Complete Guide to Data Preprocessing for Machine Learning With Theoretical Notes + Practical Python Code

🔷 PART 1 – Basic Missing Value Handling (Detailed Explanation)

🔹 Step 1: Load the Dataset

```
import pandas as pd

dataset = pd.read_csv("loan.csv")
```

- **What it does**: Loads a CSV file into a pandas DataFrame, which is the core data structure for data analysis in Python.
- **Why it's important**: Before preprocessing, you need to bring the data into memory so you can manipulate and explore it.

🔹 Step 2: Initial Data Exploration

```
dataset.head()

dataset.shape
```

- dataset.head() displays the **first 5 rows**, giving you a snapshot of the data.
- dataset.shape returns a tuple (rows, columns), telling you the **size of the dataset**.

✅ **Tip**: Use .info() and .describe() too for deeper initial understanding.

🔹 Step 3: Check for Missing Values

```
dataset.isnull()

dataset.isnull().sum()

dataset.isnull().sum().sum()

dataset.notnull().sum()
```

Function	Purpose
.isnull()	Returns True for NaN cells, False otherwise
.isnull().sum()	Number of missing values per column
.isnull().sum().sum()	Total missing values in entire dataset

Function	Purpose
.notnull().sum()	Number of non-missing (valid) values per column

🧠 **Why this matters**: Detecting null values is the first step in handling incomplete data. Ignoring this step may lead to errors in analysis or model training.

🔹 **Step 4: Check Percentage of Missing Values**

```
(dataset.isnull().sum().sum()) / (dataset.shape[0] * dataset.shape[1]) * 100

(dataset.isnull().sum() / dataset.shape[0]) * 100
```

- First line: **Total % of missing cells** in the dataset.
- Second line: **% of missing values in each column**.

📌 **Insight**: If a column has more than 30–40% missing values, it may be a good candidate for dropping (depending on context).

🔹 **Step 5: Visualize Missing Data**

```
import seaborn as sns

import matplotlib.pyplot as plt


sns.heatmap(dataset.isnull(), cbar=False, cmap='viridis')

plt.show()
```

- **Why use a heatmap**: It provides an at-a-glance view of where the missing data occurs.
- **Interpretation**: Light-colored blocks indicate missing values.

🎯 **Use-case**: Helps in identifying patterns, like if missing values are clustered in certain columns or rows.

🔹 **Step 6: Drop Column with Many Missing Values**

```
dataset.drop(columns=["Credit_History"], inplace=True)
```

- Removes entire column(s) from the dataset.

- Use inplace=True to modify the original DataFrame.

⚠️ **Caution**: Use this only if the column has too many missing values or is not important.

🔹 **Step 7: Drop All Remaining Rows with Missing Values**

dataset.dropna(inplace=True)

- Removes rows that contain even a single NaN.
- **Effective when**: You have few missing rows, or the dataset is large enough that losing some rows doesn't affect model performance.

⛔ **Don't use** if:

- The dataset is small
- Missing values are in informative rows

🔹 **Step 8: Final Check for Missing Values**

dataset.isnull().sum()

- Confirms if the dataset is clean.
- Expected output: A series of zeros for each column, indicating no missing values left.

✅ **Summary of Best Practices**

Situation	Best Handling
Few missing values	Drop rows (dropna())
Column has many missing values	Drop column (drop())
Categorical feature with missing	Fill with mode
Numeric feature with missing	Fill with mean/median
Time-series data	Use forward/backward fill
Visual inspection	Use heatmaps for pattern detection

🔷 PART 2 – Filling Missing Values Properly (Detailed Explanation)

When your dataset has missing values (NaNs), dropping them may not always be the best option—especially if the dataset is small or the data is important. Instead, we **fill** them intelligently using techniques based on the column type.

🔹 Step 1: Avoid Blind Filling with Numbers

```
# dataset.fillna(10).head(10)
```

❌ **Why this is incorrect**:

- It fills **all missing values** with the number 10 — even in **textual (categorical)** columns like 'Gender' or 'Married'.
- It causes **data type mismatches** and makes analysis meaningless.

🧠 **Key Insight**: Never use fixed numeric fills (fillna(10), fillna(0)) **without checking the column type**. It can destroy the integrity of your dataset.

🔹 Step 2: Check Column Types Before Filling

```
dataset.info()
```

- This method gives:
  - Column names
  - Number of non-null entries
  - Data types (int64, float64, object)
- Use this to decide **how to fill missing values**.

📌 **Strategy Based on Column Type**:

Type	Recommended Fill
Numerical (int, float)	Mean or Median
Categorical (object/str)	Mode (most frequent)

🔹 **Step 3: Try Forward/Backward Fill (Optional)**

```
dataset.fillna(method="ffill", axis=0)

dataset.fillna(method="bfill", axis=0)
```

📝 **Explanation**:

- **Forward Fill (ffill)**: Uses the previous non-null value to fill the missing value.
- **Backward Fill (bfill)**: Uses the next non-null value.

📈 **Use-case**: Ideal for **time-series or sequential** data where data continuity matters (e.g., stock prices, temperature readings).

❌ Not recommended for **unordered** or **categorical** data — it can create false assumptions.

🔹 **Step 4: Fill Specific Column Using Mode (for Categorical Data)**

```
dataset["Gender"] = dataset["Gender"].fillna(dataset["Gender"].mode()[0])
```

✅ **Explanation**:

- mode() returns the most frequent value.
- This fills only the "Gender" column's NaN values using that most frequent value.

📌 **Use-case**: Ideal for **categorical features** like Gender, Education, etc., where the mode is representative of the majority.

🔹 **Step 5: Fill All Categorical Columns with Mode**

```
for col in dataset.select_dtypes(include="object").columns:
  dataset[col] = dataset[col].fillna(dataset[col].mode()[0])
```

🧠 **What this does**:

- Selects all columns of type "object" (i.e., text/categorical).
- Fills each with its own mode (most frequent value).

🔥 **Why it's efficient**:

- Automates filling missing values in all text columns.
- Avoids manual repetition.

✅ **Good practice** in real-world datasets where you have dozens of categorical variables.

🔹 **Step 6: Final Missing Value Check After Filling**

dataset.isnull().sum()

- Shows if **any missing values are left** after your filling process.
- Should ideally return **0 for every column**.

