UNIT-IV

Data Visualization and Data Wrangling

Data wrangling

- Data wrangling, also known as data munging or data cleaning, is the process of transforming and preparing raw data into a format that is suitable for analysis.
- This step is often considered one of the most time-consuming and crucial parts of the data analysis process.
- Effective data wrangling ensures that your data is accurate, complete, and ready for further exploration and analysis.

- A data wrangling process, also known as a data munging process, consists of reorganizing, transforming and mapping data from one "raw" form into another in order to make it more usable and valuable for a variety of downstream uses including analytics.
- Data wrangling can be defined as the process of cleaning, organizing, and transforming raw data into the desired format for analysts to use for prompt decision-making.

1. Discovering:

- The discovery process is the initial step in the data wrangling process.
- It is a step toward gaining a better understanding of the data.
- To make your data easier to use and analyze, you must look at it and consider how you would like the data to be arranged.

2. Structuring:

- The process of taking unprocessed data and converting it so that it may be used more easily is known as data structuring.
- This is the method for extracting relevant information from new data.
- The data can be structured in a spreadsheet by adding columns, classes, headings, etc.

3. Cleaning:

- It **involves** identifying **data errors** and then changing, updating or **removing data** to correct them
- Data cleaning or remediation aims to ensure that the final data for analysis is not impacted.
- Raw data usually contains errors that must be cleaned before it can be used.

4. Enriching:

- Adding context to the data is what is meant by enriching.
- This process transforms previously cleaned and formatted data into new types.
- At this point, you need to plan strategically for the information you already have to get the most out of it.

5.Validating:

- The process of ensuring that your data is accurate and consistent is known as data validation.
- This step can reveal problems that need to be fixed or conclude that the data is ready for analysis.

6. Publishing:

- It's about putting the new wrangled data in a place where you and other stakeholders can easily find and use it.
- The information can be added to a fresh database.
- As long as you follow the previous steps, you'll have high-quality data for insights, business reports, and more.

Benefits of Data Wrangling:

- Data wrangling helps to improve data usability as it converts data into a compatible format for the end system.
- It helps to quickly build data flows within an intuitive user interface and easily schedule and automate the data-flow process.
- Integrates various types of information and their sources (like databases, web services, files, etc.)
- Help users to process very large volumes of data easily and easily share data-flow techniques.

Data Wrangling Examples

- Data wrangling techniques are used for various use-cases. The most commonly used examples of data wrangling are for:
- Merging several data sources into one data-set for analysis
- Identifying gaps or empty cells in data and either filling or removing them
- Deleting irrelevant or unnecessary data
- Identifying severe outliers in data and either explaining the inconsistencies or deleting them to facilitate analysis

Businesses also use data wrangling tools :

- Detect corporate fraud
- Support data security
- Ensure accurate and recurring data modeling results
- Ensure business compliance with industry standards
- Perform Customer Behavior Analysis
- Reduce time spent on preparing data for analysis
- Promptly recognize the business value of your data
- Find out data trends

Here's a step-by-step guide to data wrangling:

1. Data Collection:

- Gather data from various sources, such as databases, spreadsheets, APIs, or external files. Ensure that you have a clear understanding of the data's source and structure.

2. Data Inspection:

- Examine the raw data to get a sense of its size, structure, and quality.
- Check for missing values, duplicates, and outliers.
- Understand the data types of each column (numeric, categorical, text, date, etc.).

3. Data Cleaning:

- Handle missing data:
  - Decide whether to impute missing values, remove rows with missing data, or use other strategies like forward-fill or backward-fill.
- Handle duplicates:
  - Identify and remove duplicate rows if they exist.
- Handle outliers:
  - Identify and decide how to deal with outliers (e.g., remove, transform, or keep them).
- Standardize data:
  - Ensure consistent units, formats, and naming conventions across the dataset.
- Correct errors:
  - Review and correct any obvious data entry errors or inconsistencies.

4. Data Transformation:

- Encode categorical variables:
  - Convert categorical variables into numerical representations using techniques like one-hot encoding(One hot encoding generates binary columns for each category) or label encoding(label encoding provides each category a unique numeric label).
- Feature scaling:
  - Normalize or standardize numeric features to have comparable scales.
- Create derived features:
  - Generate new features based on existing ones, such as calculating ratios, aggregations, or transformations.
- Date and time parsing:
  - Extract meaningful information from date and time columns (e.g., year, month, day, hour).
- Text data preprocessing:
  - Clean and tokenize text data, remove stop words, and perform stemming or lemmatization if necessary.
- Handle data imbalances:
  - Address class imbalances in classification problems through techniques like oversampling, undersampling, or synthetic data generation.
- **5. Data Integration:**

- Combine data from multiple sources or files if your analysis requires it. Ensure that the integration process doesn't introduce data inconsistencies or errors.

6. Data Reduction:

- Reduce the dimensionality of your dataset if it's too large or complex. Techniques like Principal Component Analysis (PCA) or feature selection can help.

7. Data Validation and Quality Assurance:

- Validate the integrity and quality of the cleaned and transformed data.
- Perform checks to ensure that the data meets your specific business or analytical requirements.

8. Documentation:

- Document all the steps you've taken during data wrangling, including data cleaning, transformation, and any decisions made. This documentation is essential for reproducibility and collaboration.

9. Iteration:

- Data wrangling is often an iterative process. As you explore and analyze
- the data, you may discover further issues or transformations that need to be addressed.

