EXPLORATORY DATA ANALYISIS

30 July 2023 01:57

(Explore)

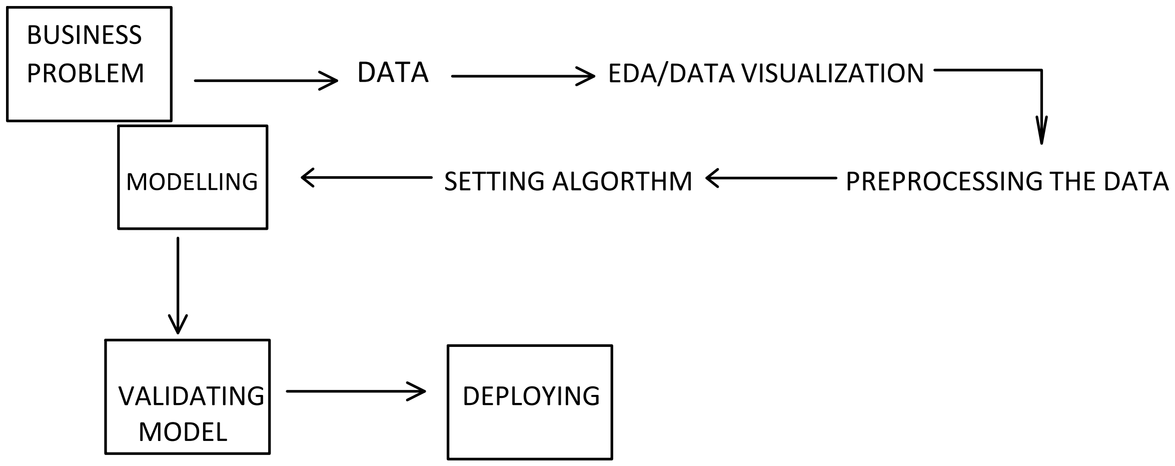
$\downarrow$

Generate Insights & it also helps in Understanding the data how to preprocess it for modelling

Ex: Before Cooking the Rice(Data) we have to check and clean (look at what type of Rice and Amount of Water needed to cook). We have to remove Gravel(clean the data)

Generate Insights is looking at the Data Visualizing. Visualization Treatment and cleaning is **EDA**

COMPLETE PROCESS



Deploying: When model is Complete we have to Build In such a way That General Audience can use it or others terms in Production Phase.

*IT Starts With Visualization and end with Visualization.

UNIVARIATE AND BIVARIATE DATA ANALYSIYS :

Data visualization is the process of representing data using visual elements like charts, graphs, etc. that helps in deriving meaningful insights from the data. It is aimed at revealing the information behind the data and further aids the viewer in seeing the structure in the data.

Need for visualizing data :

- Understand the trends and patterns of data
- Analyze the frequency and other such characteristics of data
- Know the distribution of the variables in the data.
- Visualize the relationship that may exist between different variables

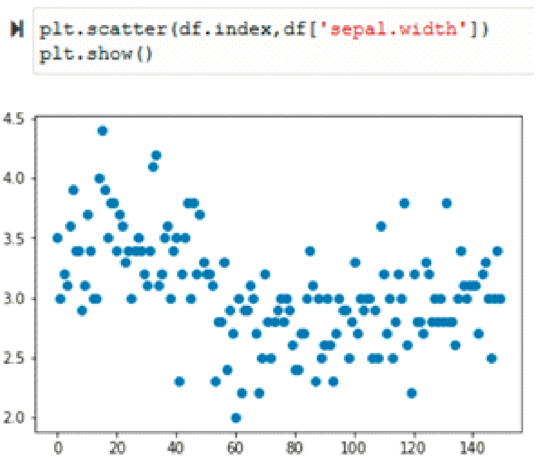
The number of variables of interest featured by the data classifies it as **univariate, bivariate, or multivariate**

Univariate enumerative Plots :

These plots enumerate/show every observation in data and provide information about the distribution of the observations on a single data variable. We now look at different enumerative plots.

