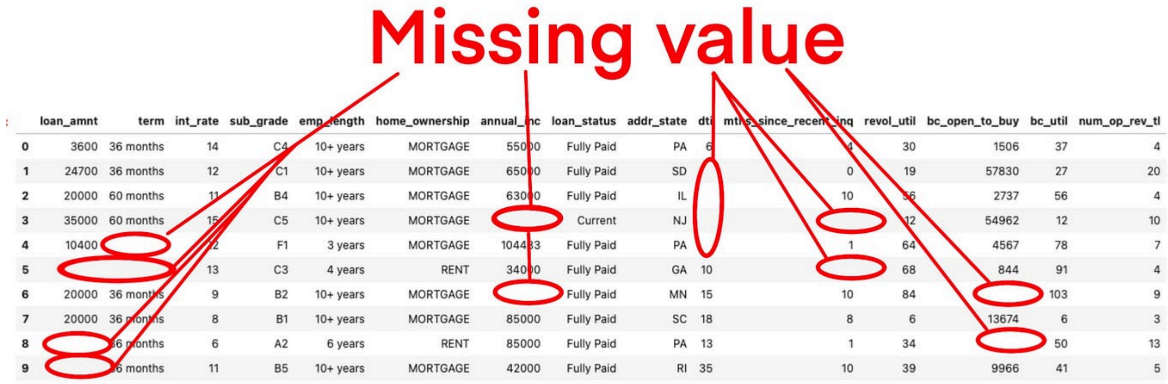
Data Cleaning in Machine Learning

Source Code:- Github

Finding Missing Values

🔍 What Are Missing Values?

Missing values are entries in your dataset where data is not available (e.g., NaN or None). Detecting and handling them is a crucial preprocessing step in any machine learning workflow.



	loan_amnt	term	int_rate	sub_grade	emp_length	home_ownership	annual_inc	loan_status	addr_state	dti	mths_since_recent_inq	revol_util	bc_open_to_buy	bc_util	num_op_rev_tl
0	3600	36 months	14	C4	10+ years	MORTGAGE	55000	Fully Paid	PA	6	4	30	1506	37	4
1	24700	36 months	12	C1	10+ years	MORTGAGE	65000	Fully Paid	SD		0	19	57830	27	20
2	20000	60 months	11	B4	10+ years	MORTGAGE	63000	Fully Paid	IL		10	56	2737	56	4
3	35000	60 months	15	C5	10+ years	MORTGAGE		Current	NJ			12	54962	12	10
4	10400		12	F1	3 years	MORTGAGE	1044[illegible]3	Fully Paid	PA		1	64	4567	78	7
5			13	C3	4 years	RENT	34000	Fully Paid	GA	10		68	844	91	4
6	20000	36 months	9	B2	10+ years	MORTGAGE		Fully Paid	MN	15	10	84		103	9
7	20000	36 months	8	B1	10+ years	MORTGAGE	85000	Fully Paid	SC	18	8	6	13674	6	3
8		36 months	6	A2	6 years	RENT	85000	Fully Paid	PA	13	1	34		50	13
9		36 months	11	B5	10+ years	MORTGAGE	42000	Fully Paid	RI	35	10	39	9966	41	5

🛠️ Common Methods to Detect Missing Values

1. dataset.isnull()

- **Description**: Returns a DataFrame of the same shape as dataset, showing True for missing (null) entries and False otherwise.

Example:

```
dataset.isnull()
```

2. dataset.isnull().sum()

- **Description**: Returns the total count of missing values **per column**.
- **Use Case**: Helps identify which columns have missing values and how many.

Example:

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

---

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

- **Description**: Returns the **total number** of missing values in the entire dataset.
- **Use Case**: Provides a quick overview of how severe the missing value problem is.

Example:

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

---

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

- **Description**: Calculates the **percentage of missing values per column**.
- **Use Case**: Helps decide whether to drop or impute columns based on the missing data ratio.

Example:

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

📊 Visualizing Missing Data

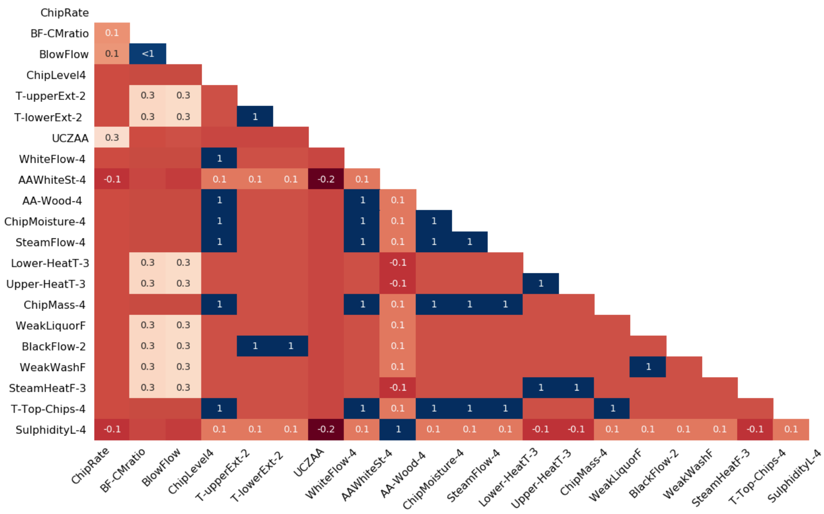
5. sns.heatmap(dataset.isnull()) + plt.show()

- **Description**: Shows a heatmap where missing values are visualized.
- **Use Case**: Easy to see patterns (e.g., if a whole row or column is missing values).

Code Example:

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

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



Dropping a Specific Column with Missing Values

✅ Code:

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

🔍 Explanation:

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

- **Purpose**: Permanently deletes the column named **"Credit_History"** from the dataset.
- **Reason**: This is often done when:

- The column has too many missing values.
- It is not useful or relevant for the analysis.
- You prefer to remove rather than impute.

✅ inplace=True means the changes will directly modify the dataset without needing reassignment.

---

2. dataset.isnull().sum()

- **Purpose**: Re-checks the dataset to see how many missing values remain **after dropping the column**.
- **Output**: A series showing missing value counts for each remaining column.

📝 Best Practice

Before dropping:

```
print(dataset["Credit_History"].isnull().sum())
```

After dropping:

```
print(dataset.columns)
print(dataset.isnull().sum())
```

dataset.dropna()

- **Description**: Removes rows with **any missing values**.
- **Example**:

```
cleaned_dataset = dataset.dropna()
```

	A	B	C	D
1	No.	Name	Date	salary
2	00001	ana varela	6/1/2016	20930
3	00002	Patricia King	6/1/2016	5410
4	00003	Charles Monaghan	6/1/2016	11350
5	00004		6/1/2016	
6	00005	John Botts	6/1/2016	25390
7	00008	Matthew Martin	6/1/2016	29520
8	00009	JP VAN BEUZEKOM	6/1/2016	
9	00010	Christian Faust	6/1/2016	23060
10	00011	Ricky Kwon	6/1/2016	20060
11	00012	Alfonso Gonzalez Pedregal	6/1/2016	28070
12	00013		6/1/2016	20710
13	00014	Simone Williams	6/1/2016	25020
14	00015	Michael Naidu	6/1/2016	12790
15	00016	Bill Waits	6/1/2016	30620
16	00017	Steffen Helmschrott	6/1/2016	
17	00018	Robert Lanza	6/1/2016	27240
18	00019	Mauro Claudio Coelho	6/1/2016	17560
19	00020	Wiebe Geldenhuys	6/1/2016	600
20	00021		6/1/2016	16280

	A	B	C	D
1	No.	Name	Date	salary
2	00001	ana varela	6/1/2016	20930
3	00002	Patricia King	6/1/2016	5410
4	00003	Charles Monaghan	6/1/2016	11350
5	00005	John Botts	6/1/2016	25390
6	00008	Matthew Martin	6/1/2016	29520
7	00010	Christian Faust	6/1/2016	23060
8	00011	Ricky Kwon	6/1/2016	20060
9	00012	Alfonso Gonzalez Pedregal	6/1/2016	28070
10	00014	Simone Williams	6/1/2016	25020
11	00015	Michael Naidu	6/1/2016	12790
12	00016	Bill Waits	6/1/2016	30620
13	00018	Robert Lanza	6/1/2016	27240
14	00019	Mauro Claudio Coelho	6/1/2016	17560
15	00020	Wiebe Geldenhuys	6/1/2016	600

Filling Missing Value

✅ Clean & Proper Way to Fill Missing Values in All Categorical Columns Using Mode:

```
# Fill missing values in all categorical columns with their mode
for col in dataset.select_dtypes(include="object").columns:
    mode_value = dataset[col].mode()[0]
    dataset[col].fillna(mode_value, inplace=True)
```

🔍 Explanation:

- `dataset.select_dtypes(include="object")`: Selects all categorical (non-numeric) columns.
- `.columns`: Gets the column names.
- `mode()[0]`: Returns the most frequent value in that column.
- `fillna(..., inplace=True)`: Fills the missing values directly in the original dataset.

📘 Optional: Add a Print to Confirm What Was Filled

If you want to **see what value was used** to fill each column:

```
for col in dataset.select_dtypes(include="object").columns:
    mode_value = dataset[col].mode()[0]
    dataset[col].fillna(mode_value, inplace=True)
    print(f"Filled missing values in '{col}' with: {mode_value}")
```

🧪 Handling Missing Values Using Scikit-Learn

📦 Import:

```
from sklearn.impute import SimpleImputer
```

🧠 What is SimpleImputer?

SimpleImputer is a part of Scikit-learn's preprocessing module.
It is used to **automatically handle missing values** in numeric and categorical columns using strategies like:

- "mean" (default)
- "median"
- "most_frequent" (mode)
- "constant" (custom value)

✨ Use Case 1: Filling Numeric Columns with Mean

```
import pandas as pd
from sklearn.impute import SimpleImputer

# Create imputer object
imputer = SimpleImputer(strategy='mean')

# Select numeric columns only
numeric_cols = dataset.select_dtypes(include=['int64', 'float64']).columns

# Apply the imputer
dataset[numeric_cols] = imputer.fit_transform(dataset[numeric_cols])
```

✨ Use Case 2: Filling Categorical Columns with Most Frequent (Mode)

```
# Create imputer object for categorical data
cat_imputer = SimpleImputer(strategy='most_frequent')

# Select categorical columns
cat_cols = dataset.select_dtypes(include=['object']).columns

# Apply imputer
dataset[cat_cols] = cat_imputer.fit_transform(dataset[cat_cols])
```

✨ Use Case 3: Filling with a Constant Value

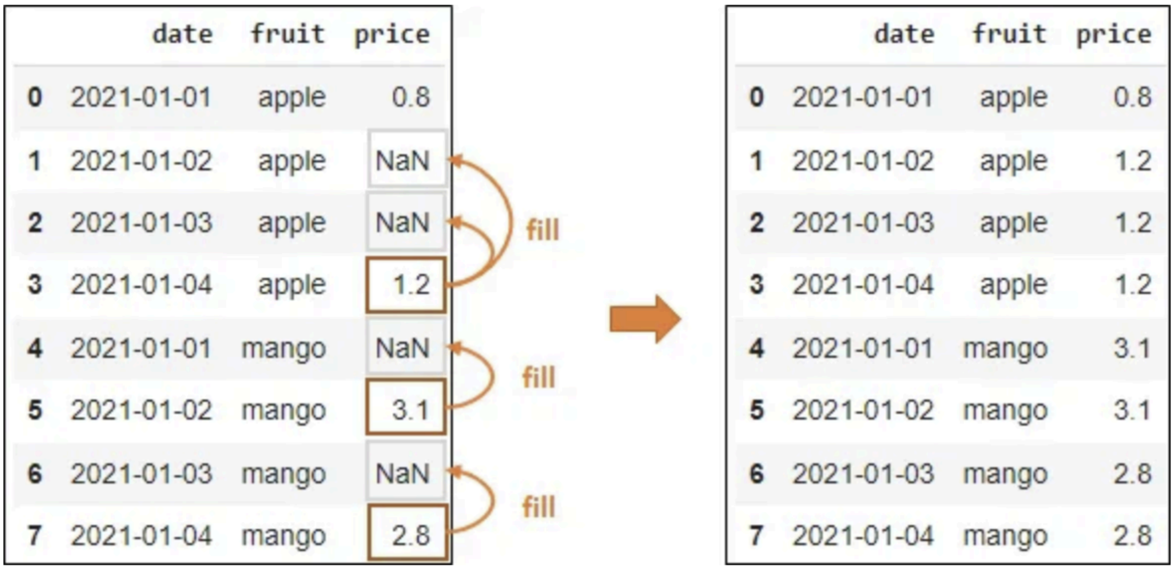
```
# Fill missing with a constant like 'Unknown' or 0
const_imputer = SimpleImputer(strategy='constant', fill_value='Unknown')

dataset[cat_cols] = const_imputer.fit_transform(dataset[cat_cols])
```

🔁 Summary Table

Strategy	Description	Best For
'mean'	Replaces with column mean	Numerical data
'median'	Replaces with column median	Skewed numeric data
'most_frequent'	Replaces with mode (most common)	Categorical data
'constant'	Replaces with a custom constant value	Missing flags, etc.

🧪 Example Dataset Workflow

```
from sklearn.impute import SimpleImputer

num_imputer = SimpleImputer(strategy="mean")
cat_imputer = SimpleImputer(strategy="most_frequent")

dataset[numeric_cols] = num_imputer.fit_transform(dataset[numeric_cols])
dataset[cat_cols] = cat_imputer.fit_transform(dataset[cat_cols])
```



	date	fruit	price
0	2021-01-01	apple	0.8
1	2021-01-02	apple	NaN
2	2021-01-03	apple	NaN
3	2021-01-04	apple	1.2
4	2021-01-01	mango	NaN
5	2021-01-02	mango	3.1
6	2021-01-03	mango	NaN
7	2021-01-04	mango	2.8

	date	fruit	price
0	2021-01-01	apple	0.8
1	2021-01-02	apple	1.2
2	2021-01-03	apple	1.2
3	2021-01-04	apple	1.2
4	2021-01-01	mango	3.1
5	2021-01-02	mango	3.1
6	2021-01-03	mango	2.8
7	2021-01-04	mango	2.8

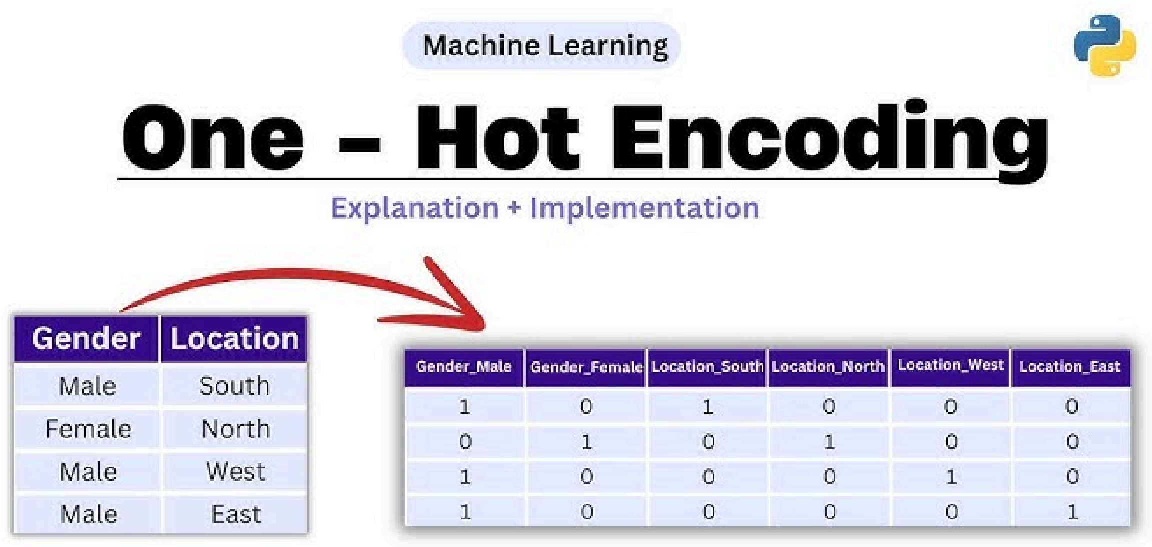
Full Source Code :- [Github]

🧠 One-Hot Encoding in Machine Learning

🔍 What is One-Hot Encoding?

One-Hot Encoding is a technique used to convert **categorical variables** into a form that can be provided to ML algorithms to do a better job in prediction.

It creates **binary (0 or 1)** columns for each unique category in the feature.



🎯 Why Use One-Hot Encoding?

Many ML models (like Linear Regression, Logistic Regression, etc.) **cannot handle categorical values directly**.
They require all features to be **numerical**.

📘 Example

Original Categorical Data:

Country
India

USA

USA

Canada

After One-Hot Encoding:

Country_Canada	Country_India	Country_USA
0	1	0
0	0	1
0	0	1
1	0	0

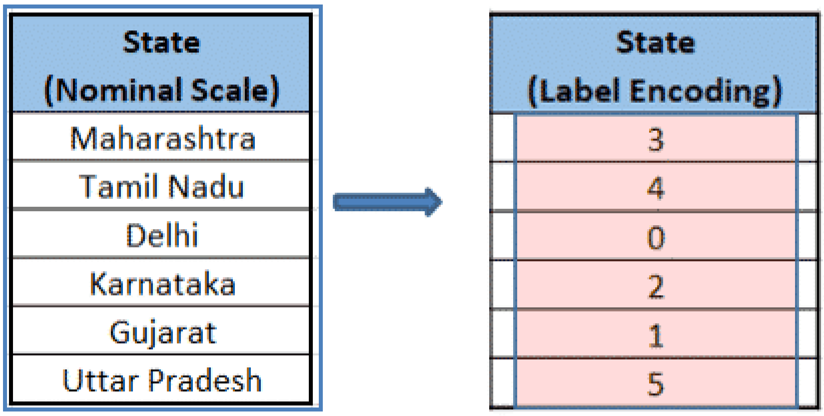
🔁 Using Scikit-Learn: OneHotEncoder

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

# Sample data
data = pd.DataFrame({'Gender': ['Male', 'Female', 'Female', 'Male']})

# Create encoder object
encoder = OneHotEncoder(sparse=False, drop=None)

# Fit and transform
encoded = encoder.fit_transform(data[['Gender']])

# Convert to DataFrame
encoded_df = pd.DataFrame(encoded, columns=encoder.get_feature_names_out(['Gender']))
print(encoded_df)
```

🔢 Label Encoding in Machine Learning

🔍 What is Label Encoding?

Label Encoding is the process of converting **categorical (text) values into numeric labels**.
Each category is assigned an integer value starting from 0.



State (Nominal Scale)	State (Label Encoding)
Maharashtra	3
Tamil Nadu	4
Delhi	0
Karnataka	2
Gujarat	1
Uttar Pradesh	5

📘 Example

Original Categorical Data:

Gender
Male
Female
Female
Male

After Label Encoding:

Gender	Gender_Encoded
Male	1
Female	0
Female	0

Male	1

🛠️ Using Scikit-Learn: LabelEncoder

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

# Sample data
data = pd.DataFrame({'Gender': ['Male', 'Female', 'Female', 'Male']})

# Create encoder object
le = LabelEncoder()

# Fit and transform
data['Gender_Encoded'] = le.fit_transform(data['Gender'])
```

🔁 How It Works

- le.fit(data['Gender']): Learns the mapping from categories to integers.
- le.transform(...): Converts values to numerical labels.

You can check the mapping:

```
print(le.classes_)  # Output: ['Female' 'Male']
```

✅ When to Use Label Encoding

Use Case	Use Label Encoding?
Ordinal Data (with order)	✅ Yes
Nominal Data (no order)	❌ No (Use One-Hot)
Tree-based models (e.g. DecisionTree, RandomForest)	✅ Yes

⚠️ Warning:

Label Encoding **introduces an ordinal relationship** (e.g., 0 < 1 < 2), which can be **misleading for nominal data** like "Red", "Green", "Blue".

> ❌ Don't use for linear models unless the data has a natural order.

🔄 Decode Labels (Optional)

You can convert encoded values back to original labels:

```
data['Original'] = le.inverse_transform(data['Gender_Encoded'])
```

🧪 Full Example:

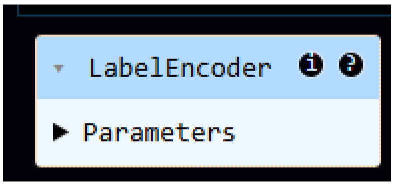
```
import pandas as pd
from sklearn.preprocessing import LabelEncoder

data = pd.DataFrame({'City': ['Delhi', 'Mumbai', 'Kolkata', 'Delhi']})

le = LabelEncoder()
data['City_Label'] = le.fit_transform(data['City'])

print(data)
```

Output:

```
     City  City_Label
0    Delhi           0
1   Mumbai           2
2  Kolkata           1
3    Delhi           0
```

🔁 Alternative: Use OrdinalEncoder for multi-column categorical data

```
from sklearn.preprocessing import OrdinalEncoder

encoder = OrdinalEncoder()
data[['City']] = encoder.fit_transform(data[['City']])
```

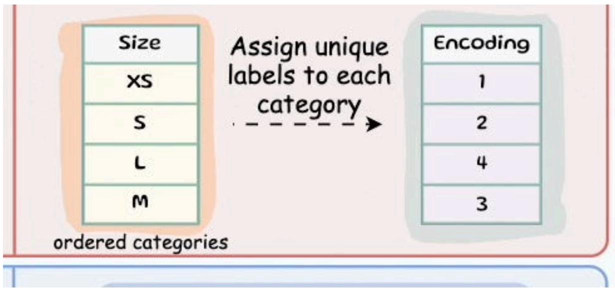


🧮 Ordinal Encoding in Machine Learning

🔍 What is Ordinal Encoding?

Ordinal Encoding converts **categorical features** into integer values **based on the rank/order** of the categories.

Unlike One-Hot or Label Encoding, **Ordinal Encoding is appropriate only when the categories have a meaningful order** (e.g., Low < Medium < High).



🧠 When to Use Ordinal Encoding?

Use Case	Ordinal Encoding?
Categorical with **order**	✅ Yes
Categorical with **no order**	❌ No (use One-Hot)
Examples: Education Level, Rank, Rating	✅ Yes

📘 Example:

Original Data:

Education
High
Medium
Low
Medium

After Ordinal Encoding:

Education	Encoded
High	2
Medium	1
Low	0
Medium	1

🛠️ Using Scikit-Learn: OrdinalEncoder

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

# Sample data
data = pd.DataFrame({'Education': ['High', 'Medium', 'Low', 'Medium']})

# Define custom order
categories = [['Low', 'Medium', 'High']]  # Must be nested list per column

# Create encoder
encoder = OrdinalEncoder(categories=categories)

# Transform
data['Education_Encoded'] = encoder.fit_transform(data[['Education']])

print(data)
```

```
  Education  Education_Encoded
0      High                2.0
1    Medium                1.0
2       Low                0.0
3    Medium                1.0
```

⚠️ Important Notes:

- **Always define the order explicitly** using categories=[['Low', 'Medium', 'High']]
- If you **don't define order**, it will default to alphabetical: High=0, Low=1, etc. — which can be **incorrect**
- Returns **float** by default; you can cast to int if needed

```
data['Education_Encoded'] = data['Education_Encoded'].astype(int)
```

🔁 Decode Back (if needed)

```
decoded = encoder.inverse_transform(data[['Education_Encoded']])
print(decoded)
```

🧪 Full Example:

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

df = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Medium']})

# Specify order
categories = [['Small', 'Medium', 'Large']]

encoder = OrdinalEncoder(categories=categories)
df['Size_Encoded'] = encoder.fit_transform(df[['Size']])

print(df)
```

🧾 Summary: Comparison with Other Encodings

Encoding Type	Preserves Order	Increases Dimensionality	Suitable For

Label Encoding	❌ No	❌ No	Nominal (Tree models)
One-Hot Encoding	❌ No	✅ Yes	Nominal
Ordinal Encoding	✅ Yes	❌ No	Ordinal

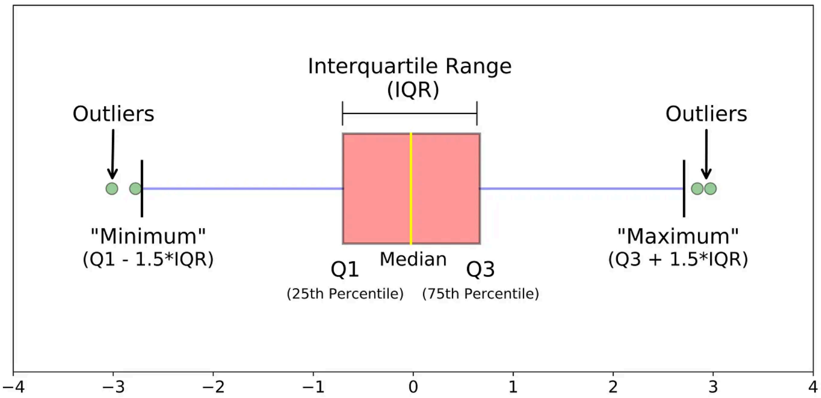
Full Source Code Link :- Github

📌 Detecting and Removing Outliers in Python

📘 What are Outliers?

Outliers are data points that **differ significantly** from other observations in a dataset. They can:

- Skew your model performance
- Mislead interpretation
- Affect statistics like mean and standard deviation



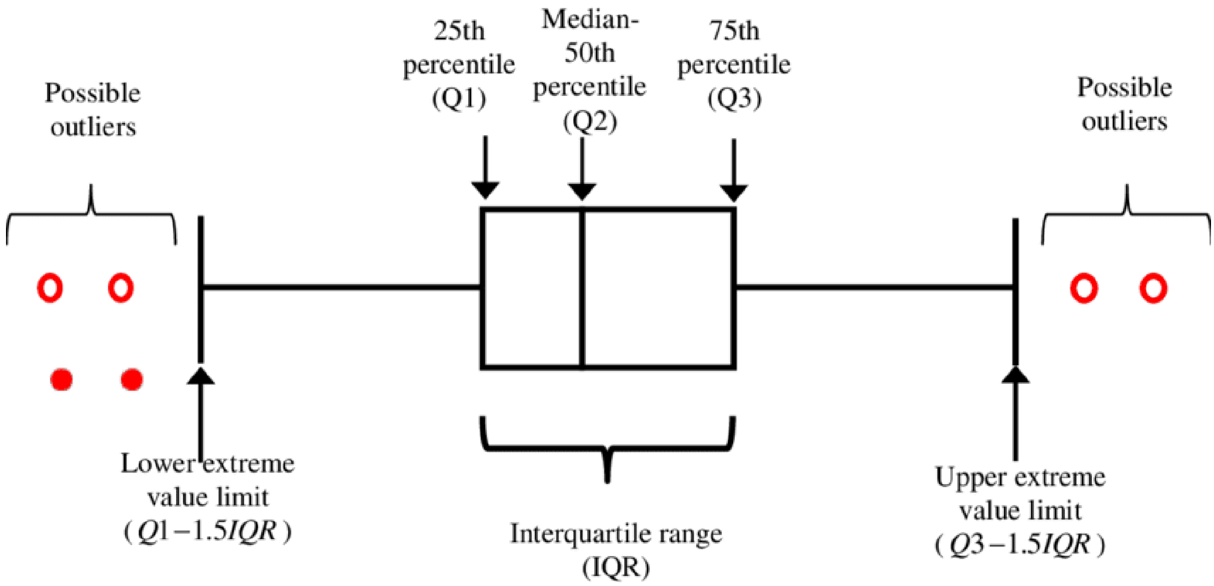
📂 Sample Dataset

```
import pandas as pd

data = {
    'Name': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J'],
    'Salary': [25000, 27000, 30000, 28000, 26000, 29000, 27500, 29500, 50000, 1000000]
}

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

	Name	Salary
0	A	25000
1	B	27000
2	C	30000
3	D	28000
4	E	26000
5	F	29000
6	G	27500
7	H	29500
8	I	50000
9	J	1000000

🔍 Step 1: Visual Detection Using Boxplot

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

sns.boxplot(x=df['Salary'])
plt.title('Boxplot of Salary')
plt.show()
```



Points far outside the boxplot whiskers are considered outliers.

🧮 Step 2: Detect Outliers Using IQR Method

```
Q1 = df['Salary'].quantile(0.25)
Q3 = df['Salary'].quantile(0.75)
IQR = Q3 - Q1

lower_limit = Q1 - 1.5 * IQR
upper_limit = Q3 + 1.5 * IQR

print("Lower Limit:", lower_limit)
print("Upper Limit:", upper_limit)

# Detecting Outliers
outliers_iqr = df[(df['Salary'] < lower_limit) | (df['Salary'] > upper_limit)]
print("Outliers Detected (IQR):\n", outliers_iqr)
```

```
Lower Limit: 23000.0
Upper Limit: 34000.0
Outliers Detected (IQR):
   Name   Salary
8    I    50000
9    J  1000000
```

✅ Step 3: Remove Outliers (IQR)

