KMeans algorithm

December 2, 2023

[1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
# from sklearn.datasets import load_iris
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import MinMaxScaler
```

[2]:
```
iris = pd.read_csv("IRIS.csv")
x = iris.iloc[:, [0, 1, 2, 3]].values
```

The second line involves using the **iloc** method to select specific columns from the DataFrame **iris**. The list **[0, 1, 2, 3]** inside the square brackets specifies the columns to be selected. In this case, it selects columns at positions 0, 1, 2, and 3. The **.values** converts the selected data from the DataFrame into a NumPy array. The resulting array is assigned to the variable **x**.

[3]:
```
iris.head()
```

[3]:

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

This displays the first five rows of the dataset by default.

[4]:
```
iris.tail()
```

[4]:

	sepal_length	sepal_width	petal_length	petal_width	species
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

This displays the last the five rows of the dataset by default.

[5]:
```
type(iris)
```

[5]:
```
pandas.core.frame.DataFrame
```

[6]:
```
print(iris)
```

```
     sepal_length  sepal_width  petal_length  petal_width         species
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
..            …            …             …            …             …
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

[150 rows x 5 columns]
```

[7]:
```
iris.info()
iris[0:10]
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
```

[7]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

The second line uses indexing to select the first 10 rows of the DataFrame **iris**. The syntax **iris[0:10]** extracts rows from index 0 to index 9 (inclusive). This is a way to visually inspect the initial rows

of the dataset and get a sense of what the data looks like. By running these two lines together, you can obtain both a summary of the dataset's information and view the first 10 rows of the data for a more detailed exploration.

[8]:
```
#Checking for missing values
missingPercentageOfValues = (iris.isnull().mean() * 100).round(2)
print(f'Percentage of Missing Values in Each Column:
 ↪\n{missingPercentageOfValues}')
```

```
Percentage of Missing Values in Each Column:
sepal_length    0.0
sepal_width     0.0
petal_length    0.0
petal_width     0.0
species         0.0
dtype: float64
```

Judging from the output, there are no missing values in the dataset.

[9]:
```
# Determining the number of observations
num_observations = iris.shape[0]
print(num_observations)
```

```
150
```

[10]:
```
# Determining the number of features
num_features = iris.shape[1]
print(num_features)
```

```
5
```

[11]:
```
irisDuplicate = iris.duplicated(keep=False)
print(iris[irisDuplicate])
```

```
     sepal_length  sepal_width  petal_length  petal_width         species
9             4.9          3.1           1.5          0.1     Iris-setosa
34            4.9          3.1           1.5          0.1     Iris-setosa
37            4.9          3.1           1.5          0.1     Iris-setosa
101           5.8          2.7           5.1          1.9  Iris-virginica
142           5.8          2.7           5.1          1.9  Iris-virginica
```

[12]:
```
#Frequency distribution of species"
iris_outcome = pd.crosstab(index=iris["species"],  # Make a crosstab
                           columns="count")      # Name the count column

iris_outcome
```

[12]:
```
col_0           count
species
Iris-setosa        50
```

```
Iris-versicolor     50
Iris-virginica      50
```

The result tabulates the counts of different categories in the "**species**" column of the DataFrame **iris**.

```
[13]: iris_setosa=iris.loc[iris["species"]=="Iris-setosa"]
iris_virginica=iris.loc[iris["species"]=="Iris-virginica"]
iris_versicolor=iris.loc[iris["species"]=="Iris-versicolor"]
```

This creates three new DataFrames ('**iris_setosa**', '**iris_virginica**', and '**iris_versicolor**') by filtering rows from the original DataFrame **iris** based on the values in the "**species**" column.

```
[15]: import os
# Set the environment variable to avoid the memory leak issue with KMeans on␣
↪Windows
os.environ['OMP_NUM_THREADS'] = '1'
# Now you can run your KMeans code
from sklearn.cluster import KMeans
# ... rest of the code ...
```

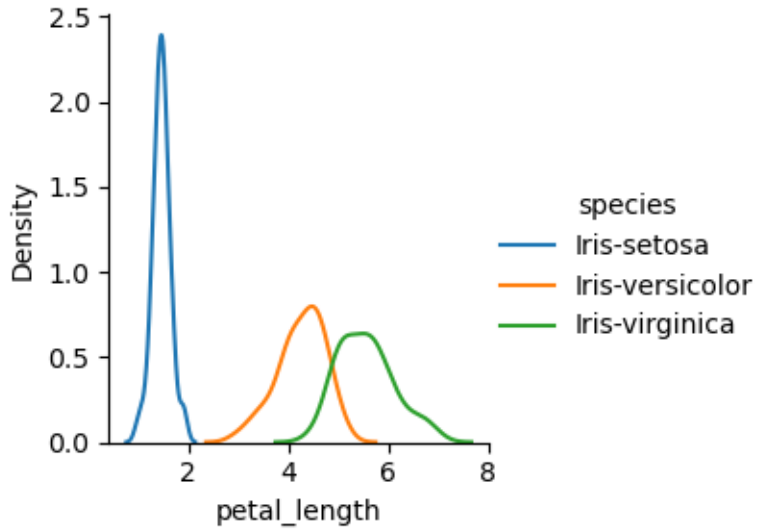
```
[17]: #Finding the optimum number of clusters for k-means classification
wcss = []

for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init␣
↪= 10, random_state = 0)
    kmeans.fit(x)
    wcss.append(kmeans.inertia_)
```

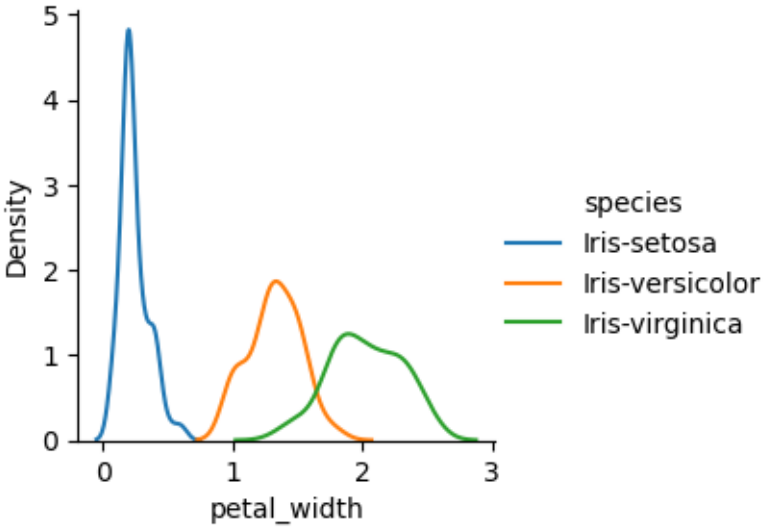
```
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
```

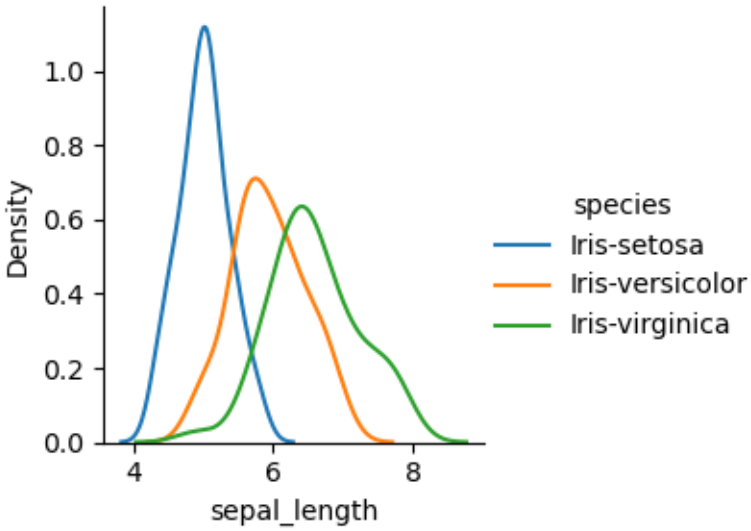
```
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```