1. UNIVARIATE SCATTER PLOT :

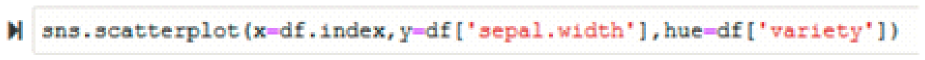
This plots different observations/values of the same variable corresponding to the index/observation number. Consider plotting of the variable ‘sepal Width(cm)’ :



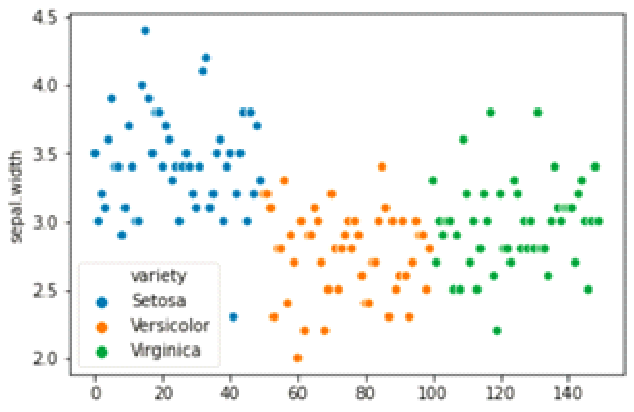
```
plt.scatter(df.index,df['sepal.width'])
plt.show()
```

Use the *plt.scatter()* function of matplotlib to plot a univariate scatter diagram. The scatter() function requires two parameters to plot. So, in this example, we plot the variable ‘sepal.width’ against the corresponding observation number that is stored as the index of the data frame (df.index).

Then visualize the same plot by considering its variety using the *sns.scatterplot()* function of the seaborn library.



```
sns.scatterplot(x=df.index,y=df['sepal.width'],hue=df['variety'])
```



One of the interesting features in seaborn is the ‘hue’ parameter. In seaborn, the hue parameter determines which column in the data frame should be used for color encoding. This helps to differentiate between the data values according to the categories they belong to. The hue parameter takes the grouping variable as it’s input using which it will produce points with different colors. The variable passed onto ‘hue’ can be either categorical or numeric, although color mapping will behave differently in the latter case.

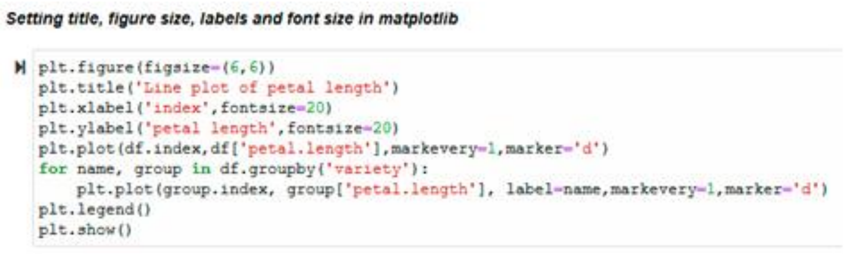
2. LINE PLOT (with markers) :

A line plot visualizes data by connecting the data points via line segments. It is similar to a scatter plot

except

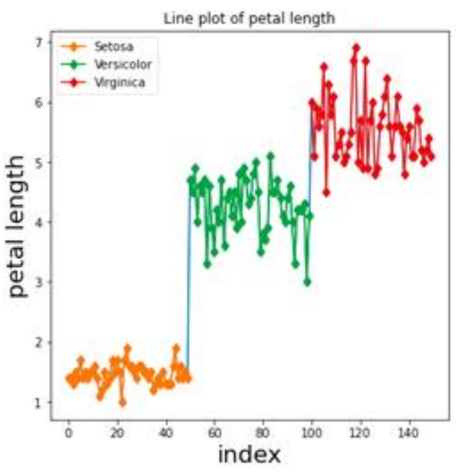
that the measurement points are ordered (typically by their x-axis value) and joined with straight line

segments.



Setting title, figure size, labels and font size in matplotlib

```
plt.figure(figsize=(6,6))
plt.title('Line plot of petal length')
plt.xlabel('index',fontsize=20)
plt.ylabel('petal length',fontsize=20)
plt.plot(df.index,df['petal.length'],markevery=1,marker='d')
for name, group in df.groupby('variety'):
    plt.plot(group.index, group['petal.length'], label=name,markevery=1,marker='d')
plt.legend()
plt.show()
```



The matplotlib *plt.plot()* function by default plots the data using a line plot.

Previously, we discussed the hue parameter of seaborn. Though there is no such automated option available in matplotlib, one can use the *groupby()* function of pandas which helps in plotting such a graph.

Note:In the above illustration, the methods to set title, font size, etc in matplotlib are also implemented.

— Explanation of the functions used :

- *plt.figure(figsize=())* : To set the size of figure
- *plt.title()* : To set title
- *plt.xlabel() / plt.ylabel()* : To set labels on X-axis/Y-axis
- *df.groupby( )* : To group the rows of the data frame according to the parameter passed onto the function
- The groupby() function returns the data frames grouped by the criterion variable passed and the criterion variable.
- The *for loop* is used to plot each data point according to its variety.
- *plt.legend()*: Adds a legend to the graph (Legend describes the different elements seen in the graph).
- *plt.show()* : to show the plot.

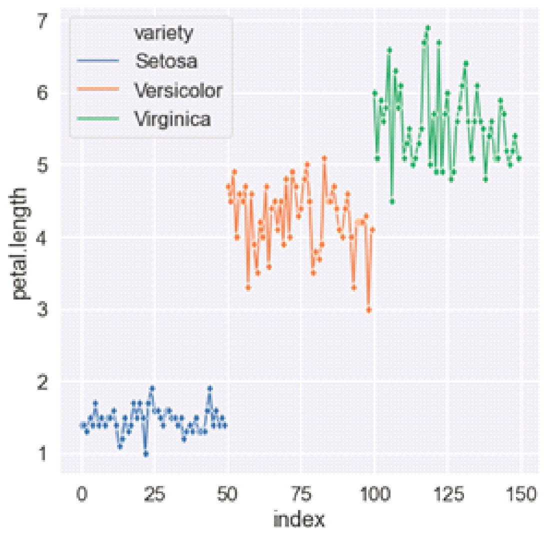
The 'markevery' parameter of the function *plt.plot()* is assigned to '1' which means it will plot every 1st marker starting from the first data point. There are various marker styles which we can pass as a parameter to the function.

The *sns.lineplot()* function can also visualize the line plot.

Setting title, figure size,labels and font size in seaborn

```
sns.set(rc={'figure.figsize':(7,7)})
sns.set(font_scale=1.5)
```

```
fig=sns.lineplot(x=df.index,y=df['petal.length'],markevery=1,marker='d',data=df,hue=df['variety'])
fig.set(xlabel='index')
```



In seaborn, the labels on axes are automatically set based on the columns that are passed for plotting. However if one desires to change it, it is possible too using the set() function.

3. STRIP PLOT :

The strip plot is similar to a scatter plot. It is often used along with other kinds of plots for better analysis. It is used to visualize the distribution of data points of the variable.

The *sns.striplot ( )* function is used to plot a strip-plot :

```
sns.stripplot(y=df['sepal.width'])
```

It also helps to plot the distribution of variables for each category as individual data points. By default, the function creates a vertical strip plot where the distributions of the continuous data points are plotted along the Y-axis and the categories are spaced out along the X-axis. In the above plot, categories are not considered. Considering the categories helps in better visualization as seen in the below plot.

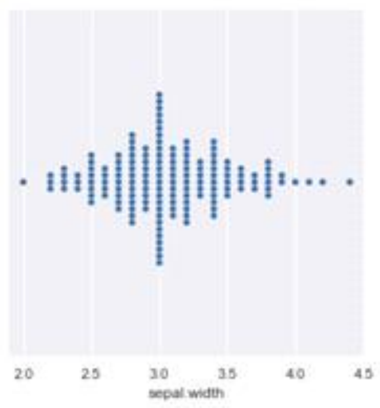
```
sns.stripplot(x=df['variety'],y=df['sepal.width'])
```

4. SWARM PLOT :

The swarm-plot, similar to a strip-plot, provides a visualization technique for univariate data to view the spread of values in a continuous variable. The only difference between the strip-plot and the swarm-plot is that the swarm-plot spreads out the data points of the variable automatically to avoid overlap and hence provides a better visual overview of the data.

The *sns.swarmplot( )* function is used to plot a swarm-plot :

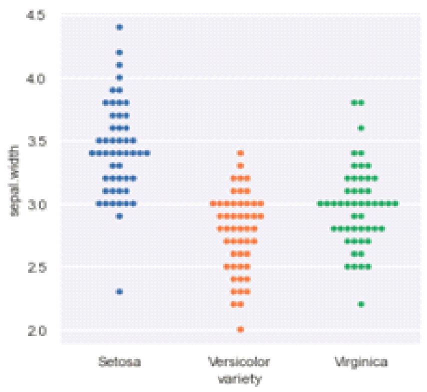
```
sns.set(rc={'figure.figsize':(5,5)})
sns.swarmplot(x=df['sepal.width'])
```



Distribution of the variable 'sepal.width' according to the categories :

Distribution of the variable 'sepal.width' according to the categories :

```
sns.swarmplot(x=df['variety'],y=df['sepal.width'])
```



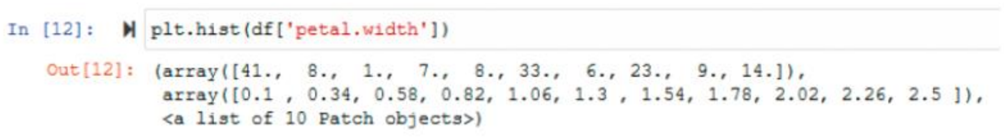
Uni-variate summary plots :

These plots give a more concise description of the location, dispersion, and distribution of a variable than an enumerative plot. It is not feasible to retrieve every individual data value in a summary plot, but it helps in efficiently representing the whole data from which better conclusions can be made on the entire data set.

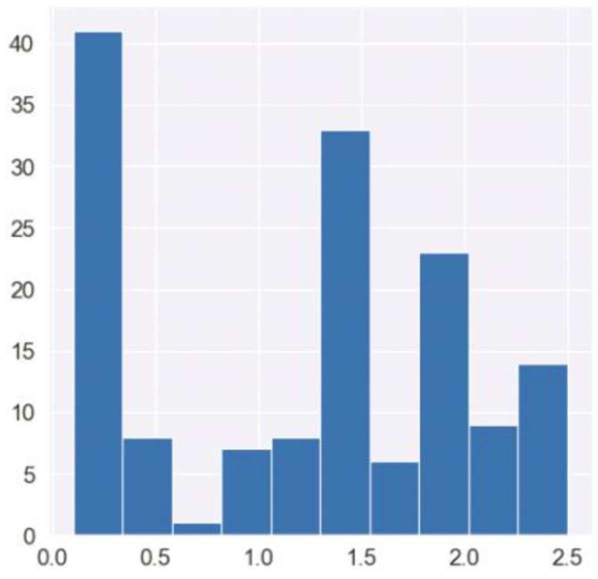
5. HISTOGRAMS :

Histograms are similar to bar charts which display the counts or relative frequencies of values falling in different class intervals or ranges. A histogram displays the shape and spread of continuous sample data. It also helps us understand the skewness and kurtosis of the distribution of the data.

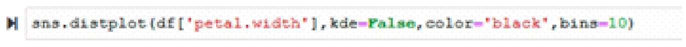
Plotting histogram using the matplotlib *plt.hist()* function :



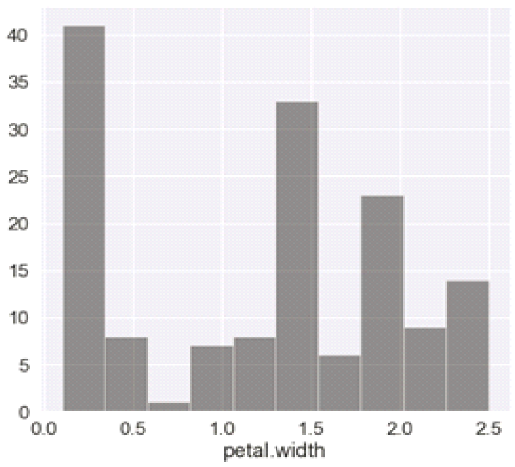
```
In [12]:  plt.hist(df['petal.width'])

Out[12]: (array([41.,  8.,  1.,  7.,  8., 33.,  6., 23.,  9., 14.]),
          array([0.1 , 0.34, 0.58, 0.82, 1.06, 1.3 , 1.54, 1.78, 2.02, 2.26, 2.5 ]),
          <a list of 10 Patch objects>)
```



The seaborn function *sns.distplot()* can also be used to plot a histogram.



```
sns.distplot(df['petal.width'],kde=False,color='black',bins=10)
```

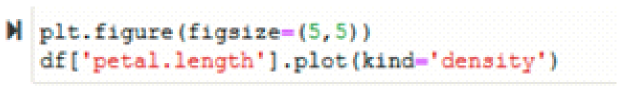


The kde (kernel density) parameter is set to False so that only the histogram is viewed. There are many parameters like bins (indicating the number of bins in histogram allowed in the plot), color, etc; which can be set to obtain the desired output.

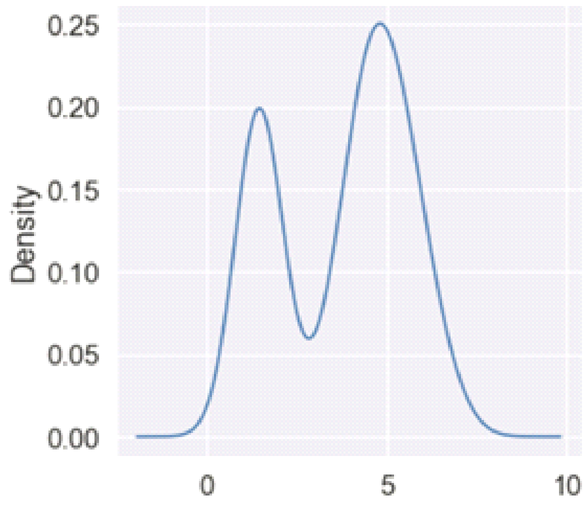
6. DENSITY PLOTS :

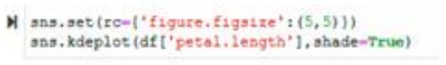
A density plot is like a smoother version of a histogram. Generally, the kernel density estimate is used in density plots to show the probability density function of the variable. A continuous curve, which is the kernel is drawn to generate a smooth density estimation for the whole data.

Plotting density plot of the variable 'petal.length' :

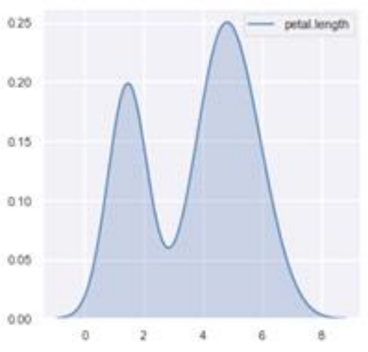


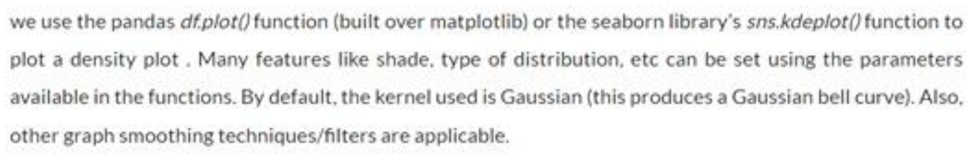
```python
plt.figure(figsize=(5,5))
df['petal.length'].plot(kind='density')
```





```python
sns.set(rc={'figure.figsize':(5,5)})
sns.kdeplot(df['petal.length'],shade=True)
```

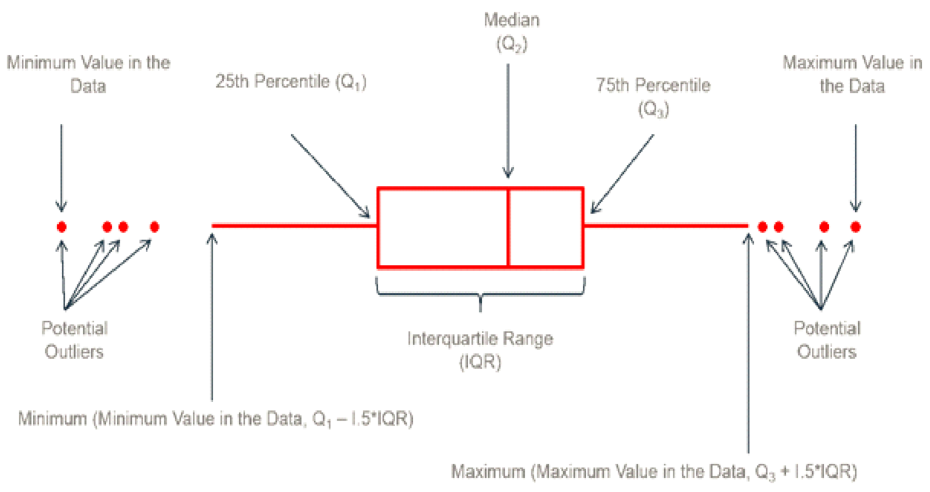




we use the pandas *df.plot()* function (built over matplotlib) or the seaborn library's *sns.kdeplot()* function to plot a density plot . Many features like shade, type of distribution, etc can be set using the parameters available in the functions. By default, the kernel used is Gaussian (this produces a Gaussian bell curve). Also, other graph smoothing techniques/filters are applicable.

8. BOX PLOTS :

A box-plot is a very useful and standardized way of displaying the distribution of data based on a five-number summary (minimum, first quartile, second quartile(median), third quartile, maximum). It helps in understanding these parameters of the distribution of data and is extremely helpful in detecting outliers.


Median (Q_2)
Minimum Value in the Data
25th Percentile (Q_1)
75th Percentile (Q_3)
Maximum Value in the Data
Potential Outliers
Interquartile Range (IQR)
Potential Outliers
Minimum (Minimum Value in the Data, $Q_1 - 1.5$*IQR)
Maximum (Maximum Value in the Data, $Q_3 + 1.5$*IQR)

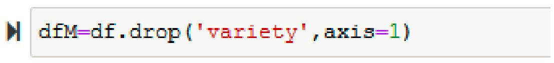
Plotting box plot of variable 'sepal.width' : **plt.boxlplot(df.['Sepal_width'])**

Plotting box plots of all variables in one frame :

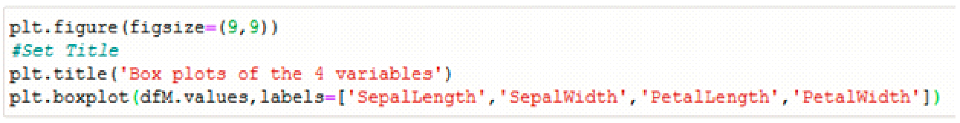
Since the box plot is for continuous variables, firstly create a data frame without the column 'variety'.

Then drop the column from the DataFrame using the *drop( )* function and specify axis=1 to indicate it.

Removing the column with categorical variables



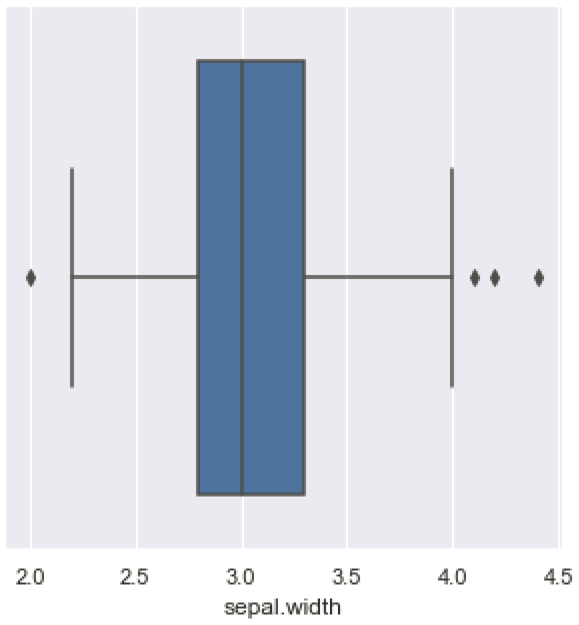
```
dfM=df.drop('variety',axis=1)
```



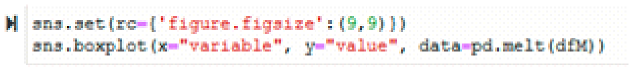
```
plt.figure(figsize=(9,9))
#Set Title
plt.title('Box plots of the 4 variables')
plt.boxplot(dfM.values,labels=['SepalLength','SepalWidth','PetalLength','PetalWidth'])
```

In matplotlib, mention the labels separately to display it in the output.

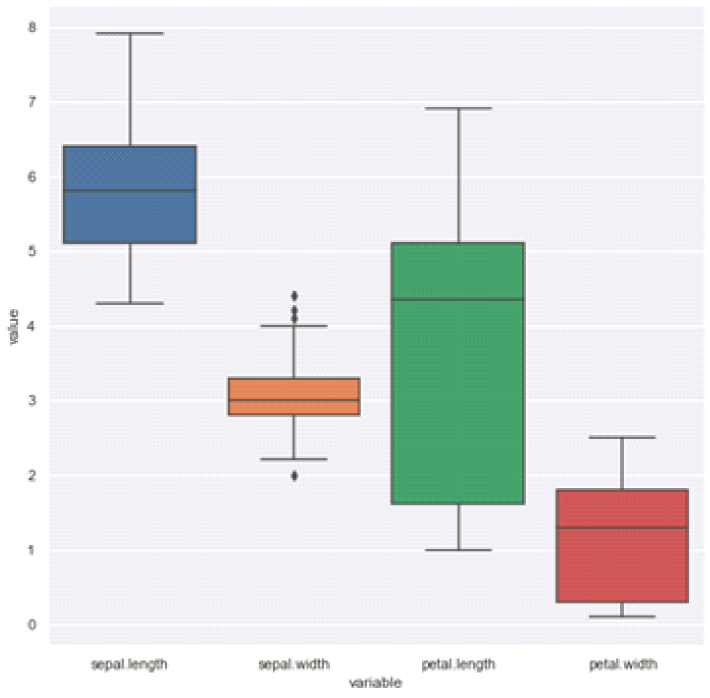
The plotting box plot in seaborn : **sns.boxplot(['Sepal_width'])**



Plotting the box plots of all variables in one frame :



```
sns.set(rc={'figure.figsize':(9,9)})
sns.boxplot(x="variable", y="value", data=pd.melt(dfM))
```



VISUALIZING CATEGORICAL VARIABLES :

11. BAR CHART :

The bar plot is a univariate data visualization plot on a two-dimensional axis. One axis is the category axis indicating the category, while the second axis is the value axis that shows the numeric value of that category, indicated by the length of the bar.

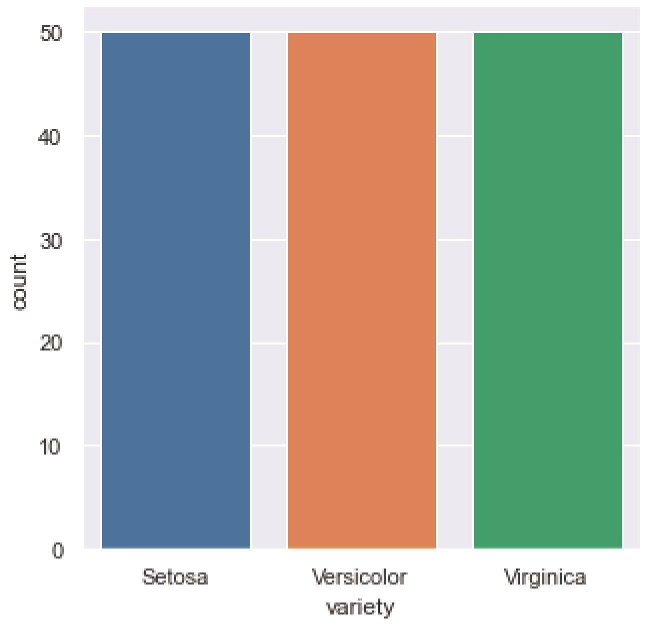
The *plot.bar()* function plots a bar plot of a categorical variable. The *value_counts()* returns a series containing the counts of unique values in the variable.

```
df['variety'].value_counts().plot.bar()
```

The *countplot()* function of the seaborn library obtains a similar bar plot. There is no need to separately calculate the count when using the *sns.countplot()* function.

Since the variety is equally distributed, we obtain bars with equal heights.

```
sns.countplot(df['variety'])
```

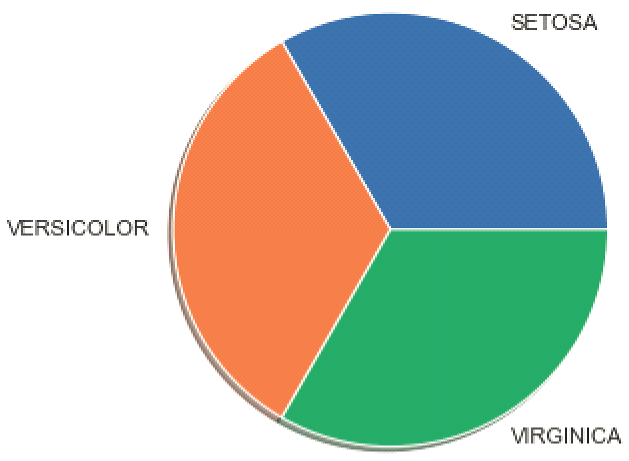


12. PIE CHART :

A pie chart is the most common way used to visualize the numerical proportion occupied by each of the categories.

Use the *plt.pie()* function to plot a pie chart. Since the categories are equally distributed, divide the sections in the pie chart is equally. Then add the labels by passing the array of values to the 'labels' parameter.

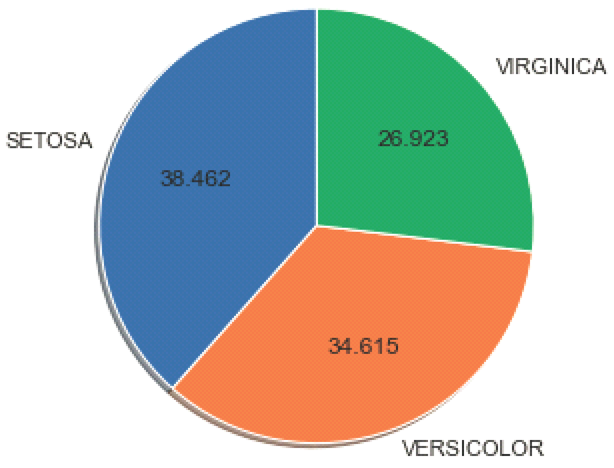
```
plt.pie(df['variety'].value_counts(),
        labels=['SETOSA','VERSICOLOR','VIRGINICA'],shadow=True)
```



The ‘startangle’ parameter of the *pie()* function rotates everything counter-clockwise at a specific angle. Further, the default value for startangle is 0. The ‘autopct’ parameter enables one to display the percentage value using Python string formatting.

```
df1=df.sample(frac=0.35)
```

```
plt.figure(figsize=(5,5))
plt.pie(df1['variety'].value_counts(),startangle=90,autopct='%.3f',
        labels=['SETOSA','VERSICOLOR','VIRGINICA'],shadow=True)
```



Most of the methods that help in the visualization of univariate data have been outlined in this article. As stated before the ability to see the structure and information carried by the data lies in its visual presentation.