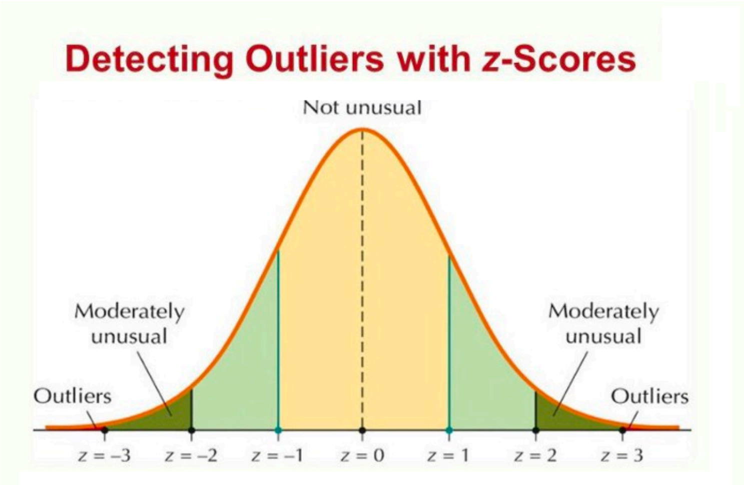
```
df_iqr_cleaned = df[(df['Salary'] >= lower_limit) & (df['Salary'] <= upper_limit)]
print("Cleaned Dataset (IQR):\n", df_iqr_cleaned)
```

```
Cleaned Dataset (IQR):
  Name  Salary
0    A   25000
1    B   27000
2    C   30000
3    D   28000
4    E   26000
5    F   29000
6    G   27500
7    H   29500
```

🧠 Step 4: Detect Outliers Using Z-Score

```
from scipy.stats import zscore

df['z_score'] = zscore(df['Salary'])

# Threshold typically used: ±3
df_z_cleaned = df[(df['z_score'] > -3) & (df['z_score'] < 3)]

# Drop z_score column if not needed
df_z_cleaned.drop(columns=['z_score'], inplace=True)

print("Cleaned Dataset (Z-Score):\n", df_z_cleaned)
```

```
Cleaned Dataset (Z-Score):
  Name   Salary
0    A    25000
1    B    27000
2    C    30000
3    D    28000
4    E    26000
5    F    29000
6    G    27500
7    H    29500
8    I    50000
9    J  1000000
```



🧾 Summary Table

Method	Good For	Threshold	Strength
Boxplot	Visualization	--	Easy to interpret
IQR	Skewed numeric data	1.5 * IQR	No assumption of normality
Z-Score	Normally distributed	±3	Best for Gaussian data

⚠️ When NOT to Remove Outliers:

- When they are **valid** observations (e.g., CEO salary)
- When you're building **robust models** (e.g., tree-based models)
- When the outliers indicate **important phenomena** (e.g., fraud detection)

Full Code:- Github

⚖️ Feature Scaling in Machine Learning

📌 Why Scale Features?

Machine learning algorithms (especially distance-based ones like KNN, K-Means, SVM, and Gradient Descent–based models) are **sensitive to the scale of features**.

✅ Feature Scaling ensures:

- All features contribute equally to the model
- Faster convergence in gradient-based models
- Better performance and accuracy

🧭 Types of Feature Scaling

1. Normalization (Min-Max Scaling)

- Scales data between **0 and 1**
- Formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Good when data **doesn't follow a normal distribution**

📘 **Code (Using MinMaxScaler):**

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

# Sample Data
data = pd.DataFrame({'Salary': [20000, 25000, 40000, 50000, 100000]})
```

```
scaler = MinMaxScaler()
data['Salary_Normalized'] = scaler.fit_transform(data[['Salary']])

print(data)
```

---

2. Standardization (Z-Score Scaling)

- Scales data to have **mean = 0** and **standard deviation = 1**
- Formula:

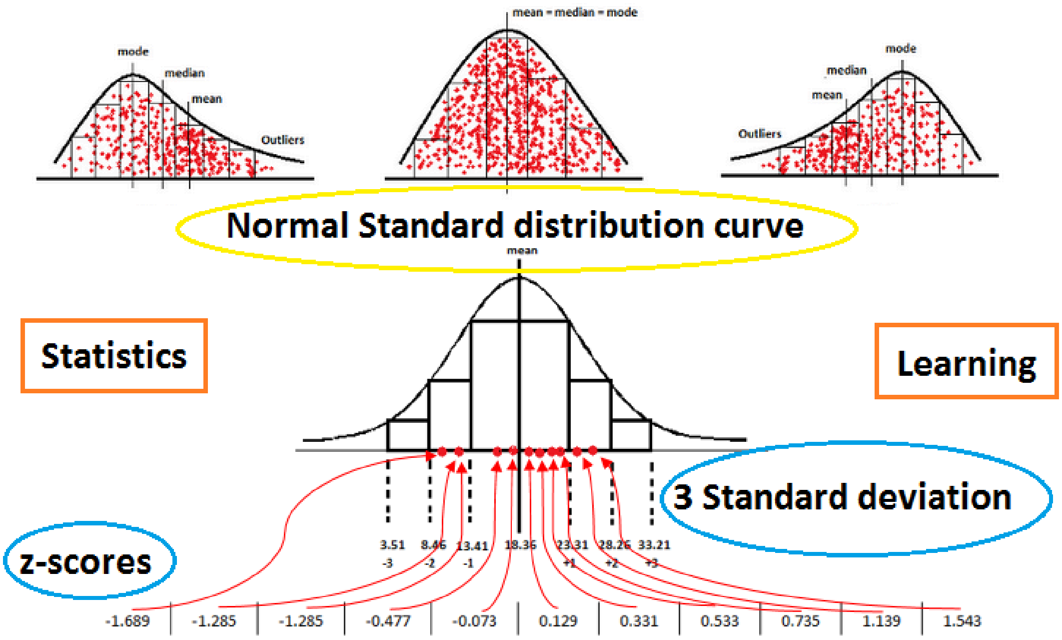
Score

Mean

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

SD

- Works well with **normally distributed** data



📘 **Code (Using StandardScaler):**

```python
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
data['Salary_Standardized'] = scaler.fit_transform(data[['Salary']])

print(data)
```

📊 Comparison Table

Technique	Scales To	Preserves Outliers?	Use When
Normalization	0 to 1	❌ No	When features are not Gaussian
Standardization	Mean = 0, SD = 1	✅ Yes	When data is normally distributed
Robust Scaling	Median-based	✅ Yes	When outliers are present

---

3. RobustScaler (Bonus)

- Scales using **median and IQR**
- Good for **datasets with outliers**

```python
from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()
data['Salary_Robust'] = scaler.fit_transform(data[['Salary']])
```

🧪 Full Example: All Scalers

```
from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler
import pandas as pd

df = pd.DataFrame({'Age': [20, 30, 40, 50, 60, 100]})

df['MinMax'] = MinMaxScaler().fit_transform(df[['Age']])
df['Standard'] = StandardScaler().fit_transform(df[['Age']])
df['Robust'] = RobustScaler().fit_transform(df[['Age']])

print(df)
```

⚠️ Important Notes

Always **fit scalers only on training data**, then transform both train and test sets.

