Data Processing

**1)Handling Missing Values**

**Why It’s Important**

Missing data can occur due to various reasons — human error, system issues, or unavailable information.  
If left untreated, it can lead to:

* Biased model training
* Incorrect statistical analysis
* Reduced model accuracy

So, **handling missing values properly** ensures that our machine learning model learns from clean, consistent data.

**Common Strategies**

| **Method** | **Description** | **When to Use** |
| --- | --- | --- |
| **Remove Rows/Columns** | Drop rows or columns containing missing values. | When only a few data points are missing, and deleting them doesn’t affect overall data. |
| **Imputation (Filling)** | Replace missing values with an estimated value. | When missing data is significant and must be preserved for training. |
| **Predictive Imputation** | Use another model to predict missing values. | When missing patterns depend on other features (advanced). |

**Types of Imputation**

1. **Mean Imputation**
   * Replace missing numeric values with the **mean** of that column.
   * Works best when data follows a **normal distribution**.
2. **Median Imputation**
   * Replace with the **median** of the column.
   * Useful when data has **outliers** (since the median is less affected by them).
3. **Mode Imputation**
   * Replace missing **categorical values** with the **most frequent category**.
   * Example:

A screen shot of a computer code

AI-generated content may be incorrect.

**2)Handling Imbalanced Data (Upsampling and Downsampling)**

**What is an Imbalanced Dataset?**

An **imbalanced dataset** occurs when the number of samples in one class (the **majority class**) is much larger than in another class (the **minority class**).

For example, in a **fraud detection** dataset:

* 98% of transactions are *normal* (label 0)
* 2% are *fraudulent* (label 1)

If you train a model directly on such data, it may learn to predict **only the majority class**, achieving high accuracy but performing poorly on the minority (important) cases.

**Why We Need to Fix It**

Machine learning models — especially classification models like logistic regression, decision trees, or SVM — tend to **bias** toward the majority class, because it dominates the learning process.

Handling class imbalance ensures that:

* The model pays **equal attention** to both classes.
* It achieves **better generalization** and fairness in predictions.

**Techniques to Handle Imbalanced Data**

**1. Upsampling (Over-sampling) the Minority Class**

Upsampling means **increasing the number of samples in the minority class** by duplicating or synthetically generating new samples until both classes are roughly balanced.

**When to Use Upsampling:**

* When your dataset is **small**, and you don’t want to lose any data.
* When the **minority class** is important (e.g., disease detection, fraud, defects).

**Drawbacks:**

* Can cause **overfitting**, since the same minority samples may be repeated multiple times.

**2. Downsampling (Under-sampling) the Majority Class**

Downsampling means **reducing the number of samples in the majority class** to match the minority class size.

**When to Use Downsampling:**

* When you have a **very large dataset**, and removing some majority samples won’t hurt performance.
* When computational efficiency is important.

**Drawbacks:**

* You **lose information** from the majority class, which could reduce accuracy.

A screenshot of a computer program

AI-generated content may be incorrect.

**3)What Is Feature Scaling?**

In many machine learning models, the **scale of the input features** can heavily influence how the model learns.  
For example:

* feature1 ranges between **1 and 10**
* feature2 ranges between **1,000 and 10,000**

Even though both are equally important, the model might **give more importance to feature2**, simply because its values are much larger in magnitude.

Feature scaling solves this problem by bringing all features to **a similar scale**, ensuring that no variable dominates others due to its range.

**What Is Standardization?**

**Standardization** (also known as **Z-score normalization**) transforms the data so that:

* The **mean** of each feature becomes **0**
* The **standard deviation** becomes **1**

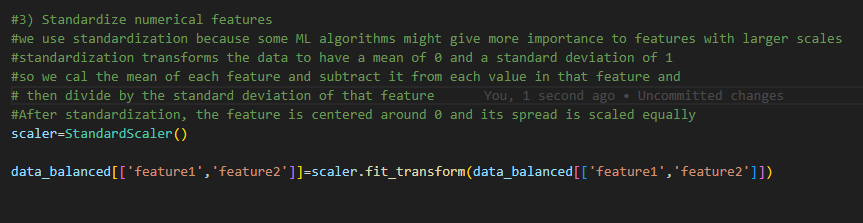
It is calculated using the formula:

Where:

* = the original value
* = mean of the feature
* = standard deviation of the feature

After standardization:

* The feature’s values are centered around **0**
* The spread of data (standard deviation) is **equal for all features**



**Why Standardization Is Important**

Standardization is crucial for algorithms that rely on **distance, gradients, or variance**, such as:

* Logistic Regression
* Linear Regression
* Support Vector Machines (SVM)
* K-Nearest Neighbors (KNN)
* Principal Component Analysis (PCA)

These algorithms assume that all features contribute equally to the model, which only holds true if the data is standardized.

**When to Use Standardization**

✅ Use standardization when:

* Your data **follows a normal (bell-shaped) distribution**
* You’re using **algorithms sensitive to scale or variance**

❌ Avoid it when:

* Features have **different units but do not require comparability** (e.g., one is categorical after encoding)
* The model type is **tree-based** (e.g., Decision Trees, Random Forests, XGBoost) — these are **scale-invariant**.