✅ **Summary of Best Practices**

Situation	Best Practice
Numerical column	Use **mean** or **median**
Categorical column	Use **mode**
Sequential data	Use **ffill** or **bfill**
Automated filling for all object columns	Use loop with fillna(mode())
Avoid	fillna(0) or fillna("Unknown") blindly

🚫 **Common Mistakes to Avoid**

1. **Filling all columns the same way** – Different columns require different logic.
2. **Not checking data type** before filling – Categorical vs numeric handling is different.
3. **Using forward fill on unordered data** – Can create misleading patterns.
4. **Forgetting to check after filling** – Always do a .isnull().sum() check at the end.

🔷 PART 3 – Imputation Using SimpleImputer (Detailed Explanation)

The SimpleImputer class from sklearn.impute is a powerful tool that lets you fill missing values in a **systematic, consistent, and scalable** way. It's especially useful when you want to apply the same strategy (mean, median, or mode) to many columns.

🔹 Step 1: Import Required Libraries

```
import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.impute import SimpleImputer
```

🧰 **What these libraries do**:

- pandas: For data manipulation and reading CSV files.
- seaborn / matplotlib: For visualizing data, trends, and missing values.
- SimpleImputer: Automatically fills missing values with chosen strategies like **mean**, **median**, or **most_frequent**.

🔹 Step 2: Load and Explore the Dataset

```
dataset = pd.read_csv("loan.csv")

dataset.head()

dataset.isnull().sum()

dataset.info()
```

🧠 **Goal**: Get an overview of:

- Which columns have missing values?
- What are their data types?

- Are they numerical or categorical?

💉 Use .select_dtypes() to isolate float or int columns for numerical imputation.

🔹 **Step 3: Identify Float Columns for Imputation**

```
dataset.select_dtypes(include="float64").columns
```

✅ **Why this step is important**:

- You should only apply mean/median to **numeric columns**.
- Applying these to strings or categories will result in errors.

💡 **You can also use**:

```
numerical_cols = dataset.select_dtypes(include=['int64', 'float64']).columns
```

🔹 **Step 4: Apply Mean Imputation using SimpleImputer**

```
from sklearn.impute import SimpleImputer


si = SimpleImputer(strategy="mean")


ar = si.fit_transform(dataset[[
  'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term', 'Credit_History'
]])
```

📌 **What this does**:

- strategy="mean": Tells the imputer to fill missing values with the **column mean**.
- fit_transform(): Fits the imputer (learns the means) and applies the transformation in one go.
- Returns a **NumPy array** of the imputed values.

🧠 **Other strategy options**:

Strategy	Fills With	Use Case
"mean"	Average of column	Numeric, normally distributed data

Strategy	Fills With	Use Case
"median"	Middle value of column	Numeric with outliers
"most_frequent"	Most common value	Categorical variables
"constant"	User-defined value	For fixed replacements like 0 or "NA"

🔹 **Step 5: Convert Result Back to DataFrame**

```
new_dataset = pd.DataFrame(ar,
columns=dataset.select_dtypes(include="float64").columns
)
```

💡 **Why this step matters**:

- SimpleImputer returns a **NumPy array**, not a DataFrame.
- You convert it back to DataFrame so:
  - You can analyze it easily
  - You can merge it back with the rest of the dataset

✅ **Best Practice**: Always give column names when creating a DataFrame from NumPy arrays.

🔹 **Step 6: Check Cleaned Data**

```
new_dataset.isnull()
new_dataset.isnull().sum()
```

🎯 **Goal**:

- Confirm that missing values have been properly filled.
- All missing value counts should now be zero.

✅ **Summary of Best Practices**

Step	What to Do	Why It's Important
1	Use SimpleImputer	More scalable and cleaner than manual .fillna()

Step	What to Do	Why It's Important
2	Apply only on numeric columns	Mean/median don't work on strings
3	Choose strategy based on data	Mean: normal, Median: skewed
4	Convert back to DataFrame	To keep your data organized
5	Check results	Always validate preprocessing steps

🚫 Common Mistakes to Avoid

1. **Applying imputation to the wrong column types**
   → Causes errors or incorrect fills.
2. **Not checking if missing values were really filled**
   → Always use .isnull().sum() to verify.
3. **Forgetting to assign the result back**
   → fit_transform() returns a new object. It doesn't modify the original.
4. **Using fit() on test data**
   → Only fit on **training data** to avoid data leakage.

🔷 PART 4 – Encoding Techniques (Detailed Explanation)

Many machine learning models work only with **numerical data**. So, we must convert **categorical features** (like "Gender", "City", "Education") into numbers using **encoding techniques**.

We'll cover:

- One-Hot Encoding
- Label Encoding
- Ordinal Encoding
- Manual Mapping

🔹 1. One-Hot Encoding using sklearn

✅ What is One-Hot Encoding?

It converts each category into a **new binary column (0 or 1)**.
For example:

Gender: Male, Female → Gender_Male, Gender_Female

✅ Code:

```
from sklearn.preprocessing import OneHotEncoder

import pandas as pd


# Sample categorical data

en_data = pd.DataFrame({

  "Gender": ["Male", "Female", "Female", "Male"],

  "Married": ["Yes", "No", "Yes", "No"]

})


ohe = OneHotEncoder()
```

```
ar = ohe.fit_transform(en_data).toarray()


# Get encoded feature names manually (or use ohe.get_feature_names_out())
df_encoded = pd.DataFrame(ar, columns=["Gender_Female", "Gender_Male", "Married_No", "Married_Yes"])
```

✅ **Output**: Each category gets its own 0/1 column.

⚠️ **Problem: Dummy Variable Trap**

If you have **n categories**, One-Hot Encoding gives you **n columns**, but **one column is redundant**. To avoid **multicollinearity**, use:

```
ohe = OneHotEncoder(drop="first")
```

This removes one dummy column from each feature to avoid linear dependency.

📌 **When to Use One-Hot Encoding:**

Use Case	Yes/No
Categorical data with **no natural order** (e.g., Gender, City)	✅ Yes
Large number of unique categories	❌ Avoid – will explode dimensionality

---

🔹 **2. Label Encoding**

✅ **What is Label Encoding?**

It converts each category into a **single integer**:

["dog", "cat", "rabbit"] → [1, 0, 2]

✅ **Code Example:**

```
from sklearn.preprocessing import LabelEncoder


df = pd.DataFrame({"name": ["yash", "robot", "car", "laptop", "rocket"]})
```

```
le = LabelEncoder()

df["en_name"] = le.fit_transform(df["name"])
```

⚠️ **Issue**: The numbers may be **misinterpreted as ordered**, even if the categories are not.

