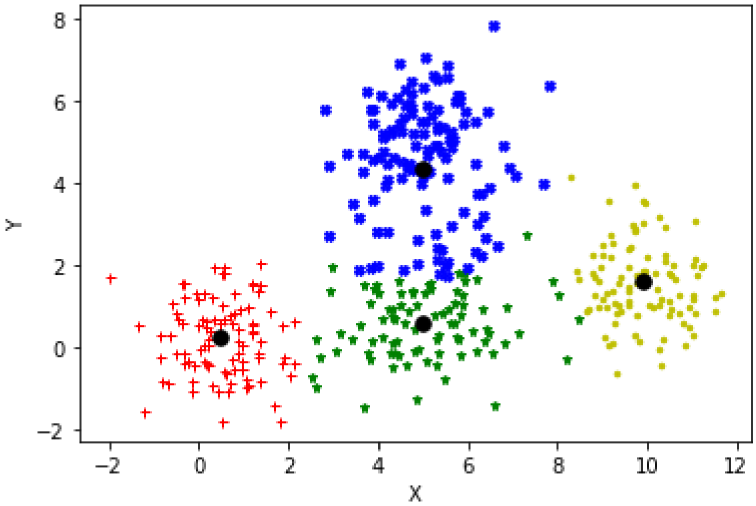


Iteration: 7

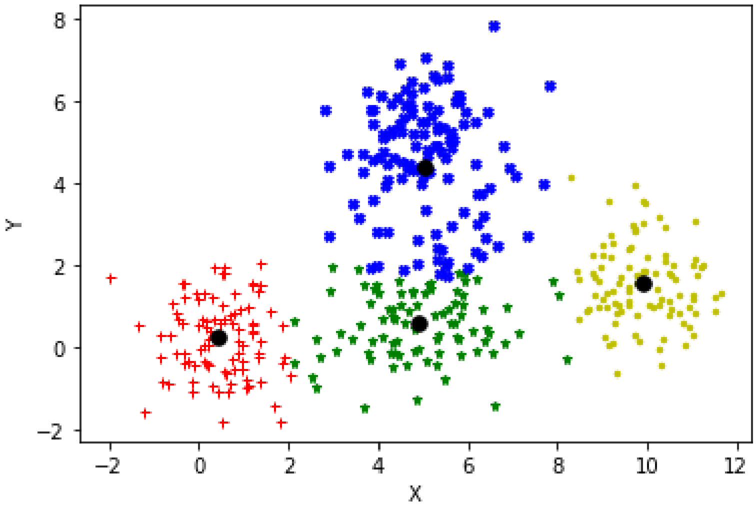




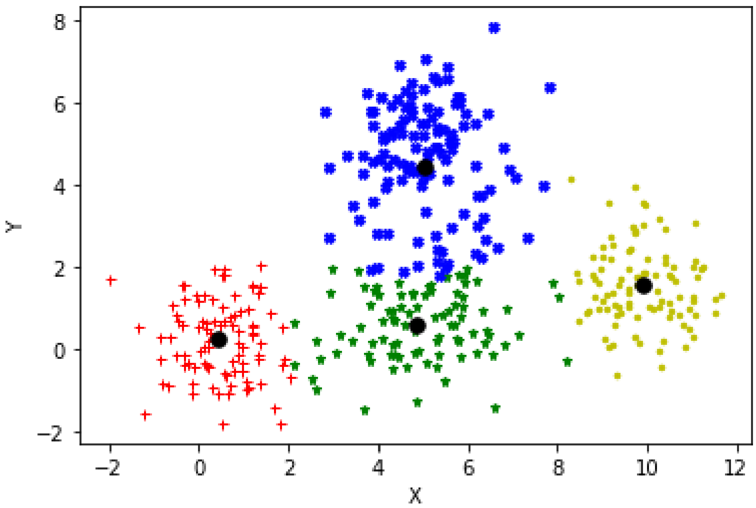
Iteration: 8



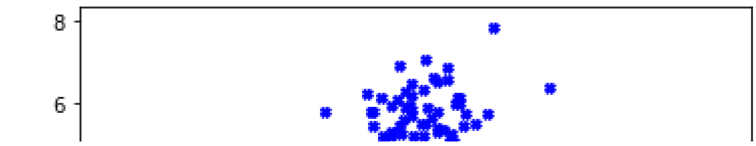
Iteration: 9

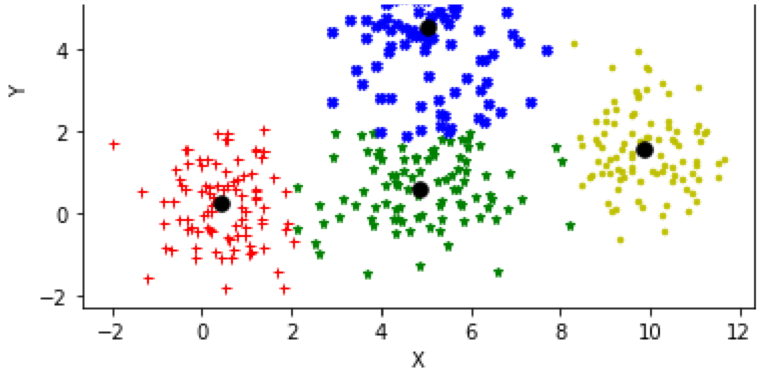


Iteration: 10

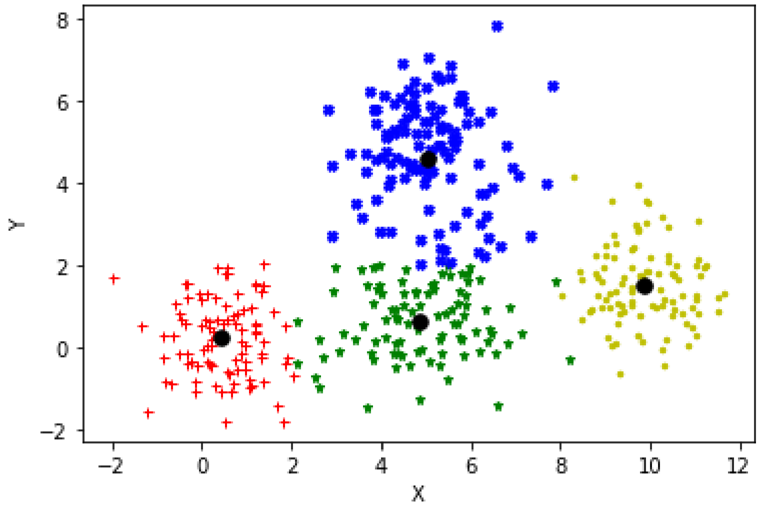


Iteration: 11

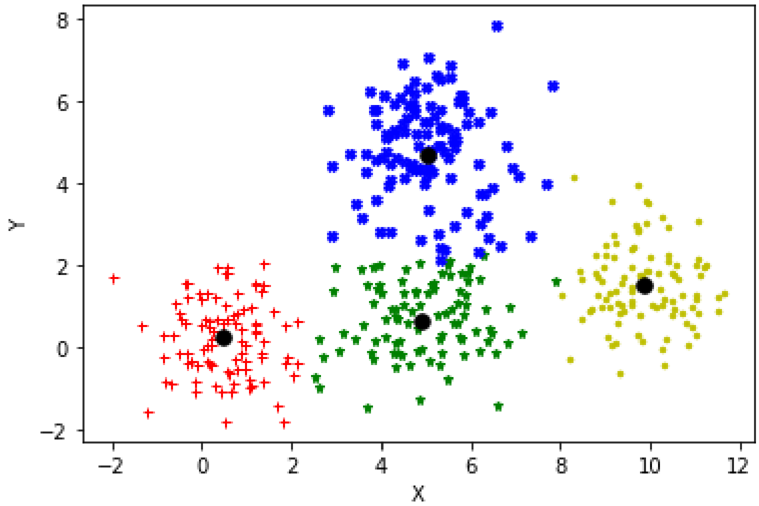




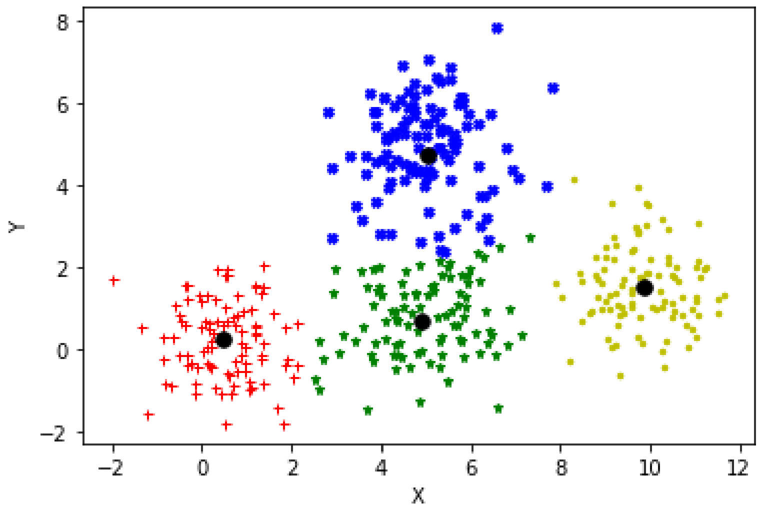
Iteration: 12



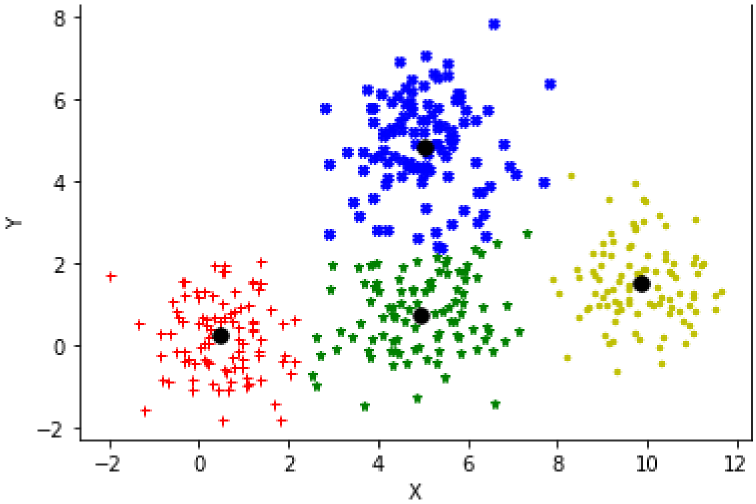
Iteration: 13



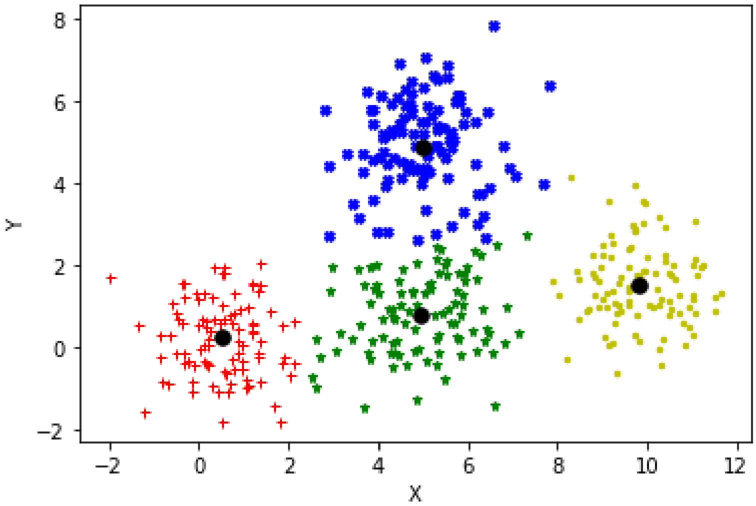
Iteration: 14



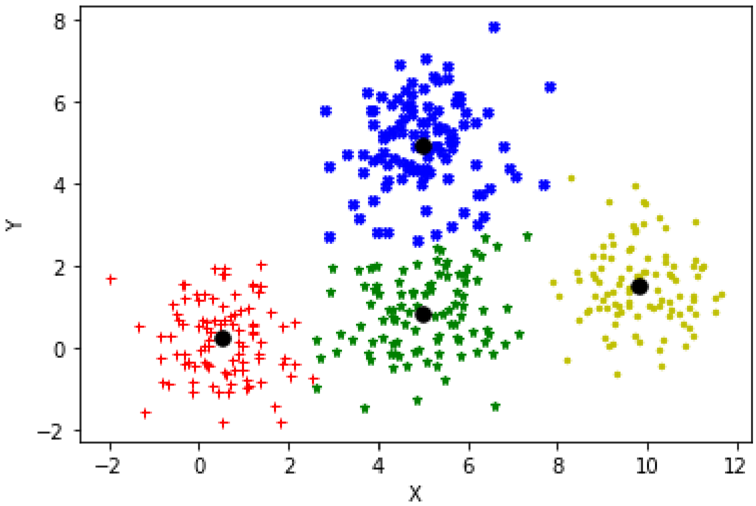
Iteration: 15



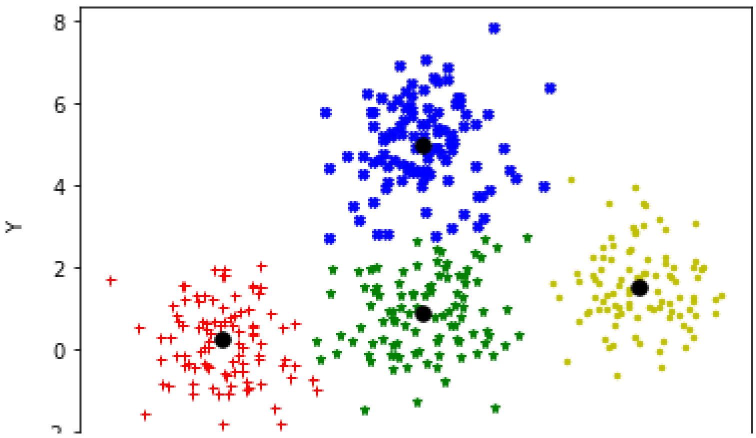
Iteration: 16

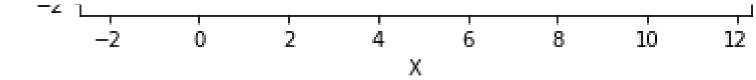


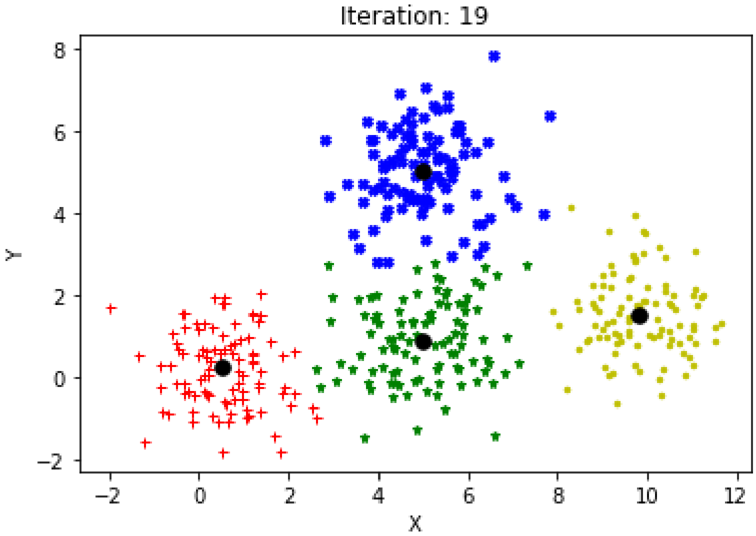
Iteration: 17

