Data Visualization 5

Data visualization is the initial move in the data analysis system toward easily understanding and communicating information. It represents information and data in graphical form using visual elements such as charts, graphs, plots, and maps. It helps analysts to understand patterns, trends, outliers, distributions, and relationships. Data visualization is an efficient way to deal with a large number of datasets.

Python offers various libraries for data visualization, such as Matplotlib, Seaborn, and Bokeh. In this chapter, we will first focus on Matplotlib, which is the basic Python library for visualization. After Matplotlib, we will explore Seaborn, which uses Matplotlib and offers high-level and advanced statistical plots. In the end, we will work on interactive data visualization using Bokeh. We will also explore `pandas` plotting. The following is a list of topics that will be covered in this chapter:

- Visualization using Matplotlib
- Advanced visualization using the Seaborn package
- Interactive visualization with Bokeh

Visualization using Matplotlib

As we know, a picture speaks a thousand words. Humans understand visual things better. Visualization helps to present things to any kind of audience and can easily explain a complex phenomenon in layman's terms. Python offers a couple of visualization libraries, such as Matplotlib, Seaborn, and Bokeh.

Matplotlib is the most popular Python module for data visualization. It is a base library for most of the advanced Python visualization modules, such as Seaborn. It offers flexible and easy-to-use built-in functions for creating figures and graphs.

In Anaconda, Matplotlib is already installed. If you still find an error, you can install it in the following ways.

We can install Matplotlib with `pip` as follows:

```
pip install matplotlib
```

For Python 3, we can use the following command:

```
pip3 install matplotlib
```

You can also simply install Matplotlib from your terminal or Command Prompt using the following command: `conda install matplotlib`

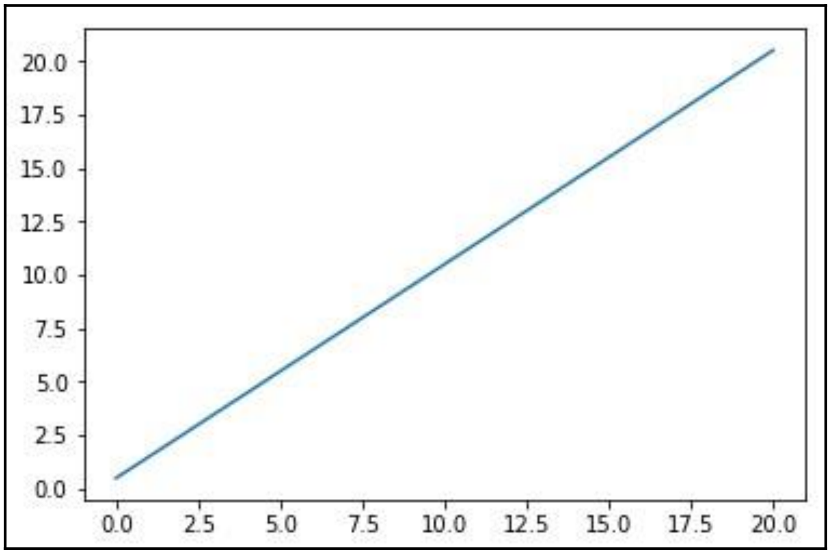
To create a very basic plot in Matplotlib, we need to invoke the `plot()` function in the `matplotlib.pyplot` subpackage. This function produces a two-dimensional plot for a single list or multiple lists of points with known *x* and *y* coordinates.

Let's see a small demo code for visualizing the line plot:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt import
numpy as np

# create the data a =
np.linspace(0, 20)

# Draw the plot
plt.plot(a, a + 0.5, label='linear')
```

```
# Display the chart
plt.show()
```

This results in the following output:



In the preceding code block, first, we are importing the Matplotlib and NumPy modules. After this, we are creating the data using the `linespace()` function of NumPy and plotting this data using the `plot()` function of Matplotlib. Finally, we are displaying the figure using the `show()` function.

There are two basic components of a plot: the figure and the axes. The figure is a container on which everything is drawn. It contains components such as plots, subplots, axes, titles, and a legend. In the next section, we will focus on these components, which act like accessories for charts.

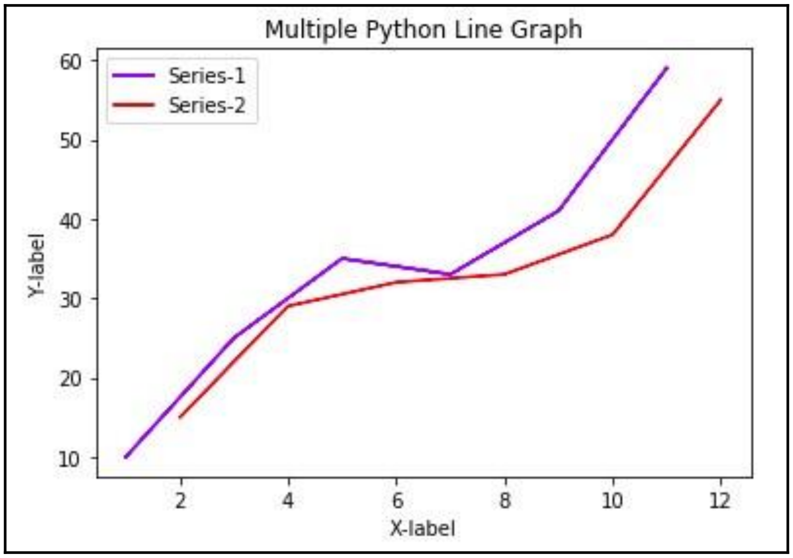
Accessories for charts

In the `matplotlib` module, we can add titles and axes labels to a graph. We can add a title using `plt.title()` and labels using `plt.xlabel()` and `plt.ylabel()`.

Multiple graphs mean multiple objects, such as line, bar, and scatter. Points of different series can be shown on a single graph. Legends or graph series reflect the *y* axis. A legend is a box that appears on either the right or left side of a graph and shows what each element of the graph represents. Let's see an example where we see how to use these accessories in our charts:

```
# Add the required libraries
import matplotlib.pyplot as plt

# Create the data x =
[1,3,5,7,9,11] y =
[10,25,35,33,41,59]

# Let's plot the data
plt.plot(x, y,label='Series-1', color='blue')

# Create the data x =
[2,4,6,8,10,12] y =
[15,29,32,33,38,55]

# Plot the data
plt.plot(x, y, label='Series-2', color='red')

# Add X Label on X-axis plt.xlabel("X-
label")

# Add X Label on X-axis plt.ylabel("Y-
label")

# Append the title to graph
plt.title("Multiple Python Line Graph")

# Add legend to graph
plt.legend()

# Display the plot
plt.show()
```

This results in the following output:



In the preceding graph, two lines are shown on a single graph. We have used two extra parameters – `label` and `color` – in the `plot()` function. The `label` parameter defines the name of the series and `color` defines the color of the line graph. In the upcoming sections, we will focus on different types of plots. We will explore a scatter plot in the next section.

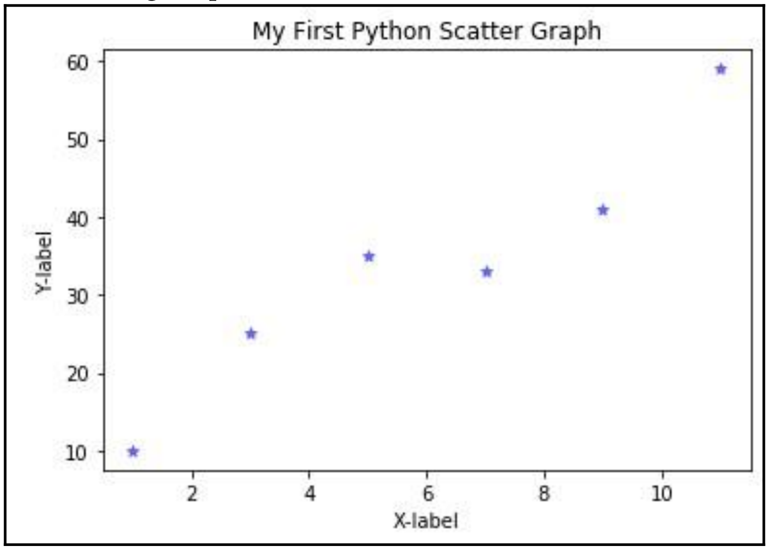
Scatter plot

Scatter plots draw data points using Cartesian coordinates to show the values of numerical values. They also represent the relationship between two numerial values. We can create a scatter plot in Matplotlib using the `scatter()` function, as follows:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data x =
[1,3,5,7,9,11] y =
[10,25,35,33,41,59]

# Draw the scatter chart
plt.scatter(x, y, c='blue', marker='*',alpha=0.5)

# Append the label on X-axis plt.xlabel("X-
label")
```

```
# Append the label on X-axis plt.ylabel("Y-
label")

# Add the title to graph
plt.title("Scatter Chart Sample")

# Display the chart
plt.show()
```

This results in the following output:



In the preceding scatter plot, the `scatter()` function takes x-axis and y-axis values. In our example, we are plotting two lists: `x` and `y`. We can also use optional parameters such as `c` for color, `alpha` for the transparency of the markers, ranging between 0 and 1, and `marker` for the shape of the points in the scatter plot, such as `*`, `o`, or any other symbol. In the next section, we will focus on the line plot.

