Data Clustering with Machine Learning Using Sklearn Library

In chapters 6 and 7, you studied how to solve regression and classification problems, respectively, using machine learning algorithms in Sklearn. Regression and Classification are types of supervised machine learning problems. In this chapter, you are going to study data clustering algorithms.

Clustering algorithms are unsupervised algorithms where the training data is not labeled. Rather, the algorithms cluster or group the data sets based on common characteristics. In this chapter, you will study two of the most common types of clustering algorithms, i.e., KMeans Clustering and Hierarchical Clustering. You will see how Python's Sklearn library can be used to implement the two clustering algorithms. So, let's begin without much ado.

8.1. K Means Clustering

K Means clustering is one of the most commonly used algorithms for clustering unlabeled data. In K Means clustering, K refers to the number of clusters that you want your data to be grouped into. In K Means clustering, the number of clusters has to be defined before K clustering can be applied to the data points.

Steps for K Means Clustering

The following are the steps that are needed to be performed in order to perform K Means clustering of data points.

1. Randomly assign centroid values for each cluster.

2. Calculate the distance (Euclidean or Manhattan) between each data point an centroid values of all the clusters.

3. Assign the data point to the cluster of the centroid with the shorted distance.

4. Calculate and update centroid values based on the mean values of the coordi of all the data points of the corresponding cluster.

5. Repeat steps 2–4 until new centroid values for all the clusters are different fi previous centroid values.

Why use K Means Clustering?

K Means clustering is particularly useful when:

1. K Means clustering is a simple to implement algorithm

2. Can be applied to large datasets

3. Scales well to unseen data points

4. Generalize well to clusters of various sizes and shapes.

Disadvantages of K Means Clustering Algorithm

The following are some of the disadvantages of K Means clustering algorithm:

1. The value of K has to be chosen manually

2. Convergence or training time depends on the initial value of K

3. Clustering performance is affected greatly by outliers.

Enough of theory. Let's see how to perform K Means clustering with Scikit learn.

8.1.1. Clustering Dummy Data with Sklearn

Importing the libraries needed is the first step, as shown in the following script:

Script 1:

```
1. import numpy as np
2. import pandas as pd
3. from sklearn.datasets.samples_generator import make_blobs
4. from sklearn.cluster import KMeans
5. from matplotlib import pyplot as plt
6. %matplotlib inline
```

Next, we create a dummy dataset containing 500 records and 4 cluster centers. The average standard deviation between the records is 2.0.

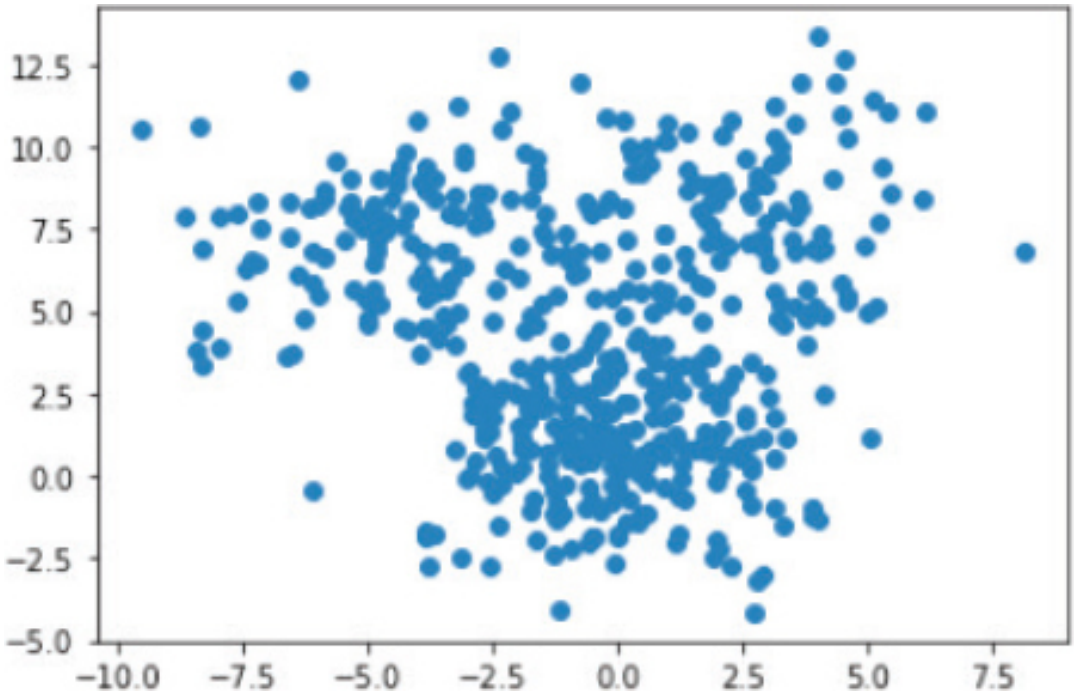
The following script creates a dummy dataset and plots data points on a plot.

Script 2:

```
1. # generating dummy data of 500 records with 4 clusters
2. features, labels = make_blobs(n_samples=500, centers=4, cluster_std = 2.00)
3.
4. #plotting the dummy data
5. plt.scatter(features[:,0], features[:,1] )
```

The output looks like this. Using K Means clustering, you will see how we will create four clusters in this dataset.

Output:



Note:

It is important to mention that dummy data is generated randomly, and hence, you can have a slightly different plot than the plot in the above figure.

To implement K Means clustering, you can use the KMeans class from the sklearn.cluster module. You have to pass the number of clusters as an attribute to the KMeans class constructor. To train the KMeans model, simply pass the dataset to the fit() method of the K Means class, as shown below.

Script 3:

```
1. # performing kmeans clustering using KMeans class
2. km_model = KMeans(n_clusters=4)
3. km_model.fit(features)
```

Once the model is trained, you can print the cluster centers using the cluster_centers_attribute of the KMeans class object.

Script 4:

```
1. #printing centroid values
2. print (km_model.cluster_centers_)
```

The four cluster centers as predicted by our K Means model has the following coordiantes:

Output:

```
[[-4.54070231 7.26625699]
[ 0.10118215 -0.23788283]
[ 2.57107155 8.17934929]
[-0.38501161 3.11446039]]
```

In addition to finding cluster centers, the KMeans class also assigns a cluster label to each data point. The cluster labels are numbers that basically serve as cluster id. For instance, in the case of four clusters, the cluster ids are 0,1,2,3.

To print the cluster ids for all the labels, you can use the labels_attribute of the KMeans class, as shown below.

Script 5:

```
1. #printing predicted label values
2. print (km_model.labels_)
```

Output:

```
[0 2 3 2 1 1 3 1 2 0 0 2 3 3 1 1 2 0 1 2 2 1 3 3 1 1 0 2 0 2 0 1 0 1 3 2 2 3 0 0 0 2 1 2 0 1 3 1 3 2 1 3 3 1 0 2 1
 3 0 0 3 3 3 1 1 1 3 0 1 3 2 1 1 2 0 2 1 2 1 0 0 2 1 2 1 0 2 0 0 2 2 3 3 0 2 0 2 3 0 0 3 1 0 3 2 1 3 2 2 0 2 1
 1 0 0 3 3 2 3 1 0 0 3 0 1 0 3 1 0 3 2 0 1 1 0 2 1 2 2 0 3 1 3 3 0 1 1 0 2 0 0 0 3 3 3 3 0 3 1 2 1 0 3 2 3 1 3
 3 0 3 2 3 0 1 3 2 3 2 1 2 2 3 0 3 2 0 3 0 1 2 2 3 2 2 1 0 1 1 2 3 2 0 1 3 3 3 3 0 0 3 1 0 1 1 3 3 1 3 1 0 0 2
 1 1 1 1 2 2 0 2 1 0 1 2 3 0 1 2 0 1 1 0 1 0 3 1 2 1 1 2 3 0 0 1 3 1 2 0 1 1 0 1 0 0 2 2 0 1 2 0 1 2 0 0 1 1 0
 1 2 3 0 1 2 3 0 0 3 2 3 0 3 1 3 1 3 0 1 3 3 1 1 2 2 2 3 1 1 3 1 3 3 0 1 1 2 0 2 2 3 1 0 3 2 1 0 2 3 1 0 2 0 0
 3 1 1 2 3 3 1 2 2 3 0 3 3 3 1 0 2 0 0 3 1 1 0 1 0 3 1 3 1 0 0 1 3 1 2 0 0 0 1 1 0 0 2 0 0 2 2 3 2 3 3 3 0 3 1
 1 1 1 3 1 1 1 2 3 0 2 3 3 1 1 3 3 3 3 3 0 0 3 2 0 3 2 1 1 3 2 1 2 1 1 1 3 3 2 3 1 1 1 2 0 2 1 1 0 0 3 1 2 3 0
 2 0 2 0 2 3 3 2 2 0 0 2 0 0 0 1 3 2 2 1 1 2 1 1 0 1 2 1 0 0 2 2 0 3 3 0 0 2 1 3 2 0 3 3 1 2 1 1 3 0 3 3 0 0 1
 2 3 1]
```

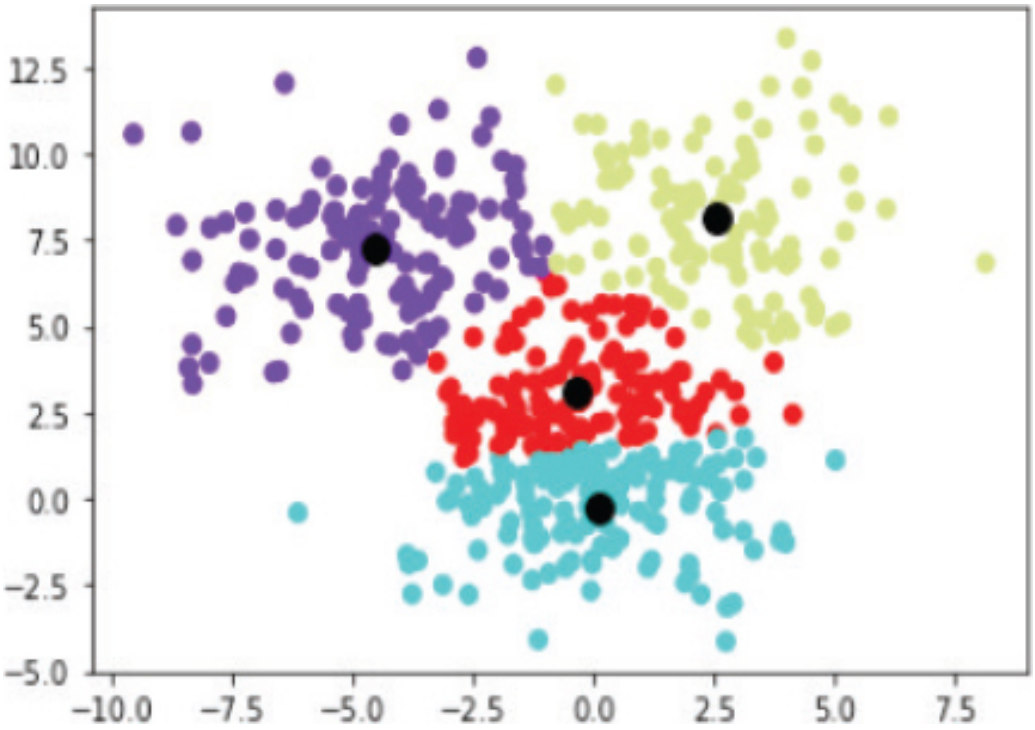
The following script prints the clusters in different colors along with the cluster centers as black data points, as shown below.

Script 6:

```
1. #pring the data points
2. plt.scatter(features[:,0], features[:,1], c= km_model.labels_, cmap='rainbow' )
3.
4. #print the centroids
5. plt.scatter(km_model.cluster_centers_[:, 0], km_model.cluster_centers_[:, 1], s=100, c='black')
```

The following output shows the four clusters identified by the K Means clustering algorithm.

