**Task 1(Beginner Level)**

**VISUALISATION**

In today's fast-paced world, vast amounts of data are produced every day. Analyzing this raw data to identify trends and patterns can often be a challenging task. This is where data visualization becomes essential. By transforming raw data into organized, graphical representations, data visualization makes it easier to comprehend, observe, and analyze the information effectively.

Visualization In Python can be done with the help of various libraries such as Matplotlib, Seaborn, Pandas, Plotly, Plotnine, Altair, Bokeh, Pygal, and Geoplotlib.

**Matplotlib**

It is a versatile library designed to help users visualise data in various formats. It is well-suited for creating a wide range of static, animated, and interactive plots.

**Key Features of Matplotlib**

* **Versatile Plotting**: Create a wide variety of visualizations, including line plots, scatter plots, bar charts, and histograms.
* **Extensive Customization**: Control every aspect of your plots, from colors and markers to labels and annotations.
* **Seamless Integration with NumPy**: Effortlessly plot data arrays directly, enhancing data manipulation capabilities.
* **High-Quality Graphics**: Generate publication-ready plots with precise control over aesthetics.
* **Cross-Platform Compatibility**: Use Matplotlib on Windows, macOS, and Linux without issues.
* **Interactive Visualizations**: Engage with your data dynamically through interactive plotting features.

**Types of plots in Matplotlib**

**1. Line Graph**

import matplotlib.pyplot as plt

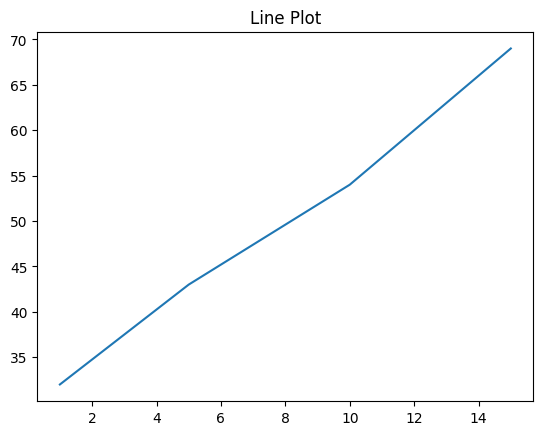
x=[1,5,10,15]

y=[32,43,54,69]

plt.plot(x,y)

plt.title('Line Plot')

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt. The variables x and y are defined as lists representing the data points for the x-axis and y-axis, respectively, with x being the independent variable and y the dependent variable. The plt.plot(x, y) function generates a line plot by connecting the data points (1, 32), (5, 43), (10, 54), and (15, 69) with straight lines. The plt.title('Line Plot') adds the title "Line Plot" to the graph, while plt.show() displays the plot in a graphical window.

**2. Bar Chart**

import matplotlib.pyplot as plt

colors=['Red','Yellow','Blue','Green']

count=[1000,500,700,1100]

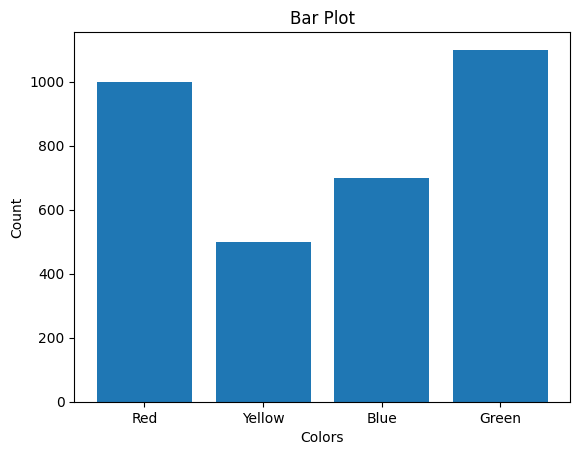
plt.bar(colors,count)

plt.title('Bar Plot')

plt.xlabel('Colors')

plt.ylabel('Count')

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt. The variables colors and count are defined as lists, where colors represents categories (Red, Yellow, Blue, Green) and count represents their corresponding numerical values (1000, 500, 700, 1100). The plt.bar(colors, count) function generates a bar plot with the colors on the x-axis and their counts as the height of the bars. The title of the plot is set to "Bar Plot" using plt.title('Bar Plot'), while plt.xlabel('Colors') and plt.ylabel('Count') label the x-axis and y-axis, respectively.

**3. Histogram**

import matplotlib.pyplot as plt

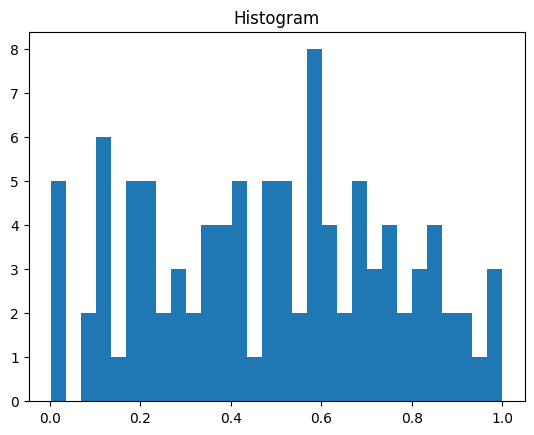
import numpy as np

x=np.random.rand(100)

plt.hist(x,bins=30)

plt.title('Histogram')

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt and the numpy library as np. The variable x is defined using np.random.rand(100), which generates an array of 100 random values uniformly distributed between 0 and 1. The plt.hist(x, bins=30) function creates a histogram, dividing the range of values in x into 30 bins and showing the frequency of data points within each bin. The plt.title('Histogram') sets the title of the plot to "Histogram," and plt.show() displays the histogram.

**4. Scatter Plot**

import matplotlib.pyplot as plt

import numpy as np

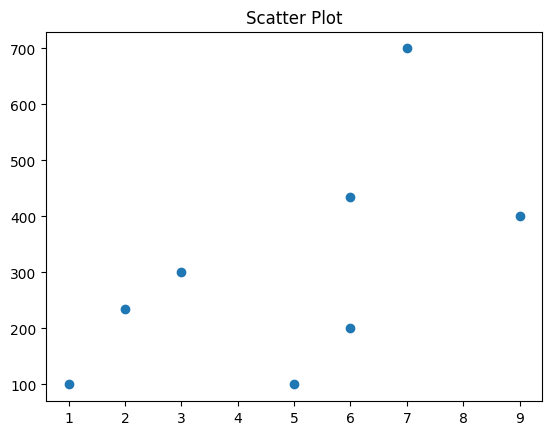
x=[1,6,3,7,5,9,2,6]

y=[100,200,300,700,100,400,234,434]

plt.scatter(x,y)

plt.title('Scatter Plot')

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt and the numpy library as np. The variables x and y are defined as lists, where x represents the data points on the x-axis (e.g., [1, 6, 3, 7, 5, 9, 2, 6]) and y represents the corresponding values on the y-axis (e.g., [100, 200, 300, 700, 100, 400, 234, 434]). The plt.scatter(x, y) function creates a scatter plot by plotting individual points at the specified (x, y) coordinates. The title of the plot is set to "Scatter Plot" using plt.title('Scatter Plot'). Finally, plt.show() displays the scatter plot.

**5. Pie Chart**

import matplotlib.pyplot as plt

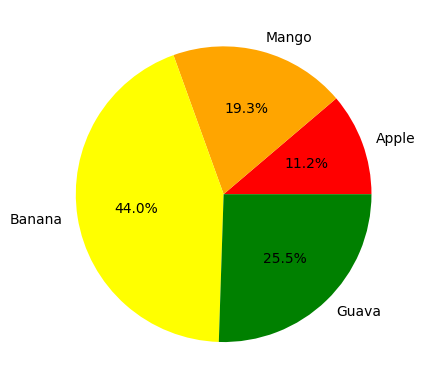
label=['Apple','Mango','Banana','Guava']

fruits=[200,344,784,455]

color=['Red','Orange','Yellow','Green']

plt.pie(count,labels=label,autopct='%1.1f%%',colors=color)

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt. The variables label, fruits, and color are defined as lists. The label list specifies the categories (Apple, Mango, Banana, Guava), fruits represents the numerical data corresponding to each category, and color defines the colors assigned to each slice (Red, Orange, Yellow, Green). The plt.pie() function generates the pie chart, where fruits is the data, labels=label specifies the category names, autopct='%1.1f%%' displays the percentage of each slice, and colors=color assigns the slice colors. Finally, plt.show() displays the pie chart.