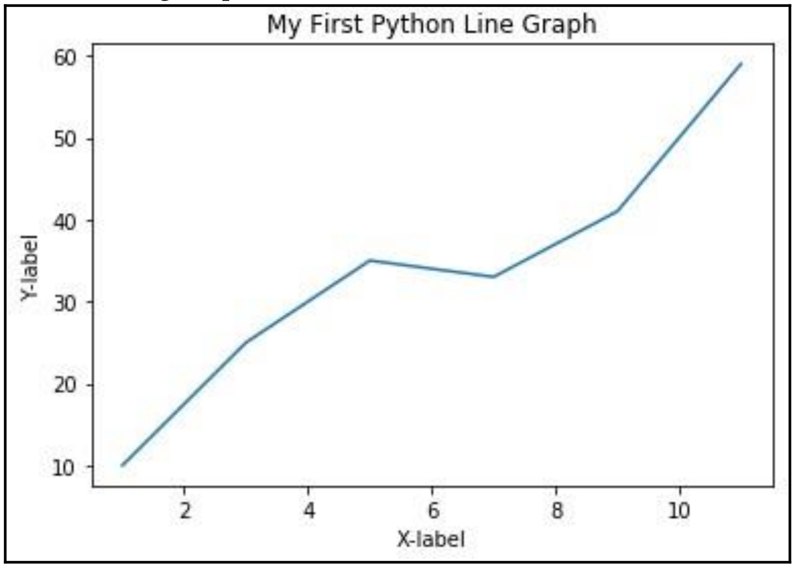
Line plot

A line plot is a chart that displays a line between two variables. It has a sequence of data points joined by a segment:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt
```

```
# create the data x =
[1,3,5,7,9,11] y =
[10,25,35,33,41,59]

# Draw the line chart
plt.plot(x, y)

# Append the label on X-axis plt.xlabel("X-
label")

# Append the label on X-axis plt.ylabel("Y-
label")

# Append the title to chart
plt.title("Line Chart Sample")

# Display the chart
plt.show()
```

This results in the following output:



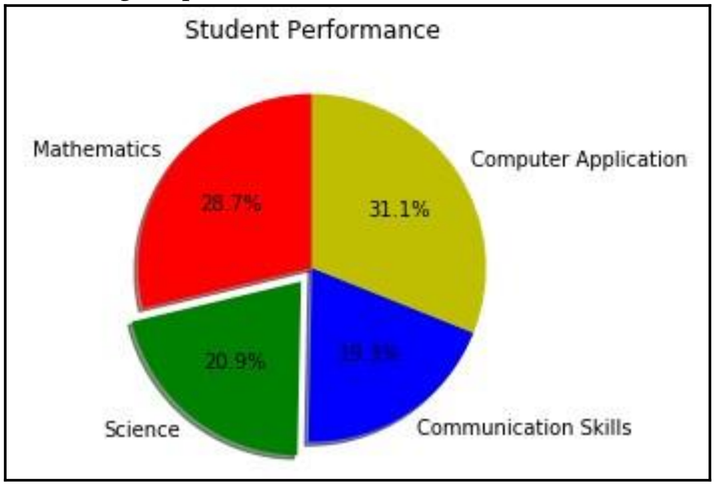
In the preceding line plot program, the `plot()` function takes x-axis and y-axis values. In the next section, we will learn how to plot a pie chart.

Pie plot

A pie plot is a circular graph that is split up into wedge-shaped pieces. Each piece is proportionate to the value it represents. The total value of the pie is 100 percent:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data
subjects = ["Mathematics","Science","Communication Skills","Computer
Application"] scores =
[85,62,57,92]

# Plot the pie plot
plt.pie(scores,
labels=subjects,
colors=['r','g','b','y'],
        startangle=90,
shadow= True,
explode=(0,0.1,0,0),
autopct='%1.1f%%')

# Add title to graph
plt.title("Student Performance")

# Draw the chart
plt.show()
```

This results in the following output:



In the preceding code of the pie chart, we specified `values`, `labels`, `colors`, `startangle`, `shadow`, `explode`, and `autopct`. In our example, `values` is the scores of the student in four subjects and `labels` is the list of subject names. We can also specify the color list for the individual subject scores. The `startangle` parameter specifies the first value angle, which is 90 degrees; this means the first line is vertical.

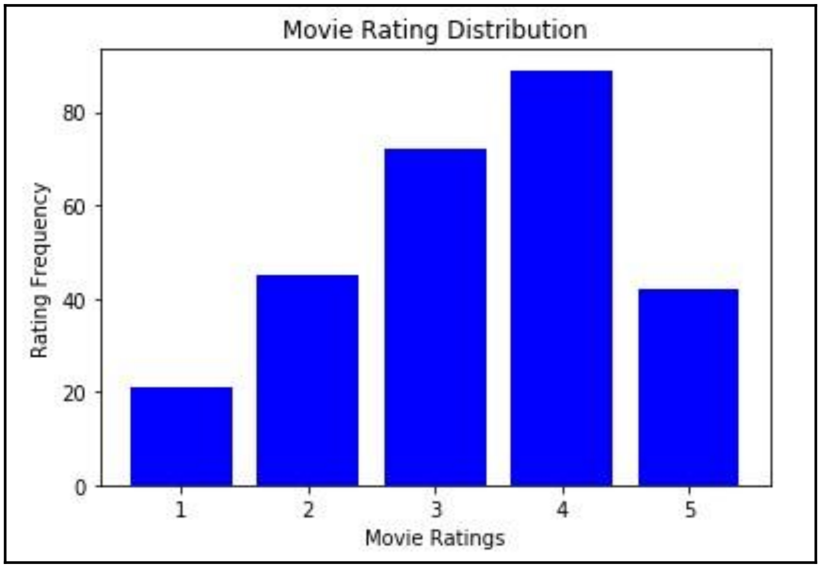
Optionally, we can also use the `shadow` parameter to specify the shadow of the pie slice and the `explode` parameter to pull out a pie slice list of the binary value. If we want to pull out a second pie slice, then a tuple of values would be (0, 0.1, 0, 0). Let's now jump to the bar plot.

Bar plot

A bar plot is a visual tool to compare the values of various groups. It can be drawn horizontally or vertically. We can create a bar graph using the `bar()` function:

```
# Add the essential library matplotlib
import matplotlib.pyplot as plt

# create the data movie_ratings
= [1,2,3,4,5] rating_counts =
[21,45,72,89,42]

# Plot the data
plt.bar(movie_ratings, rating_counts, color='blue')

# Add X Label on X-axis
plt.xlabel("Movie Ratings")

# Add X Label on X-axis
plt.ylabel("Rating Frequency")

# Add a title to graph
plt.title("Movie Rating Distribution")

# Show the plot
plt.show()
```

This results in the following output:



In the preceding bar chart program, the `bar()` function takes *x*-axis values, *y*-axis values, and a color. In our example, we are plotting movie ratings and their frequency. Movie ratings are on the *x* axis and the rating frequency is on the *y* axis. We can also specify the color of the bars in the bar graph using the `color` parameter. Let's see another variant of bar plot in the next subsection.

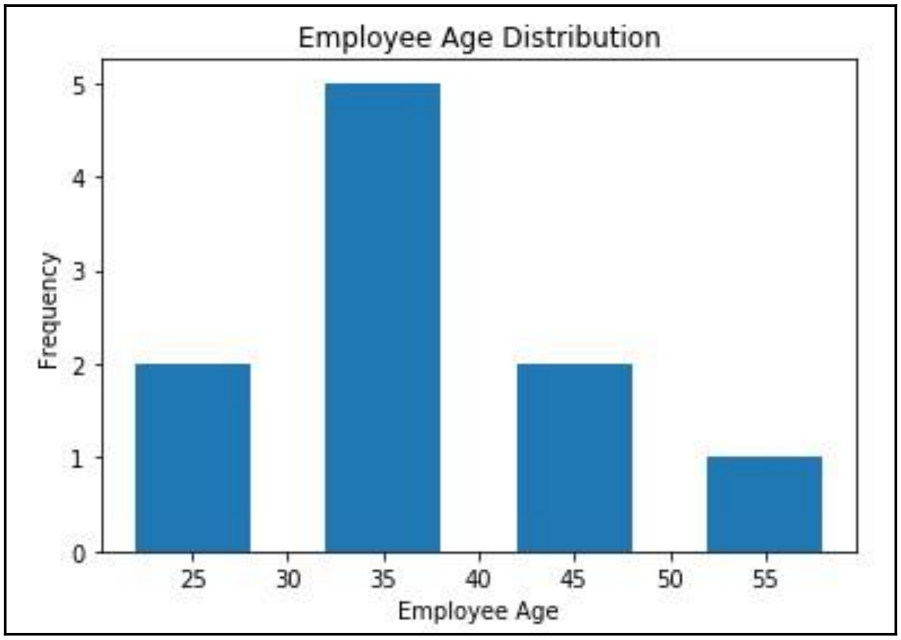
Histogram plot

A histogram shows the distribution of a numeric variable. We create a histogram using the `hist()` method. It shows the probability distribution of a continuous variable. A histogram only works on a single variable while a bar graph works on two variables:

```
# Add the essential library
import matplotlib.pyplot as plt

# Create the data
employee_age = [21,28,32,34,35,35,37,42,47,55]

# Create bins for histogram
bins = [20,30,40,50,60]

# Plot the histogram
```

```
plt.hist(employee_age, bins, rwidth=0.6)

# Add X Label on X-axis
plt.xlabel("Employee Age")

# Add X Label on X-axis
plt.ylabel("Frequency")

# Add title to graph
plt.title("Employee Age Distribution")

# Show the plot
plt.show()
```

This results in the following output:



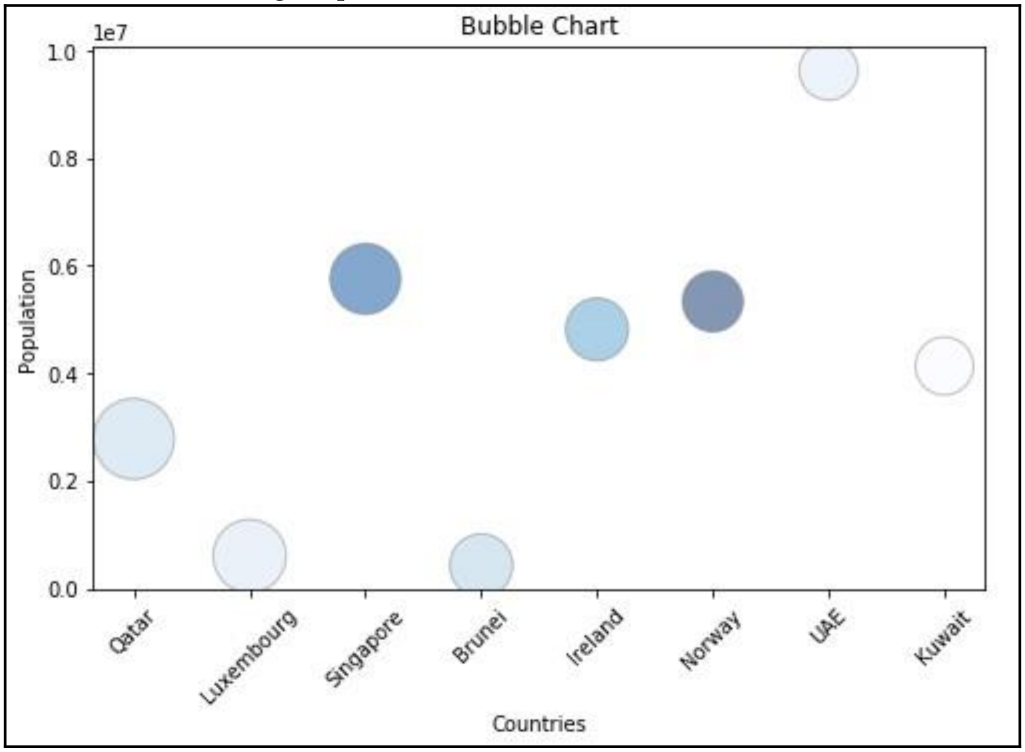
In the preceding histogram, the `hist()` function takes `values`, `bins`, and `rwidth`. In our example, we are plotting the age of the employee and using a bin of 10 years. We are starting our bin from 20 to 60 with a 10 years bin size. We are using a relative bar width of 0.6, but you can choose any size for thicker and thinner width. Now it's time to jump to the bubble plot, which can handle multiple variables in a two-dimensional plot.

Bubble plot

A bubble plot is a type of scatter plot. It not only draws data points using Cartesian coordinates but also creates bubbles on data points. Bubble shows the third dimension of a plot. It shows three numerical values: two values are on the *x* and *y* axes and the third one is the size of data points (or bubbles):

```
# Import the required modules
import matplotlib.pyplot as plt
import numpy as np

# Set figure size
plt.figure(figsize=(8,5))

# Create the data
countries =
['Qatar','Luxembourg','Singapore','Brunei','Ireland','Norway','UAE','Kuwait
']
populations = [2781682,
604245,5757499,428963,4818690,5337962,9630959,4137312]
gdp_per_capita = [130475, 106705, 100345, 79530, 78785, 74356,69382, 67000]

# scale GDP per capita income to shoot the bubbles in the graph

scaled_gdp_per_capita = np.divide(gdp_per_capita, 80) colors =

np.random.rand(8)

# Draw the scatter diagram
plt.scatter(countries, populations, s=scaled_gdp_per_capita, c=colors,
cmap="Blues",edgecolors="grey", alpha=0.5)

# Add X Label on X-axis
plt.xlabel("Countries")

# Add Y Label on X-axis
plt.ylabel("Population")

# Add title to graph
plt.title("Bubble Chart")

# rotate x label for clear visualization
plt.xticks(rotation=45)

# Show the plot
plt.show()
```

This results in the following output:



In the preceding plot, a bubble chart is created using the scatter function. Here, the important thing is the `s` (size) parameter of the scatter function. We assigned a third variable, `scaled_gdp_per_capita`, to the `size` parameters. In the preceding bubble plot, countries are on the *x* axis, the population is on the *y* axis, and GDP per capita is shown by the size of the scatter point or bubble. We also assigned a random color to the bubbles to make it attractive and more understandable. From the bubble size, you can easily see that Qatar has the highest GDP per capita and Kuwait has the lowest GDP per capita. In all the preceding sections, we have focused on most of the Matplotlib plots and charts. Now, we will see how we can plot the charts using the `pandas` module.

pandas plotting

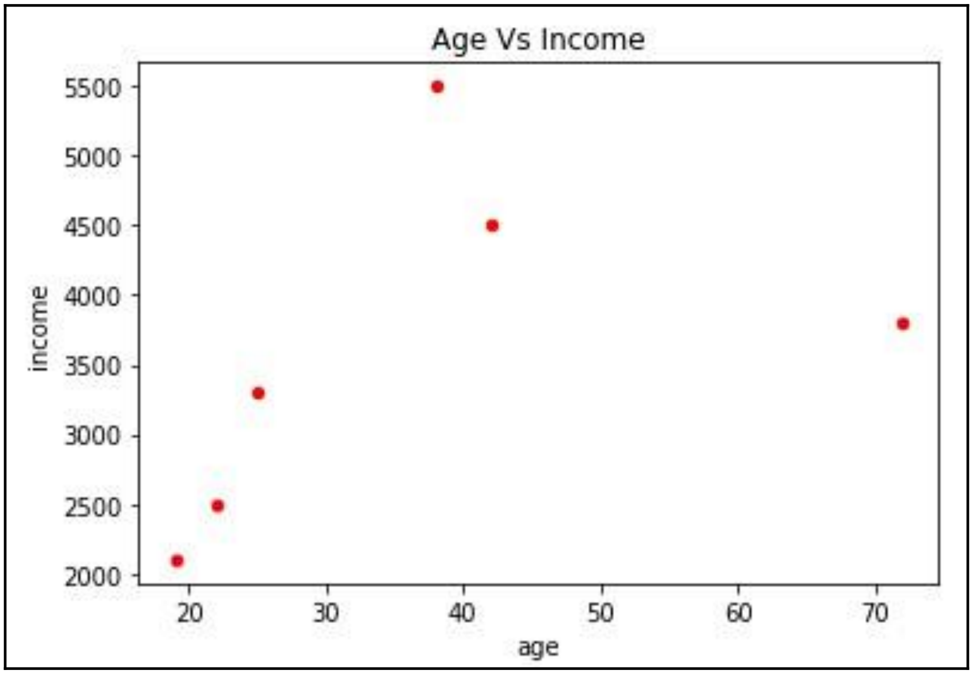
The `pandas` library offers the `plot()` method as a wrapper around the Matplotlib library. The `plot()` method allows us to create plots directly on `pandas` DataFrames. The following `plot()` method parameters are used to create the plots:

- `kind`: A string parameter for the type of graph, such as line, bar, barh, hist, box, KDE, pie, area, or scatter.
- `figsize`: This defines the size for a figure in a tuple of (width,
- height). `title`: This defines the title for the graph. `grid`: Boolean
- parameter for the axis grid line.
- `legend`: This defines the legend. `xticks`: This
- defines the sequence of x-axis ticks.
- `yticks`: This defines the sequence of y-axis ticks.

Let's create a scatter plot using the `pandas plot()` function:

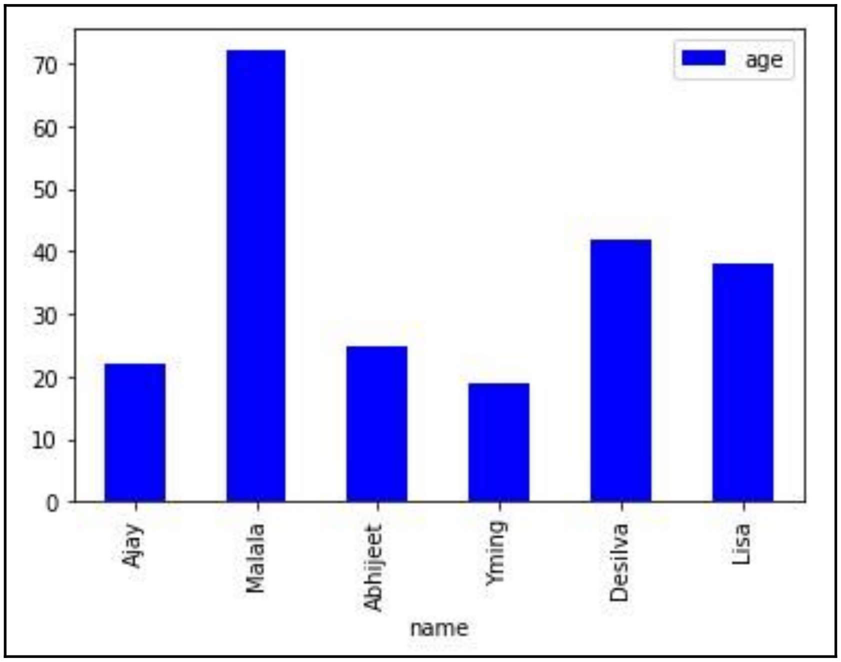
```
# Import the required modules
import pandas as pd import
matplotlib.pyplot as plt

# Let's create a Dataframe
df = pd.DataFrame({
            'name':['Ajay','Malala','Abhijeet','Yming','Desilva','Lisa'],
            'age':[22,72,25,19,42,38],
            'gender':['M','F','M','M','M','F'],
'country':['India','Pakistan','Bangladesh','China','Srilanka','UK'],
            'income':[2500,3800,3300,2100,4500,5500]
        })

# Create a scatter plot
df.plot(kind='scatter', x='age', y='income', color='red', title='Age Vs
Income')

# Show figure
plt.show()
```

This results in the following output:



In the preceding plot, the `plot()` function takes `kind`, `x`, `y`, `color`, and `title` values. In our example, we are plotting the scatter plot between age and income using the `kind` parameter as `'scatter'`. The `age` and `income` columns are assigned to the `x` and `y` parameters. The scatter point color and the title of the plot are assigned to the `color` and `title` parameters:

```
import matplotlib.pyplot as plt
import pandas as pd

# Create bar plot
df.plot(kind='bar',x='name', y='age', color='blue')

# Show figure
plt.show()
```



In the preceding plot, the `plot()` function takes `kind`, `x`, `y`, `color`, and `title` values. In our example, we are plotting the bar plot between age and income using the `kind` parameter as `'bar'`. The `name` and `age` columns are assigned to the `x` and `y` parameters. The scatter point color is assigned to the `color` parameter. This is all about `pandas` plotting. Now, from the next section onward, we will see how to visualize the data using the Seaborn library.

Advanced visualization using the Seaborn package

Visualization can be helpful to easily understand complex patterns and concepts. It represents the insights in pictorial format. In the preceding sections, we have learned about Matplotlib for visualization. Now, we will explore the new Seaborn library for highlevel and advanced statistical plots. Seaborn is an open source Python library for high-level interactive and attractive statistical visualization. Seaborn uses Matplotlib as a base library and offers more simple, easy-to-understand, interactive, and attractive visualizations.

This results in the following output:
In the Anaconda software suite, you can install the Seaborn library in the following way:

Install Seaborn with `pip`:

```
pip install seaborn
```

For Python 3, use the following command:

```
pip3 install seaborn
```

You can simply install Seaborn from your terminal or Command Prompt using the following: `conda install seaborn`

If you are installing it into the Jupyter Notebook, then you need to put the `!` sign before the `pip` command. Here is an example:

```
!pip install seaborn
```

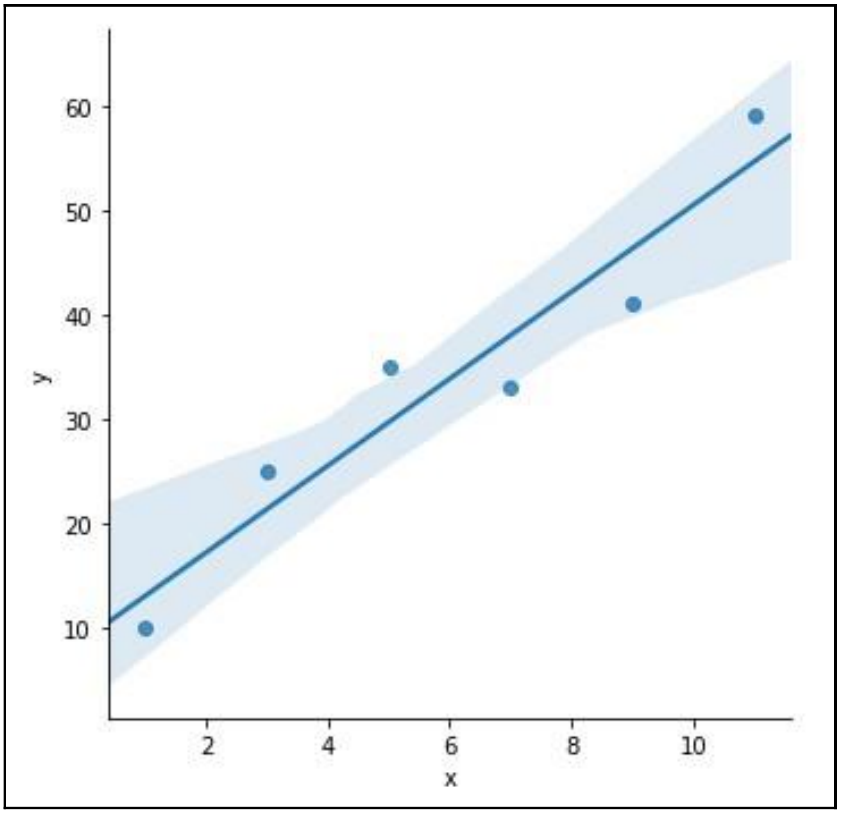
Let's jump to the `lm` plot of Seaborn.

lm plots

The `lm` plot plots the scatter and fits the regression model on it. A scatter plot is the best way to understand the relationship between two variables. Its output visualization is a joint distribution of two variables. `lmplot()` takes two column names – `x` and `y` – as a string and DataFrame variable. Let's see the following example:

```
# Import the required libraries
import pandas as pd import
seaborn as sns import
matplotlib.pyplot as plt

# Create DataFrame
df=pd.DataFrame({'x':[1,3,5,7,9,11],'y':[10,25,35,33,41,59]})

# Create lmplot
sns.lmplot(x='x', y='y', data=df)
```

```
# Show figure
plt.show()
```

