Importing the libraries

In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import warnings
warnings.filterwarnings("ignore")
```

For a model the two basic functions can be fit and predict.

- There will also be an init method which can take some parameters as defined in scikit learn
- `n_clusters` , `random_state` , `max_iter`

In [2]:

```
class Kmeans_custom:
    # Object initialization
    def __init__(self, n_clusters=8, max_iter=300, random_state=42):
        self.n_clusters = n_clusters
        self.max_iter = max_iter
        self.random_state = random_state

        # Attributes
        self.labels_ = None
        self.cluster_centers_ = None
        self.inertia_ = None


    # Applying random centroid selection method
    def random_centroid_initialization(self, X):
        # Fixing the seed of random numbers to initialize pseudo random number gener
        np.random.RandomState(seed=self.random_state)
        len_rows = X.shape[0]
        random_idx = np.random.permutation(len_rows)
        centroids = X[random_idx[:self.n_clusters]]
        return centroids

    # Calculating the euclidean distance between the centroid and each data point
    def calculate_distance_each_point(self, X, centroids):
        distance = np.zeros((X.shape[0], self.n_clusters))
        for k in range(self.n_clusters):
            # Here for calculating euclidean distance I have used norm
            # distance between a and b
            # np.linalg.norm(a - b, axis=1)
            eucl_dist = np.linalg.norm(X - centroids[k, :], axis=1)
            distance[:, k] = np.square(eucl_dist)
        return distance

    # Finding the closest clusters by comparing the euclidean distances
    def closest_cluster(self, distances):
        return np.argmin(distances, axis=1)

    # Calculating the centroids for clusters
    def calculate_cluster_centers_(self, X, labels):
        centroids = np.zeros((self.n_clusters, X.shape[1]))
        for k in range(self.n_clusters):
            centroids[k, :] = np.mean(X[labels == k, :], axis=0)
        return centroids

    """
    Calculating the sum of squared distances of samples to their closest cluster cer
    weighted by the sample weights if provided.
    """
    def calculate_inertia_(self, X, labels, centroids):
        distance = np.zeros(X.shape[0])
        for k in range(self.n_clusters):
            distance[labels == k] = np.linalg.norm(X[labels == k] - centroids[k], ax
        return np.sum(np.square(distance))

    # fit method takes in dependent variable
    def fit(self, X):
        # First initialize n_clusters centroids in the data randomly
        self.cluster_centers_ = self.random_centroid_initialization(X)
```

```
        for _ in range(self.max_iter):
            previous_centroids = self.cluster_centers_

            # Calculate the distance and then label it according to the closest dist
            distances = self.calculate_distance_each_point(X, previous_centroids)
            self.labels_ = self.closest_cluster(distances)

            """
            Now since you have found the closest distance and assigned label
            we can now again calculte centroid as it may be changed by addition of t
            """
            self.cluster_centers_ = self.calculate_cluster_centers_(X, self.labels_)

            # Now see if the centroids has converged
            if np.all(previous_centroids == self.cluster_centers_):
                break
            self.inertia_ = self.calculate_inertia_(X, self.labels_, self.cluster_ce

    def predict(self, X):
        distance = self.calculate_distance_each_point(X, self.cluster_centers_)
        return self.closest_cluster(distance)
```

Loading dataset from sklearn

In [3]:

```
from sklearn import datasets
iris = datasets.load_iris()
```

Reading the dataset into dataframe

In [4]:

```
train_df = pd.DataFrame(iris.data, columns=iris.feature_names)
```

In [5]:

```
train_df.head()
```

Out[5]:

	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

In [6]:

```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 4 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
dtypes: float64(4)
memory usage: 4.8 KB
```

In [7]:

```
train_df.describe()
```

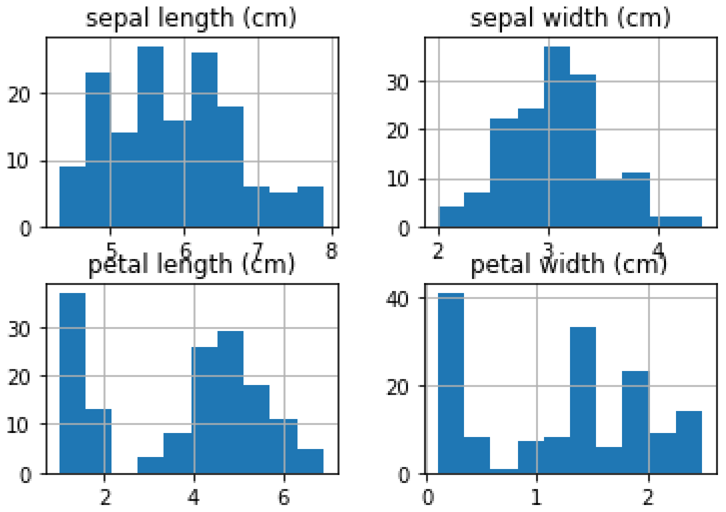
Out[7]:

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.057333	3.758000	1.199333
std	0.828066	0.435866	1.765298	0.762238
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

Plotting the histogram

In [8]:

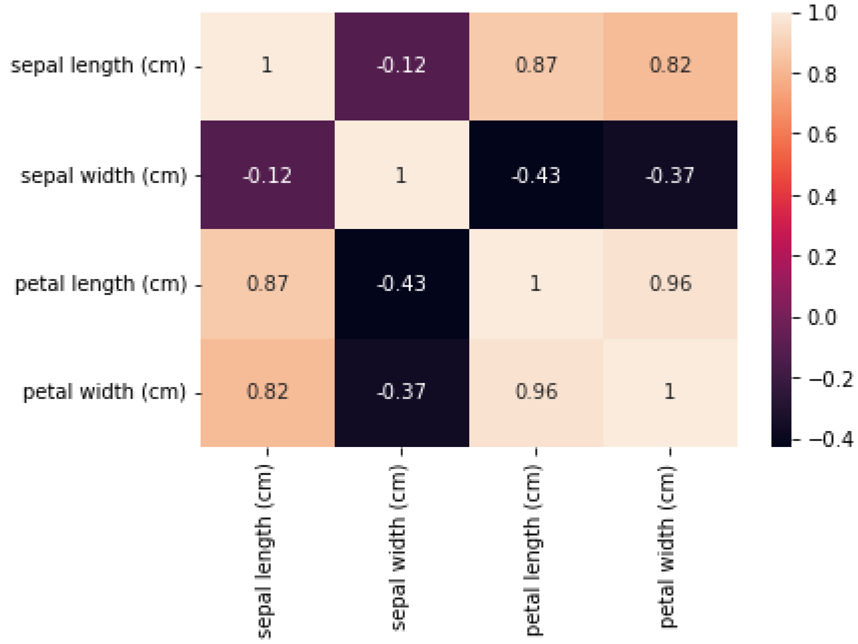
```
train_df.hist()
plt.show()
```



Plot the correlation matrix between features

In [9]:

```
sns.heatmap(train_df.corr(), annot=True)
plt.show()
```



In [10]:

```
X = train_df.values
```

Determining the values for K using the elbow method

In [11]:

```
# Implementing the elbow method
class ElbowMethod:
    def __init__(self):
        self.inertias = list()

    def calculate_inertia(self):
        for k in range(1, 10):
            km = Kmeans_custom(n_clusters=k, max_iter=100, random_state=42)
            km.fit(X)
            self.inertias.append(km.inertia_)

    def plot_elbow_curve(self):
        plt.figure(figsize=(16, 12))
        plt.plot(range(1, 10), self.inertias, 'bx-', color='red')
        plt.title('The elbow method')
        plt.xlabel('Number of clusters')
        plt.ylabel('Centers Squared Sum')
        plt.show()
```

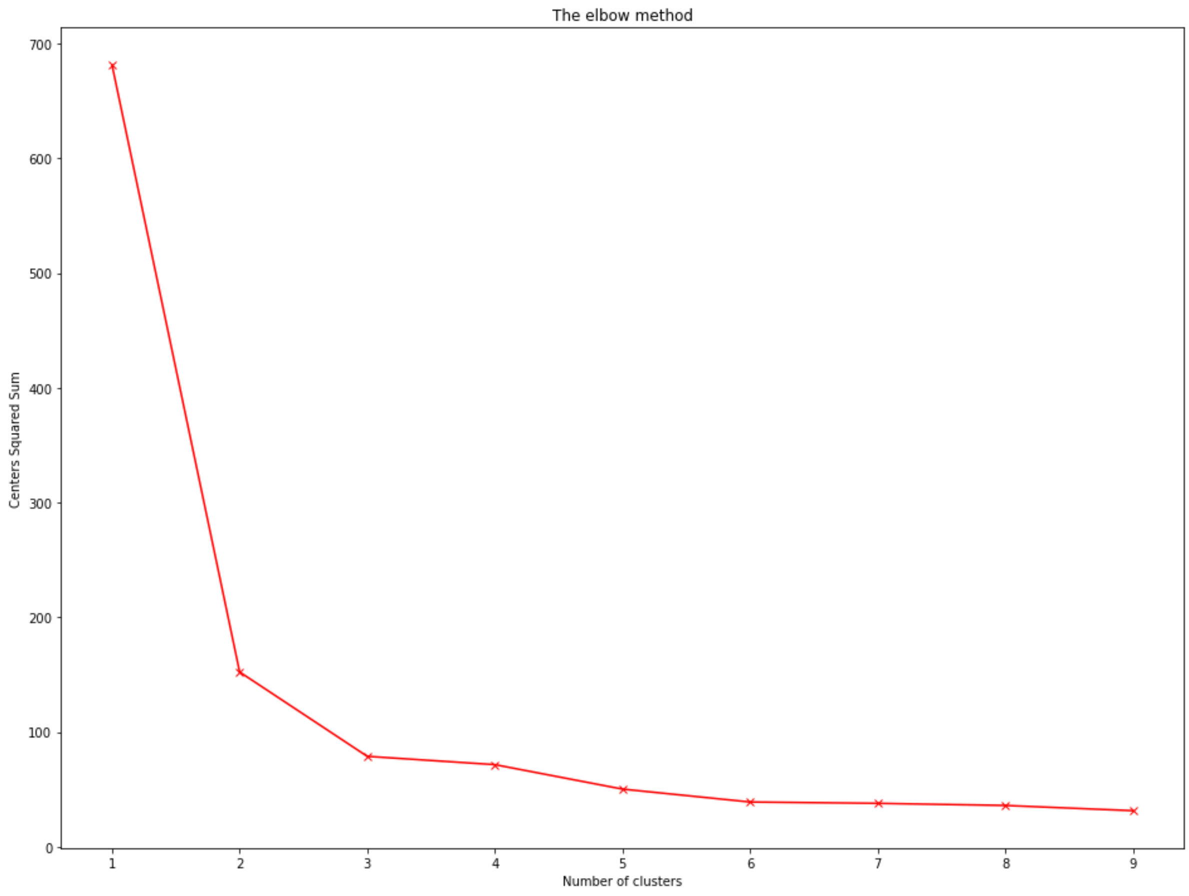
In [12]:

```
em = ElbowMethod()
em.calculate_inertia()
```

Plotting the elbow curve

In [13]:

```
em.plot_elbow_curve()
```



- From the plot we can say that minimum number of clusters should be 3

Training with the kmeans clustering model

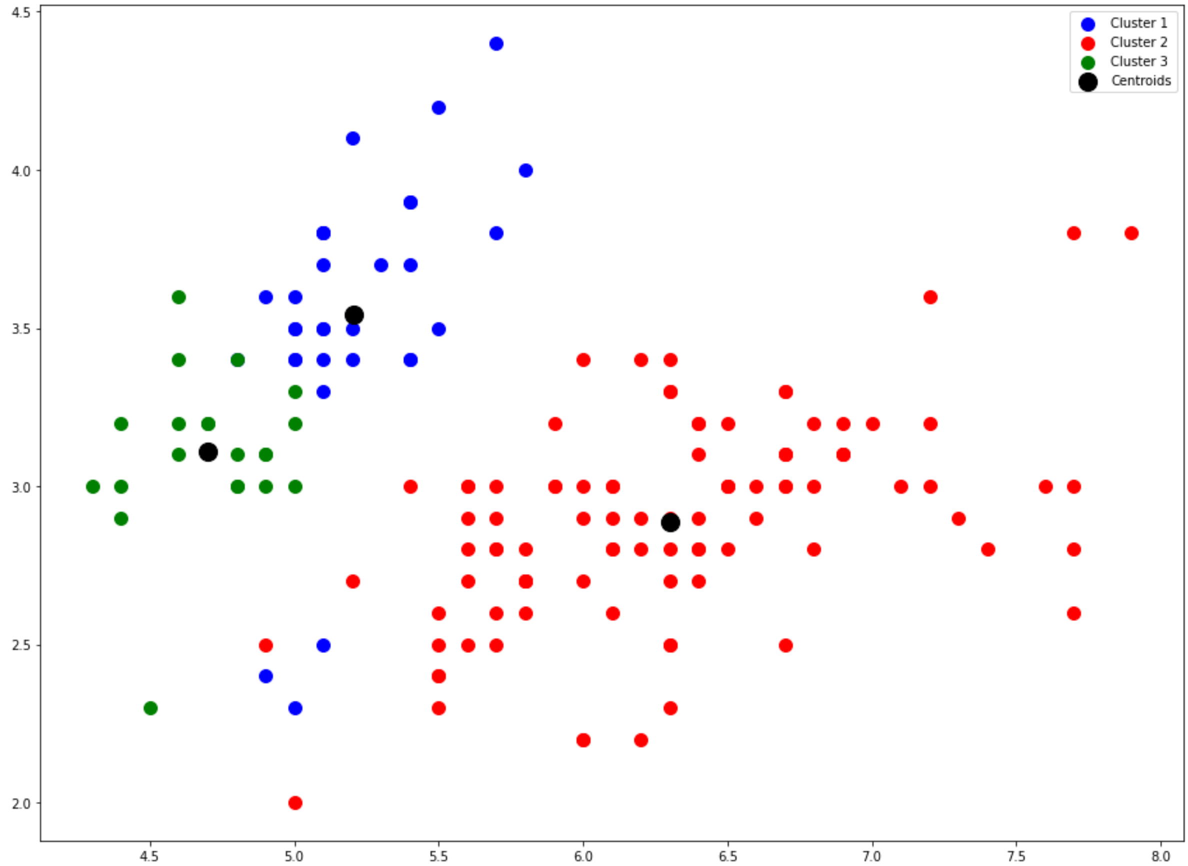
In [14]:

```
km = Kmeans_custom(n_clusters=3, max_iter=100, random_state=42)
km.fit(X)
```

Plotting the kmeans clusters

In [18]:

```
plt.figure(figsize=(16,12))

# scatter plot for label 0 cluster 1
plt.scatter(
    X[km.labels_ == 0, 0],
    X[km.labels_ == 0, 1],
    s = 100,
    c = 'blue',
    label = "Cluster 1"
)

# scatter plot for label 1 cluster 2
plt.scatter(
    X[km.labels_ == 1, 0],
    X[km.labels_ == 1, 1],
    s = 100,
    c = 'red',
    label = "Cluster 2"
)

# scatter plot for label 2 cluster 3
plt.scatter(
    X[km.labels_ == 2, 0],
    X[km.labels_ == 2, 1],
    s = 100,
    c = 'green',
    label = "Cluster 3"
)


# Plotting the centroids of the clusters
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:,1],
            s = 200, c = 'black', label = 'Centroids')

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


Cluster 1
Cluster 2
Cluster 3
Centroids
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

In [ ]: