Basics of Machine Learning: K-Means Clustering

Eshita Goel
Aug 20 · 6 min read

As we dive into the world of "Unsupervised" Machine Learning, we will encounter problems that would require us to cluster the data available to us. This means we have to divide the data into clusters based on their level of similarity. K-Means Clustering allows us to do just that.

As the name suggests, the algorithm makes use of the "means" of the data to cluster them. Here "K" is just the number of clusters we want our data to be divided in. We have to choose the value of "K" ourselves, and there are ways in which can select the right "K" for our data.

About Unsupervised Machine Learning

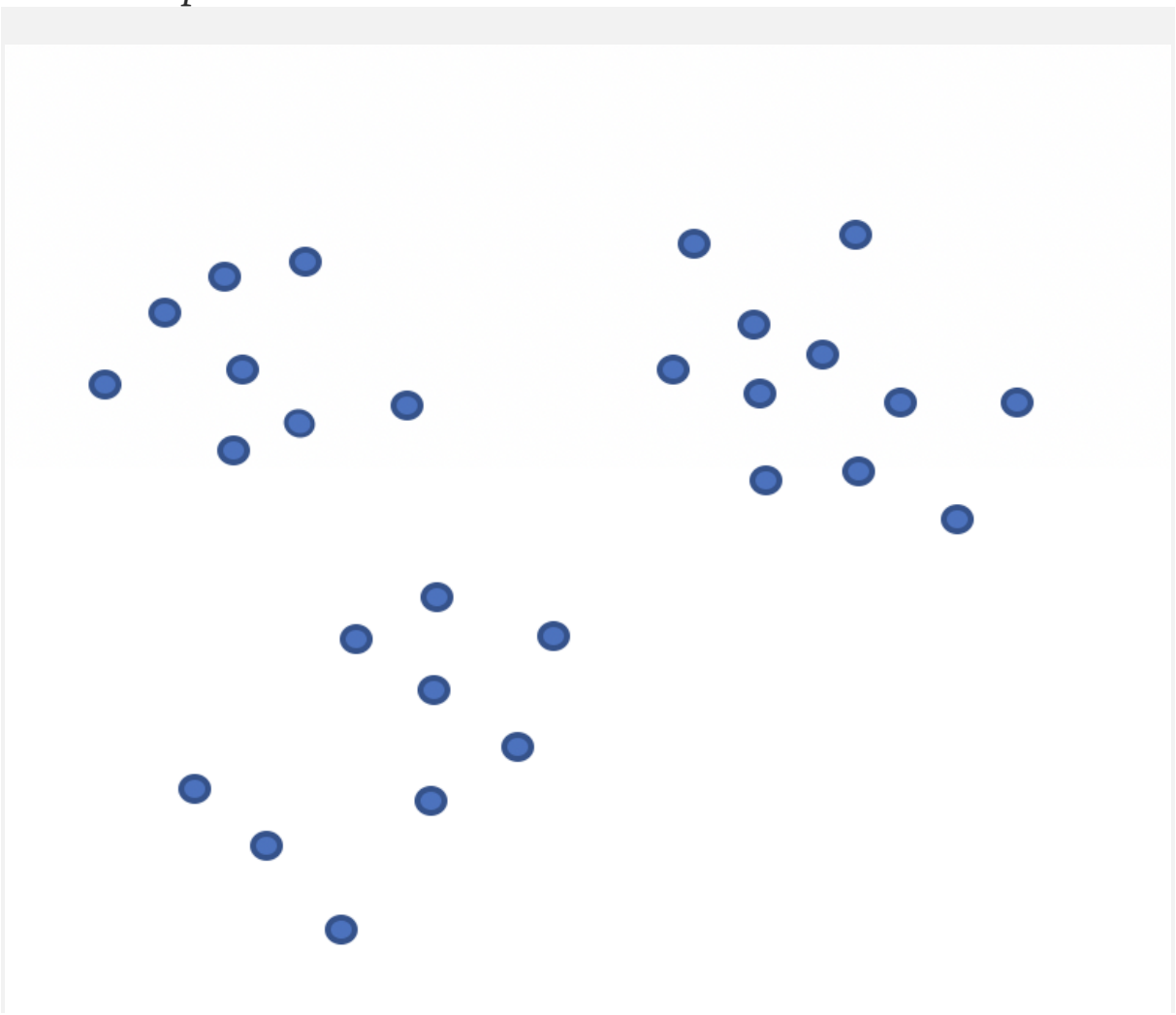
When we are dealing with unlabelled data, we are dealing with unsupervised machine learning. So our data is not already labelled. K-Means clustering is a form of unsupervised machine learning.

How does K-Means Clustering work?

We explore the logic behind K-Means using an example. Suppose we have a dataset consisting of multiple rows. We want to divide

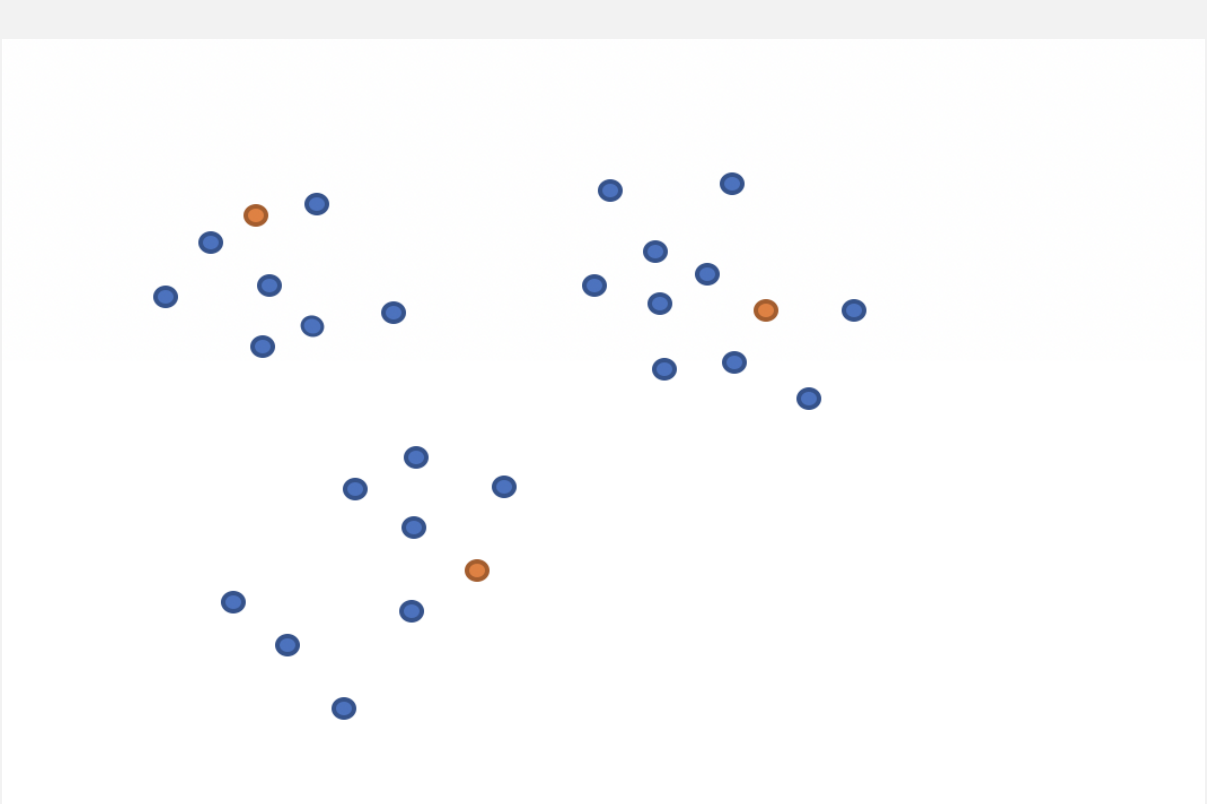
the dataset into clusters based on how similar the records are to each other.

The K-Means Clustering Algorithms works in the following way: Suppose these are our data points that we want to cluster. We have to first decide a value of K. We will see how we can optimize this value and choose the right K, but first let's assume we want to divide the points into 3 clusters.



