Data Preprocessing Part 1

*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:

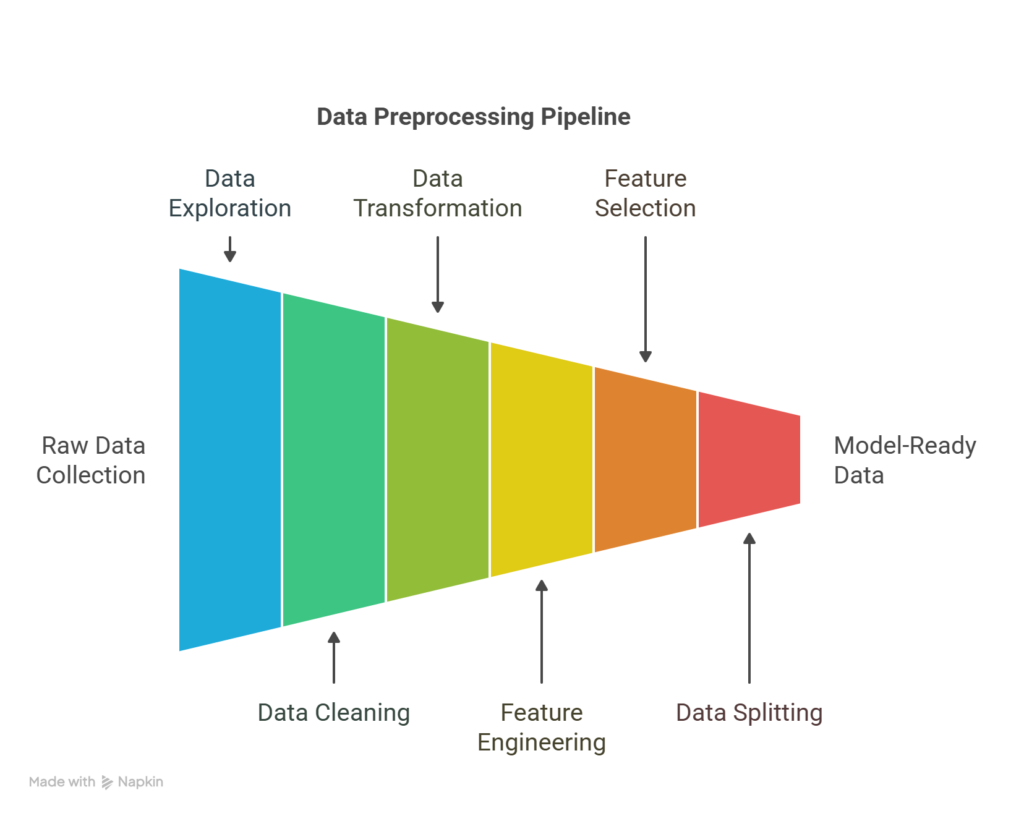
* + **Improves model accuracy:** Clean and well-structured data enables algorithms to learn patterns more effectively, leading to better predictions and outcomes.
  + **Reduces noise and inconsistencies:** Removing irrelevant or erroneous data helps prevent misleading insights and model confusion.
  + **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: fit() & transform() vs fit\_transform() methods of imputer**

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**

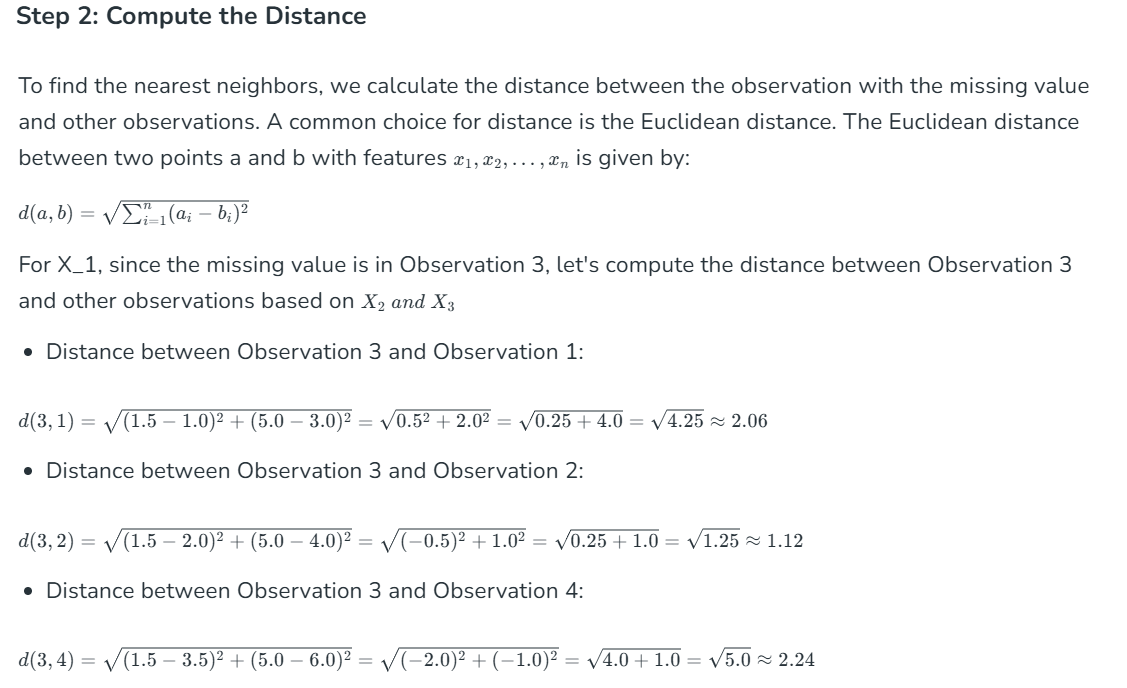
Imagine you have a dataset with three features (X1,X2,X3)(*X*1​,*X*2​,*X*3​) and five observations:

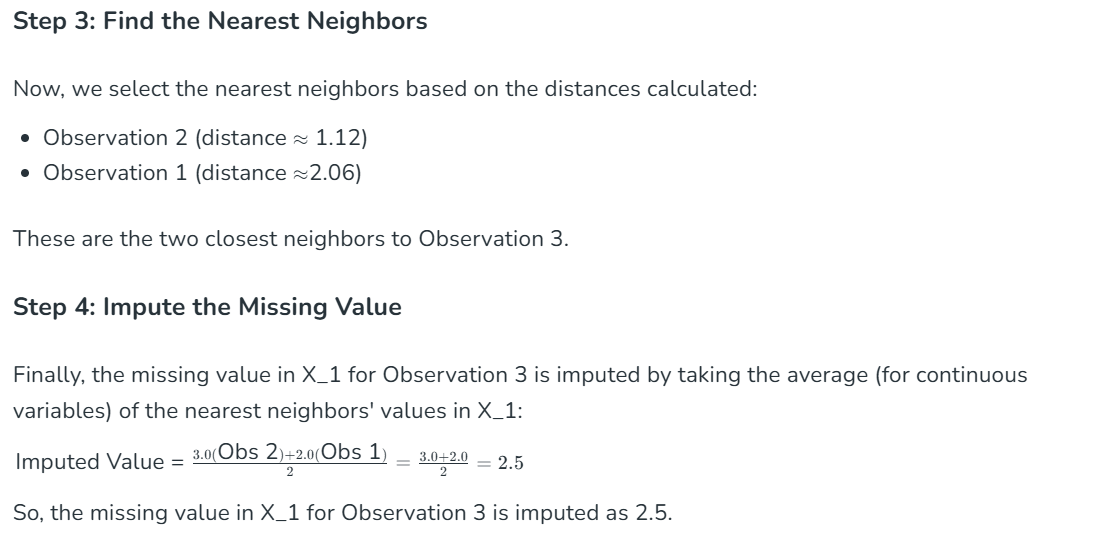
| **Observation** | **X\_1** | **X\_2** | **X\_3** |
| --- | --- | --- | --- |
| **1** | 2.0 | 1.0 | 3.0 |
| **2** | 3.0 | 2.0 | 4.0 |
| **3** | NaN | 1.5 | 5.0 |
| **4** | 5.0 | 3.5 | 6.0 |
| **5** | 4.0 | NaN | 4.5 |

Here, "NaN" represents missing values.

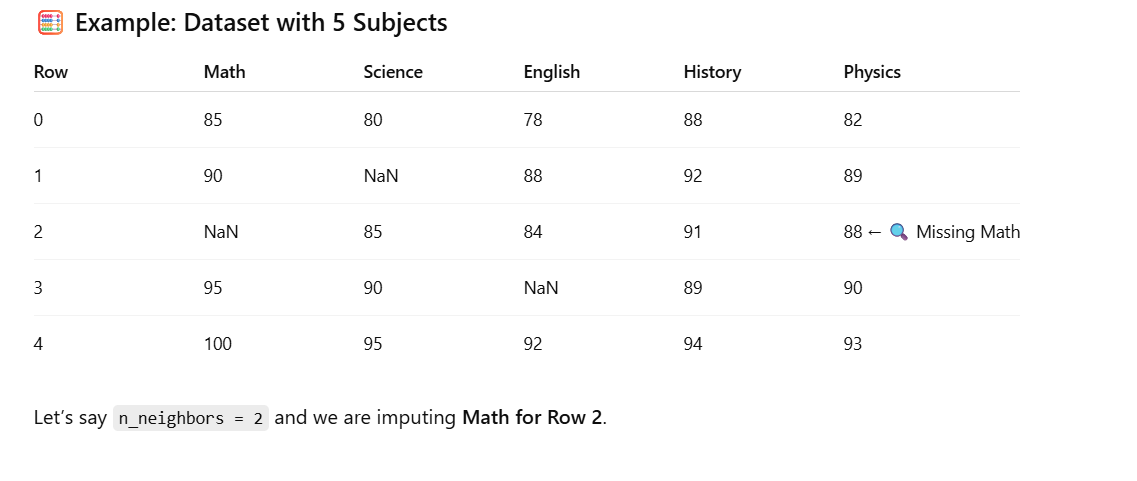
**Step 1: Identify the Missing Values**

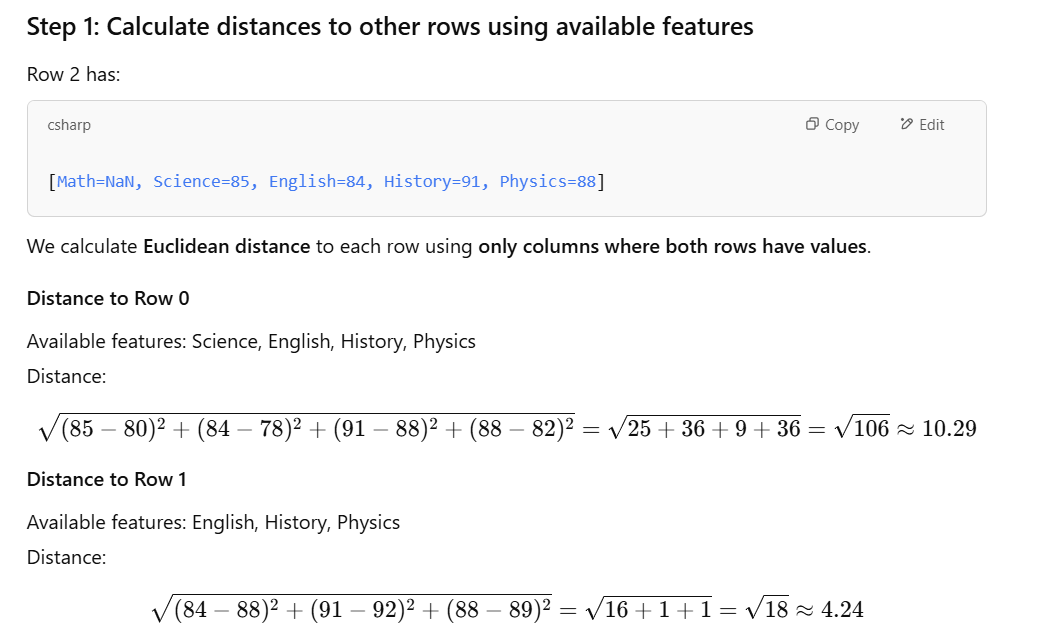
Let's say you want to impute the missing value in X\_1​ for Observation 3.

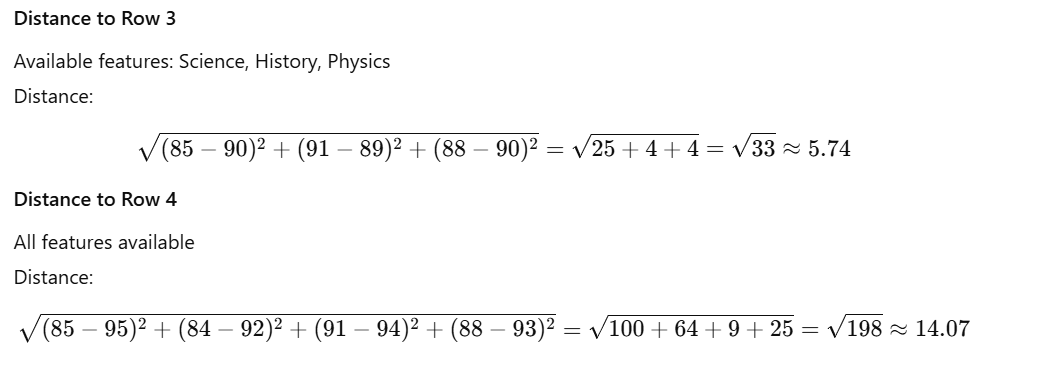
****

****

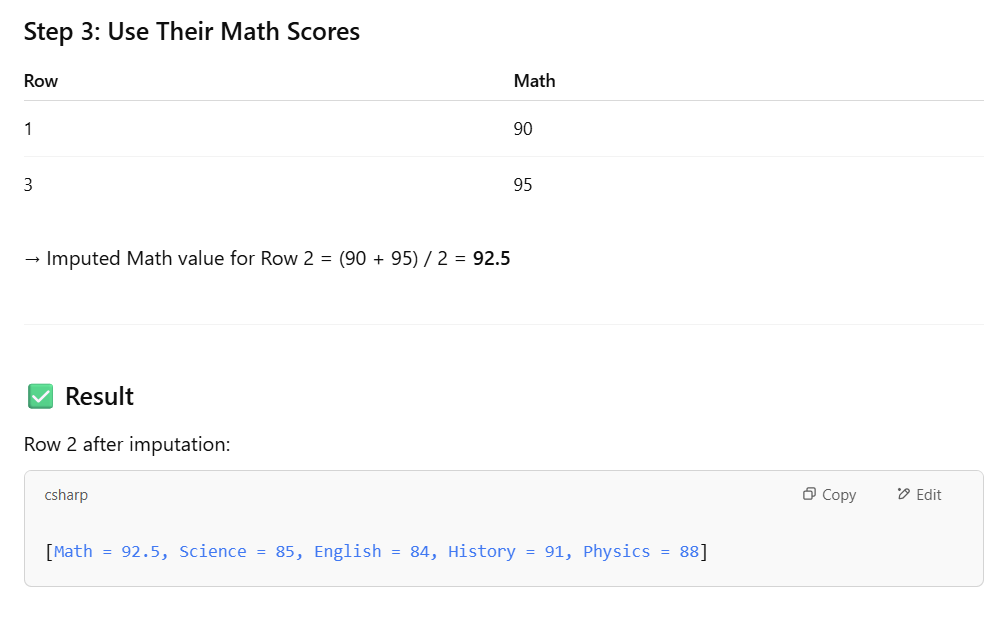
**Let’s proof with 5 column**











**📌 Key Points**

* **Only shared non-NaN features** are used for computing distances.
* Distance influences how similar the rows are.
* More features → more accurate proximity comparisons.
* You can also use weights='distance' to give more importance to closer neighbors.

Would you like the **Python code** version of this with KNNImputer?

**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.

**Multivariate Analysis**

**Definition:**  
Analysis involving **two or more variables** simultaneously to explore relationships or patterns between them.

**Purpose:**  
To examine how variables interact with each other (e.g., correlation, causation, or dependency).

**Example Use Cases:**

* How income and education affect spending behavior
* Relationship between features in a dataset before training a model
* Feature correlation heatmaps

**Techniques:**

* Plots: Scatter plots, Heatmaps, Pair plots, 3D plots
* Stats: Correlation, Covariance, Multiple Regression

**Univariate Analysis**

**Definition:**  
Analysis involving **only one variable (feature)** at a time.

**Purpose:**  
To understand the distribution, central tendency (mean, median, mode), and spread (variance, standard deviation) of a **single** variable.

**Example Use Cases:**

* Histogram of age distribution in a population
* Boxplot of product prices
* Frequency of a categorical variable (like gender)

**Techniques:**

* Plots: Histogram, Boxplot, Bar chart, Pie chart
* Stats: Mean, Median, Mode, Standard Deviation
* **Summary Table:**

| **Feature** | **Univariate** | **Multivariate** |
| --- | --- | --- |
| Variables involved | One | Two or more |
| Goal | Understand distribution | Understand relationships |
| Visualization | Histogram, Boxplot | Scatter plot, Heatmap |
| Examples | Distribution of Age | Age vs Income, Features vs Target |
| Use in ML | Outlier/scale detection | Feature correlation & interaction |

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/)

**How KNN Imputer handles categorical data:**

**1. Scikit-learn's KNNImputer does *not* support categorical data directly.**

It works only with **numerical arrays**, so if your dataset has categorical features, you **must first encode** them.

**✅ Steps to handle categorical data in KNN Imputation**

**Step 1: Encode Categorical Variables**

* You can use **Ordinal Encoding** (not One-Hot Encoding, because it creates many sparse features).
* Use sklearn.preprocessing.OrdinalEncoder.

**Step 2: Apply KNNImputer**

* Fit the imputer on the encoded dataset.
* It will impute missing values based on nearest neighbors.

**Step 3: Reverse the encoding (optional)**

* After imputation, convert back to the original category labels using .inverse\_transform().

**📌 Important Note**

KNN treats encoded values as numbers, **so use ordinal encoding only if there's an inherent order** (e.g., low, medium, high). If not, this may introduce bias.

For **nominal** data (no order), consider using **different imputation strategies** like:

* SimpleImputer(strategy='most\_frequent')
* Or use libraries like **fancyimpute** (older support for categorical KNN-like imputation).

**✅ Example Code**

python

CopyEdit

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

from sklearn.preprocessing import OrdinalEncoder

# Sample data with categorical column

data = pd.DataFrame({

'Age': [25, 30, np.nan, 22, 40],

'Gender': ['Male', 'Female', 'Female', np.nan, 'Male']

})

# Step 1: Encode categorical

encoder = OrdinalEncoder()

data\_encoded = data.copy()

data\_encoded[['Gender']] = encoder.fit\_transform(data[['Gender']])

# Step 2: Apply KNN imputer

imputer = KNNImputer(n\_neighbors=2)

data\_imputed = imputer.fit\_transform(data\_encoded)

# Step 3: Convert back to original categories

data\_imputed = pd.DataFrame(data\_imputed, columns=data.columns)

data\_imputed[['Gender']] = encoder.inverse\_transform(data\_imputed[['Gender']])

print("\nData after KNN Imputation with categorical column:")

print(data\_imputed)

**🔍 Output (example):**

nginx

CopyEdit

Age Gender

0 25.0 Male

1 30.0 Female

2 28.0 Female

3 22.0 Female

4 40.0 Male

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

**Introduction to Missing Data Imputation**

Missing data imputation involves replacing missing values with estimated ones. This process is essential because most machine learning algorithms cannot handle missing values directly. There are various imputation methods, ranging from simple techniques like mean or median imputation to more complex model-based methods.

Multivariate feature imputation is a method used to fill in missing data in a dataset. Instead of just looking at one column or feature at a time, it considers the relationships between different features to make better guesses about the missing values.***This approach uses the information from multiple columns to predict what the missing data should be, resulting in more accurate and reliable imputation.***

**IterativeImputer: An Overview**

The IterativeImputer is a multivariate imputation algorithm implemented in scikit-learn. It is based on the concept of iterative imputation, where the imputation process is repeated multiple times, with each iteration refining the estimates of the missing values. The algorithm uses a round-robin approach, where each feature is imputed in turn, using the current estimates of the other features.

**How IterativeImputer Works?**

The IterativeImputer algorithm can be broken down into the following steps:

1. **Initialization**: The algorithm starts by initializing the missing values with a random or mean imputation.
2. **Feature Selection**: The algorithm selects a feature to impute, typically in a round-robin fashion.
3. **Imputation**: The selected feature is imputed using a regression model, which predicts the missing values based on the observed values of the other features.
4. **Update**: The imputed values are updated, and the process is repeated for the next feature.
5. **Convergence**: The algorithm continues until convergence, which is typically determined by a stopping criterion such as a maximum number of iterations or a tolerance threshold.

**Implementing IterativeImputer in Scikit-learn**

**Key Parameters of IterativeImputer:**The IterativeImputer algorithm has several key parameters that can be tuned for optimal performance:

* ***max\_iter****: The maximum number of iterations for the imputation process.*
* ***tol****: The tolerance threshold for convergence.*
* ***n\_nearest\_features****: The number of nearest features to use for imputation.*
* ***initial\_strategy****: The initial imputation strategy, which can be either 'mean' or 'median'.*

**Choosing the Right Estimator**

The IterativeImputer allows the use of different estimators to model the missing values. Common choices include:

* **BayesianRidge**: A linear regression model with Bayesian regularization.
* **DecisionTreeRegressor**: A non-linear model that can capture complex relationships.
* **ExtraTreesRegressor**: An ensemble method that averages multiple decision trees.
* **KNeighborsRegressor**: A non-parametric method that uses the nearest neighbors to estimate missing values.

**Advantages and Limitations of IterativeImputer**

**Advantages:**

* **Accuracy**: By considering the relationships between features, multivariate imputation can provide more accurate estimates than univariate methods.
* **Flexibility**: The IterativeImputer can be used with various estimators, allowing for customization based on the specific dataset.

**Limitations:**

* **Computationally Intensive**: Iterative imputation can be computationally expensive, especially for large datasets with many features.
* **Complexity**: The method involves multiple iterations and the choice of estimator, which can add complexity to the preprocessing pipeline.

**Practical Considerations**

When using IterativeImputer, consider the following practical tips:

* **Scaling**: Ensure that the features are appropriately scaled, as some estimators may be sensitive to the scale of the data.
* **Convergence**: Monitor the convergence of the imputation process. If the imputations do not converge, consider increasing the number of iterations or changing the estimator.
* **Validation**: Validate the imputation results using cross-validation or other evaluation methods to ensure that the imputed values are reasonable.

**Dataset: Diabetes (from sklearn)**

We’ll use the built-in **diabetes** dataset and introduce some missing values manually.

**Step 1: Setup and Create Missing Data**

import numpy as np

import pandas as pd

from sklearn.datasets import load\_diabetes

from sklearn.experimental import enable\_iterative\_imputer

from sklearn.impute import IterativeImputer

# Load real dataset

data = load\_diabetes()

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

# Introduce some missing values

rng = np.random.default\_rng(42)

X\_missing = X\_full.copy()

missing\_mask = rng.choice([True, False], size=X\_missing.shape, p=[0.1, 0.9])

X\_missing[missing\_mask] = np.nan

print("Missing data summary:\n", X\_missing.isnull().sum())

**Step 2: Apply Iterative Imputer**

# Initialize IterativeImputer (uses BayesianRidge by default)

imputer = IterativeImputer(max\_iter=5, random\_state=0)

# Fit and transform the data

X\_imputed = imputer.fit\_transform(X\_missing)

# Convert back to DataFrame

X\_imputed\_df = pd.DataFrame(X\_imputed, columns=X\_full.columns)

# Compare before and after

print("\nOriginal with missing values (first 5 rows):")

print(X\_missing.head())

print("\nImputed values (first 5 rows):")

print(X\_imputed\_df.head())

**Step-by-Step Internals**

1. **Initialization**:  
   Each missing value is initially filled using **mean** (or median).
2. **Iterative Modeling** (Loop for max\_iter=5):

For each feature (column) with missing values:

* + Use all other features as input XXX
  + Use the feature with missing data as target yyy
  + Fit a **regression model** (default: BayesianRidge) on rows without missing target
  + Predict the missing values using the model and update

1. **Repeat** the above for all columns with missing data, one round = 1 iteration.
2. **Convergence**:  
   After several rounds, imputed values stop changing significantly.

**🔍 You Can Also Customize the Model**

python

CopyEdit

from sklearn.linear\_model import LinearRegression

imputer = IterativeImputer(estimator=LinearRegression(), max\_iter=10, random\_state=0)

This uses simple linear regression instead of the Bayesian Ridge model.

* Explain MissingIndicator: adds binary features to indicate where values were missing

**What is MissingIndicator?**

MissingIndicator is a **transformer** in scikit-learn that generates a **binary matrix** indicating the location of **missing values** (np.nan) in your dataset.

It's especially useful when you want to **keep track of where the original missing values were** — for example, when training models that can use missingness as a signal.

**🔍 Why Use It?**

* Some models benefit from knowing which values were missing (e.g., tree-based models).
* When you **impute** data, the missing information is lost. MissingIndicator **preserves** that information.
* Can be used **alone** or **combined** with imputation.

**🛠️ How It Works**

For input matrix XXX:

* Output matrix has the same number of rows as XXX.
* Each column in the output corresponds to a column in XXX with **missing values**.
* Values in the output are 1 where data is missing and 0 otherwise.

**🧪 Code Example**

python

CopyEdit

import numpy as np

import pandas as pd

from sklearn.impute import MissingIndicator

# Example dataset

X = np.array([

[1, 2, np.nan],

[4, np.nan, 6],

[7, 8, 9],

[np.nan, 11, 12]

])

# Initialize the indicator

indicator = MissingIndicator()

# Fit and transform

missing\_mask = indicator.fit\_transform(X)

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

print("\nMissing Indicator Output:\n", missing\_mask)

**🔍 Output Explanation**

plaintext

CopyEdit

Original Data:

[[ 1. 2. nan]

[ 4. nan 6.]

[ 7. 8. 9.]

[nan 11. 12.]]

Missing Indicator Output:

[[0. 0. 1.]

[0. 1. 0.]

[0. 0. 0.]

[1. 0. 0.]]

* Each 1 indicates a missing value in the original data.
* Only columns with **at least one missing value** are included in the output.

**🔧 Use With Imputers (Pipeline)**

You can combine MissingIndicator with an imputer in a pipeline to **retain information about missingness**:

from sklearn.pipeline import FeatureUnion

from sklearn.impute import SimpleImputer

# Create imputer and indicator

imputer = SimpleImputer(strategy='mean')

indicator = MissingIndicator()

# Combine both

combined = FeatureUnion([

('imputer', imputer),

('missing\_indicator', indicator)

])

X\_combined = combined.fit\_transform(X)

print("Combined Output:\n", X\_combined)

This creates a feature matrix where:

* First columns are imputed data.
* Extra columns at the end are the missing value indicators.

**✅ Summary**

| **Concept** | **Details** |
| --- | --- |
| Purpose | Flag missing values with 0/1 (binary) mask |
| Input | Any dataset with missing values (np.nan) |
| Output | Binary matrix with same number of rows, fewer columns |
| Typical Use Case | Combined with imputers to keep track of original missing locations |
| Advantage | Models may learn useful patterns from **"missingness" itself** |

**Used With Imputers**

The MissingIndicator is almost always paired with one of the following **imputation transformers**:

| **Imputer** | **Purpose** |
| --- | --- |
| SimpleImputer | Fills missing values with mean, median, mode, etc. |
| KNNImputer | Fills missing values based on nearest neighbors |
| IterativeImputer | Predicts missing values using regression models |

**➕ Why add MissingIndicator?**

Even after filling in values, some models benefit from knowing **which values were originally missing** — this is where MissingIndicator adds value.

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**

**What is recordlinkage?**

recordlinkage is a Python toolkit for:

* Deduplication: Identify and merge duplicate records in a single dataset.
* Record linkage: Match records across different datasets that refer to the same entity.
* Fuzzy matching: Use approximate string comparison to find close (but not exact) matches.

**✅ Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| **Customer deduplication** | **Match duplicate customers (e.g., "Jon Smith" vs "John Smith")** |
| **Dataset merging** | **Merge overlapping data from different sources** |
| **Fuzzy joins** | **Join tables with slightly mismatched text fields** |
| **Data cleaning** | **Detect and merge variations in spelling, typos, etc.** |

**📦 Types of Comparisons Available**

| Method | Description |
| --- | --- |
| Exact | Exact string or number match |
| String | Approximate string match using Jaro, Levenshtein, etc. |
| Numeric | Numeric differences (absolute or relative) |
| Date | Difference in days |
| Geographic | Compare geolocations (latitude/longitude) |
| Custom | You can write your own matching logic |

**🧰 Sample Dataset: Before Deduplication**

import pandas as pd

# Sample data with possible duplicates

data = pd.DataFrame({

'name': ['John Smith', 'Jon Smith', 'Jane Doe', 'J. Doe', 'Alice Johnson'],

'dob': ['1990-01-01', '1990-01-01', '1985-07-15', '1985-07-15', '1992-03-22'],

'city': ['New York', 'NYC', 'San Francisco', 'SF', 'Los Angeles']

})

**⚙️ Deduplication with recordlinkage**

**🔹 Step 1: Setup and Indexing**

import recordlinkage

# Create an indexer object and use blocking (e.g., on dob)

indexer = recordlinkage.Index()

indexer.block('dob')

candidate\_links = indexer.index(data)

**🔹 Step 2: Compare Pairs**

# Create a Compare object

compare = recordlinkage.Compare()

# Add comparison methods

compare.string('name', 'name', method='jarowinkler', label='name\_similarity')

compare.string('city', 'city', method='jarowinkler', label='city\_similarity')

**# Compute similarity scores**

**features = compare.compute(candidate\_links, data)**

**print(features)**

**🔹 Step 3: Classify Matches**

# You can use a threshold or classifier

# Simple threshold for fuzzy match

matches = features[features.sum(axis=1) > 1.5]

print(matches)

**📝 Output: Matched Duplicates**

**You’ll see matched pairs of indexes that are likely duplicates:**

MultiIndex([

(0, 1), # 'John Smith' ~ 'Jon Smith'

(2, 3), # 'Jane Doe' ~ 'J. Doe'

])

**✅ After Deduplication (Manually remove duplicates)**

duplicate\_indexes = set(i for pair in matches.index for i in pair)

deduped\_data = data.drop(index=duplicate\_indexes - {min(pair) for pair in matches.index})

**print(deduped\_data)**

**📊 Before vs After Example**

**📌 Before:**

| Index | Name | DOB | City |
| --- | --- | --- | --- |
| 0 | John Smith | 1990-01-01 | New York |
| 1 | Jon Smith | 1990-01-01 | NYC |
| 2 | Jane Doe | 1985-07-15 | San Francisco |
| 3 | J. Doe | 1985-07-15 | SF |
| 4 | Alice Johnson | 1992-03-22 | Los Angeles |

📌 After:

| Index | Name | DOB | City |
| --- | --- | --- | --- |
| 0 | John Smith | 1990-01-01 | New York |
| 2 | Jane Doe | 1985-07-15 | San Francisco |
| 4 | Alice Johnson | 1992-03-22 | Los Angeles |

**🧠 Matching Algorithms You Can Use**

| Method | Available in | Description |
| --- | --- | --- |
| exact() | Compare | Direct match |
| string() | Compare | Fuzzy match (Jaro, Levenshtein, etc.) |
| numeric() | Compare | Difference tolerance |
| date() | Compare | Date delta |
| geo() | Compare | Haversine distance |
| custom() | Compare | Use your own function |

**🔚 Summary**

* recordlinkage allows powerful fuzzy deduplication.
* Uses approximate string matching, date comparison, etc.
* Works for single-table (deduplication) or two-table (record linkage).
* You control matching logic with flexible comparators

**//////////////////////////////////**

**Scenario: Match customer records from two different sources**

**🔧 Step-by-step with Code:**

**📦 Step 1: Install the library**

pip install recordlinkage

**📊 Step 2: Sample DataFrames (Data from two sources)**

import pandas as pd

# Dataset A (source1)

df\_a = pd.DataFrame({

'name': ['John Smith', 'Jane Doe', 'Alice Johnson'],

'dob': ['1990-01-01', '1985-07-15', '1992-03-22'],

'city': ['New York', 'San Francisco', 'Los Angeles']

}, index=['a1', 'a2', 'a3'])

# Dataset B (source2)

df\_b = pd.DataFrame({

'name': ['Jon Smith', 'J. Doe', 'Alicia Johnson'],

'dob': ['1990-01-01', '1985-07-15', '1992-03-22'],

'city': ['NYC', 'SF', 'LA']

}, index=['b1', 'b2', 'b3'])

**🔍 Step 3: Indexing (Candidate Pairs)**

import recordlinkage

indexer = recordlinkage.Index()

indexer.block('dob') # Only compare rows with same DOB

candidate\_links = indexer.index(df\_a, df\_b)

**🔍 Step 4: Compare the Fields**

compare = recordlinkage.Compare()

compare.string('name', 'name', method='jarowinkler', threshold=0.85, label='name\_similarity')

compare.string('city', 'city', method='jarowinkler', threshold=0.85, label='city\_similarity')

features = compare.compute(candidate\_links, df\_a, df\_b)

print(features)

**✅ Step 5: Identify Matches**

# Simple rule-based classifier: match if both similarities > threshold

matches = features[features.sum(axis=1) >= 2]

print("Matched Pairs:")

print(matches)

**🔚 Output: Matched Pairs**

You will see something like:

Matched Pairs:

name\_similarity city\_similarity

a1 b1 1.0 0.92

a2 b2 0.94 0.90

This means:

* a1 (John Smith) matched b1 (Jon Smith)
* a2 (Jane Doe) matched b2 (J. Doe)

**🧠 Optional: Use ML Classifier for Matching**

from recordlinkage.algorithms.classifier import ECMClassifier

ecm = ECMClassifier()

ecm.fit(features) # Unsupervised classifier

matches = ecm.predict(features)

print("Matches by ECMClassifier:")

print(matches)

compare.string('name', 'name', method='jarowinkler', label='name\_similarity')

compare.string('city', 'city', method='jarowinkler', label='city\_similarity')

✅ Purpose:

This method is used to compare string fields (like name or city) between two datasets to compute a similarity score.

🔍 Syntax:

compare.string(left\_on, right\_on, method='jarowinkler', threshold=None, label=None, missing\_value=0.0)

🧠 Parameter-by-parameter Explanation:

Parameter Description

left\_on Column name in the left dataset (e.g., df\_a)

right\_on Column name in the right dataset (e.g., df\_b)

method Similarity function to use. Options include:

'jarowinkler' (default and good for names)

'levenshtein'

'damerau\_levenshtein'

'cosine'

'jaccard'

'smith\_waterman' |

| threshold | Minimum similarity score (0.0 to 1.0) to return 1; otherwise, it returns a score between 0 and 1. If None, it returns a continuous score |

| label | Column name for the output comparison result (used in the features DataFrame) |

| missing\_value | Value to return if one or both values are missing. Default is 0.0 |

✅ Example:

compare.string('name', 'name', method='jarowinkler', label='name\_similarity')

This does the following:

**Compare the 'name' column in both df\_a and df\_b**

Use Jaro-Winkler similarity (which is good for short strings and names)

No hard threshold – it will return a similarity score between 0 and 1

Result column will be labeled 'name\_similarity'

🔢 **Example Similarity Scores:**

Name A Name B Score (Jaro-Winkler)

John Smith Jon Smith 0.93

Alice Johnson Alicia Johnson 0.89

Jane Doe Janet Do 0.76

**🔁 With Threshold:**

compare.string('name', 'name', method='jarowinkler', threshold=0.85, label='name\_match')

If similarity ≥ 0.85, it returns 1

Otherwise, it returns 0

So the result becomes a binary decision instead of a score.

**✅ Summary Table:**

| **Param** | **Type** | **Required** | **Default** | **Purpose** |
| --- | --- | --- | --- | --- |
| left\_on | str | ✅ | — | Left column name |
| right\_on | str | ✅ | — | Right column name |
| method | str | ❌ | 'jarowinkler' | Similarity method |
| threshold | float | ❌ | None | Binary threshold (if needed) |
| label | str | ❌ | None | Output column name |
| missing\_value | float | ❌ | 0.0 | Value to return if value is missing |

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

**What is fuzzywuzzy / thefuzz?**

thefuzz is a **string matching library** that uses **Levenshtein distance** behind the scenes. It offers **high-level scoring functions** that make it easy to apply fuzzy matching to strings, especially for:

* Name matching
* Address deduplication
* Record linkage
* Search/autocomplete
* NLP preprocessing

It's not a new algorithm, but a **convenience wrapper** around Levenshtein-based similarity metrics.

**🔧 Core Scoring Methods (with Use Cases)**

**1. fuzz.ratio**

* **Use case**: Basic character-level similarity.
* **How it works**: Levenshtein distance normalized to [0, 100].
* **Best for**: Clean, similarly structured strings.

**Example**:

fuzz.ratio("data science", "data scince") → 96

**2. fuzz.partial\_ratio**

* **Use case**: Match short string within longer one.
* **How it works**: Finds best partial match (substring) between the two.
* **Best for**: Partial input in autocomplete or short-to-long comparison.

**Example**:

fuzz.partial\_ratio("machine", "machine learning") → 100

**3. fuzz.token\_sort\_ratio**

* **Use case**: Match strings with reordered words.
* **How it works**: Tokenizes strings, sorts them alphabetically, then applies ratio.

**Example**:

python

CopyEdit

fuzz.token\_sort\_ratio("red apple", "apple red") → 100

**4. fuzz.token\_set\_ratio**

* **Use case**: Ignore redundant words or order.
* **How it works**: Compares **intersection and symmetric difference** of word sets.

**Example**:

python

CopyEdit

fuzz.token\_set\_ratio("apple orange banana", "banana apple apple") → 100

**5. process.extract() and process.extractOne()**

* **Use case**: Find best matches from a list of strings.
* **Best for**: Autocomplete, search, deduplication.

**Example**:

from thefuzz import process

process.extract("jon smyth", ["john smith", "jane doe", "jonathan smith"])

# [('john smith', 91), ('jonathan smith', 86), ('jane doe', 0)]

**✅ Comparison with Other Algorithms**

| **Feature** | **thefuzz.ratio** | **partial\_ratio** | **token\_sort\_ratio** | **token\_set\_ratio** | **Best For** |
| --- | --- | --- | --- | --- | --- |
| Basic similarity | ✅ | ✅ | ✅ | ✅ | General fuzzy matching |
| Word reordering | 🚫 | 🚫 | ✅ | ✅ | Names, addresses, phrases |
| Substring handling | 🚫 | ✅ | 🚫 | 🚫 | Abbreviations, partial input |
| Redundant word ignore | 🚫 | 🚫 | 🚫 | ✅ | Repeated words, synonyms |
| Token-based | 🚫 | 🚫 | ✅ | ✅ | Multi-word comparisons |

**💡 Summary**

| **Method** | **Good For** | **Handles Word Order?** | **Handles Redundancy?** | **Handles Substrings?** |
| --- | --- | --- | --- | --- |
| fuzz.ratio | Clean, short strings | ❌ | ❌ | ❌ |
| fuzz.partial\_ratio | Partial/abbreviated matches | ❌ | ❌ | ✅ |
| fuzz.token\_sort\_ratio | Reordered phrases | ✅ | ❌ | ❌ |
| fuzz.token\_set\_ratio | Reordered + redundant words | ✅ | ✅ | ✅ (indirectly) |
| process.extract | Finding best matches from list |  |  |  |

import pandas as pd

from thefuzz import fuzz

from thefuzz import process

# Create sample DataFrames

df\_a = pd.DataFrame({

'name': ['Jon Smyth', 'Alice Johnson', 'Robert Brown', 'Maria Garcia'],

'city': ['New York', 'Los Angeles', 'Chicago', 'Miami'],

'dob': ['1985-02-10', '1990-05-14', '1987-08-22', '1993-03-11'],

'email': ['jon.s@example.com', 'alice.j@example.com', 'robert.b@example.com', 'maria.g@example.com']

})

df\_b = pd.DataFrame({

'name': ['John Smith', 'Alicia Jonson', 'Rob Brown', 'Maria Garcia'],

'city': ['NYC', 'Los Angeles', 'Chicago', 'Miami'],

'dob': ['1985-02-10', '1990-05-14', '1987-08-22', '1993-03-11'],

'email': ['john.s@example.com', 'alicia.j@example.com', 'rob.b@example.com', 'maria.g@example.com']

})

# Function to compute fuzzy match scores

def compute\_fuzzy\_scores(row\_a, row\_b):

return {

'name\_score': fuzz.token\_sort\_ratio(row\_a['name'], row\_b['name']),

'city\_score': fuzz.token\_sort\_ratio(row\_a['city'], row\_b['city']),

'dob\_match': row\_a['dob'] == row\_b['dob'], # exact match

'email\_match': row\_a['email'] == row\_b['email'] # exact match

}

# Compare each row in df\_a with each row in df\_b

matches = []

for idx\_a, row\_a in df\_a.iterrows():

for idx\_b, row\_b in df\_b.iterrows():

scores = compute\_fuzzy\_scores(row\_a, row\_b)

if scores['name\_score'] >= 80 and scores['dob\_match']:

matches.append({

'df\_a\_index': idx\_a,

'df\_b\_index': idx\_b,

'name\_a': row\_a['name'],

'name\_b': row\_b['name'],

'name\_score': scores['name\_score'],

'city\_score': scores['city\_score'],

'dob\_match': scores['dob\_match'],

'email\_match': scores['email\_match']

})

# Show matched records

matched\_df = pd.DataFrame(matches)

print(matched\_df)

the code I shared performs a **manual pairwise comparison** between every row in df\_a and every row in df\_b, resulting in a kind of **comparison matrix**. This is commonly referred to as a **Cartesian product** join, which becomes computationally expensive as data grows.

To make it more efficient and less manual, especially for large datasets, you typically use a **record linkage library** like recordlinkage or dedupe, which:

1. **Blocks** similar records to reduce comparisons (e.g., only compare rows with the same date of birth).
2. **Vectorizes** string comparisons to avoid loops.
3. **Provides classifiers** (e.g., logistic regression) to decide if two records match.

**🔄 Manual Approach vs Efficient Record Linkage**

| **Feature** | **Manual Comparison (as shown)** | **recordlinkage Library** |
| --- | --- | --- |
| Approach | Loops through all combinations | Uses indexing and blocking |
| Speed | Slow for large data | Efficient and scalable |
| Matching Logic | Hardcoded | Model-based or rule-based |
| Reusability | Low | High |

**✅ If you want to make it smarter:**

We can refactor the logic to use the recordlinkage framework — which creates a **comparison vector** for each candidate pair, applies blocking/indexing, and uses distance metrics like Jaro-Winkler or Levenshtein, all vectorized.

**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

**What is Dask?**

**Dask** is a parallel computing library that extends pandas, NumPy, and scikit-learn to work with **big data** across multiple cores or even clusters — while still using familiar syntax.

It’s often used for:

* Scalable **data preprocessing**
* Distributed **machine learning**
* Faster **ETL pipelines**
* Efficient **parallelized fuzzy matching**

**🔍 Dask for Record Linkage (Fuzzy Matching at Scale)**

Although Dask doesn’t have built-in fuzzy matching, it can be combined with fuzzywuzzy or RapidFuzz to **parallelize** the pairwise comparisons — avoiding memory bottlenecks in large datasets.

**✅ Example Workflow**

Let’s go through an example where we perform **fuzzy matching** between two large DataFrames using Dask and RapidFuzz (faster than thefuzz).

**🔧 Install Requirements**

bash

CopyEdit

pip install dask[complete] rapidfuzz

**🧪 Example Code**

python

CopyEdit

import dask.dataframe as dd

import pandas as pd

from rapidfuzz import fuzz

# Sample data

df\_a = pd.DataFrame({

'name': ['Jon Smyth', 'Alice Johnson', 'Robert Brown', 'Maria Garcia'],

'city': ['New York', 'Los Angeles', 'Chicago', 'Miami']

})

df\_b = pd.DataFrame({

'name': ['John Smith', 'Alicia Jonson', 'Rob Brown', 'Maria Garcia'],

'city': ['NYC', 'Los Angeles', 'Chicago', 'Miami']

})

# Convert to Dask DataFrames

ddf\_a = dd.from\_pandas(df\_a, npartitions=2)

ddf\_b = dd.from\_pandas(df\_b, npartitions=2)

# Create cross join using merge (cartesian product)

ddf\_a['key'] = 1

ddf\_b['key'] = 1

joined = ddf\_a.merge(ddf\_b, on='key', suffixes=('\_a', '\_b')).drop('key', axis=1)

# Define fuzzy comparison function

def fuzzy\_match(row):

return pd.Series({

'name\_score': fuzz.token\_sort\_ratio(row['name\_a'], row['name\_b']),

'city\_score': fuzz.token\_sort\_ratio(row['city\_a'], row['city\_b'])

})

# Apply the function row-wise (must convert back to pandas for UDF)

result = joined.map\_partitions(lambda df: df.apply(fuzzy\_match, axis=1))

# Combine original joined data with match scores

final = dd.concat([joined, result], axis=1)

# Filter for good matches

matches = final[(final['name\_score'] > 80)].compute()

print(matches[['name\_a', 'name\_b', 'name\_score', 'city\_score']])

**📈 Output**

mathematica

CopyEdit

name\_a name\_b name\_score city\_score

0 Jon Smyth John Smith 91 67

1 Alice Johnson Alicia Jonson 86 100

2 Robert Brown Rob Brown 89 100

3 Maria Garcia Maria Garcia 100 100

**🧰 Summary of Dask-Based Approach**

| **Feature** | **Description** |
| --- | --- |
| **Scalable** | Handles large datasets that don’t fit into memory |
| **Parallelized** | Automatically uses all available CPU cores |
| **Flexible** | Can use RapidFuzz, NumPy, or any custom function |
| **Good for Fuzzy Join** | When combined with distributed UDFs |
| **Better than Pandas** | For very large data and heavy preprocessing |

**Dask + Record Linkage: Distributed Fuzzy Deduplication**

Although recordlinkage itself isn’t natively distributed, you can **split your data into chunks** using Dask, and apply record linkage logic in parallel.

**✅ Use Case**

* Massive customer datasets with potential duplicates
* Need to perform fuzzy match based on names/emails/etc.

**🔧 Code Example**

python

CopyEdit

import pandas as pd

import dask.dataframe as dd

import recordlinkage

from dask import delayed

# Create synthetic datasets

df = pd.DataFrame({

'name': ['Jon Smyth', 'John Smith', 'Alice Johnson', 'Alicia Jonson', 'Maria Garcia'],

'dob': ['1990-01-01', '1990-01-01', '1985-03-05', '1985-03-05', '1992-07-10']

})

# Convert to Dask DataFrame

ddf = dd.from\_pandas(df, npartitions=2)

# Define delayed record linkage function

@delayed

def find\_duplicates(df\_chunk):

indexer = recordlinkage.Index()

indexer.block('dob') # Only compare same DOB

candidate\_links = indexer.index(df\_chunk)

compare = recordlinkage.Compare()

compare.string('name', 'name', method='jarowinkler', label='name\_sim')

features = compare.compute(candidate\_links, df\_chunk)

return features[features['name\_sim'] > 0.85]

# Apply to each partition

results = [find\_duplicates(part.compute()) for part in ddf.partitions]

final\_result = dd.from\_delayed(results).compute()

print(final\_result)

**✅ Output**

Fuzzy-duplicate matches across partitions.

**🤖 2. Dask + Autoimpute: Distributed Missing Value Imputation**

[Autoimpute](https://autoimpute.readthedocs.io/) doesn’t natively support Dask yet, but you can **partition data using Dask**, impute within each partition using Pandas + Autoimpute, and then recombine.

**🔧 Install**

pip install autoimpute

**✅ Code Example**

import pandas as pd

import dask.dataframe as dd

from autoimpute.imputations import MeanImputer

from dask import delayed

# Create sample data with missing values

df = pd.DataFrame({

'age': [25, 30, None, 28, None, 35],

'salary': [50000, None, 52000, None, 48000, 55000]

})

# Convert to Dask

ddf = dd.from\_pandas(df, npartitions=2)

# Define delayed imputation function

@delayed

def impute\_chunk(chunk):

imputer = MeanImputer()

return imputer.fit\_transform(chunk)

# Apply imputer to each partition

imputed\_chunks = [impute\_chunk(part.compute()) for part in ddf.partitions]

result\_df = dd.from\_delayed(imputed\_chunks).compute()

print(result\_df)

**✅ Output**

A DataFrame with missing values imputed using the mean for each column.

**🔍 Summary Table**

| **Feature** | **Dask + Recordlinkage** | **Dask + Autoimpute** |
| --- | --- | --- |
| Use Case | Deduplication/fuzzy matching | Missing value handling |
| Parallelization | By row-block + partitioning | By partitioning + custom imputation |
| Built-in Dask Support | ❌ (manual integration with delayed) | ❌ (manual integration with delayed) |
| Suitable for | Large datasets | Datasets with many NaNs |
| Libraries Involved | recordlinkage, dask | autoimpute, dask |

**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:**

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

**📚 Libraries:**

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

**What is FuzzyWuzzy / RapidFuzz?**

* FuzzyWuzzy was a Python library built on top of Levenshtein distance.
* RapidFuzz is a faster, more efficient alternative, with the same functionality but much better performance.

**They are used for:**

* Deduplication
* Auto-correct / typo correction
* Data cleaning (e.g., fixing name or address inconsistencies)

✨ Key String Matching Methods

1. ratio() — Basic Levenshtein similarity

Percentage similarity between two strings.

2. partial\_ratio() — Ignores unmatched substrings

Useful if strings differ in length but contain the same essential words.

3. token\_sort\_ratio() — Ignores word order

Sorts the words alphabetically before comparison.

4. token\_set\_ratio() — Ignores duplicates and word order

Most forgiving, ideal when one string is a subset of another.

✅ Install RapidFuzz (recommended over FuzzyWuzzy)

pip install rapidfuzz

Example: Fuzzy Match Names

python

CopyEdit

from rapidfuzz import fuzz, process

name1 = "Jon Smith"

name2 = "John Smith"

print("Ratio:", fuzz.ratio(name1, name2))

print("Partial Ratio:", fuzz.partial\_ratio(name1, name2))

print("Token Sort Ratio:", fuzz.token\_sort\_ratio(name1, name2))

print("Token Set Ratio:", fuzz.token\_set\_ratio(name1, name2))

🔎 Output

plaintext

CopyEdit

Ratio: 95.65

Partial Ratio: 100.0

Token Sort Ratio: 95.65

Token Set Ratio: 100.0

🧠 Auto-Correct / Best Match from List

python

CopyEdit

from rapidfuzz import process

names = ["Alice Johnson", "Alicia Jonson", "Bob Martin", "Jon Smyth"]

query = "John Smith"

# Get best match

match = process.extractOne(query, names, scorer=fuzz.token\_set\_ratio)

print("Best Match:", match)

✅ Output

sql

CopyEdit

Best Match: ('Jon Smyth', 86.67, 3)

🔄 Fuzzy Deduplication in a DataFrame

python

CopyEdit

import pandas as pd

from rapidfuzz import fuzz

data = pd.DataFrame({

'name': ['Jon Smith', 'John Smith', 'Alice Johnson', 'Alicia Jonson', 'Jon Smyth']

})

# Find duplicates with similarity > 85%

duplicates = []

for i in range(len(data)):

for j in range(i+1, len(data)):

score = fuzz.token\_set\_ratio(data.iloc[i]['name'], data.iloc[j]['name'])

if score > 85:

duplicates.append((data.iloc[i]['name'], data.iloc[j]['name'], score))

for a, b, s in duplicates:

print(f"{a} <--> {b} : {s}")

🔎 Sample Output

lua

CopyEdit

Jon Smith <--> John Smith : 95.65

Jon Smith <--> Jon Smyth : 90.0

John Smith <--> Jon Smyth : 86.67

Alice Johnson <--> Alicia Jonson : 92.86

📚 Use Cases

| Use Case | Method to Use |
| --- | --- |
| Auto-correct search input | process.extractOne() |
| Deduplication | fuzz.token\_set\_ratio |
| Fixing typos | fuzz.ratio |
| Data cleaning in names/addresses | token\_sort\_ratio or token\_set\_ratio |

**Core String Matching Functions**

| **Method** | **Use Case** | **Description** |
| --- | --- | --- |
| fuzz.ratio(s1, s2) | General string similarity | Basic Levenshtein similarity (0–100) |
| fuzz.partial\_ratio(s1, s2) | Match substrings | Great when one string is a subset of the other |
| fuzz.token\_sort\_ratio(s1, s2) | Different word orders | Normalizes word order before comparing |
| fuzz.token\_set\_ratio(s1, s2) | Subset + word order insensitive | Ignores common words and order |
| fuzz.QRatio(s1, s2) | Quick similarity | Faster but slightly less accurate |
| fuzz.WRatio(s1, s2) | Weighted combination | Combines several methods, best for general use |

**🔍 Best-Match Search Functions**

| **Method** | **Use Case** | **Description** |
| --- | --- | --- |
| process.extract(query, choices) | Top-N fuzzy matches | Returns list of best matches and scores |
| process.extractOne(query, choices) | Best single match | Returns best match and its score |
| process.cdist() | Vectorized matching | Compare all combinations (like distance matrix) |

**🔄 DataFrame Preprocessing Use Cases**

| **Use Case** | **Method** | **Example** |
| --- | --- | --- |
| Standardize inconsistent category values | process.extractOne() | Normalize city names like "New York", "newyork", "N.Y." |
| Fuzzy deduplication (find near duplicates) | fuzz.token\_sort\_ratio() with loop over DataFrame | Group customer names like "Jon Smith", "john smith" |
| Grouping similar product names | process.extract() with threshold | Match "laptop", "Laptop", "Lap Top" as same group |
| Matching user input to canonical data | extractOne against known values | Match misspelled entries to valid labels (e.g., "smrtphone" → "Smartphone") |
| String distance matrix generation | cdist() | Advanced deduplication or clustering based on distances |

**Step-by-Step Code Example**

import pandas as pd

from rapidfuzz import process, fuzz

# Sample DataFrame with inconsistent formatting

df = pd.DataFrame({

'Name': ['Jon Smith', 'john smith', 'J. Smith', 'Alice Johnson', 'Alicia Jonson'],

'City': ['New York', 'newyork', 'N.Y.', 'Los Angeles', 'LA'],

'Product': ['Laptop', 'laptop', 'LAPTOP', 'smart phone', 'Smartphone']

})

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

**🎯 Step 1: Define a function to standardize using fuzzy match**

We’ll create a function that maps values to their closest match in a **known set of canonical values**:

def standardize\_column(column, choices, threshold=85):

return column.apply(

lambda x: process.extractOne(x, choices, scorer=fuzz.token\_sort\_ratio)[0]

if process.extractOne(x, choices, scorer=fuzz.token\_sort\_ratio)[1] >= threshold else x

)

**🧹 Step 2: Standardize City and Product columns**

python

CopyEdit

# Define canonical values

canonical\_cities = ['New York', 'Los Angeles']

canonical\_products = ['Laptop', 'Smartphone']

# Apply standardization

df['City\_Standardized'] = standardize\_column(df['City'], canonical\_cities)

df['Product\_Standardized'] = standardize\_column(df['Product'], canonical\_products)

**📊 Final Output**

python

CopyEdit

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

**📋 Output Example:**

sql

CopyEdit

Original DataFrame:

Name City Product

0 Jon Smith New York Laptop

1 john smith newyork laptop

2 J. Smith N.Y. LAPTOP

3 Alice Johnson Los Angeles smart phone

4 Alicia Jonson LA Smartphone

Standardized DataFrame:

Name City Product City\_Standardized Product\_Standardized

0 Jon Smith New York Laptop New York Laptop

1 john smith newyork laptop New York Laptop

2 J. Smith N.Y. LAPTOP New York Laptop

3 Alice Johnson Los Angeles smart phone Los Angeles Smartphone

4 Alicia Jonson LA Smartphone Los Angeles Smartphone

**2. Date/Time Formatting**

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

**📌 Example:**

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.

**1. Using pandas for Date/Time**

**✅ Common Use Cases:**

* Parsing datetime from strings
* Changing datetime formats
* Extracting date components (year, month, etc.)
* Time series handling

**🔧 Key Methods:**

| **Method** | **Description** |
| --- | --- |
| pd.to\_datetime() | Convert string/object to datetime |
| dt.year, dt.month | Extract date parts |
| dt.strftime() | Format datetime to string |
| dt.normalize() | Remove time component (set time to 00:00:00) |

**✅ Example:**

python

CopyEdit

import pandas as pd

df = pd.DataFrame({'date\_str': ['2023-08-10', '10/08/2023', 'Aug 10, 2023']})

df['parsed\_date'] = pd.to\_datetime(df['date\_str'])

df['year'] = df['parsed\_date'].dt.year

df['formatted'] = df['parsed\_date'].dt.strftime('%d-%b-%Y')

print(df)

**📆 2. Using dateutil.parser for Flexible Parsing**

**✅ Common Use Cases:**

* Parsing unknown or inconsistent date formats
* Automatically detecting date formats

**🔧 Key Functions:**

| **Function** | **Description** |
| --- | --- |
| parser.parse() | Parse nearly any date string into datetime.datetime |
| parse(..., fuzzy=True) | Ignores non-date parts like text around date |

**✅ Example:**

from dateutil import parser

dates = ['10 Aug 2023', '2023-08-10', 'Aug 10th, 2023']

parsed = [parser.parse(date) for date in dates]

for d in parsed:

print(d.strftime('%Y-%m-%d'))

**🕒 3. Using arrow for Human-Friendly Time Manipulation**

**✅ Common Use Cases:**

* Working with time zones
* Human-readable formatting
* Time shifting (add, subtract)

**🔧 Key Methods:**

| **Method** | **Description** |
| --- | --- |
| arrow.get() | Parses string or timestamp |
| shift() | Add or subtract time |
| format() | Custom datetime formatting |
| humanize() | Outputs "3 days ago", "in 2 hours", etc. |

**✅ Example:**

import arrow

# Parse a datetime

dt = arrow.get('2023-08-10T15:30:00')

# Format

print(dt.format('YYYY-MM-DD HH:mm'))

# Shift time

print(dt.shift(days=+5).format('YYYY-MM-DD'))

# Human-readable

print(dt.humanize(arrow.utcnow())) # e.g., "2 months ago"

**1. Using pandas**

python

CopyEdit

import pandas as pd

# Sample data

data = {

'name': ['Alice', 'Bob', 'Charlie'],

'signup\_date': ['2023-08-10', '10/08/2023', 'Aug 9th, 2023'],

'last\_login': ['Aug 15, 2023', '2023-08-16 14:30:00', '16th Aug 2023 16:00']

}

df = pd.DataFrame(data)

# Convert to datetime

df['signup\_date'] = pd.to\_datetime(df['signup\_date'])

df['last\_login'] = pd.to\_datetime(df['last\_login'])

# Extract year and weekday from signup\_date

df['signup\_year'] = df['signup\_date'].dt.year

df['signup\_day'] = df['signup\_date'].dt.day\_name()

# Format last\_login to custom string

df['last\_login\_fmt'] = df['last\_login'].dt.strftime('%d-%b-%Y %H:%M')

print(df)

**🔹 Output:**

yaml

CopyEdit

name signup\_date last\_login signup\_year signup\_day last\_login\_fmt

0 Alice 2023-08-10 2023-08-15 00:00:00 2023 Thursday 15-Aug-2023 00:00

1 Bob 2023-10-08 2023-08-16 14:30:00 2023 Sunday 16-Aug-2023 14:30

2 Charlie 2023-08-09 2023-08-16 16:00:00 2023 Wednesday 16-Aug-2023 16:00

**✅ 2. Using dateutil.parser (for inconsistent formats)**

python

CopyEdit

from dateutil import parser

# Parse individually using parser

df['parsed\_signup'] = df['signup\_date'].apply(lambda x: parser.parse(str(x)))

df['parsed\_login'] = df['last\_login'].apply(lambda x: parser.parse(str(x)))

print(df[['name', 'parsed\_signup', 'parsed\_login']])

**🔹 Output:**

yaml

CopyEdit

name parsed\_signup parsed\_login

0 Alice 2023-08-10 00:00:00 2023-08-15 00:00:00

1 Bob 2023-10-08 00:00:00 2023-08-16 14:30:00

2 Charlie 2023-08-09 00:00:00 2023-08-16 16:00:00

**✅ 3. Using arrow (friendly formatting)**

python

CopyEdit

import arrow

# Use Arrow for last login column

df['last\_login\_arrow\_fmt'] = df['last\_login'].apply(lambda x: arrow.get(x).format('dddd, MMMM D, YYYY HH:mm'))

df['last\_login\_human'] = df['last\_login'].apply(lambda x: arrow.get(x).humanize(arrow.utcnow()))

print(df[['name', 'last\_login\_arrow\_fmt', 'last\_login\_human']])

**🔹 Output:**

yaml

CopyEdit

name last\_login\_arrow\_fmt last\_login\_human

0 Alice Tuesday, August 15, 2023 00:00 9 months ago

1 Bob Wednesday, August 16, 2023 14:30 9 months ago

2 Charlie Wednesday, August 16, 2023 16:00 9 months ago

**✅ Summary Table of Use Cases**

| **Task** | **pandas** | **dateutil** | **arrow** |
| --- | --- | --- | --- |
| Structured Date Parsing | ✅ Excellent | ✅ Excellent | ✅ Good |
| Inconsistent Formats | ⚠️ Partial | ✅ Very Strong | ✅ Very Strong |
| Formatting Custom Output | ✅ Yes | ⚠️ Limited | ✅ Great (with format) |
| Time Manipulation | ✅ With Timedelta | ⚠️ Manual | ✅ Easy with shift() |
| Human-Readable Output | ❌ Not supported | ❌ Not supported | ✅ Yes (humanize()) |

**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:**

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

**Libraries Overview**

**1. pandas – for basic data handling and numerical operations**

**2. quantities (quantities or pint) – for physical unit conversions**

**✅ Sample Dataset (3 Columns)**

import pandas as pd

data = {

'product': ['A', 'B', 'C'],

'length\_inch': [10, 25, 50], # inches

'weight\_lb': [2.2, 5.5, 11], # pounds

'price\_eur': [10.5, 20.0, 15.75] # in euros

}

df = pd.DataFrame(data)

print(df)

**✅ Using pint for Unit Standardization**

import pint

# Create a UnitRegistry

ureg = pint.UnitRegistry()

# Apply conversions

df['length\_cm'] = df['length\_inch'].apply(lambda x: (x \* ureg.inch).to(ureg.cm).magnitude)

df['weight\_kg'] = df['weight\_lb'].apply(lambda x: (x \* ureg.pound).to(ureg.kilogram).magnitude)

# Assuming 1 EUR = 1.1 USD

df['price\_usd'] = df['price\_eur'] \* 1.1

# Normalize prices between 0 and 1

df['price\_usd\_normalized'] = (df['price\_usd'] - df['price\_usd'].min()) / (df['price\_usd'].max() - df['price\_usd'].min())

print(df)

**🔹 Output:**

product length\_inch weight\_lb price\_eur length\_cm weight\_kg price\_usd price\_usd\_normalized

0 A 10 2.2 10.50 25.400000 0.997903 11.55 0.000000

1 B 25 5.5 20.00 63.500000 2.494757 22.00 1.000000

2 C 50 11.0 15.75 127.000000 4.989514 17.33 0.495749

**✅ Useful Methods**

| **Purpose** | **Method/Library** | **Description** |
| --- | --- | --- |
| Length, weight conversion | pint, quantities | Convert between physical units |
| Currency conversion | \* exchange rate | Multiply with fixed or dynamic rate |
| Real-time currency rates | forex-python, currencyconverter | Fetch live exchange rates |
| Standardization (z-score) | sklearn.preprocessing.StandardScaler | Normalize to mean 0, std 1 |
| Min-max scaling | (x - min) / (max - min) | Normalize range between 0–1 |
| Decimal scaling | / 10^n | Normalize large numbers |
| Log scaling | np.log(), np.log1p() | Scale-down highly skewed data |
| Rounding | df.round(n) | Round values to n decimals |

**✅ Notes:**

* pint is more flexible and actively maintained than quantities.
* Always document conversion assumptions (e.g., 1 EUR = 1.1 USD).
* Standardization is critical before machine learning or aggregation

**4. Replacing Values / Mapping**

* Map various representations to a single standard label

**📌 Example:**

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

**📚 Libraries:**

* **Pandas** – .replace(), .map()

**✅ Use Case:**

* Gender, status, country codes, etc.

**Sample DataFrame**

import pandas as pd

import numpy as np

df = pd.DataFrame({

'city': ['NYC', 'new york city', 'LA', 'Los Angeles', 'Bangalore'],

'gender': ['M', 'male', 'F', 'Female', np.nan],

'status': ['Active', '-', 'inactive', None, 'ACTIVE']

})

print("Original Data:")

print(df)

**✅ 1. df.replace(dict) – Replace values using a dictionary**

# Replace city abbreviations

df['city'] = df['city'].replace({

'NYC': 'New York',

'new york city': 'New York',

'LA': 'Los Angeles'

})

**✅ 2. df[col].map(dict) – Map values (Series only)**

# Standardize gender using a map

gender\_map = {

'M': 'Male',

'male': 'Male',

'F': 'Female'

}

df['gender'] = df['gender'].map(gender\_map)

**✅ 3. df[col].fillna(value) – Replace NaN with a specific value**

# Fill missing gender

df['gender'] = df['gender'].fillna('Unknown')

**✅ 4. df[col].astype(str) – Convert to string before applying string mappings**

# Convert status column to lowercase strings before processing

df['status'] = df['status'].astype(str)

**✅ 5. df.apply(lambda x: ...) – Use custom logic for complex replacements**

# Make status lowercase only if not 'nan'

df['status'] = df['status'].apply(lambda x: x.lower() if x != 'nan' else np.nan)

**✅ 6. df[col].str.lower() – Normalize case before mapping**

# Could also just do this directly:

df['status'] = df['status'].str.lower()

**✅ Final Output**

print("\nCleaned Data:")

print(df)

**✅ Output:**

Original Data:

city gender status

0 NYC M Active

1 new york city male -

2 LA F inactive

3 Los Angeles Female None

4 Bangalore NaN ACTIVE

Cleaned Data:

city gender status

0 New York Male active

1 New York Male -

2 Los Angeles Female inactive

3 Los Angeles Female none

4 Bangalore Unknown active

**🔍 Summary Table**

| **Method** | **Example Line** | **Purpose** |
| --- | --- | --- |
| df.replace(dict) | df['city'] = df['city'].replace({...}) | Replace multiple values directly |
| df[col].map(dict) | df['gender'] = df['gender'].map(gender\_map) | Map values in a Series |
| df[col].fillna(val) | df['gender'] = df['gender'].fillna('Unknown') | Fill missing entries |
| df[col].astype(str) | df['status'] = df['status'].astype(str) | Convert non-strings to string |
| df.apply(lambda x: ...) | df['status'] = df['status'].apply(lambda x: ...) | Custom processing per row |
| df[col].str.lower() | df['status'] = df['status'].str.lower() | Normalize text to lowercase |

**5. Column Naming Cleanup**

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

**📌 Example:**

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

**📚 Libraries:**

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

**✅ Use Case:**

* Standardizing column names before modelling

**Goal: Column Naming Cleanup**

* ✅ Remove **special characters**
* ✅ Strip **leading/trailing whitespaces**
* ✅ Convert to **snake\_case** (i.e., lowercase + underscores instead of spaces)

**🧾 Example DataFrame with Messy Column Names**

import pandas as pd

df = pd.DataFrame({

'First Name': ['Alice', 'Bob'],

'Last-Name': ['Smith', 'Jones'],

'AGE (Years)': [25, 30],

'E-mail Address ': ['alice@example.com', 'bob@example.com']

})

print("Original Columns:")

print(df.columns.tolist())

**🖨 Output:**

Original Columns:

['First Name', 'Last-Name', 'AGE (Years)', 'E-mail Address ']

**🛠️ Clean Column Names Function (Step-by-step)**

import re

def clean\_column\_names(columns):

clean\_cols = []

for col in columns:

col = col.strip() # Remove leading/trailing spaces

col = col.lower() # Convert to lowercase

col = re.sub(r'[^\w\s]', '', col) # Remove special characters

col = re.sub(r'\s+', '\_', col) # Replace whitespace with underscore

clean\_cols.append(col)

return clean\_cols

**✅ Apply Column Cleanup**

df.columns = clean\_column\_names(df.columns)

print("\nCleaned Columns:")

print(df.columns.tolist())

**🖨 Output:**

Cleaned Columns:

['first\_name', 'lastname', 'age\_years', 'email\_address']

**🔍 Summary of Transformations**

| **Original** | **Transformation Step** | **Result** |
| --- | --- | --- |
| "First Name" | Lowercase + space → underscore | "first\_name" |
| "Last-Name" | Hyphen removed | "lastname" |
| "AGE (Years)" | Parentheses removed, lowercase | "age\_years" |
| "E-mail Address " | Trailing space, hyphen removed | "email\_address" |

**🧠 Bonus Tip (Pandas 2.0+ style with str methods)**

python

CopyEdit

df.columns = (

df.columns.str.strip()

.str.lower()

.str.replace(r'[^\w\s]', '', regex=True)

.str.replace(r'\s+', '\_', regex=True)

)

# Now df.columns are clean!

**🔹 6. Language or Locale Normalization**

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

**📌 Example:**

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

**LangDetect / TextBlob**

* 🔤 **Purpose**: Detect the **language** of a given text.
* ✅ Automatically identifies the most probable language (like en, fr, de, etc.)

**2. Unidecode**

* 🧹 **Purpose**: Remove **accents/diacritics** and convert Unicode characters to plain ASCII.
* ✅ Useful for normalizing names and locations (e.g., “José” → "Jose")

**🧪 Sample DataFrame**

import pandas as pd

df = pd.DataFrame({

'name': ['José', 'François', 'Müller', 'Łukasz', 'Chloé'],

'comment': ['Hola, ¿cómo estás?', 'Bonjour tout le monde!', 'Guten Morgen', 'Cześć!', 'Salut!']

})

**🧠 1. Detect Language (with langdetect and TextBlob)**

**➤ Using langdetect**

from langdetect import detect

df['language\_detect'] = df['comment'].apply(detect)

**➤ Using TextBlob**

from textblob import TextBlob

df['language\_blob'] = df['comment'].apply(lambda x: TextBlob(x).detect\_language())

⚠️ Note: TextBlob requires an internet connection and Google Translate API behind the scenes, so it's slower and might fail without access.

**🧠 2. Remove Accents (with Unidecode)**

from unidecode import unidecode

df['name\_ascii'] = df['name'].apply(unidecode)

**🖨 Final Output**

print(df)

**🔍 Output:**

| **name** | **comment** | **language\_detect** | **name\_ascii** |
| --- | --- | --- | --- |
| José | Hola, ¿cómo estás? | es | Jose |
| François | Bonjour tout le monde! | fr | Francois |
| Müller | Guten Morgen | de | Muller |
| Łukasz | Cześć! | pl | Lukasz |
| Chloé | Salut! | fr | Chloe |

**📚 Use Cases**

| **Library** | **Use Case** |
| --- | --- |
| langdetect | Auto-language detection before translation or NLP |
| TextBlob | Language detection + sentiment analysis |
| unidecode | Normalize names/emails/labels for matching and storage |

**📌 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**

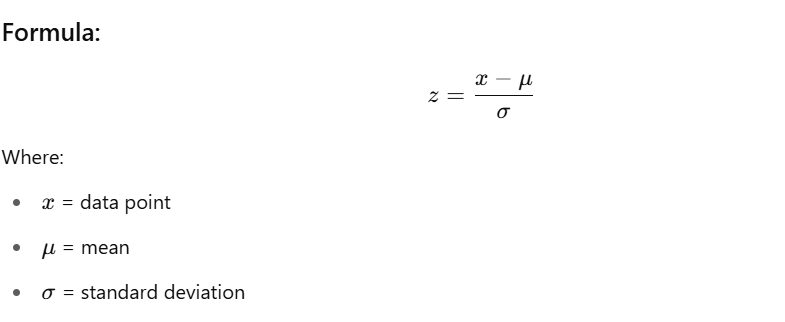
1. **Using Z-Score (Standard Score)**
2. **Using IQR (Interquartile Range)**
3. **Using Boxplot (Visualization-Based)**
4. **Using np.where() on Value Conditions**
5. **Using Isolation Forest (ML-based)**
6. **Using DBSCAN (Density-Based Clustering)**
7. **Using Mahalanobis Distance (for Multivariate Outliers)**

**🔹 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:**

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:**

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:**

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:**

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

**Contamination?**

The model **assumes** that **5%** of the data are outliers

Isolation Forest does **not automatically know** how many outliers exist. The contamination parameter:

* Helps the model **calibrate the decision threshold**
* Tells it **how many** samples to classify as outliers

**🔹 5. DBSCAN Clustering**

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

**📌 Example:**

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

**1. eps: Epsilon (ε) – Radius of Neighborhood**

* Defines the **maximum distance** between two samples for them to be considered as **neighbors**.
* Think of it as the **radius of a circle** drawn around each point.
* Smaller eps → tighter clusters
* Larger eps → looser clusters

🔸 If a point has **enough neighbors within this radius**, it becomes a **core point**.

**🔍 2. min\_samples: Minimum Number of Points in Neighborhood**

* Minimum number of **points (including itself)** required to form a **dense region** (i.e., a **core point**).
* If a point has **fewer than min\_samples** within eps, it might be considered **noise** or an **edge point**.

**✅ Definitions Based on These Two:**

| **Type of Point** | **Criteria** |
| --- | --- |
| **Core point** | ≥ min\_samples points within eps |
| **Border point** | < min\_samples within eps but in the neighborhood of a core point |
| **Noise point** | Not in any cluster, not even near a core point |

**🔹 6. LOF (Local Outlier Factor)**

Detects density-based local outliers.

**📌 Example:**

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:**

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**

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}")

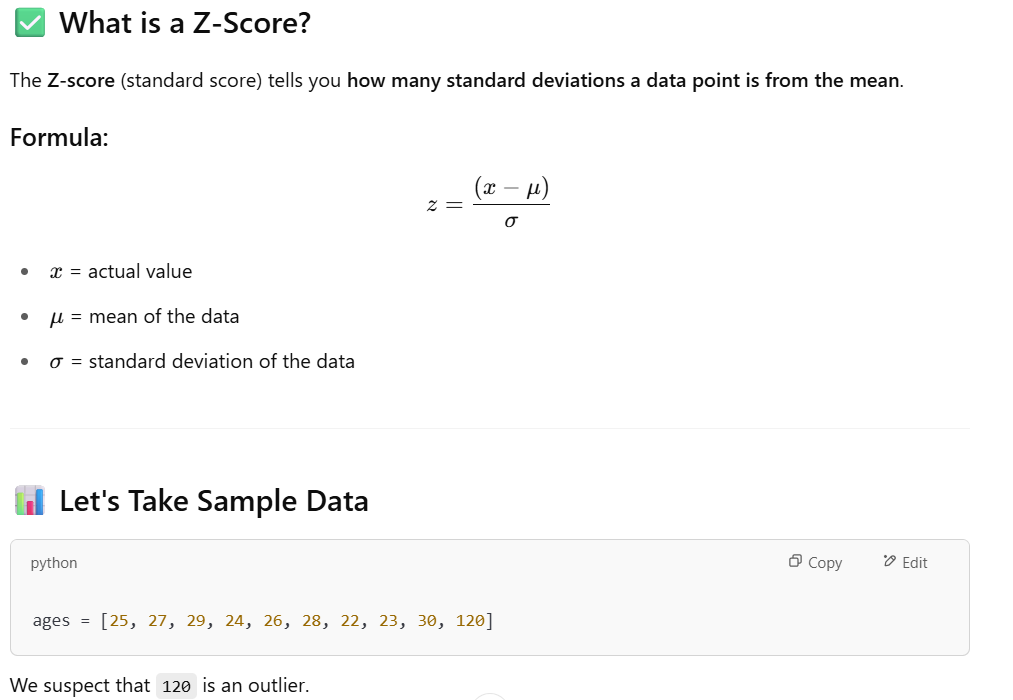
**✅ When Not to Use IQR?**

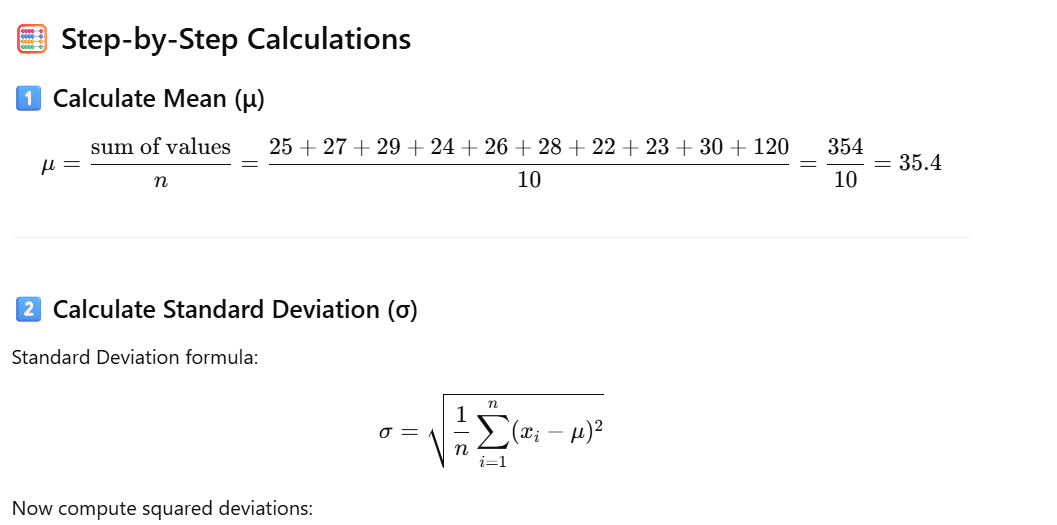
* **For normally distributed data**, z-score might be more appropriate.
* **For time series data**, other techniques like moving averages or seasonal decomposition might be better.

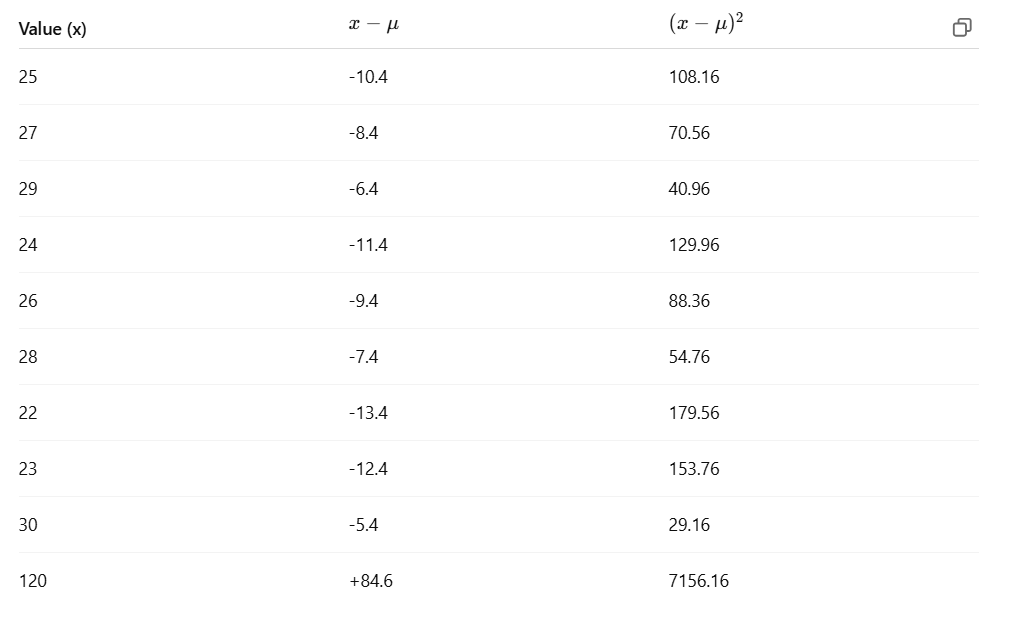
**🔁 Recap**

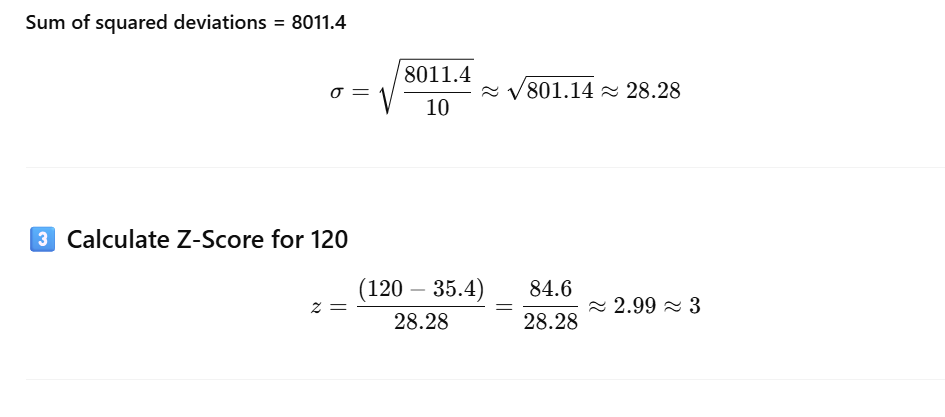
| **Term** | **Meaning** |
| --- | --- |
| Q1 | 25th percentile |
| Q3 | 75th percentile |
| IQR | Interquartile range (Q3 - Q1) |
| 1.5 \* IQR | Threshold for detecting moderate outliers |

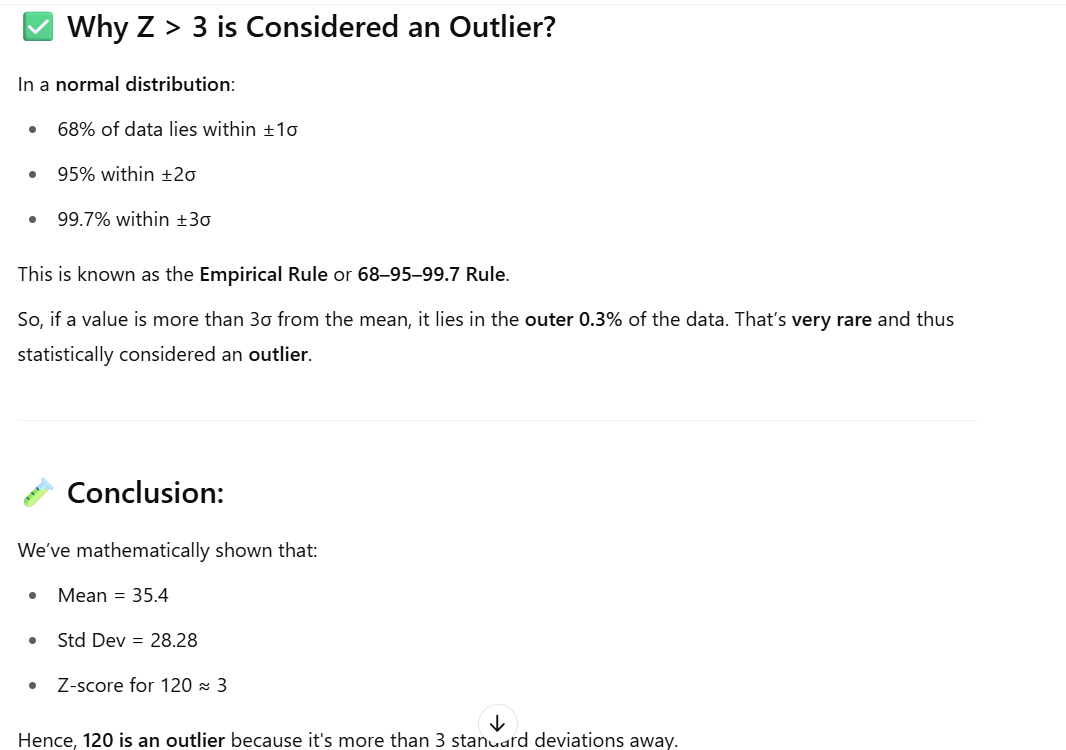
**Proof of the Z-Score Algorithms:**











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.

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.

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**

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.