📌 **When to Use Label Encoding:**

Condition	Recommendation
Target variable (e.g., Yes/No, Male/Female)	✅ Yes
Input features with unordered categories	❌ Avoid – may mislead model
For decision trees or XGBoost	✅ Fine – they handle integer-encoded categories well

---

🔹 **3. Label Encoding on Real Dataset**

```
dataset = pd.read_csv("loan.csv")

le = LabelEncoder()

dataset["Property_Area_en"] = le.fit_transform(dataset["Property_Area"])
```

🔍 **Example**: If Property_Area = ["Urban", "Rural", "Semiurban"], the output may be:

- Urban → 2
- Rural → 1
- Semiurban → 0
  ...but these values **have no order**, which may mislead some models.

🔹 **4. Ordinal Encoding using OrdinalEncoder**

✅ **What is Ordinal Encoding?**

It encodes categories **based on a defined order**:

["s", "m", "l", "xl"] → [0, 1, 2, 3]

✅ **Code Example:**

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

df = pd.DataFrame({"Size": ["s", "m", "l", "xl", "s"]})

ord_data = [["s", "m", "l", "xl"]]
oe = OrdinalEncoder(categories=ord_data)

df["Size_en"] = oe.fit_transform(df[["Size"]])
```

✅ Use **categories=[[...]]** to ensure the correct order.

🔹 **5. Manual Ordinal Mapping**

Sometimes you want full control, so you can use Python's map():

```
ord_data1 = {"s": 0, "m": 1, "l": 2, "xl": 3}
df["Size_en_map"] = df["Size"].map(ord_data1)
```

📌 Great for custom orders like ["low", "medium", "high"].

🔹 **6. Ordinal Encoding on Real Dataset**

```
from sklearn.preprocessing import OrdinalEncoder
dataset = pd.read_csv("loan.csv")

oen = OrdinalEncoder(categories=[["Rural", "Semiurban", "Urban"]])
dataset["Property_Area_en"] = oen.fit_transform(dataset[["Property_Area"]])
```

This encodes:

- Rural → 0

- Semiurban → 1
- Urban → 2

✅ This is valid only because the order makes sense in some socio-economic contexts.

✅ **Summary Table**

Technique	Description	Use Case	Handles Order?
One-Hot Encoding	Binary columns for each category	Unordered categories	❌
Label Encoding	Converts to integers (0,1,2,...)	Target variables, tree models	❌
Ordinal Encoding	Encodes with order	Ordered categories (e.g., Size)	✅
Manual Mapping	Custom integer mapping	Specific ordered logic	✅

🚫 **Common Mistakes to Avoid**

1. ❌ **LabelEncoding unordered data** → Misleads algorithms like Linear Regression, KNN.
2. ❌ **One-hot encoding too many categories** → Creates high-dimensional sparse data.
3. ❌ **Ignoring order in ordinal data** → Leads to incorrect encoding logic.
4. ❌ **Using drop="first" blindly** → Useful, but not always necessary unless you're doing linear regression.

🔷 PART 5 – Outlier Detection using IQR (Detailed Explanation)

🔹 What is an Outlier?

An **outlier** is a data point that significantly differs from the rest of the data.
It might be:

- A valid extreme observation (e.g., a CEO's salary)
- A data entry error (e.g., extra zero)
- An influential value that distorts **mean, standard deviation, and model accuracy**

📉 Why Remove Outliers?

- They **skew statistical analysis**
- Hurt **regression-based** and **distance-based** models (like KNN, K-Means)
- Can **introduce bias** if not handled properly

✅ Step-by-Step: Outlier Detection using IQR

🔹 Step 1: Visual Detection with Boxplot

```
import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd


dataset = pd.read_csv("loan.csv")


sns.boxplot(dataset["ApplicantIncome"])

plt.show()
```

📌 **Boxplot Insights**:

- Central box: 25th percentile (Q1) to 75th percentile (Q3)
- Middle line: Median
- Whiskers: Range of normal data
- Dots outside whiskers: **Outliers**

📊 Helps visually identify which data points may need attention.

🔹 **Step 2: Calculate IQR and Outlier Bounds**

```
q1 = dataset["ApplicantIncome"].quantile(0.25)

q3 = dataset["ApplicantIncome"].quantile(0.75)

IQR = q3 - q1


lower_bound = q1 - 1.5 * IQR

upper_bound = q3 + 1.5 * IQR
```

🧠 **Explanation**:

- **IQR (Interquartile Range)** = Q3 - Q1 → Measures spread of middle 50% data
- **Outliers** lie outside the range:
  Lower Bound = Q1 - 1.5 × IQR
  Upper Bound = Q3 + 1.5 × IQR

💡 This 1.5 rule is based on **Tukey's method** — a statistical rule of thumb.

🔹 **Step 3: Filter Outliers**

```
outliers = dataset[
   (dataset["ApplicantIncome"] < lower_bound) |
   (dataset["ApplicantIncome"] > upper_bound)
]


outliers.shape  # Check how many outliers are detected
```

📊 This step:

- Extracts all rows where ApplicantIncome is outside the IQR bounds
- Helps you **quantify** how many rows are affected

🔹 **Step 4: Remove Outliers**

```
dataset = dataset[
  (dataset["ApplicantIncome"] >= lower_bound) &
  (dataset["ApplicantIncome"] <= upper_bound)
]
```

✅ Keeps only data points **within the IQR range**.
Outliers are removed and dataset is now "cleaned."

🔹 **Step 5: Verify with Boxplot Again**

```
sns.boxplot(dataset["ApplicantIncome"])
plt.show()
```

📊 After removing outliers, the plot should:

- Have fewer dots outside whiskers
- Appear more balanced
- Be free of extreme skew

🔍 **Summary: IQR Method**

Term	Meaning
Q1	25th percentile
Q3	75th percentile
IQR	Interquartile Range = Q3 − Q1
Outlier Rule	Value < Q1 − 1.5×IQR or > Q3 + 1.5×IQR

✅ **Best Practices for Using IQR**

Situation	Recommendation
Skewed data (e.g., income, price)	IQR is better than Z-score
Small datasets	Be cautious — removing too much can lose valuable info
Numerical columns only	IQR works only for continuous variables
Visualization	Use boxplots before and after removal

⚠️ **Common Mistakes to Avoid**

1. ❌ **Applying IQR to categorical data** – Only use with numerical columns.
2. ❌ **Blindly removing all outliers** – Sometimes outliers are important!
3. ❌ **Not visualizing before/after** – Always use boxplots or histograms to confirm.
4. ❌ **Removing outliers across the whole dataset** – Instead, apply it **column by column** if needed.

🔷 PART 6 – Outlier Detection using Z-Score (Detailed Explanation)

🔹 What is a Z-Score?

The **Z-Score** tells you how many **standard deviations** a data point is away from the **mean** of the dataset.

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x: value of the data point
- μ: mean of the dataset
- σ: standard deviation

✅ Why Use Z-Score for Outlier Detection?

- Useful when data is **normally distributed**
- Helps detect values that are **far from the average**
- Common threshold: **Z-score > 3 or < -3** is considered an outlier

---

📊 Step-by-Step Guide: Detecting Outliers Using Z-Score

🔹 Step 1: Calculate Z-Scores