Hierarchical Indexing

- Hierarchical indexing, also known as multi-level indexing, can be a valuable technique in data wrangling, especially when dealing with complex, structured datasets.
- It helps you organize and manipulate data with multiple levels of categorization or grouping, making it easier to work with hierarchical or multi-dimensional data.
- Here's how hierarchical indexing can be used in data wrangling:

1.Creating Hierarchical Indexing:

- You can use hierarchical indexing to structure your data during the data wrangling process.
- For example, when merging or joining multiple datasets, you might have multiple keys or levels of grouping. Hierarchical indexing allows you to create a structured index that represents these levels.
- import pandas as pd

```
# Create DataFrames
df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]}, index=['X', 'Y', 'Z'])
df2 = pd.DataFrame({'C': [7, 8, 9], 'D': [10, 11, 12]}, index=['X', 'Y', 'Z'])
# Merge DataFrames with hierarchical indexing
df_merged = pd.concat([df1, df2], keys=['df1', 'df2'])
```

- In this example, the ‘**keys**’ parameter creates a multi-level index with levels 'df1' and 'df2', which can help you keep track of the source of each row in the merged DataFrame.

2.Grouping and Aggregation:

- During data wrangling, you often need to group data by one or more attributes and perform aggregation operations.
- Hierarchical indexing can be used to group data at multiple levels, allowing you to compute aggregates separately for each level of the index.

```
# Grouping by multiple index levels
grouped = df_merged.groupby(level=0)
result = grouped.sum()
```

- Here, you're grouping by the first level of the hierarchical index ('df1' and 'df2') and computing the sum for each group.

3.Slicing and Subsetting:

- Hierarchical indexing makes it easy to slice and subset data at various levels of granularity.
- You can select data for specific levels of the index or perform cross-section operations to extract data from multiple levels.

```
# Selecting data at a specific index level
df1_data = df_merged.loc['df1']
```

4.Reshaping Data:

- When restructuring data during data wrangling, hierarchical indexing can be used to create pivot tables or change data layouts between wide and long formats while preserving the hierarchical structure.

```
# Pivoting data with hierarchical index
pivoted_data = df_merged.unstack()
```

- Here, **‘unstack’** reshapes the data from a multi-level index to a more traditional DataFrame.

5.Merging and Joining:

- When merging datasets with different hierarchical structures, you can use hierarchical indexing to ensure that the resulting merged DataFrame maintains the original structure.

```
# Merging DataFrames with hierarchical index
merged_df = df1.join(df2, how='outer')
```

- Hierarchical indexing can help ensure that you can correctly identify and merge data based on multiple index levels.

Combining and Merging Data Sets Reshaping and Pivoting.

- Combining and merging data sets, as well as reshaping and pivoting data, are essential data wrangling techniques when working with structured and relational data.
- These operations help you prepare and structure your data for analysis or reporting.

Combining and Merging Data Sets:

- Concatenation is used to combine two or more data sets along a particular axis, typically stacking them on top of each other (row-wise) or side by side (column-wise).
- You can use tools like '**pd.concat**' in pandas for this purpose.

```
import pandas as pd
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})
result = pd.concat([df1, df2])  # Stacking vertically (default)
```

Merging (Joining):

- Merging combines data sets by linking them using common columns (keys).
- It's similar to SQL joins and is useful when you have related data in different tables.

You can use tools like ‘pd.merge’ in pandas.

```
df1 = pd.DataFrame({'key': ['A', 'B'], 'value': [1, 2]})
df2 = pd.DataFrame({'key': ['B', 'C'], 'value': [3, 4]})
result = pd.merge(df1, df2, on='key')  # Inner join on the 'key' column
```

Joining on Multiple Columns:

- You can merge data sets on multiple columns for more complex relationships.

```
result = pd.merge(df1, df2, on=['key1', 'key2'])
```

Different Types of Joins:

- Merges can be inner joins, left joins, right joins, or outer joins, depending on how you want to handle missing values in the merged data.

Reshaping and Pivoting Data:

- **Pivoting Data:** Pivoting is the process of transforming data from a long format to a wide format. You specify columns to be used as the new index, columns to be used as new columns, and values to be filled in the new cells.
- Pandas provides the '**pivot**' and '**pivot_table**' functions for this.

```
df = pd.DataFrame({'date': ['2021-01-01', '2021-01-01', '2021-01-02'],
                   'variable': ['A', 'B', 'A'],
                   'value': [1, 2, 3]})
pivoted = df.pivot(index='date', columns='variable', values='value')
```

Melting Data:

- Melting is the reverse of pivoting. It transforms wide data into a long format.
- This is useful when you want to stack columns into rows.

```
melted = pivoted.melt(id_vars='date', var_name='variable',
value_name='value')
```

Stacking and Unstacking:

- Stacking and unstacking operations allow you to change the hierarchical structure of a DataFrame, converting between long and wide formats.

```
stacked = df.set_index(['date', 'variable']).unstack('variable')
unstacked = stacked.stack('variable')
```

Transposing:

- You can also transpose a DataFrame to switch rows and columns.

```
transposed = df.transpose()
```

Data Visualization matplotlib: Basics of matplotlib,

- Matplotlib is a popular Python library for creating static, animated, and interactive visualizations.
- It provides a wide range of tools for creating various types of plots and charts, making it a valuable tool for data visualization.
- Here's a brief overview of how to get started with Matplotlib for data visualization:

1.Import Matplotlib:

- You need to import the Matplotlib library, usually as **‘matplotlib.pyplot’.**
- You can also use ‘ **%matplotlib inline’** or **‘%matplotlib notebook’** in Jupyter Notebook to display plots inline.

```
import matplotlib.pyplot as plt
```

2. Create Basic Plots:

- Matplotlib supports various types of plots, including line plots, scatter plots, bar charts, histograms, and more.
- To create a basic plot, you can use the '**plot**' function.

```
# Example of a simple line plot
x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 9]
plt.plot(x, y)
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Simple Line Plot')
plt.show()
```

3. Customize Plots:

- Matplotlib provides extensive customization options for your plots.
- You can set axis labels, titles, legends, colors, line styles, and more:

```
plt.plot(x, y, color='red', linestyle='--', marker='o', label='Data Points')
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Customized Line Plot')
plt.legend()
plt.grid(True)
plt.show()
```

4.Multiple Plots:

- You can create multiple subplots within a single figure using ‘ **subplot’ .**
- This allows you to display multiple plots side by side or in a grid:

```
plt.subplot(2, 2, 1)  # 2x2 grid, first plot
plt.plot(x, y, 'r--')
plt.subplot(2, 2, 2)  # 2x2 grid, second plot
plt.scatter(x, y, color='green')
plt.subplot(2, 2, 3)  # 2x2 grid, third plot
plt.bar(x, y, color='blue')
plt.subplot(2, 2, 4)  # 2x2 grid, fourth plot
plt.hist(y, bins=5, color='purple')
plt.show()
```

5. Save Plots:

- You can save your Matplotlib plots to various file formats, such as PNG, JPEG, PDF, or SVG, using the ‘**savefig**’ function:

```
plt.plot(x, y)
plt.savefig('my_plot.png')
```

6. Additional Customization:

- Matplotlib provides many more customization options and features for creating advanced visualizations.
- You can explore topics like color mapping, 3D plotting, annotations, and more in the Matplotlib documentation and tutorials.

Plotting with pandas and seaborn

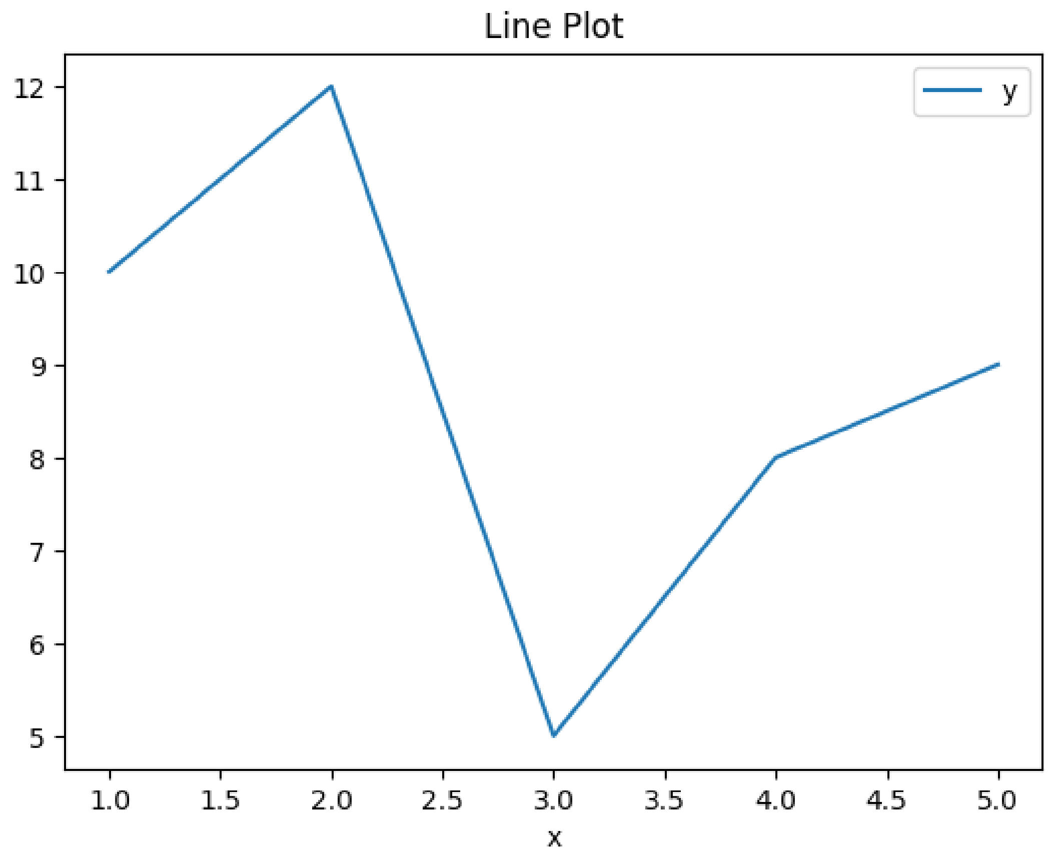
- Pandas and Seaborn are two powerful Python libraries that work well together for data visualization.
- Pandas allows you to manipulate and analyze data, while Seaborn provides a high-level interface for creating aesthetically pleasing statistical plots.
- Here's how you can use both libraries for data plotting:

Using Pandas for Basic Plotting:

- Pandas includes basic plotting functionality based on Matplotlib.
- You can create simple plots directly from Pandas DataFrames and Series.

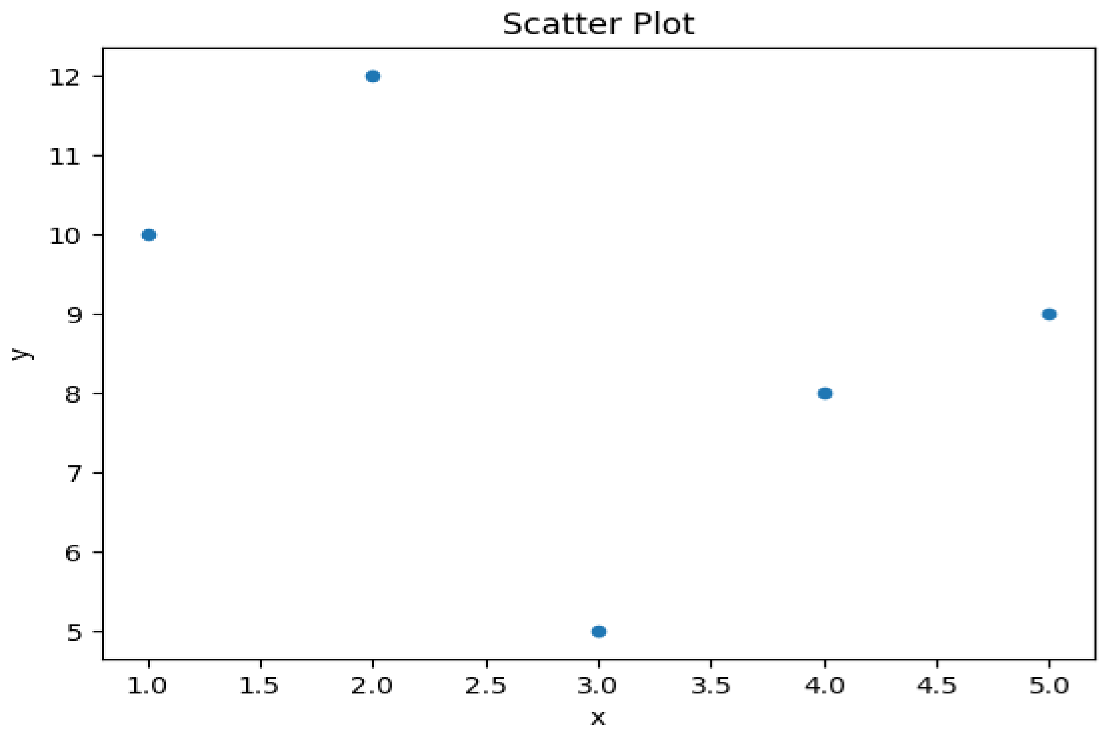
1.Line Plot:

```
import pandas as pd
# Create a Pandas DataFrame
data = {'x': [1, 2, 3, 4, 5], 'y': [10, 12, 5, 8, 9]}
df = pd.DataFrame(data)
# Plot a line chart
df.plot(x='x', y='y', title='Line Plot')
```



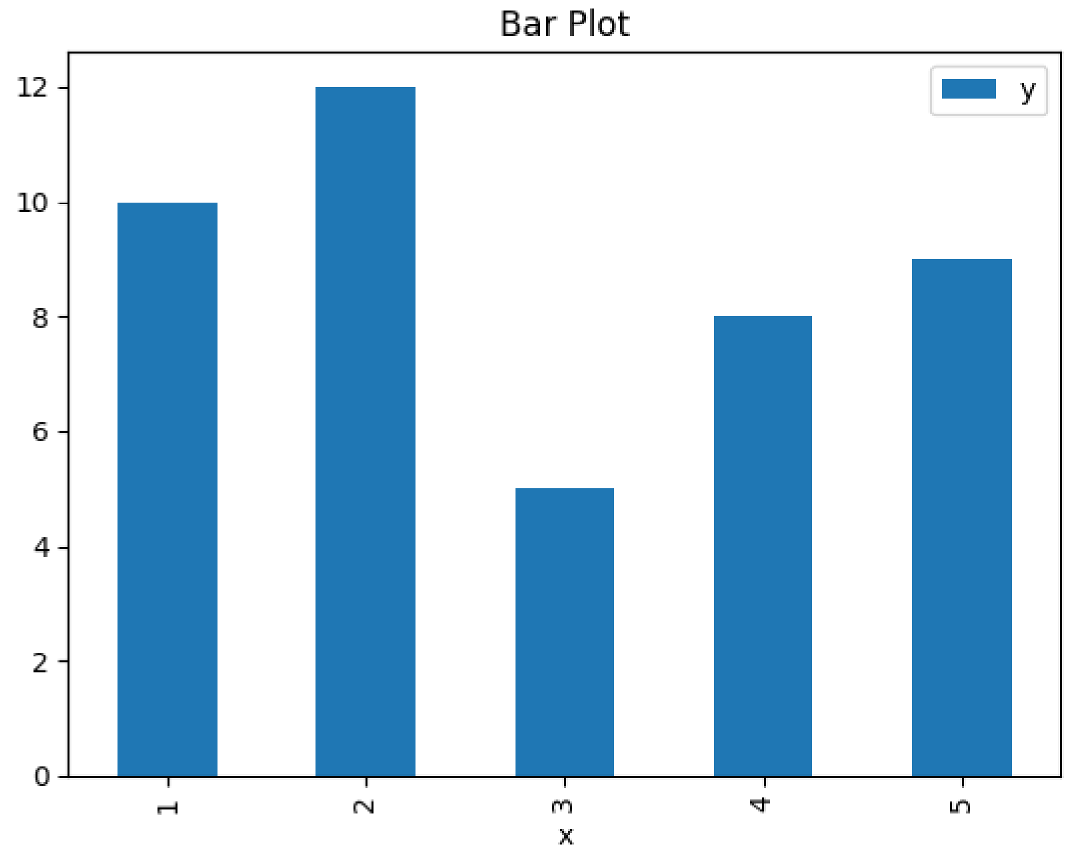
2.Scatter Plot:

```
# Create a scatter plot
df.plot(x='x', y='y', kind='scatter', title='Scatter Plot')
```



3.Bar Plot:

```
# Create a bar chart
df.plot(x='x', y='y', kind='bar', title='Bar Plot')
```



Using Seaborn for Enhanced Visualization:

- Seaborn is known for its ability to create visually appealing and informative statistical visualizations.
- You can use Seaborn in conjunction with Pandas for more advanced plotting.

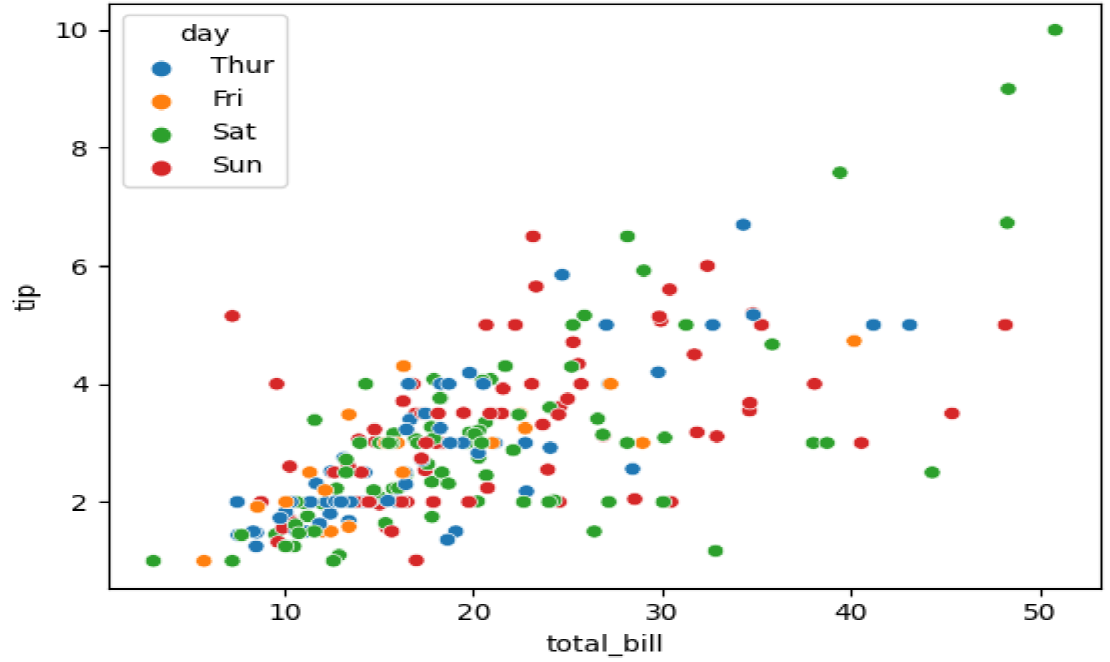
1.Installing Seaborn:

- If you haven't already installed Seaborn, you can do so with **‘pip’.**

```
pip install seaborn
```

2.Using Seaborn:

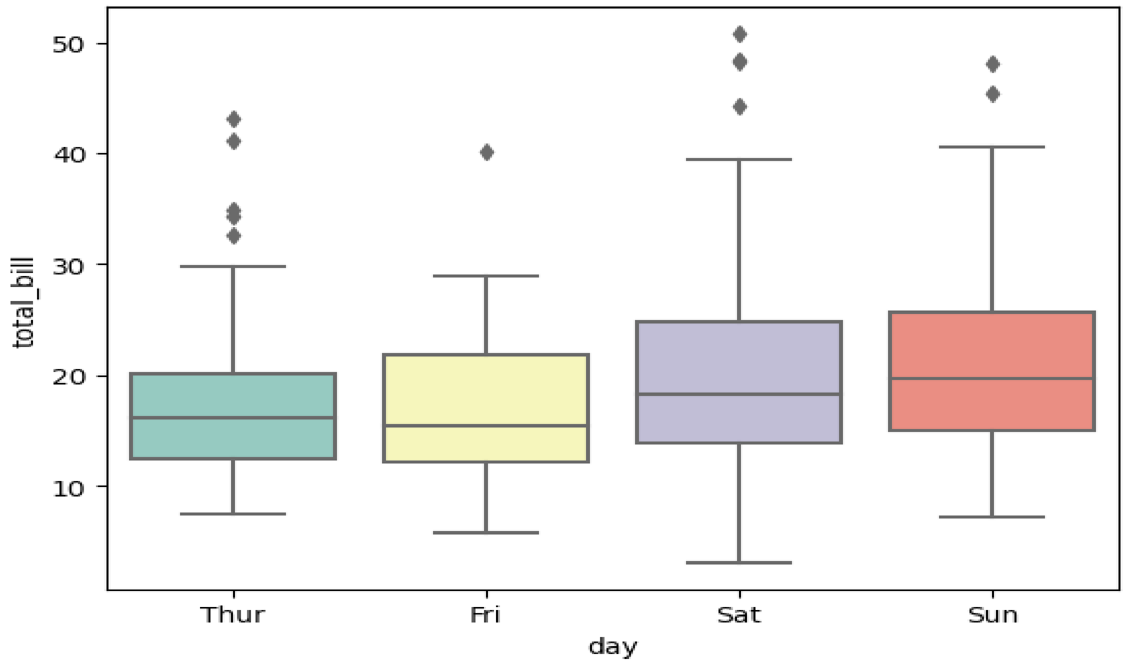
```
import seaborn as sns
# Load a sample dataset from Seaborn
tips = sns.load_dataset("tips")
# Create a scatter plot with Seaborn
sns.scatterplot(data=tips, x="total_bill", y="tip", hue="day")
```



3. Customizing Seaborn Plots:

- Seaborn offers various functions for creating different types of plots, such as **‘sns.lineplot()’ ,’ sns.barplot()’ and ‘sns.histplot()’.**
- You can customize these plots with Seaborn's aesthetic options.

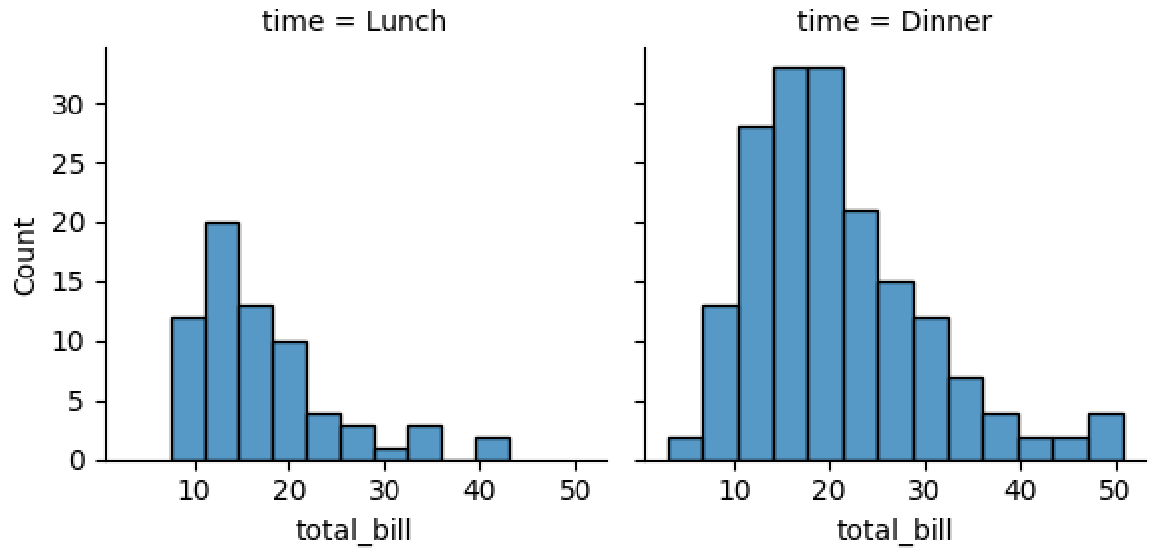
```
# Create a box plot with custom colors
sns.boxplot(data=tips, x="day", y="total_bill", palette="Set3")
```



4.FacetGrids:

- Seaborn's ‘**FacetGrid’** allows you to create grids of plots based on subsets of your data.

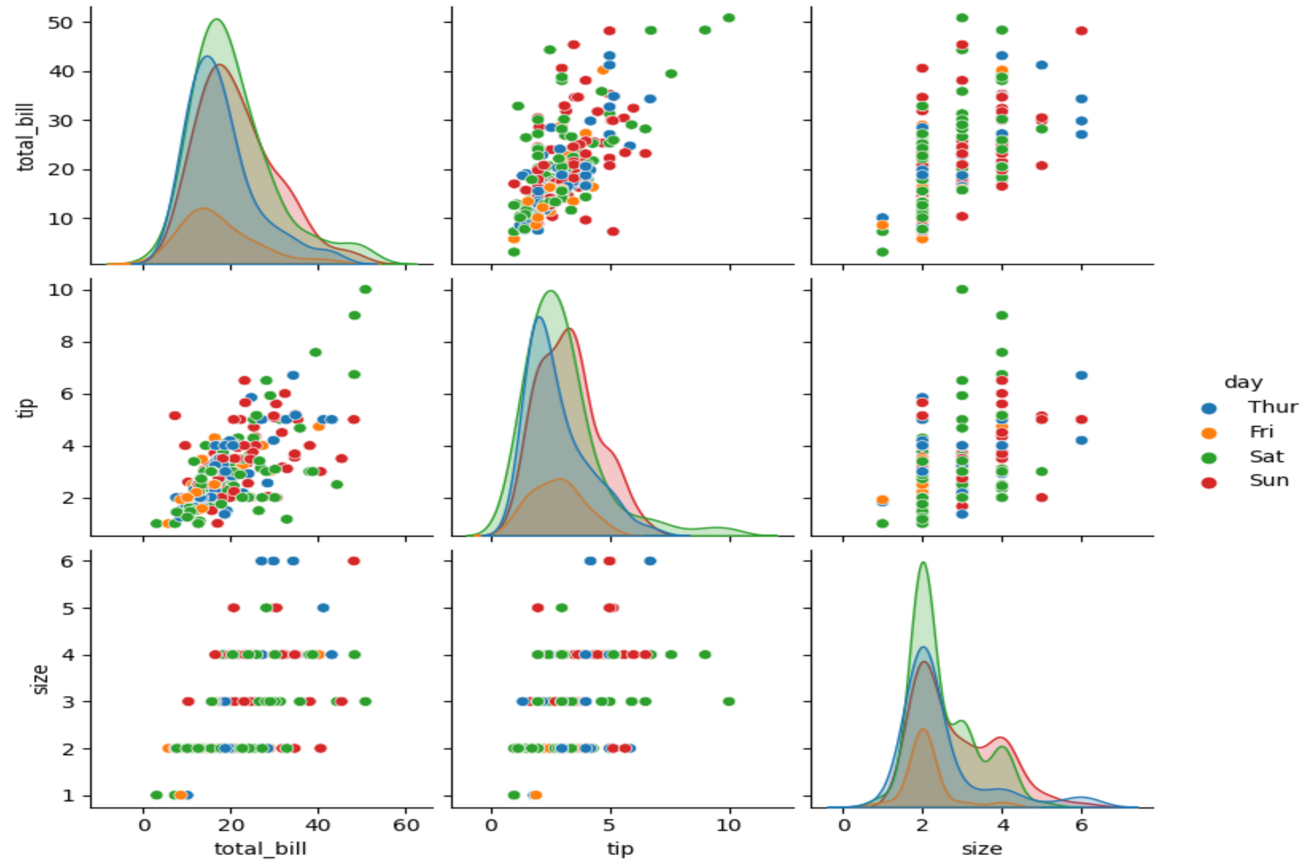
```
# Create a FacetGrid of histograms
g = sns.FacetGrid(tips, col="time")
g.map(sns.histplot, "total_bill")
```



5.Pair Plots:

- Pair plots are excellent for visualizing relationships between multiple variables.

```
# Create a pair plot
sns.pairplot(data=tips, hue="day")
```



Other Python Visualization tools

- In addition to Matplotlib and Seaborn, there are several other Python visualization tools and libraries that cater to specific use cases, offer unique features, or focus on interactive visualizations.
- Here are some notable alternatives:

1.Plotly:

- Plotly is a versatile library for creating interactive and web-based visualizations. It supports various chart types, including scatter plots, line charts, bar charts, heatmaps, and 3D plots. Plotly can be used in Jupyter Notebooks, Dash web applications, and as a JavaScript library.
- Website: https://plotly.com/python/
- Notable Feature: Highly interactive visualizations with tooltips, zoom, and pan capabilities.

2.Bokeh:

- Bokeh is another interactive plotting library that targets web browsers for rendering. It's well-suited for creating interactive dashboards and applications. Bokeh provides high-level and low-level APIs for creating various types of visualizations.
- Website: Bokeh
- Notable Feature: Supports server applications for real-time updates and interactivity.

3. Altair:

- Altair is a declarative statistical visualization library that generates Vega-Lite JSON specifications. It's designed to be user-friendly and intuitive, making it easy to create complex visualizations with minimal code.
- Website: https://altair-viz.github.io/
- Notable Feature: Declarative syntax simplifies the creation of complex visualizations.

4. Holoviews:

- Holoviews is a library for creating interactive, composable visualizations using a high-level, declarative syntax. It integrates with Bokeh, Matplotlib, and other plotting libraries.
- Website: https://holoviews.org/
- Notable Feature: Supports easy composition of visualizations and dashboards.

5. Geopandas:

- Geopandas extends the capabilities of Pandas to geospatial data. It allows you to work with geospatial datasets, perform spatial operations, and create maps and geospatial visualizations.
- Website: https://geopandas.org/en/stable/
- Notable Feature: Integrates seamlessly with Pandas for geospatial data manipulation.

6. **Folium:**

- Folium is a Python library that makes it easy to create interactive leaflet maps for web applications. It's particularly useful for visualizing geographical data and adding interactive maps to web-based projects.
- Website: Folium
- Notable Feature: Generates interactive maps with various basemap styles.

7. Mayavi:

- Mayavi is a 3D scientific data visualization tool that's particularly useful for visualizing complex 3D datasets and simulations. It's often used in scientific and engineering domains.
- Website: https://docs.enthought.com/mayavi/mayavi/
- Notable Feature: Supports 3D plotting and visualization with advanced features.

8. NetworkX:

- NetworkX is a library for creating and analyzing complex networks and graphs. It provides tools for visualizing network structures and analyzing graph data.
- Website: NetworkX
- Notable Feature: Supports the creation and analysis of complex networks and graphs.

Data Visualization Through Their Graph Representations:

- Data visualization through graph representations is a powerful way to explore and understand complex data structures, especially when dealing with network or graph data.
- Graph layout techniques play a crucial role in arranging the nodes and edges of a graph in a visually meaningful and informative way. Let's explore the relationship between data and graphs, as well as common graph layout techniques:

Data and Graphs:

- **Graphs as Data Structures:** In the context of data visualization, a graph is a data structure that consists of nodes (vertices) and edges (connections) that represent relationships between the nodes.
- Nodes can represent entities, while edges represent relationships or connections between entities.
- Graphs are used to model various real-world scenarios, such as social networks, transportation networks, and knowledge graphs.
- **Graph Data Representation:** Graph data can be represented in various ways, including adjacency matrices, adjacency lists, and edge lists.
- These representations determine how data is stored and accessed, which can impact the efficiency of graph algorithms and visualizations.

Graph Layout Techniques:

1. **Force-Directed Layout:** This technique treats the graph as a physical system, where nodes repel each other, and edges act as springs attracting connected nodes. The system reaches an equilibrium, and nodes settle into positions that minimize energy. Force-directed layouts often result in visually pleasing graphs with well-separated nodes.
2. **Hierarchical Layout:** Hierarchical layout organizes nodes into levels or layers based on their relationships. It's often used for tree-like structures or flowcharts. Hierarchical layouts help visualize hierarchical relationships and dependencies.
3. **Circular Layout:** In a circular layout, nodes are placed around a circle, and edges connect nodes in a radial manner. This layout is useful for displaying cyclic or symmetric structures, such as social network connections.
4. **Kamada-Kawai Layout:** This is another force-directed layout algorithm that optimizes the positions of nodes in a graph based on the length of edges and their weights. It tends to produce graphs with evenly distributed nodes and clear connections.
5. **Fruchterman-Reingold Layout:** Similar to force-directed layouts, this algorithm minimizes the overlap between nodes and edges by simulating attractive forces between connected nodes and repulsive forces between all nodes. It's effective for visualizing small to medium-sized graphs.

6. **Grid Layout:** In a grid layout, nodes are arranged in a grid-like structure. While it may not work well for all types of graphs, it can be useful for regular or structured graphs.
7. **Spectral Layout:** Spectral layout techniques use eigenvalues and eigenvectors of the graph's adjacency matrix or Laplacian matrix to determine node positions. These techniques are often used in clustering and community detection.
8. **Layered Layout:** Layered layouts are commonly used for directed acyclic graphs (DAGs) and hierarchical data structures. Nodes are organized into layers, and edges flow from one layer to the next. This layout helps visualize dependencies and control flow.
9. **Orthogonal Layout:** In orthogonal layouts, edges are represented as sequences of horizontal and vertical segments. This technique is used for circuit diagrams and flowcharts.

Force-directed Techniques Multidimensional Scaling

- Force-directed techniques and Multidimensional Scaling (MDS) are two distinct approaches used in data visualization and dimensionality reduction, often employed for various purposes like graph layout or exploring high-dimensional data. Let's explore each of these techniques:

Force-Directed Techniques:

- Force-directed techniques are a class of algorithms used to visualize graphs or networks by simulating physical forces between nodes (vertices) and edges (links) of the graph.
- The primary idea is to treat nodes as charged particles and edges as springs or repulsive forces.
- By iteratively adjusting the positions of nodes based on these forces, the algorithm aims to find a layout that minimizes the total energy of the system, resulting in an aesthetically pleasing and informative graph representation.

Some common force-directed algorithms include:

1. **Spring Embedders (e.g., Fruchterman-Reingold algorithm):** Nodes connected by edges exert attractive forces like springs, causing them to cluster together, while nodes repel each other to prevent overlapping.
2. **Repulsive-Attractive Algorithms (e.g., Kamada-Kawai algorithm):** These algorithms balance attractive forces between adjacent nodes and repulsive forces between all nodes to find an equilibrium position.
3. Force-directed techniques are commonly used for graph visualization, social network analysis, and network science, as they create visually meaningful layouts that reveal the structure and connections within complex networks.

Multidimensional Scaling (MDS):

- Multidimensional Scaling is a dimensionality reduction technique used to visualize high-dimensional data in lower-dimensional spaces (usually 2D or 3D) while preserving the pairwise distances between data points as closely as possible.
- MDS is not limited to graph data but can be applied to various types of data, such as distance matrices or similarity data.

The key steps in MDS are as follows:

1. **Compute Distance/Similarity Matrix:** Start with a distance or similarity matrix that describes the relationships between data points. This matrix could be based on Euclidean distances, correlation coefficients, or any other appropriate measure.
2. **Choose Dimensionality:** Decide on the desired lower-dimensional space (usually 2D or 3D) in which you want to visualize the data.
3. **Optimize Positions:** Use optimization techniques (such as classical MDS, metric MDS, or non-metric MDS) to find the positions of data points in the lower-dimensional space that best approximate the distances or similarities from the original matrix.
4. **Plot the Result:** Plot the data points in the lower-dimensional space, where the positions have been determined by the MDS algorithm. This visualization helps reveal patterns and relationships in the data.

Applications:

- MDS is often used in fields like psychology, marketing, geography, and biology to visualize and analyze similarity or dissimilarity data when the dimensionality of the data is too high to interpret easily.

The Pulling Under Constraints Model

- The concept of "pulling under constraints" in the context of data visualization or dimensionality reduction typically refers to a technique used to manipulate or adjust the positions of data points (or objects) in a lower-dimensional space while follow to certain constraints.
- These constraints could be based on user preferences, data characteristics, or specific requirements of the visualization or analysis task.
- This approach is often used in multidimensional scaling (MDS) or force-directed graph layout algorithms to fine-tune the positions of objects.

Here's a simplified overview of the process:

1. **Initial Configuration:** Start with an initial configuration of data points in a lower-dimensional space. This configuration can be generated using techniques like classical MDS, metric MDS, or a force-directed layout algorithm.
2. **Define Constraints:** Specify the constraints that need to be satisfied during the adjustment process. These constraints can take various forms, such as fixing the position of specific data points, maintaining specific distances or angles between data points, or ensuring that certain data points are grouped together.
3. **Optimization:** Use an optimization algorithm to adjust the positions of the data points while minimizing or maximizing an objective function that incorporates the constraints. The objective function could be designed to penalize deviations from the desired constraints.
4. **Iterative Process:** The adjustment process is often iterative, meaning that the positions of data points are refined incrementally to approach a solution that satisfies the constraints as closely as possible.
5. **Final Visualization:** Once the optimization process converges or meets a stopping criterion, the final configuration of data points is used for visualization or further analysis. This configuration should adhere to the specified constraints while providing a meaningful representation of the data.

Constraints can vary widely depending on the specific application. For example:

1. In graph layout, constraints might involve ensuring that certain nodes are placed in fixed positions or that edges maintain a certain length or curvature.
2. In dimensionality reduction, constraints might involve preserving pairwise distances or angles between data points.
3. In geographic visualization, constraints might include keeping specific locations at fixed coordinates while adjusting the positions of other points.

Bipartite Graphs

- A bipartite graph, also known as a bigraph, is a type of graph in which the set of vertices (nodes) can be divided into two distinct and non-overlapping sets such that every edge connects a vertex from one set to a vertex in the other set.
- In other words, there are no edges that connect vertices within the same set. This property defines the bipartite nature of the graph.

Here are some key characteristics and concepts associated with bipartite graphs:

- **Bipartite Sets:** The two sets of vertices in a bipartite graph are often referred to as "Left" and "Right," but they can be named differently depending on the context. The goal is to partition the vertices in such a way that all edges connect vertices from one set to the other.
- **Vertices and Edges:** A bipartite graph consists of two types of vertices: those in the first set and those in the second set. Edges only exist between vertices of different sets. Mathematically, if you have two sets of vertices, U and V, then a bipartite graph can be represented as G = (U, V, E), where E is the set of edges.
- **Applications:** Bipartite graphs are commonly used to represent relationships between two different types of entities. Some real-world examples include:
  - Matching students to schools based on preferences.
  - Modeling relationships between actors and movies in a recommendation system.
  - Analyzing the co-authorship network of researchers, where one set represents authors, and the other represents papers.

- **Bipartite Matching:** One important problem associated with bipartite graphs is finding a maximum bipartite matching. A matching in a bipartite graph is a set of edges with no common vertices. A maximum bipartite matching is a matching with the maximum number of edges, and it has various applications, such as job assignments or resource allocation.
- **Bipartite Projection:** In some cases, you might want to analyze the relationships within each set independently. Bipartite projection is a technique where you create two separate graphs from a bipartite graph by projecting the relationships within each set. For example, in a co-authorship network, you can project authors into one graph and papers into another.
- **Testing Bipartiteness:** It's possible to determine if a graph is bipartite using algorithms such as depth-first search (DFS). If during the traversal of the graph, you encounter an odd cycle (a cycle with an odd number of vertices), the graph cannot be bipartite.

THANK YOU