Output:



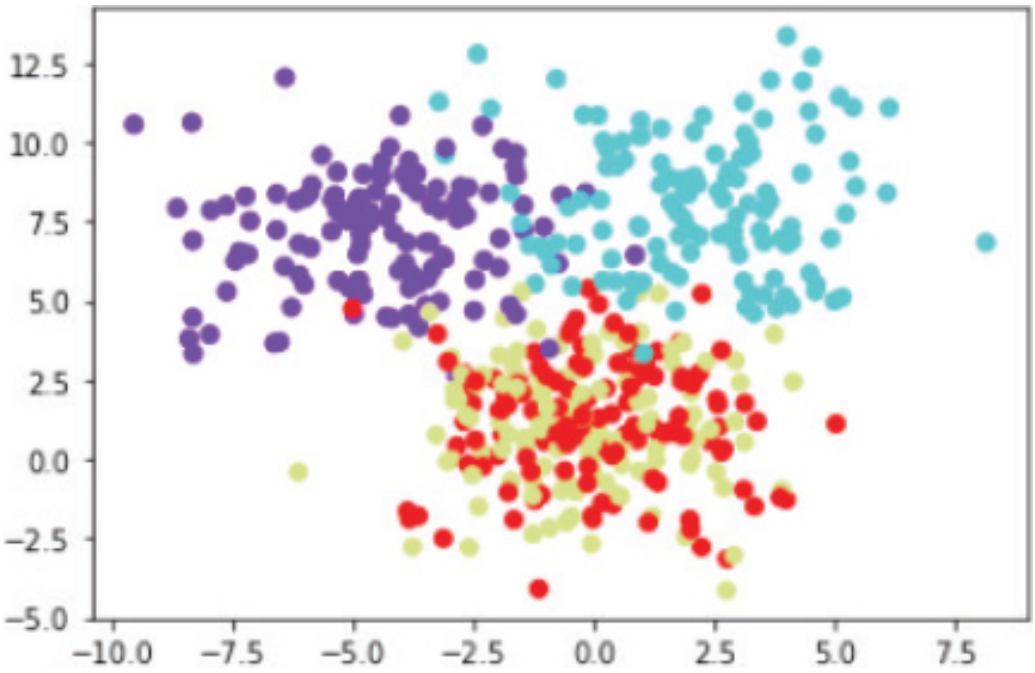
The following script prints the actual four clusters in the dataset.

Script 7:

```
1. #print actual datapoints
2. plt.scatter(features[:,0], features[:,1], c= labels, cmap='rainbow' )
```

The output shows that in the actual dataset, the clusters represented by red and yellow data points overlap. However, the predicted clusters do not contain any overlapping data points.

Output:



Note:

The color of the clusters doesn't have to be the same since cluster colors are randomly generated at runtime—only the cluster positions matter.

8.1.2. Clustering Iris Dataset

In the previous section, you saw a clustering example of some dummy dataset. In this section, we will cluster the Iris dataset. The Iris dataset can be imported via the following script.

Script 8:

```
1. import seaborn as sns
2.
3. iris_df = sns.load_dataset("iris" )
4. iris_df.head()
```

Output:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

We do not use data labels for clustering. Hence, we will separate features from labels. Execute the following script to do so:

Script 9:

```
1. # dividing data into features and labels
2. features = iris_df.drop([“species”], axis = 1)
3. labels = iris_df.filter([“species”], axis = 1)
4. features.head()
```

Here is the feature set we want to cluster.

Output:

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Let’s first choose 4 as a random number for the number of clusters. The following script performs K Means clustering on the Iris dataset.