```
from scipy.stats import zscore

import pandas as pd


dataset = pd.read_csv("loan.csv")

dataset["z_score"] = zscore(dataset["ApplicantIncome"])
```

🧠 **What this does**:

- Computes the Z-score for every ApplicantIncome value
- Stores it in a new column "z_score" for inspection

🔹 **Step 2: Identify Outliers**

```
outliers = dataset[(dataset["z_score"] > 3) | (dataset["z_score"] < -3)]
print(outliers.shape)
```

✅ **Explanation**:

- Filters rows where the absolute Z-score exceeds 3
- These are values considered statistically rare or **extreme**

📌 **Tip**: You can try thresholds like ±2.5 or ±2 for less strict filtering.

🔹 **Step 3: Remove Outliers**

```
dataset = dataset[
  (dataset["z_score"] <= 3) &
  (dataset["z_score"] >= -3)
]
```

- Keeps only the **"normal" values** based on Z-score
- Outliers are effectively removed from the dataset

🔹 **Step 4: Drop the Z-Score Column (Optional Cleanup)**

```
dataset.drop(columns=["z_score"], inplace=True)
```

- Deletes the helper column after its job is done
- Keeps your dataset clean and free from temporary features

✅ **Summary Table**

Concept	Explanation
Z-score	Standardized value showing how far a point is from the mean

Concept	Explanation
Threshold	±3 (or ±2.5/2 for less strict)
When to Use	On **normally distributed**, continuous numerical columns
Tool	scipy.stats.zscore()

🧠 When to Use Z-Score vs. IQR?

Criteria	IQR Method	Z-Score Method
Works on skewed data	✅ Yes	❌ No (assumes normal distribution)
Sensitive to extreme outliers	❌ Less	✅ More
Easy to visualize	✅ Boxplot	❌ Not always intuitive
Best for	Income, Price	Test scores, Sensor readings

✅ Best Practices

1. Always **visualize your data first** (histogram/boxplot) to check distribution.
2. Use **Z-score only on numerical, normally distributed columns**.
3. Don’t blindly drop outliers — consider:
    - Is it a valid extreme?
    - Does it carry business importance?
4. Combine **Z-score with domain knowledge** to avoid losing meaningful data.

🚫 Common Mistakes to Avoid

Mistake	Why it's wrong
Applying Z-score on skewed data	Z-score assumes normal distribution
Using Z-score for categorical columns	Only valid for numerical values

Mistake	Why it's wrong
Not resetting index after dropping rows	Can cause confusion when merging or plotting
Blindly dropping outliers	May remove meaningful edge-cases (e.g., high-income customers)

🔷 PART 7 – Feature Scaling (Standardization & Normalization)

🔹 Why Scale Features?

Many machine learning models are **sensitive to the range and magnitude of features**. Without scaling:

- Features with **large numeric values** can **dominate** others
- Algorithms that rely on **distance (like KNN, SVM, K-Means)** give **biased results**
- Models like **Gradient Descent** converge **slowly or incorrectly**

📌 **Example**: In a dataset with Age (0–100) and Income (0–100,000), income would overwhelm age unless scaled properly.

🔵 Standardization (Z-score Normalization)

✅ What is Standardization?

Standardization transforms data to have:

- **Mean = 0**
- **Standard deviation = 1**

$$z = \frac{x - \mu}{\sigma}$$

Where:

- x: original value
- μ: mean of the column
- σ: standard deviation

🔹 Code Example:

```
from sklearn.preprocessing import StandardScaler


scaler = StandardScaler()
```

```
scaler.fit(dataset[["ApplicantIncome"]])  # Learn mean and std

standardized = scaler.transform(dataset[["ApplicantIncome"]])  # Scale the values

print(standardized[:5])
```

✅ The output will have:

- Mean close to 0
- Standard deviation close to 1

📌 **When to Use Standardization:**

Situation	**Use Standardization?**
Data is **normally distributed**	✅ Yes
Algorithm is **distance-based or gradient-based**	✅ Yes
Outliers are present	⚠️ Use with caution (use RobustScaler instead)
Features vary in scale (e.g., Age vs Salary)	✅ Absolutely

🟢 **Normalization (Min-Max Scaling)**

✅ **What is Normalization?**

Normalization rescales data into a **fixed range**, usually **[0, 1]**:

$$x_{norm} = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

Where:

- x: original value
- x_{min}, x_{max}: min and max of the column

🔹 **Code Example:**

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(dataset[["ApplicantIncome"]])  # Learn min and max

normalized = scaler.transform(dataset[["ApplicantIncome"]])
print(normalized[:5])
```

✅ Output: All values are scaled between **0 and 1**

📌 **When to Use Normalization:**

Situation	**Use Normalization?**
Algorithm uses **Euclidean or Manhattan distance** (KNN, K-Means)	✅ Yes
You want all features on **exactly same scale**	✅ Yes
Data contains **no extreme outliers**	✅ Yes
Neural networks (especially in deep learning)	✅ Often preferred

---

📊 Visualization: Standardized vs Normalized

```
import matplotlib.pyplot as plt


plt.figure(figsize=(10, 4))


plt.subplot(1, 2, 1)
plt.hist(standardized, bins=30, color='blue', label='Standardized')
plt.title("Standardized Data")
plt.legend()


plt.subplot(1, 2, 2)
plt.hist(normalized, bins=30, color='green', label='Normalized')
plt.title("Normalized Data")
plt.legend()


plt.tight_layout()
plt.show()
```

🧠 This helps you **see the difference**:

- Standardized data: Centered around 0, spreads over negative and positive values
- Normalized data: Rescaled to fit between 0 and 1

✅ Summary Table – Standardization vs Normalization

Technique	Output Range	Centered at 0?	Sensitive to Outliers	Best For
Standardization	No limit	✅ Yes	⚠️ Yes	Normal distributions, SVM, Logistic Regression
Normalization	[0, 1]	❌ No	✅ Yes	KNN, Neural Networks, K-Means

---

🧠 Additional Tips

- Use **StandardScaler** when you don't need bounded values but want normalized behavior.
- Use **MinMaxScaler** when all features must be within the same fixed scale (e.g., image pixels).
- Use **RobustScaler** if your data has many outliers (it uses median and IQR).
- Scale features **after splitting** the dataset to avoid **data leakage**.

