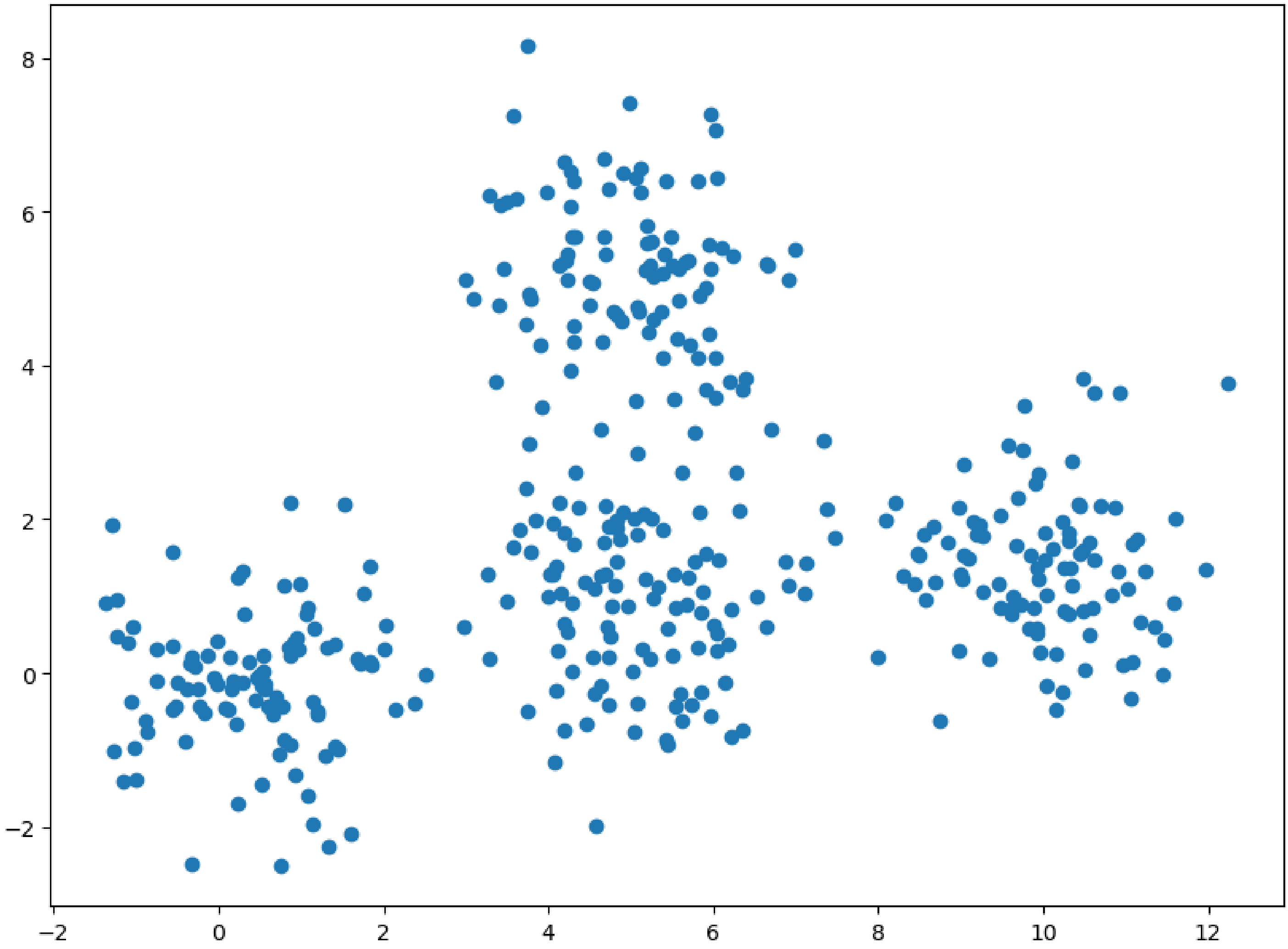
In [39]:

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

## Data generation
# write your code here
a=np.random.multivariate_normal([0.5,0], [[1, 0],[0, 1]], 100)
b=np.random.multivariate_normal([5,5], [[1, 0],[0, 1]], 100)
c=np.random.multivariate_normal([5,1], [[1, 0],[0, 1]], 100)
d=np.random.multivariate_normal([10,1.5], [[1, 0],[0, 1]], 100)
X=np.concatenate((a,b,c,d), axis=0)
plt.scatter(X[:,0],X[:,1])

n=X.shape[1]
n_iter=10
```




In [40]:

```
K=4
import random

# creating an empty centroid array
centroids=np.array([]).reshape(n,0)

# creating 5 random centroids
for k in range(K):
    centroids=np.c_[centroids,X[random.randint(0,m-1)]]

print(centroids)
```

```
[[ 6.03389968  1.07922277  6.52521602 -0.5505451 ]
 [ 0.52556462  0.84542642  1.00397515  1.56554586]]
```

In [41]:

```
output={}

# creating an empty array
euclid=np.array([]).reshape(m,0)

# finding distance between for each centroid
for k in range(K):
    dist=np.sum((X-centroids[:,k])**2,axis=1)
    euclid=np.c_[euclid,dist]

# storing the minimum value we have computed
minimum=np.argmin(euclid,axis=1)+1
```

In [42]:

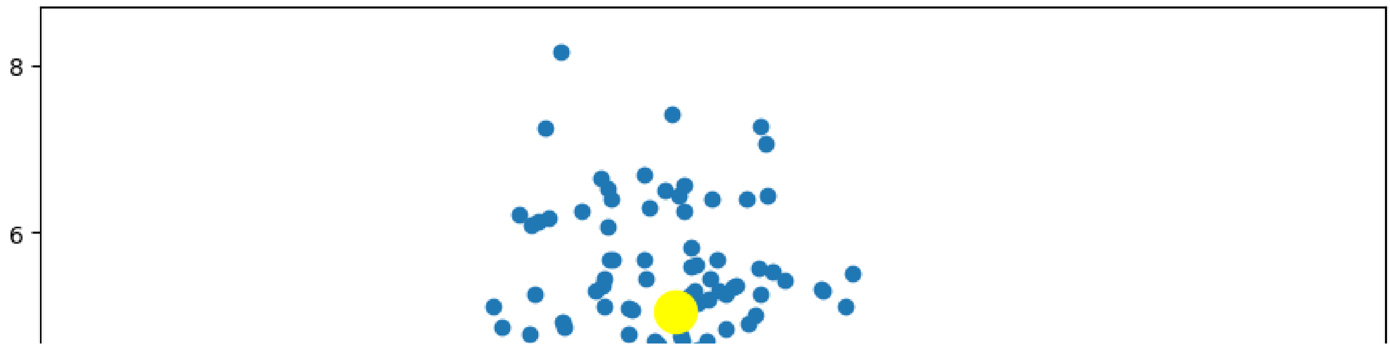
```
# computing the mean of separated clusters
cent={}
for k in range(K):
    cent[k+1]=np.array([]).reshape(2,0)

# assigning of clusters to points
for k in range(m):
    cent[minimum[k]]=np.c_[cent[minimum[k]],X[k]]
for k in range(K):
    cent[k+1]=cent[k+1].T

# computing mean and updating it
for k in range(K):
     centroids[:,k]=np.mean(cent[k+1],axis=0)
```

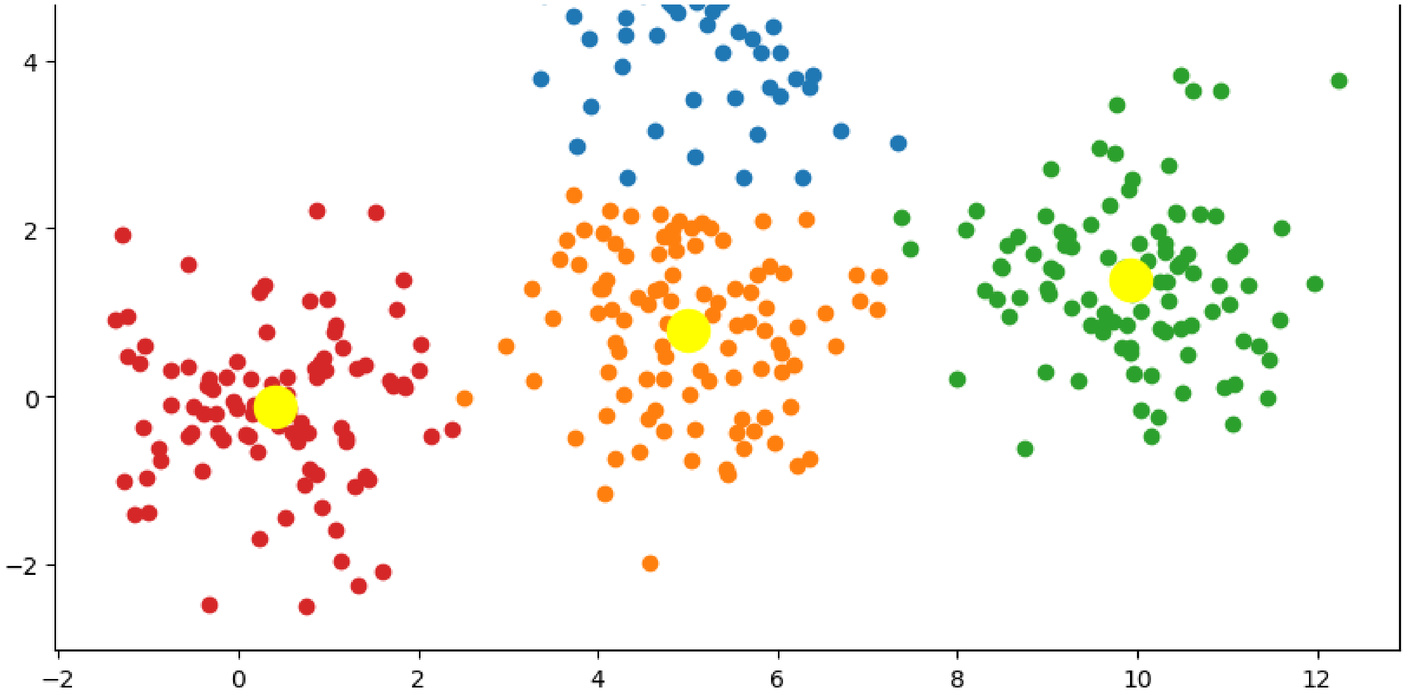
In [43]:

```
# repeating the above steps again and again
for i in range(n_iter):
    euclid=np.array([]).reshape(m,0)
    for k in range(K):
        dist=np.sum((X-centroids[:,k])**2,axis=1)
        euclid=np.c_[euclid,dist]
    C=np.argmin(euclid,axis=1)+1
    cent={}
    for k in range(K):
        cent[k+1]=np.array([]).reshape(2,0)
    for k in range(m):
        cent[C[k]]=np.c_[cent[C[k]],X[k]]
    for k in range(K):
        cent[k+1]=cent[k+1].T
    for k in range(K):
        centroids[:,k]=np.mean(cent[k+1],axis=0)
    final=cent
```

In [44]:

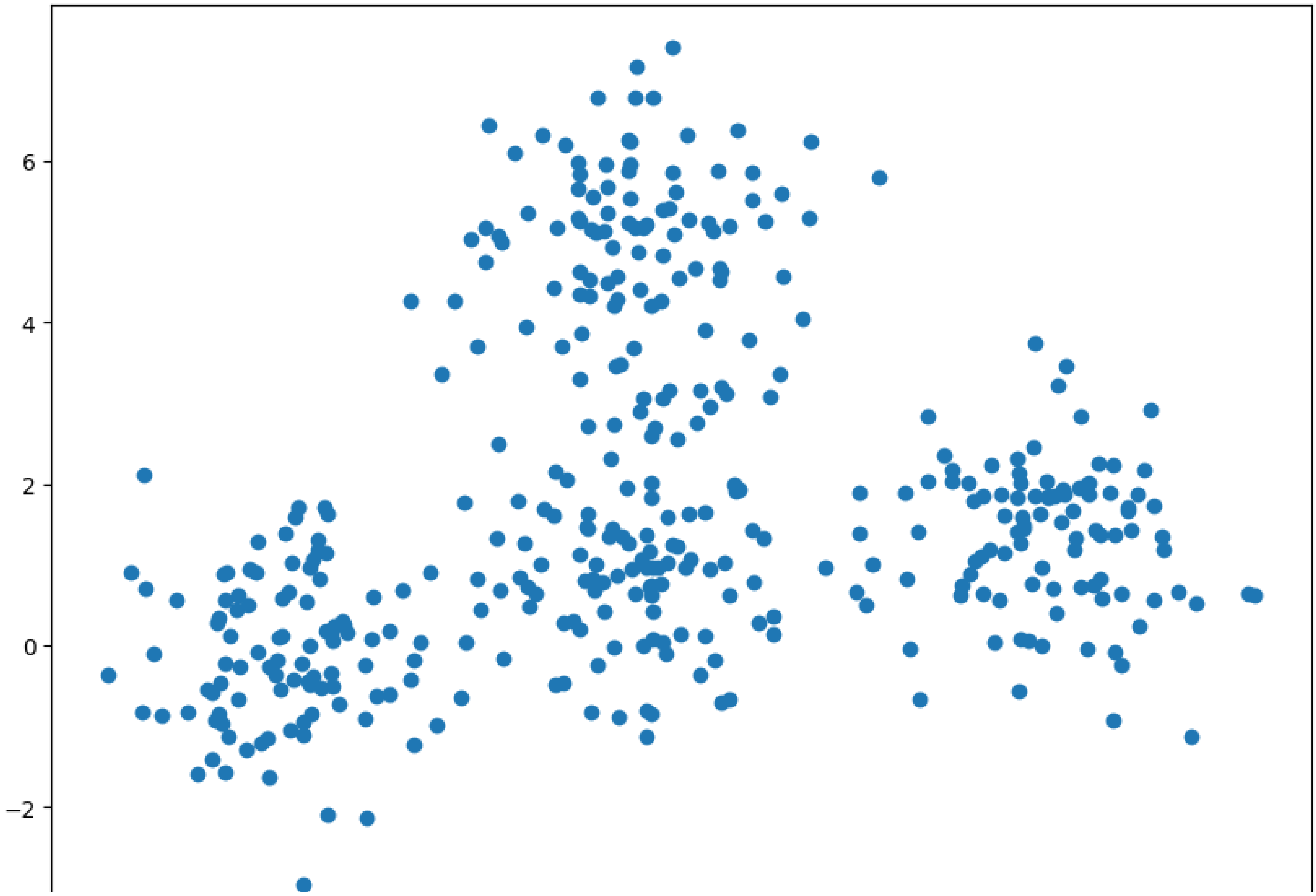
```
for k in range(K):
    plt.scatter(final[k+1][:,0],final[k+1][:,1])
plt.scatter(centroids[0,:],centroids[1,:],s=300,c='yellow')
plt.rcParams.update({'figure.figsize':(10,7.5), 'figure.dpi':100})
plt.show()
```

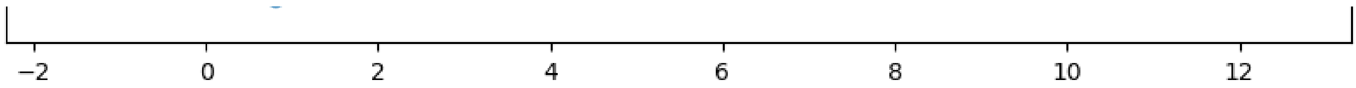





In [45]:

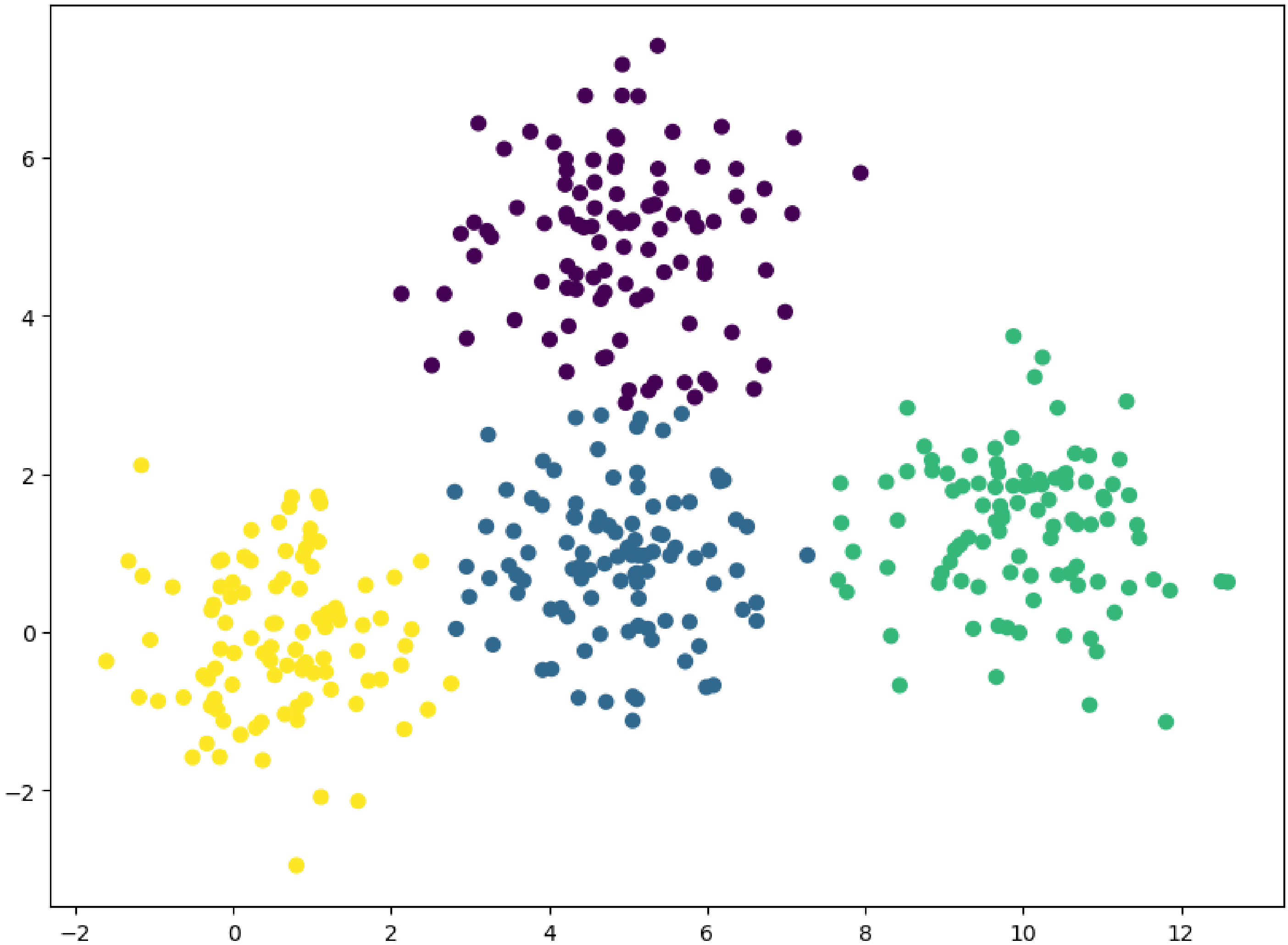
```
import numpy as np
import matplotlib.pyplot as plt