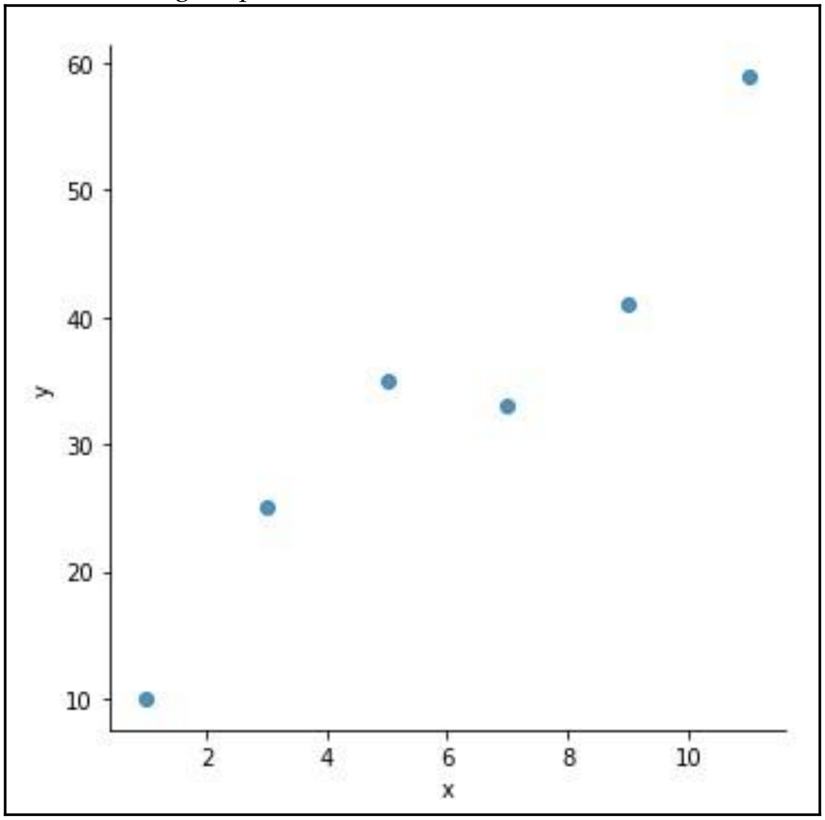


By default, `lmplot()` fits the regression line. We can also remove this by setting the `fit_reg` parameter as `False`:

```
# Create lmplot
sns.lmplot(x='x', y='y', data=df, fit_reg=False)

# Show figure
plt.show()
```

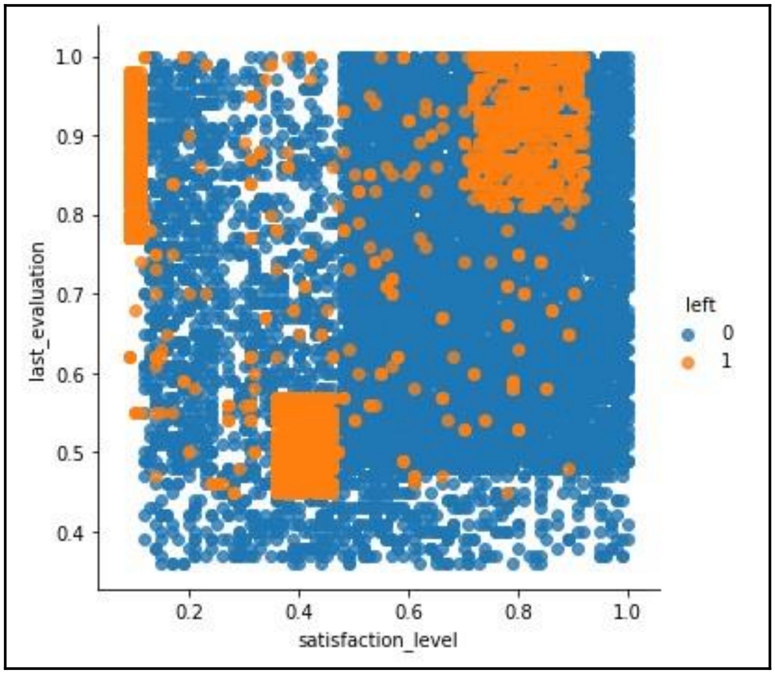
This results in the following output:

This results in the following output:



Let's take a dataset of HR Analytics and try to plot `lmplot()`:

```
# Load the dataset
df=pd.read_csv("HR_comma_sep.csv")

# Create lmplot
sns.lmplot(x='satisfaction_level', y='last_evaluation', data=df,
fit_reg=False, hue='left')

# Show figure
plt.show()
```



In the preceding example, `last_evaluation` is the evaluated performance of the employee, `satisfaction_level` is the employee's satisfaction level in the company, and `left` means whether the employee left the company or not. `satisfaction_level` and `last_evaluation` were drawn on the *x* and *y* axes, respectively. The third variable left is passed in the `hue` parameter. The `hue` property is used for color shade. We are passing a `left` variable as `hue`. We can clearly see in the diagram that employees that have left are scattered into three groups. Let's now jump to bar plots.

Bar plots

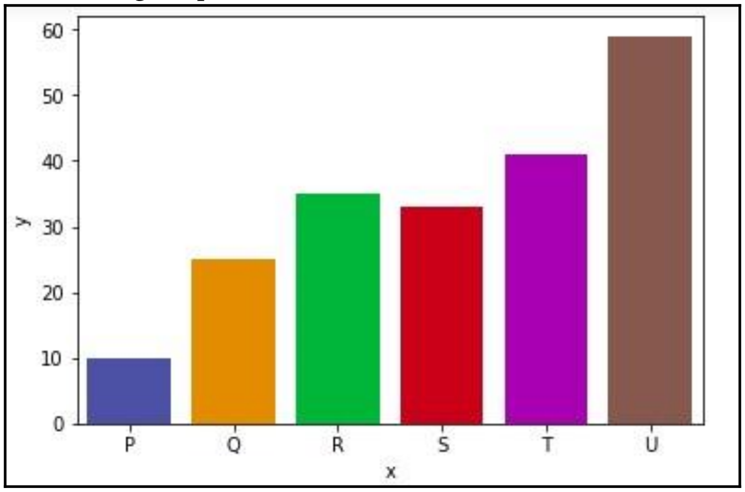
`barplot()` offers the relationship between a categorical and a continuous variable. It uses rectangular bars with variable lengths:

```
# Import the required libraries
import pandas as pd import
seaborn as sns import
matplotlib.pyplot as plt #
Create DataFrame
df=pd.DataFrame({'x':['P','Q','R','S','T','U'],'y':[10,25,35,33,41,59]})
```

This results in the following output:

```
# Create lmplot
sns.barplot(x='x', y='y', data=df)

# Show figure
plt.show()
```

This results in the following output:

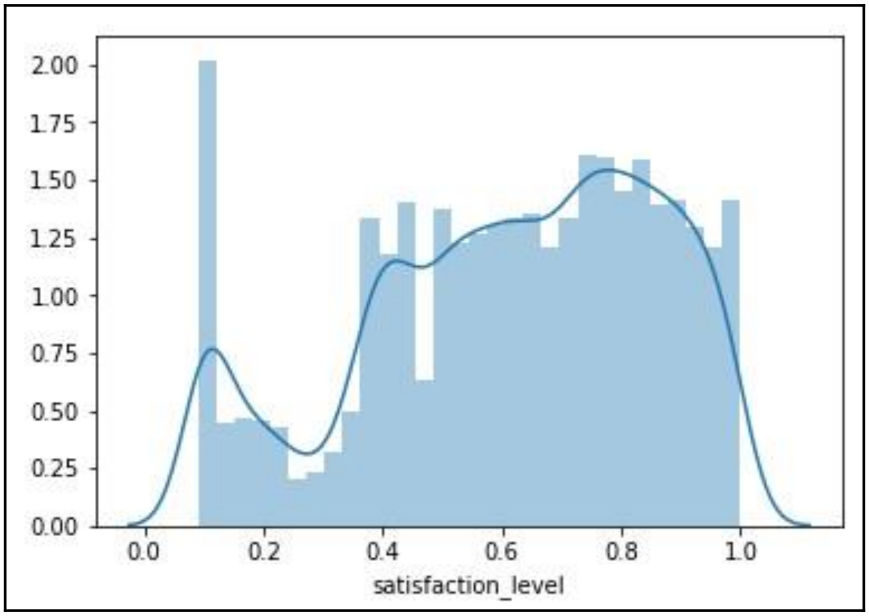


In the preceding example, the bar plot is created using the `bar()` function. It takes two columns – `x` and `y` – and a DataFrame as input. In the next section, we will see how to plot a distribution plot.

Distribution plots

This plots a univariate distribution of variables. It is a combination of a histogram with the default bin size and a **Kernel Density Estimation** (**KDE**) plot. In our example, `distplot()` will take the `satisfaction_level` input column and plot the distribution of it. Here, the distribution of `satisfaction_level` has two peaks:

```
# Create a distribution plot (also known as Histogram)
sns.distplot(df.satisfaction_level)

# Show figure
plt.show()
```



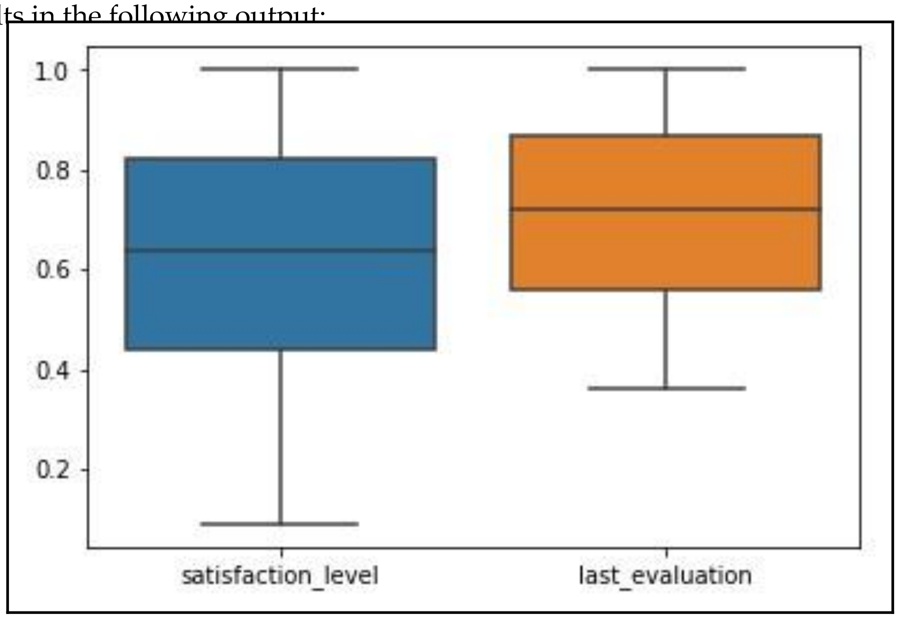
In the preceding code block, we have created the distribution plot using `distplot()`. It's time to jump to the box plot diagram.

Box plots

Box plot, aka box-whisker plot, is one of the best plots to understand the distribution of each variable with its quartiles. It can be horizontal or vertical. It shows quartile distribution in a box, which is known as a whisker. It also shows the minimum and maximum and outliers in the data. We can easily create a box plot using Seaborn:

```
# Create boxplot
sns.boxplot(data=df[['satisfaction_level','last_evaluation']])

# Show figure
plt.show()
```

This results in the following output:



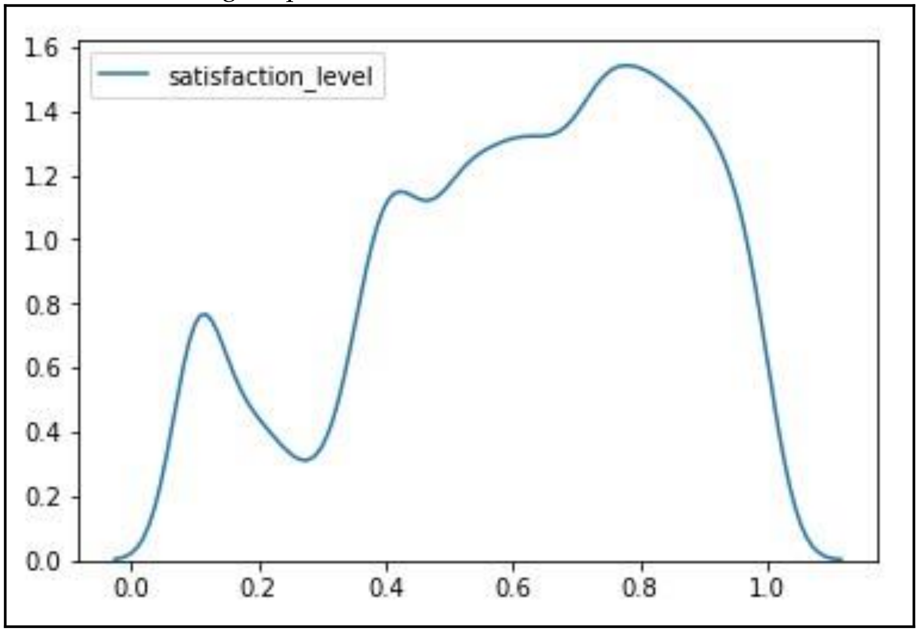
In the preceding example, we have used two variables for the box plot. Here, the box plot indicates that the range of `satisfaction_level` is higher than `last_evaluation` (performance). Let's jump to the KDE plot in Seaborn.

KDE plots

The `kde()` function plots the probability density estimation of a given continuous variable. It is a non-parametric kind of estimator. In our example, the `kde()` function takes one parameter, `satisfaction_level`, and plots the KDE:

```
# Create density plot
sns.kdeplot(df.satisfaction_level)

# Show figure
plt.show()
```

This results in the following output:



In the preceding code block, we have created a density plot using `kdeplot()`. In the next section, we will see another distribution plot, which is a combination of a density and box plot, known as a violin plot.

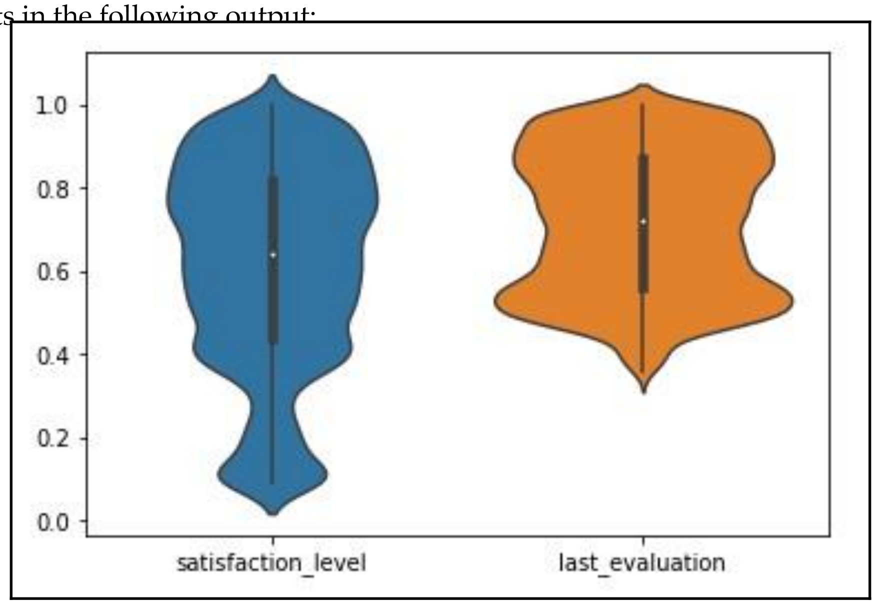
Violin plots

Violin plots are a combined form of box plots and KDE, which offer easy-to-understand analysis of the distribution:

```
# Create violin plot
sns.violinplot(data=df[['satisfaction_level','last_evaluation']])

# Show figure
plt.show()
```

In the preceding example, we have used two variables for the violin plot. Here, we can conclude that the range of `satisfaction_level` is higher than `last_evaluation` (performance) and both the variables have two peaks in the distribution. After working on

This results in the following output:



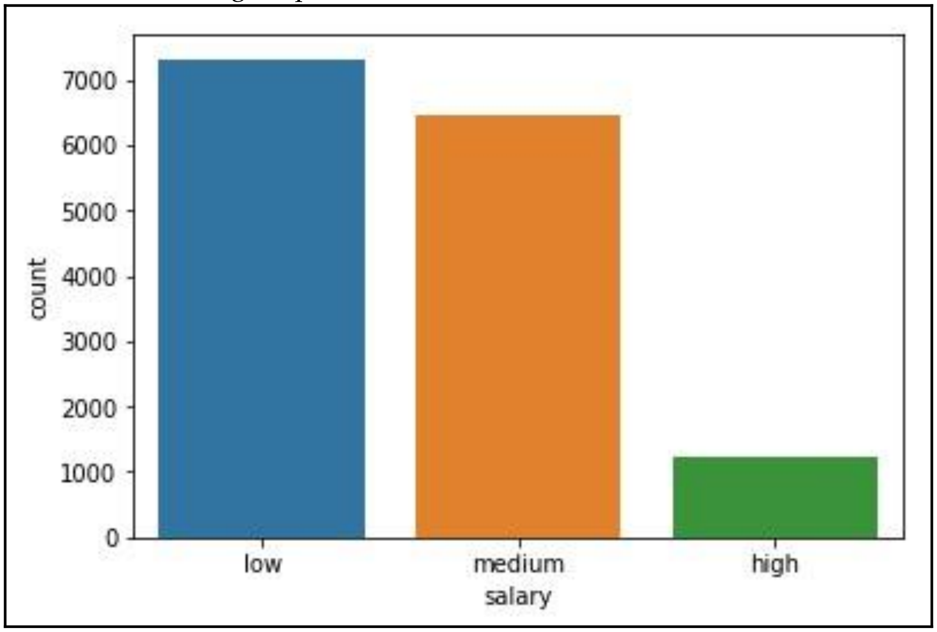
distribution plots, we will see how we can combine the `groupby` operation and box plot into a single plot using a count plot.

Count plots

`countplot()` is a special type of bar plot. It shows the frequency of each categorical variable. It is also known as a histogram for categorical variables. It makes operations very simple compared to Matplotlib. In Matplotlib, to create a count plot, first we need to group by the category column and count the frequency of each class. After that, this count is consumed by Matplotlib's bar plot. But the Seaborn count plot offers a single line of code to plot the distribution:

```
# Create count plot (also known as Histogram)
sns.countplot(x='salary', data=df)

# Show figure
plt.show()
```

This results in the following output:

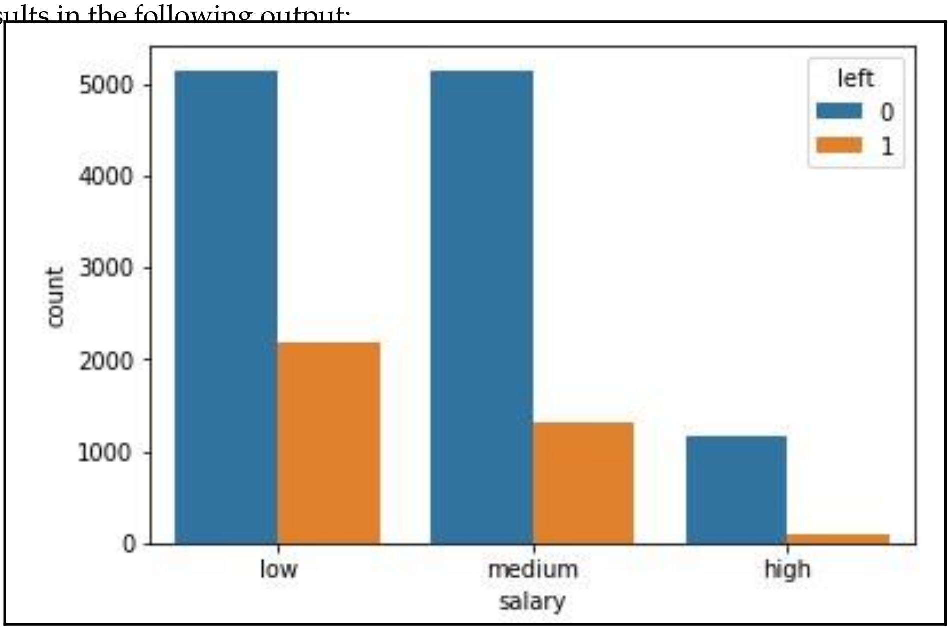


In the preceding example, we are counting the `salary` variable. The `count()` function takes a single column and DataFrame. So, we can easily conclude from the graph that most of the employees have low and medium salaries. We can also use `hue` as the second variable. Let's see the following example:

```
# Create count plot (also known as Histogram)
sns.countplot(x='salary', data=df, hue='left')

# Show figure
plt.show()
```

In the preceding example, we can see that `left` is used as the hue or color shade. This indicates that most of the employees with the lowest salary left the company. Let's see another important plot for visualizing the relationship and distribution of two variables.

This results in the following output:

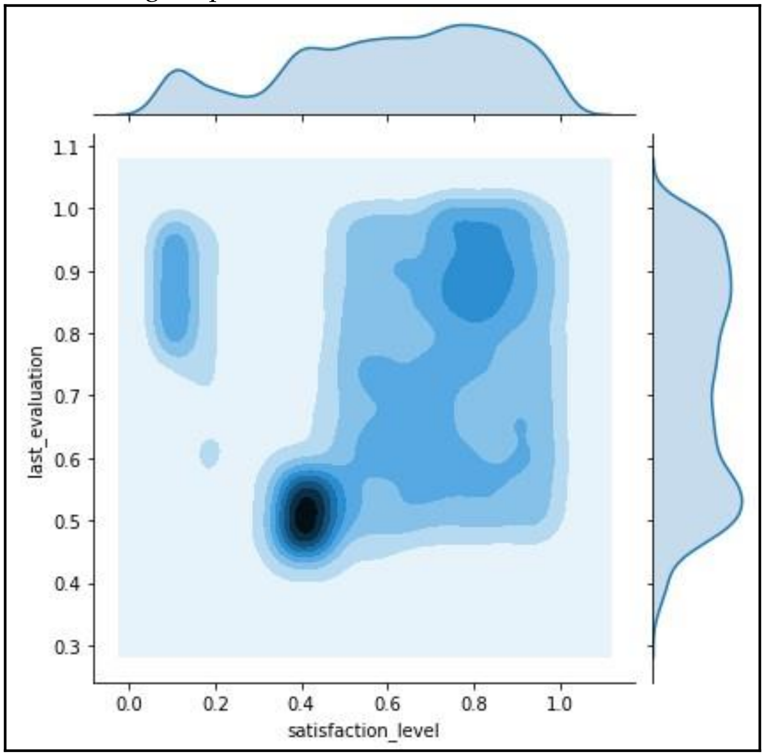


Joint plots

The joint plot is a multi-panel visualization; it shows the bivariate relationship and distribution of individual variables in a single graph. We can also plot a KDE using the `kind` parameter of `jointplot()`. By setting the `kind` parameter as `"kde"`, we can draw the KDE plot. Let's see the following example:

```
# Create joint plot using kernel density estimation(kde)
sns.jointplot(x='satisfaction_level', y='last_evaluation', data=df,
kind="kde")

# Show figure
plt.show()
```

This results in the following output:



In the preceding plot, we have created the joint plot using `jointplot()` and also added the `kde` plot using a `kind` parameter as `"kde"`. Let's jump to heatmaps for more diverse visualization.

Heatmaps

Heatmap offers two-dimensional grid representation. The individual cell of the grid contains a value of the matrix. The heatmap function also offers annotation on each cell:

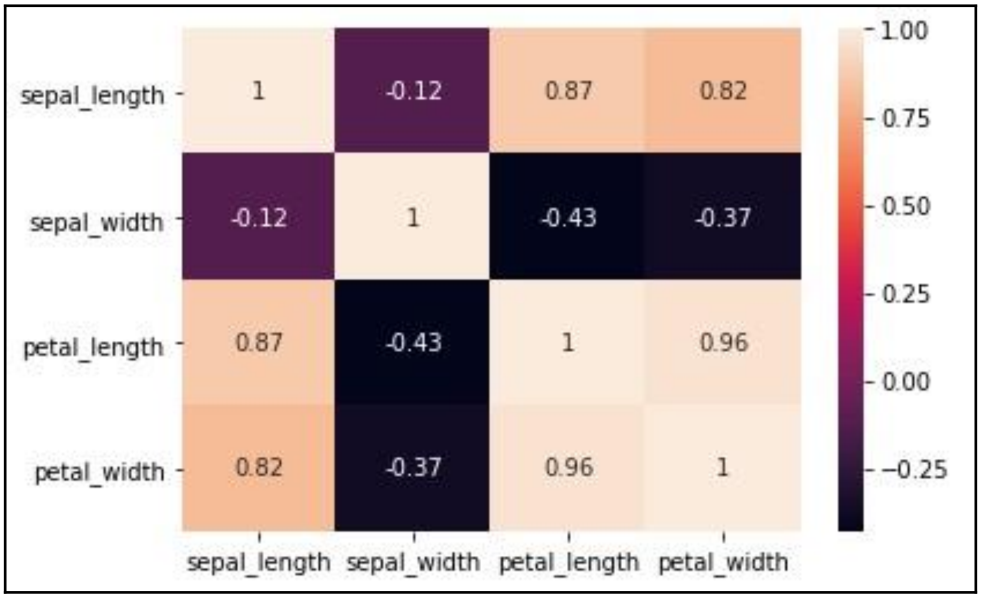
```
# Import required library
import seaborn as sns

# Read iris data using load_dataset() function
data = sns.load_dataset("iris")
```