🚫 Common Mistakes to Avoid

Mistake	Why it’s wrong
Scaling before splitting data	Leaks info from test set into training set
Not scaling for KNN/SVM	Distorts distance calculations
Normalizing with outliers present	Pulls most values close to 0
Using wrong scaler type	Different algorithms benefit from different scalers

🔷 PART 8 – Handling Duplicate Values (Detailed Explanation)

🔹 What Are Duplicate Values?

- A **duplicate** row is an exact **copy** of another row in the dataset.
- Duplicates can occur due to:
  - **Data entry errors**
  - **Merging datasets** multiple times
  - **Scraping or importing** repeated entries
  - Lack of proper **row-level unique identifiers**

🧠 Why remove them?

- They **bias analysis**
- Affect **summary statistics**
- Mislead **machine learning models**

🧪 Step-by-Step Guide to Detect and Remove Duplicates

🔹 Step 1: Create Sample DataFrame (Example)

```
import pandas as pd


data = {
   "name": ["a", "b", "c", "d", "a", "c"],
   "eng": [8, 7, 5, 8, 8, 4],
   "hindi": [2, 3, 4, 5, 2, 6]
}


df = pd.DataFrame(data)
```

✅ **Creates a DataFrame** where rows 0 and 4 are duplicates ("a", 8, 2 appears twice), and "c" appears twice with different scores — not considered a duplicate row unless all column values match.

🔹 **Step 2: Drop Duplicates from Sample**

df.drop_duplicates(inplace=True)

- Removes **all completely identical rows**
- inplace=True applies the change directly to df

🧠 **Note**: If only one column is duplicated, this method **won't remove** it unless the entire row is the same.

🔹 **Step 3: Load and Inspect Real Dataset**

dataset = pd.read_csv("loan.csv")

dataset.head(3)

- Load your actual dataset
- Check the first few rows to visually inspect any potential repetitions

---

🔹 **Step 4: Identify Duplicate Rows**

dataset.duplicated()

- Returns a **Boolean Series** indicating if a row is a duplicate of a **previous** row
- Returns True for all **subsequent** occurrences of duplicates (not the first one)

🔹 **Step 5: Count Duplicate Rows**

dataset.duplicated().sum()

📌 Tells you how many **duplicate rows** exist in the dataset.

🔹 **Step 6: Check Shape Before Removal**

print(dataset.shape)

✅ Knowing the shape before and after helps verify how many duplicates were removed.

🔹 **Step 7: Remove Duplicate Rows**

dataset.drop_duplicates(inplace=True)

- Removes **all repeated rows**
- Keeps the **first occurrence** only

🔹 **Step 8: Confirm Shape After Removal**

print(dataset.shape)

- Compare with the earlier shape to ensure duplicate rows are successfully dropped

---

✅ **Summary Table**

Step	Function	Purpose
Check duplicates	dataset.duplicated()	Returns True for repeated rows
Count them	.duplicated().sum()	Number of repeated rows
Drop them	.drop_duplicates()	Removes all but the first occurrence
Apply permanently	inplace=True	Saves change in same DataFrame

---

🔍 **Additional Tips**

1. **Partial Duplicates**: To check duplicates in specific columns:

dataset.duplicated(subset=['Gender', 'LoanAmount'])

➤ Use when some columns repeat but others don’t.

2. **Drop duplicate rows only from certain columns**:

dataset.drop_duplicates(subset=['Name', 'DOB'], inplace=True)

3. **Sort before dropping**:

- Helps retain the most recent or relevant version of the duplicate row.
- Example:

```
dataset.sort_values(by="Loan_ID", ascending=False, inplace=True)

dataset.drop_duplicates(subset="ApplicantIncome", keep="first", inplace=True)
```

🚫 Common Mistakes to Avoid

Mistake	Why It's a Problem
Ignoring duplicates	Leads to biased statistics or model overfitting
Dropping duplicates without checking	May lose important repeated info (e.g., time-series data)
Assuming .duplicated() marks **all** duplicates	It only marks second and later occurrences
Using drop_duplicates() without inplace=True	Changes won't persist unless reassigned or saved

🔷 PART 9 – Changing Data Types (Detailed Explanation)

🔹 Why Change Data Types?

Changing or correcting data types is essential because:

- **Wrong data types** prevent statistical and ML operations.
- **Numeric operations** can't be done on **strings** (e.g., "3+" or "25 years").
- Proper types improve **memory efficiency** and **code performance**.

🧪 Step-by-Step Guide: Handling and Converting Data Types

🔹 Step 1: Load the Dataset

```
import pandas as pd

dataset = pd.read_csv("loan.csv")
```

- Loads the dataset into a pandas DataFrame named dataset.

🔹 Step 2: Preview the Dataset

```
dataset.head(3)
```

- Shows the **first 3 rows** — a quick check on what data looks like.
- Useful to identify issues like values stored as strings (e.g., '3+' instead of an int).

🔹 Step 3: Check Dataset Info

```
dataset.info()
```

✅ Shows:

- Column names
- Number of non-null entries
- **Data types** (int64, float64, object)
- Helps identify where type conversion is needed

🧠 Look for:

- Numerical columns stored as **object** (string)
- Date/time values not in **datetime** format
- Mixed-type columns like "3+" in "Dependents"

🔹 **Step 4: Check for Missing Values**

dataset.isnull().sum()

- Before conversion, it's best to fill missing values — otherwise astype() may fail.

🔹 **Step 5: Explore Categorical Column with Numeric Meaning**

dataset["Dependents"].value_counts()

🧠 The Dependents column may contain values like:

'0', '1', '2', '3+'

- '3+' is **not** numeric — needs to be cleaned.

🔹 **Step 6: Fill Missing Values Using Mode**

dataset["Dependents"] =
dataset["Dependents"].fillna(dataset["Dependents"].mode()[0])

- Ensures there are **no NaNs** before performing string replacement or conversion.

🔹 **Step 7: Replace '3+' with Integer '3'**

dataset["Dependents"] = dataset["Dependents"].replace("3+", "3")

🎯 Converts all '3+' entries to '3', making the entire column suitable for numeric conversion.

🔹 **Step 8: Convert to Integer Type**

dataset["Dependents"] = dataset["Dependents"].astype("int64")

✅ Column is now stored as integer — ready for numerical analysis or ML.