**Normalization (Min-Max Scaling)**

**Concept**

Normalization (also known as **Min-Max scaling**) rescales the data to a **fixed range**, usually **[0, 1]** (but you can choose any range, like [-1, 1]).  
It preserves the **shape of the original distribution**, but changes the **scale** of the values.

**When to Use**

* When you know your data **doesn’t follow a normal (Gaussian) distribution**.
* When using algorithms that are **distance-based**, such as:
  + K-Nearest Neighbors (KNN)
  + K-Means Clustering
  + Neural Networks (e.g., using sigmoid or tanh activation)

**Why We Use It**

Because these models compute distances or similarities (like Euclidean distance), having features on **different scales** can make one feature dominate others.  
Normalization ensures all features contribute **equally** to the result.

Imagine you have a dataset with **two features**:

| **Feature** | **Example Values** |
| --- | --- |
| Age | 18 – 70 |
| Income | 10,000 – 200,000 |

If you plot them together, you’ll notice that **Income** has a much larger range than **Age**.  
Now, let’s see why this matters 👇

**📏 Without normalization:**

Income (0–200,000) dominates Age (18–70) in the distance calculation —  
so the model mostly learns based on **income** and almost ignores **age**,  
because income changes cause **much bigger distance jumps** than age does.

**Example:**

If one feature (income) is in the hundreds of thousands, the other (age) becomes nearly invisible in the equation.

**⚖️ With normalization:**

We rescale both features to the same range, typically **[0, 1]**.

Now:

* Age: 18 → 0, 70 → 1
* Income: 10,000 → 0, 200,000 → 1

✅ Each feature now contributes **equally** to the distance, regardless of its original scale.

**4)Encoding Categorical Features**

**What it does:**  
Machine learning models can only work with numbers, so we need to **convert categorical (non-numeric) features into numeric values**.

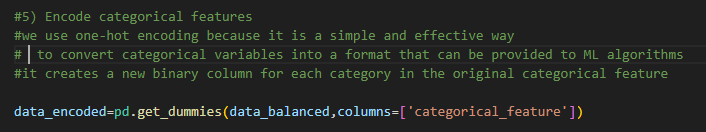
**Common method:** **One-Hot Encoding**

* Creates a **new binary column** for each category.
* Example: A “Color” feature with values **Red, Blue, Green** becomes:

| **Color** | **Red** | **Blue** | **Green** |
| --- | --- | --- | --- |
| Red | 1 | 0 | 0 |
| Blue | 0 | 1 | 0 |
| Green | 0 | 0 | 1 |

**Why we use it:**

* ML algorithms cannot understand text labels directly.
* One-hot encoding avoids giving **false order/priority** to categories.  
  (e.g., Red ≠ 1, Blue ≠ 2 — they are just different categories)



**Label Encoding**

**What it does:**  
Label encoding converts categorical values into **numbers**. Each category is assigned a **unique integer**.

**Example:**  
A “Color” feature with values **Red, Blue, Green** becomes:

| **Color** | **Encoded** |
| --- | --- |
| Red | 0 |
| Blue | 1 |
| Green | 2 |

**When to use:**

* Works well for **ordinal data**, where categories have a meaningful order (e.g., Low, Medium, High).

**Caution:**

* For **nominal data** (no order), label encoding can mislead models into thinking one category is larger or smaller than another.
* In such cases, **one-hot encoding** is usually better.

**5) Splitting the Dataset into Training and Test Sets**

**What it does:**  
Divides your dataset into two parts:

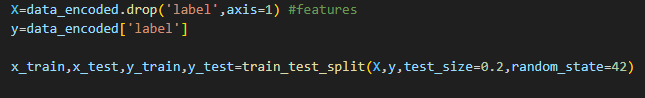
1. **Training set:** Used to **train the model**.
2. **Test set:** Used to **evaluate the model’s performance** on unseen data.

**Why we do it:**

* To check if the model can **generalize** to new data, not just memorize the training data.
* Prevents **overfitting**, where the model works well on training data but fails on real-world data.

**How it works:**

* Common split: **80% training, 20% testing** (can vary).
* Use train\_test\_split from **sklearn** to do this easily and reproducibly.
* You can also set a **random seed** to get the same split every time.



**6)feature selection**

**What it does:**  
Feature selection chooses the **most important features** from your dataset that are most relevant to predicting the target variable.

**Why we do it:**

* Reduces **dimensionality**, making the model simpler and faster.
* Improves **model performance** by removing irrelevant or noisy features.
* Helps prevent **overfitting**.

**Common methods:**

1. **Filter methods:** Use statistical tests to select features (e.g., ANOVA F-test, correlation).
2. **Wrapper methods:** Test different feature combinations using a model (e.g., recursive feature elimination).
3. **Embedded methods:** Feature selection is done during model training (e.g., feature importance in Random Forest).