This results in the following output:

```
    # Find correlation
```

```
cor_matrix=data.corr()

# Create heatmap
sns.heatmap(cor_matrix, annot=True)

# Show figure
plt.show()
```

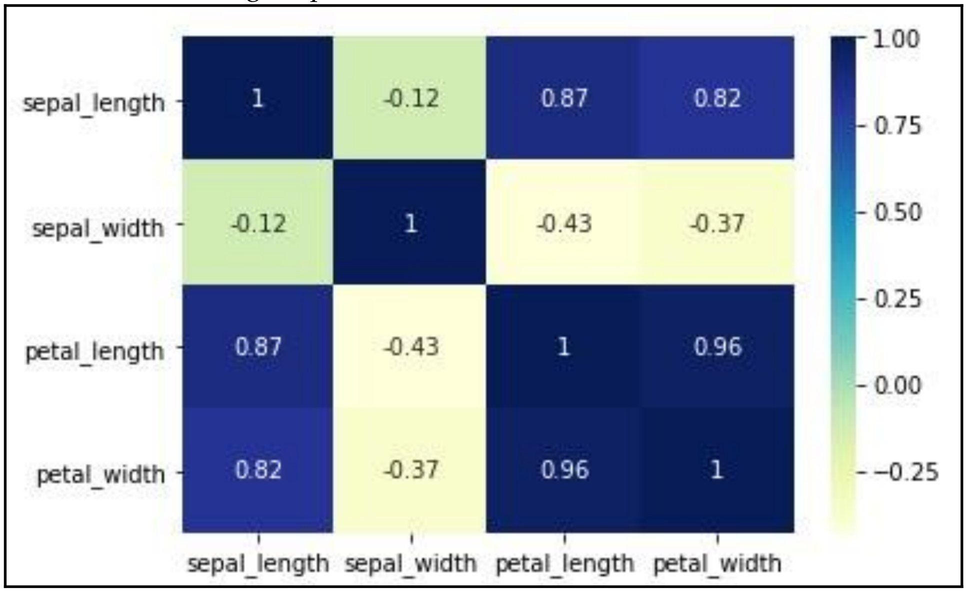
This results in the following output:



In the preceding example, the Iris dataset is loaded using `load_dataset()` and the correlation is calculated using the `corr()` function. The `corr()` function returns the correlation matrix. This correlation matrix is plotted using the `heatmap()` function for the grid view of the correlation matrix. It takes two parameters: the correlation matrix and `annot`. The `annot` parameter is passed as `True`. In the plot, we can see a symmetric matrix, and all the values on the diagonal are ones, which indicates a perfect correlation of a variable with itself. We can also set a new color map using the `cmap` parameter for different colors:

```
# Create heatmap
sns.heatmap(cor_matrix, annot=True, cmap="YlGnBu")

# Show figure
plt.show()
```

This results in the following output:



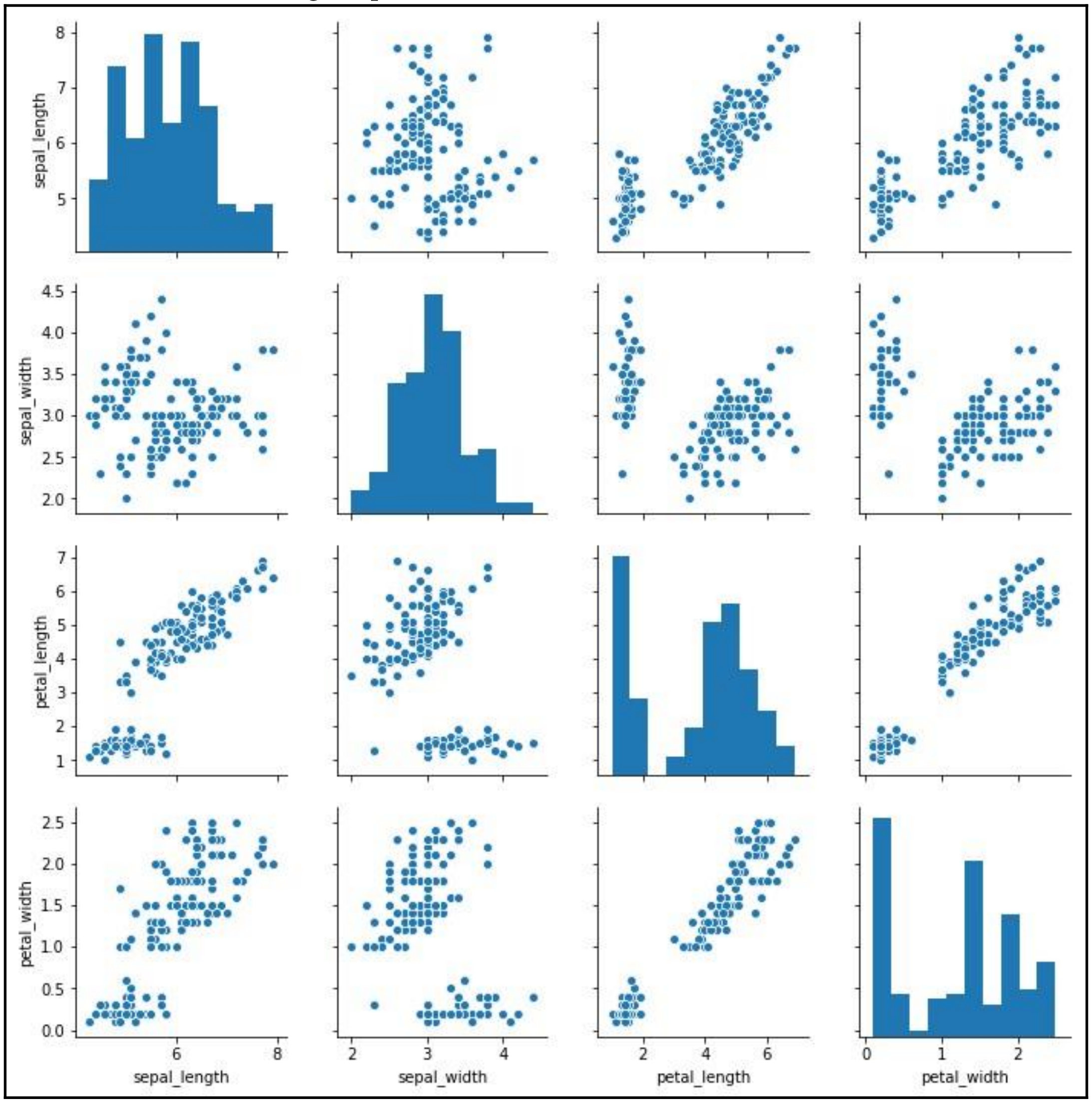
In the preceding heatmap, we have changed the color map using the `cmap` parameter for different colors. Here, we are using the `YlGnBu` (yellow, green, and blue) combination for `cmap`. Now, we will move on to the pair plot for faster exploratory analysis.

Pair plots

Seaborn offers quick exploratory data analysis with relationships and individual distribution using a pair plot. A pair plot offers a single distribution using a histogram and joint distribution using a scatter plot:

```
# Load iris data using load_dataset() function
data = sns.load_dataset("iris")

# Create a pair plot
sns.pairplot(data)

# Show figure
plt.show()
```

This results in the following output:



In the preceding example, the Iris dataset is loaded using `load_dataset()` and that dataset is passed into the `pairplot()` function. In the plot, it creates an *n* by *n* matrix or a grid of graphs. The diagonal shows the distribution of the columns, and the non-diagonal elements of the grid show the scatter plot to understand the relationship among all the variables.