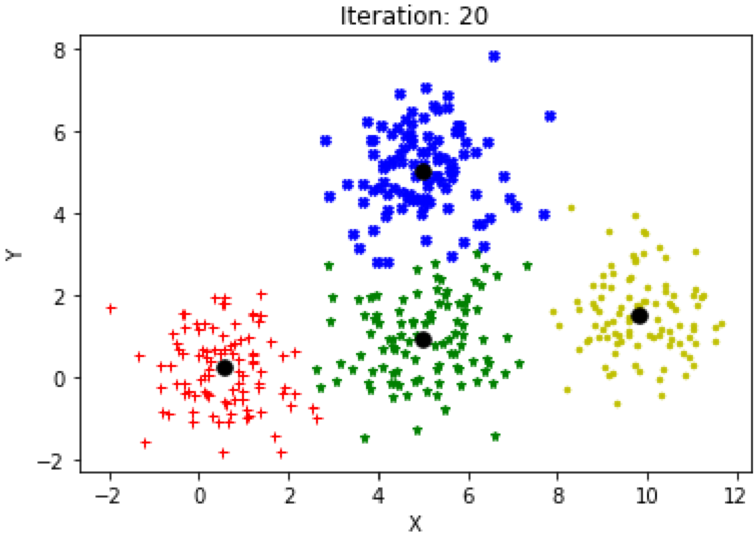


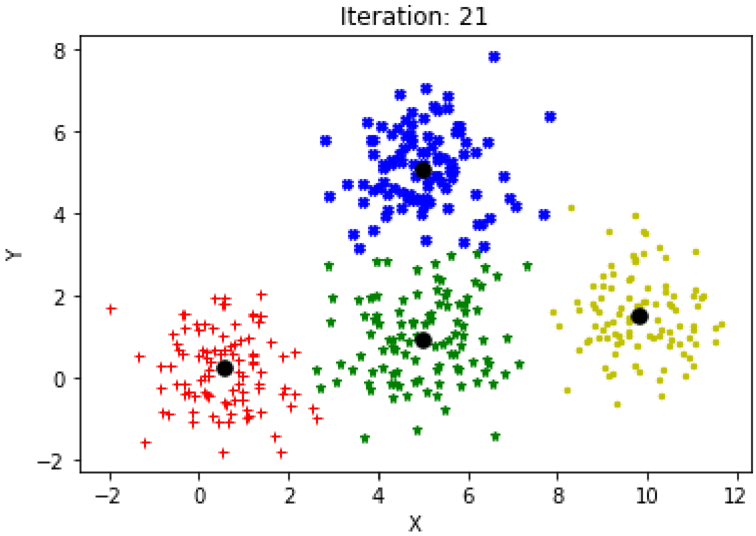
Iteration: 18

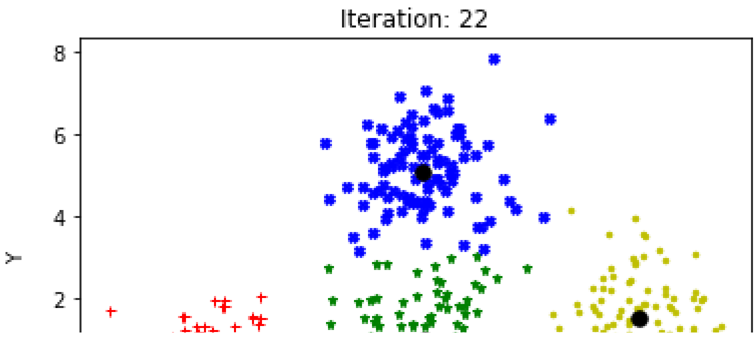


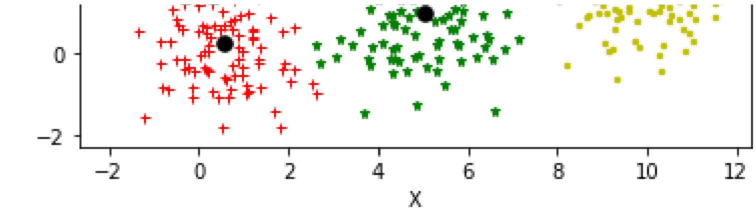

−2
0
2
4
6
8
10
12
X


Iteration: 19
X
Y

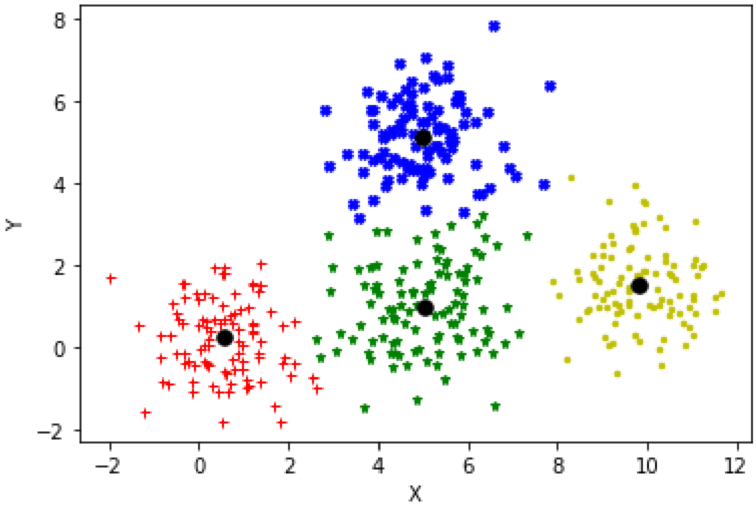

Iteration: 20
X
Y


Iteration: 21
X
Y

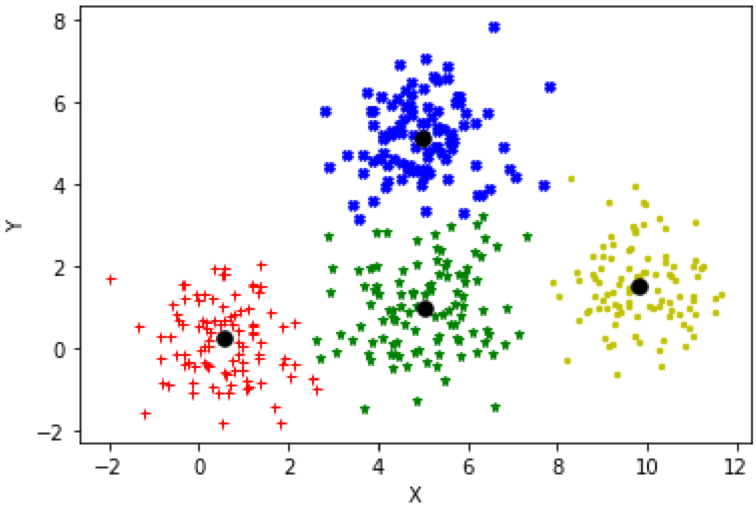

Iteration: 22
Y



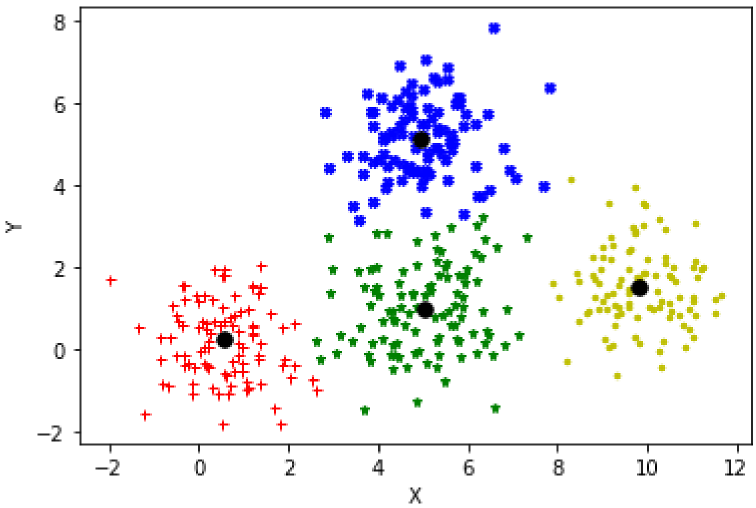
Iteration: 23



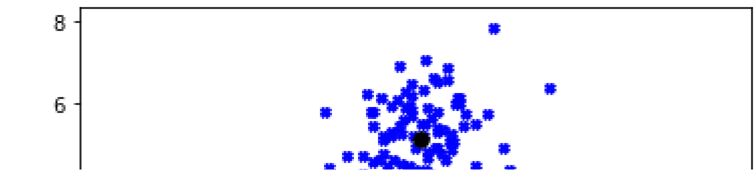
Iteration: 24

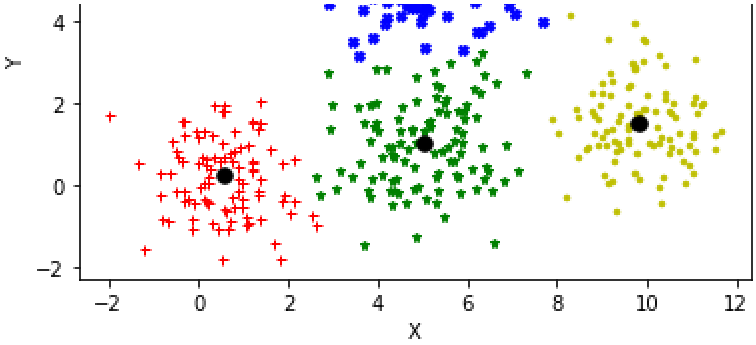


Iteration: 25

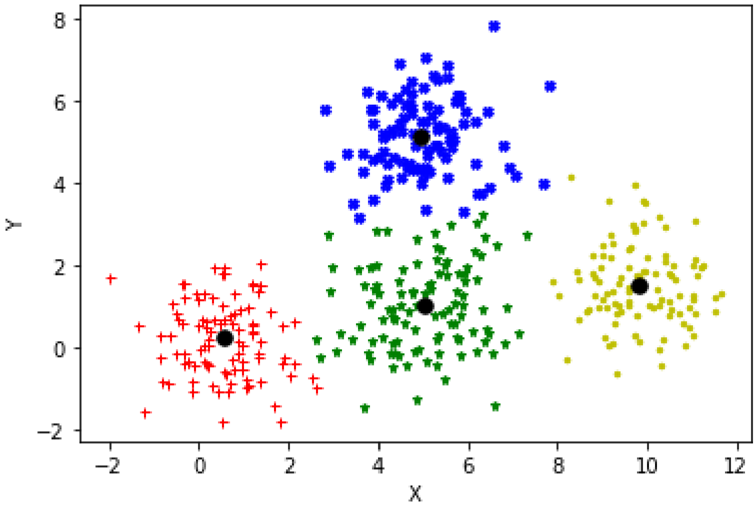


Iteration: 26

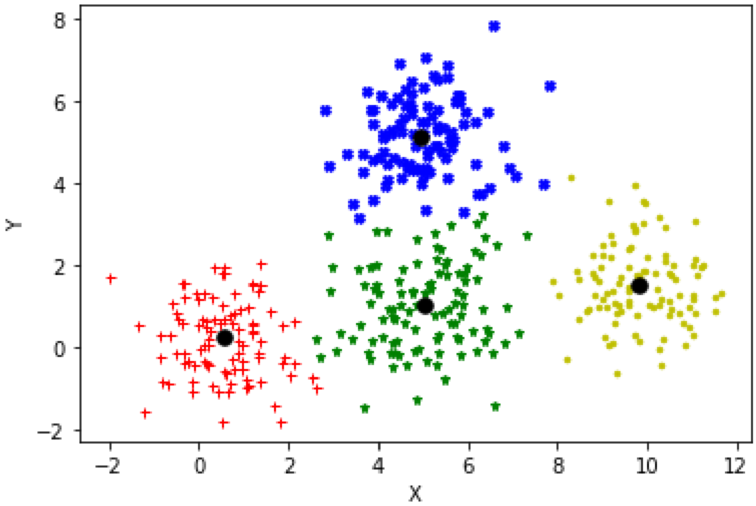




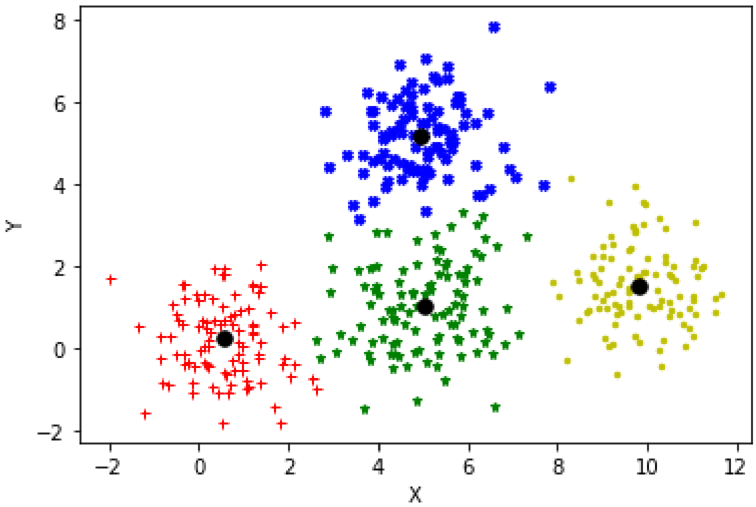
Iteration: 27



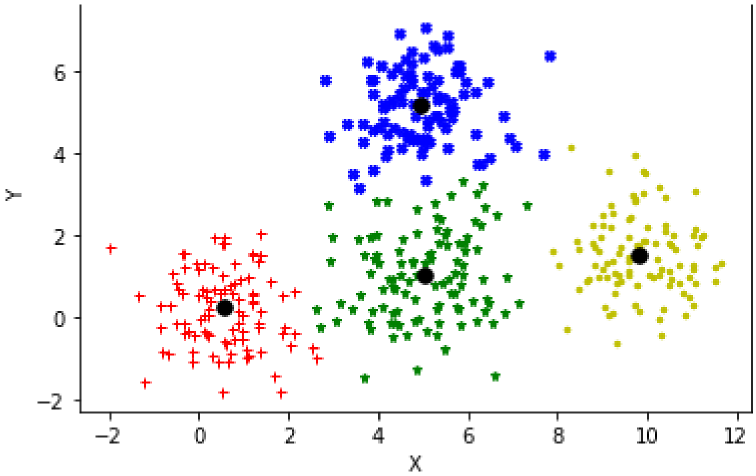
Iteration: 28



