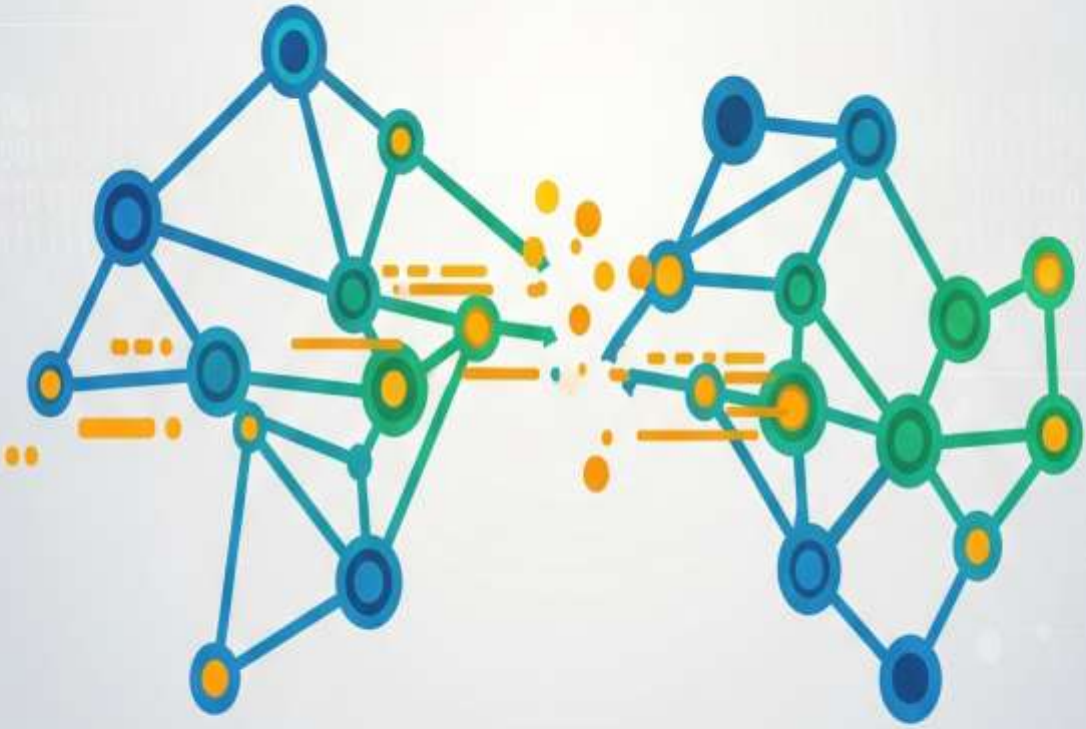
Data Preprocessing



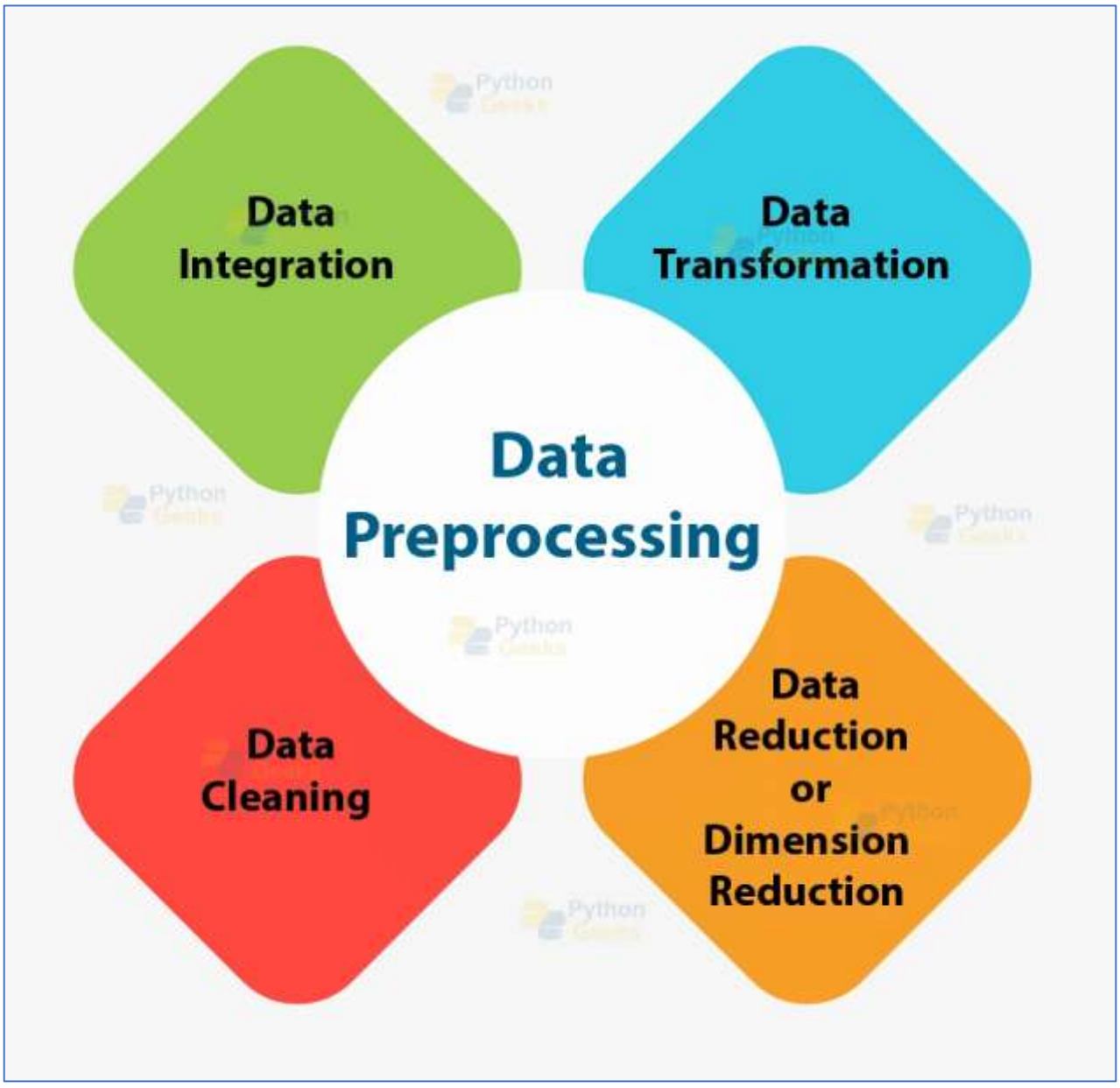
The Art of Refining Raw Information

Data Preprocessing

Data preprocessing involves cleaning and transforming raw data into a usable format. It includes handling missing values, removing duplicates, converting data types and making sure the data is in the right format for accurate results.

- Data Preprocessing
- What is Data Cleaning?
- Handling Missing Data
- Handling outliers
- Data Transformation
- Feature Engineering
- Data Sampling

Data Preprocessing in Data Mining



Data preprocessing is the process of preparing raw data for analysis by cleaning and transforming it into a usable format. In data mining it refers to preparing raw data for mining by performing tasks like cleaning, transforming, and organizing it into a format suitable for mining algorithms.

- Goal is to improve the quality of the data.
- Helps in handling missing values, removing duplicates, and normalizing data.
- Ensures the accuracy and consistency of the dataset.

Steps in Data Preprocessing

Some key steps in data preprocessing are Data Cleaning, Data Integration, Data Transformation, and Data Reduction.

1. Data Cleaning: It is the process of identifying and correcting errors or inconsistencies in the dataset. It involves handling missing values, removing duplicates, and correcting incorrect or outlier data to ensure the dataset is accurate and reliable. Clean data is essential for effective analysis, as it improves the quality of results and enhances the performance of data models.

- **Missing Values:** This occur when data is absent from a dataset. You can either ignore the rows with missing data or fill the gaps manually, with the attribute mean, or by using the most probable value. This ensures the dataset remains accurate and complete for analysis.
- **Noisy Data:** It refers to irrelevant or incorrect data that is difficult for machines to interpret, often caused by errors in data collection or entry. It can be handled in several ways:
    - **Binning Method:** The data is sorted into equal segments, and each segment is smoothed by replacing values with the mean or boundary values.
    - **Regression:** Data can be smoothed by fitting it to a regression function, either linear or multiple, to predict values.
    - **Clustering:** This method groups similar data points together, with outliers either being undetected or falling outside the clusters. These techniques help remove noise and improve data quality.
- **Removing Duplicates:** It involves identifying and eliminating repeated data entries to ensure accuracy and consistency in the dataset. This process prevents errors and ensures reliable analysis by keeping only unique records.

2. Data Integration: It involves merging data from various sources into a single, unified dataset. It can be challenging due to differences in data formats, structures, and meanings. Techniques like record linkage and data fusion help in combining data efficiently, ensuring consistency and accuracy.

- **Record Linkage** is the process of identifying and matching records from different datasets that refer to the same entity, even if they are represented differently. It helps in combining data from various sources by finding corresponding records based on common identifiers or attributes.
- **Data Fusion** involves combining data from multiple sources to create a more comprehensive and accurate dataset. It integrates information that may be inconsistent or incomplete from different sources, ensuring a unified and richer dataset for analysis.

3. Data Transformation: It involves converting data into a format suitable for analysis. Common techniques include normalization, which scales data to a common range; standardization, which adjusts data to have zero mean and unit variance; and discretization, which converts continuous data into discrete categories. These techniques help prepare the data for more accurate analysis.

- **Data Normalization**: The process of scaling data to a common range to ensure consistency across variables.
- **Discretization**: Converting continuous data into discrete categories for easier analysis.
- **Data Aggregation**: Combining multiple data points into a summary form, such as averages or totals, to simplify analysis.
- **Concept Hierarchy Generation**: Organizing data into a hierarchy of concepts to provide a higher-level view for better understanding and analysis.

4. Data Reduction: It reduces the dataset's size while maintaining key information. This can be done through feature selection, which chooses the most relevant features, and feature extraction, which transforms the data into a lower-dimensional space while preserving important details. It uses various reduction techniques such as,

- **Dimensionality Reduction (e.g., Principal Component Analysis)**: A technique that reduces the number of variables in a dataset while retaining its essential information.
- **Numerosity Reduction**: Reducing the number of data points by methods like sampling to simplify the dataset without losing critical patterns.
- **Data Compression**: Reducing the size of data by encoding it in a more compact form, making it easier to store and process.

Uses of Data Preprocessing

Data preprocessing is utilized across various fields to ensure that raw data is transformed into a usable format for analysis and decision-making. Here are some key areas where data preprocessing is applied:

1. Data Warehousing: In data warehousing, preprocessing is essential for cleaning, integrating, and structuring data before it is stored in a centralized repository. This ensures the data is consistent and reliable for future queries and reporting.

2. Data Mining: Data preprocessing in data mining involves cleaning and transforming raw data to make it suitable for analysis. This step is crucial for identifying patterns and extracting insights from large datasets.

3. Machine Learning: In machine learning, preprocessing prepares raw data for model training. This includes handling missing values, normalizing features, encoding categorical variables, and splitting datasets into training and testing sets to improve model performance and accuracy.

4. Data Science: Data preprocessing is a fundamental step in data science projects, ensuring that the data used for analysis or building predictive models is clean, structured, and relevant. It enhances the overall quality of insights derived from the data.

5. Web Mining: In web mining, preprocessing helps analyze web usage logs to extract meaningful user behavior patterns. This can inform marketing strategies and improve user experience through personalized recommendations.

6. Business Intelligence (BI): Preprocessing supports BI by organizing and cleaning data to create dashboards and reports that provide actionable insights for decision-makers.

7. Deep Learning Purpose: Similar to machine learning, deep learning applications require preprocessing to normalize or enhance features of the input data, optimizing model training processes.

Advantages of Data Preprocessing

- **Improved Data Quality**: Ensures data is clean, consistent, and reliable for analysis.
- **Better Model Performance**: Reduces noise and irrelevant data, leading to more accurate predictions and insights.
- **Efficient Data Analysis**: Streamlines data for faster and easier processing.
- **Enhanced Decision-Making**: Provides clear and well-organized data for better business decisions.

Disadvantages of Data Preprocessing

- **Time-Consuming**: Requires significant time and effort to clean, transform, and organize data.

- **Resource-Intensive**: Demands computational power and skilled personnel for complex preprocessing tasks.
- **Potential Data Loss**: Incorrect handling may result in losing valuable information.
- **Complexity**: Handling large datasets or diverse formats can be challenging.

Data Cleaning in ML

Data cleaning is a step in machine learning (ML) which involves identifying and removing any missing, duplicate or irrelevant data. The goal of data cleaning is to ensure that the data is accurate, consistent and free of errors as raw data is often noisy, incomplete and inconsistent which can negatively impact the accuracy of model. Clean datasets also helps in EDA which enhances the interpretability of data so that the right actions can be taken based on insights.

Benefits of Data Cleaning

How to Perform Data Cleaning

The process begins by thoroughly understanding the data and its structure to identify issues like missing values, duplicates and outliers. Performing data cleaning involves a systematic process to identify and remove errors in a dataset. The following steps are essential to perform data cleaning:

- **Remove Unwanted Observations:** Eliminate duplicates, irrelevant entries or redundant data that add noise.
- **Fix Structural Errors:** Standardize data formats and variable types for consistency.
- **Manage Outliers:** Detect and handle extreme values that can skew results, either by removal or transformation.
- **Handle Missing Data:** Address gaps using imputation, deletion or advanced techniques to maintain accuracy and integrity.

Implementation for Data Cleaning

Let's understand each step for Database Cleaning using titanic dataset.

Step 1: Import Libraries and Load Dataset

We will import all the necessary libraries i.e **pandas** and **numpy**.

```
import pandas as pd
import numpy as np

df = pd.read_csv('Titanic-Dataset.csv')
df.info()
df.head()
```

Step 2: Check for Duplicate Rows

df.duplicated(): Returns a boolean Series indicating duplicate rows.

```
df.duplicated()
```

Step 3: Identify Column Data Types

- List comprehension with .dtype attribute to separate categorical and numerical columns.
- **object dtype:** Generally used for text or categorical data.

```
cat_col = [col for col in df.columns if df[col].dtype == 'object']
num_col = [col for col in df.columns if df[col].dtype != 'object']

print('Categorical columns:', cat_col)
print('Numerical columns:', num_col)
```

Step 4: Count Unique Values in the Categorical Columns

df[numeric_columns].nunique(): Returns count of unique values per column.

```
df[cat_col].nunique()
```

Step 5: Calculate Missing Values as Percentage

- **df.isnull():** Detects missing values, returning boolean DataFrame.
- Sum missing across columns, normalize by total rows and multiply by 100.

```
round((df.isnull().sum() / df.shape[0]) * 100, 2)
```

Step 6: Drop Irrelevant or Data-Heavy Missing Columns

- **df.drop(columns=[])**: Drops specified columns from the DataFrame.
- **df.dropna(subset=[])**: Removes rows where specified columns have missing values.
- **fillna()**: Fills missing values with specified value (e.g., mean).

```
df1 = df.drop(columns=['Name', 'Ticket', 'Cabin'])
df1.dropna(subset=['Embarked'], inplace=True)
df1['Age'].fillna(df1['Age'].mean(), inplace=True)
```

Step 7: Detect Outliers with Box Plot

- **matplotlib.pyplot.boxplot():** Displays distribution of data, highlighting median, quartiles and outliers.
- **plt.show()**: Renders the plot.

```
import matplotlib.pyplot as plt

plt.boxplot(df3['Age'], vert=False)
plt.ylabel('Variable')
plt.xlabel('Age')
plt.title('Box Plot')
```

```
plt.show()
```

Step 8: Calculate Outlier Boundaries and Remove Them

- Calculate mean and standard deviation (std) using df['Age'].mean() and df['Age'].std().
- Define bounds as mean ± 2 * std for outlier detection.
- Filter DataFrame rows within bounds using Boolean indexing.

```
mean = df1['Age'].mean()
std = df1['Age'].std()

lower_bound = mean - 2 * std
upper_bound = mean + 2 * std

df2 = df1[(df1['Age'] >= lower_bound) & (df1['Age'] <= upper_bound)]
```

Step 9: Impute Missing Data Again if Any

fillna() applied again on filtered data to handle any remaining missing values.

```
df3 = df2.fillna(df2['Age'].mean())
df3.isnull().sum()
```

Step 10: Recalculate Outlier Bounds and Remove Outliers from the Updated Data

- **mean = df3['Age'].mean()**: Calculates the average (mean) value of the Age column in the DataFrame df3.
- **std = df3['Age'].std()**: Computes the standard deviation (spread or variability) of the Age column in df3.
- **lower_bound = mean - 2 * std**: Defines the lower limit for acceptable Age values, set as two standard deviations below the mean.
- **upper_bound = mean + 2 * std**: Defines the upper limit for acceptable Age values, set as two standard deviations above the mean.
- **df4 = df3[(df3['Age'] >= lower_bound) & (df3['Age'] <= upper_bound)]**: Creates a new DataFrame df4 by selecting only rows where the Age value falls between the lower and upper bounds, effectively removing outlier ages outside this range.

```
mean = df3['Age'].mean()
std = df3['Age'].std()

lower_bound = mean - 2 * std
upper_bound = mean + 2 * std

print('Lower Bound :', lower_bound)
print('Upper Bound :', upper_bound)
```

```
df4 = df3[(df3['Age'] >= lower_bound) & (df3['Age'] <= upper_bound)]
```

Step 11: Data validation and verification

Data validation and verification involve ensuring that the data is accurate and consistent by comparing it with external sources or expert knowledge. For the machine learning prediction we separate independent and target features. Here we will consider only 'Sex' 'Age' 'SibSp', 'Parch' 'Fare' 'Embarked' only as the independent features and Survived as target variables because PassengerId will not affect the survival rate.

```
X = df3[['Pclass','Sex','Age', 'SibSp','Parch','Fare','Embarked']]
Y = df3['Survived']
```

Step 12: Data formatting

Data formatting involves converting the data into a standard format or structure that can be easily processed by the algorithms or models used for analysis. Here we will discuss commonly used data formatting techniques i.e. Scaling and Normalization.

Scaling involves transforming the values of features to a specific range. It maintains the shape of the original distribution while changing the scale. It is useful when features have different scales and certain algorithms are sensitive to the magnitude of the features. Common scaling methods include:

1. Min-Max Scaling: Min-Max scaling rescales the values to a specified range, typically between 0 and 1. It preserves the original distribution and ensures that the minimum value maps to 0 and the maximum value maps to 1.

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))

num_col_ = [col for col in X.columns if X[col].dtype != 'object']
x1 = X
x1[num_col_] = scaler.fit_transform(x1[num_col_])
x1.head()
```

2. Standardization (Z-score scaling): Standardization transforms the values to have a mean of 0 and a standard deviation of 1. It centers the data around the mean and scales it based on the standard deviation. Standardization makes the data more suitable for algorithms that assume a Gaussian distribution or require features to have zero mean and unit variance.

$Z = (X - \mu) / \sigma$

Where,

- X = Data
- μ = Mean value of X
- σ = Standard deviation of X

ML | Handling Missing Values

Missing values are a common challenge in machine learning and data analysis. They occur when certain data points are missing for specific variables in a dataset. These gaps in information can take the form of blank cells, null values or special symbols like "NA", "NaN" or "unknown." If not addressed properly, missing values can harm the accuracy and reliability of our models. They can reduce the sample size, introduce bias and make it difficult to apply certain analysis techniques that require complete data. Efficiently handling missing values is important to ensure our machine learning models produce accurate and unbiased results. In this article, we'll see more about the methods and strategies to deal with missing data effectively.

	School ID	Name	Address	City	Subject	Marks	Rank	Grade
0	101.0	Alice	123 Main St	Los Angeles	Math	85.0	2	B
1	102.0	Bob	456 Oak Ave	New York	English	92.0	1	A
2	103.0	Charlie	789 Pine Ln	Houston	Science	78.0	4	C
3	NaN	David	101 Elm St	Los Angeles	Math	89.0	3	B
4	105.0	Eva	NaN	Miami	History	NaN	8	D
5	106.0	Frank	222 Maple Rd	NaN	Math	95.0	1	A
6	107.0	Grace	444 Cedar Blvd	Houston	Science	80.0	5	C
7	108.0	Henry	555 Birch Dr	New York	English	88.0	3	B

Importance of Handling Missing Values

Handling missing values is important for ensuring the accuracy and reliability of data analysis and machine learning models. Key reasons include:

- **Improved Model Accuracy:** Addressing missing values helps avoid incorrect predictions and boosts model performance.
- **Increased Statistical Power:** Imputation or removal of missing data allows the use of more analysis techniques, maintaining the sample size.
- **Bias Prevention:** Proper handling ensures that missing data doesn't introduce systematic bias, leading to more reliable results.
- **Better Decision-Making:** A clean dataset leads to more informed, trustworthy decisions based on accurate insights.

Challenges Posed by Missing Values

Missing values can introduce several challenges in data analysis including:

- **Reduce sample size:** If rows or data points with missing values are removed, it reduces the overall sample size which may decrease the reliability and accuracy of the analysis.
- **Bias in Results:** When missing data is not handled carefully, it can introduce bias. This is especially problematic when the missingness is not random, leading to misleading conclusions.
- **Difficulty in Analysis:** Many statistical techniques and machine learning algorithms require complete data for all variables. Missing values can cause certain analyses or models inapplicable, limiting the methods we can use.

Reasons Behind Missing Values in the Dataset

Data can be missing from a dataset for several reasons and understanding the cause is important for selecting the most effective way to handle it. Common reasons for missing data include:

- **Technical issues:** Failed data collection or errors during data transmission.
- **Human errors:** Mistakes like incorrect data entry or oversights during data processing.
- **Privacy concerns:** Missing sensitive or personal information due to confidentiality policies.
- **Data processing issues:** Errors that occur during data preparation.

By identifying the reason behind the missing data, we can better assess its impact whether it's causing bias or affecting the analysis and select the proper handling method such as imputation or removal.

Types of Missing Values

Missing values in a dataset can be categorized into three main types each with different implications for how they should be handled:

1. **Missing Completely at Random (MCAR):** In this case, the missing data is completely random and unrelated to any other variable in the dataset. The absence of data points occurs without any systematic pattern such as a random technical failure or data omission.
2. **Missing at Random (MAR):** The missingness is related to other observed variables but not to the value of the missing data itself. For example, if younger individuals are more likely to skip a particular survey question, the missingness can be explained by age but not by the content of the missing data.
3. **Missing Not at Random (MNAR):** Here, the probability of missing data is related to the value of the missing data itself. For example, people with higher incomes may be less likely to report their income, leading to a direct connection between the missingness and the value of the missing data.

Methods for Identifying Missing Data

Detecting and managing missing data is important for data analysis. Let's see some useful functions for detecting, removing and replacing null values in Pandas DataFrame.

Functions	Descriptions
.isnull()	Identifies missing values in a Series or DataFrame.
.notnull()	Opposite of **.isnull()**, returns True for non-missing values and False for missing values.
.info()	Displays DataFrame summary including data types, memory usage and the count of missing values.
.isna()	Works similarly to **.notnull()** but returns True for missing data and False for valid data.
dropna()	Removes rows or columns with missing values with customizable options for axis and threshold.
fillna()	Fills missing values with a specified value (like mean, median) or method (forward/backward fill).
replace()	Replaces specified values in the DataFrame, useful for correcting or standardizing data.
drop_duplicates()	Removes duplicate rows based on specified columns.
unique()	Finds unique values in a Series or DataFrame.

Representation of Missing Values in Datasets

Missing values can be represented by blank cells, specific values like "NA" or codes. It's important to use consistent and documented representation to ensure transparency and ease in data handling.

Common representations include:

1. **Blank Cells:** Empty cells in data tables or spreadsheets are used to signify missing values. This is common in many data formats like CSVs.
2. **Specific Values:** It is commonly used placeholders for missing data include "**NA**", "**NaN**", "**NULL**" or even arbitrary values like **-999.** It's

important to choose a standardized value and document its meaning to prevent confusion.

3. **Codes or Flags:** In some cases, non-numeric codes or flags (e.g "**MISSING**", "**UNKNOWN**") are used to show missing data. These can be useful in distinguishing between different types of missingness or categorizing missing data based on its origin.

Strategies for Handling Missing Values in Data Analysis

Depending on the nature of the data and the missingness, several strategies can help maintain the integrity of our analysis. Let's see some of the most effective methods to handle missing values.

Before moving to various strategies, let's first create a Sample Dataframe so that we can use it for different methods.

Creating a Sample Dataframe

Here we will be using Pandas and Numpy libraries.

```
import pandas as pd
import numpy as np

data = {
   'School ID': [101, 102, 103, np.nan, 105, 106, 107, 108],
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank', 'Grace', 'Henry'],
   'Address': ['123 Main St', '456 Oak Ave', '789 Pine Ln', '101 Elm St', np.nan, '222 Maple Rd', '444 Cedar Blvd', '555 Birch Dr'],
   'City': ['Los Angeles', 'New York', 'Houston', 'Los Angeles', 'Miami', np.nan, 'Houston', 'New York'],
   'Subject': ['Math', 'English', 'Science', 'Math', 'History', 'Math', 'Science', 'English'],
   'Marks': [85, 92, 78, 89, np.nan, 95, 80, 88],
   'Rank': [2, 1, 4, 3, 8, 1, 5, 3],
   'Grade': ['B', 'A', 'C', 'B', 'D', 'A', 'C', 'B']
}

df = pd.DataFrame(data)
print("Sample DataFrame:")

print(df)
```

1. Removing Rows with Missing Values

Removing rows with missing values is a simple and straightforward method to handle missing data, used when we want to keep our analysis clean and minimize complexity.

Advantages:

- **Simple and efficient:** It's easy to implement and quickly removes data points with missing values.
- **Cleans data:** It removes potentially problematic data points, ensuring that only complete rows remain in the dataset.

Disadvantages:

- **Reduces sample size:** When rows are removed, the overall dataset shrinks which can affect the power and accuracy of our analysis.
- **Potential bias:** If missing data is not random (e.g if certain groups are more likely to have missing values) removing rows could introduce bias.

In this example, we are removing rows with missing values from the original DataFrame (df) using the **dropna()** method and then displaying the cleaned DataFrame (df_cleaned).

```
df_cleaned = df.dropna()

print("\nDataFrame after removing rows with missing values:")
print(df_cleaned)
```

2. Imputation Methods

Imputation involves replacing missing values with estimated values. This approach is beneficial when we want to preserve the dataset's sample size and avoid losing data points. However, it's important to note that the accuracy of the imputed values may not always be reliable.

Let's see some common imputation methods:

2.1 Mean, Median and Mode Imputation:

This method involves replacing missing values with the mean, median or mode of the relevant variable. It's a simple approach but it doesn't account for the relationships between variables.

In this example, we are explaining the imputation techniques for handling missing values in the 'Marks' column of the DataFrame (df). It calculates and fills missing values with the mean, median and mode of the existing values in that column and then prints the results for observation.

- **df['Marks'].fillna(df['Marks'].mean())**: Fills missing values in the 'Marks' column with the **mean** value.
- **df['Marks'].fillna(df['Marks'].median())**: Fills missing values in the 'Marks' column with the median value.
- **df['Marks'].fillna(df['Marks'].mode():** Fills missing values in the 'Marks' column with the **mode** value.
- **.iloc[0]:** Accesses the first element of the Series which represents the mode.

```
mean_imputation = df['Marks'].fillna(df['Marks'].mean())
```

```
median_imputation = df['Marks'].fillna(df['Marks'].median())
mode_imputation = df['Marks'].fillna(df['Marks'].mode().iloc[0])

print("\nImputation using Mean:")
print(mean_imputation)

print("\nImputation using Median:")
print(median_imputation)

print("\nImputation using Mode:")
print(mode_imputation)
```

Advantages:

- **Simple and efficient:** Easy to implement and quick.
- **Works well with numerical data:** It is useful for numerical variables with a normal distribution.

Disadvantages:

- **Inaccuracy:** It assumes the missing value is similar to the central tendency (mean/median/mode) which may not always be the case.

2.2 Forward and Backward Fill

Forward and backward fill techniques are used to replace missing values by filling them with the nearest non-missing values from the same column. This is useful when there's an inherent order or sequence in the data.

The method parameter in **fillna()** allows to specify the filling strategy.

- **df['Marks'].fillna(method='ffill'):** This method fills missing values in the 'Marks' column of the DataFrame (df) using a forward fill strategy. It replaces missing values with the last observed non-missing value in the column.
- **df['Marks'].fillna(method='bfill'):** This method fills missing values in the 'Marks' column using a backward fill strategy. It replaces missing values with the next observed non-missing value in the column.

```
forward_fill = df['Marks'].fillna(method='ffill')
backward_fill = df['Marks'].fillna(method='bfill')

print("\nForward Fill:")
print(forward_fill)

print("\nBackward Fill:")
print(backward_fill)
```

Advantages:

- **Simple and Intuitive:** Preserves the temporal or sequential order in data.

- **Preserves Patterns:** Fills missing values logically, especially in time-series or ordered data.

Disadvantages:

- **Assumption of Closeness:** Assumes that the missing values are similar to the observed values nearby which may not always be true.
- **Potential Inaccuracy:** May not work well if there are large gaps between non-missing values.

Note:

- *Forward fill uses the last valid observation to fill missing values.*
- *Backward fill uses the next valid observation to fill missing values.*

3. Interpolation Techniques

Interpolation is a technique used to estimate missing values based on the values of surrounding data points. Unlike simpler imputation methods (e.g mean, median, mode), interpolation uses the relationship between neighboring values to make more informed estimations.

The **interpolate()** method in pandas are divided into Linear and Quadratic.

- **df['Marks'].interpolate(method='linear'):** This method performs linear interpolation on the 'Marks' column of the DataFrame (df).
- **df['Marks'].interpolate(method='quadratic'):** This method performs quadratic interpolation on the 'Marks' column.

```
linear_interpolation = df['Marks'].interpolate(method='linear')
quadratic_interpolation = df['Marks'].interpolate(method='quadratic')

print("\nLinear Interpolation:")
print(linear_interpolation)

print("\nQuadratic Interpolation:")
print(quadratic_interpolation)
```

Advantages:

- **Sophisticated Approach:** Interpolation is more accurate than simple imputation methods like mean or median, as it considers the underlying data structure.
- **Preserves Data Relationships:** Captures patterns or trends that exist between data points, which helps maintain the integrity of the dataset.

Disadvantages:

- **Complexity:** Requires more computational resources and additional libraries.
- **Assumptions on Data:** Assumes that data points follow a specific pattern (e.g., linear or quadratic), which may not always be true.

Note:

- *Linear interpolation assumes a straight line between two adjacent non-missing values.*
- *Quadratic interpolation assumes a quadratic curve that passes through three adjacent non-missing values.*

Impact of Handling Missing Values

Handling missing values effectively is important to ensure the accuracy and reliability of our findings.

Let's see some key impacts of handling missing values:

1. **Improved data quality:** A cleaner dataset with fewer missing values is more reliable for analysis and model training.
2. **Enhanced model performance:** Properly handling missing values helps models perform better by training on complete data, leading to more accurate predictions.
3. **Preservation of Data Integrity:** Imputing or removing missing values ensures consistency and accuracy in the dataset, maintaining its integrity for further analysis.
4. **Reduced bias:** Addressing missing values prevents bias in analysis, ensuring a more accurate representation of the underlying patterns in the data.

Detect and Remove the Outliers using Python

Outliers are data points that deviate significantly from other data points in a dataset. They can arise from a variety of factors such as measurement errors, rare events or natural variations in the data. If left unchecked it can distort data analysis, skew statistical results and impact machine learning model performance. In this article, we'll see how to detect and handle outliers in Python using various techniques to improve the quality and reliability of our data.

Common Causes of Outliers

Understanding the causes of outliers helps in finding the best approach to handle them. Some common causes include:

1. **Measurement errors**: Errors during data collection or from instruments can result in extreme values that don't reflect the underlying data distribution.
2. **Sampling errors**: Outliers can arise if the sample we collected isn't representative of the population we're studying.
3. **Natural variability**: Certain data points naturally fall outside the expected range especially in datasets with inherently high variability.

4. **Data entry errors**: Mistakes made during manual data entry such as incorrect values or typos can create outliers.
5. **Experimental errors**: Outliers can occur due to equipment malfunctions, environmental factors or unaccounted variables in experiments.
6. **Sampling from multiple populations**: Combining data from distinct populations with different characteristics can create outliers if researchers don't properly segment the datasets.
7. **Intentional outliers**: Sometimes outliers are deliberately introduced into datasets for testing purposes to evaluate the robustness of models or algorithms.

Need for Outliers Removal

Outliers can create significant issues in data analysis and machine learning which makes their removal important:

- **Skewed Statistical Measures**: Outliers can distort the mean, standard deviation and correlation values. For example an extreme value can make the mean unrepresentative of the actual data which leads to incorrect conclusions.
- **Reduced Model Accuracy**: Outliers can influence machine learning models especially those sensitive to extreme values like linear regression. They may cause the model to focus too much on these rare events helps in reducing its ability to generalize to new, unseen data.
- **Misleading Visualizations**: Outliers can stretch the scale of charts and graphs helps in making it difficult to interpret the main data trends. For example when visualizing a dataset with a few extreme values, it might find meaningful patterns in the majority of the data.

By removing or handling outliers, we prevent these issues and ensure more accurate analysis and predictions.

Methods for Detecting and Removing Outliers

There are several ways to detect and handle outliers in Python. We can use visualization techniques or statistical methods depending on the nature of our data Each method serves different purposes and is suited for specific types of data. Here we will be using Pandas and Matplotlib libraries on the Diabetes dataset which is preloaded in the Sckit-learn library.

1. Visualizing and Removing Outliers Using Box Plots

A boxplot is an effective way for visualizing the distribution of data using quartiles and the points outside the "whiskers" of the plot are considered outliers. They provide a quick way to see where the data is concentrated and where potential outliers lie.

```
import sklearn
from sklearn.datasets import load_diabetes
```

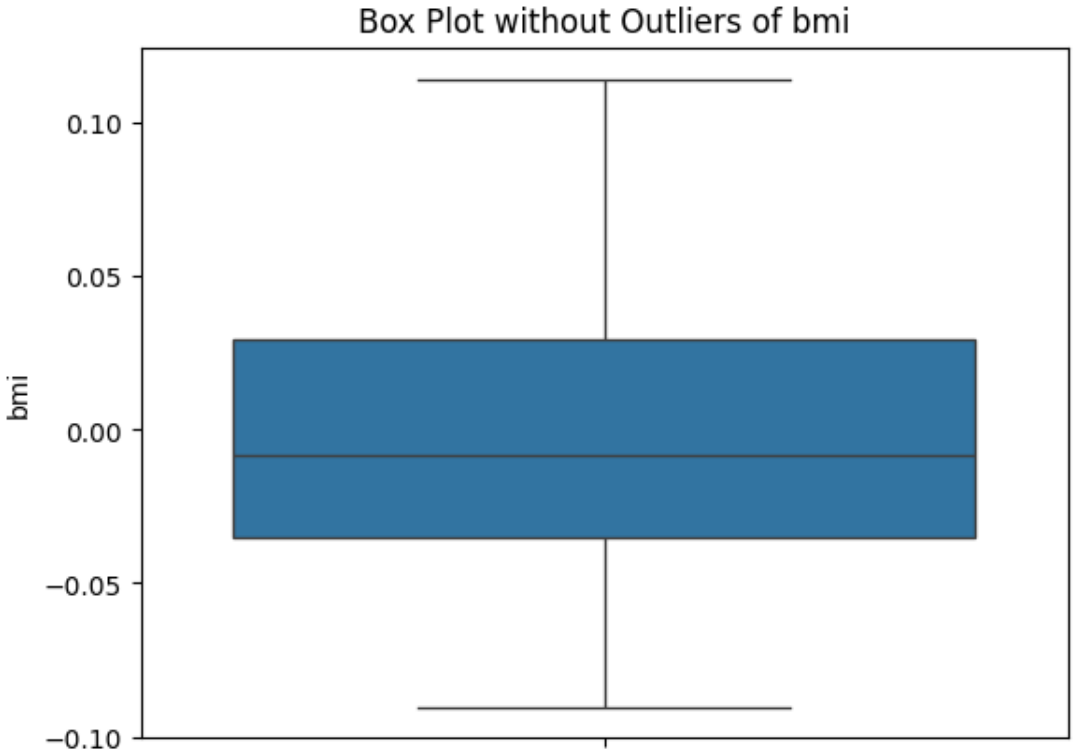
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

diabetes = load_diabetes()

column_name = diabetes.feature_names
df_diabetics = pd.DataFrame(diabetes.data, columns=column_name)

sns.boxplot(df_diabetics['bmi'])
plt.title('Boxplot of BMI')
plt.show()
```

Output:

To remove outliers, we can define a threshold value and filter the data.

```
def removal_box_plot(df, column, threshold):
    removed_outliers = df[df[column] <= threshold]

    sns.boxplot(removed_outliers[column])
    plt.title(f'Box Plot without Outliers of {column}')
    plt.show()
    return removed_outliers


threshold_value = 0.12

no_outliers = removal_box_plot(df_diabetics, 'bmi', threshold_value)
```

Output:



2. Visualizing and Removing Outliers Using Scatter Plots

Scatter plots help visualize relationships between two variables. It is used when we have paired numerical data and when our dependent variable has multiple values for each reading independent variable. Outliers appear as points far from the main cluster of data.

```
fig, ax = plt.subplots(figsize=(6, 4))
ax.scatter(df_diabetics['bmi'], df_diabetics['bp'])
ax.set_xlabel('BMI')
ax.set_ylabel('Blood Pressure')
plt.title('Scatter Plot of BMI vs Blood Pressure')
plt.show()
```

Looking at the graph can summarize that most of the data points are in the bottom left corner of the graph but there are few points that are exactly opposite that is the top right corner of the graph. Those points in the top right corner can be regarded as Outliers.

