

Data Mining and Business Intelligence

Experiment - 2

Name: **Abhishek Vishwakarma**

Class: **D15C**

Roll No: **73**

Aim: Data preprocessing using python

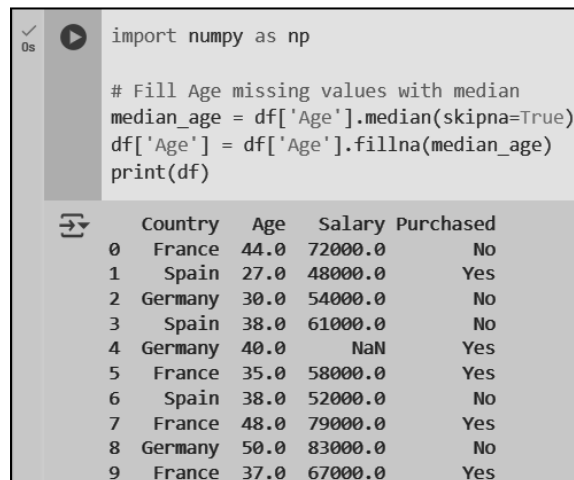
Theory:

Data preprocessing is a crucial step in data analysis and machine learning, ensuring that raw data is cleaned, transformed, and ready for further processing. The following operations are commonly performed to improve data quality and consistency:

1. Handling Missing Values

Missing data can arise due to various reasons such as data entry errors, equipment failure, or incomplete surveys. If left untreated, missing values can lead to biased results or errors in model training.

Median imputation is a robust method for handling missing numeric values, especially when the data is skewed or contains outliers. In this method, missing values in a numeric column (e.g., *Age*) are replaced with the median of the available values, minimizing the effect of extreme values.



```
import numpy as np

# Fill Age missing values with median
median_age = df['Age'].median(skipna=True)
df['Age'] = df['Age'].fillna(median_age)
print(df)
```

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	38.0	52000.0	No
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

2. Removing Duplicates

Duplicate records can occur due to multiple data entries or integration from different sources. These duplicates inflate the dataset size, distort statistics, and can lead to overfitting in machine learning models. Removing duplicates ensures each observation in the dataset is unique, maintaining data integrity.



A Jupyter Notebook interface showing a code cell with the following code:

```
df = df.drop_duplicates()
print(df)
```

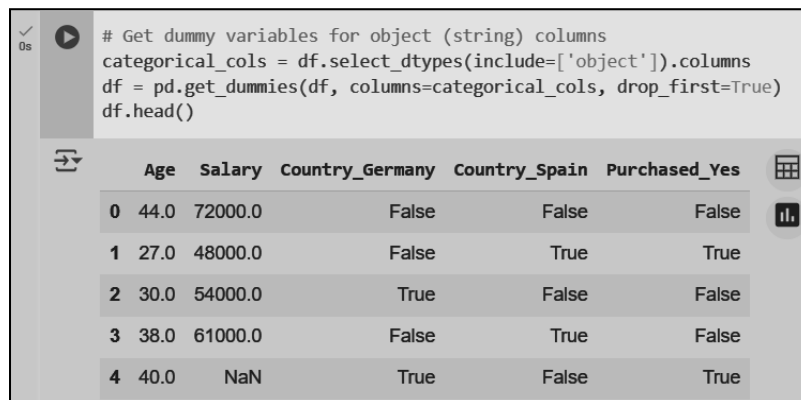
Below the code cell is a table representing the output of the code. The table has 5 columns: an index column (0-9), 'Country', 'Age', 'Salary', and 'Purchased'. The data is as follows:

	Country	Age	Salary	Purchased
0	France	44.0	72000.0	No
1	Spain	27.0	48000.0	Yes
2	Germany	30.0	54000.0	No
3	Spain	38.0	61000.0	No
4	Germany	40.0	NaN	Yes
5	France	35.0	58000.0	Yes
6	Spain	38.0	52000.0	No
7	France	48.0	79000.0	Yes
8	Germany	50.0	83000.0	No
9	France	37.0	67000.0	Yes

3. Encoding Categorical Variables

Machine learning algorithms typically require numeric input. Categorical variables (e.g., *Country*, *Purchased*) must be converted into numeric form. Common methods include:

- Label Encoding: Assigning an integer to each category (useful for ordinal data).
- One-Hot Encoding: Creating binary columns for each category (useful for nominal data). This transformation enables algorithms to interpret categorical features correctly without implying false ordering.



A Jupyter Notebook interface showing a code cell with the following code:

```
# Get dummy variables for object (string) columns
categorical_cols = df.select_dtypes(include=['object']).columns
df = pd.get_dummies(df, columns=categorical_cols, drop_first=True)
df.head()
```

Below the code cell is a table representing the output of the code. The table has 6 columns: 'Age', 'Salary', 'Country_Germany', 'Country_Spain', and 'Purchased_Yes'. The data is as follows:

	Age	Salary	Country_Germany	Country_Spain	Purchased_Yes
0	44.0	72000.0	False	False	False
1	27.0	48000.0	False	True	True
2	30.0	54000.0	True	False	False
3	38.0	61000.0	False	True	False
4	40.0	NaN	True	False	True

4. Fixing Data Types

Incorrect data types (e.g., storing numeric values as strings) can prevent proper analysis and computations. For example, *Salary* values should be stored as `float` to allow arithmetic operations. Correct data type assignment ensures accurate calculations, efficient storage, and compatibility with analytical tools.

✓ 0s	▶	<pre>df['Salary'] = df['Salary'].astype(float) print(df)</pre>				
↕		Age	Salary	Country_Germany	Country_Spain	Purchased_Yes
0		44.0	72000.0	False	False	False
1		27.0	48000.0	False	True	True
2		30.0	54000.0	True	False	False
3		38.0	61000.0	False	True	False
4		40.0	NaN	True	False	True
5		35.0	58000.0	False	False	True
6		38.0	52000.0	False	True	False
7		48.0	79000.0	False	False	True
8		50.0	83000.0	True	False	False
9		37.0	67000.0	False	False	True

5. Handling Outliers

Outliers are values that deviate significantly from the majority of the data. They can result from measurement errors, data entry mistakes, or rare events. Outliers can distort statistical measures and mislead model training.

In this experiment, outliers in the *Age* column (values greater than 100) are handled by replacing them with the median age, thus preserving the dataset's integrity while mitigating extreme values.

✓ 0s	▶	<pre>df.loc[df['Age'] > 100, 'Age'] = median_age print(df)</pre>				
↕		Age	Salary	Country_Germany	Country_Spain	Purchased_Yes
0		44.0	72000.0	False	False	False
1		27.0	48000.0	False	True	True
2		30.0	54000.0	True	False	False
3		38.0	61000.0	False	True	False
4		40.0	NaN	True	False	True
5		35.0	58000.0	False	False	True
6		38.0	52000.0	False	True	False
7		48.0	79000.0	False	False	True
8		50.0	83000.0	True	False	False
9		37.0	67000.0	False	False	True

Conclusion:

By performing these preprocessing steps—handling missing values, removing duplicates, encoding categorical variables, fixing data types, and addressing outliers—we ensure the dataset is clean, consistent, and suitable for reliable analysis or machine learning. Proper preprocessing directly impacts the accuracy and performance of analytical models.