🧠 .astype("int64") throws an error if:

- There are still strings or NaNs in the column
- You try to convert improperly formatted values

🔹 Step 9: Final Data Check

dataset.info()

- Verify:
    - No missing values
    - Column is now stored as int64
    - Ready for use in models or stats

✅ Summary Table

Step	Action	Purpose
.value_counts()	Explore column values	See if weird entries like '3+' exist
.fillna(mode)	Fill missing values	Prepares for clean conversion
.replace("3+", "3")	Normalize values	Convert text to numeric representation
.astype("int64")	Convert type	Enables math and model input
.info()	Confirm result	Final validation

🔍 Additional Examples of Type Conversions

Use Case	Example Code
Convert to float	df["col"] = df["col"].astype("float64")
Convert to int	df["col"] = df["col"].astype("int")
Convert to string	df["col"] = df["col"].astype("str")
Convert to datetime	df["col"] = pd.to_datetime(df["col"])

🧠 Best Practices

1. Always **handle missing values first** before type conversion.
2. Clean inconsistent string values (e.g., '3+') before applying astype().
3. Use .info() and .value_counts() to check before and after conversion.
4. Avoid converting to int if the column contains **NaN** — use float instead.

🚫 Common Mistakes to Avoid

Mistake	Why It's a Problem
Converting strings like '3+' directly to int	Raises ValueError
Using .astype() on columns with missing values	Fails on NaNs (use float64 if unsure)
Forgetting to replace text representations	Leads to incorrect or failed conversions
Assuming all values are clean	Unexpected formatting will crash your script

🔷 PART 10 – FunctionTransformer & Outlier Removal (Detailed Explanation)

🔹 Why Use FunctionTransformer?

Many machine learning models **assume that input features follow a normal distribution**. But in real-world datasets:

- Income, price, and transaction data are usually **right-skewed**.
- Applying a **log transformation** can make the data **more normal-like**.
- This improves **model performance**, especially for linear regression, logistic regression, and neural networks.

🧪 Step-by-Step Guide

🔹 Step 1: Import Required Libraries

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

- pandas → for loading/manipulating data
- seaborn/matplotlib → for visualization
- numpy → for mathematical operations

🔹 Step 2: Load the Dataset

```
dataset = pd.read_csv("loan.csv")
dataset.head(3)
```

- Loads the dataset and shows the first 3 rows to get an overview.

🔹 Step 3: Check for Missing Values

```
dataset.isnull().sum()
```

✅ Always clean missing values **before applying transformation** to avoid mathematical errors (e.g., log of NaN = error).

🔹 **Step 4: Visualize Distribution (Before Cleaning)**

```
sns.distplot(dataset["CoapplicantIncome"])

plt.show()
```

📊 A **right-skewed** distribution will appear as a curve concentrated on the left with a long tail on the right.
This tells us: ✔️ Log transformation is appropriate.

🔹 **Step 5: Detect Outliers Using IQR**

```
q1 = dataset["CoapplicantIncome"].quantile(0.25)

q3 = dataset["CoapplicantIncome"].quantile(0.75)

iqr = q3 - q1


min_r = q1 - 1.5 * iqr

max_r = q3 + 1.5 * iqr
```

- **IQR (Interquartile Range)** helps identify extreme outliers that are too far from the core distribution.

🔹 **Step 6: Remove Outliers**

```
dataset = dataset[dataset["CoapplicantIncome"] <= max_r]
```

✅ Keeps only values below the upper threshold (removes large outliers).

⚠️ **Note**: We usually only filter out **upper outliers** when the data is right-skewed.

🔹 **Step 7: Recheck Distribution**

```
sns.distplot(dataset["CoapplicantIncome"])

plt.title("After Outlier Removal")
```

plt.show()

🎯 Distribution should now be **less extreme** — but may still be skewed.

🔹 **Step 8: Apply Log Transformation using FunctionTransformer**

```
from sklearn.preprocessing import FunctionTransformer

ft = FunctionTransformer(func=np.log1p)
ft.fit(dataset["CoapplicantIncome"])  # Fit isn't required but safe
dataset["CoapplicantIncome_tf"] = ft.transform(dataset[["CoapplicantIncome"]])
```

🔍 np.log1p(x) applies:

log⁡(1+x)\log(1 + x)log(1+x)

✅ This handles 0 values safely (unlike np.log(x) which fails at 0).

🔹 **Step 9: Visual Comparison (Before vs After)**

```
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
sns.distplot(dataset["CoapplicantIncome"])
plt.title("Before Transformation")

plt.subplot(1, 2, 2)
sns.distplot(dataset["CoapplicantIncome_tf"])
plt.title("After Log Transformation")

plt.tight_layout()
plt.show()
```

📊 You'll observe:

- **Before**: Highly skewed curve
- **After**: Smoother, more symmetric distribution — closer to normal

---

✅ Summary Table

Step	Action	Purpose
1	Visualize distribution	Identify skew
2	Use IQR method	Remove extreme values
3	Apply log1p	Normalize skewed data
4	Visualize after transform	Validate improvement

🧠 Best Practices

Tip	Explanation
Use np.log1p not np.log	Handles 0 safely
Apply on right-skewed features	Income, prices, etc.
Remove outliers before transform	Prevents distortion
Always visualize both before and after	Confirms the transformation worked
Don’t apply on negative values	log of negative = math error

🚫 Common Mistakes to Avoid

Mistake	Why It’s a Problem
Applying np.log() to 0	Causes math error
Not removing outliers first	Log compresses, but doesn’t fix outliers
Forgetting to create a new column	Overwrites raw data
Using log on already-normal data	Distorts good data

🔷 PART 11 – Feature Selection: Forward and Backward Elimination (Detailed Explanation)

🔹 Why Feature Selection?

Feature selection is the process of **choosing the most relevant features** (columns) from your dataset. It's critical because:

- ✅ Reduces **overfitting**
- ✅ Improves **model performance and accuracy**
- ✅ Shortens **training time**
- ✅ Makes models **easier to interpret**
- ✅ Removes **redundant or irrelevant** data

🔷 FORWARD SELECTION

✅ What is Forward Selection?

