Data Wrangling

Data Wrangling (also called **Data Munging**) is the process of **cleaning, transforming, and organizing raw data** into a usable format for analysis and visualization.

It involves handling missing values, merging datasets, reshaping tables, and managing complex data structures.. Therefore, it must be **cleaned, transformed, and structured** into a proper analytical format — this process is known as **Data Wrangling** (also called **Data Munging**).

Main Objectives:

1. To make data consistent and structured.
2. To remove noise, missing values, and duplication.
3. To merge multiple sources of data into one dataset.
4. To prepare data for analysis, modeling, or visualization.

Key Steps in Data Wrangling Process:

1. **Data Collection** – Gathering data from various sources (CSV, Excel, APIs, etc.).
2. **Data Cleaning** – Handling missing or inconsistent data.
3. **Data Transformation** – Changing data format, type, or layout.
4. **Data Combination** – Merging multiple datasets.
5. **Data Reshaping and Structuring** – Organizing data into meaningful structures.
6. **Data Validation** – Ensuring the transformed data is correct and ready for analysis.

Hierarchical Indexing (MultiIndexing)

Hierarchical Indexing, also known as **MultiIndexing**, is a feature in **Pandas** that allows a **DataFrame** or **Series** to have **multiple index levels** (two or more) for both **rows** and/or **columns**. It helps in representing and working with **higher-dimensional data** (like 3D or 4D) in a **two-dimensional tabular format**.

Purpose and Importance:

- It helps organize complex data efficiently.
- Allows grouping of data logically based on multiple keys (e.g., City and Year).
- Makes it easier to perform advanced data selection, filtering, and analysis.
- Reduces redundancy by structuring related data together.
- Simplifies pivoting, reshaping, and aggregating data.

In a normal DataFrame, we have **a single index** for identifying rows (like a row number or a column name). With **Hierarchical Indexing**, we can use **two or more columns as index levels**, which creates a **tree-like structure** of indexing.

Each level of the index can be accessed independently or together. This provides more flexibility while working with multi-level categorical data, such as sales by city and year, region and department, or country and product.

Example:

City	Year	Sales
Pune	2023	12000
Pune	2024	15000
Mumbai	2023	18000
Mumbai	2024	21000

Here, **City** and **Year** together act as a **hierarchical (multi-level) index**.

- **Level 1 Index:** City (Pune, Mumbai)
- **Level 2 Index:** Year (2023, 2024)

This structure allows operations like:

- Accessing all data for a particular city.
- Selecting data for a particular year within a city.
- Performing grouped calculations more easily.

```python
import pandas as pd

# Creating data
data = {
    'Sales': [12000, 15000, 18000, 21000]
}

# Creating hierarchical index
index = pd.MultiIndex.from_tuples(
    [('Pune', 2023), ('Pune', 2024), ('Mumbai', 2023), ('Mumbai', 2024)],
    names=['City', 'Year']
)

# Creating DataFrame
df = pd.DataFrame(data, index=index)

print(df)
```

2. Combining and Merging Datasets

In data analysis, information often comes from multiple sources such as **CSV files, Excel sheets, or database tables**. To perform meaningful analysis, these separate datasets must be **combined or merged** into one structured dataset.

This process is known as **Combining and Merging Datasets**, which is a **core step in data wrangling**.

2. Definition

Combining and Merging Datasets refers to integrating multiple data tables into a single dataset using a **common key, index, or structure** so that data from different sources can be analyzed together.

3. Purpose

- To bring together data from different systems.
- To enrich data by adding additional attributes.
- To prepare a complete dataset for analysis or visualization.
- To reduce redundancy and ensure consistency.

4. Major Techniques of Combining and Merging

There are **three major techniques** to combine datasets in Python (especially using the **pandas** library):

1. **Merging**
2. **Joining**
3. **Concatenation**

(a) Merging

Merging is similar to **SQL joins**. It combines two or more datasets based on a **common column (key)** such as Customer_ID, Student_ID, or Product_Code.

Syntax:

pd.merge(left, right, on='key', how='join_type')

- **left, right:** DataFrames to merge
- **on:** The common column or key
- **how:** Type of join — *inner, outer, left,* or *right*

Types of Merging

Type	Description	Result
Inner Join	Keeps only matching records from both datasets	Common values only
Left Join	Keeps all records from left and matching from right	Left + matched right
Right Join	Keeps all records from right and matching from left	Right + matched left
Outer Join	Keeps all records from both; fills missing with NaN	Union of both

Example

Dataset 1: Sales

Customer_ID	Product	Amount
101	Laptop	60000
102	Mouse	800
103	Printer	4000

Dataset 2: Customers

Customer_ID	Name
101	Riya
102	Arjun
104	Neha

Merging:

pd.merge(sales, customers, on='Customer_ID', how='inner')

Result (Inner Join):

Customer_ID	Product	Amount	Name
101	Laptop	60000	Riya
102	Mouse	800	Arjun

Advantages of Merging

1. Combines data efficiently using key columns.
2. Easy for users familiar with SQL joins.
3. Allows flexible data integration.
4. Preserves structure and meaning of both datasets.

(b) Joining

Joining is similar to merging but uses the **index** of the DataFrame instead of a column key.
It is useful when both datasets share the **same row index**.

Syntax:

df1.join(df2)

Example:

DataFrame 1:

Name	Marks
Riya	85
Arjun	90

DataFrame 2:

Name	City
Riya	Pune
Arjun	Mumbai

Joining:

df1.join(df2)

Result:

Name	Marks	City
Riya	85	Pune
Arjun	90	Mumbai

Advantages of Joining

1. Simple and fast when datasets share the same index.
2. Reduces the need for additional key columns.
3. Works well with **time-series** and **hierarchical data**.

(c) Concatenation

Concatenation stacks datasets **vertically (rows)** or **horizontally (columns)**.
It is useful when datasets have the same structure or schema.

Syntax:

```
pd.concat([df1, df2], axis=0)   # Vertical stacking (rows)
pd.concat([df1, df2], axis=1)   # Horizontal stacking (columns)
```

Advantages of Concatenation

1. Combines multiple datasets easily.
2. Handles both **row-wise** and **column-wise** combinations.
3. Maintains dataset structure.
4. Ideal for appending **monthly or yearly** data.

Example Use Cases

- Combining **monthly sales reports** into one dataset.
- Merging **customer info** with **purchase history**.

- Appending **new year data** to existing records.

Reshaping and Pivoting

Data collected from different sources can appear in various **formats or orientations**.
For effective analysis, data often needs to be **reshaped** or **pivoted** into the desired structure.

These operations allow easy **conversion between "long" and "wide" formats**, helping analysts summarize and visualize data efficiently.

Reshaping means changing the structure or layout of data (rows and columns).
Pivoting means rotating or reorganizing data so that unique values become new rows or columns.

Purpose:

1. To convert data between long and wide formats.
2. To summarize or aggregate data by specific categories.
3. To prepare data for visualization or machine learning.
4. To simplify multi-dimensional data handling.

Reshaping Operations

(a) Stack()

- Converts **columns into rows**.
- Changes data from **wide to long** format.

df.stack()

Example:

City	2023	2024
Pune	12000	15000

After stack():

City	Year	Sales
Pune	2023	12000
Pune	2024	15000

(b) Unstack()

- Converts **rows into columns**.
- Changes data from **long to wide** format.

df.unstack()

Example:

City	Year	Sales
Pune	2023	12000
Pune	2024	15000

After unstack():

City	2023	2024
Pune	12000	15000

(c) Melt()

- Converts data from **wide to long format** by unpivoting columns.
- Helps when multiple column names should become values in a single column.

pd.melt(df, id_vars=['City'], value_vars=['2023', '2024'])

Result:

City	variable	value
Pune	2023	12000
Pune	2024	15000

Pivoting Operations

(a) Pivot()

- Reorganizes data by turning unique column values into new columns.
- Converts **long to wide** format.

df.pivot(index='City', columns='Year', values='Sales')

Before Pivot:

City	Year	Sales
Pune	2023	12000
Pune	2024	15000

After Pivot:

City	2023	2024
Pune	12000	15000

(b) Pivot Table

- Similar to Excel pivot tables.
- Summarizes data using aggregation functions like sum, mean, count, etc.

df.pivot_table(values='Sales', index='City', columns='Year', aggfunc='sum')

Result:

City	2023	2024
Pune	12000	15000

Advantages of Reshaping and Pivoting:

1. Converts data between different structures easily.
2. Simplifies summarization and grouping.
3. Enhances readability and visualization readiness.
4. Supports reporting and dashboard generation.
5. Helps analyze time-series or multi-year data.

Applications:

1. Converting sales data per month to per region format.
2. Preparing data for visualization in matplotlib or seaborn.
3. Creating business intelligence summary tables.
4. Transforming experimental data for statistical analysis.

Data Visualization using matplotlib

Data Visualization is the graphical representation of data and information using charts, graphs, and plots.

- It helps to **understand patterns, relationships, and trends** in data quickly and effectively.
- In Python, the **matplotlib** library is the most fundamental and widely used visualization tool.
- It provides full control to create **high-quality 2D and limited 3D graphics**.

2. What is matplotlib?

- **matplotlib** is a **low-level Python library** used to create static, animated, and interactive visualizations.
- It is similar to MATLAB's plotting system and forms the base for higher-level libraries like **pandas.plot()** and **seaborn**.
- It works well with **NumPy arrays**, **pandas DataFrames**, and other Python data tools.

3. Components of matplotlib

Matplotlib is structured in a hierarchical way. The main components are:

Component	Description
Figure	The entire plotting window or canvas. Can contain multiple subplots.
Axes	The area where data is actually plotted (each subplot is an Axes).
Axis	Controls the x and y limits, ticks, and labels.
Artist	All visual elements (lines, text, legends, titles, etc.) drawn on the Axes.

4. Importing and Basic Syntax

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

The **pyplot** module provides functions similar to MATLAB commands.

5. Basic Plotting Functions

Function	Description
plt.plot(x, y)	Line plot (default)
plt.scatter(x, y)	Scatter plot
plt.bar(x, height)	Vertical bar chart
plt.barh(y, width)	Horizontal bar chart
plt.hist(data)	Histogram
plt.boxplot(data)	Box plot
plt.pie(data)	Pie chart
plt.xlabel(), plt.ylabel()	Label axes
plt.title()	Add title
plt.legend()	Display legend
plt.grid()	Add grid lines
plt.show()	Display the plot

6. Example: Simple Line Plot

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 14, 18, 20, 25]

plt.plot(x, y, color='blue', marker='o', linestyle='--', label='Growth')
plt.title('Sales Growth Over 5 Years')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.legend()
plt.grid(True)
plt.show()
```

9. Advantages of matplotlib

1. **Highly customizable:** Fine control over every part of the plot.
2. **Supports multiple plot types:** line, bar, scatter, histogram, pie, etc.
3. **Integration:** Works smoothly with pandas, NumPy, and Jupyter Notebook.
4. **Publication-quality figures:** Suitable for scientific papers and reports.
5. **Extensible:** Base library for advanced tools like seaborn, plotly, etc.

10. Limitations

- Syntax can be **verbose** for complex plots.
- Default styles are **plain** (need customization for better look).
- **Interactive** and **animated** plots are limited compared to modern tools (Plotly, Bokeh).

11. Applications of matplotlib

1. Exploratory Data Analysis (EDA)
2. Statistical or scientific plotting
3. Visualizing trends, patterns, and correlations
4. Machine Learning model evaluation (accuracy curves, confusion matrix)
5. Data presentation in dashboards or reports

Plotting with pandas and seaborn

1. Introduction

- Data visualization is an essential part of **data analysis** — it helps to summarize data and identify trends.
- While **matplotlib** is the base library, **pandas** and **seaborn** provide **higher-level and easier-to-use interfaces** for visualization.
- Both integrate directly with **DataFrames**, making them ideal for exploratory data analysis (EDA).

A. Plotting with pandas

2. Overview

- **pandas** has a built-in .plot() method that uses **matplotlib** internally.
- It provides a **simple syntax** to plot directly from DataFrame or Series objects.
- Most common plot types like **line**, **bar**, **histogram**, **pie**, and **scatter** are supported.

3. Syntax

```
DataFrame.plot(kind='plot_type', ...)
```

Common kind values:

Plot Type	Description
'line'	Default; shows trends over time
'bar' or 'barh'	Bar charts (vertical/horizontal)
'hist'	Histogram for distribution
'box'	Box plot for spread/outliers
'pie'	Pie chart for composition
'scatter'	Relationship between two variables
'area'	Area plot for cumulative values

4. Example: Simple pandas plot

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

data = {'Year': [2021, 2022, 2023, 2024],
```

```
    'Sales': [250, 300, 350, 420]}
df = pd.DataFrame(data)

df.plot(x='Year', y='Sales', kind='line', marker='o', color='green', title='Annual Sales Growth')
plt.show()
```

Explanation:

- Creates a line plot directly from DataFrame.
- No need to call plt.plot() manually.
- Integrates with DataFrame labels automatically.

5. Advantages of pandas plotting

1. Very **simple syntax** compared to pure matplotlib.
2. Works directly with **DataFrames and Series**.
3. Automatically handles **labels and legends**.
4. Ideal for **quick visual inspection** during data analysis.
5. Supports **multiple columns** in one command.

6. Example: Multiple column plots

```
df[['Sales', 'Profit']].plot(kind='bar')
```

This draws grouped bar charts for multiple columns.

B. Plotting with seaborn

7. Overview

- **seaborn** is a **high-level visualization library** built on top of matplotlib.
- It provides **beautiful, informative, and statistical graphics** with minimal code.
- Works seamlessly with pandas DataFrames and supports **automatic color themes** and **statistical plotting**.

8. Import and Basic Use

```
import seaborn as sns
import matplotlib.pyplot as plt
```

Seaborn works directly with DataFrame columns:

```
sns.lineplot(data=df, x='Year', y='Sales')
plt.show()
```

9. Common seaborn Functions

Function	Type of Plot	Description
sns.lineplot()	Line Plot	Trends over time
sns.barplot()	Bar Plot	Mean comparison of categories
sns.countplot()	Count Plot	Frequency of categorical values
sns.boxplot()	Box Plot	Spread and outliers
sns.violinplot()	Violin Plot	Distribution + density
sns.scatterplot()	Scatter Plot	Relationship between two variables
sns.heatmap()	Heatmap	Correlation or matrix visualization
sns.pairplot()	Pair Plot	Multiple scatter plots for variable pairs
sns.distplot() / sns.histplot()	Distribution Plot	Data distribution and density curve

10. Example: Seaborn Bar Plot

```
import seaborn as sns
import pandas as pd

data = {'Product': ['A', 'B', 'C', 'D'],
        'Sales': [250, 300, 400, 350]}
df = pd.DataFrame(data)

sns.barplot(x='Product', y='Sales', data=df, palette='viridis')
plt.title("Sales by Product")
plt.show()
```

11. Advantages of seaborn

1. High-level interface – minimal code for complex plots.
2. Built-in **statistical support** (mean, confidence intervals).
3. Attractive **default color schemes and themes**.
4. Integration with **pandas** and **NumPy**.
5. Provides advanced plots: **heatmaps, pair plots, joint plots**

12. Difference between matplotlib, pandas, and seaborn

Feature	matplotlib	pandas	seaborn
Level	Low-level	Mid-level	High-level
Ease of use	Moderate	Easy	Very easy
Integration with DataFrame	Manual	Direct	Direct
Plot Appearance	Basic	Basic	Attractive
Statistical Features	Limited	None	Advanced
Customization	Maximum	Moderate	Good

Other Python Visualization Tools

- Besides **matplotlib**, **pandas**, and **seaborn**, Python offers many **modern visualization libraries** that provide **interactive, web-based, and real-time graphics**.
- These tools help data scientists and analysts create **dynamic dashboards**, **3D visualizations**, and **interactive charts** suitable for business and research.

2. Need for Advanced Visualization Tools

- Matplotlib and Seaborn are mostly **static** (non-interactive).
- Modern data applications need **interactive and customizable visuals**.
- Other tools like **Plotly, Bokeh, Altair, Dash, and ggplot** fill this gap by offering **web-based, dynamic visualization** features.

3. Major Python Visualization Tools

(a) Plotly

- **Plotly** is an open-source library for **interactive and web-based visualizations**.
- It supports a wide range of charts: line, scatter, 3D, heatmaps, maps, etc.
- Works with **Python, R, JavaScript**, and integrates with **Dash** for dashboard development.

Example:

```
import plotly.express as px
df = px.data.gapminder()
px.scatter(df, x="gdpPercap", y="lifeExp", color="continent",
        size="pop", hover_name="country", animation_frame="year")
```

Features:

- Interactive zoom, hover, and filter.

- 2D and 3D plots with animations.
- Excellent for **web dashboards** and **data presentations**.

Use Case: Visualizing global economic and demographic data dynamically.

(b) Bokeh

- **Bokeh** is another library designed for **interactive visualizations** in **web browsers**.
- It uses **JavaScript under the hood** and can produce dashboards and streaming data visuals.

Example:

```
from bokeh.plotting import figure, show
p = figure(title="Simple Line Plot")
p.line([1,2,3,4,5], [6,7,2,4,5], line_width=2)
show(p)
```

Features:

- Highly **interactive and web-embeddable**.
- Supports **large datasets and streaming**.
- Can be linked with **Flask or Django** web apps.

Use Case: Real-time monitoring dashboards and interactive reports.

(c) Altair

- **Altair** is a **declarative statistical visualization library** based on the **Vega-Lite** grammar.
- It allows users to describe *what* they want to visualize, not *how* to draw it.

Example:

```
import altair as alt
from vega_datasets import data
cars = data.cars()
alt.Chart(cars).mark_point().encode(x='Horsepower', y='Miles_per_Gallon', color='Origin')
```

Features:

- Simple, concise, and elegant syntax.
- Integrates perfectly with **pandas DataFrames**.
- Best suited for **exploratory data analysis (EDA)**.

Use Case: Correlation visualization between multiple attributes (e.g., car horsepower vs mileage).

(d) Dash

- **Dash** is a framework built on **Plotly** for creating **interactive web dashboards** using pure Python.
- It combines **visualization and web development** without requiring JavaScript.

Features:

- Used for **data analytics dashboards**.
- Interactive filters, dropdowns, sliders.
- Integration with **Plotly charts**.

Use Case: Building real-time analytics dashboards for business reporting.

Data Visualization Through Their Graph Representations

In data visualization, **graphs** (or networks) are used to represent **relationships between entities**.A **graph** is a structure consisting of **nodes (vertices)** and **edges (links)** connecting them. Graph-based visualization helps in understanding **connections, dependencies, flows, or hierarchies** in complex datasets.

2. Importance of Graph-Based Visualization

- Graphs are widely used to visualize **social networks, transportation systems, web structures, biological networks**, etc.
- They provide **intuitive and relational views** of data, making hidden patterns visible.
- Useful in both **qualitative** (structure understanding) and **quantitative** (measuring relationships) analysis.

3. Components of a Graph

Component	Description
Nodes (Vertices)	Represent entities or data points (e.g., people, cities, servers).
Edges (Links)	Represent relationships or connections between nodes.
Weights	Represent strength, cost, or capacity of a connection.
Direction	Defines one-way or two-way relationships (Directed or Undirected Graphs).

4. Types of Graphs in Visualization

Type	Description	Example
Undirected Graph	Links have no direction.	Friendship network.
Directed Graph (Digraph)	Links have direction (A → B).	Web page links or workflow.
Weighted Graph	Edges have weights or costs.	Road distances or network latency.
Bipartite Graph	Two sets of nodes; edges only between sets.	Students–Courses or Buyers–Products.
Hierarchical Graph	Shows parent-child relationships.	Organizational charts, file systems.

Graph Layout Techniques

Graph layout techniques determine **how nodes and edges are positioned visually** in 2D or 3D space. The goal is to make the graph **clear, readable, and meaningful**, revealing relationships effectively. A good layout minimizes **edge crossings**, **overlapping nodes**, and **distortion**.

Objectives of Graph Layout Techniques

1. Improve **readability** and **aesthetics** of the graph.
2. Show **clusters or communities** clearly.
3. Minimize **edge crossing** and overlapping nodes.
4. Preserve **important structural properties** (hierarchy, distance, similarity).
5. Support **interactive and scalable visualization**.

7. Common Graph Layout Techniques

Technique	Description	Example / Use
Force-directed Layout	Positions nodes by simulating physical forces: edges attract, nodes repel.	Social networks, citation networks.

Technique	Description	Example / Use
Circular Layout	Places nodes around a circle; emphasizes cyclic or symmetric relationships.	Communication patterns, ring topologies.
Hierarchical Layout	Arranges nodes in layers to show direction or dependency.	Organizational charts, flow diagrams.
Grid Layout	Nodes are placed in a grid pattern; easy for small structured datasets.	Tabular or spatial networks.
Random Layout	Nodes placed randomly; used for testing or initial visualization.	Random graphs or simulations.
Geographical Layout	Node positions determined by real-world coordinates (latitude/longitude).	Transport or location-based networks.

9. Advantages of Graph Representation

1. Clear visualization of **relationships and dependencies**.
2. Helps in **network analysis** and **pattern detection**.
3. Suitable for **hierarchical and dynamic systems**.
4. Works well for **social, biological, and computer networks**.
5. Easy integration with data science tools (Python, R, etc.).

10. Applications

- **Social Network Analysis** – relationships among users.
- **Recommendation Systems** – connections between users and products.
- **Biological Networks** – protein interaction networks.
- **Transportation Networks** – route optimization and connectivity.
- **Web Graphs** – page linking and search optimization.

Force-Directed Techniques

Force-directed techniques are one of the most widely used methods for **graph layout visualization**.

- They are **physics-based algorithms** that simulate forces acting on the nodes and edges of a graph to find a visually appealing arrangement.
- The main idea is to position connected nodes close together while keeping unrelated nodes apart, creating a **natural-looking graph structure**.

2. Basic Concept

- The technique models a graph as a **physical system**:
  - **Nodes** act like charged particles that **repel** each other.
  - **Edges** act like **springs** that pull connected nodes **closer together**.
- The system is simulated iteratively until it reaches a **state of equilibrium** (minimum energy), where forces are balanced.

3. Forces Involved

Force Type	Description	Effect
Attractive Force (Spring Force)	Acts along edges, pulling connected nodes together.	Keeps related nodes close.
Repulsive Force (Coulomb Force)	Acts between all pairs of nodes, pushing them apart.	Prevents node overlap.
Damping / Cooling	Gradually reduces movement over iterations.	Helps layout stabilize.

4. Working Principle

1. **Initialization:**
   Nodes are placed randomly in the 2D or 3D space.
2. **Force Calculation:**
   Compute attractive and repulsive forces for every node.
3. **Position Update:**
   Move nodes slightly in the direction of net force.
4. **Cooling:**
   Gradually reduce movement (like temperature cooling in physics).
5. **Equilibrium:**
   When total system energy is minimized, layout becomes stable and balanced.

5. Mathematical Model (Simplified)

Let:

- $F_a(d) = k \times \log(\frac{d}{l})$ → attractive force
- $F_r(d) = \frac{k^2}{d}$ → repulsive force

Where:

- d = distance between nodes
- k = ideal distance constant
- l = preferred edge length

These formulas ensure that:

- Too close nodes repel.
- Too far nodes attract.

7. Advantages of Force-Directed Layouts

1. **Aesthetic and Intuitive:**
   Produces visually pleasing and natural graph structures.
2. **Automatic Clustering:**
   Related nodes naturally group together.
3. **Applicable to Any Graph:**
   Works for both directed and undirected graphs.
4. **Interactive and Dynamic:**
   Can adapt as new nodes/edges are added (used in real-time visualization).
5. **Dimensional Flexibility:**
   Works in both 2D and 3D visualizations.

8. Limitations

1. **Computationally Expensive:**
   Requires $O(n^2)$ calculations for repulsion among n nodes.
2. **May Get Stuck in Local Minima:**
   Layout can stabilize in a suboptimal configuration.
3. **Not Ideal for Very Large Graphs:**
   Visualization can become cluttered or slow.
4. **Random Initialization Sensitivity:**
   Different runs may give slightly different layouts.

10. Applications

- **Social Network Analysis** – visualize friendships or follower connections.
- **Biological Networks** – protein–gene interactions.
- **Web Graphs** – linking structures between websites.
- **Citation and Collaboration Networks** – researcher or publication relationships.
- **Knowledge Graphs** – entity relationship representation.

Multidimensional Scaling (MDS)

Multidimensional Scaling (MDS) is a **data visualization technique** used to represent high-dimensional data in **a lower-dimensional space (usually 2D or 3D)** while preserving the **relative distances or similarities** between data points.

- It helps to visualize how similar or different objects are, based on a given **distance (dissimilarity) matrix**.
- In short:
  ☞ MDS converts complex relationships into a **geometric map** where similar items appear close and dissimilar items appear far apart.

2. Purpose of MDS

- To visualize **hidden structures** in complex datasets.
- To **reduce dimensionality** while maintaining inter-object distances.
- To discover **clusters, patterns, or trends** among data points.
- Often used when the data is **non-numeric** but pairwise dissimilarities are known.

3. Basic Concept

1. Start with a **distance matrix (D)** that measures how far apart each pair of objects is.
2. MDS places each object as a **point in a geometric space** (2D or 3D).
3. The algorithm tries to preserve the **original distances** in this new space.
4. The final layout is such that the **spatial distances** between points represent their **original similarities/dissimilarities**.

4. Types of MDS

Type	Description	Example Use
Metric MDS	Preserves actual numerical distances between objects.	Quantitative data (e.g., Euclidean distance)
Non-Metric MDS	Preserves the **rank order** of distances (ordinal relationships).	Qualitative data (e.g., preferences, survey data)

5. Steps in MDS Process

1. **Data Preparation:**
   Prepare a dissimilarity (distance) matrix between all object pairs.
2. **Initial Configuration:**
   Randomly assign coordinates to all points in a low-dimensional space.
3. **Distance Computation:**
   Compute distances between points in the new space.
4. **Stress Calculation:**
   Measure how different the computed distances are from the original distances.
   The difference is called **stress** or **strain**.
   $$Stress = \sqrt{\frac{\sum (d_{ij} - \hat{d}_{ij})^2}{\sum d_{ij}^2}}$$
   where:
   - d_{ij} = original distance between objects i and j
   - $\hat{d}_{ij}$ = distance in the new space
5. **Optimization:**
   Adjust point positions to minimize stress (using iterative algorithms).
6. **Visualization:**
   Once stress is minimal, plot the 2D or 3D map for analysis.

7. Advantages of MDS

1. **Visualizes high-dimensional data** in 2D/3D easily.
2. **Reveals patterns or clusters** that may not be visible in raw data.
3. Works with **distance or similarity data** (even non-numeric).
4. **Flexible** — can be applied to psychological, biological, or business data.
5. Helps in **feature reduction** and exploratory data analysis.

8. Limitations of MDS

1. **Computationally expensive** for large datasets ($O(n^2)$).
2. May **lose some information** while reducing dimensions.
3. **Interpretation can be subjective** (depends on scaling).
4. Sensitive to **noise or errors** in the distance matrix.
5. **Non-unique solutions** — different initializations can give slightly different results.

9. Applications of MDS

Domain	Application
Market Research	Visualizing customer preferences or brand similarity.
Psychology	Mapping similarity of emotions or behaviors.
Bioinformatics	Visualizing genetic or protein similarity.
Recommendation Systems	Representing similarity between users/items.
Social Networks	Visualizing relationships among people or entities.

1. The Pulling Under Constraints Model (PUC Model)

The **Pulling Under Constraints (PUC) Model** is a **graph layout technique** used in **data and network visualization**.

It positions nodes (data points) in a 2D or 3D space while **satisfying specific geometric or logical constraints**.

The model is based on **force-directed principles**, where nodes "pull" or "repel" each other under certain conditions, but movement is restricted by **constraints** such as distance, hierarchy, or boundaries.

2. Concept

- Each node in the graph experiences forces:
  - **Attractive forces** – between connected nodes (edges act like springs).
  - **Repulsive forces** – between unconnected nodes (to prevent overlap).
- The layout stabilizes when total energy (sum of all forces) is minimal.
- In PUC, these forces are **subject to constraints**, e.g.:
  - Fixed distances between certain nodes.
  - Nodes must remain within a region or hierarchy level.
  - Maintaining symmetry or specific geometric shapes.

3. Working Principle

1. **Initialization:**
   Nodes are placed randomly in a plane.
2. **Force Computation:**
   Attractive and repulsive forces are calculated for each node.
3. **Constraint Application:**
   Movement of nodes is adjusted to obey predefined constraints (e.g., distance limits, boundary boxes, fixed anchor nodes).
4. **Energy Minimization:**
   The system iteratively adjusts positions until the total energy is minimal.
5. **Final Layout:**
   Nodes are positioned clearly and meaningfully without violating constraints.

4. Advantages

- Produces **aesthetic and readable** graph layouts.
- Maintains **important relationships** while respecting constraints.
- Useful for **hierarchical or spatial data**.

- Avoids **node overlap** and preserves **relative distances**.

5. Applications

Domain	Example Use
Social Networks	Constraining nodes to represent specific groups or communities.
Biological Networks	Keeping proteins or genes in predefined clusters.
Geographical Visualization	Placing nodes within bounded regions like countries or cities.
Engineering/Systems Design	Keeping components fixed while adjusting connections.

6. Limitations

- Computationally heavy for large graphs.
- May require manual tuning of constraint parameters.
- Can get stuck in local minima (non-optimal layouts).

Bipartite Graphs

A **Bipartite Graph** is a **special type of graph** where **nodes (vertices)** are divided into **two distinct sets** — such that **edges only connect nodes from different sets**, never within the same set.

Represented as **G = (U, V, E)** where:

  - **U** and **V** are disjoint sets of vertices
  - **E** is the set of edges connecting nodes from U → V

A graph is said to be *bipartite* if its vertices can be divided into two disjoint sets **U** and **V** such that every edge connects a vertex from **U** to a vertex from **V**.

Example:
U = {Students}, V = {Courses}, E = {(student, course) enrolled in}

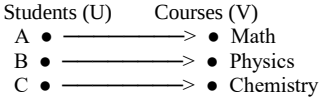
3. Properties

1. No edges exist between vertices of the same set.
2. A graph is bipartite **if and only if it has no odd-length cycles**.
3. Can be represented using a **bi-adjacency matrix**.
4. Often drawn as two parallel columns: left for set U, right for set V.

4. Visualization

- In data visualization, bipartite graphs are used to represent **two-mode relationships** (two different entity types).
- Nodes are positioned in two groups, and edges (lines) show connections between them.

Example Diagram (conceptual):



5. Applications

Domain	Use
Recommendation Systems	Users ↔ Products relationships (e.g., Netflix, Amazon).
Social Networks	People ↔ Groups or Actors ↔ Movies.
Text Mining	Words ↔ Documents representation.
Biology	Proteins ↔ Chemical compounds interactions.
Project Management	Employees ↔ Tasks allocation.

6. Variations

- **Weighted Bipartite Graphs:** Edges have weights (e.g., rating scores).
- **Directed Bipartite Graphs:** Edges have direction (e.g., user → item).
- **Bipartite Network Projection:** Converts bipartite graphs into single-mode networks by connecting nodes that share common neighbors.

7. Advantages

- Simplifies representation of two-mode data.
- Useful for **matching problems** and **network flow analysis**.
- Clear and interpretable visualization.
- Supports mathematical algorithms (e.g., Hungarian algorithm for matching).

8. Limitations

- Cannot represent single-type relationships (only two-mode).
- Layout may become **cluttered for large graphs**.
- Interpretation can be difficult if there are too many edges.