## Data generation
# write your code here
a=np.random.multivariate_normal([0.5,0], [[1, 0],[0, 1]], 100)
b=np.random.multivariate_normal([5,5], [[1, 0],[0, 1]], 100)
c=np.random.multivariate_normal([5,1], [[1, 0],[0, 1]], 100)
d=np.random.multivariate_normal([10,1.5], [[1, 0],[0, 1]], 100)
X=np.concatenate((a,b,c,d), axis=0)
plt.scatter(X[:,0],X[:,1])

n=X.shape[1]
n_iter=10
```





In [47]:

```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=4
                     ).fit(X)
labels = gmm.predict(X)
plt.scatter(X[:, 0], X[:, 1], c=labels, s=40, cmap='viridis');
probs = gmm.predict_proba(X)
```



In [48]:

```
from sklearn.cluster import KMeans
def calculate_mean_covariance(X, prediction):
    C = 3
    d = X.shape[1]
    labels = np.unique(prediction)
    initial_means = np.zeros((C, d))
    initial_cov = np.zeros((C, d, d))
    initial_pi = np.zeros(C)

    counter=0
    for label in sorted(labels):
        ids = np.where(prediction == label) # returns indices
        initial_pi[counter] = len(ids[0]) / X.shape[0]
        initial_means[counter,:] = np.mean(X[ids], axis = 0)
        de_meaned = X[ids] - initial_means[counter,:]
        Nk = X[ids].shape[0]
        initial_cov[counter,:, :] = np.dot(initial_pi[counter] * de_meaned.T, de_meaned)
/ Nk
        counter+=1
    assert np.sum(initial_pi) == 1
    return (initial_means, initial_cov, initial_pi)
```

```
n_clusters = 3
kmeans = KMeans(n_clusters= n_clusters, max_iter=500, algorithm = 'auto')
fitted = kmeans.fit(X)
prediction = kmeans.predict(X)

m, c, pi = calculate_mean_covariance(X, prediction)
```

In [49]:

```
from scipy.stats import multivariate_normal as mvn
class GMM:
    """ Gaussian Mixture Model

    Parameters
    -----------
        k: int , number of gaussian distributions

        seed: int, will be randomly set if None

        max_iter: int, number of iterations to run algorithm, default: 200

    Attributes
    -----------
       centroids: array, k, number_features

       cluster_labels: label for each data point

    """
    def __init__(self, C, n_runs):
        self.C = C # number of Guassians/clusters
        self.n_runs = n_runs


    def get_params(self):
        return (self.mu, self.pi, self.sigma)



    def calculate_mean_covariance(self, X, prediction):
        """Calculate means and covariance of different
            clusters from k-means prediction

        Parameters:
        ------------
        prediction: cluster labels from k-means

        X: N*d numpy array data points

        Returns:
        -------------
        intial_means: for E-step of EM algorithm

        intial_cov: for E-step of EM algorithm

        """
        d = X.shape[1]
        labels = np.unique(prediction)
        self.initial_means = np.zeros((self.C, d))
        self.initial_cov = np.zeros((self.C, d, d))
        self.initial_pi = np.zeros(self.C)

        counter=0
        for label in labels:
            ids = np.where(prediction == label) # returns indices
            self.initial_pi[counter] = len(ids[0]) / X.shape[0]
            self.initial_means[counter,:] = np.mean(X[ids], axis = 0)
            de_meaned = X[ids] - self.initial_means[counter,:]
            Nk = X[ids].shape[0] # number of data points in current gaussian
            self.initial_cov[counter,:, :] = np.dot(self.initial_pi[counter] * de_meaned
.T, de_meaned) / Nk
```

```
            counter+=1
        assert np.sum(self.initial_pi) == 1

        return (self.initial_means, self.initial_cov, self.initial_pi)



    def _initialise_parameters(self, X):
        """Implement k-means to find starting
            parameter values.
            https://datascience.stackexchange.com/questions/11487/how-do-i-obtain-the-wei
ght-and-variance-of-a-k-means-cluster

        Parameters:
        ------------
        X: numpy array of data points

        Returns:
        ----------
        tuple containing initial means and covariance

        _initial_means: numpy array: (C*d)

        _initial_cov: numpy array: (C,d*d)


        """
        n_clusters = self.C
        kmeans = KMeans(n_clusters= n_clusters, init="k-means++", max_iter=500, algorith
m = 'auto')
        fitted = kmeans.fit(X)
        prediction = kmeans.predict(X)
        self._initial_means, self._initial_cov, self._initial_pi = self.calculate_mean_c
ovariance(X, prediction)


        return (self._initial_means, self._initial_cov, self._initial_pi)



    def _e_step(self, X, pi, mu, sigma):
        """Performs E-step on GMM model

        Parameters:
        ------------
        X: (N x d), data points, m: no of features
        pi: (C), weights of mixture components
        mu: (C x d), mixture component means
        sigma: (C x d x d), mixture component covariance matrices

        Returns:
        ----------
        gamma: (N x C), probabilities of clusters for objects
        """
        N = X.shape[0]
        self.gamma = np.zeros((N, self.C))

        const_c = np.zeros(self.C)


        self.mu = self.mu if self._initial_means is None else self._initial_means
        self.pi = self.pi if self._initial_pi is None else self._initial_pi
        self.sigma = self.sigma if self._initial_cov is None else self._initial_cov

        for c in range(self.C):
            # Posterior Distribution using Bayes Rule
            self.gamma[:,c] = self.pi[c] * mvn.pdf(X, self.mu[c,:], self.sigma[c])

        # normalize across columns to make a valid probability
        gamma_norm = np.sum(self.gamma, axis=1)[:,np.newaxis]
        self.gamma /= gamma_norm
```

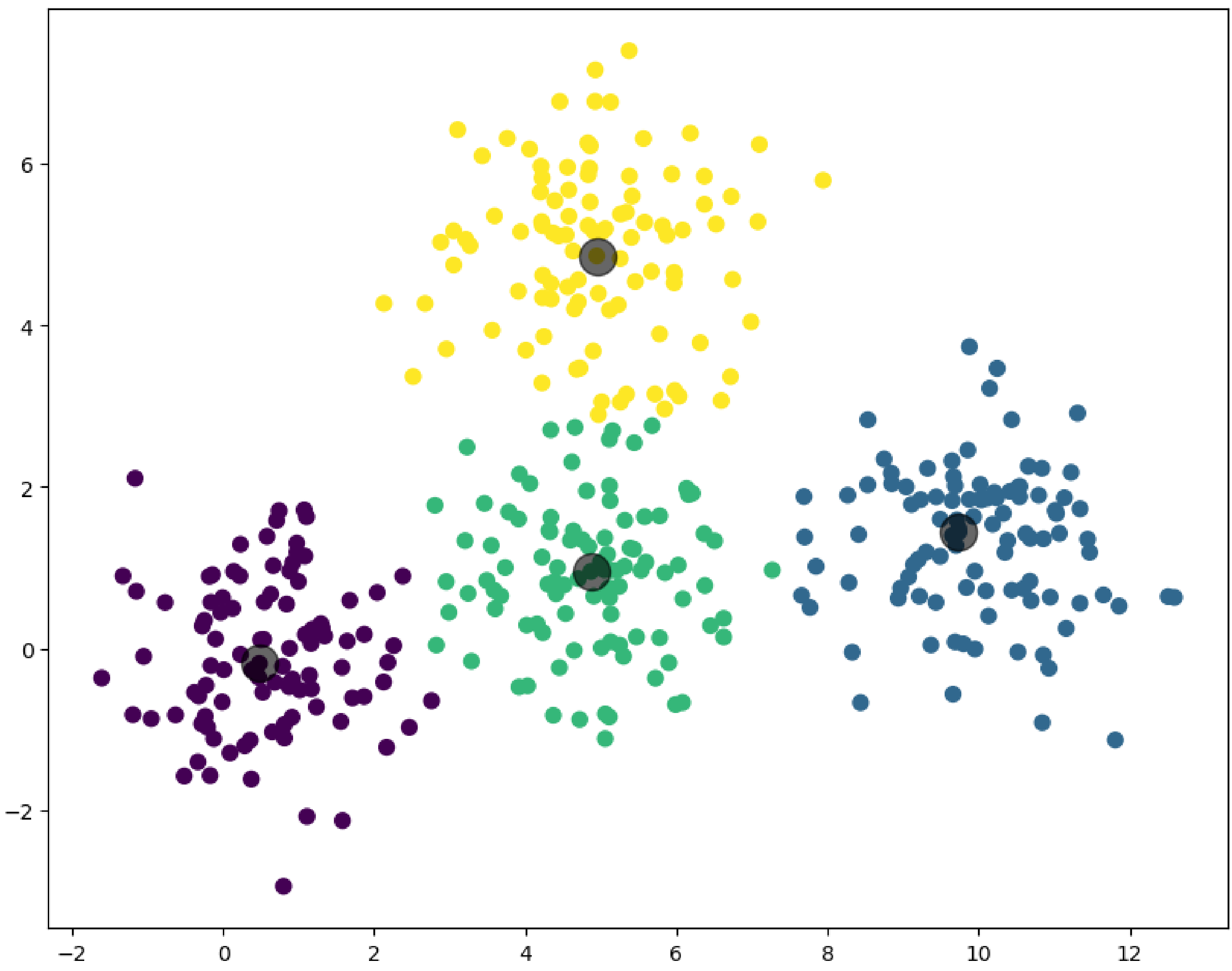
```
        return self.gamma