- Start with **no features**
- Add features **one by one**
- Keep adding the one that **improves model performance the most**
- Stop when adding more doesn't help

📌 Performance metric: Usually **AIC** (Akaike Information Criterion) or $\mathbf{R^2}$, depending on model

---

🔹 Code Example using statsmodels

```
import pandas as pd

import statsmodels.api as sm


def forward_selection(X, y):

  initial_features = []
```

```
remaining_features = list(X.columns)
best_features = []

while remaining_features:
    scores_with_candidates = []
    for candidate in remaining_features:
        model = sm.OLS(y, sm.add_constant(X[initial_features + [candidate]])).fit()
        aic = model.aic
        scores_with_candidates.append((aic, candidate))

    scores_with_candidates.sort()
    best_aic, best_candidate = scores_with_candidates[0]

    remaining_features.remove(best_candidate)
    initial_features.append(best_candidate)
    best_features.append((best_aic, list(initial_features)))

return best_features
```

🔍 **Explanation:**

- sm.OLS: Ordinary Least Squares Regression from statsmodels
- aic: Lower AIC = better model (penalizes complexity)
- Tracks and adds the best-performing features step-by-step

🔷 **BACKWARD ELIMINATION**

✅ **What is Backward Elimination?**

- Start with **all features**
- Remove the **least significant** feature (based on **p-value**)

- Repeat until all features are **statistically significant**

📌 Threshold: Common cutoff = **0.05** (5% significance level)

🔹 **Code Example:**

```
def backward_elimination(X, y, threshold=0.05):
  while True:
    model = sm.OLS(y, sm.add_constant(X)).fit()
    p_values = model.pvalues.iloc[1:]  # skip intercept
    max_p = p_values.max()

    if max_p > threshold:
      excluded_feature = p_values.idxmax()
      X = X.drop(columns=[excluded_feature])
    else:
      break

  return X.columns
```

🔍 **Explanation:**

- Checks p-values for each feature
- Drops the feature with the **highest p-value** (least statistically relevant)
- Stops when **all remaining features have p-values < 0.05**

✅ **Summary Table**

Method	Starts With	Process	Stops When	Best For
Forward Selection	No features	Add best one each time	No more performance gain	Small datasets

Method	Starts With	Process	Stops When	Best For
Backward Elimination	All features	Drop worst one each time	All p-values < 0.05	Statistical testing
Combined Stepwise	Mix of both	Add & remove dynamically	No performance or p-value improvement	Auto selection

---

🧠 Best Practices

Tip	Reason
Use statsmodels for interpretability	Gives full p-values, coefficients, AIC
Don’t mix with highly correlated features	It may include redundant columns
Check multicollinearity	Use VIF (Variance Inflation Factor)
Apply scaling beforehand if needed	Especially for linear regression

---

🚫 Common Mistakes to Avoid

Mistake	Problem
Using all features blindly	Increases overfitting, slows model
Ignoring p-values	May include statistically useless features
Using forward selection with too many variables	Becomes computationally expensive
Relying only on accuracy	Accuracy alone doesn’t reflect overfitting or bias

---

✅ **Note**: These techniques are often used for **linear models**, where **interpretability and feature relevance** are more important than black-box performance.

🔷 PART 12 – Feature Selection using Sequential Feature Selector (SFS)

🔹 What is Sequential Feature Selection (SFS)?

Sequential Feature Selection (SFS) is a **wrapper method** used to select the best set of features by:

- **Forward selection**: Start with none, add one feature at a time
- **Backward elimination**: Start with all, remove one feature at a time
- It evaluates combinations using **cross-validation and scoring metrics**

✅ It works with any **scikit-learn compatible model**.

💉 Step-by-Step Guide Using mlxtend

🔹 Step 1: Import Required Libraries

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from mlxtend.feature_selection import SequentialFeatureSelector
```

- mlxtend.feature_selection.SequentialFeatureSelector: The core tool for SFS
- You can also use **SVM, DecisionTree, RandomForest** instead of Logistic Regression

---

🔹 Step 2: Load Dataset

```
dataset = pd.read_csv("your_file.csv")
```

- Replace "your_file.csv" with your actual dataset

🔹 Step 3: Separate Features and Target

```
X = dataset.drop(columns=["target_column"])
y = dataset["target_column"]
```

📌 Replace "target_column" with the name of your target variable

✅ X contains the features

✅ y contains the labels

🔹 **Step 4: Initialize the Model**

```
model = LogisticRegression(max_iter=1000)
```

- You can use any estimator (SVC, DecisionTreeClassifier, etc.)
- max_iter=1000 ensures convergence for logistic regression

🔹 **Step 5: Configure the Sequential Feature Selector**

```
from mlxtend.feature_selection import SequentialFeatureSelector

sfs = SequentialFeatureSelector(
  model,
  k_features='auto',      # Or choose a number like 5 or 10
  forward=True,           # Set to False for backward selection
  floating=False,         # If True, allows dynamic inclusion/exclusion
  scoring='accuracy',     # Or 'f1', 'r2', etc.
  cv=5                    # 5-fold cross-validation
)
```

📌 **Parameter Breakdown:**

Parameter	Meaning
k_features	Number of features to select (can be 'auto' or int)
forward	True for forward selection, False for backward
floating	Allows dynamic adding/removing (like stepwise selection)
scoring	Evaluation metric ('accuracy', 'r2', 'f1', etc.)
cv	Number of cross-validation folds

🔹 **Step 6: Fit the Selector to the Data**

sfs.fit(X, y)

- Internally performs training, evaluation, and selection
- Uses cross-validation to ensure reliability

🔹 **Step 7: Retrieve Selected Features**

print("Selected feature indices:", sfs.k_feature_idx_)

print("Selected feature names:", sfs.k_feature_names_)

✅ Returns:

- Tuple of column indices (k_feature_idx_)
- Tuple of column names (k_feature_names_) — more readable

✅ **Summary Table**

Step	Action	Purpose
Initialize model	LogisticRegression()	Base model for feature selection
Configure SFS	Set parameters	Defines type of selection and scoring
Fit	sfs.fit(X, y)	Runs the selection process
Output	.k_feature_names_	Gets the best-performing features

🧠 **Benefits of Sequential Feature Selection**

Benefit	Explanation
Model-agnostic	Works with any sklearn-compatible model
Cross-validation based	More robust than simple p-value methods
Customizable scoring	Choose what matters most — accuracy, F1, etc.
Avoids overfitting	Selects optimal number of features

Benefit	Explanation
Floating mode (optional)	Tries to add/remove dynamically

🚫 Common Mistakes to Avoid

Mistake	Why It's a Problem
Using default k_features='auto' without inspecting	May select too many or too few features
Choosing wrong scoring metric	Skews feature selection toward wrong goal
Not scaling features (if needed)	Some models (e.g., logistic, SVM) are scale-sensitive
Using non-cleaned data	Missing values or duplicates can break the process

---

📌 Use Cases

- When you have **many features** but need to narrow down to the most effective ones
- When **model performance is more important than statistical significance**
- Works well in both **classification and regression problems**

🔷 PART 13 – Train-Test Split and Feature Scaling (Detailed Explanation)

🔹 Why Do We Split the Dataset?

Splitting your dataset into **training and testing sets** is essential because:

- You train your model on one portion (**training set**)
- And evaluate it on unseen data (**testing set**)
- This helps detect **overfitting** and ensures your model **generalizes well**

🔹 Why Scale Features?

Some machine learning models (like **KNN, SVM, Logistic Regression, Gradient Descent**) are **sensitive to feature magnitude**. Without scaling:

- Features with larger ranges dominate smaller ones
- Optimization may converge slowly or get stuck
- Distance-based models give biased results

✅ Scaling brings all features to a comparable range.

🧪 Step-by-Step: Train-Test Split & Scaling

🔹 Step 1: Import Required Libraries

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
```

- train_test_split → for splitting data into train and test sets
- StandardScaler → for standardizing numerical features

🔹 Step 2: Load the Dataset

```
iris_df = pd.read_csv('iris.csv')
```

🧠 We are using the famous **Iris dataset** here. You can use your own dataset similarly.

🔹 **Step 3: Separate Features and Target**

```
X = iris_df.drop('target', axis=1)  # Features
y = iris_df['target']           # Labels
```

- X: All columns except the target
- y: The column you're trying to predict

🔹 **Step 4: Train-Test Split**

```
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.2, random_state=42
)
```

Parameter	**Description**
test_size=0.2	20% test, 80% train
random_state=42	Ensures reproducibility
stratify=y *(optional)*	Preserves class ratio (for classification tasks)

🧠 Now:

- X_train, y_train: Used to **fit/train** the model
- X_test, y_test: Used to **evaluate** the model's performance

---

🔹 **Step 5: Feature Scaling (Standardization)**

```
scaler = StandardScaler()

# Fit on training data only
X_train = scaler.fit_transform(X_train)

# Use same transformation on test data
X_test = scaler.transform(X_test)
```

✅ **Important Rule**:

- Always **fit** the scaler on **training data only**
- Then **transform both** training and test data
- This prevents **data leakage**

🔹 **Step 6: (Optional) Output Scaled Data**

```
print("Scaled Training Set:")
print(X_train)


print("\nScaled Test Set:")
print(X_test)
```

✅ **Summary Table**

Step	Function	Purpose
Split	train_test_split()	Divide data into training/testing
Scale	StandardScaler()	Normalize feature ranges
Fit & transform	.fit_transform() on train	Learn and apply scaling
Transform only	.transform() on test	Apply same scaling logic

🧠 **Best Practices**

Tip	Why It's Important
Always split before scaling	Prevents information leakage
Scale only numerical columns	Encoding should be done first
Use random_state	For reproducibility
Use stratify=y (optional)	To maintain class distribution

🚫 Common Mistakes to Avoid

Mistake	Why It's a Problem
Scaling before splitting	Leaks test data info into model
Not scaling features	Some models will perform poorly
Using fit() on test data	Causes unrealistic evaluation results
Applying scaler to all columns (incl. categorical)	Should only be used on numerical data

✅ Final Summary – Data Preprocessing (All 13 Parts)

🔷 PART 1: Basic Missing Value Handling

- **Check for missing values** using .isnull().sum()
- Drop columns with many nulls using drop(columns=...)
- Drop rows using dropna() if losing them doesn't affect analysis
- Visualize missingness using sns.heatmap()

🔷 PART 2: Filling Missing Values Properly

- Avoid blindly using fillna(0) — check column type
- Use **mean/median** for numeric columns
- Use **mode** for categorical columns
- Use **ffill/bfill** for time-series continuity
- Verify missing values are filled with .isnull().sum()

🔷 PART 3: Imputation using SimpleImputer

- Use SimpleImputer for consistent and scalable filling
- Strategy options: "mean", "median", "most_frequent", "constant"
- Apply only on numeric columns and convert result to DataFrame
- Fit on training data; transform both train and test

🔷 PART 4: Encoding Techniques

- **One-Hot Encoding**: for nominal categorical variables (no order)
- **Label Encoding**: for target variables or tree models
- **Ordinal Encoding**: for ordered categories
- Use .replace() or map() for manual ordinal logic

🔷 PART 5: Outlier Detection using IQR

- IQR = Q3 - Q1; outliers lie outside **[Q1 - 1.5×IQR, Q3 + 1.5×IQR]**
- Use boxplot to visualize
- Remove outliers by filtering rows outside the IQR bounds

🔷 PART 6: Outlier Detection using Z-Score

- Z-score = (x - mean) / std deviation
- Outliers: z > 3 or z < -3
- Use scipy.stats.zscore() on numeric columns
- Remove and drop z-score column after filtering

🔷 PART 7: Feature Scaling (Standardization & Normalization)

- **Standardization**: mean = 0, std = 1 (use StandardScaler)
- **Normalization**: scales between 0 and 1 (use MinMaxScaler)
- StandardScaler for linear/SVM; MinMaxScaler for KNN, Neural Nets
- Scale after train-test split to avoid leakage

🔷 PART 8: Handling Duplicate Values

- Use .duplicated() to check, .drop_duplicates() to remove
- Check shape before and after
- Use subset=[...] to remove partial duplicates

🔷 PART 9: Changing Data Types

- Use .astype() to convert columns (str → int, object → float)
- Replace values like "3+" → "3" before converting
- Check with .info() and .value_counts()

🔷 PART 10: FunctionTransformer & Outlier Removal

- Use IQR to remove right-skewed outliers
- Use np.log1p(x) inside FunctionTransformer to apply log transform
- Helps normalize skewed data like income
- Visualize before and after transformation

🔷 PART 11: Feature Selection – Forward & Backward Elimination

- **Forward**: Start with none, keep adding best
- **Backward**: Start with all, drop least useful (based on p-value > 0.05)
- Use statsmodels.OLS and AIC/p-values for evaluation
- Helps reduce overfitting and model complexity

🔷 PART 12: Feature Selection using SFS

- Use SequentialFeatureSelector from mlxtend
- Works with any sklearn model (Logistic, SVM, etc.)
- Supports forward, backward, and floating selection
- Cross-validated, scoring-based automated feature selection

🔷 PART 13: Train-Test Split & Feature Scaling

- Use train_test_split() to split into training/testing sets
- Always **fit scalers on training data**, then transform both sets
- Use StandardScaler for standardization
- Essential for fair model evaluation

🔚 Final Tips for Effective Preprocessing

✅ **Always check .info() and .describe()** after each step
✅ Use **visualizations** (heatmap, boxplot, hist) to inspect effects
✅ Create **custom pipelines** for consistent preprocessing

- ✅ Document each step for **reproducibility**
- ✅ Use **train-test split before scaling or imputation**