Script 10:

```
1. # training KMeans model
```

```
2. features = features.values
3. km_model = KMeans(n_clusters=4)
4. km_model.fit(features)
```

To print labels of the Iris dataset, execute the following script:

Script 11:

```
1. print (km_model.labels_)
```

Output:

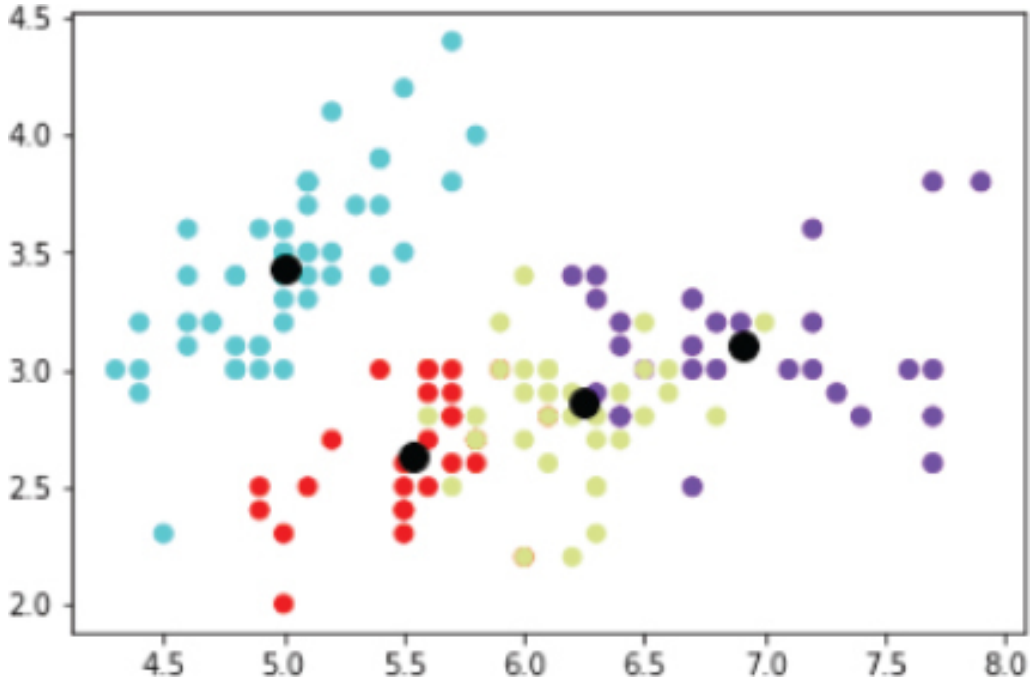
```
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 3 2 3 2
 3 2 3 3 3 3 2 3 2 3 3 2 3 2 3 2 2 2 2 2 2 2 3 3 3 3 2 3 2 2 2 3 3 3 2 3 3 3 3 3 2 3 3 0 2 0 0 0 0 3 0 0 0 2 2
 0 2 2 0 0 0 0 2 0 2 0 2 0 0 2 2 0 0 0 0 0 2 2 0 0 0 2 0 0 0 2 0 0 0 2 2 0 2 0]
```

Finally, to plot the 4 clusters found by the K Means algorithm in the Iris dataset, along with the predicted cluster centroids, execute the following script.

Script 12:

```
1. #pring the data points
2. plt.scatter(features[:,0], features[:,1], c= km_model.labels_, cmap='rainbow' )
3.
4. #print the centroids
5. plt.scatter(km_model.cluster_centers_[:, 0], km_model.cluster_centers_[:, 1], s=100, c='black' )
```

Output:



Till now, in this chapter, we have been randomly initializing the value of K or the number of clusters. However, there is a way to find the ideal number of clusters. The method is known as the elbow method. In the elbow method, the value of inertia obtained by training K Means clusters with different number of K is plotted.

The inertia represents the total distance between the data points within a cluster. Smaller inertia means that the predicted clusters are robust and close to the actual clusters.

To calculate the inertia value, you can use the inertia_attribute of the KMeans class object. The following script creates inertial values for K=1 to 10 and plots in the form of a line plot, as shown below:

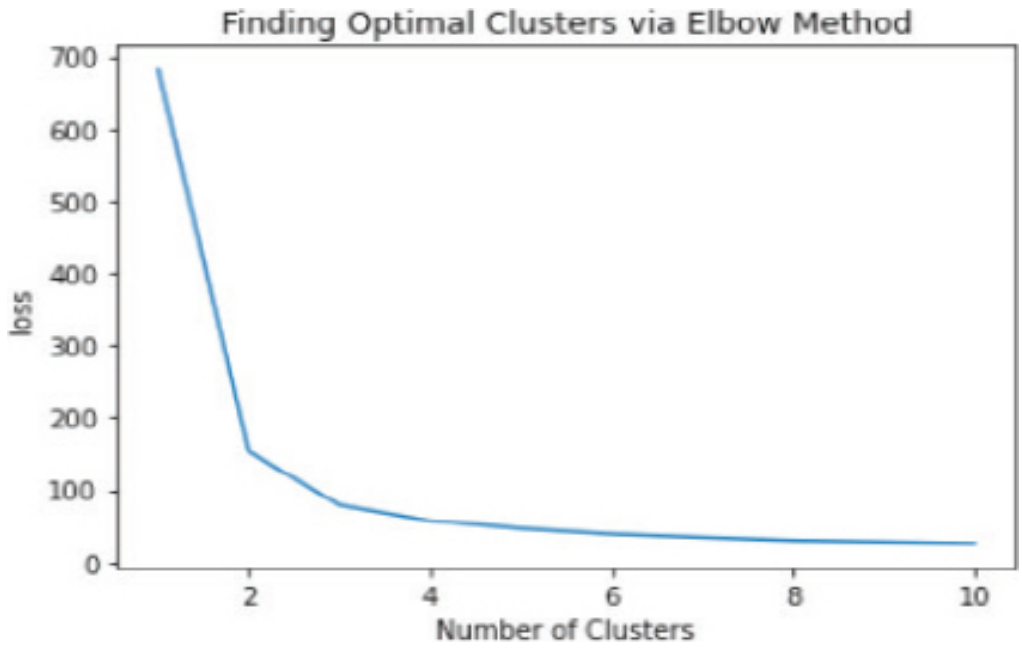
Script 13:

```
1. # training KMeans on K values from 1 to 10
2. loss =[]
3. for i in range(1, 11):
4.                 km = KMeans(n_clusters = i).fit(features)
5.                 loss.append(km.inertia_)
6.
7. #printing loss against number of clusters
8.
9. import matplotlib.pyplot as plt
10. plt.plot(range(1, 11), loss)
11. plt.title('Finding Optimal Clusters via Elbow Method' )
```

```
12. plt.xlabel('Number of Clusters' )
13. plt.ylabel('loss' )
14. plt.show()
```

From the output below, it can be seen that the value of inertia didn't decrease much after 3 clusters.

Output:



Let's now cluster the Iris data using 3 clusters and see if we can get close to the actual clusters.

Script 14:

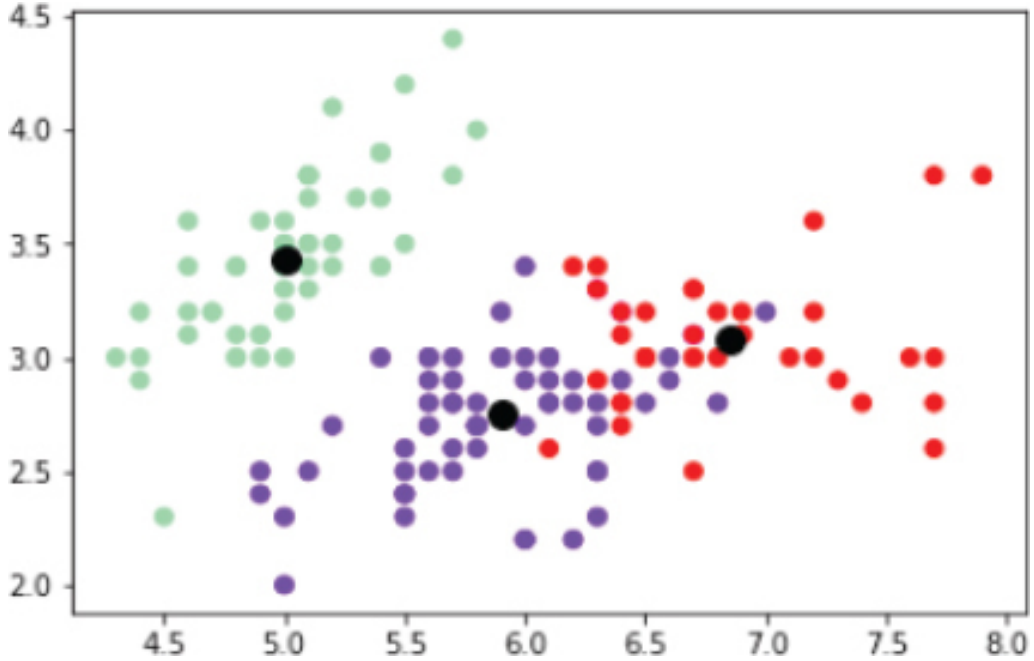
```
1. # training KMeans with 3 clusters
2. km_model = KMeans(n_clusters=3)
3. km_model.fit(features)
```

Script 15:

```
1. #pring the data points with prediced labels
2. plt.scatter(features[:,0], features[:,1], c= km_model.labels_, cmap='rainbow' )
3.
4. #print the predicted centroids
5. plt.scatter(km_model.cluster_centers_[:, 0], km_model.cluster_centers_[:, 1], s=100, c='black' )
```

When K is 3, the number of clusters predicted by the K Means clustering algorithm is as follows:

Output:



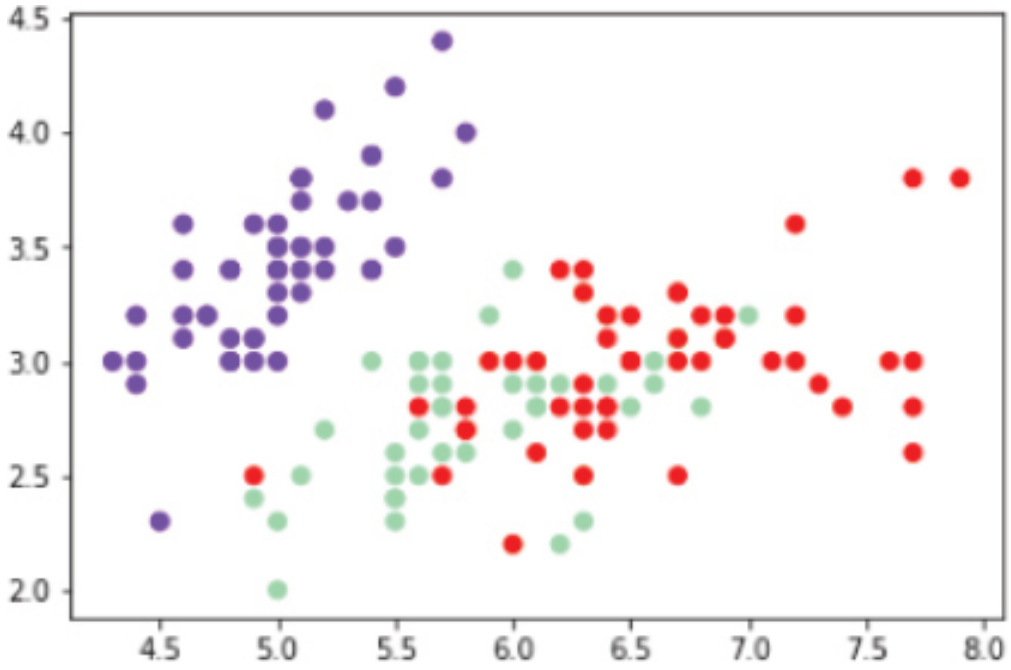
Let's now plot the actual clusters and see how close the actual clusters are to predicted clusters.

Script 16:

```
1. # converting categorical labels to numbers
2.
3. from sklearn import preprocessing
4. le = preprocessing.LabelEncoder()
5. labels = le.fit_transform(labels)
6.
7. #pring the data points with original labels
8. plt.scatter(features[:,0], features[:,1], c= labels, cmap='rainbow' )
```

The output shows that the actual clusters are pretty close to predicted clusters.

Output:


4.5
4.0
3.5
3.0
2.5
2.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0