[19]:

```
sns.FacetGrid(iris, hue="species", height=3).map(sns.kdeplot, "petal_length").
  ↪add_legend()
sns.FacetGrid(iris, hue="species", height=3).map(sns.kdeplot, "petal_width").
  ↪add_legend()
sns.FacetGrid(iris, hue="species", height=3).map(sns.kdeplot, "sepal_length").
  ↪add_legend()
plt.show()
```

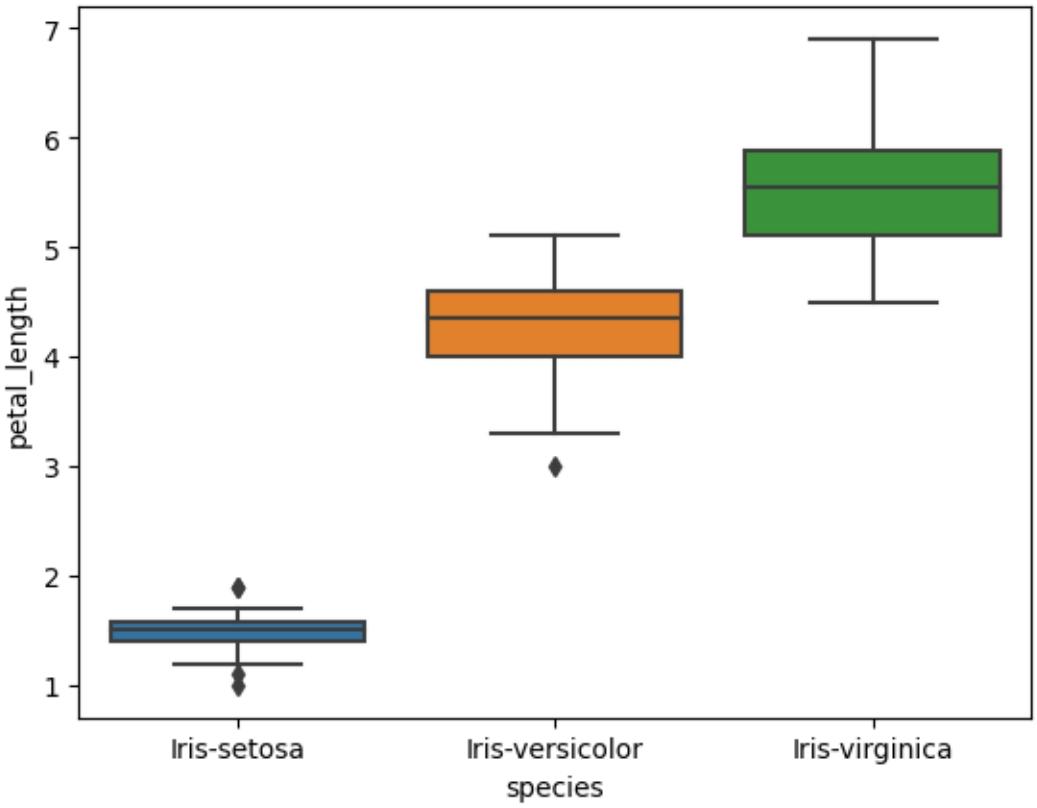

2.5
2.0
1.5
1.0
0.5
0.0
Density
2
4
6
8
petal_length
species
Iris-setosa
Iris-versicolor
Iris-virginica


5
4
3
2
1
0
Density
0
1
2
3
petal_width
species
Iris-setosa
Iris-versicolor
Iris-virginica