Here's how we can remove the outliers identified visually from the scatter plot.

- **np.where():** Used to find the positions (indices) where the condition is true in the DataFrame.
- **(df_diabetics['bmi'] > 0.12) & (df_diabetics['bp'] < 0.8):** Checks for outliers where 'bmi' is greater than 0.12 and 'bp' is less than 0.8.

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

outlier_indices = np.where((df_diabetics['bmi'] > 0.12) &
(df_diabetics['bp'] < 0.8))

no_outliers = df_diabetics.drop(outlier_indices[0])

fig, ax_no_outliers = plt.subplots(figsize=(6, 4))
ax_no_outliers.scatter(no_outliers['bmi'], no_outliers['bp'])
ax_no_outliers.set_xlabel('(body mass index of people)')
ax_no_outliers.set_ylabel('(bp of the people )')
plt.show()
```

3. Z-Score Method for Outlier Detection

Z- Score is also called a standard score. This score measures how far a data point is from the mean, in terms of standard deviations. If the Z-score exceeds a given threshold (commonly 3) the data point is considered an outlier.

$Z-score=x-\mu\sigma Z-score=\sigma x-\mu$

Where:

- xx = data point

- $\mu\mu$ = mean
- $\sigma\sigma$ = standard deviation

Here we are calculating the Z scores for the 'age' column in the DataFrame `df_diabetics` using the `z-score` function from the SciPy stats module. The resulting array `z` contains the absolute Z scores for each data point in the 'age' column which shows how many standard deviations each value is from the mean.

```
from scipy import stats
import numpy as np
z = np.abs(stats.zscore(df_diabetics['age']))
print(z)
```

Let's remove rows where Z value is greater than 2.

- **np.where()** : Used to find the positions (indices) in the Z-score array where the condition is true.
- **z > threshold** : Checks for outliers in the 'age' column where the absolute Z-score exceeds the defined threshold (typically 2 or 3).
- **threshold = 2** : A cutoff value used to identify outliers, data points with a Z-score greater than 2 are considered outliers.

```
import numpy as np

threshold_z = 2

outlier_indices = np.where(z > threshold_z)[0]
no_outliers = df_diabetics.drop(outlier_indices)
print("Original DataFrame Shape:", df_diabetics.shape)
print("DataFrame Shape after Removing Outliers:", no_outliers.shape)
```

Output:

Original DataFrame Shape: (442, 10)
DataFrame Shape after Removing Outliers: (426, 10)

4. Interquartile Range (IQR) Method

IQR (Inter Quartile Range) method is a widely used and reliable technique for detecting outliers. It is robust to skewed data and helps identify extreme values based on quartiles and it most trusted approach used in the research field. The IQR is calculated as the difference between the third quartile (Q3) and the first quartile (Q1):

$IQR=Q3-Q1$

Syntax : numpy.percentile(arr, n, axis=None, out=None)

Parameters:

- **arr**: Input array.
- **n**: Percentile value.

Here we are calculating the interquartile range (IQR) for the 'bmi' column in the DataFrame `df_diabetics`. It first finds the first quartile (Q1) and third quartile (Q3) using the midpoint method then calculates the IQR as the difference between Q3 and Q1 which providing a measure of the spread of the middle 50% of the data in the 'bmi' column.

```
Q1 = np.percentile(df_diabetics['bmi'], 25, method='midpoint')
Q3 = np.percentile(df_diabetics['bmi'], 75, method='midpoint')
IQR = Q3 - Q1
print(IQR)
```

Output:

0.06520763046978838

To define the outlier base value is defined above and below dataset's normal range namely Upper and Lower bounds define the upper and the lower bound (1.5*IQR value is considered i.e:

- upper=Q3+1.5∗IQR$upper = Q3 + 1.5 * IQR$
- lower=Q1−1.5∗IQR$lower = Q1 - 1.5 * IQR$

In the above formula the 0.5 scale-up of IQR (new_IQR = IQR + 0.5*IQR) is taken to consider all the data between 2.7 standard deviations in the Gaussian Distribution.

```
upper = Q3+1.5*IQR
upper_array = np.array(df_diabetics['bmi'] >= upper)
print("Upper Bound:", upper)
print(upper_array.sum())

lower = Q1-1.5*IQR
lower_array = np.array(df_diabetics['bmi'] <= lower)
print("Lower Bound:", lower)
print(lower_array.sum())
```

Now lets detect and remove outlier using the interquartile range (IQR).

Here we are using the interquartile range (IQR) method to detect and remove outliers in the 'bmi' column of the diabetes dataset. It calculates the upper and lower limits based on the IQR it identifies outlier indices using Boolean arrays and then removes the corresponding rows from the DataFrame which results in a new DataFrame with outliers excluded. The before and after shapes of the DataFrame are printed for comparison.

```
import sklearn
from sklearn.datasets import load_diabetes
import pandas as pd
diabetes = load_diabetes()

column_name = diabetes.feature_names
```

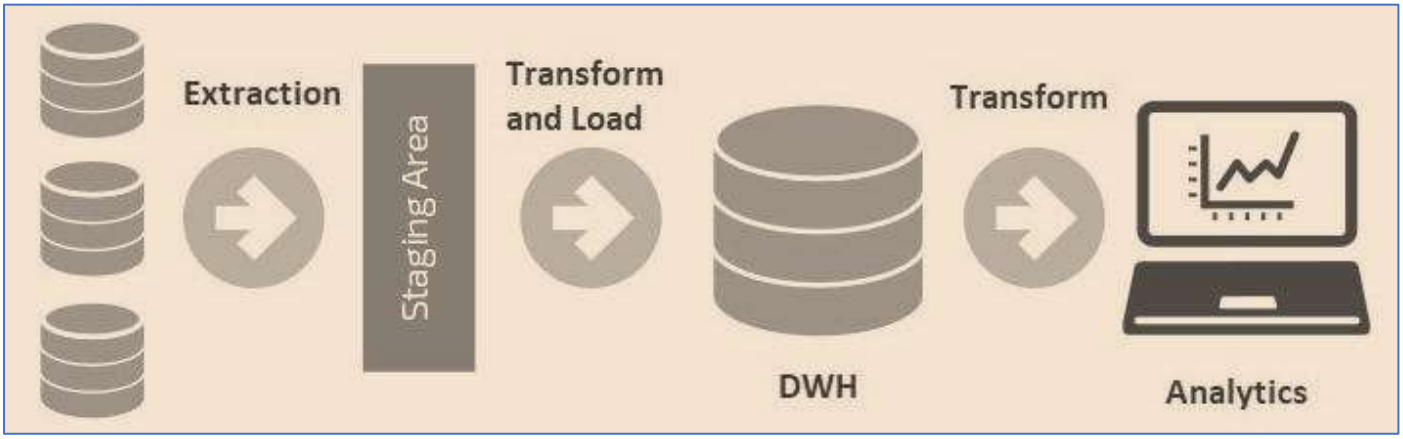
```
df_diabetes = pd.DataFrame(diabetes.data)
df_diabetes .columns = column_name
df_diabetes .head()
print("Old Shape: ", df_diabetes.shape)


Q1 = df_diabetes['bmi'].quantile(0.25)
Q3 = df_diabetes['bmi'].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5*IQR
upper = Q3 + 1.5*IQR

upper_array = np.where(df_diabetes['bmi'] >= upper)[0]
lower_array = np.where(df_diabetes['bmi'] <= lower)[0]

df_diabetes.drop(index=upper_array, inplace=True)
df_diabetes.drop(index=lower_array, inplace=True)

print("New Shape: ", df_diabetes.shape)
```

Output:

Old Shape: (442, 10)
New Shape: (439, 10)

What is Data Transformation

Data transformation is an important step in data analysis process that involves the conversion, cleaning, and organizing of data into accessible formats. It ensures that the information is accessible, consistent, secure, and finally recognized by the intended business users. This process is undertaken by organizations to utilize their data to generate timely business insights and support decision-making processes.



The transformations can be divided into two categories:

1. **Simple Data Transformations** involve basic tasks like cleansing, standardization, aggregation, and filtering used to prepare data for analysis or reporting through straightforward manipulation techniques
2. **Complex Data Transformations** involve advanced tasks like integration, migration, replication, and enrichment. They require techniques such as data modeling, mapping, and validation, and are used to prepare data for machine learning, advanced analytics, or data warehousing.

Importance of Data Transformation

Data transformation is important because it improves data quality, compatibility, and utility. The procedure is critical for companies and organizations that depend on data to make informed decisions because it assures the data's accuracy, reliability, and accessibility across many systems and applications.

1. **Improved Data Quality:** Data transformation eliminates mistakes, inserts in missing information, and standardizes formats, resulting in higher-quality, more dependable, and accurate data.
2. **Enhanced Compatibility**: By converting data into a suitable format, companies may avoid possible compatibility difficulties when integrating data from many sources or systems.
3. **Simplified Data Management:** Data transformation is the process of evaluating and modifying data to maximize storage and discoverability, making it simpler to manage and maintain.
4. **Broader Application:** Transformed data is more useable and applicable in a larger variety of scenarios, allowing enterprises to get the most out of their data.
5. **Faster Queries:** By standardizing data and appropriately storing it in a warehouse, query performance and BI tools may be enhanced, resulting in less friction during analysis.

Data Transformation Techniques and Tools

There are several ways to alter data, including:

- **Programmatic Transformation**: Automated using scripts in Python, R, or SQL.
- **ETL Tools**: Automate extract-transform-load processes for large-scale data (e.g., Talend, Informatica).
- **Normalization/Standardization**:
  Use `MinMaxScaler`, `StandardScaler` from *Scikit-learn*.
- **Categorical Encoding** such as **One-hot** `get_dummies` (*Pandas*) and **`LabelEncoder`** (*Scikit-learn*)
- `fillna` (*Pandas*)
- `SimpleImputer` (*Scikit-learn*) for mean/median/mode imputation.

- **Feature Engineering**: Create new features using `apply`, `map`, `transform` in *Pandas*.
- **Aggregation/Grouping**: Use `groupby` in *Pandas* for `sum`, `mean`, `count`, etc.
- **Text Preprocessing**: Tokenization, stemming, stop-word removal using *NLTK* and *SpaCy*.
- **Dimensionality Reduction**: Use *Scikit-learn's* `PCA` and `TruncatedSVD`.

Advantage and Disadvantage

Advantage	Disadvantage
Improves Data Quality	**Time-Consuming** – Especially for large datasets.
Ensures Compatibility Across Systems	**Complex Process** – Requires specialized skills.
Enables Accurate Analysis	**Possible Data Loss** – E.g., during discretization.
Enhances Data Security – Masks or removes sensitive info.	**Bias Risk** – Poor understanding can lead to biased results.
Boosts Algorithm Performance – Through scaling and dimensionality reduction.	**High Cost** – Needs investment in tools and expertise.

Applications of Data Transformation

Applications for data transformation are found in a number of industries:

1. **Business intelligence (BI)** is the process of transforming data for use in real-time reporting and decision-making using BI technologies.
2. **Healthcare:** Ensuring interoperability across various healthcare systems by standardization of medical records.
3. **Financial Services:** Compiling and de-identifying financial information for reporting and compliance needs.
4. **Retail:** Improving customer experience through data transformation into an analytics-ready format and customer behavior analysis.
5. **Customer Relationship Management (CRM):** By converting customer data, firms may obtain insights into consumer behavior, tailor marketing strategies, and increase customer satisfaction.

Feature Engineering: Scaling, Normalization and Standardization

Feature engineering is a crucial step in preparing data for machine learning models, where raw features are transformed to improve model performance and accuracy.

They make sure that features with different ranges or units are transformed into a comparable scale so that models can learn effectively.

- **Scaling** adjusts the range of features without changing their distribution.
- **Normalization** brings values into a fixed range, usually [0,1].
- **Standardization** centers data around the mean with unit variance.

These techniques improve model accuracy, speed up training and ensure that no single feature dominates the learning process. Lets see various techniques:

1. Absolute Maximum Scaling

Absolute Maximum Scaling rescales each feature by dividing all values by the maximum absolute value of that feature. This ensures the feature values fall within the range of -1 to 1. While simple and useful in some contexts, it is highly sensitive to outliers which can skew the max absolute value and negatively impact scaling quality.

$Xscaled=Ximax(|X|)Xscaled=max(|X|)Xi$

- Scales values between -1 and 1.
- Sensitive to outliers, making it less suitable for noisy datasets.

Code Example: We will first Load the Dataset

Dataset can be downloaded from here.

```
import pandas as pd
import numpy as np

df = pd.read_csv('Housing.csv')

df = df.select_dtypes(include=np.number)
df.head()
```

- Computes max absolute value per column with np.max(np.abs(df), axis=0).
- Divides each value by that max absolute to scale features between -1 and 1.
- Displays first few rows of scaled data with scaled_df.head().

```
max_abs = np.max(np.abs(df), axis=0)

scaled_df = df / max_abs

scaled_df.head()
```

2. Min-Max Scaling

Min-Max Scaling transforms features by subtracting the minimum value and dividing by the difference between the maximum and minimum values. This method

maps feature values to a specified range, commonly 0 to 1, preserving the original distribution shape but is still affected by outliers due to reliance on extreme values.
$Xscaled=Xi−XminXmax−XminX$scaled$=Xmax−XminXi−Xmin$

- Scales features to range.
- Sensitive to outliers because min and max can be skewed.

Code Example: Performing Min-Max Scaling

- Creates MinMaxScaler object to scale features to range.
- Fits scaler to data and transforms with scaler.fit_transform(df).
- Converts result to DataFrame maintaining column names.
- Shows first few scaled rows with scaled_df.head().

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled_data, columns=df.columns)

scaled_df.head()
```

3. Normalization (Vector Normalization)

Normalization scales each data sample (row) such that its vector length (Euclidean norm) is 1. This focuses on the direction of data points rather than magnitude making it useful in algorithms where angle or cosine similarity is relevant, such as text classification or clustering.

$Xscaled=Xi‖X‖Xscaled=‖X‖Xi$

Where:

- Xi is each individual value.
- $‖X‖$ represents the Euclidean norm (or length) of the vector X.
- Normalizes each sample to unit length.
- Useful for direction-based similarity metrics.

Code Example: Performing Normalization

- Scales each row (sample) to have unit norm (length = 1) based on Euclidean distance.
- Focuses on direction rather than magnitude of data points.
- Useful for algorithms relying on similarity or angles (e.g., cosine similarity).
- scaled_df.head() shows normalized data where each row is scaled individually.

```
from sklearn.preprocessing import Normalizer

scaler = Normalizer()
scaled_data = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled_data, columns=df.columns)
```

```
scaled_df.head()
```

4. Standardization

Standardization centers features by subtracting the mean and scales them by dividing by the standard deviation, transforming features to have zero mean and unit variance. This assumption of normal distribution often benefits models like linear regression, logistic regression and neural networks by improving convergence speed and stability.

$Xscaled=Xi−\mu\sigma X\text{scaled}=\sigma Xi−\mu$

- where $\mu\mu$ = mean, $\sigma\sigma$ = standard deviation.
- Produces features with mean 0 and variance 1.
- Effective for data approximately normally distributed.

Code Example: Performing Standardization

- Centers features by subtracting mean and scales to unit variance.
- Transforms data to have zero mean and standard deviation of 1.
- Assumes roughly normal distribution; improves many ML algorithms' performance.
- scaled_df.head() shows standardized features.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled_data,
                         columns=df.columns)
print(scaled_df.head())
```

5. Robust Scaling

Robust Scaling uses the median and interquartile range (IQR) instead of the mean and standard deviation making the transformation robust to outliers and skewed distributions. It is highly suitable when the dataset contains extreme values or noise.

$Xscaled=Xi−Xmedian\ IQRX\text{scaled}=IQRXi−Xmedian$

- Reduces influence of outliers by centering on median
- Scales based on IQR, which captures middle 50% spread

Code Example: Performing Robust Scaling

- Uses median and interquartile range (IQR) for scaling instead of mean/std.
- Robust to outliers and skewed data distributions.
- Centers data around median and scales based on spread of central 50% values.
- scaled_df.head() shows robustly scaled data minimizing outlier effects.

```
from sklearn.preprocessing import RobustScaler
```

```
scaler = RobustScaler()
scaled_data = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled_data,
                         columns=df.columns)
print(scaled_df.head())
```

Comparison of Various Feature Scaling Techniques

Let's see the key differences across the five main feature scaling techniques commonly used in machine learning preprocessing.

Type	Method Description	Sensitivity to Outliers	Typical Use Cases
Absolute Maximum Scaling	Divides values by max absolute value in each feature	High	Sparse data, simple scaling
Min-Max Scaling (Normalization)	Scales features to by min-max normalization	High	Neural networks, bounded input features
Normalization (Vector Norm)	Scales each sample vector to unit length (norm = 1)	Not applicable (per row)	Direction-based similarity, text classification
Standardization (Z-Score)	Centers features to mean 0 and scales to unit variance	Moderate	Most ML algorithms, assumes approx. normal data
Robust Scaling	Centers on median and scales using IQR	Low	Data with outliers, skewed distributions

What is Data Sampling - Types, Importance

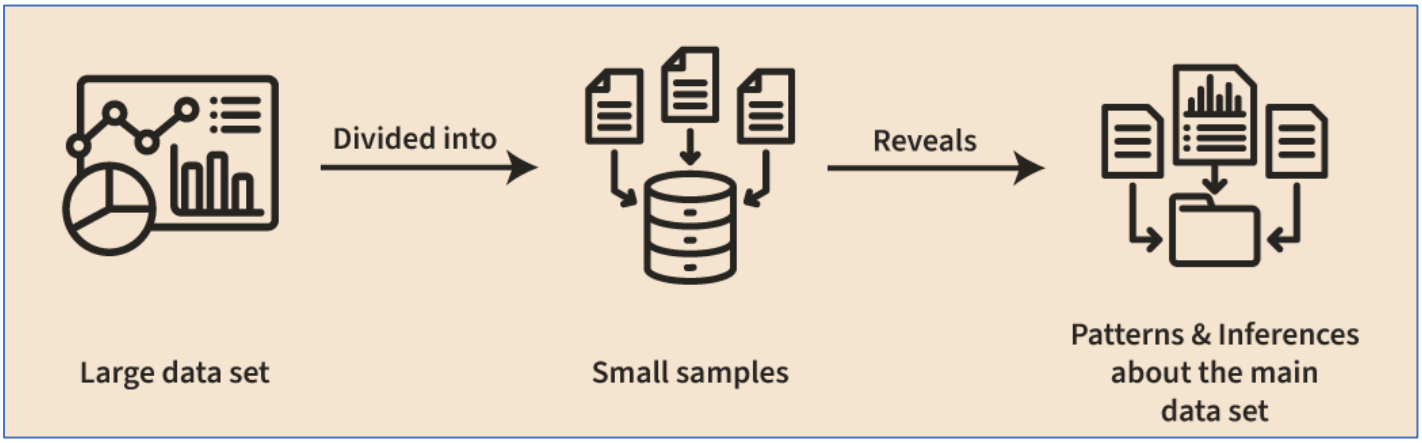
Data sampling is a statistical method that selects a **representative subset** (sample) from a large dataset. Analysts then study this sample to make inferences and draw conclusions about the entire dataset. It's a powerful tool for handling large volumes of data efficiently

Data Sampling Process

The process of data sampling involves the following steps:

1. **Identify the Target Dataset:** Choose the large dataset you want to study—it represents the whole population.
2. **Determine Sample Size:** Decide how many data points to include in your sample. This subset will be used for analysis.

3. **Choose a Sampling Method**: Select a suitable technique like Random, Systematic, Cluster, Stratified, or Snowball sampling, based on your goals and data type.
4. **Collect the Sample**: Apply the chosen method to extract the sample from the dataset systematically.
5. **Analyze the Sample:** Study the sample to understand patterns and characteristics using statistical tools.
6. **Generalize to the Population**: Use insights from the sample to make predictions or conclusions about the entire dataset.



Importance of Data Sampling

Data sampling is important for given reasons:

- **Cost & Time Efficient**: Sampling reduces the time and resources needed by analyzing just a portion of the data instead of the entire dataset.
- **Feasible for Large Populations**: When it's too costly or impractical to study the whole population, sampling offers a manageable and effective solution.
- **Reduces Risk of Error**: By using proper sampling methods, researchers can avoid biases and minimize the influence of outliers.
- **Maintains Accuracy**: A well-chosen sample can accurately reflect the larger population—ideal when testing or analyzing everything isn't possible.

Types of Data Sampling Techniques

There are mainly two types of Data Sampling techniques which are further divided into 4 sub-categories each. They are as follows:

1. Probability Data Sampling Technique

Probability Sampling ensures every data point has a known, non-zero chance of being selected. This helps create a representative sample, allowing reliable generalization to the whole population.

- **Simple Random Sampling**: Each data point has an equal chance of selection.
  Example: Tossing a coin—head or tail has equal probability.

- **Systematic Sampling**: Data is selected at regular intervals from an ordered list.
  Example: From 10 entries, selecting every 2nd one (2nd, 4th, 6th...).
- **Stratified Sampling**: Data is divided into groups (strata) based on shared traits, and samples are drawn from each group.
  Example: Dividing employees by gender, then sampling from each group.
- **Cluster Sampling**: Random groups (clusters) are selected, and then all or some members within them are sampled.
  Example: Choosing random user groups from different mobile networks.

2. Non-Probability Data Sampling

Non-probability data sampling means that the selection happens on a non-random basis, and it depends on the individual as to which data does it want to pick. There is no random selection and every selection is made by a thought and an idea behind it.

- **Convenience Sampling**: Data is selected based on ease of access and minimal effort.
  Example: Choosing the most recent or easily available IT recruitment data.
- **Voluntary Response Sampling**: Participants choose to take part in the study on their own.
  Example: A blood group survey where only willing participants respond.
- **Purposive Sampling**: Data is selected for a specific purpose or characteristic.
  Example: Surveying rural areas to study educational needs.
- **Snowball Sampling:** Participants recruit others, growing the sample like a snowball.
  Example: One slum resident leads to another in a housing conditions survey.

Advantage	Disadvantage
Helps draw conclusion about large datasets from smaller samples.	Difference between sample and population reduces accuracy
saves times with faster data analysis	Difficulties in some methods like cluster formation
cost effective reduces expenses in data collection and processing	Sample may not represent the population due to poor sampling technique
can produce accurate and reliable result when sampling is done correctly	Inaccurate conclusions if the sample lacks proper representation or is too small

Sample Size Determination

Sample size is the universal dataset concerning to which several other smaller datasets are created that help in inferring the properties of the entire dataset. Following are a series of steps that are involved during sample size determination.

1. Firstly calculate the population size, as in the total sample space size on which the sampling has to be performed.
2. Find the values of confidence levels that represent the accuracy of the data.
3. Find the value of error margins if any with respect to the sample space dataset.
4. Calculate the deviation from the mean or average value from that of standard deviation value calculated.