Data Pre-processing

These are the steps for data pre-processing:

Step 1: Import Necessary Libraries
Step 2: Read the Dataset
Step 3: Sanity Check of Data
Step 4: Exploratory Data Analysis (EDA)
Step 5: Missing Value Treatments
Step 6: Outlier Treatments
Step 7: Duplicates and Garbage Value Treatments
Step 8: Encoding of Data
Step 9: Feature Scaling (Normalization or Standardization)
Step 10: Feature Selection

Step-1: Importing Necessary Liberaries

In [2]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Step-2: Read the Data Set

Reading the Dataset in Different Formats

In data analysis, the first step is to import the dataset into the Python environment. Python, especially with the help of the pandas library, provides simple functions to read data from various file formats.

Common formats include CSV, Excel, JSON, text files, SQL databases, and more. Choosing the correct method to read a dataset depends on the source and structure of the data.

CSV files are commonly used due to their simplicity and compatibility. Excel files are also widely used, especially in business and academic settings. JSON is useful for structured data, especially from APIs or web sources. SQL databases are used for storing large and relational data.

Reading data accurately is essential for further cleaning, exploration, and modeling. Using the right function for the correct file format ensures data is properly loaded into a DataFrame for analysis.

In [3]:
```
# Load the dataset into a DataFrame (df)
# Ensure the CSV file is in your current working directory, or provide full path
# This dataset contains global life expectancy data

df = pd.read_csv("Life Expectancy Data.csv")
```

In [ ]:
```
# Display the first few rows of the dataset
# 'head()' helps to understand the structure, columns, and sample values of the data

# This setting ensures all columns are visible in the output (helpful for wide datasets)
pd.set_option('display.max_columns', None)

# Display the first five rows of the DataFrame
df.head()
```

In [ ]:
```
# tail of the data (last five rows of the data set))
df.tail()
```

Step-3: Sanity Check of the Data

What is a Sanity Check of the Data?

A sanity check in data preprocessing refers to a quick initial examination of the dataset to make sure it looks reasonable, consistent, and usable before you proceed to deeper analysis or modeling.

3.1 Shape of the Data Set

The shape of a dataset refers to its dimensions, specifically:

1. Number of rows (samples): Each row usually represents one observation or data point.
2. Number of columns (features): Each column represents a variable or attribute.

In [ ]:
```
# Check the shape of the DataFrame (rows, columns)
# Returns a tuple: (number of rows, number of columns)
df.shape
```

3.2 Information of the Data Set

The info of a dataset provides a concise summary of the DataFrame. It includes:

1. Total number of entries (rows)
2. Column names
3. Number of non-null values in each column
4. Data types of each column (e.g., int, float, object)
5. Memory usage

In [ ]:
```
# Display a summary of the DataFrame
# Shows number of non-null values, data types, and memory usage for each column
df.info
```

3.3 Checking Data Types in Data Set

Each column in a dataset has a data type, such as integer, float, object (string), or boolean.

Knowing the data types helps in:

1. Selecting appropriate preprocessing techniques
2. Avoiding errors during analysis or modeling
3. Understanding which columns are numerical and which are categorical

In [ ]:
```
# Display the data type of each column in the DataFrame
# Helps identify which columns are numeric, categorical (object), or date/time types
df.dtypes
```

3.4 Checking for Missing Data in Data Set

Missing data refers to the absence of values in one or more cells of a dataset. In Python (Pandas), missing values are usually represented as NaN, which stands for "Not a Number".

Before performing any analysis or applying machine learning models, it is important to check for missing data. Most algorithms cannot handle missing values directly, and they may produce errors or inaccurate results if such values are present.

Missing data can occur due to various reasons such as:

1. Human error during data entry
2. Sensor or equipment failure
3. Data not collected or lost

Identifying missing values allows us to take appropriate actions such as:

1. Removing rows or columns with too many missing values
2. Filling missing values using statistical techniques (mean, median, mode)
3. Using advanced imputation methods like KNN or regression models

In Pandas, functions like `isnull()` and `sum()` are commonly used to detect missing values. Visualizations such as heatmaps can also help to quickly identify patterns of missing data across the dataset.

Proper handling of missing data is a key step in data preprocessing that ensures the quality and reliability of the final model or analysis.

In [ ]:
```
# Check for missing (null) values in each column
# Returns the total number of missing values per column
df.isnull().sum()
```

In [ ]:
```
# Checking for Percentage of missing values in each column
df.isnull().sum()/df.shape[0]*100
```

3.5 Checking for Duplicate Values

Duplicate values in a dataset are rows that are exactly the same in all or selected columns. These can occur due to repeated data collection, system errors, or improper data entry.

Duplicates can negatively affect data analysis and model performance by:

1. Introducing bias in the results
2. Skewing summary statistics
3. Increasing the size of the dataset unnecessarily

Identifying and removing duplicate records is an important step in the data cleaning process.

Why Checking Duplicates is Important:

1. Ensures each observation is unique
2. Prevents redundancy in training data
3. Improves model accuracy and fairness

In Pandas, the following functions are used to detect and handle duplicate values:

1. `df.duplicated()` : Returns a Boolean Series indicating which rows are duplicates.
2. `df.duplicated().sum()` : Returns the total number of duplicate rows.
3. `df.drop_duplicates()` : Removes duplicate rows from the dataset.

You can also check for duplicates based on specific columns by passing column names to these functions.

Removing duplicates helps keep the dataset clean, consistent, and ready for accurate analysis or machine learning modeling.

In [ ]:
```
# Check for duplicate rows in the DataFrame
# Returns the total number of rows that are exact duplicates of earlier rows
df.duplicated().sum()
```

3.6 Identifying Garbage Values in a Data Set

Garbage values refer to incorrect, meaningless, or inconsistent data entries that do not follow the expected format or valid range. These values can occur due to human errors, data corruption, improper data collection, or incorrect encoding.

Examples of garbage values include:

- Negative values in age or salary columns
- Strings like N/A, ?, unknown, or random characters in numeric columns
- Inconsistent category names such as Male, male, MALE
- Out-of-range values (e.g., test scores above 100)

Why Identifying Garbage Values is Important

- Garbage values can distort analysis and reduce the accuracy of machine learning models
- They may lead to wrong statistical summaries
- They can cause errors during model training and predictions

How to Identify Garbage Values

1. Value Counts
   Use `df['column_name'].value_counts()` to see all unique values in a column and spot invalid entries.
2. Describe and Summary Statistics
   Use `df.describe()` to check for unusual minimum, maximum, or mean values.
3. Range Checks
   Manually check if numeric values fall within acceptable limits.
4. Regular Expressions
   Use pattern matching to identify unexpected characters or strings in a column.
5. Conditional Filters
   Use logical conditions (e.g., `df[df['Age'] < 0]`) to detect entries that violate business rules.

Handling Garbage Values

- Replace invalid values with NaN and impute them
- Normalize or standardize inconsistent formats
- Remove rows that contain irreparable garbage data

Cleaning garbage values is an essential part of data preprocessing to ensure the dataset is valid, clean, and ready for meaningful

analysis.

In [ ]:
```
# Loop through each categorical (object-type) column in the DataFrame
# and print the count of unique values in each column
for i in df.select_dtypes(include="object").columns:
    print (df[i].value_counts())
```

Step-4: Exploratory Data Analysis

4.1 Descriptive Statistics

Exploratory Data Analysis (EDA) is the process of examining and understanding the dataset before applying any machine learning or statistical models. It helps identify patterns, detect anomalies, check assumptions, and uncover relationships between variables.

EDA includes:

- Checking data structure and types
- Handling missing and duplicate values
- Descriptive statistics (mean, median, mode, etc.)
- Visualizing data using plots and charts
- Understanding distributions and correlations

EDA is an essential step to ensure the data is clean, meaningful, and ready for analysis or modeling.

In [ ]:
```
# df.describe() generates descriptive statistics for all numeric columns,
# including count, mean, standard deviation, min, max, and quartiles (25%, 50%, 75%)
# .T transposes the output to make columns as rows — easier to read when there are many features
df.describe().T
```

In [ ]:
```
# Incase we want to know about the decriptive statistics of objects we give this command.
```

In [ ]:
```
df.describe(include="object")
```

4.2: Histogram to Understand the Distribution

A histogram is a graphical representation used to understand the distribution of a single numerical variable. It displays the data using bars, where each bar represents a range of values (called a bin), and the height of the bar shows the number of observations in that range.

Histograms help to:

- Understand the shape of the distribution (normal, skewed, uniform, etc.)
- Identify the central tendency of the data
- Detect the spread or variability of values
- Spot outliers or unusual values

It is one of the most common tools in exploratory data analysis.

In [ ]:
```
# This code loops through all numerical columns in the dataset and plots their histograms.
# Purpose:
# Understand the distribution of each numeric feature
# Detect skewness, outliers, or unusual patterns
# Useful for deciding on normalization, transformations, or outlier treatment
# Notes:
# We use seaborn's histplot for clear visuals
# KDE (Kernel Density Estimate) is added for a smooth distribution curve
# plt.show() is used to separate each plot for better readability
# import warnings
# warnings.filterwarnings("ignore")
# The above command is incase we do not want to show warnings

for i in df.select_dtypes(include="number").columns:
    sns.histplot(data=df,x=i)
    plt.show()
```

4.3: Box Plot to Identify Outliers

A box plot is a graphical representation that shows the distribution of a numerical variable through its quartiles. It is useful for detecting outliers and understanding the spread of the data.

The box plot displays:

- Minimum value (excluding outliers)
- First quartile (Q1)
- Median (Q2)
- Third quartile (Q3)
- Maximum value (excluding outliers)
- Outliers as individual points beyond the whiskers

Outliers are typically defined as values that fall below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR, where IQR is the interquartile range (Q3 - Q1).

In [ ]:
```
# This code loops through all numeric columns in the DataFrame and plots a boxplot for each one.
# Purpose:
# Visualize the distribution of numerical features
# Easily detect outliers using the IQR (Interquartile Range) method
# Understand the spread, skewness, and central tendency of the data
# Notes:
# Boxplots show the median, quartiles, and outliers
# plt.show() is used to display each plot separately for clarity
for i in df.select_dtypes(include="number").columns:
    sns.boxplot(data=df,x=i)
    plt.show()
```

In [ ]:
```
#Display the data types of all columns in the DataFrame
df.dtypes
```

4.4 Ploting of Scatter Plot to Identify Relation Among Input Parameters and Output or Target

Scatter Plot to Identify Relationship Among Input Parameters and Target:

A scatter plot is a graphical tool used to visualize the relationship between two numerical variables. In the context of machine learning or data analysis, it helps to identify how an input (independent) variable relates to an output (dependent or target) variable.

Each point in the plot represents one observation in the dataset, with its position determined by the values of the two variables being compared.

Why Use a Scatter Plot?

- To detect linear or non-linear relationships between variables
- To identify trends or patterns in data
- To observe clusters or gaps
- To detect outliers
- To support feature selection by examining correlation with the target

In [ ]:
```
#Scatter Plots for Each Numeric Feature vs. Life Expectancy(Target)

# This loop creates scatter plots between each selected numeric feature and the target variable: 'Life expectan
# Purpose:
# Visually explore relationships and patterns (linear, non-linear, no correlation)
# Detect potential trends, clusters, or outliers
# Helps decide which features may have predictive value

# Notes:
# sns.scatterplot() plots a feature on x-axis and 'Life expectancy' on y-axis
# plt.show() ensures each plot is shown separately
for i in ['Year', 'Adult Mortality', 'infant deaths',
       'Alcohol', 'percentage expenditure', 'Hepatitis B', 'Measles ', ' BMI ',
       'under-five deaths ', 'Polio', 'Total expenditure', 'Diphtheria ',
       ' HIV/AIDS', 'GDP', 'Population', ' thinness  1-19 years',
       ' thinness 5-9 years', 'Income composition of resources', 'Schooling']:
    sns.scatterplot(data=df, x=i, y='Life expectancy')
    plt.show()
```

4.5 Correlation with Heat Map to Interpret the Relation and Multicolinearity

Correlation is a statistical measure that describes the strength and direction of a relationship between two numerical variables. It ranges from -1 to 1:

- A value near 1 indicates a strong positive relationship
- A value near -1 indicates a strong negative relationship
- A value near 0 indicates no linear relationship

Multicollinearity refers to the situation where two or more input features are highly correlated with each other. This can cause problems in machine learning models, especially linear regression, by making the model unstable and difficult to interpret.

A heatmap is a visual representation of the correlation matrix, where colors are used to represent the strength of correlations between variables.

Why Use a Correlation Heatmap?

- To understand relationships between all numeric variables
- To detect multicollinearity between features
- To select or remove features based on their correlation
- To improve model performance and avoid redundancy

In [5]:
```
#Correlation Heatmap of Numeric Features

# This code performs the following steps:
# Selects all numeric columns from the DataFrame
# Calculates the pairwise Pearson correlation between numeric features
# Visualizes the correlation matrix as a heatmap using Seaborn
#
# Purpose:
# Identify strong positive or negative relationships between features
# Detect multicollinearity before model building (especially for regression)
# Understand which variables are most related to each other
#
# Notes:
# Correlation values range from -1 (perfect negative) to +1 (perfect positive)
# annot=True adds numeric values to each cell for clarity
# figsize is set to 20x20 for better readability when there are many features
s=df.select_dtypes(include="number").corr()
plt.figure(figsize= (20,20))
sns.heatmap(s,annot=True)
```

Step-5: Missing Values Treatment in Data Set

Missing values occur when no data is stored for a variable in an observation. They can affect data analysis and machine learning models if not handled properly.

Missing data may occur due to data entry errors, sensor failures, skipped questions, or data corruption.

Types of Missing Data

1. Missing Completely at Random (MCAR): Missing values have no relation to any other data.
2. Missing at Random (MAR): Missing values are related to observed data.
3. Missing Not at Random (MNAR): Missing values are related to unobserved data.

Common Techniques to Handle Missing Data

1. Deletion Methods
   - Remove rows with missing values using `df.dropna()`
   - Remove columns with too many missing values using `df.drop(columns)`
2. Imputation Methods
   - Fill with a constant value using `df.fillna(0)` or a custom value
   - Fill with mean, median, or mode of the column
     - `df['column'].fillna(df['column'].mean(), inplace=True)`
     - `df['column'].fillna(df['column'].median(), inplace=True)`
     - `df['column'].fillna(df['column'].mode()[0], inplace=True)`
   - Forward fill or backward fill using `method='ffill'` or `method='bfill'`
   - Use advanced techniques like KNN imputer or regression imputation

Choosing the Right Method

- Use deletion only when the missing portion is very small
- Use mean or median for numerical data
- Use mode for categorical data
- Use advanced imputation for large or complex datasets

Proper handling of missing values is a critical part of data cleaning and ensures reliable analysis and model performance.

In [ ]:
```
# For continious values in data set we use Mean, Median and KNN Imputer to fill the missing values
# For Discrete and catagorical values we use Mode to fill the missing values
```

In [ ]:
```
df.isnull().sum() # Used to check the missing values
```

5.1 Using Median to Replace Missing values is "Alcohol","Polio","Income composition of Resources"

In [4]:
```
# Import the warnings module to manage warning messages
import warnings
# Ignore all warning messages (useful during data cleaning to avoid clutter)
warnings.filterwarnings("ignore")
# The line `df.head()` is commented out; it would normally display the first 5 rows of the DataFrame
df.head()
# Fill missing values (NaN) in specific columns with the median value of that column
for i in ["Alcohol","Polio","Income composition of resources"]:
# Replace NaN values in the column with its median value (in-place modification)
    df[i].fillna(df[i].median(), inplace=True)
```

5.2 Replacing the missing values in "Adult Mortality","Hepatitis B" with mean.

In [5]:
```
# This code fills missing (NaN) values in specific numeric columns using the **mean** of each column.
# It is a common data preprocessing step to handle missing data before analysis or machine learning.
for i in ["Adult Mortality","Hepatitis B"]:
    df[i].fillna(df[i].mean(), inplace=True)
```

5.3 Replacing the remaining missing values with KNN.

In [6]:
```
# KNN (K-Nearest Neighbors) imputation is a method to fill in missing values in your dataset by using the value
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=5)
i = df.select_dtypes(include="number").columns
df[i] = imputer.fit_transform(df[i])
```

In [ ]:
```
df.isnull().sum() # Check if there is still missing value or not.
```

Step-6: Outliers Treatment

Outliers are data points that are significantly different from other observations in the dataset. They can affect the accuracy of statistical analysis and the performance of machine learning models.

Outliers may occur due to data entry errors, measurement mistakes, or genuine variability in the data.

Why Treat Outliers

- They can distort mean and standard deviation
- They can lead to biased or unstable models
- They may affect the overall interpretation of data patterns

Methods to Detect Outliers

1. Using Box Plots
    - Visual method to detect values outside the whiskers
2. Using Z-score
    - Identifies how many standard deviations a data point is from the mean
    - Formula: z = (value - mean) / standard deviation
3. Using IQR (Interquartile Range)
    - Values outside the range: Q1 - 1.5*IQR and Q3 + 1.5*IQR are considered outliers

Methods to Treat Outliers

1. Remove Outliers
    - Drop rows that contain outlier values
2. Replace Outliers
    - Cap values at a certain percentile (Winsorizing)
    - Replace with mean or median of the column
3. Transform Data
    - Apply log, square root, or box-cox transformation to reduce impact
4. Use Robust Models
    - Choose models that are less sensitive to outliers (like decision trees)

In [ ]:
```
# Outlier treatment is typically performed on numerical (continuous or discrete) data, where extreme values can
```

In [ ]:
```
print(df.head())       # View first 5 rows
print(df.columns)      # View all columns
```

In [8]:
```
# Sample function to compute upper and lower whiskers
def wisker(series):
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    return upper, lower

# Ensure all column names are clean
df.columns = df.columns.str.strip()

# List of columns for outlier treatment
columns = ['Life expectancy', 'Adult Mortality',
           'infant deaths', 'percentage expenditure', 'Hepatitis B',
           'Measles', 'under-five deaths', 'Polio', 'Total expenditure',
           'Diphtheria', 'HIV/AIDS', 'GDP', 'Population',
           'thinness  1-19 years', 'thinness 5-9 years', 'Schooling']

# Apply outlier treatment
for col in columns:
    if col in df.columns:  # Make sure column exists
        if pd.api.types.is_numeric_dtype(df[col]):  # Only numeric columns
            uw, lw = wisker(df[col].dropna())  # drop NaNs for whisker calc
            df[col] = df[col].apply(lambda x: lw if x < lw else uw if x > uw else x)
```

In [ ]:
```
df.columns # To Show columns
```

In [ ]:
```
# Check if still there is any outlier present or not.
for i in ['Life expectancy', 'Adult Mortality',
           'infant deaths', 'percentage expenditure', 'Hepatitis B',
           'Measles', 'under-five deaths', 'Polio', 'Total expenditure',
           'Diphtheria', 'HIV/AIDS', 'GDP', 'Population',
           'thinness  1-19 years', 'thinness 5-9 years', 'Schooling']:
    sns.boxplot(df[i])
    plt.show()
```

Step-7: Duplicates and Garbage Value Treatment

Cleaning data involves identifying and handling duplicate entries and garbage values. These issues can affect the accuracy and reliability of data analysis and machine learning models.

In [54]:
```
df.duplicated().sum() # To Check Duplicate Values
```

Out[54]:
```
np.int64(0)
```

In [55]:
```
df = df.drop_duplicates() # If there is any duplicate this command will remove it.

# df = df.drop_duplicates(subset='column_name') # For specific columns
```

Step-8 Encoding of data

What is Encoding in Machine Learning?

Encoding is the process of converting categorical data (like text labels or string values) into a numerical format so that machine learning algorithms can process them. Most ML models require all input features to be numeric, so encoding is essential for converting features like "Male"/"Female" or "Yes"/"No" into numbers.

Method | Description | Suitable For | |---|---|---| | **1. Label Encoding** | Assigns a unique integer to each category (e.g., `"Male" → 0`, `"Female" → 1`) | Ordinal data (where order matters) | | **2. One-Hot Encoding** | Creates binary columns for each category (e.g., `"Red" → [1,0,0]`, `"Blue" → [0,1,0]`) | Nominal data (no order) | | **3. Ordinal Encoding** | Like Label Encoding but with known order (e.g., `"Low" → 1`, `"Medium" → 2`, `"High" → 3`) | Ordered categories | | **4. Binary Encoding** | Converts categories into binary code and splits it into columns | High cardinality (many unique values) | | **5. Frequency Encoding** | Replaces categories with the frequency of their occurrence | Quick but may leak target info | | **6. Target Encoding** | Replaces categories with the mean of the target variable | Risk of overfitting; used with caution | | **7. Hash Encoding** | Uses hashing function to assign numbers | High cardinality with memory constraints |

In [ ]:
```
# This line performs one-hot encoding on the "Country" and "Status" columns of the DataFrame `df`.
```

```
# It creates new binary columns for each unique category in "Country" and "Status",
# but drops the first category in each to avoid the dummy variable trap (multicollinearity).
# The resulting encoded columns are added to the DataFrame, replacing the original ones.

pd.get_dummies(data=df, columns=["Country", "Status"], drop_first=True)
```

Step-9: Feature Scaling (Normalization or Standardization)

Recommended Scaling Methods for Supervised ML Algorithms

Supervised ML Algorithm	Recommended Scaling	Reason
Linear Regression	Standardization	Assumes normal distribution; improves convergence
Logistic Regression	Standardization	Sensitive to feature scale
Support Vector Machine (SVM)	Normalization	Distance-based; scale affects margins
K-Nearest Neighbors (KNN)	Normalization	Distance-based; requires features on same scale
Naive Bayes	No Scaling Needed	Based on probability; scale has no effect
Decision Trees	No Scaling Needed	Splits on feature thresholds; scale doesn't matter
Random Forest	No Scaling Needed	Ensemble of trees; scaling is not required
Gradient Boosting (e.g., XGBoost)	No Scaling Needed	Uses decision trees internally
Neural Networks (MLP, DNN)	Standardization or Normalization	Helps with training stability and faster convergence
Ridge / Lasso Regression	Standardization	Regularization assumes features are on similar scales
ElasticNet	Standardization	Combination of Ridge and Lasso; scaling is important
Linear Discriminant Analysis (LDA)	Standardization	Assumes Gaussian distribution; uses covariance matrix
Quadratic Discriminant Analysis (QDA)	Standardization	Same as LDA; needs scaled inputs

In [ ]:
```
# Normalize using MinMaxScaler
# 1st Method

import pandas as pd
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
for i in df.select_dtypes(include='number').columns:
    df[i] = scaler.fit_transform(df[[i]])
#Display result
df.head
```

In [ ]:
```
# 2nd Method

import pandas as pd
from sklearn.preprocessing import MinMaxScaler

# Step 1: Select only numeric columns from df
numeric_df = df.select_dtypes(include=["number"])

# Step 2: Normalize using MinMaxScaler
scaler = MinMaxScaler()
normalized_data = scaler.fit_transform(numeric_df)

# Step 3: Convert back to DataFrame
normalized_df = pd.DataFrame(normalized_data, columns=numeric_df.columns)

# Step 4: (Optional) Add back non-numeric columns like 'Country' or 'Status'
final_df = pd.concat([df[['Country', 'Status']].reset_index(drop=True), normalized_df], axis=1)

# Step 5: Display result
print(final_df.head())
```

Step-10: Feature Selection

Feature selection is the process of identifying and selecting the most relevant input variables (features) from a dataset that contribute the most to predicting the target variable.

Selecting the right features helps to:

- Improve model accuracy and performance
- Reduce overfitting
- Decrease training time and complexity

- Enhance model interpretability

Feature selection is used to reduce the number of input variables by selecting the most relevant features. This improves model performance and reduces overfitting.

Below are commonly used feature selection techniques in Python:

1. Correlation Matrix
   Identify features that are highly correlated with each other using `df.corr()`. Drop one of the features from highly correlated pairs to reduce redundancy.

2. Univariate Selection
   Use statistical tests to select features that have a strong relationship with the output variable.

   - Example: Chi-Square Test using `SelectKBest` from `sklearn.feature_selection`
3. Recursive Feature Elimination (RFE)
   Selects features by recursively considering smaller sets of features. A model is trained and the least important features are pruned.

   - Available via `RFE` in `sklearn.feature_selection`
4. Feature Importance from Tree-Based Models
   Models like Random Forest and ExtraTrees can calculate feature importance scores.

   - Use `.feature_importances_` to identify top contributing features
5. L1 Regularization (Lasso Regression)
   Performs feature selection by shrinking less important feature coefficients to zero.

6. Variance Threshold
   Removes features with low variance (i.e., features that do not change much across observations).

   - Use `VarianceThreshold` from `sklearn.feature_selection`
7. Mutual Information
   Measures the mutual dependence between two variables.

   - Use `mutual_info_classif` or `mutual_info_regression` from `sklearn.feature_selection`

Choosing the right method depends on the type of problem (classification or regression), dataset size, and model being used.

Note: In Civil Engineering most of the time input parameters ranges from 7-15, so there is not need to select features as all are important. Incase there is large data set where the inputs parameters are very large like 100 then feature selection is handy.

Congratulations! Shahid 😎 Your Data Pre-Processing is Completed!
Now Your Dataset is Ready for Model Training.

In [ ]: