Data Preprocessing

*Data preprocessing is a vital step in machine learning that transforms raw, messy data into a clean and structured format for model training. It involves cleaning, transforming, encoding, and splitting data to improve model accuracy, prevent data leakage, and ensure compatibility with algorithms. While often confused with data cleaning, preprocessing encompasses a broader set of tasks critical to reliable machine learning pipelines. Using tools like Pandas, Scikit-learn, and Apache Spark helps streamline this process, making it scalable and effective across different project sizes and complexities.*

**What is data preprocessing in machine learning?**

Data preprocessing in machine learning refers to the steps taken to clean, organize, and transform raw data into a format that machine learning algorithms can use effectively. Real-world data is often messy because it includes missing values, inconsistent formats, outliers, and irrelevant features. Without proper preprocessing, even the most sophisticated machine learning models can struggle to find patterns or may produce misleading results.

Effective data preprocessing not only improves the accuracy and efficiency of ML models but also helps uncover deeper insights hidden within the data. It sets the foundation for any successful ML project by ensuring the input data is high quality, consistent, and relevant.

**Data preprocessing vs. data cleaning**

While data preprocessing and data cleaning are often used interchangeably, they refer to different stages in the data preparation pipeline. Data cleaning is actually a subset of the broader data preprocessing process. Understanding the differences between the two is crucial to building reliable machine learning models, as each plays a unique role in preparing [raw data for analysis](https://www.couchbase.com/blog/what-is-data-analysis/). The table below clarifies their specific purposes, tasks, and importance.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Data Cleaning** | **Data Preprocessing** |
| **Scope** | Narrow – focuses on removing data issues | Broad – includes cleaning, transforming, and preparing data for machine learning |
| **Main Goal** | Improve data quality | Make data suitable for model training and evaluation |
| **Typical Tasks** | Removing duplicates, handling missing values | Cleaning, normalization, encoding, feature engineering, and splitting |
| **Involves Transformation?** | Rarely | Frequently (e.g., scaling, encoding, aggregation) |
| **Used In** | Data wrangling, early analysis | Full machine learning pipeline – from raw data to model-ready format |
| **Tools Used** | Pandas, OpenRefine, Excel | Scikit-learn, Pandas, TensorFlow, NumPy |
| **Example** | Filling in missing values with the mean | Filling in missing values and one-hot encoding, along with standardization and train/test split |

**Why data preprocessing is important in machine learning**

Effective data preprocessing is a critical step in the machine learning pipeline. It ensures that the data fed into a model is clean, consistent, and informative, directly impacting its performance and reliability. Here are some key reasons why data preprocessing is important in machine learning:

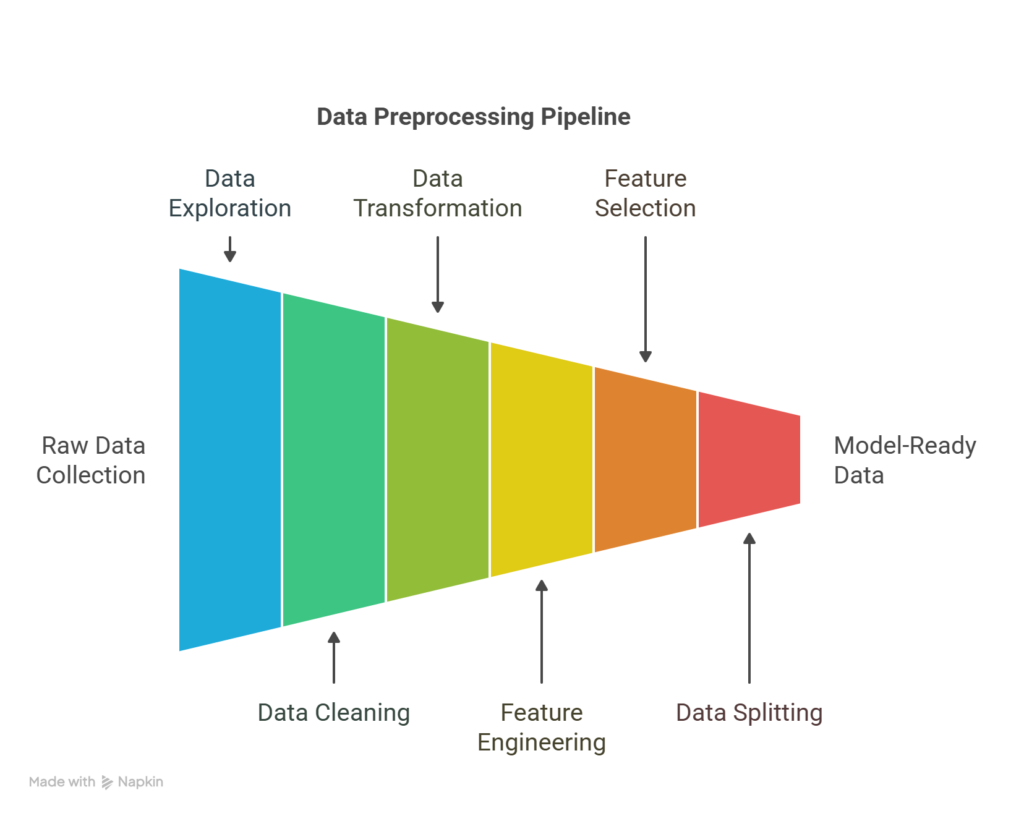
* + **Improves model accuracy:** Clean and well-structured data enables algorithms to learn patterns more effectively, leading to better predictions and outcomes.
  + **Reduces noise and inconsistencies:** Removing irrelevant or erroneous data helps prevent misleading insights and model confusion.
  + **Handles missing or incomplete data:** Preprocessing techniques such as imputation or deletion ensure that gaps in data don’t degrade model performance.
  + **Ensures data compatibility:** Many machine learning algorithms require data in specific formats; preprocessing steps like [normalization](https://www.couchbase.com/blog/normalization-vs-denormalization/) or encoding make the data compatible with these requirements.
  + **Prevents data leakage:** Proper data splitting during preprocessing (into training, validation, and test sets) helps avoid overfitting and ensures fair model evaluation.
  + **Saves time and resources:** Clean, organized data streamlines model training, reduces computational costs, and shortens development cycles.

**Data preprocessing techniques**

Data preprocessing involves various techniques designed to prepare raw data for use in machine learning models. Each technique addresses specific challenges in the dataset and contributes to cleaner, more reliable inputs. Below are some of the most commonly used data preprocessing techniques:

* + **Data cleaning:** Detects and corrects errors, removes duplicates, and handles missing values through strategies like imputation or deletion.
  + **Encoding categorical variables:** Converts non-numeric data (e.g., labels or categories) into numeric formats using one-hot encoding or label encoding.
  + **Outlier detection and removal:** Identifies data points that deviate significantly from others, which can negatively impact model performance if left unaddressed.
  + **Dimensionality reduction:** Reduces the number of input features while preserving important information, using methods like principal component analysis (PCA).
  + **Normalization and scaling (Data Transformation and scaling):** Adjusts numeric values to a common scale without distorting differences in the ranges, often essential for algorithms like KNN or gradient descent-based models.
  + **Data splitting:** Divides the dataset into training, validation, and test sets to evaluate the model effectively and prevent overfitting.

**Data preprocessing steps in machine learning**



**Steps in the data preprocessing pipeline**

Data preprocessing is a multi-step process that prepares raw data for machine learning. Each step helps ensure the dataset is accurate, consistent, and optimized for model performance. Here’s a step-by-step breakdown of the typical data preprocessing workflow:

**Data collection**

The process begins with gathering data from relevant sources such as [databases](https://www.couchbase.com/resources/concepts/types-of-databases/), [APIs](https://www.couchbase.com/blog/api-vs-sdk/), sensors, or files. The quality and relevance of collected data directly influence the success of downstream tasks.

**Data exploration**

Before making changes, it’s essential to understand the dataset through exploratory data analysis (EDA). This step involves summarizing data characteristics, visualizing distributions, detecting patterns, and identifying anomalies or inconsistencies.

**Data cleaning**

This step addresses missing values, duplicate records, inconsistent formatting, and outliers. Cleaning ensures the dataset is reliable and free of noise or errors that could interfere with model training.

**Data transformation**

At this stage, the data is formatted for model compatibility. This process includes normalizing or scaling numerical values, encoding categorical variables, and transforming skewed distributions to improve model learning.

**Feature engineering**

New features are created based on existing data to better capture underlying patterns. This process might include extracting time-based variables, combining fields, or applying domain knowledge to enrich the dataset.

**Feature selection**

Not all features contribute equally to model performance. This step involves selecting the most relevant variables and removing redundant or irrelevant ones, which helps reduce overfitting and improve efficiency.

**Data splitting**

The cleaned and engineered dataset is divided into training, validation, and test sets. Doing this ensures that the model is evaluated on unseen data and generalizes to real-world scenarios.

**Final review**

Before modeling, a final check ensures that all preprocessing steps were correctly applied. This stage involves verifying distributions, feature quality, and data splits to prevent issues like data leakage or imbalance.

**Data preprocessing example**

Suppose you’re building a model to predict whether a customer will churn from a subscription service. Imagine you have a dataset from a telecom company with the following columns:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Customer\_ID** | **Age** | **Gender** | **Monthly\_Charges** | **Contract\_Type** | **Churn** |
| 1 | 34 | Male | 70.5 | Month-to-month | Yes |
| 2 | NaN | Female | 85 | One year | No |
| 3 | 45 | Female | NaN | Month-to-month | Yes |
| 4 | 29 | Male | 65.5 | Two year | No |

Let’s walk through the preprocessing steps:

* 1. **Handling missing values**
     + Fill in the missing Age with the average age (36).
     + Fill in the missing Monthly\_Charges with the column median (73.5).
  2. **Encoding categorical variables**
     + **Gender** (Male/Female) and **Contract\_Type** (Month-to-month, One year, Two year) are categorical.
     + Apply:
       - **Label encoding** for Gender (Male = 0, Female = 1)
       - **One-hot encoding** for Contract\_Type, resulting in:
         * Contract\_Month\_to\_month, Contract\_One\_year, Contract\_Two\_year
  3. **Feature scaling**
     + Normalize Age and Monthly\_Charges to bring them to the same scale (this is especially useful for distance-based models like KNN).
  4. **Target encoding**
     + Convert Churn (Yes/No) to binary:
       - Yes = 1
       - No = 0
  5. **Cleaned and preprocessed dataset**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Age** | **Gender** | **Monthly\_Charges** | **Contract\_Month** | **Contract\_One** | **Contract\_Two** | **Churn** |
| 34 | 0 | 70.5 | 1 | 0 | 0 | 1 |
| 36 | 1 | 85 | 0 | 1 | 0 | 0 |
| 45 | 1 | 73.5 | 1 | 0 | 0 | 1 |
| 29 | 0 | 65.5 | 0 | 0 | 1 | 0 |

Now the dataset is clean, numeric, and ready for model training.

**Data preprocessing tools**

Choosing the right tools for data preprocessing can impact the effectiveness of your machine learning workflow. Below is a list of commonly used tools, along with their strengths and limitations:

Pandas (Python)

**Best suited for:**

* + Handling structured data (e.g., CSVs, Excel, SQL tables)
  + Data cleaning, filtering, and transformation
  + Quick exploratory data analysis

**Not suited for:**

* + Large-scale distributed processing
  + Complex ETL pipelines or [unstructured data (e.g., images, audio)](https://www.couchbase.com/resources/concepts/unstructured-data/)

NumPy (Python)

**Best suited for:**

* + Numerical operations and handling multidimensional arrays
  + Performance-optimized matrix computations

**Not suited for:**

* + High-level data manipulation or cleaning
  + Working directly with labeled datasets (Pandas is more appropriate)

Scikit-learn (Python)

**Best suited for:**

* + Feature scaling, encoding, and selection
  + Data splitting (train/test/validation)
  + Integration with ML models and pipelines

**Not suited for:**

* + Deep learning tasks
  + Heavy data manipulation (use with Pandas)

OpenRefine

**Best suited for:**

* + Cleaning messy, unstructured, or inconsistent data
  + Reconciling and transforming data from different sources
  + Non-programmers needing a GUI-based tool

**Not suited for:**

* + Large datasets
  + Integration into automated machine learning workflows

Apache Spark (with PySpark or Scala)

**Best suited for:**

* + Processing large-scale datasets in a distributed environment
  + Data preprocessing in big data pipelines
  + Integration with cloud platforms (AWS, Azure, GCP)

**Not suited for:**

* + Small-to-medium datasets (overhead may not be justified)
  + Fine-grained, interactive data manipulation

Dataiku

**Best suited for:**

* + End-to-end ML workflows, including preprocessing, modeling, and deployment
  + Teams with both technical and non-technical users
  + Visual programming and automation

**Not suited for:**

* + Deep customization or low-level data control
  + Lightweight personal projects or code-only workflows

TensorFlow Data Validation (TFDV)

**Best suited for:**

* + Validating data pipelines in production ML workflows
  + Detecting schema anomalies and data drift at scale
  + Use within the TensorFlow Extended (TFX) ecosystem

**Not suited for:**

* + General-purpose data cleaning
  + Use outside TensorFlow or TFX environments

The strengths and limitations of these tools ultimately depend on the size of your project, the complexity, and the technical environment. Combining tools (e.g., Pandas for cleaning and Scikit-learn for feature scaling) usually provides the best results.

**Data cleaning**

This step addresses missing values, duplicate records, inconsistent formatting, and outliers. Cleaning ensures the dataset is reliable and free of noise or errors that could interfere with model training.

1. **Handle Missing Values:**

Handling **missing values** is a crucial step in data preprocessing. In Python, several libraries offer different tools for this. Here's a detailed guide to **the most commonly used libraries** and their **use cases**, along with **when and why to use each**.

**🧰 Libraries for Handling Missing Values in Python**

**1. pandas**

**✅ Use Case:**

* Simple or exploratory data analysis (EDA)
* Direct manipulation of DataFrames

**🛠️ Common Methods:**

* df.isnull() / df.notnull() — detect missing values
* df.dropna() — remove missing values
* df.fillna(value) — fill with constants, mean, forward fill (method='ffill'), etc.
* df.ffill(inplace=False, limit=None)
* df.bfill(inplace=False, limit=None)
* df.isnull().sum()
* df.isnull().tail(50)
* df.notna()
* df.isna()
* df.where(df.notnull()
* df.replace(0, np.nan, inplace=True)
* df.replace([np.nan,0],1.0, inplace=True)
* df.fillna(1.0, inplace=True)
* df[df.isnull().any(axis=1)]
* df.T.fillna(df.median(axis=1), inplace=False).T
* df = df.mask((df < 0.4) & (df > 0),np.nan)
* df = df.mask( (df > 0.5) & (df < 1), np.nan)

**💡 When to Use:**

* Quick fixes during EDA (Exploratory Data Analysis)
* Easy data inspection
* Situations where you don’t need scikit-learn Pipelines

import pandas as pd

import numpy as np

df = pd.DataFrame({

'age': [25, np.nan, 30],

'salary': [50000, 60000, np.nan]

})

# Fill NaNs with column mean

df['age'] = df['age'].fillna(df['age'].mean())

df['salary'] = df['salary'].fillna(df['salary'].mean())

print(df)

**2. sklearn.impute (scikit-learn)**

**✅ Use Case:**

* Building ML pipelines
* Need for consistent preprocessing in training and testing data

**🛠️ Tools:**

* SimpleImputer: fills missing values using strategies like mean, median, most frequent, constant
* KNNImputer: uses K-nearest neighbors for imputation
* IterativeImputer: multivariate imputation using models
* MissingIndicator: adds binary features to indicate where values were missing

**💡 When to Use:**

* During model training and deployment
* When you want to include imputation as part of a pipeline
* When data needs more sophisticated imputation (e.g., correlated features)

**Explain SimpleImputer:**

**SimpleImputer** is a scikit-learn class which is helpful in handling the missing data in the predictive model dataset. It replaces the NaN values with a specified placeholder.   
It is implemented by the use of the **SimpleImputer()** method which takes the following arguments :

***missing\_values*** *: The missing\_values placeholder which has to be imputed. By default is NaN****strategy*** *: The data which will replace the NaN values from the dataset. The strategy argument can take the values - 'mean'(default), 'median', 'most\_frequent' and 'constant'.****fill\_value*** *: The constant value to be given to the NaN data using the constant strategy.*

**Code: Python code illustrating the use of SimpleImputer class.**

import numpy as np

# Importing the SimpleImputer class

from sklearn.impute import SimpleImputer

# Imputer object using the mean strategy and

# missing\_values type for imputation

imputer = SimpleImputer(missing\_values = np.nan,

strategy ='mean')

data = [[12, np.nan, 34], [10, 32, np.nan],

[np.nan, 11, 20]]

print("Original Data : \n", data)

# Fitting the data to the imputer object

imputer = imputer.fit(data)

# Imputing the data

data = imputer.transform(data)

print("Imputed Data : \n", data)

**Output**

Original Data :

[[12, nan, 34]

[10, 32, nan]

[nan, 11, 20]]

Imputed Data :

[[12, 21.5, 34]

[10, 32, 27]

[11, 11, 20]]

**Remember: The mean or median is taken along the column of the matrix**

Code example with different strategies –

**# 1. SimpleImputer with mean strategy**

mean\_imputer = SimpleImputer(strategy='mean', missing\_values=np.nan)

df\_mean\_imputed = df.copy()

transformed\_data = mean\_imputer.fit\_transform(df\_mean\_imputed.iloc[:, :-1])

df\_mean\_imputed.iloc[:, :-1] = transformed\_data

print("\nDataFrame after mean imputation:")

print(df\_mean\_imputed.head())

# Explain me this mean stategy how it works step by step

# 1. Calculate the mean of each feature (column) in the dataset, ignoring the missing values.

#    For example, for the 'plas' column, the mean is calculated as:

#    mean\_plas = (148 + 85 + 183 + 89 + 137 + 116 + 78 + 115 + 197 + 125 + 110 + 140 + 130 + 99 + 120 + 100 + 95 + 105 + 143 + 129) / 19 = 122.68

#    (Note: The missing value is ignored in the calculation)

# 2. Replace the missing values in each feature with the calculated mean.

#    For the 'plas' column, the missing value at index 0 is replaced with the mean value:

#    df\_mean\_imputed.loc[0, 'plas'] = mean\_plas = 122.68

# 3. Repeat the process for all features with missing values.

# iloc[:, :-1] means - select all rows and all columns except the last one (target variable 'class').

# 4. The final imputed DataFrame will have the missing values replaced with the mean of their respective columns.

**# 2. SimpleImputer with median strategy**

median\_imputer = SimpleImputer(strategy='median')

df\_median\_imputed = df.copy()

df\_median\_imputed.iloc[:, :-1] = median\_imputer.fit\_transform(df\_median\_imputed.iloc[:, :-1])

print("\nDataFrame after median imputation:")

print(df\_median\_imputed.head())

# Explain me this median stategy how it works step by step

# 1. Calculate the median of each feature (column) in the dataset, ignoring the missing values.

#    For example, for the 'plas' column, the median is calculated as:

#    median\_plas = 122.68 (the middle value when the values are sorted)

#    (Note: The missing value is ignored in the calculation)

# 2. Replace the missing values in each feature with the calculated median.

#    For the 'plas' column, the missing value at index 0 is replaced with the median value:

**# 3. SimpleImputer with most frequent strategy**

most\_frequent\_imputer = SimpleImputer(strategy='most\_frequent')

df\_most\_frequent\_imputed = df.copy()

df\_most\_frequent\_imputed.iloc[:, :-1] = most\_frequent\_imputer.fit\_transform(df\_most\_frequent\_imputed.iloc[:, :-1])

print("\nDataFrame after most frequent imputation:")

print(df\_most\_frequent\_imputed.head())

# Explain me this most frequent stategy how it works step by step

# 1. Calculate the most frequent value of each feature (column) in the dataset, ignoring the missing values.

#    For example, for the 'plas' column, the most frequent value is calculated as:

#    most\_frequent\_plas = 148 (the value that appears most often in the column)

#    (Note: The missing value is ignored in the calculation)

# 2. Replace the missing values in each feature with the calculated most frequent value.

**# 4. SimpleImputer with constant strategy**

constant\_imputer = SimpleImputer(strategy='constant', fill\_value=0)

df\_constant\_imputed = df.copy()

df\_constant\_imputed.iloc[:, :-1] = constant\_imputer.fit\_transform(df\_constant\_imputed.iloc[:, :-1])

print("\nDataFrame after constant imputation:")

print(df\_constant\_imputed.head())

# Explain me this constant stategy how it works step by step

# 1. Replace all missing values in each feature (column) with a specified constant value (in this case, 0).

#    For example, for the 'plas' column, the missing value at index 0 is replaced with 0:

#    df\_constant\_imputed.loc[0, 'plas'] = 0

# 2. Repeat the process for all features with missing values

**Difference between:**

python

CopyEdit

imputer = imputer.fit(data)

data = imputer.transform(data)

vs.

python

CopyEdit

data = imputer.fit\_transform(data)

**🔹 1. imputer.fit(data)**

This **learns** from the data.

* For example, if the strategy is 'mean', it computes the **mean of each column** (ignoring NaNs) and stores it internally.
* No transformation happens at this step — it's just learning what to do.

✅ **Used when you want to separate the fitting and transforming steps** — often useful in machine learning pipelines where you **fit on training data** and **transform both train and test data**.

**🔹 2. imputer.transform(data)**

This **applies** the learned transformation.

* Replaces the missing values in data using the statistics computed during .fit().

✅ You call this on **new data** (e.g., test data or validation set) using the same logic learned from training data.

**🔹 Combined version: fit\_transform(data)**

python

CopyEdit

data = imputer.fit\_transform(data)

* Equivalent to calling fit() followed by transform().
* Commonly used for **training data**, where you want to fit and transform in a single step.

**Imputer with add\_indicator=True to allow inverse\_transform**

# 8. SimpleImputer.inverse\_transform

# Reverses the transformation applied by the SimpleImputer

print("\nInverse transform example:")

mean\_imputer = SimpleImputer(strategy='mean', add\_indicator=True)

transformed = mean\_imputer.fit\_transform(df.iloc[:, :-1])  # fit on input columns

original\_data = mean\_imputer.inverse\_transform(transformed)

print(pd.DataFrame(original\_data, columns=df.columns[:-1]).head()) # Display the original data without missing values

# 9. SimpleImputer.get\_params

# Gets the hyperparameters of the SimpleImputer model

print("\nGet parameters of SimpleImputer:")

params = mean\_imputer.get\_params() # Get the hyperparameters of the SimpleImputer model

print(params)

# hyperparameters means the parameters that control the behavior of the SimpleImputer model.

# Hyperparameters include the strategy used for imputation (mean, median, most frequent, constant), the fill value (if applicable), and other settings that affect how the model processes the data.

# 10. SimpleImputer.set\_params

# Sets the hyperparameters of the SimpleImputer model

print("\nSet parameters of SimpleImputer:")

mean\_imputer.set\_params(strategy='median')

print(mean\_imputer)

* Explain KNNImputer: uses K-nearest neighbors for imputation

Handling missing data in data science and machine learning, is a crucial preprocessing step. The K-Nearest Neighbors (KNN) Imputer is a sophisticated technique used for imputing missing values by leveraging the relationships within the dataset. This article delves into the workings of the KNN Imputer, its implementation, and its advantages over traditional imputation methods.

**What is K-Nearest Neighbors Imputer (KNN)?**

The KNN Imputer is a multivariate imputation method that fills in missing values by considering the values of the nearest neighbors of the data point with missing values. Unlike univariate methods, which consider only one variable at a time, the KNN Imputer uses multiple variables, making it a more robust and reliable approach for estimating missing data.

**How Does K-Nearest Neighbors Imputer Work?**

The KNN Imputer operates on the principle of the [K-Nearest Neighbors algorithm](https://www.geeksforgeeks.org/k-nearest-neighbours/), which is widely used for classification and regression tasks. Here’s a step-by-step breakdown of how the KNN Imputer works:

1. **Distance Calculation**: For each missing value, the KNN Imputer calculates the distance between the data point with missing values and all other data points in the dataset. The default distance metric used is the [Euclidean distance](https://www.geeksforgeeks.org/euclidean-distance/), which is NaN-aware, meaning it can handle missing values without biasing the distance calculation.
2. **Identifying Neighbors**: The algorithm identifies the 'k' nearest neighbors to the data point with the missing value. These neighbors are the data points with the smallest distance to the point with the missing value.
3. **Imputation**: The missing value is imputed using the mean (or median) of the identified nearest neighbors. This approach ensures that the imputed value is influenced by the most similar data points, thereby maintaining the integrity of the dataset.
4. **Handling Multivariate Data**: In multivariate datasets, the KNN Imputer considers all available features, making it more effective in capturing the underlying patterns and relationships between variables.

**Example: Imputing Missing Values with KNN Imputer**

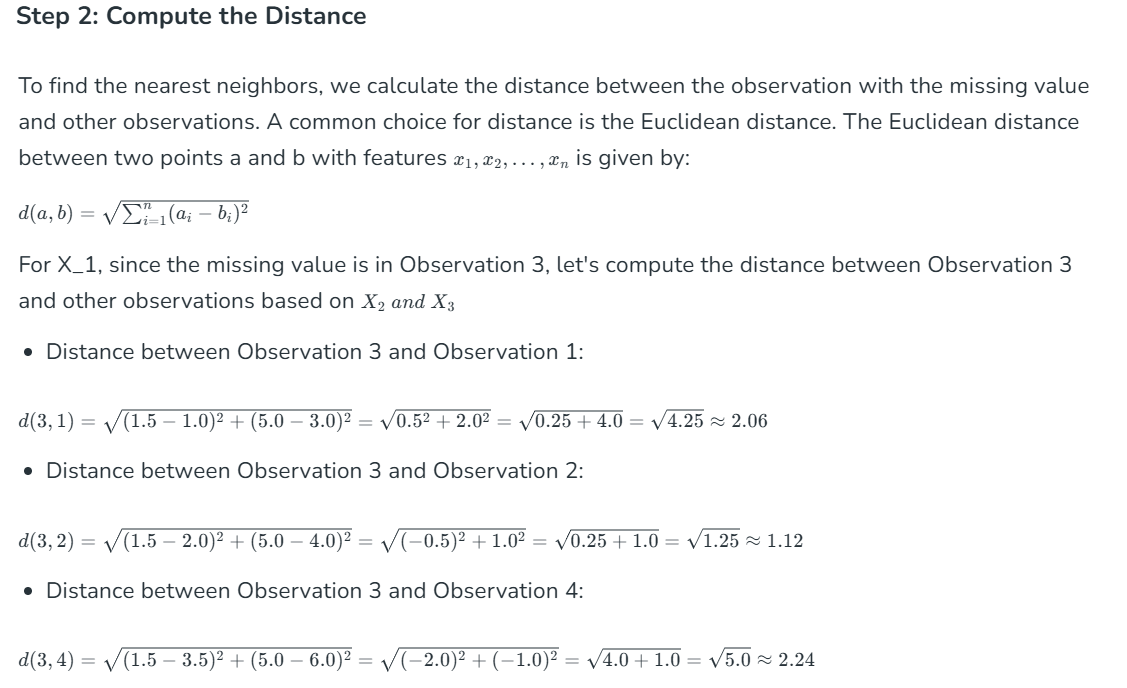
Imagine you have a dataset with three features (X1,X2,X3)(*X*1​,*X*2​,*X*3​) and five observations:

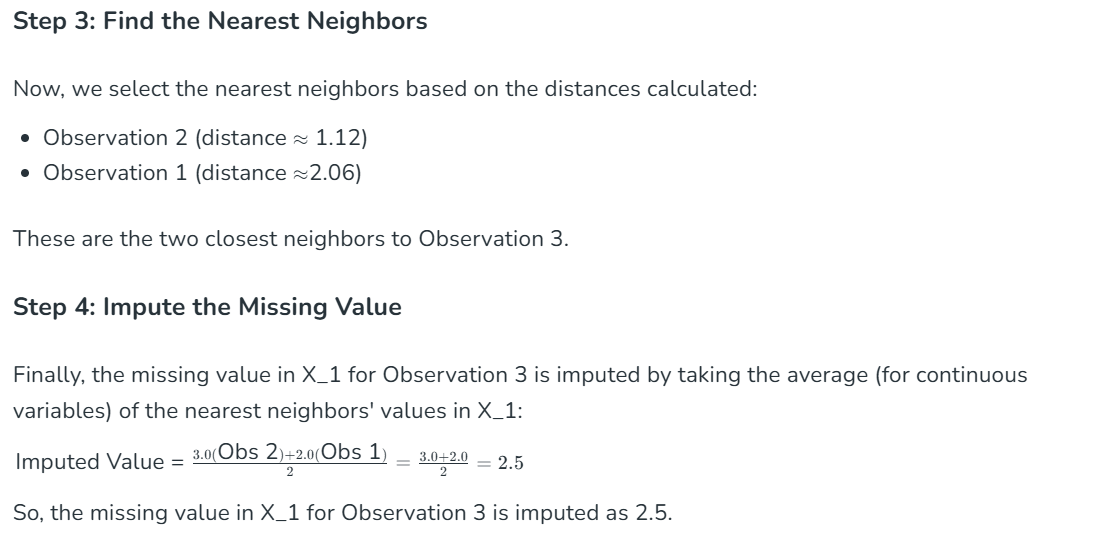
| **Observation** | **X\_1** | **X\_2** | **X\_3** |
| --- | --- | --- | --- |
| **1** | 2.0 | 1.0 | 3.0 |
| **2** | 3.0 | 2.0 | 4.0 |
| **3** | NaN | 1.5 | 5.0 |
| **4** | 5.0 | 3.5 | 6.0 |
| **5** | 4.0 | NaN | 4.5 |

Here, "NaN" represents missing values.

**Step 1: Identify the Missing Values**

Let's say you want to impute the missing value in X\_1​ for Observation 3.

****

****

**Summary**

Mathematically, the KNN Imputer works by:

1. **Calculating distances** between the observation with missing data and all other observations (using available features).
2. **Selecting the nearest neighbors** based on these distances.
3. **Imputing the missing value** by averaging (or voting, in case of categorical data) the values of the nearest neighbors for the feature with the missing data.

**Advantages of KNN Imputer**

* **Multivariate Approach**: By considering multiple variables, the KNN Imputer provides a more comprehensive estimation of missing values compared to univariate methods.
* **Flexibility**: The method can be customized by choosing different distance metrics and the number of neighbors (k) based on the dataset's characteristics.
* **Preservation of Data Integrity**: By using similar data points for imputation, the KNN Imputer helps preserve the original distribution and relationships within the dataset.

**Challenges and Considerations with KNN Imputer**

While the KNN Imputer offers several advantages, there are some challenges and considerations to keep in mind:

* **Computational Complexity**: The method can be computationally expensive, especially for large datasets, as it involves calculating distances between all data points.
* **Choice of 'k'**: Selecting the appropriate number of neighbors (k) is crucial. A small k might lead to overfitting, while a large k might smooth out important variations.
* **Handling Categorical Data**: The KNN Imputer is primarily designed for numerical data. Imputing categorical data requires additional preprocessing or alternative strategies.

Code example –

# import necessary libraries

import numpy as np

import pandas as pd

# import the KNNimputer class

from sklearn.impute import KNNImputer

# create dataset for marks of a student

dict = {'Maths': [80, 90, np.nan, 95],

'Chemistry': [60, 65, 56, np.nan],

'Physics': [np.nan, 57, 80, 78],

'Biology': [78, 83, 67, np.nan]}

# creating a data frame from the list

Before\_imputation = pd.DataFrame(dict)

# print dataset before imputation

print("Data Before performing imputation\n", Before\_imputation)

# create an object for KNNImputer

imputer = KNNImputer(n\_neighbors=2)

After\_imputation = imputer.fit\_transform(Before\_imputation)

# print dataset after performing the operation

print("\n\nAfter performing imputation\n", After\_imputation)

**Output:**

Data Before performing imputation  
 Maths Chemistry Physics Biology  
0 80.0 60.0 NaN 78.0  
1 90.0 65.0 57.0 83.0  
2 NaN 56.0 80.0 67.0  
3 95.0 NaN 78.0 NaN  
  
  
After performing imputation  
 [[80. 60. 68.5 78. ]  
 [90. 65. 57. 83. ]  
 [87.5 56. 80. 67. ]  
 [95. 58. 78. 72.5]]

**Note:** After transforming the data becomes a[numpy](https://www.geeksforgeeks.org/numpy-in-python-set-1-introduction/)[array.](https://www.geeksforgeeks.org/array-data-structure/)

KNNImputer code examples with different strategies

**ChatGPT said:**

The KNNImputer in scikit-learn is a powerful tool that fills in missing values using the **K-Nearest Neighbors** algorithm. It imputes missing values by finding the **k nearest rows** (using Euclidean distance) and then averaging their values for the missing feature.

**✅ Key Parameters of KNNImputer**

* **n\_neighbors**: Number of nearest neighbors to use.
* **weights**:
  + 'uniform': All neighbors have equal weight.
  + 'distance': Closer neighbors get more weight.
* **metric**: Distance metric (default is 'nan\_euclidean').

**📘 Libraries to Import**

python

CopyEdit

import numpy as np

import pandas as pd

from sklearn.impute import KNNImputer

**📌 Sample DataFrame with Missing Values**

python

CopyEdit

df = pd.DataFrame({

'Feature1': [1, 2, np.nan, 4, 5],

'Feature2': [3, np.nan, 1, 2, np.nan],

'Feature3': [5, 3, 4, np.nan, 2]

})

print("Original DataFrame:\n", df)

**✅ Example 1: KNNImputer with Default Strategy**

python

CopyEdit

imputer = KNNImputer(n\_neighbors=2) # Default weights='uniform'

df\_imputed = imputer.fit\_transform(df)

print("\nImputed with uniform weights:\n", pd.DataFrame(df\_imputed, columns=df.columns))

**✅ Example 2: KNNImputer with weights='distance'**

python

CopyEdit

imputer\_distance = KNNImputer(n\_neighbors=2, weights='distance')

df\_imputed\_dist = imputer\_distance.fit\_transform(df)

print("\nImputed with distance-based weights:\n", pd.DataFrame(df\_imputed\_dist, columns=df.columns))

**✅ Example 3: KNNImputer with More Neighbors**

python

CopyEdit

imputer\_3 = KNNImputer(n\_neighbors=3)

df\_imputed\_3 = imputer\_3.fit\_transform(df)

print("\nImputed with 3 neighbors:\n", pd.DataFrame(df\_imputed\_3, columns=df.columns))

**🧠 How It Works**

For each missing value:

* Finds the **k nearest rows** (with non-missing values in that column).
* Computes the **average** (or weighted average) of the feature from those neighbors.
* Replaces the missing value with that average.
* Explain IterativeImputer: multivariate imputation using models
* Explain MissingIndicator: adds binary features to indicate where values were missing

**3. fancyimpute**

⚠️ Deprecated/Not actively maintained — but still educational for advanced imputation.

**✅ Use Case:**

* Advanced methods like Matrix Completion, KNN, MICE (Multiple Imputation)

**🛠️ Methods:**

* KNN: similar to sklearn but with a different implementation
* IterativeSVD, SoftImpute: good for datasets with low-rank matrix assumptions

**💡 When to Use:**

* Academic/research settings
* Experimenting with different imputation strategies

**4. missingno**

**✅ Use Case:**

* Visualizing missing data patterns

**🛠️ Methods:**

* missingno.matrix(df): shows a bar for missing vs. available
* missingno.heatmap(df): correlation of missingness
* missingno.dendrogram(df): hierarchical clustering of nullity

**💡 When to Use:**

* During EDA
* To explore patterns and structure in missing data
* Before choosing an imputation method

**5. datawig**

**✅ Use Case:**

* Deep learning-based imputation

**🛠️ Tools:**

* Uses neural networks to impute missing values based on other columns

**💡 When to Use:**

* Textual/categorical data
* When conventional methods fail to capture data complexity
* When high accuracy in missing value prediction is needed

**6. Autoimpute**

**✅ Use Case:**

* Statistical imputation with multiple strategies

**🛠️ Tools:**

* Supports Single, Multiple, Hotdeck, Bayesian regression-based imputation

**💡 When to Use:**

* When you want a statistical imputation framework
* Can be used in survey or medical data preprocessing

1. **Handling duplicate records:**

Handling **duplicate records** is a key step in data cleaning. While **pandas** is the primary library used, there are **other libraries and tools** that complement it when working with more complex deduplication logic (e.g., fuzzy matching, record linkage). Here's a detailed breakdown:

**🧰 Libraries to Handle Duplicate Records in Python**

**1. pandas – The Primary Tool**

**✅ Use Case:**

* Quick identification and removal of **exact duplicate** rows
* Manipulating DataFrames during EDA or preprocessing

**🛠️ Key Methods:**

* df.duplicated()  
  → Returns boolean Series indicating duplicate rows.
* df.drop\_duplicates()  
  → Removes duplicate rows.
* df[df.duplicated(keep=False)]  
  → Filters only duplicates (useful for inspection).

**🔍 Options:**

* subset: Limit check to certain columns.
* keep: 'first', 'last', or False to keep no duplicates.

**💡 When to Use:**

* **Most use cases** when duplicates are exactly matching
* Simple, fast, and works out-of-the-box

**2. recordlinkage – For Fuzzy Deduplication**

**✅ Use Case:**

* Linking records across datasets (e.g., joining customer data from multiple sources)
* Identifying **near-duplicates** based on similarity scores

**🛠️ Key Features:**

* Index(), Compare(), Compare.exact(), Compare.string()
* Supports blocking/indexing for large datasets
* Uses metrics like Levenshtein, Jaro-Winkler, etc.

**💡 When to Use:**

* When duplicates are **not exactly equal** (e.g., typos: “John Smith” vs “Jon Smith”)
* Entity resolution tasks in messy data
* Matching across different datasets

**3. fuzzywuzzy / thefuzz (successor)**

**✅ Use Case:**

* Matching similar strings in a single or across columns
* Useful in deduplicating columns like names, addresses, etc.

**🛠️ Functions:**

* fuzz.ratio(), fuzz.partial\_ratio(), process.extract()

**💡 When to Use:**

* Cleaning **fuzzy duplicates** within string columns
* Great for **non-tabular**, string-based comparisons

**4. dask**

**✅ Use Case:**

* Parallelized version of pandas for **large-scale data**

**🛠️ Methods:**

* dask.dataframe.drop\_duplicates(), duplicated()

**💡 When to Use:**

* When you’re working with **very large datasets**
* Similar API to pandas but scales better

**5. datasketch**

**✅ Use Case:**

* Probabilistic deduplication using **MinHash** and **LSH**

**🛠️ Features:**

* MinHash, MinHashLSH, ideal for near-duplicate document detection

**💡 When to Use:**

* For **high-dimensional data** or when deduplicating based on entire documents or paragraphs
* NLP or content-based deduplication

**6. pyjanitor**

**✅ Use Case:**

* Enhances pandas with **more readable syntax**

**🛠️ Method:**

* df.remove\_duplicates(subset=..., keep=...)

**💡 When to Use:**

* You want **cleaner syntax** and additional chaining-friendly functionality

**🧠 Summary: When to Use Which**

| **Task** | **Best Tool** |
| --- | --- |
| Simple row-wise deduplication | pandas |
| Large dataset, parallel processing | dask |
| String-based or fuzzy deduplication | thefuzz, recordlinkage |
| Deduplicating documents/content | datasketch |
| Cleaner, chainable pandas-style code | pyjanitor |

**Note -** Scikit-learn (sklearn) does not provide direct functions to remove duplicate records in the way pandas does. This is because scikit-learn is a machine learning library focused on model training, transformation, and prediction—not on raw data wrangling or cleaning.

**Why does scikit-learn (sklearn) handle missing values but not duplicates?**

| **Aspect** | **Missing Values** | **Duplicate Records** |
| --- | --- | --- |
| 🔍 **Nature** | Feature-level issue | Row-level (entire record) issue |
| 🧠 **Model relevance** | Models can’t train on NaN values | Models *can* train on duplicates, but biased |
| 🔧 **Handling granularity** | Needs imputation (mean, median, etc.) | Just remove or deduplicate entire rows |
| 📦 **sklearn Support** | Yes: SimpleImputer, KNNImputer, etc. | ❌ Not in sklearn — use pandas |

**Why sklearn doesn't include duplicate removal:**

* **Duplicate handling is purely data hygiene**, not model logic.
* It’s a **one-time global operation**, independent of modeling steps.
* It’s better done using tools like:
  + pandas.drop\_duplicates()
  + Fuzzy matching libraries (e.g., fuzzywuzzy, recordlinkage)
* Including it in a pipeline could lead to **unintended behavior**, like deduplication during every cross-validation split.

**✅ Summary:**

| **Task** | **When to handle** | **Tool / Library** |
| --- | --- | --- |
| Remove duplicates | Before ML pipeline | pandas |
| Handle missing values | Inside ML pipeline | sklearn.impute |

1. **Handle Inconsistent Formatting:**

**Inconsistent Formatting**

**🔍 What is it?**

**Inconsistent formatting** refers to data entries that represent the same thing in different ways due to variations in:

* Capitalization
* Spelling
* Units
* Date/time formats
* Punctuation or whitespace

**🧾 Examples:**

| **Issue Type** | **Examples** | **Problem** |
| --- | --- | --- |
| Capitalization | "USA", "usa", "Usa" | Should be normalized to "USA" |
| Whitespace | "John", " John ", "john" | Should be "John" |
| Date Formats | "2024-01-01", "01/01/2024" | Ambiguous or inconsistent parsing |
| Units | "5kg", "5000g" | Different units, same measurement |
| Spelling Variants | "colour", "color" | British vs. American English |

**Ways to Handle Inconsistent Formatting in a DataFrame**

**🔹 1. Standardizing Text (Case, Spacing, Typos)**

* Convert all text to lowercase or uppercase
* Strip whitespace
* Fix typos using fuzzy matching

**📌 Example:**

python

CopyEdit

df['country'] = df['country'].str.strip().str.upper()

**📚 Libraries:**

* **Pandas** – for text manipulation (.str methods)
* **FuzzyWuzzy / RapidFuzz** – to match/fix similar strings

**✅ Use Case:**

* Normalize categorical values (e.g., " USA ", "usa", "Usa")

**🔹 2. Date/Time Formatting**

* Convert all dates to a standard format (datetime64)
* Handle different regional formats (MM/DD/YYYY vs DD/MM/YYYY)

**📌 Example:**

python

CopyEdit

df['date'] = pd.to\_datetime(df['date'], dayfirst=True, errors='coerce')

**📚 Libraries:**

* **Pandas** – pd.to\_datetime
* **Dateutil** – advanced date parsing
* **Arrow** – easier date manipulation

**✅ Use Case:**

* Cleaning transaction logs, timestamps, etc.

**🔹 3. Numeric Standardization (Units, Currency)**

* Convert all measurements to a single unit (e.g., cm → m)
* Remove currency symbols or commas from strings before converting

**📌 Example:**

python

CopyEdit

df['amount'] = df['amount'].str.replace('$', '').str.replace(',', '').astype(float)

**📚 Libraries:**

* **Pandas**
* **Quantities / Pint** – for unit conversions

**✅ Use Case:**

* Standardizing financial, scientific, or health-related data

**🔹 4. Replacing Values / Mapping**

* Map various representations to a single standard label

**📌 Example:**

python

CopyEdit

df['gender'] = df['gender'].replace({'M': 'Male', 'F': 'Female', 'male': 'Male'})

**📚 Libraries:**

* **Pandas** – .replace(), .map()

**✅ Use Case:**

* Gender, status, country codes, etc.

**🔹 5. Column Naming Cleanup**

* Remove special characters, whitespace
* Convert to snake\_case

**📌 Example:**

python

CopyEdit

df.columns = df.columns.str.strip().str.lower().str.replace(' ', '\_')

**📚 Libraries:**

* **Pandas**
* **inflection** – convert camelCase to snake\_case

**✅ Use Case:**

* Standardizing column names before modeling

**🔹 6. Language or Locale Normalization**

* Translate values to a standard language
* Normalize encodings (e.g., UTF-8)

**📌 Example:**

python

CopyEdit

df['description'] = df['description'].apply(lambda x: x.encode('utf-8').decode('utf-8'))

**📚 Libraries:**

* **LangDetect / TextBlob** – detect language
* **Unidecode** – remove accents (e.g., from “José” to “Jose”)

**✅ Use Case:**

* International datasets, social media, customer reviews

**📌 Summary Table**

| **Method** | **Library** | **Use Case** |
| --- | --- | --- |
| Text normalization | pandas, fuzzywuzzy | Country names, categories |
| Date formatting | pandas, dateutil | Timestamps, logs |
| Numeric cleaning | pandas, pint | Prices, units |
| Value mapping | pandas | Gender, status, currency codes |
| Column renaming | pandas, inflection | Prepare for modeling |
| Language/locale normalization | unidecode, langdetect | Multilingual datasets |

1. **Detect and Filter Outliers Data:**

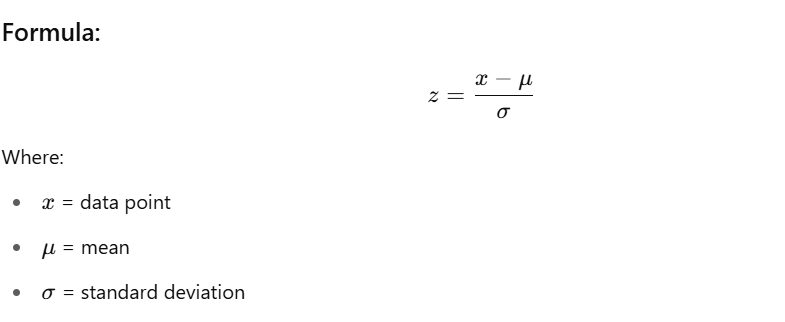
**Ways to Detect Outliers in a DataFrame**

**🔹 1. Z-Score Method (Standard Score)**

Outliers are points that are a certain number of standard deviations away from the mean.

**What is Z-Score?**

The **z-score** (standard score) tells you **how many standard deviations** a data point is from the mean.



**✅ Outlier Criteria (Rule of Thumb)**

* If the **absolute value of z > 3**, it's considered an **outlier**.
* You may adjust the threshold based on sensitivity, e.g. z > 2.5 for mild detection.

**📌 Example:**

python

CopyEdit

from scipy import stats

import numpy as np

z\_scores = np.abs(stats.zscore(df['column']))

df\_outliers = df[z\_scores > 3]

**📚 Libraries:**

* scipy, numpy, pandas

**✅ Use Case:**

* Normally distributed data (e.g., height, IQ scores)

**🔹 2. IQR Method (Interquartile Range)**

Outliers lie outside 1.5×IQR (Q1–1.5×IQR or Q3+1.5×IQR)

**📌 Example:**

python

CopyEdit

Q1 = df['column'].quantile(0.25)

Q3 = df['column'].quantile(0.75)

IQR = Q3 - Q1

df\_outliers = df[(df['column'] < (Q1 - 1.5 \* IQR)) | (df['column'] > (Q3 + 1.5 \* IQR))]

**📚 Libraries:**

* pandas

**✅ Use Case:**

* Any numerical feature, especially skewed data

**🔹 3. Boxplot Visualization**

Used for visually spotting outliers using boxplots.

**📌 Example:**

python

CopyEdit

import seaborn as sns

import matplotlib.pyplot as plt

sns.boxplot(x=df['column'])

plt.show()

**📚 Libraries:**

* seaborn, matplotlib

**✅ Use Case:**

* Exploratory data analysis (EDA)

**🔹 4. Isolation Forest (Machine Learning-Based)**

Unsupervised ML model that isolates anomalies.

**📌 Example:**

python

CopyEdit

from sklearn.ensemble import IsolationForest

clf = IsolationForest(contamination=0.05)

df['outlier'] = clf.fit\_predict(df[['column1', 'column2']])

df\_outliers = df[df['outlier'] == -1]

**📚 Libraries:**

* scikit-learn

**✅ Use Case:**

* Multivariate data, fraud detection, network intrusions

**🔹 5. DBSCAN Clustering**

Density-based clustering algorithm that labels sparse points as outliers.

**📌 Example:**

python

CopyEdit

from sklearn.cluster import DBSCAN

import numpy as np

db = DBSCAN(eps=0.5, min\_samples=5).fit(df[['column1', 'column2']])

df['outlier'] = (db.labels\_ == -1)

**📚 Libraries:**

* scikit-learn

**✅ Use Case:**

* Non-linear spatial clusters with noise/outliers

**🔹 6. LOF (Local Outlier Factor)**

Detects density-based local outliers.

**📌 Example:**

python

CopyEdit

from sklearn.neighbors import LocalOutlierFactor

lof = LocalOutlierFactor(n\_neighbors=20)

df['outlier'] = lof.fit\_predict(df[['column1', 'column2']])

df\_outliers = df[df['outlier'] == -1]

**📚 Libraries:**

* scikit-learn

**✅ Use Case:**

* When outliers are close to each other in dense regions

**🔹 7. Quantile Clipping**

Cap the values to a given percentile range (e.g., 1st to 99th)

**📌 Example:**

python

CopyEdit

df['column'] = df['column'].clip(lower=df['column'].quantile(0.01),

upper=df['column'].quantile(0.99))

**📚 Libraries:**

* pandas

**✅ Use Case:**

* To mitigate effect of extreme values without dropping

**📌 Summary Table**

| **Method** | **Library** | **Use Case** |
| --- | --- | --- |
| Z-score | scipy, numpy | Normal distributions |
| IQR | pandas | General numeric data |
| Boxplot | seaborn, matplotlib | Visual EDA |
| Isolation Forest | scikit-learn | Multivariate anomaly detection |
| DBSCAN | scikit-learn | Spatial clusters with noise |
| LOF | scikit-learn | Local density-based outliers |
| Quantile Clipping | pandas | Keep outliers but reduce impact |

**What is IQR (Interquartile Range)?**

The **IQR** is a measure of **statistical dispersion**, or how spread out the values in a dataset are. It helps you understand where the *middle 50%* of your data lies.

**💡 Definitions:**

* **Q1 (1st Quartile)**: The value below which 25% of the data falls (i.e., 25th percentile or quantile(0.25)).
* **Q3 (3rd Quartile)**: The value below which 75% of the data falls (i.e., 75th percentile or quantile(0.75)).
* **IQR = Q3 - Q1**: This represents the **middle 50%** of your data (between the 25th and 75th percentile).

**✅ Why Use IQR to Detect Outliers?**

IQR gives a robust view of central tendency that is **not affected by extreme values** (unlike mean and standard deviation).

Outliers are defined as values that lie **significantly outside** this middle 50%.

**✅ Why Use the 1.5 Multiplier?**

This is a **standard statistical rule** known as **Tukey’s Rule**:

A data point is considered an outlier if it is:

* **less than** Q1 − 1.5 × IQR
* **greater than** Q3 + 1.5 × IQR

**Why 1.5?**

* It’s a **convention**, not a strict rule — designed to **balance sensitivity and robustness**.
* It **flags unusually distant values** from the central range without being too aggressive.
* Statistically, under a normal distribution:
  + Only ~0.7% of values are expected to lie outside 1.5×IQR.
  + So it works well in practice even for skewed data.

You can use **3 × IQR** if you want to catch only **extreme outliers**, and reduce false positives.

**📊 Visual Representation**

java

CopyEdit

|--------|========================|--------|

Q1 Q3

|<-- IQR -->|

Lower Bound = Q1 - 1.5\*IQR

Upper Bound = Q3 + 1.5\*IQR

**✅ Python Example for Intuition**

python

CopyEdit

import numpy as np

data = [10, 12, 14, 15, 16, 17, 18, 19, 21, 25, 100] # 100 is an outlier

Q1 = np.percentile(data, 25)

Q3 = np.percentile(data, 75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

print(f"Q1: {Q1}, Q3: {Q3}, IQR: {IQR}")

print(f"Outlier bounds: {lower\_bound} to {upper\_bound}")

outliers = [x for x in data if x < lower\_bound or x > upper\_bound]

print(f"Outliers: {outliers}")

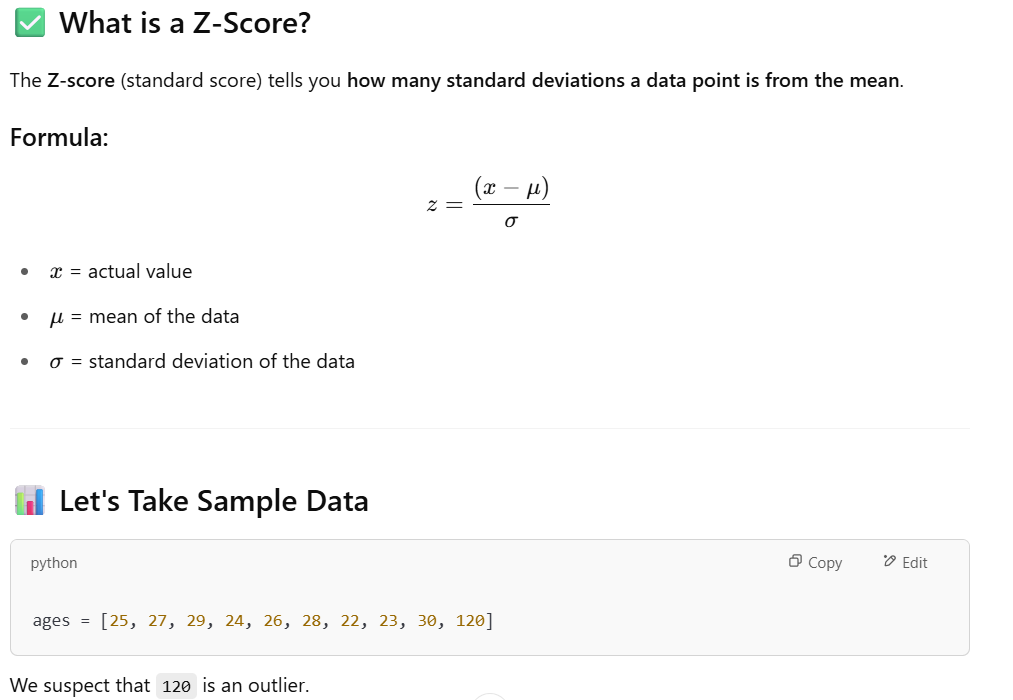
**✅ When Not to Use IQR?**

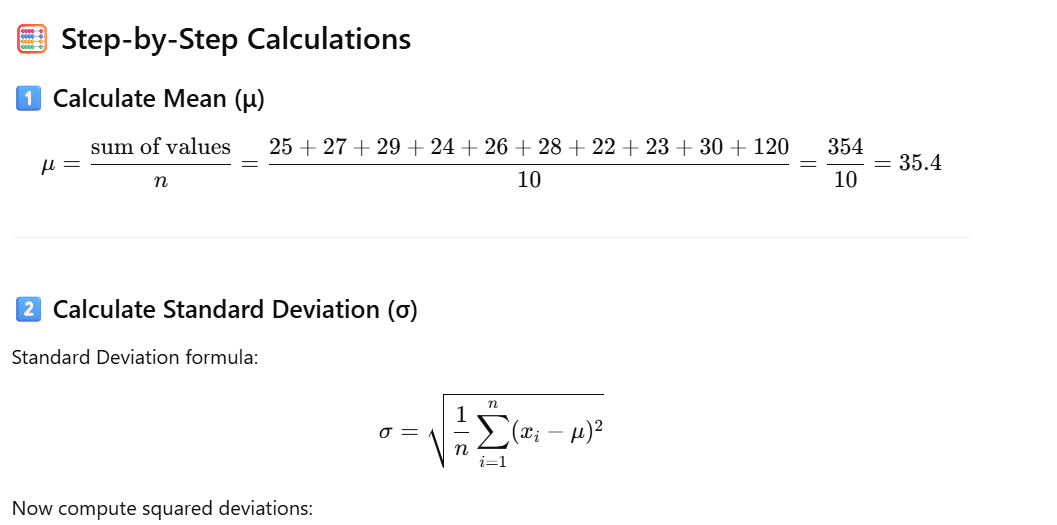
* **For normally distributed data**, z-score might be more appropriate.
* **For time series data**, other techniques like moving averages or seasonal decomposition might be better.

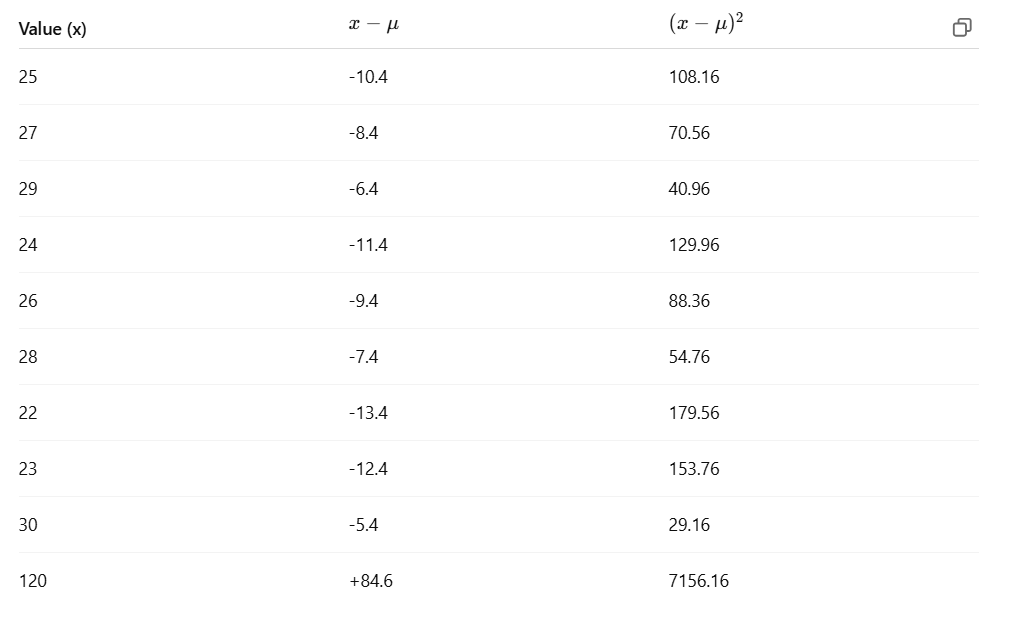
**🔁 Recap**

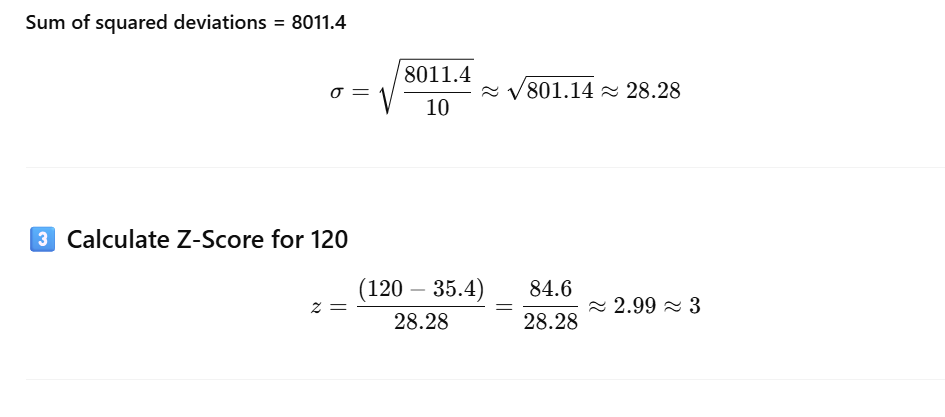
| **Term** | **Meaning** |
| --- | --- |
| Q1 | 25th percentile |
| Q3 | 75th percentile |
| IQR | Interquartile range (Q3 - Q1) |
| 1.5 \* IQR | Threshold for detecting moderate outliers |

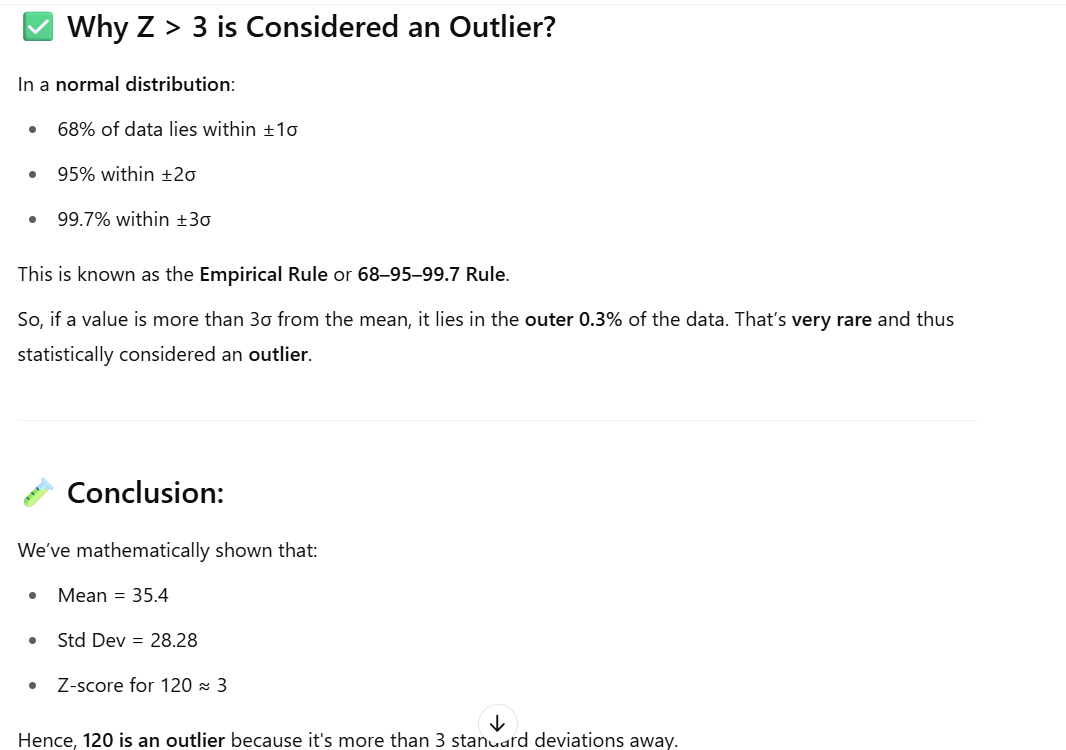
**Proof of the Z-Score Algorithms:**











import pandas as pd

import numpy as np

from scipy.stats import zscore

import matplotlib.pyplot as plt

# Step 1: Create a sample DataFrame

data = {

'age': [25, 27, 29, 24, 26, 28, 22, 23, 30, 120], # 120 is an outlier

'salary': [40000, 42000, 41000, 39000, 41500, 43000, 44000, 39500, 40500, 250000] # 250000 is an outlier

}

df = pd.DataFrame(data)

print("Original DataFrame:\n", df)

# Step 2: Calculate Z-scores

z\_scores = np.abs(zscore(df))

print("\nZ-scores:\n", z\_scores)

# Step 3: Identify rows with any Z-score > 3

outliers = (z\_scores > 3).any(axis=1)

print("\nOutlier mask:\n", outliers)

# Step 4: Filter out outliers

df\_cleaned = df[~outliers]

print("\nCleaned DataFrame (Outliers Removed):\n", df\_cleaned)

# Step 5: Optional - Plot before and after

plt.figure(figsize=(12, 5))

# Original data

plt.subplot(1, 2, 1)

plt.scatter(df['age'], df['salary'], color='red')

plt.title("With Outliers")

plt.xlabel("Age")

plt.ylabel("Salary")

# Cleaned data

plt.subplot(1, 2, 2)

plt.scatter(df\_cleaned['age'], df\_cleaned['salary'], color='green')

plt.title("Outliers Removed")

plt.xlabel("Age")

plt.ylabel("Salary")

plt.tight\_layout()

plt.show()

**Can I integrate z-score algo with ML pipeline:**

Yes, absolutely! You **can and should integrate the Z-score outlier detection method into your ML pipeline** — especially in the data preprocessing phase.

**✅ Why Integrate Z-score in ML Pipeline?**

Removing outliers:

* Helps improve model accuracy and generalization.
* Prevents models (especially linear ones) from being skewed by extreme values.
* Makes normalization/scaling more effective.

**🔧 How to Integrate Z-Score into a Scikit-learn Pipeline**

There are **2 main approaches**:

**🔹 Option 1: Manual Preprocessing Before the Pipeline**

You compute the Z-scores, remove rows where z > threshold, and then feed the clean data into your pipeline.

python

CopyEdit

import numpy as np

import pandas as pd

from sklearn.pipeline import Pipeline

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import StandardScaler

# Sample data

df = pd.DataFrame({

'age': [25, 27, 29, 24, 26, 28, 22, 23, 30, 120],

'salary': [50, 52, 53, 51, 49, 54, 45, 44, 56, 500]

})

# Detect outliers using z-score

from scipy.stats import zscore

z\_scores = np.abs(zscore(df))

# Remove outliers (z > 3)

filtered\_entries = (z\_scores < 3).all(axis=1)

df\_clean = df[filtered\_entries]

# Pipeline

pipeline = Pipeline([

('scaler', StandardScaler()),

('model', LinearRegression())

])

X = df\_clean[['age']]

y = df\_clean['salary']

pipeline.fit(X, y)

**🔹 Option 2: Use a Custom Transformer Inside the Pipeline**

This is more reusable and production-friendly.

python

CopyEdit

from sklearn.base import BaseEstimator, TransformerMixin

class ZScoreOutlierRemover(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, threshold=3.0):

self.threshold = threshold

def fit(self, X, y=None):

return self

def transform(self, X):

z\_scores = np.abs(zscore(X))

return X[(z\_scores < self.threshold).all(axis=1)]

# Pipeline

pipeline = Pipeline([

('remove\_outliers', ZScoreOutlierRemover()),

('scaler', StandardScaler()),

('model', LinearRegression())

])

X = df[['age', 'salary']]

y = df['salary']

pipeline.fit(X, y) # Notice: this may drop rows internally

⚠️ **Caveat:** If outlier removal affects y, you need to keep track of which rows are removed and apply the same mask to y.

**🧠 Use Cases**

| **Use Case** | **Z-Score Useful?** | **Notes** |
| --- | --- | --- |
| Linear regression | ✅ | Outliers heavily impact coefficients |
| Tree-based models (Random Forest, XGBoost) | ❌ Optional | These are more robust to outliers |
| Clustering (KMeans) | ✅ | Distance-based; sensitive to outliers |
| Image/Text classification | ❌ Not required | Often handled by domain-specific preprocessing |

**Can we integrate IQR outliers in ML pipeline:**

Yes, **IQR (Interquartile Range)-based outlier detection can also be integrated into an ML pipeline** — just like Z-score. It's commonly used in data preprocessing to remove extreme values that may skew model performance.

**✅ Why Use IQR in a Pipeline?**

* Robust to non-normal distributions (unlike Z-score, which assumes normality).
* Easy to compute using quantiles (Q1 and Q3).
* Works well for numerical tabular data where extreme values are invalid or rare.

**🔧 How to Integrate IQR in an ML Pipeline**

**🔹 Option 1: Manual Preprocessing**

python

CopyEdit

import pandas as pd

from sklearn.pipeline import Pipeline

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import StandardScaler

# Sample data

df = pd.DataFrame({

'age': [25, 27, 29, 24, 26, 28, 22, 23, 30, 120],

'salary': [50, 52, 53, 51, 49, 54, 45, 44, 56, 500]

})

# IQR-based outlier removal

Q1 = df.quantile(0.25)

Q3 = df.quantile(0.75)

IQR = Q3 - Q1

# Keep only rows within 1.5\*IQR

df\_clean = df[~((df < (Q1 - 1.5 \* IQR)) | (df > (Q3 + 1.5 \* IQR))).any(axis=1)]

# ML Pipeline

pipeline = Pipeline([

('scaler', StandardScaler()),

('model', LinearRegression())

])

X = df\_clean[['age']]

y = df\_clean['salary']

pipeline.fit(X, y)

**🔹 Option 2: Custom IQR Transformer in a Pipeline**

This is the **recommended scalable method**.

python

CopyEdit

from sklearn.base import BaseEstimator, TransformerMixin

import numpy as np

class IQROutlierRemover(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, factor=1.5):

self.factor = factor

def fit(self, X, y=None):

Q1 = X.quantile(0.25)

Q3 = X.quantile(0.75)

self.IQR\_bounds\_ = {

'lower': Q1 - self.factor \* (Q3 - Q1),

'upper': Q3 + self.factor \* (Q3 - Q1)

}

return self

def transform(self, X):

mask = ~((X < self.IQR\_bounds\_['lower']) | (X > self.IQR\_bounds\_['upper'])).any(axis=1)

self.mask\_ = mask

return X[mask]

# Create pipeline

pipeline = Pipeline([

('iqr\_outliers', IQROutlierRemover()),

('scaler', StandardScaler()),

('model', LinearRegression())

])

X = df[['age', 'salary']]

y = df['salary']

# Apply mask to both X and y

pipeline.named\_steps['iqr\_outliers'].fit(X)

X\_filtered = pipeline.named\_steps['iqr\_outliers'].transform(X)

y\_filtered = y[pipeline.named\_steps['iqr\_outliers'].mask\_]

# Fit remaining pipeline

pipeline.steps = pipeline.steps[1:] # remove custom step

pipeline.fit(X\_filtered, y\_filtered)

**📌 Summary of When to Use IQR vs Z-Score**

| **Criterion** | **IQR** | **Z-score** |
| --- | --- | --- |
| Assumes normal distribution | ❌ No | ✅ Yes |
| Robust to skewed data | ✅ Yes | ❌ Not robust |
| Easy to tune | ✅ Yes (factor=1.5 usually) | ✅ Yes (threshold=3) |
| Ideal for production | ✅ With custom transformer | ✅ With custom transformer |

**Comparison: Z-Score vs IQR**

| **Feature / Criterion** | **Z-Score** | **IQR Method** |
| --- | --- | --- |
| **Assumes Normal Distribution** | ✅ Yes | ❌ No (distribution-free) |
| **Best For** | Symmetrical, Gaussian (normal) data | Skewed or non-normal data |
| **Affected by Outliers** | ❌ Yes (mean and std can be skewed) | ✅ No (uses median and quantiles) |
| **Robustness** | Less robust to extreme values | More robust (resistant to outliers) |
| **Intuition** | Standard deviation-based distance | Range-based thresholding (middle 50%) |
| **Tunable Parameters** | Z-threshold (e.g. 2.5 or 3) | IQR multiplier (e.g. 1.5) |
| **Multivariate Extension** | ❌ Not easily extended | ❌ Needs separate handling for each feature |
| **Use in Production Pipelines** | ✅ Yes with scaling | ✅ Yes, especially with skewed data |
| **Handling Sparse / Sparse+Skewed Data** | ❌ Not recommended | ✅ Recommended |

**💡 When to Use What?**

**✅ Use Z-Score When:**

* Data is approximately normally distributed.
* You have fewer features or well-scaled data.
* You are fine with distance-based assumptions.
* You want a simple, fast, and interpretable method.

**✅ Use IQR When:**

* Data is **skewed**, has **heavy tails**, or non-normal.
* You're working with real-world messy data (e.g., financial transactions, user behavior).
* You want a robust method that’s not affected by extreme values.

**🎓 Summary Table**

| **Data Scenario** | **Preferred Method** |
| --- | --- |
| Normally distributed features | Z-Score |
| Skewed distributions | IQR |
| Data with many extreme outliers | IQR |
| Need for intuitive standard scoring | Z-Score |
| Using Boxplot visualization | IQR |

**✅ Example Conclusion:**

| **Data Sample** | **Best Method** |
| --- | --- |
| Ages of people (normal) | Z-Score |
| House prices (skewed, heavy outliers) | IQR |
| Sensor readings (normal, low noise) | Z-Score |
| E-commerce order amounts | IQR |

**What is Isolation Forest outlier detection Algo?**

Isolation Forest is an **unsupervised anomaly detection algorithm** that isolates anomalies instead of profiling normal data points.

**🔧 How it works:**

* Randomly selects a feature and a split value.
* Builds isolation trees (binary trees).
* Outliers get isolated **faster** → have **shorter average path length** in the tree.
* Scores each point → lower score means higher likelihood of being an anomaly.

**🆚 Comparison: Isolation Forest vs Z-Score vs IQR**

| **Feature / Criterion** | **Z-Score** | **IQR Method** | **Isolation Forest** |
| --- | --- | --- | --- |
| **Type** | Statistical | Statistical | Machine Learning |
| **Assumption on Data** | Normal distribution | No assumptions | No distribution assumption |
| **Multivariate Detection** | ❌ Not ideal | ❌ Hard | ✅ Natively supports multivariate data |
| **Handles High Dimensions** | ❌ No | ❌ No | ✅ Yes |
| **Handles Skewed Data** | ❌ Poor | ✅ Good | ✅ Excellent |
| **Robustness to Outliers** | ❌ Sensitive | ✅ Robust | ✅ Very Robust |
| **Tunable Parameters** | Z threshold | IQR multiplier | Contamination rate (expected outlier %) |
| **Supervised Required?** | ❌ No | ❌ No | ❌ No (unsupervised) |
| **Interpretability** | ✅ High | ✅ High | ❌ Low (black box) |
| **Speed/Complexity** | ✅ Fast | ✅ Fast | ⚠️ Slower (tree-based model building) |
| **Best Use Case** | Clean, normal data | Small-scale, skewed | High-dimensional, complex, large-scale data |

**✅ When to Use Each?**

**1. Z-Score**

* Simple, interpretable.
* Use if data is **univariate and normally distributed**.
* Use in **exploratory or real-time pipelines** when speed is critical.

**2. IQR**

* Ideal for **univariate skewed data** (e.g., income, sales).
* Great for **small or medium datasets**.
* Visualizable via boxplots.

**3. Isolation Forest**

* Use for **multivariate**, **high-dimensional**, or **large datasets**.
* Great for **fraud detection**, **network intrusion**, **complex real-world data**.
* Can handle **interactions between features** that simple methods miss.

**Data transformation**

At this stage, the data is formatted for model compatibility. This process includes normalizing or scaling numerical values, encoding categorical variables, and transforming skewed distributions to improve model learning.

**What is Normalization and Standardization:**

**What is Normalization?**

**Normalization** is the process of **scaling numeric data** so that its values fall within a specific range, typically **[0, 1]**.

**📌 Why Normalize?**

Some machine learning algorithms (like **KNN**, **Neural Networks**, or **Gradient Descent-based models**) perform better when input features are on a similar scale.

**🛠️ How to Normalize?**

The formula for **Min-Max Normalization** is:

x′=x−xmin⁡xmax⁡−xmin⁡x' = \frac{x - x\_{\min}}{x\_{\max} - x\_{\min}}x′=xmax​−xmin​x−xmin​​

**📚 Libraries You Can Use**

1. **scikit-learn** – Recommended
2. **Pandas** – Manual calculation
3. **TensorFlow / PyTorch** – When working with deep learning
4. **NumPy** – Manual normalization

**✅ Code Example Using scikit-learn**

Here’s how you can normalize a dataset using MinMaxScaler from sklearn.preprocessing.

python

CopyEdit

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

# Sample data

data = pd.DataFrame({

'age': [20, 30, 40, 50],

'income': [20000, 50000, 100000, 150000]

})

# Initialize the scaler

scaler = MinMaxScaler()

# Fit and transform the data

normalized\_data = scaler.fit\_transform(data)

# Convert back to DataFrame

normalized\_df = pd.DataFrame(normalized\_data, columns=data.columns)

print(normalized\_df)

**🔍 Output:**

markdown

CopyEdit

age income

0 0.00 0.00000

1 0.33 0.23077

2 0.67 0.61538

3 1.00 1.00000

Would you like to see how to normalize manually using NumPy or Pandas next? Or should we move on to **Standardization**?

Great! Let’s move on to **Standardization**, another essential transformation technique.

**✅ What is Standardization?**

**Standardization** (also called **Z-score normalization**) transforms data so that it has:

* A **mean (μ)** of 0
* A **standard deviation (σ)** of 1

Each value is adjusted based on how far it is from the mean, relative to the spread of the data.

**📌 Why Standardize?**

Many ML algorithms (like **Logistic Regression**, **SVM**, **K-Means**, **PCA**, etc.) **assume or perform better** when the data:

* Follows a **standard normal distribution** (bell curve)
* Is centered around 0 with unit variance

**📐 Formula for Standardization:**

x′=x−μσx' = \frac{x - \mu}{\sigma}x′=σx−μ​

Where:

* xxx is the original value
* μ\muμ is the mean of the feature
* σ\sigmaσ is the standard deviation

**🛠️ How to Standardize (with scikit-learn)**

Here’s a Python example using StandardScaler from sklearn.preprocessing.

**🔢 Code Example:**

python

CopyEdit

import pandas as pd

from sklearn.preprocessing import StandardScaler

# Sample data

data = pd.DataFrame({

'age': [20, 30, 40, 50],

'income': [20000, 50000, 100000, 150000]

})

# Initialize the scaler

scaler = StandardScaler()

# Fit and transform the data

standardized\_data = scaler.fit\_transform(data)

# Convert back to DataFrame

standardized\_df = pd.DataFrame(standardized\_data, columns=data.columns)

print(standardized\_df)

**🔍 Output Example (values will have mean ≈ 0, std ≈ 1):**

markdown

CopyEdit

age income

0 -1.341641 -1.183216

1 -0.447214 -0.507093

2 0.447214 0.507093

3 1.341641 1.183216

**🔧 Alternative Libraries You Can Use:**

* scipy.stats.zscore(data) – also does standardization
* pandas (manual with .mean() and .std())
* Deep learning frameworks like TensorFlow and PyTorch also have standardization layers

**Scaling numerical values:**

**Scaling** is the process of transforming numerical features so that they fall within a specific range or follow a particular distribution. It ensures that no single feature dominates due to its magnitude.

**📌 Why Do We Scale Numerical Values?**

Many ML algorithms are **sensitive to the scale of features**, especially:

* Distance-based algorithms: **KNN**, **K-Means**, **SVM**
* Gradient-based models: **Logistic Regression**, **Neural Networks**
* Dimensionality reduction: **PCA**

Without scaling, features with **large numeric ranges** (like income or age) can **skew the learning process**.

**🛠️ Common Scaling Techniques**

| **Technique** | **Description** | **Range or Target** |
| --- | --- | --- |
| **Min-Max Scaling** | Scales to a fixed range (e.g., [0, 1]) | [0, 1] (default) |
| **Standardization** | Scales to mean = 0 and std dev = 1 | Centered & normal |
| **Robust Scaling** | Uses median and IQR (robust to outliers) | Variable |
| **MaxAbs Scaling** | Scales by dividing by the max absolute value | [-1, 1] |
| **Quantile Transform** | Maps data to uniform or normal distribution | Customizable |

**📚 Libraries for Scaling**

You can use the following Python libraries:

**✅ scikit-learn (sklearn.preprocessing)**

Most recommended and widely used.

python

CopyEdit

from sklearn.preprocessing import MinMaxScaler, StandardScaler, RobustScaler, MaxAbsScaler

**✅ pandas**

You can manually apply scaling using .mean(), .std(), .min(), .max(), etc.

**✅ numpy**

For manual scaling and operations.

**✅ scipy**

Has scipy.stats.zscore() for standardization.

**✅ tensorflow / pytorch**

Used in deep learning pipelines (tf.keras.layers.Normalization, etc.)

**💡 Summary of Use Cases:**

| **Algorithm** | **Needs Scaling?** | **Preferred Method** |
| --- | --- | --- |
| **K-Nearest Neighbors (KNN)** | ✅ Yes | Min-Max or StandardScaler |
| **Support Vector Machines (SVM)** | ✅ Yes | StandardScaler |
| **Neural Networks** | ✅ Yes | Standard or Min-Max |
| **Decision Trees / Random Forest** | ❌ No | Not needed |
| **Gradient Descent Algorithms** | ✅ Yes | StandardScaler |
| **PCA / Clustering** | ✅ Yes | StandardScaler |

Would you like me to show a quick **code demo comparing multiple scalers**? Or move on to **encoding categorical variables**?

**What is Encoding Categorical Variables?**

In datasets, many features are **categorical** (non-numeric), like:

* Gender: Male, Female
* Color: Red, Blue, Green
* Country: USA, India, UK

Machine learning algorithms **require numerical input**, so **encoding** means converting these categorical values into **numbers** without losing their meaning.

**📌 Why Encode Categorical Variables?**

* ML algorithms cannot directly work with text or labels.
* Encoding allows the model to interpret categories as numeric values.
* Proper encoding preserves information and prevents misleading relationships.

**🛠️ Common Encoding Techniques**

| **Technique** | **Description** | **When to Use** |
| --- | --- | --- |
| **Label Encoding** | Assigns an integer to each category | Ordinal categories (with order) |
| **One-Hot Encoding** | Creates binary columns for each category | Nominal categories (no order) |
| **Ordinal Encoding** | Similar to label encoding but respects order | Ordered categories (e.g. small, medium, large) |
| **Target Encoding** | Replaces categories with target variable mean | When you have target leakage control and large categories |
| **Binary Encoding** | Encodes categories as binary digits | High-cardinality categories |

**What is Transforming Skewed Distributions?**

When your data is **skewed**, it means the values are not symmetrically distributed — they lean toward one side.

* **Right skew (positive skew)**: Most values are small but a few very large values stretch out the right tail.
* **Left skew (negative skew)**: Most values are large but a few very small values stretch out the left tail.

Many machine learning models assume data is roughly **normally distributed** (symmetrical), so skewness can hurt model accuracy and statistical analysis.

**Transforming skewed data** means applying mathematical functions to reduce this skewness and make the data distribution more balanced (closer to normal).

**Techniques to Transform Skewed Data**

| **Technique** | **Description** | **Use When** |
| --- | --- | --- |
| **Log Transformation** | Replace each value xxx with log⁡(x)\log(x)log(x) | Right skew; data > 0 |
| **Square Root Transformation** | Replace xxx with x\sqrt{x}x​ | Moderate right skew; data ≥ 0 |
| **Reciprocal Transformation** | Replace xxx with 1/x1/x1/x | Strong right skew; data > 0 |
| **Box-Cox Transformation** | A power transform that optimizes λ\lambdaλ to make data normal | Right skew; data > 0 |
| **Yeo-Johnson Transformation** | Like Box-Cox but supports zero and negative values | Skewed data with zero/negatives |

**Why Use These?**

* **Reduce the influence of extreme values (outliers)**
* **Improve model assumptions of normality**
* **Help models converge faster and perform better**

**Code Example: Log Transformation with Before and After Data**

python

CopyEdit

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

# Sample skewed data (income with big outlier)

data = pd.DataFrame({

'income': [2000, 3000, 4000, 5000, 120000]

})

print("Before Transformation:")

print(data)

# Apply log transformation (only works if all values > 0)

data['log\_income'] = np.log(data['income'])

print("\nAfter Log Transformation:")

print(data)

# Visualize distributions

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

plt.subplot(1, 2, 1)

plt.hist(data['income'], bins=5, color='skyblue')

plt.title('Original Income Distribution')

plt.subplot(1, 2, 2)

plt.hist(data['log\_income'], bins=5, color='lightgreen')

plt.title('Log Transformed Income Distribution')

plt.show()

**Explanation:**

* The **original income** has a huge outlier (120,000), making the data skewed to the right.
* The **log transformation** compresses large values, reducing skewness and making the data more symmetrical.
* The histogram shows how the data distribution changes visibly.

Would you like me to show examples for **Box-Cox** or **Yeo-Johnson** transformations next?

4.1-mini

**You said:**

yes please

**ChatGPT said:**

Great! Let me explain **Box-Cox** and **Yeo-Johnson** transformations along with code examples showing before and after data.

**1️⃣ Box-Cox Transformation**

* It’s a **power transform** that finds the best exponent λ\lambdaλ to make data more normal.
* **Requires all data to be positive** (> 0).
* Often used to reduce right skewness and stabilize variance.

**Box-Cox Code Example:**

python

CopyEdit

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from scipy.stats import boxcox

# Sample skewed positive data

data = pd.DataFrame({

'income': [2000, 3000, 4000, 5000, 120000]

})

print("Before Box-Cox Transformation:")

print(data)

# Apply Box-Cox transform (returns transformed data and lambda)

data['income\_boxcox'], fitted\_lambda = boxcox(data['income'])

print(f"\nLambda used for Box-Cox: {fitted\_lambda:.4f}")

print("\nAfter Box-Cox Transformation:")

print(data)

# Plot distributions

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

plt.subplot(1, 2, 1)

plt.hist(data['income'], bins=5, color='skyblue')

plt.title('Original Income Distribution')

plt.subplot(1, 2, 2)

plt.hist(data['income\_boxcox'], bins=5, color='lightcoral')

plt.title('Box-Cox Transformed Income')

plt.show()

**2️⃣ Yeo-Johnson Transformation**

* Similar to Box-Cox but **supports zero and negative values**.
* Also a power transform that tries to normalize the data.
* Useful when data contains zeros or negative numbers.

**Yeo-Johnson Code Example (using scikit-learn):**

python

CopyEdit

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PowerTransformer

# Sample skewed data including zero

data = pd.DataFrame({

'score': [0, 1, 2, 3, 100]

})

print("Before Yeo-Johnson Transformation:")

print(data)

# Initialize transformer

pt = PowerTransformer(method='yeo-johnson')

# Fit and transform the data

data['score\_yeojohnson'] = pt.fit\_transform(data[['score']])

print("\nAfter Yeo-Johnson Transformation:")

print(data)

# Plot distributions

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

plt.subplot(1, 2, 1)

plt.hist(data['score'], bins=5, color='skyblue')

plt.title('Original Score Distribution')

plt.subplot(1, 2, 2)

plt.hist(data['score\_yeojohnson'], bins=5, color='lightgreen')

plt.title('Yeo-Johnson Transformed Score')

plt.show()

**Summary:**

* **Box-Cox**: Only for positive data, finds optimal power transform.
* **Yeo-Johnson**: Works for all real numbers including zero/negatives.

Let me know if you want me to explain how to choose between these or other transformations!

**Some other preprocessing library classes:**

**Binarizer**

**What it does:**

* Converts numerical features to binary (0 or 1) based on a **threshold**.
* Values above the threshold become 1, others 0.

**Use case:**

* Useful for **feature engineering** when you want to create binary indicators from continuous data.

**Code Example:**

python

CopyEdit

from sklearn.preprocessing import Binarizer

import numpy as np

data = np.array([[1.2, -0.8, 3.5], [0.5, 2.1, -1.0]])

# Initialize Binarizer with threshold 1.0

binarizer = Binarizer(threshold=1.0)

binarized\_data = binarizer.fit\_transform(data)

print("Original Data:\n", data)

print("Binarized Data (threshold=1.0):\n", binarized\_data)

**2️⃣ Normalizer**

**What it does:**

* Scales **each sample (row)** to have unit norm (length 1).
* Normalizes data **row-wise**, not feature-wise.

**Use case:**

* Useful when you want to focus on **direction** of data vectors, such as in text classification or clustering.

**Code Example:**

python

CopyEdit

from sklearn.preprocessing import Normalizer

import numpy as np

data = np.array([[3, 4], [1, 2]])

normalizer = Normalizer(norm='l2')

normalized\_data = normalizer.fit\_transform(data)

print("Original Data:\n", data)

print("L2 Normalized Data:\n", normalized\_data)

**3️⃣ PolynomialFeatures**

**What it does:**

* Generates new features by creating **polynomial combinations** (up to a given degree) of existing features.
* Expands feature space with interactions and power terms.

**Use case:**

* Allows linear models to capture **non-linear relationships** by including polynomial terms.

**Code Example:**

python

CopyEdit

from sklearn.preprocessing import PolynomialFeatures

import numpy as np

data = np.array([[2, 3]])

poly = PolynomialFeatures(degree=2, include\_bias=False)

poly\_features = poly.fit\_transform(data)

print("Original Data:\n", data)

print("Polynomial Features (degree=2):\n", poly\_features)

If you want, I can explain how these are used in typical ML pipelines!

**What is Feature engineering:**

New features are created based on existing data to better capture underlying patterns. This process might include extracting time-based variables, combining fields, or applying domain knowledge to enrich the dataset.

**Common Techniques in Feature Engineering**

Here’s a categorized list of key techniques:

**1. Feature Creation**

* **Interaction terms** (e.g., feature1 \* feature2)
* **Polynomial features** (e.g., x^2, x^3)
* **Date/time decomposition** (extract year, month, day, hour from a timestamp)
* **Aggregations** (e.g., mean, sum, count over groups)
* **Domain-specific logic** (e.g., BMI = weight / height²)

✅ **Library Support:**

* sklearn.preprocessing.PolynomialFeatures
* pandas for date/time and domain logic

**2. Feature Selection (optional but related)**

* **Removing low-variance features**
* **Correlation filtering**
* **Recursive Feature Elimination (RFE)**

✅ **Library Support:**

* sklearn.feature\_selection
* statsmodels

**Common Libraries & Classes for Feature Engineering**

| **Purpose** | **Library/Class** |
| --- | --- |
| Polynomial Features | sklearn.preprocessing.PolynomialFeatures |
| Categorical Encoding | sklearn.preprocessing.OneHotEncoder, LabelEncoder, category\_encoders |
| Scaling/Normalization | StandardScaler, MinMaxScaler, Normalizer |
| Binning | pandas.cut, KBinsDiscretizer |
| Datetime Decomposition | pandas.to\_datetime, .dt.month, .dt.weekday, etc. |
| Missing Value Imputation | sklearn.impute.SimpleImputer, pandas.fillna |
| Feature Selection | sklearn.feature\_selection, SelectKBest, RFE |

**Feature selection**

Not all features contribute equally to model performance. This step involves selecting the most relevant variables and removing redundant or irrelevant ones, which helps reduce overfitting and improve efficiency.

**What is Feature Selection?**

**Feature Selection** is the process of identifying and selecting the **most relevant and important features** (input variables) from the dataset, while removing irrelevant, redundant, or noisy features.

It helps build simpler, faster, and more accurate machine learning models by reducing dimensionality.

**🎯 Why Feature Selection is Important**

* ✅ **Improves model accuracy** by reducing overfitting.
* ✅ **Reduces training time** and computational cost.
* ✅ **Simplifies models** for better interpretability.
* ✅ Helps avoid the **curse of dimensionality**.

**📌 Types of Feature Selection Techniques**

Feature selection techniques are generally divided into three categories:

**1. Filter Methods**

Use statistical tests and metrics **independent of any model**.

**Techniques:**

* **Variance Threshold** – Removes features with low variance.
* **Correlation Matrix** – Identifies and removes highly correlated features.
* **Chi-Square Test** – Measures association between categorical features and labels.
* **ANOVA F-test** – For comparing categorical targets with continuous features.

**📚 Classes:**

* sklearn.feature\_selection.VarianceThreshold
* sklearn.feature\_selection.SelectKBest
* sklearn.feature\_selection.chi2
* sklearn.feature\_selection.f\_classif

**2. Wrapper Methods**

Use a predictive model to score feature subsets by training repeatedly on different combinations.

**Techniques:**

* **Recursive Feature Elimination (RFE)** – Removes least important features iteratively.
* **Sequential Feature Selection** – Adds/removes features one at a time based on model performance.

**📚 Classes:**

* sklearn.feature\_selection.RFE
* sklearn.feature\_selection.SequentialFeatureSelector

**3. Embedded Methods**

Feature selection is **built into the model training** process.

**Techniques:**

* **Lasso Regression (L1 penalty)** – Shrinks some feature coefficients to zero.
* **Decision Trees and Random Forests** – Provide feature importance scores.

**📚 Classes:**

* sklearn.linear\_model.Lasso
* sklearn.ensemble.RandomForestClassifier
* xgboost.XGBClassifier (for tree-based selection)

**🧪 Example: Filter Method using SelectKBest with Chi-Square**

python

CopyEdit

from sklearn.datasets import load\_iris

from sklearn.feature\_selection import SelectKBest, chi2

import pandas as pd

# Load data

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = iris.target

# Apply SelectKBest

selector = SelectKBest(score\_func=chi2, k=2)

X\_selected = selector.fit\_transform(X, y)

# Show selected features

selected\_columns = X.columns[selector.get\_support()]

print("Selected Features:", selected\_columns.tolist())

**📊 Summary Table**

| **Category** | **Technique** | **Library Class** |
| --- | --- | --- |
| Filter | Variance Threshold | VarianceThreshold |
| Filter | Chi2, ANOVA F-test | SelectKBest, chi2, f\_classif |
| Wrapper | Recursive Feature Elim. | RFE |
| Wrapper | Sequential Selection | SequentialFeatureSelector |
| Embedded | Lasso Regularization | Lasso |
| Embedded | Tree-Based Importance | RandomForestClassifier, XGBClassifier |

Would you like examples of wrapper or embedded methods too?

**Data splitting**

The cleaned and engineered dataset is divided into training, validation, and test sets. Doing this ensures that the model is evaluated on unseen data and generalizes to real-world scenarios.

**Data Splitting** is the process of dividing your dataset into multiple parts—**typically training, validation, and test sets**—to ensure that your machine learning model is **trained**, **tuned**, and **evaluated** properly.

This process helps the model learn effectively, validate its performance during tuning, and be tested on unseen data to simulate real-world scenarios.

**🎯 Why is Data Splitting Important?**

* ✅ Prevents **overfitting** by validating on unseen data.
* ✅ Measures **generalization performance**.
* ✅ Allows for **model tuning** without data leakage.
* ✅ Ensures **reliable evaluation** metrics.

**📌 Common Data Splits**

| **Split Type** | **Purpose** |
| --- | --- |
| **Training** | Used to train the model. |
| **Validation** | Used to tune hyperparameters and prevent overfitting. |
| **Test** | Used for final model evaluation on unseen data. |

Typical ratios:

* **Train/Validation/Test** = 60/20/20 or 70/15/15 or 80/10/10

**📚 Libraries and Classes**

**1. sklearn.model\_selection.train\_test\_split**

* Used to split data into training and test (or validation) sets.

**2. sklearn.model\_selection.KFold / StratifiedKFold**

* For cross-validation: splits data into k-folds for model evaluation.

**3. sklearn.model\_selection.GroupKFold / TimeSeriesSplit**

* Specialized for grouped or time-series data.

**🧪 Basic Example Using train\_test\_split**

python

CopyEdit

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

import pandas as pd

# Load dataset

iris = load\_iris()

X = pd.DataFrame(iris.data, columns=iris.feature\_names)

y = iris.target

# Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, random\_state=42

)

print("Training size:", X\_train.shape)

print("Test size:", X\_test.shape)

**🔁 Example Using K-Fold Cross-Validation**

python

CopyEdit

from sklearn.model\_selection import KFold

import numpy as np

data = np.arange(10)

kf = KFold(n\_splits=5)

for train\_index, test\_index in kf.split(data):

print("Train:", train\_index, "Test:", test\_index)

**📊 Summary Table**

| **Technique** | **Use Case** | **Class/Function** |
| --- | --- | --- |
| Simple Train/Test Split | Standard splitting | train\_test\_split |
| Cross Validation | Robust evaluation | KFold, StratifiedKFold |
| Group-based Splitting | Group-aware tasks | GroupKFold |
| Time Series Split | Sequential/temporal data | TimeSeriesSplit |

**Final review**

Before modeling, a final check ensures that all preprocessing steps were correctly applied. This stage involves verifying distributions, feature quality, and data splits to prevent issues like data leakage or imbalance.

**What is Final Review?**

**Final Review** is the **last quality check** before training your machine learning model. It ensures that:

* ✅ All **preprocessing steps** were correctly applied.
* ✅ The **data is clean, well-balanced**, and **appropriately split**.
* ✅ There is **no data leakage** (information from the test set appearing in training).
* ✅ Features are in a form that the model can understand.

Think of this step as a **pre-launch checklist** before model building begins.

**🎯 Why Final Review Matters**

* Prevents **model bias** due to poor splits or data imbalance.
* Ensures **correct encoding, scaling, and transformation** of features.
* Avoids **data leakage**, which leads to misleadingly high performance.
* Confirms **consistency** in training, validation, and test sets.

**🔍 Key Techniques in Final Review**

| **Task** | **Goal** |
| --- | --- |
| Check for **Data Leakage** | Ensure target variable or derived features aren't in input data |
| Review **Feature Distributions** | Confirm no distortion after scaling/encoding |
| Validate **Scaling/Encoding** | Ensure transformations were applied consistently |
| Check for **Missing Values** | No missing values should remain |
| Verify **Target Distribution** | Check for **class imbalance** (esp. in classification) |
| Ensure **Correct Data Split** | Confirm stratification or time-based split logic |

**🧪 Libraries and Tools Used**

| **Tool/Library** | **Purpose** |
| --- | --- |
| pandas | Inspect datasets, check nulls, dtypes, distributions |
| seaborn, matplotlib | Plot feature distributions, histograms, boxplots |
| sklearn.preprocessing | Check scaling/encoding was applied |
| sklearn.model\_selection | Validate train/test/val splits |
| sklearn.metrics | Evaluate class balance, distribution, etc. |

**📊 Code Example: Final Review Checklist**

python

CopyEdit

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

# Assume you have a cleaned and transformed dataset

df = pd.read\_csv("final\_dataset.csv")

# 1. Check for null values

print("Missing values:\n", df.isnull().sum())

# 2. Review distributions

sns.histplot(df['feature1'], kde=True)

plt.title("Distribution of Feature1")

plt.show()

# 3. Check target distribution

sns.countplot(x='target', data=df)

plt.title("Target Class Distribution")

plt.show()

# 4. Split data again to ensure no leakage

X = df.drop('target', axis=1)

y = df['target']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=0.2, stratify=y, random\_state=42

)

# 5. Final shape check

print("Train shape:", X\_train.shape, "| Test shape:", X\_test.shape)

**✅ Final Thoughts**

Final review is not about adding new transformations, but about **auditing** the entire pipeline:

* Did I scale my data?
* Are categories encoded correctly?
* Is class imbalance addressed?
* Was stratified splitting done?
* Are there any data leaks?