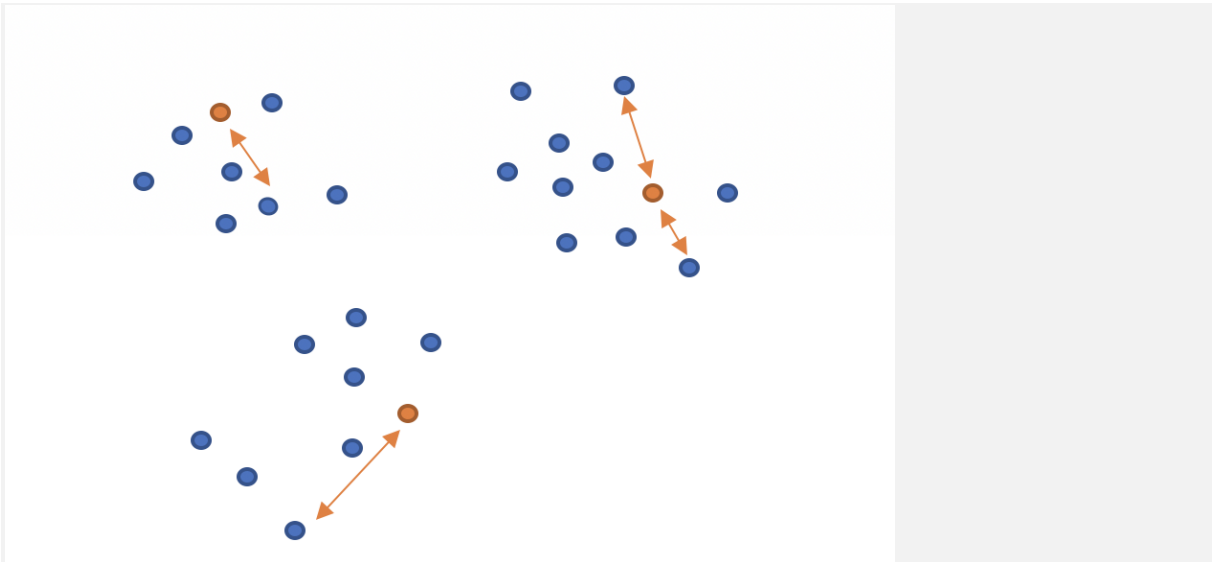
Our random data set

1. *Using the value of K that we supply, it chooses k random points that may or may not be the means of the dataset. These will be our cluster "centroids". They are called centroids even though they are randomly chosen.*