    def _m_step(self, X, gamma):
        """Performs M-step of the GMM
        We need to update our priors, our means
        and our covariance matrix.

        Parameters:
        -----------
        X: (N x d), data
        gamma: (N x C), posterior distribution of lower bound

        Returns:
        ---------
        pi: (C)
        mu: (C x d)
        sigma: (C x d x d)
        """
        N = X.shape[0] # number of objects
        C = self.gamma.shape[1] # number of clusters
        d = X.shape[1] # dimension of each object

        # responsibilities for each gaussian
        self.pi = np.mean(self.gamma, axis = 0)

        self.mu = np.dot(self.gamma.T, X) / np.sum(self.gamma, axis = 0)[:,np.newaxis]

        for c in range(C):
            x = X - self.mu[c, :] # (N x d)

            gamma_diag = np.diag(self.gamma[:,c])
            x_mu = np.matrix(x)
            gamma_diag = np.matrix(gamma_diag)

            sigma_c = x.T * gamma_diag * x
            self.sigma[c,:,:]=(sigma_c) / np.sum(self.gamma, axis = 0)[:,np.newaxis][c]

        return self.pi, self.mu, self.sigma


    def _compute_loss_function(self, X, pi, mu, sigma):
        """Computes lower bound loss function

        Parameters:
        -----------
        X: (N x d), data

        Returns:
        ---------
        pi: (C)
        mu: (C x d)
        sigma: (C x d x d)
        """
        N = X.shape[0]
        C = self.gamma.shape[1]
        self.loss = np.zeros((N, C))

        for c in range(C):
            dist = mvn(self.mu[c], self.sigma[c],allow_singular=True)
            self.loss[:,c] = self.gamma[:,c] * (np.log(self.pi[c]+0.00001)+dist.logpdf(X
)-np.log(self.gamma[:,c]+0.000001))
        self.loss = np.sum(self.loss)
        return self.loss



    def fit(self, X):
        """Compute the E-step and M-step and
            Calculates the lowerbound

        Parameters:
```

```
        ------------
        X: (N x d), data

        Returns:
        ----------
        instance of GMM

        """

        d = X.shape[1]
        self.mu, self.sigma, self.pi =  self._initialise_parameters(X)

        try:
            for run in range(self.n_runs):
                self.gamma  = self._e_step(X, self.mu, self.pi, self.sigma)
                self.pi, self.mu, self.sigma = self._m_step(X, self.gamma)
                loss = self._compute_loss_function(X, self.pi, self.mu, self.sigma)

                if run % 10 == 0:
                    print("Iteration: %d Loss: %0.6f" %(run, loss))


        except Exception as e:
            print(e)


        return self




    def predict(self, X):
        """Returns predicted labels using Bayes Rule to
        Calculate the posterior distribution

        Parameters:
        -------------
        X: ?*d numpy array

        Returns:
        ----------
        labels: predicted cluster based on
        highest responsibility gamma.

        """
        labels = np.zeros((X.shape[0], self.C))

        for c in range(self.C):
            labels [:,c] = self.pi[c] * mvn.pdf(X, self.mu[c,:], self.sigma[c])
        labels  = labels .argmax(1)
        return labels

    def predict_proba(self, X):
        """Returns predicted labels

        Parameters:
        -------------
        X: N*d numpy array

        Returns:
        ----------
        labels: predicted cluster based on
        highest responsibility gamma.

        """
        post_proba = np.zeros((X.shape[0], self.C))

        for c in range(self.C):
            # Posterior Distribution using Bayes Rule, try and vectorise
            post_proba[:,c] = self.pi[c] * mvn.pdf(X, self.mu[c,:], self.sigma[c])

        return post_proba
```

In [50]:

```
model = GMM(4, n_runs = 100)

fitted_values = model.fit(X)

predicted_values = model.predict(X)
# compute centers as point of highest density of distribution
centers = np.zeros((4,2))
for i in range(model.C):
    density = mvn(cov=model.sigma[i], mean=model.mu[i]).logpdf(X)
    centers[i, :] = X[np.argmax(density)]

plt.figure(figsize = (10,8))
plt.scatter(X[:, 0], X[:, 1],c=predicted_values ,s=50, cmap='viridis')

plt.scatter(centers[:, 0], centers[:, 1],c='black', s=300, alpha=0.6);
```

```
Iteration: 0 Loss: -1660.501398
Iteration: 10 Loss: -1653.847525
Iteration: 20 Loss: -1653.847330
Iteration: 30 Loss: -1653.847329
Iteration: 40 Loss: -1653.847329
Iteration: 50 Loss: -1653.847329
Iteration: 60 Loss: -1653.847329
Iteration: 70 Loss: -1653.847329
Iteration: 80 Loss: -1653.847329
Iteration: 90 Loss: -1653.847329
```




In [51]:

```
class FCM:
```

```python
    def _init_(self, n_clusters=10, max_iter=150, m=2, error=1e-5, random_state=42):
        assert m > 1
        self.u, self.centers = None, None
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.m = m
        self.error = error
        self.random_state = random_state

    def fit(self, X):
        self.n_samples = X.shape[0]
        r = np.random.RandomState(self.random_state)
        u = r.rand(self.n_samples, self.n_clusters)
        u = u / np.tile(u.sum(axis=1)[np.newaxis].T, self.n_clusters)

        r = np.random.RandomState(self.random_state)
        self.u = r.rand(self.n_samples,self.n_clusters)
        self.u = self.u / np.tile(self.u.sum(axis=1)[np.newaxis].T, self.n_clusters)

        for iteration in range(self.max_iter):
            u_old = self.u.copy()

            self.centers = self.next_centers(X)
            self.u = self._predict(X)

            selfClusterOut = self.predict(X)
            centers = self.centers

            for i in range(nPoints):
                plt.scatter(x[i], y[i], c = plotColor[selfClusterOut[i]], s = 10)
            for i in range(self.n_clusters):
                plt.scatter(centers[i][0], centers[i][1], c = 'black' ,marker = 'X')
            plt.show()

            # Stopping rule
            if norm(self.u - u_old) < self.error:
                break

    def next_centers(self, X):
        um = self.u ** self.m
        return (X.T @ um / np.sum(um, axis=0)).T

    def _predict(self, X):
        power = float(2 / (self.m - 1))
        temp = cdist(X, self.centers) ** power
        denominator_ = temp.reshape((X.shape[0], 1, -1)).repeat(temp.shape[-1], axis=1)
        denominator_ = temp[:, :, np.newaxis] / denominator_

        return 1 / denominator_.sum(2)

    def predict(self, X):
        if len(X.shape) == 1:
            X = np.expand_dims(X, axis=0)

        u = self._predict(X)
        return np.argmax(u, axis=-1)
```

In [53]:

```python
from google.colab import drive
drive.mount('/content/drive')
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-53-d5df0069828e> in <module>
----> 1 from google.colab import drive
      2 drive.mount('/content/drive')

ModuleNotFoundError: No module named 'google'
```

In [54]:

```python
X, y = make_circles(n_samples=750, factor=0.3, noise=0.1)
X = StandardScaler().fit_transform(X)
y_pred = DBSCAN(eps=0.3, min_samples=10).fit_predict(X)
y_kmeans = KMeans(2).fit_predict(X)

plt.scatter(X[:,0], X[:,1], c=y_pred)
plt.show()
plt.scatter(X[:,0], X[:,1], c=y_kmeans)
plt.show()
print('Number of clusters: {}'.format(len(set(y_pred[np.where(y_pred != -1)]))))
print('Homogeneity: {}'.format(metrics.homogeneity_score(y, y_pred)))
print('Completeness: {}'.format(metrics.completeness_score(y, y_pred)))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-54-c6e26e7f8c07> in <module>
----> 1 X, y = make_circles(n_samples=750, factor=0.3, noise=0.1)
      2 X = StandardScaler().fit_transform(X)
      3 y_pred = DBSCAN(eps=0.3, min_samples=10).fit_predict(X)
      4 y_kmeans = KMeans(2).fit_predict(X)
      5

NameError: name 'make_circles' is not defined
```

In [ ]:

```python

```