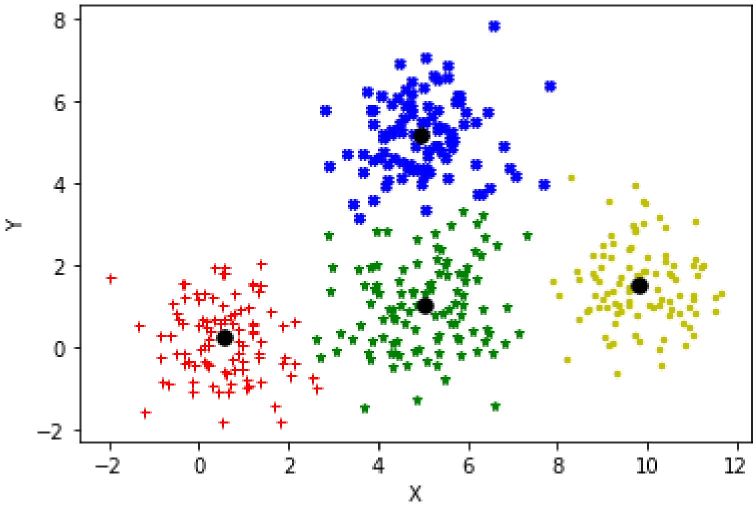
Iteration: 29



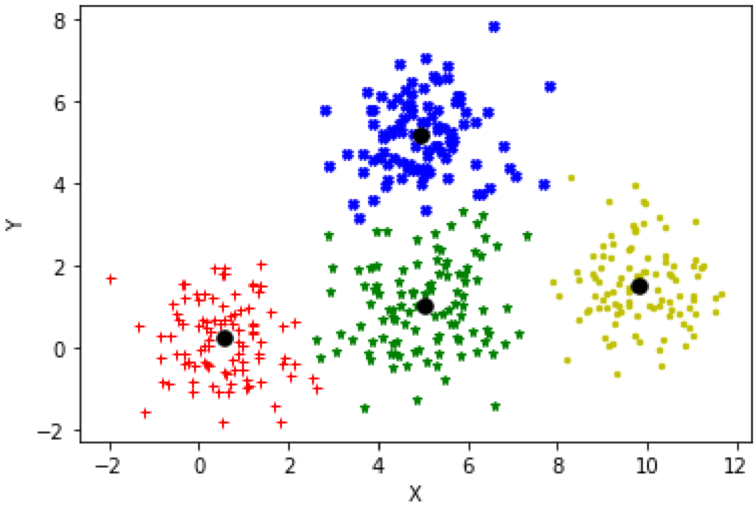
Iteration: 30



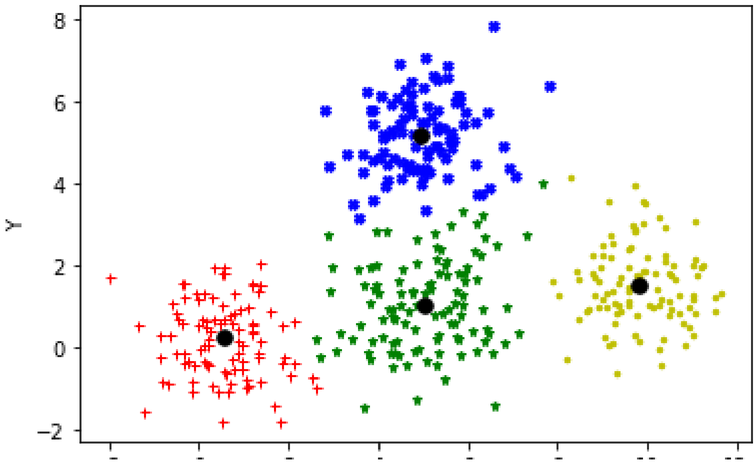
Iteration: 31



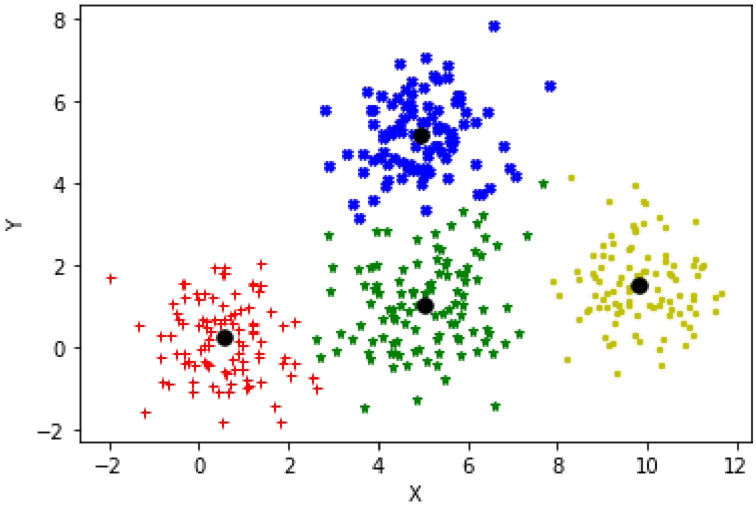
Iteration: 32



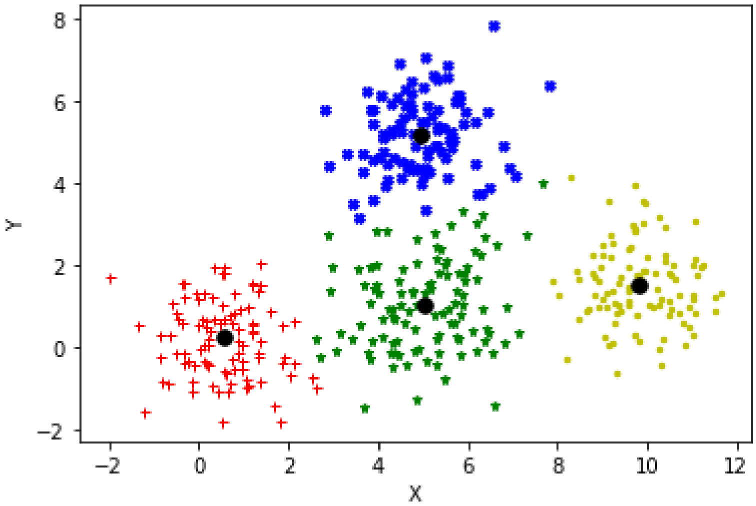
Iteration: 33



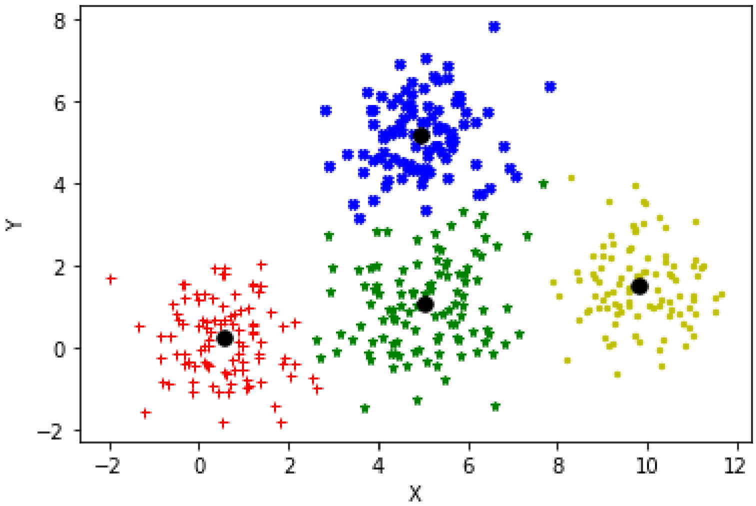
Iteration: 34



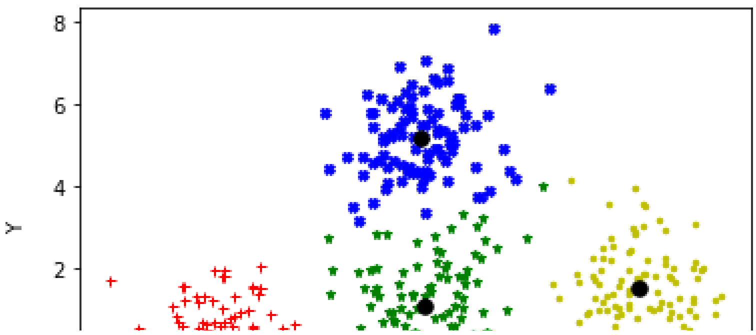
Iteration: 35

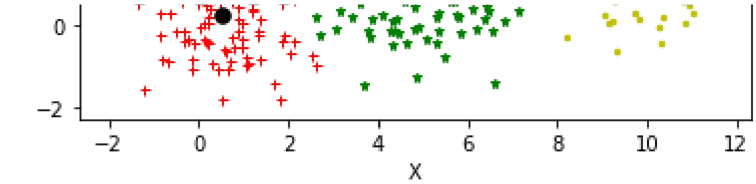


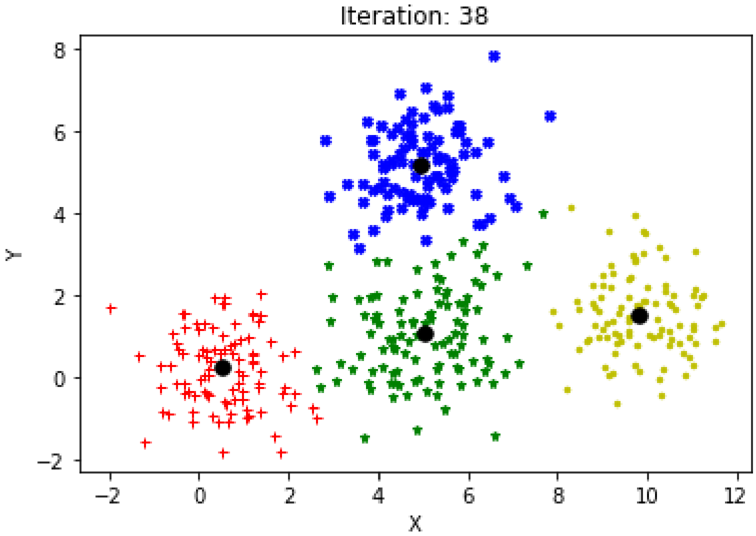
Iteration: 36

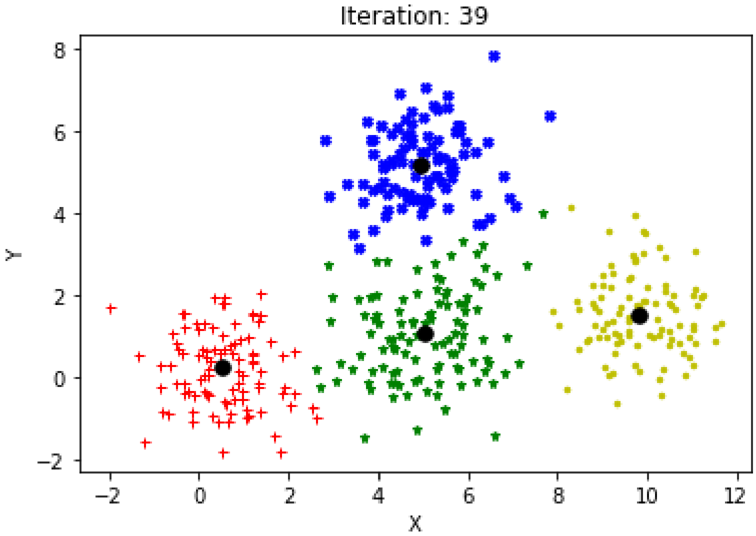


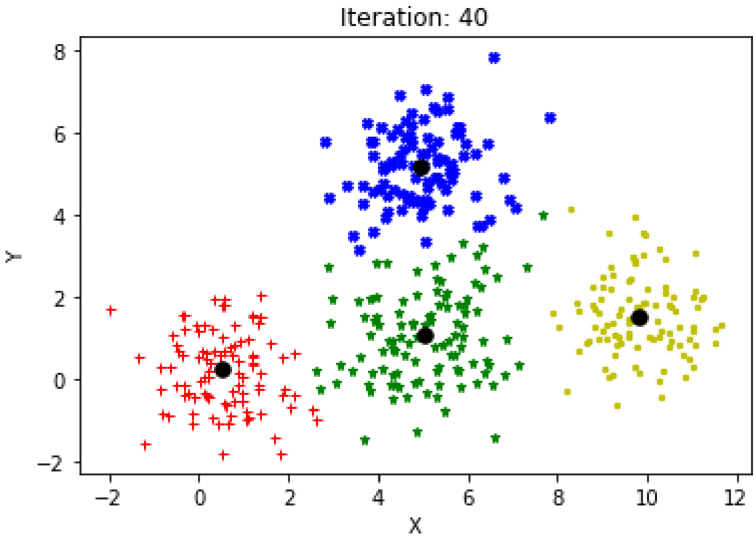
Iteration: 37

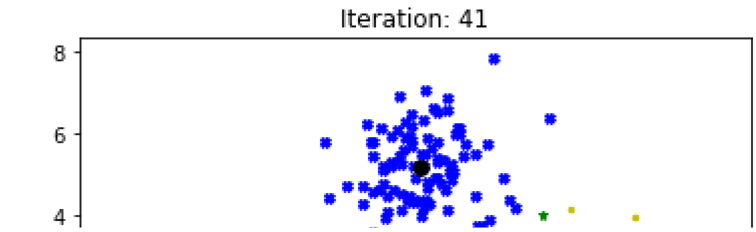


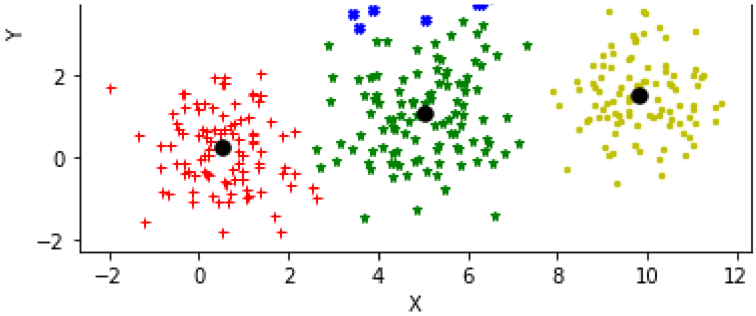

0
−2
−2
0
2
4
6
8
10
12
X


Iteration: 38
8
6
4
2
0
−2
Y
−2
0
2
4
6
8
10
12
X