```
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- **Don't scale categorical variables** directly — encode them first (One-Hot, Label).

Source Code:- Github

🔁 Handling Duplicate Values in a Dataset

🧠 Why Remove Duplicates?

Duplicate rows can:

- Mislead data analysis and model performance
- Inflate the importance of certain patterns
- Affect accuracy, especially in classification/regression tasks

📊 Step-by-Step Guide

🔍 1. Check for Duplicate Rows

```
# Returns True/False for each row if it is duplicated
dataset.duplicated()
```

✅ See the count of duplicates:

```
dataset.duplicated().sum()
```

📌 This tells you how many **duplicate rows** exist (excluding the first occurrence).

🔎 2. View Duplicate Rows

```
# Display only duplicate rows
duplicates = dataset[dataset.duplicated()]
print(duplicates)
```

❌ 3. Remove Duplicate Rows

```
# Drop all duplicate rows and keep the first occurrence
dataset.drop_duplicates(inplace=True)
```

⚙️ Optional Parameters:

Parameter	Description
keep='first'	(default) keeps the first occurrence
keep='last'	keeps the last occurrence
keep=False	removes **all** duplicates (no exceptions)

```
dataset.drop_duplicates(keep=False, inplace=True)
```

🧪 4. Remove Duplicates Based on Specific Columns

```
# Drop duplicates considering only selected columns
dataset.drop_duplicates(subset=['Name', 'Email'], inplace=True)
```

🔁 5. Reset Index (Optional)

After removing duplicates, you might want to reset the index:

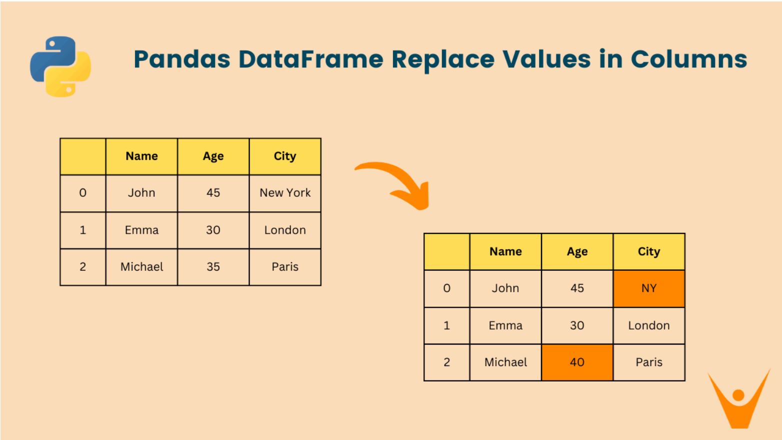
```
dataset.reset_index(drop=True, inplace=True)
```

[Source Code:- Github]

🔄 Replacing Values & Changing Data Types in Pandas

📌 1. Replacing Values in a Dataset

🧠 Why Replace Values?

- To clean dirty data (e.g., “Male ” → “Male”)
- To encode categorical strings as numbers (e.g., “Yes” → 1)
- To unify formats (e.g., “N/A” → np.nan)
- 

✅ Replace Single Value

```
dataset['Gender'].replace('M', 'Male', inplace=True)
```

✅ Replace Multiple Values

```
dataset['Gender'].replace({'M': 'Male', 'F': 'Female'}, inplace=True)
```

✅ Replace Missing/Custom Strings with NaN

```
import numpy as np

dataset.replace(['N/A', 'na', 'unknown'], np.nan, inplace=True)
```

✅ Replace in Entire DataFrame

```
dataset.replace({'Yes': 1, 'No': 0}, inplace=True)
```

🧪 Example:

```
data = {'Gender': ['M', 'F', 'F', 'M', 'M'],
        'Status': ['Yes', 'No', 'Yes', 'No', 'Yes']}

import pandas as pd
df = pd.DataFrame(data)

df.replace({'M': 'Male', 'F': 'Female', 'Yes': 1, 'No': 0}, inplace=True)
```

🔀 2. Changing Data Types

🔎 Check Data Types

```
print(dataset.dtypes)
```

🔁 Convert Column to Integer

```
dataset['Age'] = dataset['Age'].astype(int)
```

🔁 Convert to Float

```
dataset['Salary'] = dataset['Salary'].astype(float)
```

🔁 Convert to String

```
dataset['ID'] = dataset['ID'].astype(str)
```

🔁 Convert to Datetime

```
dataset['Join_Date'] = pd.to_datetime(dataset['Join_Date'])
```

⚠️ Handle Conversion Errors

```
# If some values can't be converted, force errors to NaT (missing datetime)
dataset['Join_Date'] = pd.to_datetime(dataset['Join_Date'], errors='coerce')
```

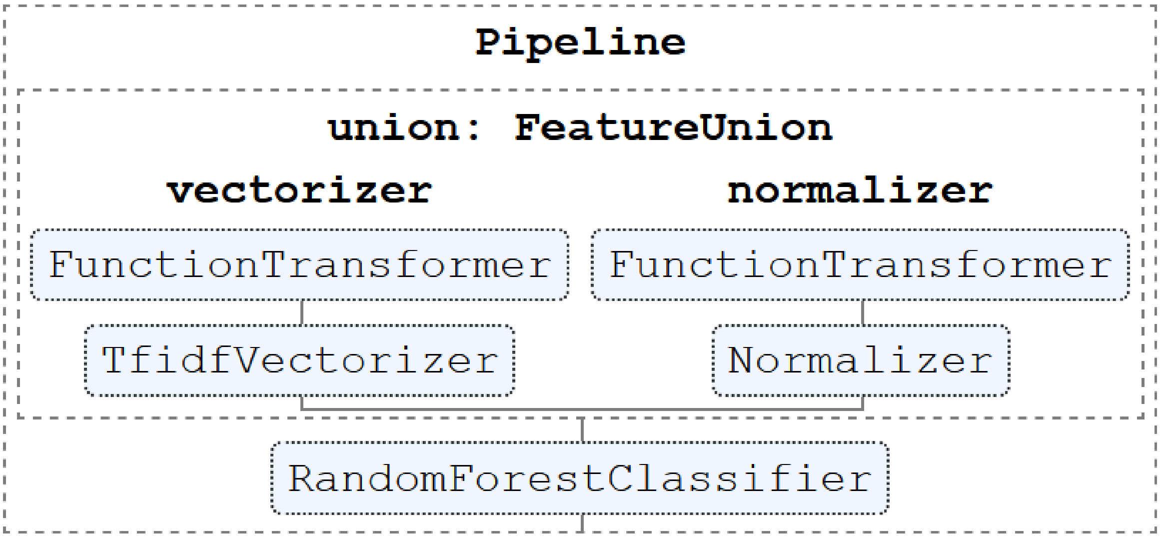
[Source Code:- Github]()

🔧 FunctionTransformer in Scikit-Learn

📌 What is FunctionTransformer?

FunctionTransformer is a utility in **Scikit-learn** that allows you to wrap a custom or existing function (like log, sqrt, etc.) and apply it consistently in preprocessing pipelines.

> It is especially useful in pipelines where transformations need to be **repeatable**, **fitted**, and **applied consistently** on train/test splits.



⚙️ Syntax

```
from sklearn.preprocessing import FunctionTransformer

transformer = FunctionTransformer(func, inverse_func=None, validate=True)
```

Parameter	Description
func	Function to apply (e.g., np.log1p)
inverse_func	Function to inverse-transform (optional)

validate	Validates input shape (ensure 2D)

🎯 Why Use It?

- Compatible with **Pipeline** and **ColumnTransformer**
- Useful for **log**, **square root**, **power** transformations
- Allows **custom preprocessing** steps

🧪 Example 1: Apply log1p to a Column

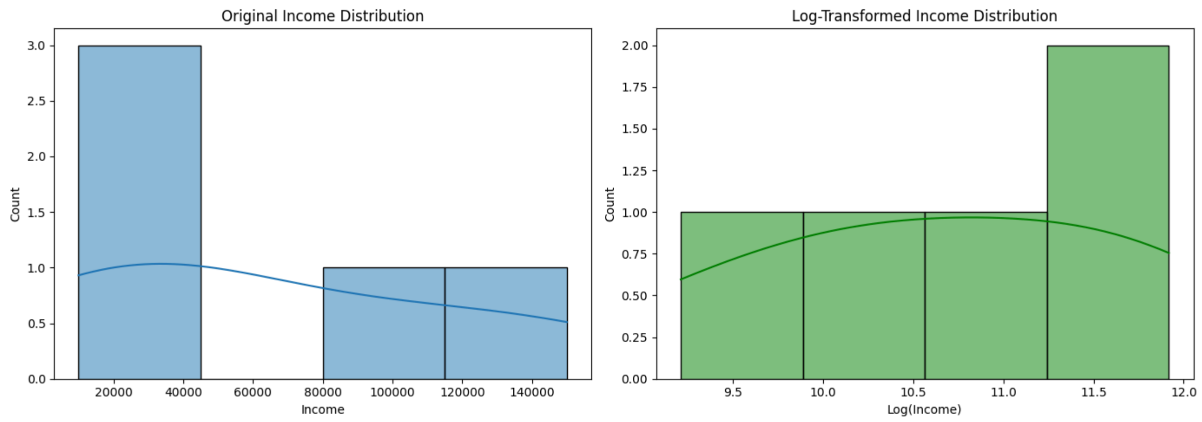
```python
import numpy as np
import pandas as pd
from sklearn.preprocessing import FunctionTransformer

data = pd.DataFrame({'Income': [10000, 25000, 40000, 100000, 150000]})

# Apply log transformation: log(1 + x)
log_transformer = FunctionTransformer(np.log1p, validate=True)

data['Income_Log'] = log_transformer.fit_transform(data[['Income']])
print(data)
```

```
   Income  Income_Log
0   10000    9.210440
1   25000   10.126671
2   40000   10.596660
3  100000   11.512935
4  150000   11.918397
```

📈 Visualizing Before & After (Optional)

```python
import seaborn as sns
```

```python
import matplotlib.pyplot as plt

# Create subplots: 1 row, 2 columns
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot original distribution
sns.histplot(data['Income'], kde=True, ax=axes[0])
axes[0].set_title("Original Income Distribution")
axes[0].set_xlabel("Income")

# Plot log-transformed distribution
sns.histplot(data['Income_Log'], kde=True, color='green', ax=axes[1])
axes[1].set_title("Log-Transformed Income Distribution")
axes[1].set_xlabel("Log(Income)")

# Improve layout
plt.tight_layout()
plt.show()
```



🧪 Example 2: Custom Transformation (e.g., Square Root)

🧠 Use Case in a Pipeline

```python
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression

pipeline = Pipeline(steps=[
    ('log_transform', FunctionTransformer(np.log1p)),
    ('model', LinearRegression())
])
```

🔁 Inverse Transformation

You can use `inverse_func` to recover the original values:

```python
log_trans = FunctionTransformer(np.log1p, inverse_func=np.expm1)
X_log = log_trans.transform(data[['Income']])
X_original = log_trans.inverse_transform(X_log)
```

[Source Code :- Github]()