**6. 3D Plot**

import matplotlib.pyplot as plt

fig=plt.figure()

ax=plt.axes(projection='3d')

x=[1,2,4,5,6,9]

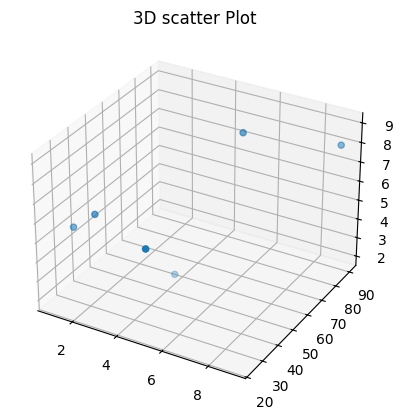
y=[34,34,55,24,70,90]

z=[5,6,2,6,9,8]

ax.scatter(x,y,z)

plt.title('3D scatter Plot')

plt.show()



**Description:**

Begins by importing the pyplot module from matplotlib as plt. The fig=plt.figure() line initializes a new figure object, and ax=plt.axes(projection='3d') creates a 3D plotting area using a 3D projection. The variables x, y, and z are defined as lists representing the coordinates for the x-axis, y-axis, and z-axis, respectively. The ax.scatter(x, y, z) function creates a 3D scatter plot, plotting points at the specified (x, y, z) coordinates. The plt.title('3D Scatter Plot') adds a title to the plot, and plt.show() displays the 3D scatter plot.

**Seaborn**

Seaborn is a Python library that simplifies **the creation of attractive and informative statistical graphics**. It integrates seamlessly with Pandas DataFrames and offers a range of functions tailored for visualizing statistical relationships and distributions.

**Key Features of Seaborn:**

* **High-level interface**: Simplifies the creation of complex visualizations.
* **Integration with Pandas**: Works seamlessly with Pandas DataFrames for data manipulation.
* **Built-in themes**: Offers attractive default themes and color palettes.
* **Statistical plots**: Provides various plot types to visualize statistical relationships and distributions.

**Types of Plots in Seaborn**

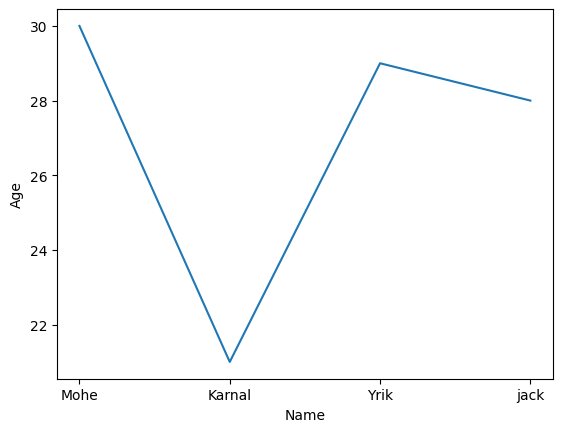
1. **Line Plot**

import pandas as pd

import seaborn as sns

data=pd.DataFrame({'Name':[ 'Mohe','Karnal','Yrik','jack' ],'Age':[30,21,29,28]})

sns.lineplot(x=data['Name'],y=data['Age'],data=data)



**Description:**

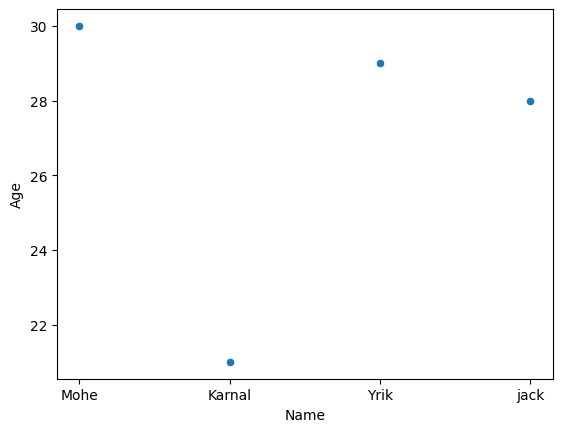
Begins by importing pandas as pd to manage the dataset and seaborn as sns for visualization. A DataFrame is created with two columns: 'Name', containing names of individuals (e.g., ['Mohe', 'Karnal', 'Yrik', 'Jack']), and 'Age', representing their respective ages (e.g., [30, 21, 29, 28]). The sns.lineplot() function is used to generate a line plot with names on the x-axis and ages on the y-axis. While the plot displays a clear trend, line plots are typically more suited for numerical or time-series data on the x-axis.

1. **Scatter Plot**

import seaborn as sns

data=pd.DataFrame({'Name':[ 'Mohe' , 'Karnal' , 'Yrik' , 'jack' ],'Age':[ 30, 21 , 29 ,28]})

sns.scatterplot(x=data['Name'],y=data['Age'],data=data)



**Description:**

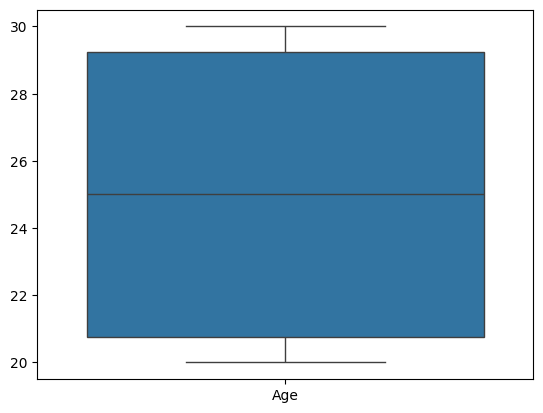
The Python code uses the seaborn library to create a scatter plot. A pandas DataFrame is constructed with two columns: 'Name', which lists the names of individuals, and 'Age', which provides their corresponding ages. The sns.scatterplot() function is employed to generate a scatter plot, plotting the 'Name' column on the x-axis and the 'Age' column on the y-axis.

1. **Boxplot**

import seaborn as sns

data=pd.DataFrame({'Name':['Mohe','Karnal','Yrik','jack'],'Age':[30,21,29,20]})

sns.boxplot(data=data)



**Description:**

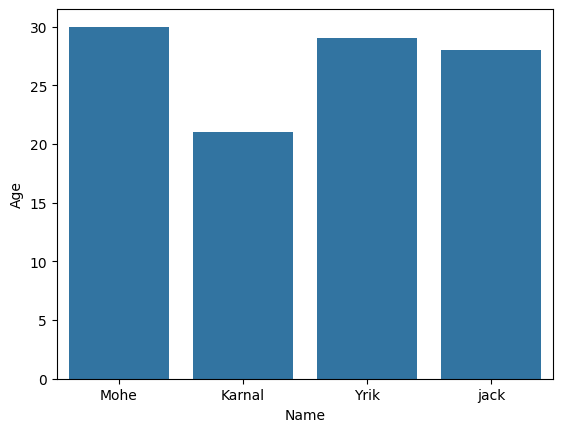
The Python code uses the seaborn library to create a box plot. A pandas DataFrame is created with two columns: 'Name', representing the names of individuals, and 'Age', representing their corresponding ages. The sns.boxplot() function is used to generate a box plot from the data.

1. **Bar Plot**

import seaborn as sns

data=pd.DataFrame({'Name':['Mohe','Karnal','Yrik','jack'],'Age':[30,21,29,28]})

sns.barplot(x=data['Name'],y=data['Age'],data=data)



**Description:**

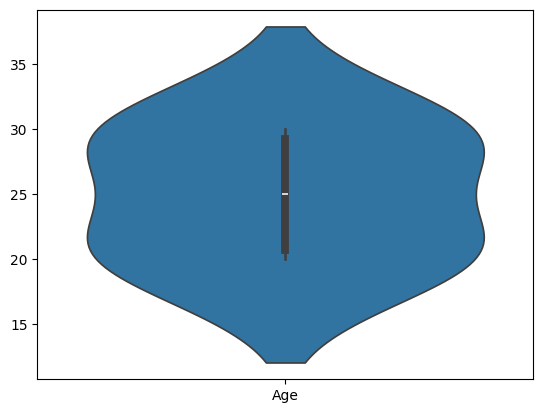
The Python code uses the seaborn library to create a bar plot. A pandas DataFrame is defined with two columns: 'Name', which lists the names of individuals, and 'Age', representing their respective ages. The sns.barplot() function is used to generate a bar plot with the 'Name' column on the x-axis and the 'Age' column on the y-axis.

1. **Violin Plot**

import seaborn as sns

data=pd.DataFrame({'Name':[ 'Mohe','Karnal','Yrik','jack'],'Age':[ 30,21, 29,20 ]})

sns.violinplot(data=data)



**Description:**

The Python code uses the seaborn library to create a violin plot. A pandas DataFrame is created with two columns: 'Name', representing individuals' names, and 'Age', representing their respective ages. The sns.violinplot() function is used to generate a violin plot, which visualizes the distribution of the 'Age' data.

1. **Swarm Plot**

Import numpy as np

Import pandas as pd

Import seaborn as sns

Import matplotlib.pyplot as plt

Np.random.seed(42)

Categories = [‘A’, ‘B’, ‘C’, ‘D’]

Data = {

‘category’: np.random.choice(categories, size=100),

‘value’: np.random.rand(100) \* 100

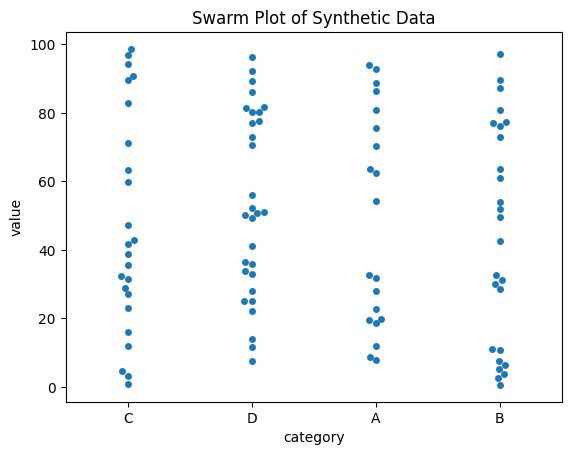
}

Df = pd.DataFrame(data)

Sns.swarmplot(x=’category’, y=’value’, data=df)

Plt.title(‘Swarm Plot of Synthetic Data’)

Plt.show()



**Description:**

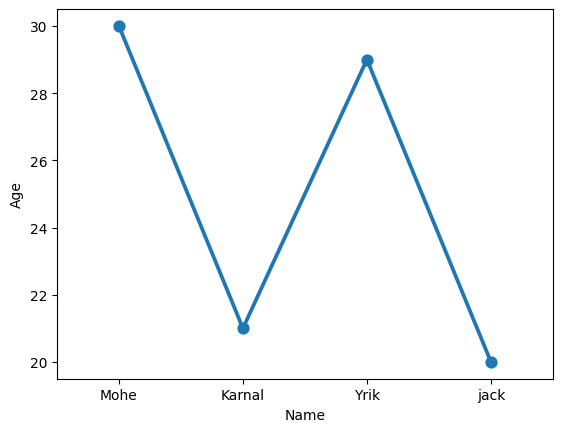
Begins by importing necessary libraries: NumPy for numerical operations, Pandas for data manipulation, Seaborn for visualization, and Matplotlib for displaying the plot. A random seed is set for reproducibility, and synthetic data is created with 100 random values assigned to four categories (A, B, C, D). This data is organized into a Pandas DataFrame. Finally, a swarm plot is created using Seaborn to visualize the distribution of values across categories, complete with a title, and displayed using Matplotlib.

1. **Point Plot**

import seaborn as sns

data=pd.DataFrame({'Name':['Mohe','Karnal','Yrik','jack'],'Age':[30,21, 29,20]})

sns.pointplot(x=data['Name'],y=data['Age'],data=data)



**Description:**

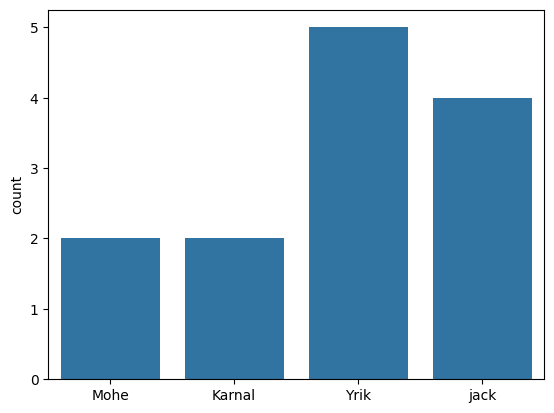
The Python code uses the seaborn library to create a point plot. A pandas DataFrame is created with two columns: 'Name', which contains individuals' names, and 'Age', representing their respective ages. The sns.pointplot() function is used to generate the point plot, with 'Name' on the x-axis and 'Age' on the y-axis.

1. **Count Plot**

import seaborn as sns

data=['Mohe','Karnal','Yrik','jack','Mohe','Karnal','Yrik','jack','Yrik','jack','Yrik','jack','Yrik']

sns.countplot(x=data)



**Description:**

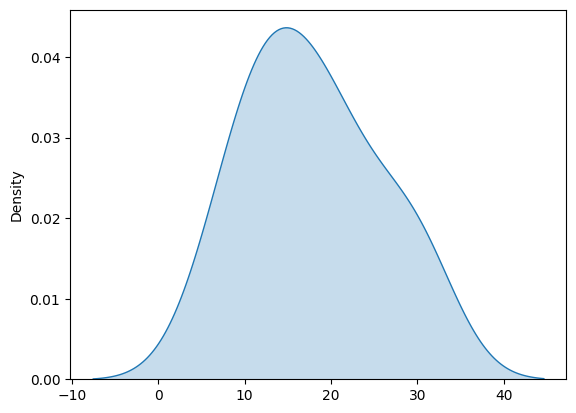
The Python code uses the seaborn library to create a count plot. A list named data is defined, containing repeated names of individuals. The sns.countplot() function is used to generate the count plot, with the data list on the x-axis.

1. **KDE Plot**

import seaborn as sns

data = [12, 15, 7, 10, 22, 29, 18, 21, 14, 30]

sns.kdeplot(data,shade=True)



**Description:**

The Python code uses the seaborn library to create a Kernel Density Estimate (KDE) plot. A list named data is defined, containing numerical values. The sns.kdeplot() function is used to generate the KDE plot, which estimates and visualizes the probability density function of the data. The shade=True parameter fills the area under the curve with color, making the plot easier to interpret.

**Matplotlib(Strengths and Weaknesses):**

|  |  |  |
| --- | --- | --- |
|  | **Strength** | **Weakness** |
| **Ease of Use** | •Matplotlib provides a flexible and powerful interface for creating a wide variety of plots and visualizations, from basic to advanced.  •However, the syntax can be verbose and less intuitive compared to high-level libraries like Seaborn. Users need to explicitly define each element (axes, labels, colors, etc.), which can make it harder for beginners.  •Once users are familiar with its structure, Matplotlib allows for the creation of almost any type of plot, making it a foundational tool in Python for data visualization. | •Matplotlib can be challenging for beginners due to its verbose syntax and low-level approach. Unlike libraries like Seaborn, which offer a simpler interface, Matplotlib often requires more lines of code to create visually appealing plots.  •It requires the user to explicitly define all plot components, such as axis labels, titles, and legends, making it more time-consuming to produce simple visualizations. |
| **Customization Options** | •Matplotlib is highly customizable and allows detailed control over every aspect of the plot (e.g., axis labels, font sizes, colors, line styles, tick marks, legends, etc.).  •It supports extensive styling options, including customizing the figure's background, adjusting plot aesthetics, or adding specific annotations and markers.  •Advanced users can modify low-level details, such as tick positioning and complex figure layouts, to create tailored visualizations. | •While Matplotlib provides extensive customization options, the sheer number of options can be overwhelming for new users.  •The customization process often requires dealing with complex object-oriented programming concepts, such as managing Axes, Figure, and Artist objects, which may not be intuitive for everyone.  •Customizing interactive or dynamic plots (e.g., adding hover effects) may require additional coding or integration with other libraries, making the process more cumbersome than using more specialized libraries. |
| **Interactivity** | •Matplotlib offers basic interactivity through its interactive mode, allowing users to zoom, pan, or update the plot in real-time during exploration.  •It also supports integration with interactive environments like Jupyter notebooks, where plots can be updated dynamically.  •For more complex interactivity (e.g., hover or click events), users typically need to rely on additional libraries, such as mplcursors or plotly, for enhanced interaction. | •Matplotlib’s interactive features are basic and not as sophisticated as those offered by libraries like Plotly or Bokeh. While it supports zooming and panning, it doesn’t natively support more advanced interactions (e.g., tooltips, clicking events) without relying on external packages.  •Although Matplotlib can be embedded into interactive environments, it lacks the seamless interactivity required for modern web applications or dashboards, where more dynamic libraries are preferred. |
| **Performance with Large Datasets** | •Matplotlib performs relatively well with large datasets, especially when compared to high-level libraries like Seaborn.  •It can handle large numbers of data points, but performance might degrade if the plot contains millions of points or very high-resolution data. Techniques like downsampling or simplifying data can help manage performance.  •Matplotlib provides fine-grained control over data rendering, which allows users to implement optimizations for performance in large-scale visualizations (e.g., thinning data or plotting only a subset of points). | •Matplotlib is capable of handling large datasets, but performance issues may arise when dealing with extremely large volumes of data (millions of points).  •Rendering times can increase significantly with larger datasets, and memory usage can become a concern when plotting high-resolution or dense data. Optimization techniques like downsampling or using scatter instead of plot for dense datasets can help alleviate this.  •It doesn’t have the same built-in optimizations for big data visualization as other libraries such as Datashader or HoloViews, which are specifically designed for handling massive datasets. |

**Conclusion:**

Matplotlib excels in customization, flexibility, and the ability to create virtually any type of plot. However, its verbosity and steep learning curve make it less accessible for beginners. While it performs well with large datasets, it can struggle with very large data volumes and lacks advanced interactive features. For simple visualizations and intricate customizations, Matplotlib is an excellent choice, but for interactive and high-performance visualizations, other tools like Plotly or Bokeh might be more appropriate.

**Seaborn(Strengths and Weaknesses):**

|  |  |  |
| --- | --- | --- |
|  | **Strength** | **Weakness** |
| **Ease of Use** | •Seaborn provides a simple and high-level interface to create complex statistical plots with just a few lines of code.  •It is built on top of Matplotlib, which means it inherits a lot of functionality but streamlines the process for common types of visualizations like bar plots, scatter plots, heatmaps, etc.  •With functions like sns.scatterplot() or sns.barplot(), you can quickly generate aesthetically pleasing and informative plots without needing to manually adjust axis labels, colors, or styles. | •While Seaborn simplifies complex visualizations, it might be a bit abstracted for users who want complete control over every aspect of the plot. For example, creating very customized or unique types of plots may require diving into Matplotlib or other libraries for more granular control. |
| **Customization Options** | •Seaborn allows users to easily customize their plots by adjusting color palettes, themes, axis labels, and plot sizes.  •It has built-in functions for modifying the appearance of plots (e.g., sns.set\_theme(), sns.set\_palette()), making it user-friendly to fine-tune visualizations.  •The integration with Matplotlib means you can further customize Seaborn plots using Matplotlib’s capabilities for precise control over every element. | •While Seaborn offers many customization options, its interface may sometimes be limiting for more advanced or niche customizations. In such cases, users may need to revert to Matplotlib for fine-tuning.  •Seaborn's plotting style is fixed, and although customization is possible, it might require more effort compared to other libraries like Plotly, which offer built-in features for interactive customization. |
| **Interactivity** | •Seaborn itself is not inherently designed for interactivity but works well with interactive environments like Jupyter notebooks and can integrate with Matplotlib's interactive features.  •For more advanced interactivity, additional tools like Plotly or Bokeh are often used in combination with Seaborn. | •Seaborn lacks interactive capabilities out of the box, which limits its use in web applications or dashboards. Users typically need to integrate Seaborn with other libraries like Plotly or use Matplotlib’s interactive backends for that purpose.  •Seaborn’s static plots do not support features like zoom, pan, or hover without additional work. |
| **Performance with Large Datasets** | •Seaborn is not as optimized for performance with large datasets as some other libraries (like Datashader or HoloViews).  •When dealing with large datasets (e.g., millions of data points), Seaborn may experience slow rendering times and be less efficient compared to libraries designed for big data visualization.  •However, Seaborn works fine for medium-sized datasets (up to several hundred thousand rows) but may require aggregation or downsampling techniques for large datasets. | •Seaborn can struggle with performance when working with very large datasets. For large-scale data, Seaborn often requires data preprocessing or aggregation (e.g., grouping or downsampling) before plotting.  •It doesn’t have specialized optimizations for handling large data like libraries designed specifically for big data visualizations, which could lead to slower rendering times or memory issues when plotting large datasets directly. |

**Conclusion:**

Seaborn excels in ease of use, customization, and creating high-quality static plots quickly. However, it is not as interactive or performant with very large datasets. For interactive or real-time visualizations, other libraries like Plotly or Bokeh may be more suitable. For extremely large datasets, specialized tools like Datashader might be more effective.