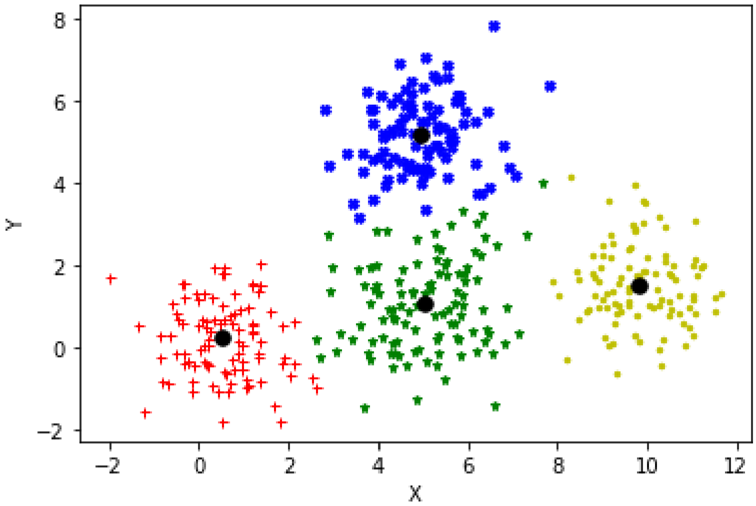

Iteration: 39
8
6
4
2
0
−2
Y
−2
0
2
4
6
8
10
12
X


Iteration: 40
8
6
4
2
0
−2
Y
−2
0
2
4
6
8
10
12
X

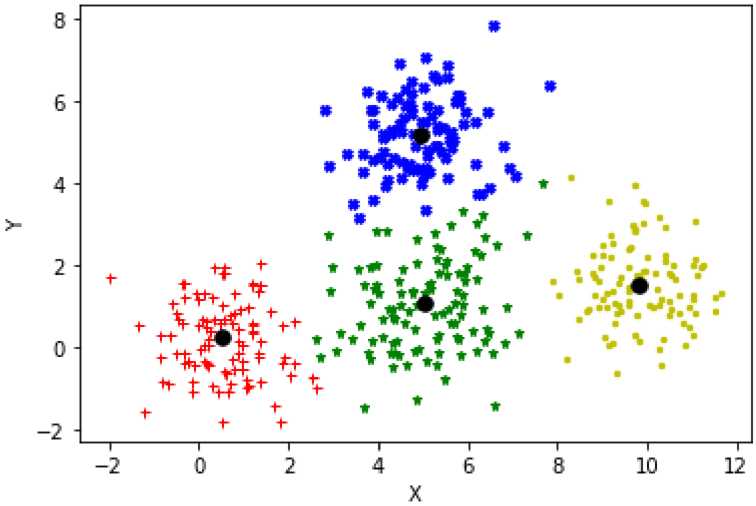

Iteration: 41
8
6
4



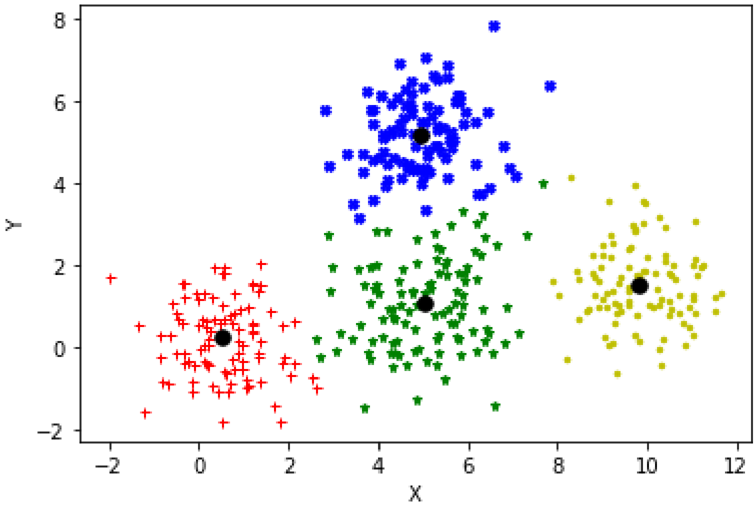
Iteration: 42



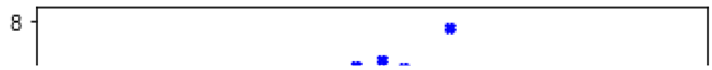
Iteration: 43

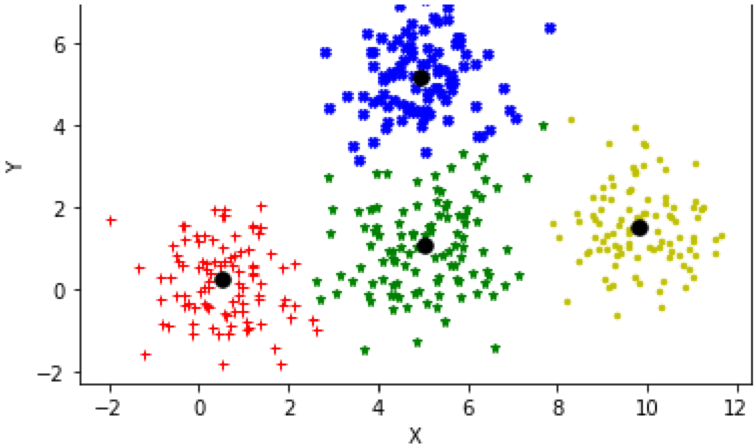


Iteration: 44

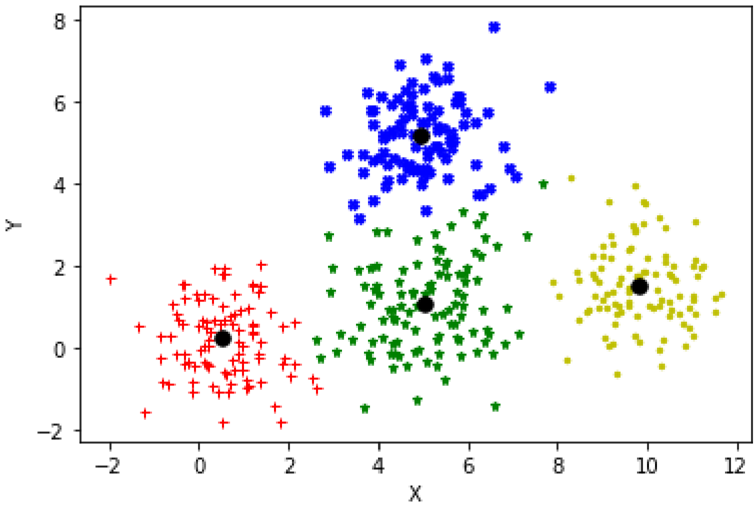


Iteration: 45

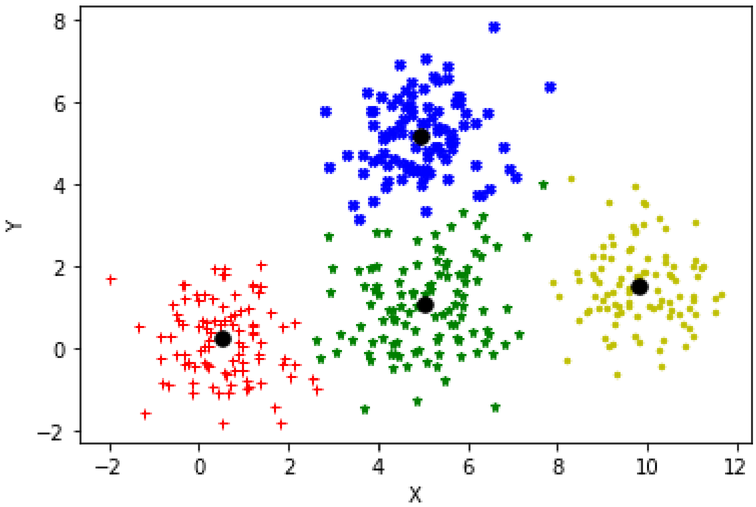




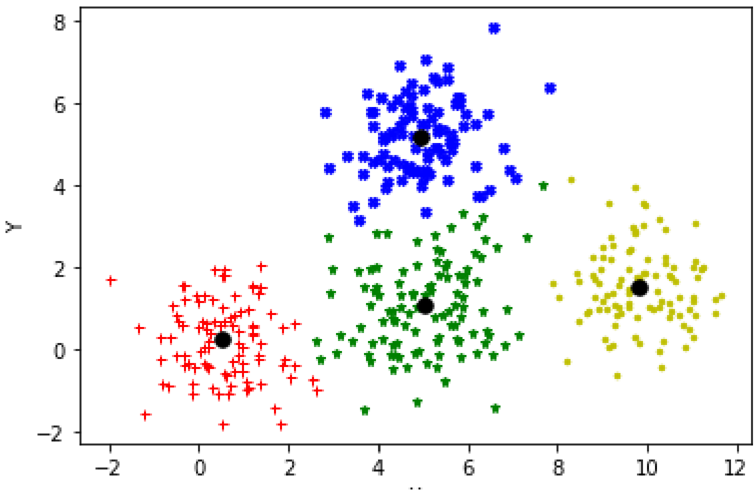
Iteration: 46

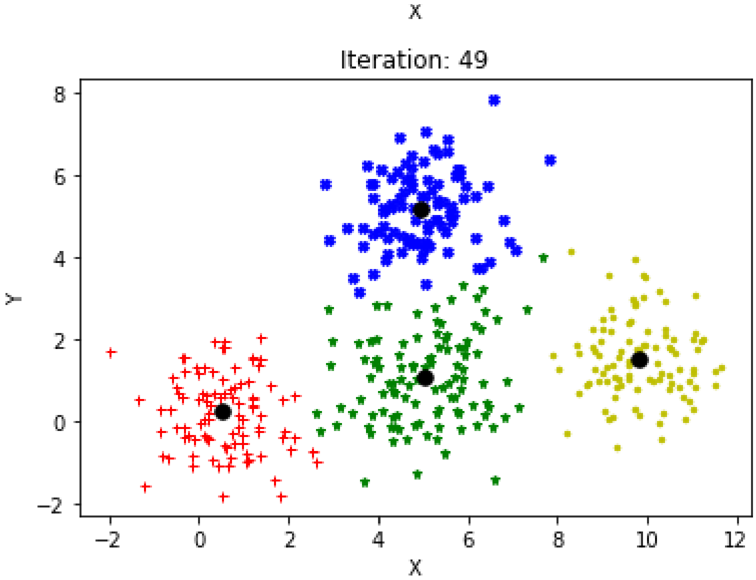


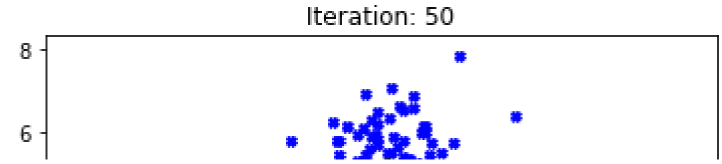
Iteration: 47



Iteration: 48







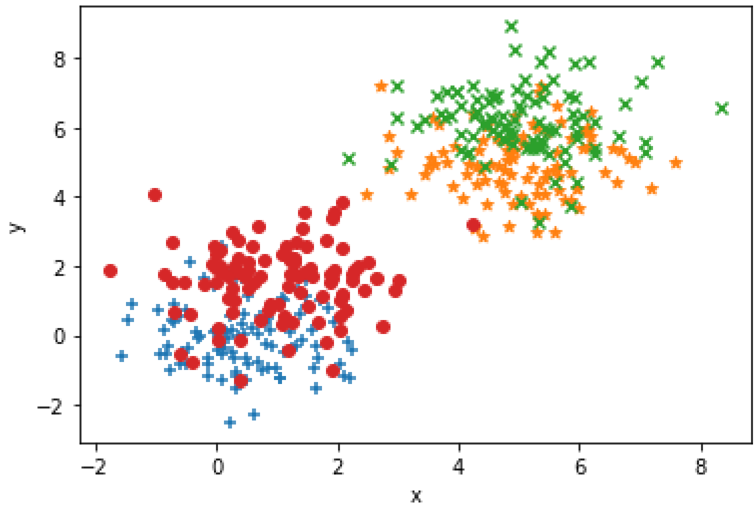
3. Write a code and report similar demonstration for Fuzzy c-means

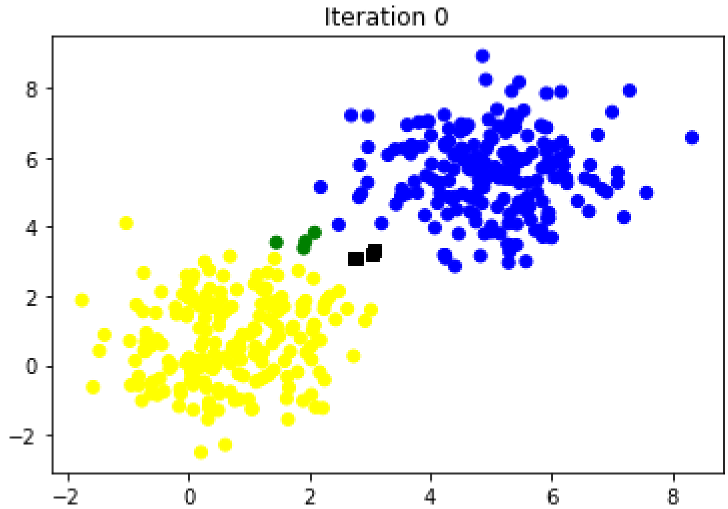
(Note : Generate the data such that you can demonstare the drawback of K-means, and able to solve through GMM and fuzzy C-means, have to demonstrate clearly during viva)

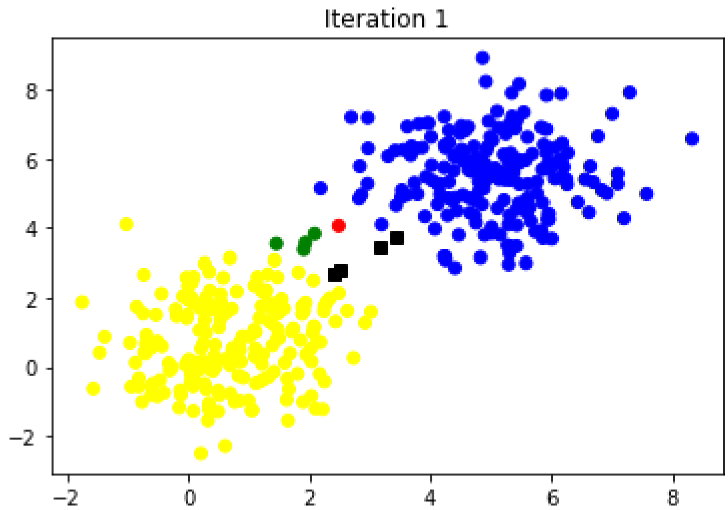
```
import numpy as np
import matplotlib.pyplot as plt
import random
import operator
import math
from scipy.spatial.distance import cdist

## Data generation
k=4
mean1 = [0.5,0]
mean2 = [5,5]
mean3 = [5,6]
mean4 = [1,1.5]
cov = np.identity(2)
x1,y1= np.random.multivariate_normal(mean1, cov, 100).T
x2, y2 = np.random.multivariate_normal(mean2, cov, 100).T
x3, y3 = np.random.multivariate_normal(mean3, cov, 100).T
x4, y4 = np.random.multivariate_normal(mean4, cov, 100).T
data1= np.stack((x1, y1), axis=1)
data2= np.stack((x2, y2), axis=1)
data3= np.stack((x3, y3), axis=1)
data4= np.stack((x4, y4), axis=1)
dataset = (data1,data2,data3,data4)
d = np.vstack(dataset)
```

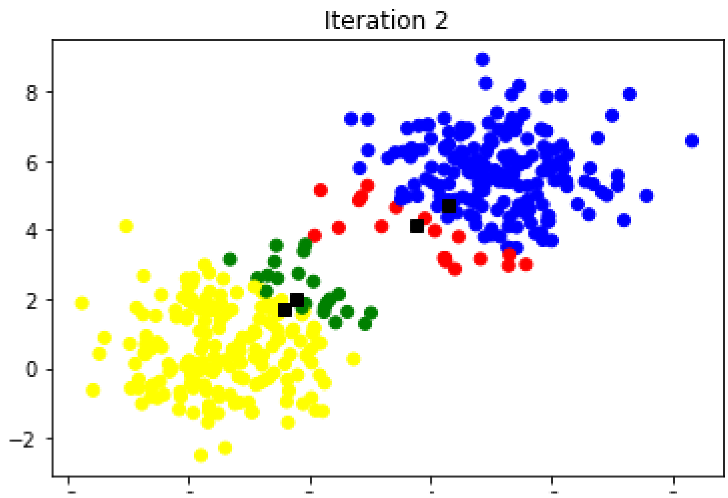
```
plt.scatter(x1,y1,marker= '+')
plt.scatter(x2,y2,marker = '*')
plt.scatter(x3,y3,marker = 'x')
plt.scatter(x4,y4,marker = 'o')
plt.xlabel('x')
plt.ylabel('y')
plt.show()

color_map = {0:'blue',1:'green',2:'yellow',3:'red'}

def initializeMembershipMatrix(n,k):
    membership_mat = list()
    for i in range(n):
        random_num_list = [random.random() for i in range(k)]
        summation = sum(random_num_list)
        temp_list = [x/summation for x in random_num_list]
        membership_mat.append(temp_list)
    return membership_mat

def calculate_cluster_center(data,membership_mat,k,r):
    cluster_mem_val = list(zip(*membership_mat))
    cluster_centers = list()
    df = pd.DataFrame(membership_mat)
    df1 = pd.DataFrame(data)
    for j in range(k):
        x = list(cluster_mem_val[j])
        xraised = [e ** r for e in x]
        denominator = sum(xraised)
        temp_num = list()
        for i in range(len(data)):
            data_point = list(df1.iloc[i])
            prod = [xraised[i] * val for val in data_point]
            temp_num.append(prod)
        numerator = map(sum, zip(*temp_num))
        center = [z/denominator for z in numerator]
        cluster_centers.append(center)
    return cluster_centers

def update_membership_values(d,membership_mat, cluster_centers,k,r):
   power = float(2 / (r - 1))
   temp = cdist(d, cluster_centers) ** power
   denominator_ = temp.reshape((d.shape[0], 1, -1)).repeat(temp.shape[-1], axis=1)
   denominator_ = temp[:, :, np.newaxis] / denominator_
   return 1 / denominator_.sum(2)

def get_clusters(membership_mat):
    cluster_labels = list()
    for i in range(len(membership_mat)):
        max_val, idx = max((val, idx) for (idx, val) in enumerate(membership_mat[i]))
        cluster_labels.append(idx)
    return cluster_labels

def error(df,centroids):
  err = 0
  a =[((df[df['cluster']==c][0]-centroids[c][0])**2) + ((df[df['cluster']==c][1]-centroids
```

```
    for i in range(len(a)):
      err = err + sum(a[i])
    return err/len(df)

  def fuzzyCMeansClustering(itera):
      membership_mat = initializeMembershipMatrix(len(d),4)
      curr = 0
      error_list = []
      while curr <= itera:
          cluster_centers = calculate_cluster_center(d,membership_mat,4,2)
          membership_mat = update_membership_values(d,membership_mat, cluster_centers,4,2)
          cluster_labels = get_clusters(membership_mat)
          df = pd.DataFrame(d)
          df['cluster']=cluster_labels
          df['color']=df['cluster'].map(lambda x:color_map[x])
          plt.scatter(df[0],df[1],color = df['color'])
          for c in range(0,k):
            plt.plot(*cluster_centers[c],'s',color='black')
          plt.title('Iteration {}'.format(curr))
          plt.show()
          curr += 1
          err=error(df,cluster_centers)
          error_list.append(err)
      return cluster_labels, cluster_centers,error_list
  labels, centers,error_list = fuzzyCMeansClustering(10)
  it=[]
  for i in range(11):
      it.append(i)
  plt.plot(it,error_list)
  plt.title("error")
  plt.show()
```

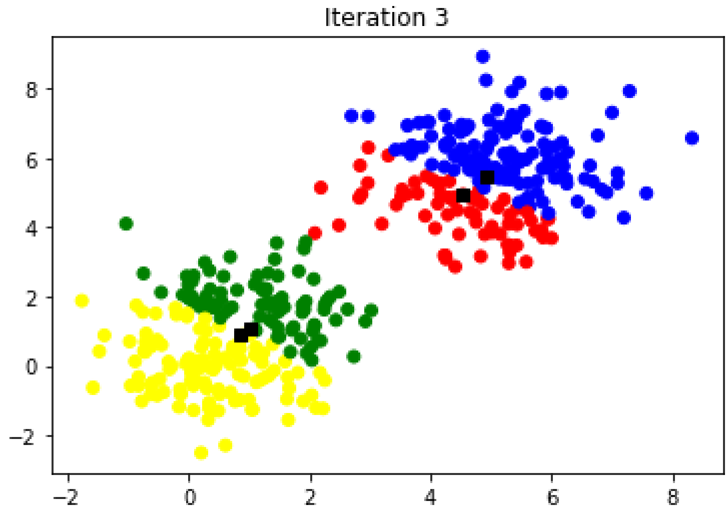

x
y

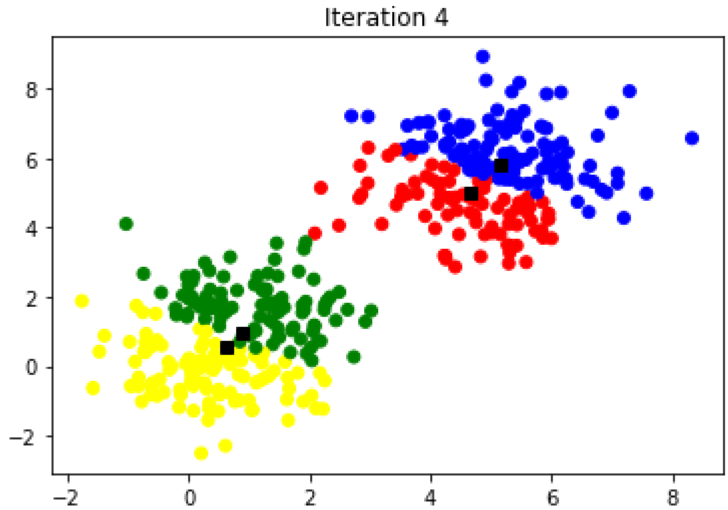

Iteration 0

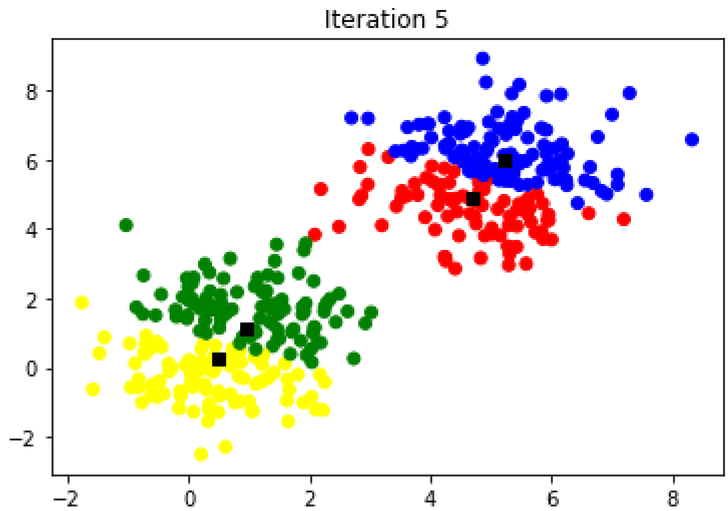

Iteration 1

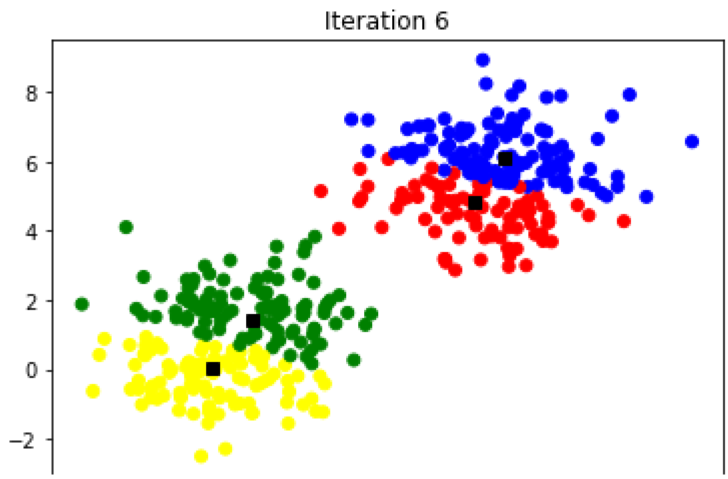

Iteration 2

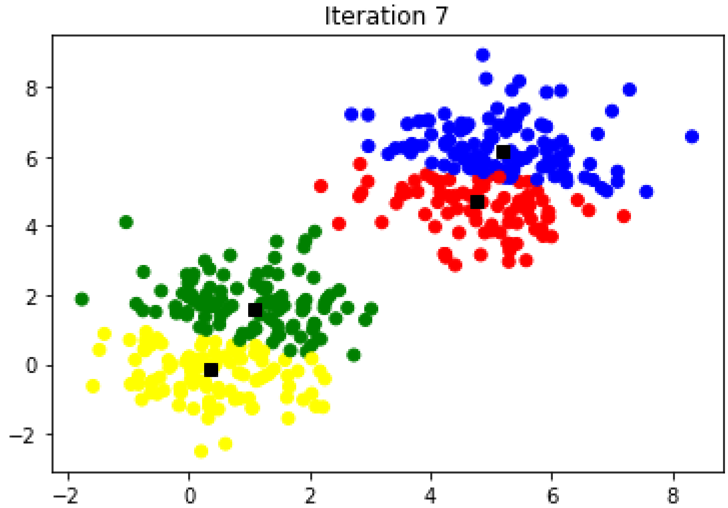

−2 0 2 4 6 8

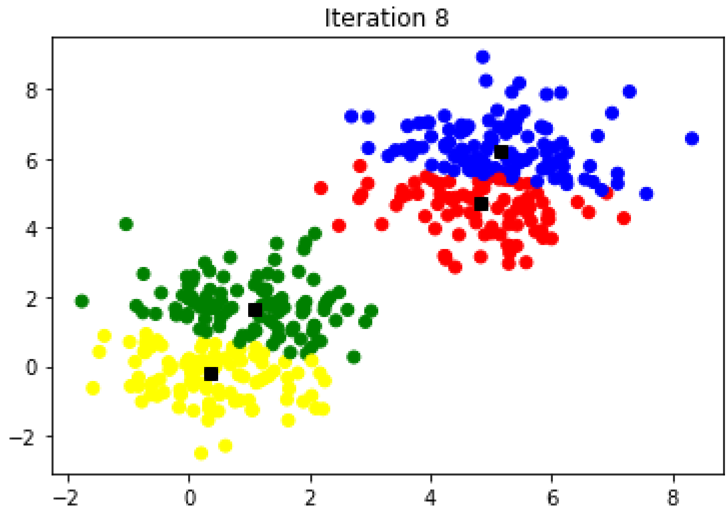

Iteration 3

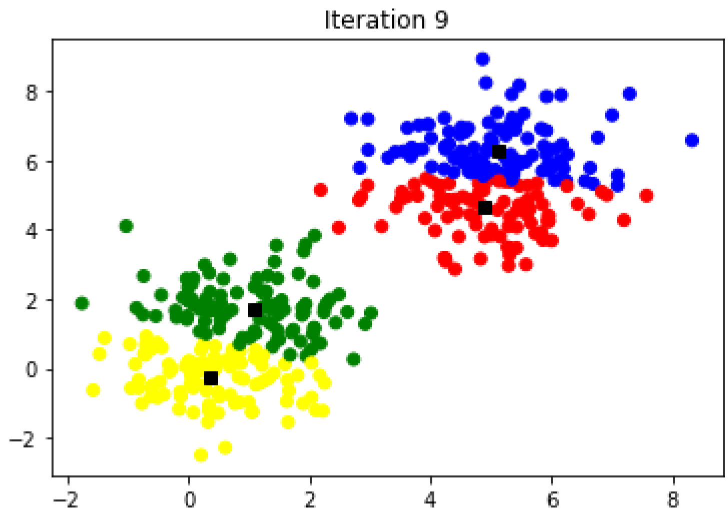

Iteration 4

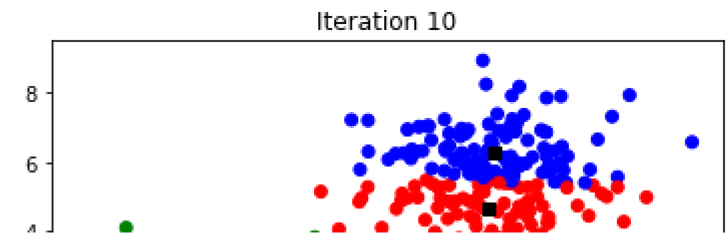

Iteration 5


Iteration 6









4. Practical Example

Using K-means

a) Data preparation

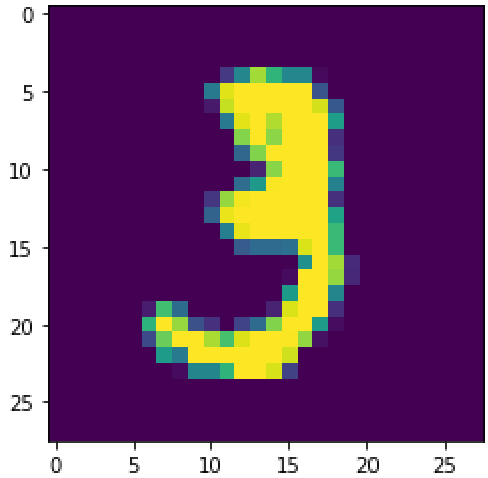
1. Load Mnist data
2. Take only two class '1' and '5'

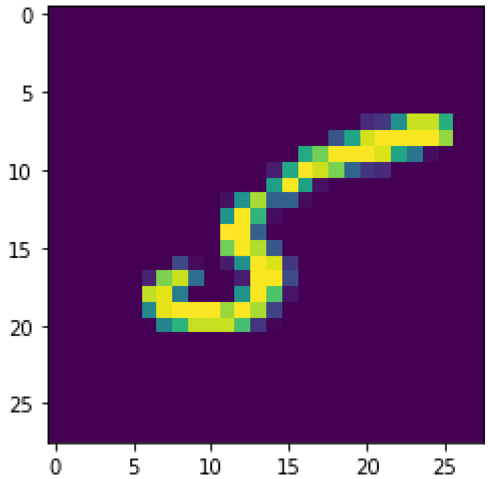
```
from google.colab import drive
drive.mount('/gdrive')
!pip install idx2numpy
```

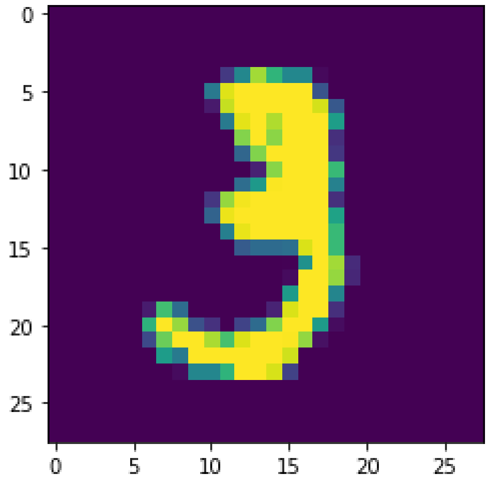
```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947:

Enter your authorization code:
··········
Mounted at /gdrive
Collecting idx2numpy
  Downloading https://files.pythonhosted.org/packages/23/6b/abab4652eb249f432c624319(
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from
Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from ic
Building wheels for collected packages: idx2numpy
  Building wheel for idx2numpy (setup.py) ... done
  Created wheel for idx2numpy: filename=idx2numpy-1.2.2-cp36-none-any.whl size=8032 s
  Stored in directory: /root/.cache/pip/wheels/7a/b5/69/3e0757b3086607e95db70661798fc
Successfully built idx2numpy
Installing collected packages: idx2numpy
Successfully installed idx2numpy-1.2.2
```

```
import numpy as np
import matplotlib.pyplot as plt

"""file1='/gdrive/My Drive/Machine learning workshop blr/Colab_notebooks/train-images.idx3
file2='/gdrive/My Drive/Machine learning workshop blr/Colab_notebooks/train-labels.idx1-ub

import idx2numpy

Images= idx2numpy.convert_from_file(file1)
labels= idx2numpy.convert_from_file(file2)"""

file1='train-images.idx3-ubyte'
file2='train-labels.idx1-ubyte'
import idx2numpy

Images= idx2numpy.convert_from_file(file1)
labels= idx2numpy.convert_from_file(file2)
Images = Images.reshape((Images.shape[0], Images.shape[1]*Images.shape[1])
# write you code here
for i in range(3):
  plt.figure()
  plt.imshow(Images[i].reshape((28, 28)), cmap='viridis')
  plt.show()
```

```
(11552, 784)
(11552,)
(11552, 784)
[[0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
```







2. Write a function of Kmeans as written earlier

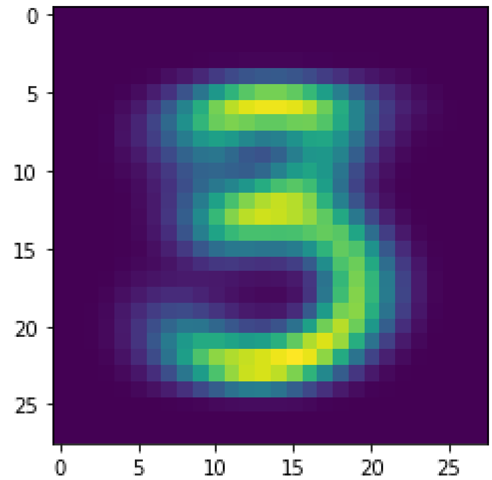
```
# k-means
class_1 = Images[labels == 1]
```

```
class_5 = Images[labels == 5]
X = np.concatenate([class_1, class_5], axis=0)

print(class_1.shape, class_5.shape, X.shape)
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)
mu = kmeans.cluster_centers_
```

3. Call the K-means function and plot the mean vectors of the cluster

```
sse = {}
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(X)
    sse[k] = kmeans.inertia_ # Inertia: Sum of distances of samples to their closest clust
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.ylabel("SSE")
plt.show()
plt.figure()
plt.imshow(np.reshape(mu[0,:],(28,28)))
plt.show()

plt.figure()
plt.imshow(np.reshape(mu[1,:],(28,28)))
plt.show()
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
[<matplotlib.lines.Line2D at 0x7f2c28ffca20>]
```



5. Perform the same task for GMM and fuzzy c-means

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random

a = int(''.join(format(ord(i), 'b') for i in 'b')[:6])
np.random.seed(a),random.seed(a)

#def data_plot():
col = ['+g','*r','xy','.b']

from scipy.spatial.distance import euclidean as distance
from scipy.stats import multivariate_normal
class GMM:
  def __init__(self, k: int, n_iters: int, tol: float):
    self.n_components, self.n_iters,self.tol  = k, n_iters, tol
    np.random.seed(a),random.seed(a)

  def _do_estep(self, X):

    for k in range(self.n_components):
      prior = self.weights[k]
      likelihood = multivariate_normal(self.means[k], self.covs[k]).pdf(X)
      self.resp[:, k] = prior * likelihood

    log_likelihood = np.sum(np.log(np.sum(self.resp, axis = 1)))

        # normalize over all possible cluster assignments
    self.resp = self.resp / self.resp.sum(axis = 1, keepdims = 1)
    return log_likelihood

  def _do_mstep(self, X):
        # total responsibility assigned to each cluster, N^{soft}
    resp_weights = self.resp.sum(axis = 0)
        # weights
    self.weights = resp_weights / X.shape[0]
        # means
    weighted_sum = np.dot(self.resp.T, X)
    self.means = weighted_sum / resp_weights.reshape(-1, 1)
        # covariance
    for k in range(self.n_components):
      diff = (X - self.means[k]).T
      weighted_sum = np.dot(self.resp[:, k] * diff, diff.T)
      self.covs[k] = weighted_sum / resp_weights[k]

  def fit(self, X):
        # data's responsibility vector
    self.resp = np.zeros((X.shape[0], self.n_components))
```

```
    self.resp = np.zeros((X.shape[0], self.n_components))

        # initialize parameters #self.covs = np.full(shape, np.cov(X.T))
    self.means = X[np.random.choice(X.shape[0], self.n_components)]
    self.weights = np.full(self.n_components, 1 / self.n_components)
    self.covs = np.full((self.n_components, X.shape[1], X.shape[1]), cor*np.max(np.asarray
    log_likelihood, self.log_likelihood_trace, clr, mrk = 0, [], ['g','b','y','r'], ['*','

    for i in range(self.n_iters):
      log_likelihood_new = self._do_estep(X)
      self._do_mstep(X)
      cluster_label=np.argmax(self.resp,axis=1) #Label Points
      for l in range(self.n_components):
        id=np.where(cluster_label==l)
        plt.plot(X[id,0],X[id,1],'.',color=clr[l],marker=mrk[l],markersize=5)
      plt.plot(self.means[:,0],self.means[:,1],'.',color='black',markersize=15,label="Mean
      plt.xlabel('X')
      plt.ylabel('Y')
      plt.title('Iteration: '+str(i+1))
      plt.show()
      if abs(log_likelihood_new - log_likelihood) <= self.tol:
        print("Converged")
        break
            #print("Difference in Log Likelihood: "+str(abs(log_likelihood_new - log_likel
      log_likelihood = log_likelihood_new
      self.log_likelihood_trace.append(log_likelihood)
    plt.plot(np.asarray(self.log_likelihood_trace))
    plt.xlabel('no.of iterations')
    plt.ylabel('Log Likelihood')
    plt.title('Log Likelihood Trace')
    plt.show()
gmm = GMM(k = 2, n_iters = 60, tol = 1e-4)
gmm.fit(X)
```

```
import numpy as np
import matplotlib.pyplot as plt
import random
import operator
import math
from scipy.spatial.distance import cdist
k=2
d=X
color_map = {0:'blue',1:'green',2:'yellow',3:'red'}

def initializeMembershipMatrix(n,k):
    membership_mat = list()
    for i in range(n):
        random_num_list = [random.random() for i in range(k)]
        summation = sum(random_num_list)
        temp_list = [x/summation for x in random_num_list]
        membership_mat.append(temp_list)
    return membership_mat
```

```
def calculate_cluster_center(data,membership_mat,k,r):
    cluster_mem_val = list(zip(*membership_mat))
    cluster_centers = list()
    df = pd.DataFrame(membership_mat)
    df1 = pd.DataFrame(data)
    for j in range(k):
        x = list(cluster_mem_val[j])
        xraised = [e ** r for e in x]
        denominator = sum(xraised)
        temp_num = list()
        for i in range(len(data)):
            data_point = list(df1.iloc[i])
            prod = [xraised[i] * val for val in data_point]
            temp_num.append(prod)
        numerator = map(sum, zip(*temp_num))
        center = [z/denominator for z in numerator]
        cluster_centers.append(center)
    return cluster_centers

def update_membership_values(d,membership_mat, cluster_centers,k,r):
   power = float(2 / (r - 1))
   temp = cdist(d, cluster_centers) ** power
   denominator_ = temp.reshape((d.shape[0], 1, -1)).repeat(temp.shape[-1], axis=1)
   denominator_ = temp[:, :, np.newaxis] / denominator_
   return 1 / denominator_.sum(2)

def get_clusters(membership_mat):
    cluster_labels = list()
    for i in range(len(membership_mat)):
        max_val, idx = max((val, idx) for (idx, val) in enumerate(membership_mat[i]))
        cluster_labels.append(idx)
    return cluster_labels

def error(df,centroids):
  err = 0
  a =[((df[df['cluster']==c][0]-centroids[c][0])**2) + ((df[df['cluster']==c][1]-centroids
  for i in range(len(a)):
    err = err + sum(a[i])
  return err/len(df)

def fuzzyCMeansClustering(itera):
    membership_mat = initializeMembershipMatrix(len(d),4)
    curr = 0
    error_list = []
    while curr <= itera:
        cluster_centers = calculate_cluster_center(d,membership_mat,4,2)
        membership_mat = update_membership_values(d,membership_mat, cluster_centers,4,2)
        cluster_labels = get_clusters(membership_mat)
        df = pd.DataFrame(d)
        df['cluster']=cluster_labels
        df['color']=df['cluster'].map(lambda x:color_map[x])
        plt.scatter(df[0],df[1],color = df['color'])
        for c in range(0,k):
          plt.plot(*cluster_centers[c],'s',color='black')
        plt.title('Iteration {}'.format(curr))
```

```
        plt.show()
        curr += 1
        err=error(df,cluster_centers)
        error_list.append(err)
    return cluster_labels, cluster_centers,error_list
labels, centers,error_list = fuzzyCMeansClustering(10)
it=[]
for i in range(11):
    it.append(i)
plt.plot(it,error_list)
plt.title("error")
plt.show()
```

6. Repeat the same for 3 class and perform the K-means, GMM and Fuzzy c-means clustering

```
# k-means
class_1 = Images[labels == 1]
class_5 = Images[labels == 5]
class_2 = Images[labels == 2]
X = np.concatenate([class_1, class_5,class_2], axis=0)

print(class_1.shape, class_5.shape, class_2.shape,X.shape)
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
kmeans.fit(X)
mu = kmeans.cluster_centers_
sse = {}
for k in range(1, 10):
    kmeans = KMeans(n_clusters=k, max_iter=1000).fit(X)
    sse[k] = kmeans.inertia_ # Inertia: Sum of distances of samples to their closest clust
plt.figure()
plt.plot(list(sse.keys()), list(sse.values()))
plt.xlabel("Number of cluster")
plt.ylabel("SSE")
plt.show()
plt.figure()
plt.imshow(np.reshape(mu[0,:],(28,28)))
plt.show()
plt.figure()
plt.imshow(np.reshape(mu[1,:],(28,28)))
plt.show()
plt.figure()
plt.imshow(np.reshape(mu[2,:],(28,28)))
plt.show()
```

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

```
import random

a = int(''.join(format(ord(i), 'b') for i in 'b')[:6])
np.random.seed(a),random.seed(a)

#def data_plot():
col = ['+g','*r','xy','.b']

from scipy.spatial.distance import euclidean as distance
from scipy.stats import multivariate_normal
class GMM:
  def __init__(self, k: int, n_iters: int, tol: float):
    self.n_components, self.n_iters,self.tol  = k, n_iters, tol
    np.random.seed(a),random.seed(a)

  def _do_estep(self, X):

    for k in range(self.n_components):
      prior = self.weights[k]
      likelihood = multivariate_normal(self.means[k], self.covs[k]).pdf(X)
      self.resp[:, k] = prior * likelihood

    log_likelihood = np.sum(np.log(np.sum(self.resp, axis = 1)))

        # normalize over all possible cluster assignments
    self.resp = self.resp / self.resp.sum(axis = 1, keepdims = 1)
    return log_likelihood

  def _do_mstep(self, X):
        # total responsibility assigned to each cluster, N^{soft}
    resp_weights = self.resp.sum(axis = 0)
        # weights
    self.weights = resp_weights / X.shape[0]
        # means
    weighted_sum = np.dot(self.resp.T, X)
    self.means = weighted_sum / resp_weights.reshape(-1, 1)
        # covariance
    for k in range(self.n_components):
      diff = (X - self.means[k]).T
      weighted_sum = np.dot(self.resp[:, k] * diff, diff.T)
      self.covs[k] = weighted_sum / resp_weights[k]

  def fit(self, X):
        # data's responsibility vector
    self.resp = np.zeros((X.shape[0], self.n_components))

    self.means = X[np.random.choice(X.shape[0], self.n_components)]
    self.weights = np.full(self.n_components, 1 / self.n_components)
    self.covs = np.full((self.n_components, X.shape[1], X.shape[1]), cor*np.max(np.asarray
    log_likelihood, self.log_likelihood_trace, clr, mrk = 0, [], ['g','b','y','r'], ['*','

    for i in range(self.n_iters):
      log_likelihood_new = self._do_estep(X)
      self._do_mstep(X)
      cluster_label=np.argmax(self.resp,axis=1) #Label Points
```

```
      for l in range(self.n_components):
        id=np.where(cluster_label==l)
        plt.plot(X[id,0],X[id,1],'.',color=clr[l],marker=mrk[l],markersize=5)
      plt.plot(self.means[:,0],self.means[:,1],'.',color='black',markersize=15,label="Mean
      plt.xlabel('X')
      plt.ylabel('Y')
      plt.title('Iteration: '+str(i+1))
      plt.show()
      if abs(log_likelihood_new - log_likelihood) <= self.tol:
        print("Converged")
        break
      log_likelihood = log_likelihood_new
      self.log_likelihood_trace.append(log_likelihood)
    plt.plot(np.asarray(self.log_likelihood_trace))
    plt.xlabel('no.of iterations')
    plt.ylabel('Log Likelihood')
    plt.title('Log Likelihood Trace')
    plt.show()
gmm = GMM(k = 3, n_iters = 60, tol = 1e-4)
gmm.fit(X)


import numpy as np
import matplotlib.pyplot as plt
import random
import operator
import math
from scipy.spatial.distance import cdist
k=2
d=X
color_map = {0:'blue',1:'green',2:'yellow',3:'red'}

def initializeMembershipMatrix(n,k):
    membership_mat = list()
    for i in range(n):
        random_num_list = [random.random() for i in range(k)]
        summation = sum(random_num_list)
        temp_list = [x/summation for x in random_num_list]
        membership_mat.append(temp_list)
    return membership_mat

def calculate_cluster_center(data,membership_mat,k,r):
    cluster_mem_val = list(zip(*membership_mat))
    cluster_centers = list()
    df = pd.DataFrame(membership_mat)
    df1 = pd.DataFrame(data)
    for j in range(k):
        x = list(cluster_mem_val[j])
        xraised = [e ** r for e in x]
        denominator = sum(xraised)
        temp_num = list()
        for i in range(len(data)):
            data_point = list(df1.iloc[i])
            prod = [xraised[i] * val for val in data_point]
            temp_num.append(prod)
        numerator = map(sum, zip(*temp_num))
```

```
        numerator = map(sum, zip(*temp_num))
        center = [z/denominator for z in numerator]
        cluster_centers.append(center)
    return cluster_centers

def update_membership_values(d,membership_mat, cluster_centers,k,r):
   power = float(2 / (r - 1))
   temp = cdist(d, cluster_centers) ** power
   denominator_ = temp.reshape((d.shape[0], 1, -1)).repeat(temp.shape[-1], axis=1)
   denominator_ = temp[:, :, np.newaxis] / denominator_
   return 1 / denominator_.sum(2)

def get_clusters(membership_mat):
    cluster_labels = list()
    for i in range(len(membership_mat)):
        max_val, idx = max((val, idx) for (idx, val) in enumerate(membership_mat[i]))
        cluster_labels.append(idx)
    return cluster_labels

def error(df,centroids):
  err = 0
  a =[((df[df['cluster']==c][0]-centroids[c][0])**2) + ((df[df['cluster']==c][1]-centroids
  for i in range(len(a)):
    err = err + sum(a[i])
  return err/len(df)

def fuzzyCMeansClustering(itera):
    membership_mat = initializeMembershipMatrix(len(d),4)
    curr = 0
    error_list = []
    while curr <= itera:
        cluster_centers = calculate_cluster_center(d,membership_mat,4,2)
        membership_mat = update_membership_values(d,membership_mat, cluster_centers,4,2)
        cluster_labels = get_clusters(membership_mat)
        df = pd.DataFrame(d)
        df['cluster']=cluster_labels
        df['color']=df['cluster'].map(lambda x:color_map[x])
        plt.scatter(df[0],df[1],color = df['color'])
        for c in range(0,k):
          plt.plot(*cluster_centers[c],'s',color='black')
        plt.title('Iteration {}'.format(curr))
        plt.show()
        curr += 1
        err=error(df,cluster_centers)
        error_list.append(err)
    return cluster_labels, cluster_centers,error_list
labels, centers,error_list = fuzzyCMeansClustering(10)
it=[]
for i in range(11):
    it.append(i)
plt.plot(it,error_list)
plt.title("error")
plt.show()
```

7. Perform DBSCAN and show the advantages of DBSCAN over model and distance based clustering.

expected: (should visualize the cluster pattern that Model and distance based clustering can not able to capture but can be captured through DBSCAN)

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.datasets import make_circles
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import DBSCAN
from sklearn.manifold import TSNE
X, y = make_circles(n_samples=750, factor=0.3, noise=0.1)
X = StandardScaler().fit_transform(X)
plt.scatter(X[:,0],X[:,1],c=y)
plt.title("our dataset")
plt.show()

import queue

noise = 0
unassigned = 0
core=-1
border=-2
def neighbor_points(data, pointId, radius):
  points = []
  for i in range(len(data)):
    if np.linalg.norm(data[i] - data[pointId]) <= radius:
      points.append(i)
  return points

def dbscan(data, Eps, MinPt):
  pointlabel  = [unassigned] * len(data)
  pointcount = []
  corepoint=[]
  noncore=[]
  for i in range(len(data)):
    pointcount.append(neighbor_points(data,i,Eps))

  for i in range(len(pointcount)):
    if (len(pointcount[i])>=MinPt):
      pointlabel[i]=core
      corepoint.append(i)
    else:
      noncore.append(i)
  for i in noncore:
    for j in pointcount[i]:
      if j in corepoint:
        pointlabel[i]=border
        break
```

```
    clu = 1
    for i in range(len(pointlabel)):
      q = queue.Queue()
      if (pointlabel[i] == core):
        pointlabel[i] = clu
        for x in pointcount[i]:
          if (pointlabel[x]==core):
            q.put(x)
            pointlabel[x]=clu
          elif (pointlabel[x]==border):
            pointlabel[x]=clu
        while not q.empty():
          neighbors = pointcount[q.get()]
          for y in neighbors:
            if (pointlabel[y]==core):
              pointlabel[y]=clu
              q.put(y)
            if (pointlabel[y]==border):
              pointlabel[y]=clu
        clu=clu+1

    return pointlabel,clu
  eps=2.5
  minpts=2
  pointlabel,cl = dbscan(X,eps,minpts)
  #plotRes(X, pointlabel, cl)
  plt.scatter(X[:,0],X[:,1],c=pointlabel)
  plt.show()
```

8. Hierarchical Clustering

Hierarchical clustering is an unsupervised clustering technique which groups together the unlabelled data of similar characteristics.

There are two types of hierarchical clustering:

- Agglomerative Clustering
- Divisive Clustering

Agglomerative Clustering:

In this type of hierarchical clustering all data set are considered as indivisual cluster and at every iterations clusters with similar characteristics are merged to give bigger clusters. This is repeated untill one single cluster is reached. It is also called bottem-top approach.

Divisive Clustering:

It is an opposite of Agglomerative clustering. In this we start from one cluster which contains all data points in one. Iteratively we separate all the cluster of points which aren't similar in characteristics. It is also called top-bottom approach.

Agglomerative Clustering:

Lets start with some domy example :

X=$[x_1, x_2, \ldots, x_5]$, with

$$x_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, x_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, x_3 = \begin{bmatrix} 5 \\ 4 \end{bmatrix}, x_4 = \begin{bmatrix} 6 \\ 5 \end{bmatrix}, x_5 = \begin{bmatrix} 6.5 \\ 6 \end{bmatrix}$$

Steps to perform Agglomerative Clustering:

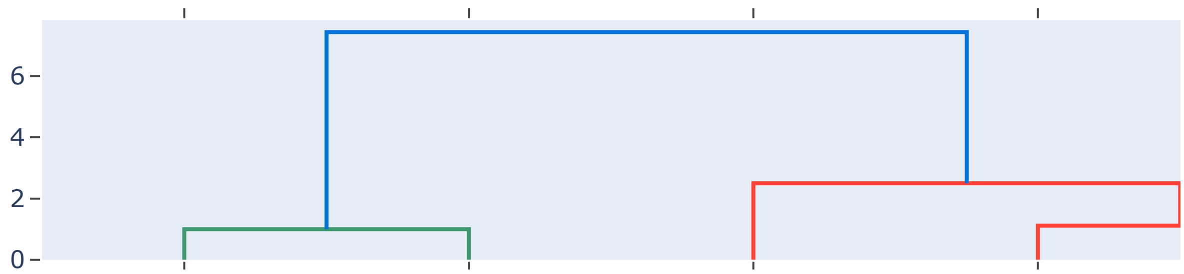
1. Compute Distance matrix ($N \times N$ matrix, where N number of vectors present in the dataset): $D(a, b) = ||x_a - x_b||_2$
2. Replace the diagonal elements with inf and find the index of the minimum element present in the distance matrix (suppose we get the location (l, k)).
3. Replace $x_{min(l,k)} = .5 \times [x_l + x_m]$ and delete $x_{max(l,m)}$ vector from X(i.e now $(N = N - 1)$),

repeat from step 1 again untill all the vectors combined to a single cluster.

```
import numpy as np
import math
def Euclidian_Dist(x,y):
  return np.linalg.norm(x-y)
```

```
def Dist_mat(X):
 #write your code here
 dist_mat=np.zeros([len(X),len(X)])
 for i in range(len(X)):
   temp = np.square(X-np.tile(X[i],(len(X),1)))
   dist_mat[i] = np.sqrt(temp[:,0]+temp[:,1])
 diag = (math.inf)*np.ones([len(X),])
 np.fill_diagonal(dist_mat, diag)
 return dist_mat

def Combine(X,arr):
  X_new = np.zeros([len(X)-1,2])
  tem = (X[arr[0]]+X[arr[1]])/2
  com=False
  t=0
  for i in range(len(X)):
    if (i!=arr[0] and i!= arr[1]):
      X_new[t]=X[i]
      t=t+1
    elif (i==arr[0] or i==arr[1] and com==False):
      X_new[t] = tem
      t=t+1
      com = True
  dis_matx = Dist_mat(X_new)
  return X_new,dis_matx


X=np.array([[1,1],[2,1],[5,4],[6,5],[6.5,6]])
#X=X.transpose()
import plotly.figure_factory as ff
X_t = X.transpose()
#write your code here
n_clu=len(X)
while n_clu>1:
  distance_matrix = Dist_mat(X)
  combine = np.where(distance_matrix == np.min(distance_matrix))
  min_co = np.array(list(zip(combine[0], combine[1])))
  print("Vector of X to be combined :")
  print(min_co[0])
  X,dis_m=Combine(X,min_co[0])
  print("Mean of clusters after every iteration:")
  print(X)
  print(dis_m)
  n_clu = len(X)
## velidate from inbuilt Dendogram
lab=np.linspace(1,X_t.shape[1],X_t.shape[1])
fig = ff.create_dendrogram(X_t.T, labels=lab)
fig.update_layout(width=800, height=300)
fig.show()
```

```
Vector of X to be combined :
[0 1]
Mean of clusters after every iteration:
[[1.5 1. ]
 [5.  4. ]
 [6.  5. ]
 [6.5 6. ]]
[[       inf 4.60977223 6.02079729 7.07106781]
 [4.60977223        inf 1.41421356 2.5       ]
 [6.02079729 1.41421356        inf 1.11803399]
 [7.07106781 2.5        1.11803399        inf]]
Vector of X to be combined :
[2 3]
Mean of clusters after every iteration:
[[1.5  1.  ]
 [5.   4.  ]
 [6.25 5.5 ]]
[[       inf 4.60977223 6.54312616]
 [4.60977223        inf 1.95256242]
 [6.54312616 1.95256242        inf]]
Vector of X to be combined :
[1 2]
Mean of clusters after every iteration:
[[1.5   1.   ]
 [5.625 4.75 ]]
[[       inf 5.57477578]
 [5.57477578        inf]]
Vector of X to be combined :
[0 1]
Mean of clusters after every iteration:
[[3.5625 2.875 ]]
[[inf]]
```



Divisive clustering:

It is a top down approach of hierarchial clustering

Lets start with some domy example :

X=$[x_1, x_2, \ldots, x_5]$, with

$$x_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, x_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, x_3 = \begin{bmatrix} 5 \\ 4 \end{bmatrix}, x_4 = \begin{bmatrix} 6 \\ 5 \end{bmatrix}, x_5 = \begin{bmatrix} 6.5 \\ 6 \end{bmatrix}$$

1. Find the biggest cluster (having highest diameter), initially the single cluster is the biggest cluster.

$$Diameter_{cluster} = \max_{i,j} ||x_i - x_j||_2$$

i, j will move over all the elements in the cluster.

2. find the splinter element of the cluster by using the maximum average distance between the other elements.

$$d_k = \frac{1}{N-1} \sum_{i=1}^{N} ||x_k - x_i||_2$$

$$splinter - group - element = arg\max_{1 \leq k \leq N}(d_k)$$

repeat the same and assign element to the splinter group untill the differance between average incluster distance and average splinter group distance of each element turns negative.

$$d_{avgsplint_k} = \frac{1}{M-1} \sum_{i=1}^{M} ||x_k - x_i||_2$$

Stop:

$$d_k - d_{avgsplint_k} < 0$$

and assign the splinter group as a new cluster.

3. Repeat the step 1 and 2 untill each cluster have only one element.
4. **Plot the cluster split with respect to their diameter**

```
import numpy as np

def Dist_mat(X):
  dist_mat=np.zeros([len(X[0]),len(X[0])])
  for i in range(len(X[0])):
    temp =np.square(X-(np.array([X[:,i],]*(len(X[0]))).transpose()))
    dist_mat[i] = np.sqrt(temp[0]+temp[1])
  return dist_mat

def avg_distance(X):
  te = Dist_mat(X)
  print(te)
  su = np.zeros([len(te),1])
  for i in range(len(te)):
      su[i]=np.sum(te[i])
  dis_avg = su/(len(X[0])-1)
  return dis_avg


def get_diameter(X, i):
    """Returns the diameter of the ith cluster in X"""
    #write your code here
    return diameter

def get_biggest_cluster(X):
    """ Returns the cluster index having largest diameter"""
```

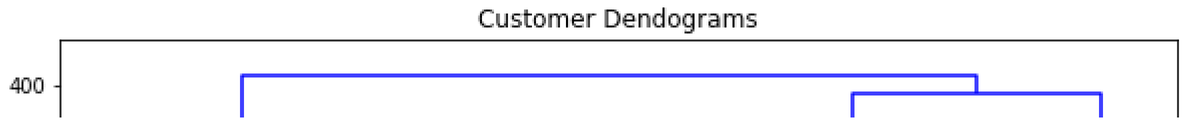
```
    #write your code here
    # index having max diameter
    return max_cluster_ind

def avg_spl_dists(cluster, splinter):
    """ Return the average of distances of each point belonging to cl wrt splinter"""
    #write your code here
    return avg_dists


# Implement Divisive Clustering
import numpy as np
X = np.array([[1,1], [2,1], [5,4], [6,5], [6.5,6]])
X = X.transpose() # Shape after transpose: [2, 5]
num_points = X.shape[1]
print(f'X:\n {X}')
```

```
X:
 [[1.  2.  5.  6.  6.5]
 [1.  1.  4.  5.  6. ]]
Initial Number of clusters: 1
---------------- Iteraion - 1 -------------------
Biggest cluster ind is: 0
Biggest Cluster is:
 [[1.  2.  5.  6.  6.5]
 [1.  1.  4.  5.  6. ]]
0
Cluster:
 [[2.  5.  6.  6.5]
 [1.  4.  5.  6. ]]
 Shape: (2, 4)
Splinter:
 [[1.]
 [1.]]
 Shape: (2, 1)
New member added to splinter of index 0 and new member is
 [[2.]
 [1.]]
New cluster shape is (2, 3)
 [[5.  6.  6.5]
 [4.  5.  6. ]]
New splinter shape is (2, 2)
 [[1. 2.]
 [1. 1.]]
Final splinter and cluster shapes: (2, 2), (2, 3)
New num of clusters after splitting is: 2
[[5.  6.  6.5]
 [4.  5.  6. ]],
[[1. 2.]
 [1. 1.]],
---------------- Iteraion - 2 -------------------
Biggest cluster ind is: 0
Biggest Cluster is:
 [[5.  6.  6.5]
 [4.  5.  6. ]]
0
Cluster:
 [[6.  6.5]
 [5.  6. ]]
```

```
 Shape: (2, 2)
Splinter:
 [[5.]
 [4.]]
 Shape: (2, 1)
Final splinter and cluster shapes: (2, 1), (2, 2)
New num of clusters after splitting is: 3
[[1. 2.]
 [1. 1.]],
[[6.  6.5]
 [5.  6. ]],
[[5.]
 [4.]],
---------------- Iteraion - 3 -------------------
Biggest cluster ind is: 1
Biggest Cluster is:
 [[6.  6.5]
```

9.Take a real data example and demonstrate both Agglomerative and Divisive clustering.

```
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
import numpy as np
from sklearn.cluster import AgglomerativeClustering
customer_data = pd.read_csv('shopping-data.csv')
data = customer_data.iloc[:, 3:5].values
import scipy.cluster.hierarchy as shc
plt.figure(figsize=(10, 7))
plt.title("Customer Dendograms")
dend = shc.dendrogram(shc.linkage(data, method='ward'))
```

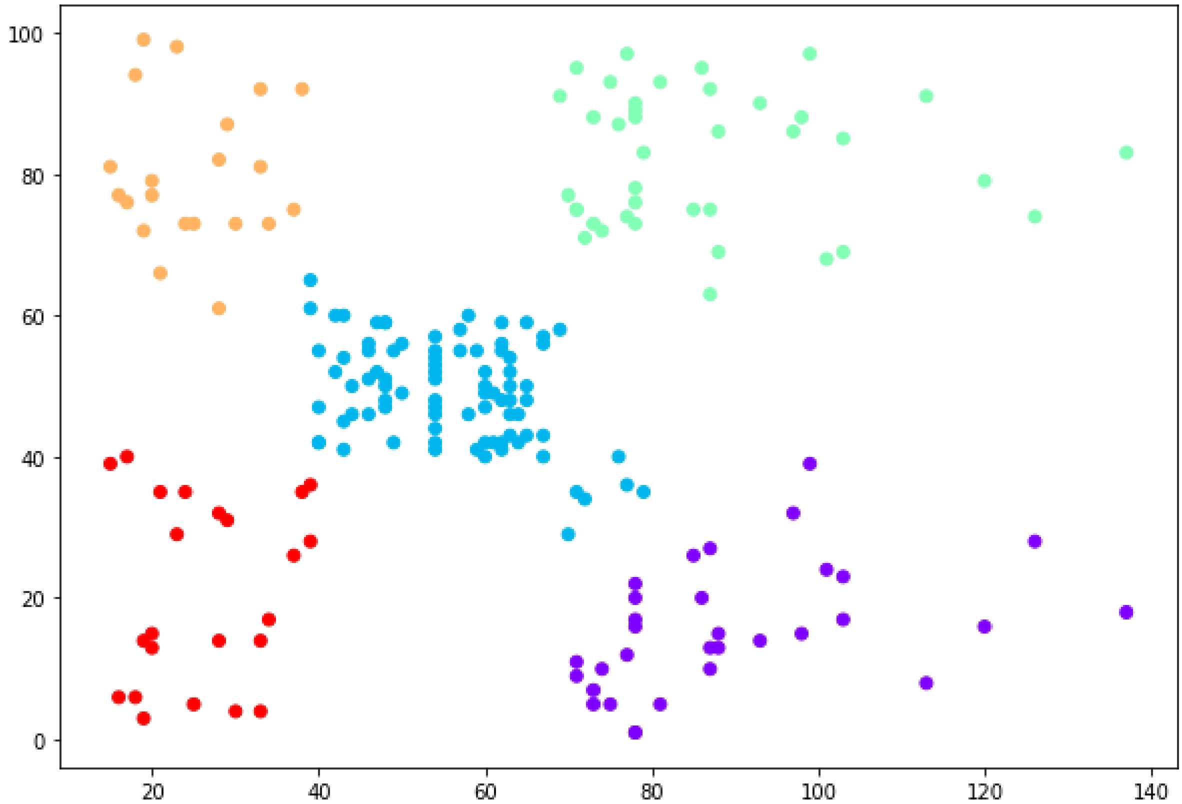



```
cluster = AgglomerativeClustering(n_clusters=5, affinity='euclidean', linkage='ward')
cluster.fit_predict(data)
```

```
array([4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3,
       4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 3, 4, 1,
       4, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 2, 0, 2, 0, 2,
       1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 1, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2, 0, 2,
       0, 2])
```


```
plt.figure(figsize=(10, 7))
plt.scatter(data[:,0], data[:,1], c=cluster.labels_, cmap='rainbow')
```

```
<matplotlib.collections.PathCollection at 0x7f8ffa87d390>
```



https://drive.google.com/file/d/1xe8HZyz8XcC8-X5XYKk_s2_yF0RHrKq_/view?usp=sharing