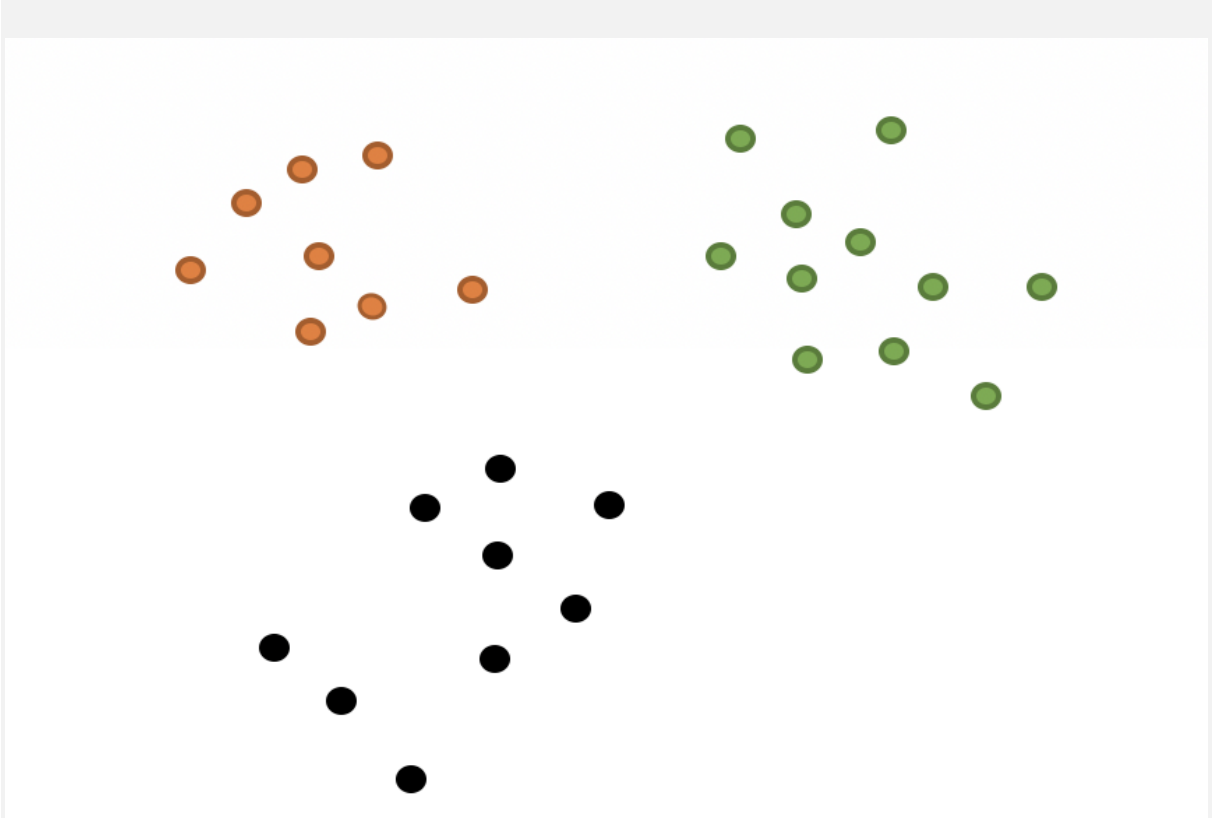
Choosing K random centroids

2. It calculates the distance of each point of the dataset from these centroids and assigns that data point to that cluster whose centroid is closest to the data point.



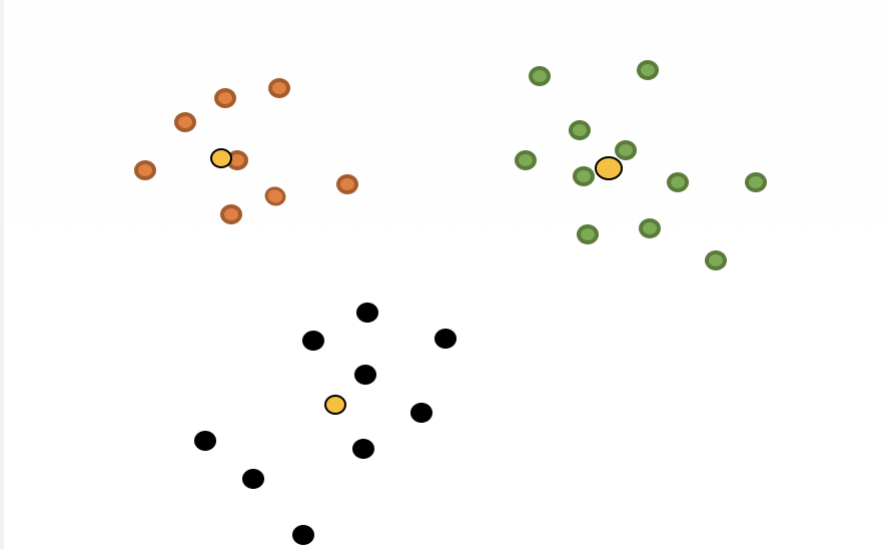
Assigning each data point to a cluster based on the minimum distance from the centroids

3. And thus, in our first iteration, it clusters our data. We cannot say that the clusters obtained are the ideal clusters, as our choice of the k centroids was arbitrary. We have to do more iterations to attain better accuracy.



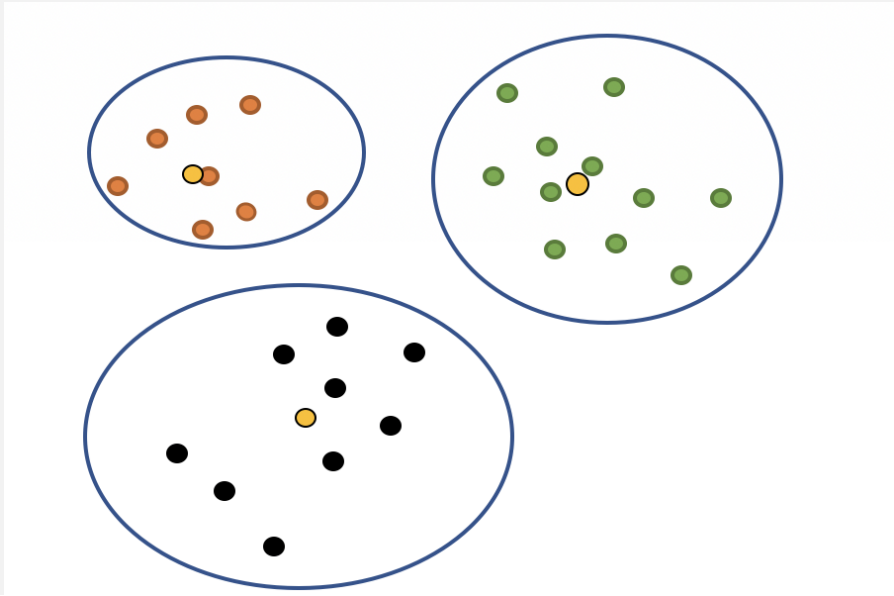
Clustered data after the first iteration

4. Now, we calculate the centroids of these clusters. These calculated centroids would serve as the centroids for our next iteration!



New centroids calculated

5. The algorithm runs again and again until either the centroids no longer need to be re-allocated or the maximum number of iterations are reached. We finally get our clustered data points!



Our final clusters

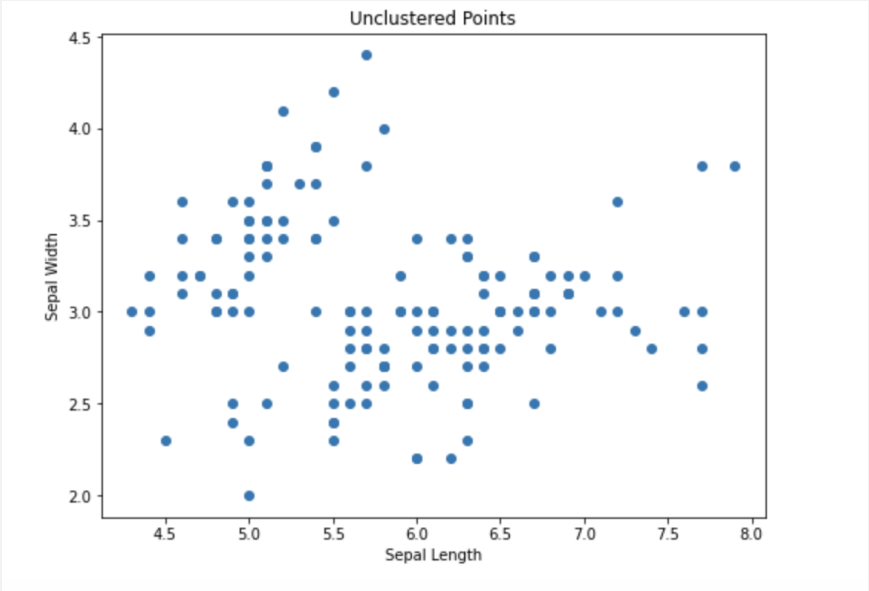
Let us take a very popular dataset, called the iris dataset, as our example.

```
from sklearn import datasets
iris = datasets.load_iris()
df = pd.DataFrame(iris.data, columns = iris.feature_names)
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
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

We first plot the dataset to see our points:

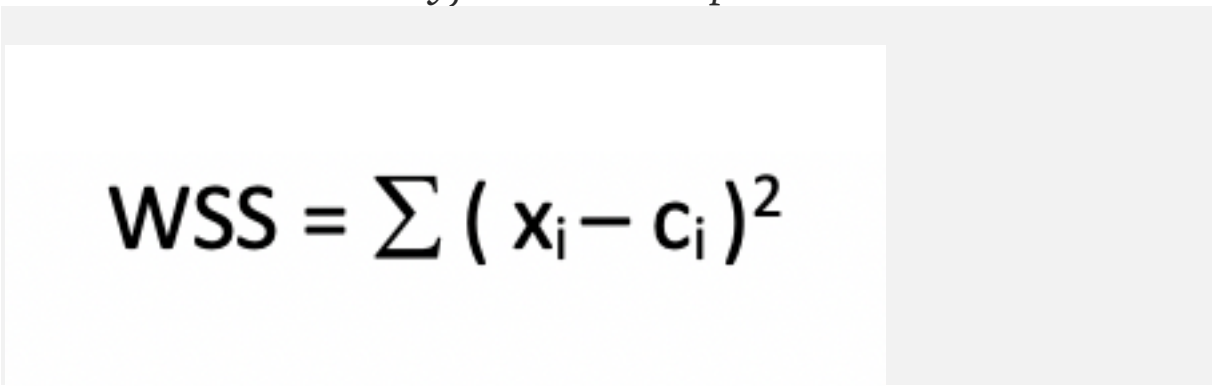
```
# Plotting the points
plt.figure(figsize=(8,6))
plt.scatter(X[:,0],X[:,1])
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.title("Unclustered Points")
plt.show()
```



Finding the right K value — The Elbow Method

The Elbow Method is one of the best ways to find out which value of 'K' will give us the most stable clusters. The Elbow Method is based on the concept that the best value of 'K' is the one where the 'Within Sum of Squares' value is the least. The total WSS measures the compactness of the clustering and it should be as small as possible.

The **'Within Sum of squares' or WSS** *value is defined as the sum of the squared distance between each member of the cluster and its centroid. Mathematically, this can be represented as:*



Formula for WSS

Here the x values are the data points and the corresponding c values are the centroids to which these data points are assigned. We want a minimum WSS value for good clustering. When we plot this value along with the corresponding value of 'k', we will observe a sharp decrease in the WSS value at some value of 'k'. After this, the value will keep on decreasing, but very slowly. Clearly, the larger the k, the lesser the WSS value, but we don't want the least WSS value in that sense. We want a suitable value of 'k'

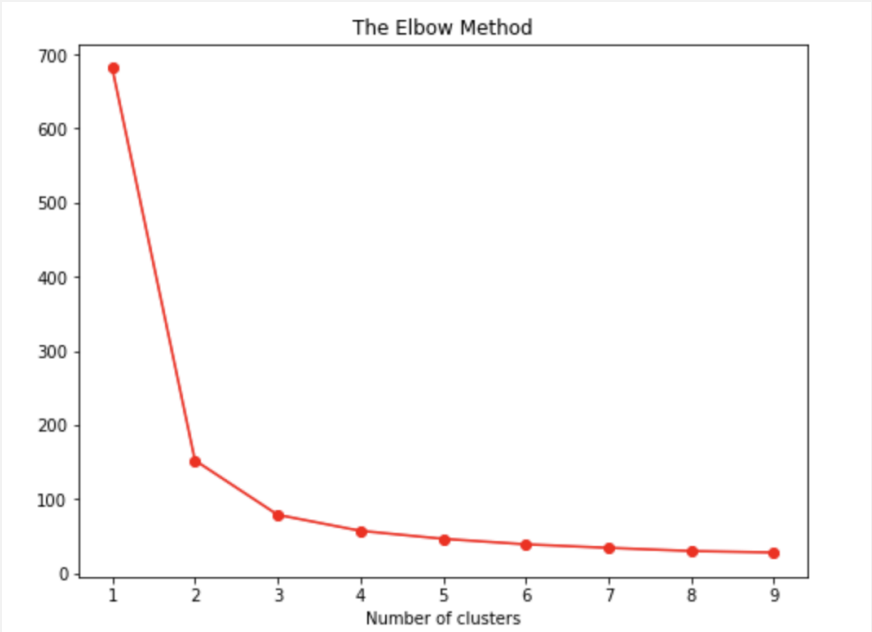
such that our WSS value is low and if we take any value larger than this 'k', the change in WSS value is minimal.

Let's try this out with our example. Execute the K-Means clustering algorithm with different value of k. Plot the graph with the k values and the corresponding values of inertia (Which can be accessed using **.inertia_** *)*

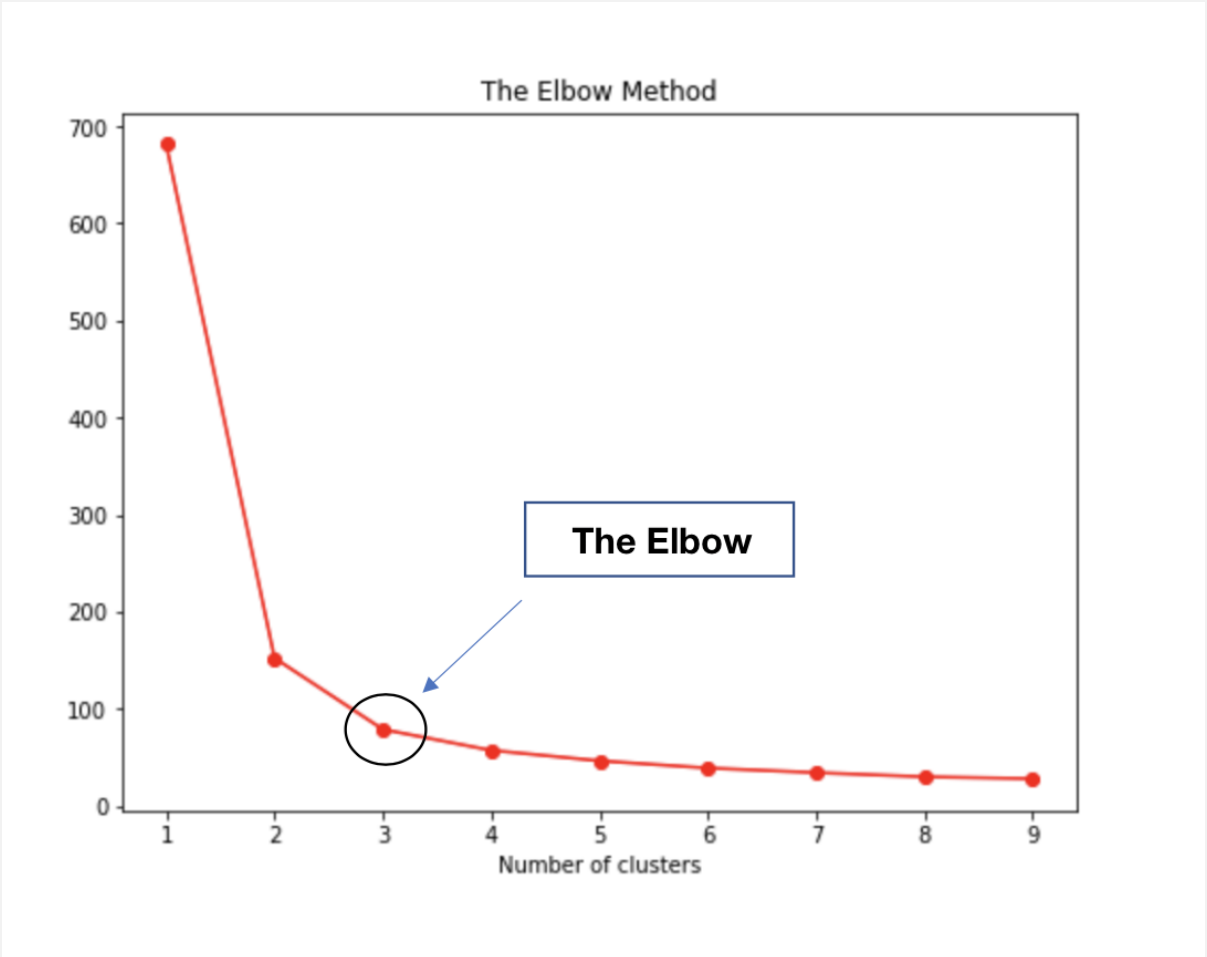
We observe that we get an 'elbow point' at k=3. This is our required k value!

```
from sklearn.cluster import KMeans
inert = []

for i in range(1, 10):
    kmeans = KMeans(n_clusters = i, init = 'k-means++',
                    max_iter = 300, n_init = 9, random_state = 0)
    kmeans.fit(X)
    inert.append(kmeans.inertia_)

# Plotting the results onto a line graph,
# `allowing us to observe 'The elbow'

plt.figure(figsize=(8,6))
plt.plot(range(1, 10), inert, marker="o",c="r")
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.show()
```

Commands for finding the right k-value



We get this graph after plotting



We can see the elbow point at k = 3

Finally, clustering our data!

Now that we know the appropriate value of k for our K-Means clustering, we can perform the clustering using this value of k.

```
# Keeping k = 3, we perform  clustering on our data

clus = KMeans(n_clusters = 3, init = 'k-means++',
              max_iter = 300, n_init = 10, random_state = 0)
y_clus = clus.fit_predict(X)

y_clus
```

Performing k-means clustering with k = 3

Here, the **y_clus** *variable is an array where each value will signify what cluster the corresponding data point at that value will belong to. Since we have k = 3, we will get 3 clusters. This means the values in y_clus would be 0, 1 or 2.*

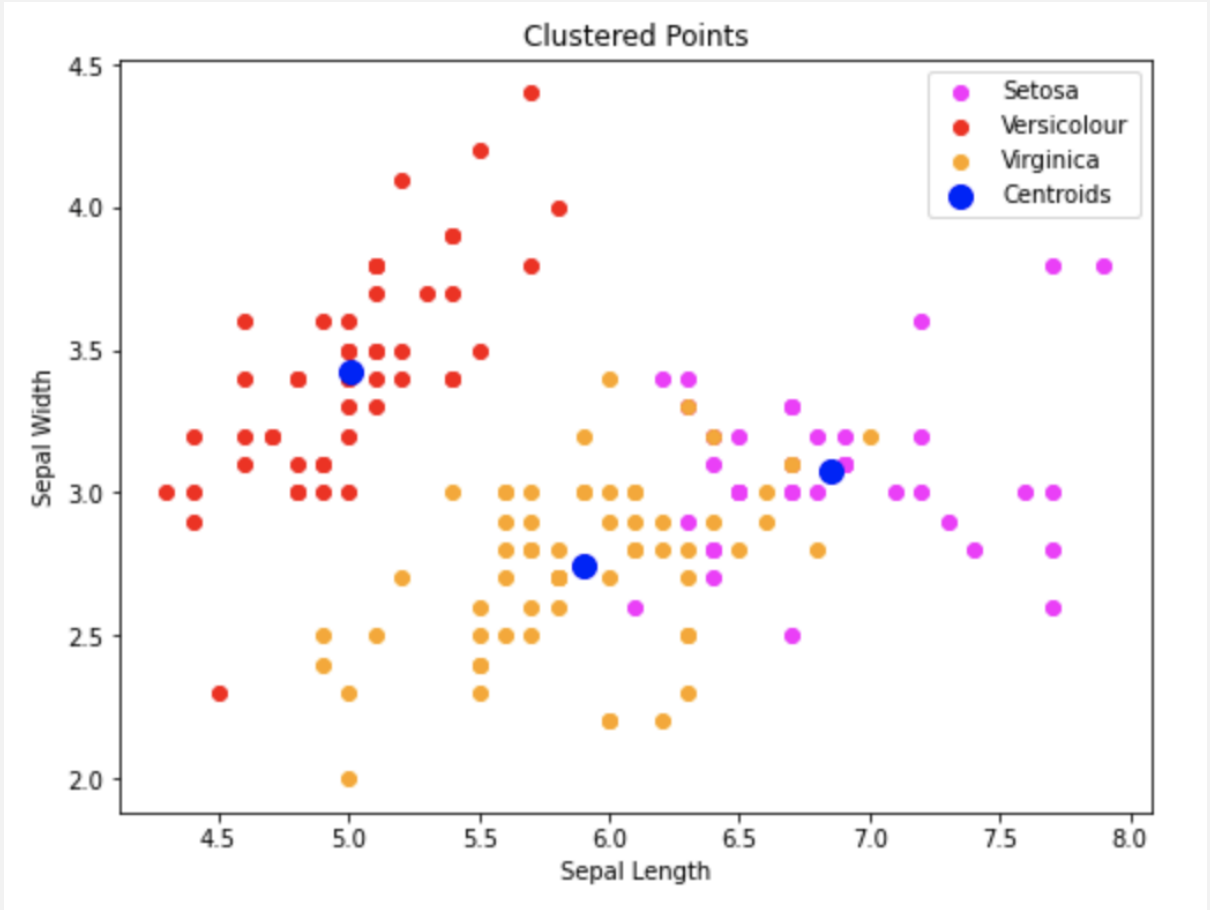
```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0,
       0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 0, 2, 0, 2, 0, 0, 2, 2, 0, 0, 0, 0,
       0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0, 2, 0, 0, 2], dtype=int32)
```

y_clus

From this we can see that the first data point belongs to Cluster-1, and the last data point belongs to Cluster-2 and so on.

Now we can plot our data points again, but this time colour code them based on their clusters and also see where our final centroids are. The centroids of a k-means clustering can be accessed using **.cluster_centers_**

```
# Visualising the clusters - On the first two columns

plt.figure(figsize=(8,6))
plt.scatter(X[y_clus == 0, 0], X[y_clus == 0, 1],
            c = 'magenta', label = 'Setosa')
plt.scatter(X[y_clus == 1, 0], X[y_clus == 1, 1],
            c = 'red', label = 'Versicolour')
plt.scatter(X[y_clus == 2, 0], X[y_clus == 2, 1],
            c = 'orange', label = 'Virginica')

# Plotting the centroids of the clusters
plt.scatter(clus.cluster_centers_[:, 0], clus.cluster_centers_[:,1],
            c = 'blue', label = 'Centroids',s=100)

plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.title("Clustered Points")

plt.legend()
plt.show()
```



Here, the big blue dots are the final centroids that gave us stable clusters.