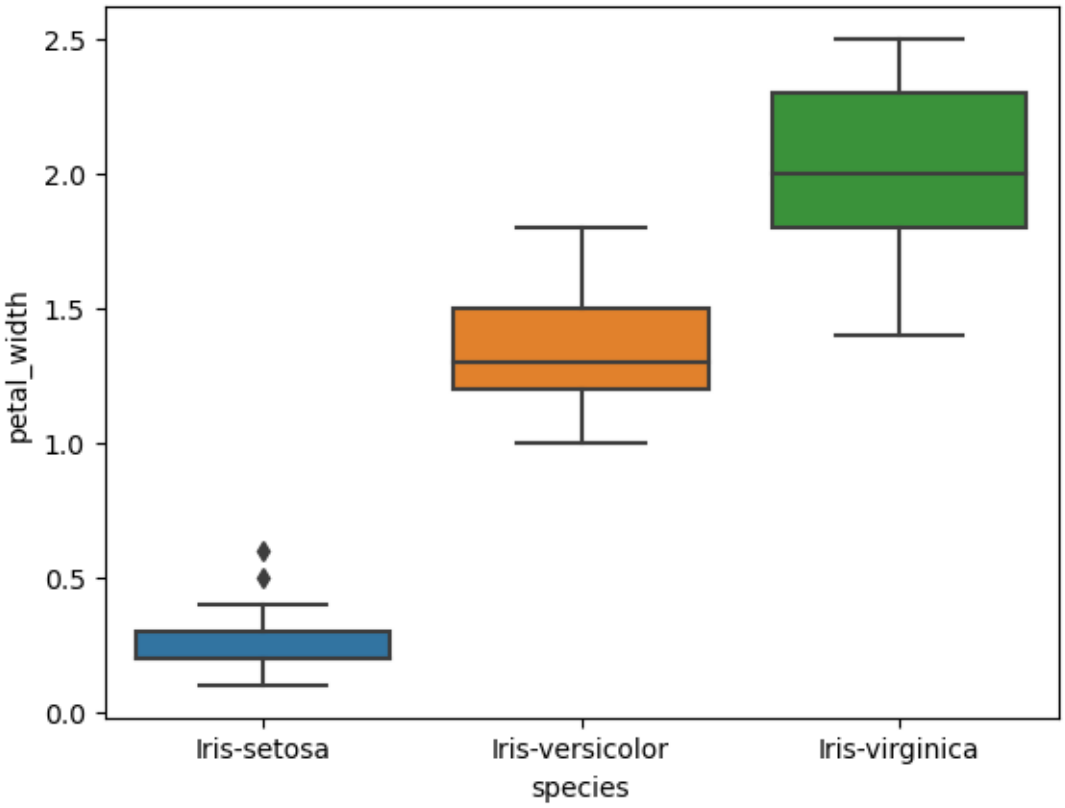
This code utilizes the Seaborn library to create three separate Kernel Density Estimation (KDE) plots for different features (petal length, petal width, and sepal length) of the Iris dataset. (Each KDE plot) The plots are organized in a FacetGrid, and each subplot is color-coded by the species of the Iris flowers.

[20]:
```
sns.boxplot(x="species", y="petal_length", data=iris)
plt.show()
```



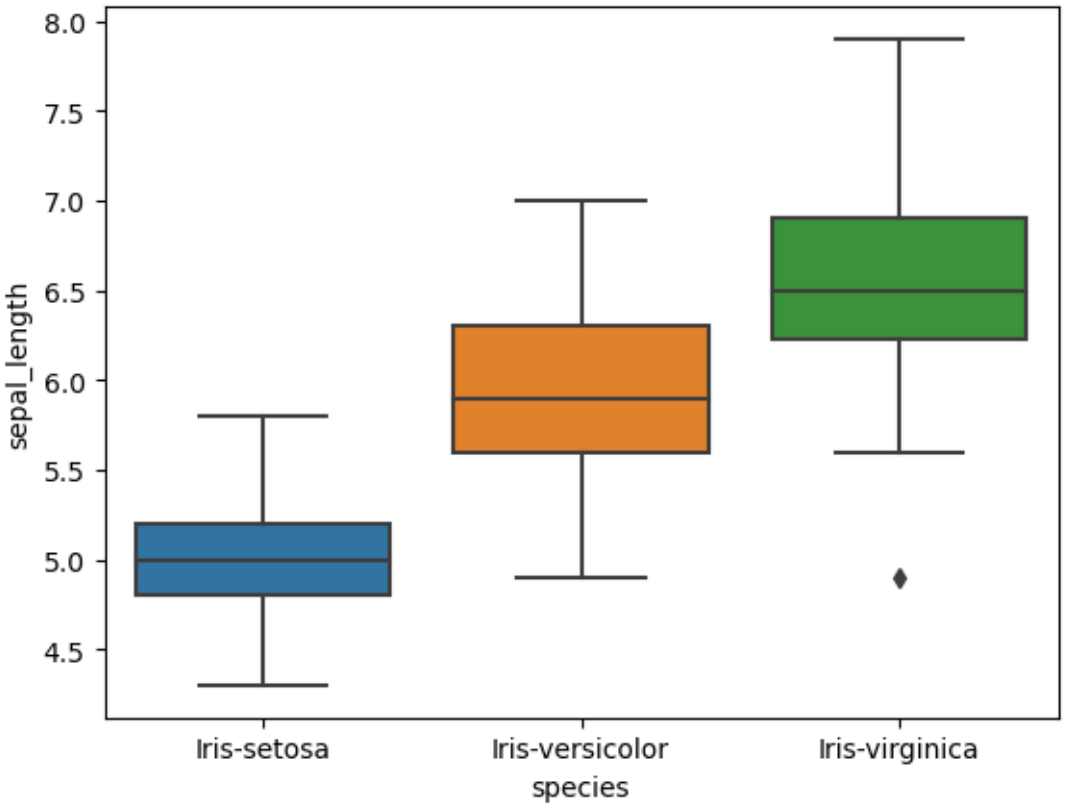
The boxplot visually represents the distribution of petal lengths for each species of Iris flowers. The box shows the interquartile range (IQR), with the median marked by a line inside the box. Whiskers extend to the minimum and maximum values within a certain range. The plot helps to compare the central tendency and spread of petal lengths across different Iris species.

[21]:
```
sns.boxplot(x="species", y="petal_width", data=iris)
plt.show()
```



The boxplot visually represents the distribution of petal widths for each species of Iris flowers. Similar to the previous boxplot for petal length, it provides a comparative view of central tendency and spread across different Iris species. The box shows the interquartile range (IQR), with the median marked by a line inside the box, and whiskers extending to the minimum and maximum values within a certain range.

[22]:
```
sns.boxplot(x="species", y="sepal_length", data=iris)
plt.show()
```



The boxplot visually illustrates the distribution of sepal lengths for each species of Iris flowers. Similar to the previous boxplots, it provides insights into the central tendency and spread of the "sepal_length" feature across different Iris species. The box in the plot represents the interquartile range (IQR), with the median marked by a line inside the box. Whiskers extend to the minimum and maximum values within a certain range.

[23]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
```

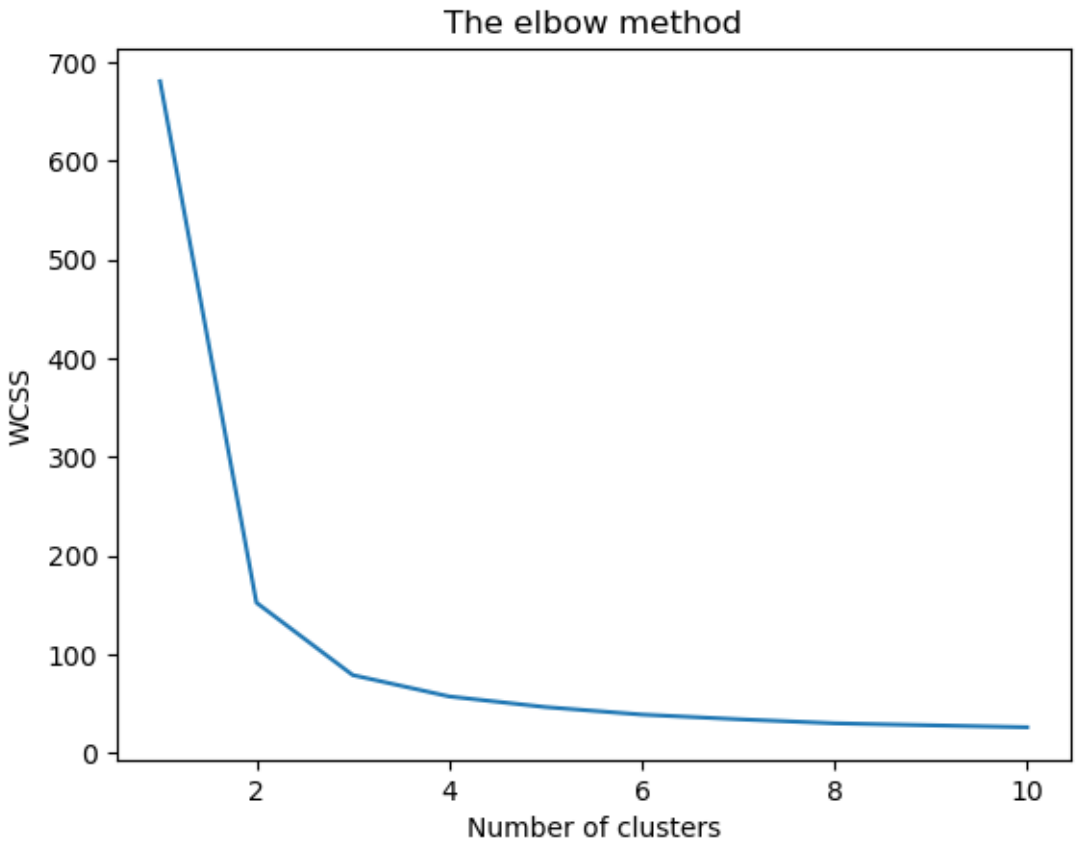
The **LabelEncoder** is a utility class that is commonly used for encoding categorical labels (text-based labels) into numerical format. Thus the code not only imports the LabelEncoder class, but also creates an instance of the class and assigns it to the variable '**le**'.

[24]:
```
iris['species'] = le.fit_transform(iris['species'])
```

This line of code uses the LabelEncoder object (**le**) to transform the values in the "species" column of the **iris** DataFrame from categorical (text-based) labels into numerical representations.

[25]:
```
# Using the elbow method to determine the optimal number of clusters for␣
  ↪k-means clustering
plt.plot(range(1, 11), wcss)
plt.title('The elbow method')
```

```
plt.xlabel('Number of clusters')
plt.ylabel('WCSS') #within cluster sum of squares
plt.show()
```



The purpose of this code is to visualize the relationship between the number of clusters and the corresponding WCSS values. The "elbow" in the plot represents a point where adding more clusters doesn't significantly reduce the WCSS, suggesting that it might be a suitable choice for the optimal number of clusters. The specific number of clusters at the elbow can guide the selection of the k value for k-means clustering in a real-world application.

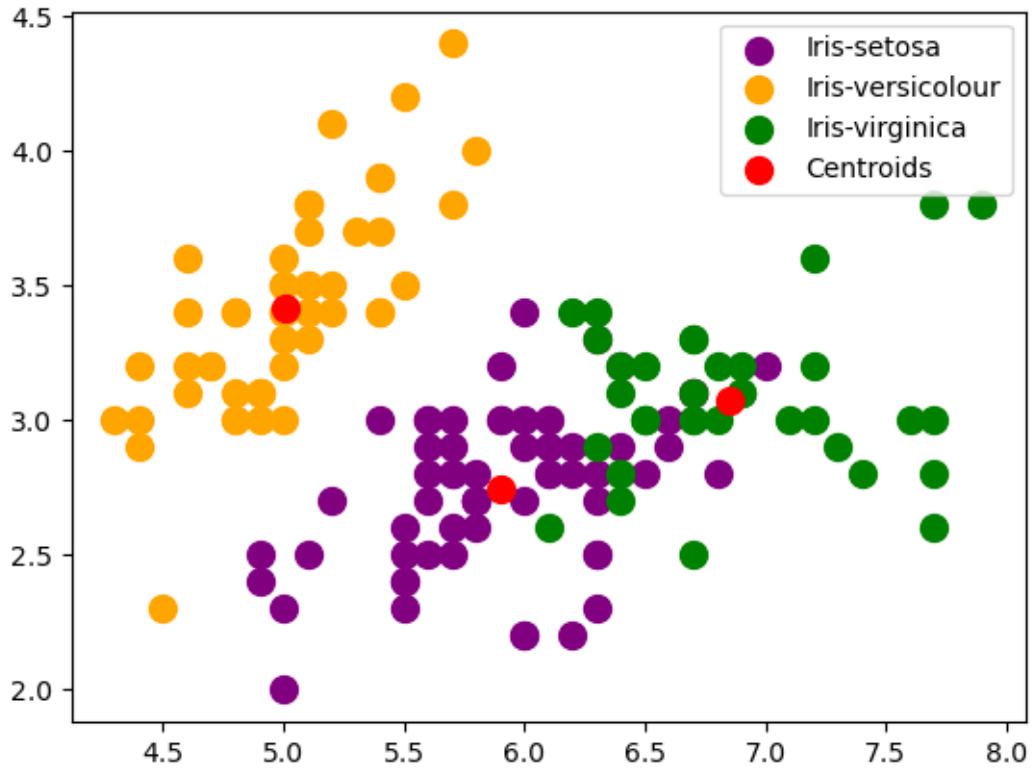
[26]:
```
# Implementing K-Means Clustering
kmeans = KMeans(n_clusters = 3, init = 'k-means++', max_iter = 300, n_init =␣
 ↪10, random_state = 0)
y_kmeans = kmeans.fit_predict(x)
```

```
C:\Users\Samuel\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436:
UserWarning: KMeans is known to have a memory leak on Windows with MKL, when
there are less chunks than available threads. You can avoid it by setting the
environment variable OMP_NUM_THREADS=1.
  warnings.warn(
```

This above code performs k-means clustering on the data (x) with three clusters, using the specified parameters for initialization, maximum iterations, and number of runs. The resulting cluster assignments are stored in y_kmeans.

[27]:
```
#Visualising the clusters
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'purple',␣
 ↪label = 'Iris-setosa')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'orange',␣
 ↪label = 'Iris-versicolour')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green',␣
 ↪label = 'Iris-virginica')

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

plt.legend()
```

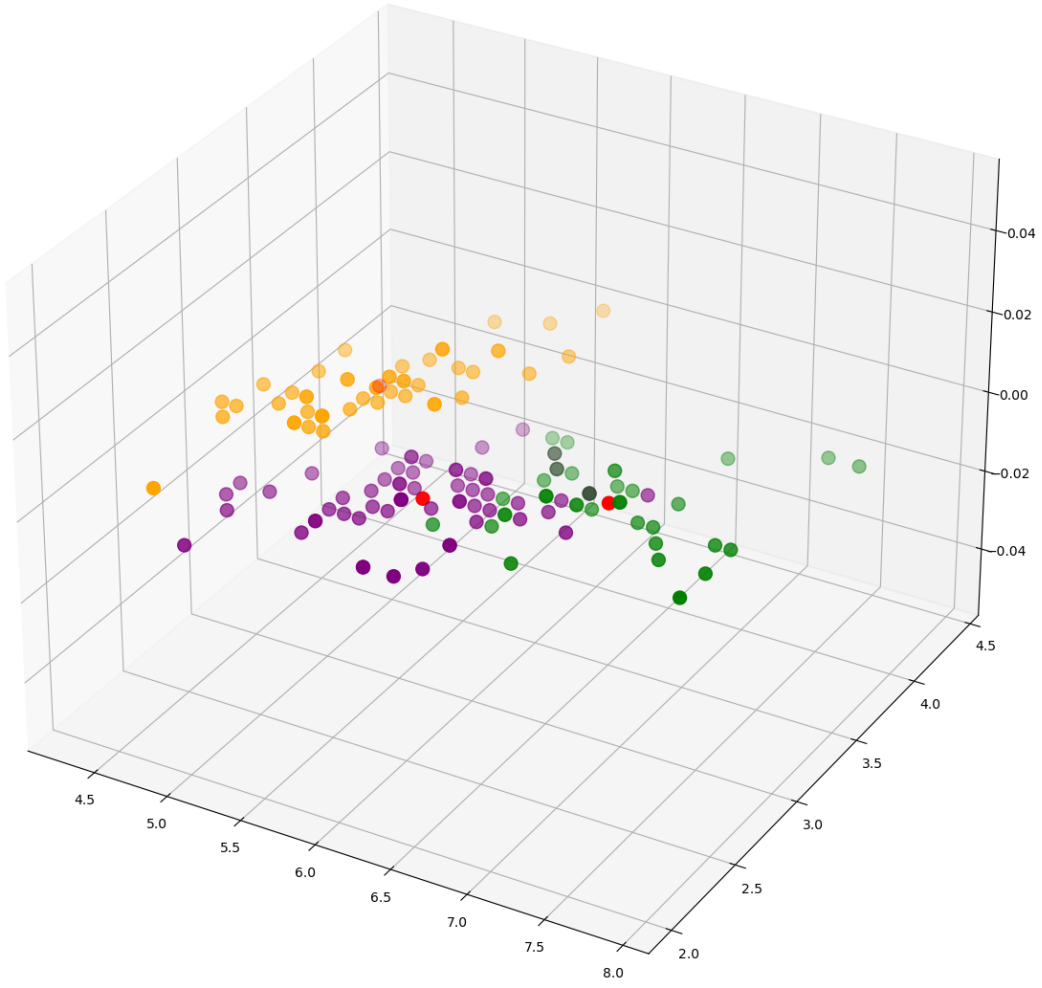
[27]:
```
<matplotlib.legend.Legend at 0x233b6c11010>
```



The above code creates a scatter plot visualizing the clustered data points with different colors for each cluster and additionally marks the cluster centroids with red markers. The legend provides

labels for each cluster and the centroids. This visualization helps to understand the distribution of the data points and the location of the cluster centroids in the feature space.

[28]:
```
# 3d scatterplot using matplotlib
fig = plt.figure(figsize = (15,15))
ax = fig.add_subplot(111, projection='3d')
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s = 100, c = 'purple',␣
 ↪label = 'Iris-setosa')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s = 100, c = 'orange',␣
 ↪label = 'Iris-versicolour')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s = 100, c = 'green',␣
 ↪label = 'Iris-virginica')

#Plotting the centroids of the clusters
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:,1], s =␣
 ↪100, c = 'red', label = 'Centroids')
plt.show()
```



The above code creates a 3D scatter plot to visualize the clustered data points with different colors for each cluster and additionally marks the cluster centroids with red markers. The legend provides labels for each cluster and the centroids. This 3D visualization helps to understand the distribution of the data points and the location of the cluster centroids in the three-dimensional feature space.