Exercise 2: Demonstrate various data pre-processing techniques for a given dataset.

Program:

Sample Dataset: Let's use a small custom dataset to simulate the scenario:

```
import pandas as pd
import numpy as np

data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
    'Age': [25, np.nan, 35, 45, 29],
    'Gender': ['F', 'M', 'M', np.nan, 'F'],
    'Income': [50000, 60000, 80000, 120000, np.nan],
    'Loan_Status': ['Y', 'N', 'Y', 'N', 'Y']
}

df = pd.DataFrame(data)
print(df)
```

1. Handling Missing Values:

◆ **Identify Missing Data**

```
print(df.isnull().sum())
```

◆ **Fill Missing Values**

```
# Fill Age with mean
df['Age'].fillna(df['Age'].mean(), inplace=True)

# Fill Gender with mode
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)

# Fill Income with median
df['Income'].fillna(df['Income'].median(), inplace=True)
```

2. Encoding Categorical Variables

◆ **Label Encoding for Binary Classes**

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
```

```
df['Gender'] = le.fit_transform(df['Gender'])  # F = 0, M = 1
df['Loan_Status'] = le.fit_transform(df['Loan_Status'])  # N = 0, Y = 1
```

◆ **One-Hot Encoding (alternative)**

```
# pd.get_dummies(df['Gender']) # if there are more than two categories
```

3. Feature Scaling

◆ **Standardization (Z-score)**

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```

◆ **Normalization (Min-Max)**

```
# from sklearn.preprocessing import MinMaxScaler
# mms = MinMaxScaler()
# df[['Age', 'Income']] = mms.fit_transform(df[['Age', 'Income']])
```

4. Outlier Detection and Treatment

Using **Z-Score** method:

```
from scipy.stats import zscore

z_scores = zscore(df[['Age', 'Income']])
outliers = (np.abs(z_scores) > 3).any(axis=1)
print("Outliers:\n", df[outliers])
```

Treat outliers (e.g., cap them):

```
df['Income'] = np.where(df['Income'] > 2.5, 2.5, df['Income'])
```

5. Feature Selection (Optional)

Using Variance Threshold

```
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(threshold=0.1)
features = selector.fit_transform(df[['Age', 'Income', 'Gender']])
```

6. Final Preprocessed Data

```
print(df)
```

Importance of Data Pre-processing

- **Data cleaning** is one of the first steps in ML or analytics.
- Identifying missing values helps decide:
    - Whether to fill (impute), drop, or leave them.
- Avoids runtime errors and improves model quality.

Let's use a small custom dataset to simulate the scenario:

```
import pandas as pd
import numpy as np

data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
   'Age': [25, np.nan, 35, 45, 29],
   'Gender': ['F', 'M', 'M', np.nan, 'F'],
   'Income': [50000, 60000, 80000, 120000, np.nan],
   'Loan_Status': ['Y', 'N', 'Y', 'N', 'Y']
}

df = pd.DataFrame(data)
print(df)
```

Let go through the code line by line:

data = { ... }

- You are **creating a Python dictionary** named data.
- The dictionary contains **five key-value pairs**, where each key (like 'Name', 'Age') is associated with a list of values.

◆ **'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve']**

- This is a **list of strings**.
- Python stores this as a list object in memory, under the key 'Name'.

◆ **'Age': [25, np.nan, 35, 45, 29]**

- This is a **list of numbers**, but it contains np.nan.

np.nan:

- np.nan is a constant in the **NumPy library** that represents **"Not a Number"**, or a missing/undefined value in numerical arrays.
- For this to work, the line import numpy as np must have been executed before this block.

◆ **'Gender': ['F', 'M', 'M', np.nan, 'F']**

- A list of string values, one of which is np.nan, indicating a **missing value** in the categorical Gender column.

◆ **'Income': [50000, 60000, 80000, 120000, np.nan]**

- A list of numerical income values.
- Again, np.nan represents a **missing income entry**.

◆ **'Loan_Status': ['Y', 'N', 'Y', 'N', 'Y']**

- A **binary categorical column**, where:
  - 'Y' = Yes (loan approved)
  - 'N' = No (loan not approved)

After Execution:

- data is a **Python dictionary** in memory.
- You can use this dictionary to create a **Pandas DataFrame**, as shown below:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(data)
print(df)
```

Output:

	Name	Age	Gender	Income	Loan_Status
0	Alice	25.0	F	50000.0	Y
1	Bob	NaN	M	60000.0	N
2	Charlie	35.0	M	80000.0	Y
3	David	45.0	NaN	120000.0	N
4	Eve	29.0	F	NaN	Y

1. Handling Missing Values

Identify Missing Data

```
print(df.isnull().sum())
```

This line performs **missing value analysis** on a **Pandas DataFrame (df)**, and prints the **number of missing (NaN) values in each column**.

Output:

Name	0
Age	1
Gender	1
dtype:	int64

Meaning:

- No missing values in Name
- 1 missing value in Age
- 1 missing value in Gender

print(...)

- Simply **prints the result** to the console.

Let's understand it **step-by-step**.

Step-by-Step Explanation:

df

- This refers to a **Pandas DataFrame** object.
- The DataFrame typically looks like a table with rows and columns (like an Excel sheet).
- It may contain numerical, categorical, or text data.

df.isnull()

- This method **checks for missing values (NaN)** in the DataFrame.
- Returns a **DataFrame of the same shape**, but with:
    - True if a value is missing (NaN)
    - False if the value is present

Example: If your DataFrame looks like:

Name	Age	Gender
Alice	25.0	F
Bob	NaN	M
Charlie	35.0	NaN

Then df.isnull() will return:

Name	Age	Gender
False	False	False
False	True	False
False	False	True

.sum()

- This method performs a **column-wise sum** by default.
- Since True = 1 and False = 0 in Python, it adds up the number of True values in each column.
- So, it **counts the number of missing values (NaNs)** per column.

Fill Missing Values for "Age" with mean

```
# Fill Age with mean
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

This line is used to **handle missing values** in the Age column of a **Pandas DataFrame** (df) by **replacing the missing values (NaN) with the mean value** of the Age column.

Step-by-Step Explanation

df['Age']

- This accesses the **Age column** from the DataFrame df.
- The result is a **Pandas Series** (1D labeled array) containing the values of that column.

Example:

```
0    25.0
1     NaN
2    35.0
3    45.0
4    29.0
Name: Age, dtype: float64
```

.mean()

- This calculates the **mean (average)** of the numeric values in the Age column, **ignoring any NaNs**.
- In our example:

(25.0 + 35.0 + 45.0 + 29.0) / 4 = 33.5

So, df['Age'].mean() returns 33.5.

.fillna(...)

- This is a **Pandas function** that **replaces missing values (NaN)** with a specified value.
- Here, you are telling it to **fill all NaN values in Age with the mean value**, which is 33.5.

Before:
0 25.0
1 NaN
2 35.0
3 45.0
4 29.0

After:
0 25.0
1 33.5
2 35.0
3 45.0
4 29.0

inplace=True

- This tells Pandas to **make the change directly** in the original DataFrame (df) **without creating a new object**.
- If you omit this or set inplace=False, Pandas returns a **new Series**, and the original column remains unchanged.

What Happens Internally

1. Pandas calculates the mean of all non-missing values in df['Age'].
2. It finds the indices where Age is NaN.
3. It fills those NaN values with the mean (e.g., 33.5).
4. It updates df['Age'] directly because of inplace=True.

Fill Missing Values for “Gender” with mode

```
# Fill Gender with mode
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
```

This line is used to **fill missing values (NaN) in the Gender column** of a Pandas DataFrame (df) using the **most frequently occurring value** (called the **mode**) in that column.

df['Gender']

- This accesses the Gender column from the DataFrame df.
- It returns a **Pandas Series** containing all values in the column.

Example:

```
0      F
1      M
2      M
3    NaN
4      F
Name: Gender, dtype: object
```

.mode()

- .mode() computes the **mode** — the **most frequently occurring value** in the column.
- Unlike .mean() (used for numeric data), .mode() can also be applied to **categorical data** like strings.

In the above example:

- Values: 'F', 'M', 'M', NaN, 'F'
- Frequencies: 'F' → 2 times, 'M' → 2 times
- So .mode() returns **a Series with multiple values** because there’s a tie:
-

```
0    F
1    M
dtype: object
```

.mode()[0]

- This selects the **first mode** value from the result (even if there’s a tie).
- In this case, it will be 'F' (because it appears first in the list).

- So this returns a **single value**: 'F'

.fillna(...)

- This replaces any NaN values in the Gender column with the specified value.
- So all missing gender entries will be filled with 'F' (the mode).

Before:
0 F
1 M
2 M
3 NaN
4 F

After:
0 F
1 M
2 M
3 F
4 F

Fill Missing Values for "Income" with median

Why Use Median Instead of Mean?

- **Median** is more **robust to outliers**.
- Example:
    - Income values: [30000, 35000, 40000, 1500000] → Mean ≈ very large, Median = 37500
    - In such skewed distributions, **median** gives a better central value.

```
# Fill Income with median
df['Income'].fillna(df['Income'].median(), inplace=True)
```

This statement is used to **fill missing (NaN) values** in the `'Income'` column of a pandas DataFrame named `df`. Here's a detailed explanation of each part:

df['Income']
This accesses the 'Income' column of the DataFrame df.

.fillna(...)
This is a **Pandas Series method** used to **replace all missing values (NaN)** in the column with a specified value.

df['Income'].median()
This calculates the **median** (middle value) of all the non-missing values in the 'Income' column.

- Median is a measure of central tendency that is **less affected by outliers** than the mean.
- For example, if Income = [50000, 60000, 80000, 120000], the median is **70000**.

What It Does Altogether
It replaces all missing values in the 'Income' column with the **median value** of the existing (non-missing) incomes, directly modifying the DataFrame.

Example

```
import pandas as pd
import numpy as np

data = {
   'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve'],
   'Income': [50000, 60000, 80000, 120000, np.nan]
}
df = pd.DataFrame(data)

Before:
df['Income']
# 0     50000.0
# 1     60000.0
# 2     80000.0
# 3    120000.0
# 4         NaN

Apply:
        df['Income'].fillna(df['Income'].median(), inplace=True)

After:
        df['Income']
```

```
# 0     50000.0
# 1     60000.0
# 2     80000.0
# 3    120000.0
# 4     70000.0   ← Filled with median
```

2. Encoding Categorical Variables

Label Encoding for Binary Classes

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['Gender'] = le.fit_transform(df['Gender'])  # F = 0, M = 1
df['Loan_Status'] = le.fit_transform(df['Loan_Status'])  # N = 0, Y = 1
```

One-Hot Encoding (alternative)

```
# pd.get_dummies(df['Gender']) # if there are more than two categories
```

Why Encode Categorical Variables?

Machine learning algorithms usually work only with numerical data. So, categorical columns like "Gender" ('M', 'F') or "Loan_Status" ('Y', 'N') must be converted into **numbers** before they can be fed into models.

There are two common ways:

1. **Label Encoding** – replaces each category with a unique integer.
2. **One-Hot Encoding** – creates **binary columns** (0/1) for each category.

1. Label Encoding for Binary Classes

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
df['Gender'] = le.fit_transform(df['Gender'])        # F = 0, M = 1
df['Loan_Status'] = le.fit_transform(df['Loan_Status'])  # N = 0, Y = 1
```

What is LabelEncoder?

LabelEncoder from sklearn.preprocessing is used to **convert categorical labels** into **numeric values** by assigning a **unique integer** to each class.

What it does:

- For 'Gender':
  - 'F' → 0
  - 'M' → 1
- For 'Loan_Status':
  - 'N' → 0
  - 'Y' → 1

The encoder learns from the unique values using .fit() and applies the transformation using .transform(). fit_transform() does both.

Example:

Before encoding:
df['Gender'] = ['F', 'M', 'M', 'F']
df['Loan_Status'] = ['Y', 'N', 'Y', 'N']

After:
df['Gender'] = [0, 1, 1, 0]
df['Loan_Status'] = [1, 0, 1, 0]

Caution: Label encoding is only appropriate for **binary** or **ordinal** categories. If used on **non-ordinal multiple categories**, it might imply a false ranking (e.g., Red = 0, Green = 1, Blue = 2).

2. One-Hot Encoding (Alternative)

```
# pd.get_dummies(df['Gender'])  # if there are more than two categories
```

What is One-Hot Encoding?
This creates **dummy variables**—new columns for each category in the original column, where each row gets a 0 or 1 depending on the value.

Example:
Original:
df['Color'] = ['Red', 'Green', 'Blue']

One-hot encoded:
pd.get_dummies(df['Color'])

Output:

#	Blue	Green	Red
# 0	0	0	1
# 1	0	1	0
# 2	1	0	0

When to Use One-Hot Encoding:

- When the categorical variable is **non-binary** or **non-ordinal**
- For example: 'City', 'Color', 'Profession'

Usage:

```
df = pd.get_dummies(df, columns=['City'])
```

Tip for Practice:

Use the following to explore both encodings:

```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Sample Data
data = {'Gender': ['F', 'M', 'M', 'F', 'F'],
        'Loan_Status': ['Y', 'N', 'Y', 'N', 'Y']}
df = pd.DataFrame(data)

# Label Encoding
le = LabelEncoder()
df['Gender_Label'] = le.fit_transform(df['Gender'])
df['Loan_Label'] = le.fit_transform(df['Loan_Status'])

# One-Hot Encoding
df_onehot = pd.get_dummies(df['Gender'], prefix='Gender')
```

3. Feature Scaling

```
Standardization (Z-score)
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])

Normalization (Min-Max)
# from sklearn.preprocessing import MinMaxScaler
# mms = MinMaxScaler()
# df[['Age', 'Income']] = mms.fit_transform(df[['Age', 'Income']])
```

Feature scaling is a **data pre-processing technique** used to standardize or normalize the range of independent variables or features of data. Machine learning algorithms perform better or converge faster when features are on a relatively similar scale and close to normally distributed.

There are two commonly used feature scaling techniques:

1. Standardization (Z-score Normalization)

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df[['Age', 'Income']] = scaler.fit_transform(df[['Age', 'Income']])
```

What it does:

- Standardization transforms the data to have **zero mean** and **unit variance**.
- Formula:
  z=(x−μ)σz = \frac{(x - \mu)}{\sigma}z=σ(x−μ)
  Where:
  - xxx is the original value,
  - μ\muμ is the mean of the feature,
  - σ\sigmaσ is the standard deviation.

Why it's used:

- Useful when features follow a **Gaussian (normal) distribution**.
- Works well with algorithms like **SVM**, **Logistic Regression**, and **KNN**.

Example:
If Age has values [25, 30, 45] and mean = 33.3, std = 10:

- Transformed value for 25 would be ≈ -0.83

2. Normalization (Min-Max Scaling)

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
df[['Age', 'Income']] = mms.fit_transform(df[['Age', 'Income']])
```

What it does:

- Normalization rescales features to a range **[0, 1]**.
- Formula:
  X_{norm} = [(X − Xmin) / (Xmax − Xmin)]

Why it's used:

- Useful for algorithms that compute **distance-based** metrics like **KNN** or **Neural Networks**.
- Preserves the **shape of the original distribution**.

Example:
If Income ranges from 40K to 120K, and a sample is 60K:

(60K−40K)/(120K−40K) = 20K/80K =0.25

4. Outlier Detection and Treatment

Using **Z-Score** method:

```
from scipy.stats import zscore

z_scores = zscore(df[['Age', 'Income']])
outliers = (np.abs(z_scores) > 3).any(axis=1)
print("Outliers:\n", df[outliers])
```

Treat outliers (e.g., cap them):

```
df['Income'] = np.where(df['Income'] > 2.5, 2.5, df['Income'])
```

5. Feature Selection (Optional)

Using Variance Threshold

```
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(threshold=0.1)
features = selector.fit_transform(df[['Age', 'Income', 'Gender']])
```

Feature Selection is a critical preprocessing step in machine learning. It involves selecting only the most relevant features (or columns) from the dataset that contribute the most to the prediction variable or output. This helps in:

- Reducing overfitting
- Improving model accuracy
- Decreasing training time

Using Variance Threshold

```
from sklearn.feature_selection import VarianceThreshold

selector = VarianceThreshold(threshold=0.1)
features = selector.fit_transform(df[['Age', 'Income', 'Gender']])
```

What does VarianceThreshold do?

VarianceThreshold is a **filter-based** feature selection technique that removes features with low variance. The idea is:

- A **feature with low variance** does **not change much across samples**, so it likely **does not carry useful information** for distinguishing between outputs.
- It **drops features** whose variance is **below a given threshold**.

Parameter:

- threshold: A float value that defines the **minimum variance** a feature must have to be kept.
    - If threshold = 0.1, only features with variance ≥ 0.1 will be retained.
    - Default is 0, meaning features with **zero variance** (i.e., same value for all rows) are dropped.

Example:

Imagine this dataset:

Age	Income	Gender
25	50000	0
25	50000	0
25	50000	0
25	50000	0

- Variance of Age, Income, Gender = 0 (no variation at all).
- These columns will be dropped **if threshold = 0**.

Now, if the dataset is:

Age	Income	Gender
25	50000	0
30	55000	1
45	80000	1
50	90000	0

The variance will be:

- Age → High
- Income → High
- Gender → Lower (since it only varies between 0 and 1)

If threshold = 0.1, the column Gender might be dropped depending on actual calculated variance.

Output:

The fit_transform() method returns a **NumPy array** of the selected features. For example, if only Age and Income passed the threshold, it returns:

```
array([
    [25, 50000],
    [30, 55000],
    [45, 80000],
    ...
])
```