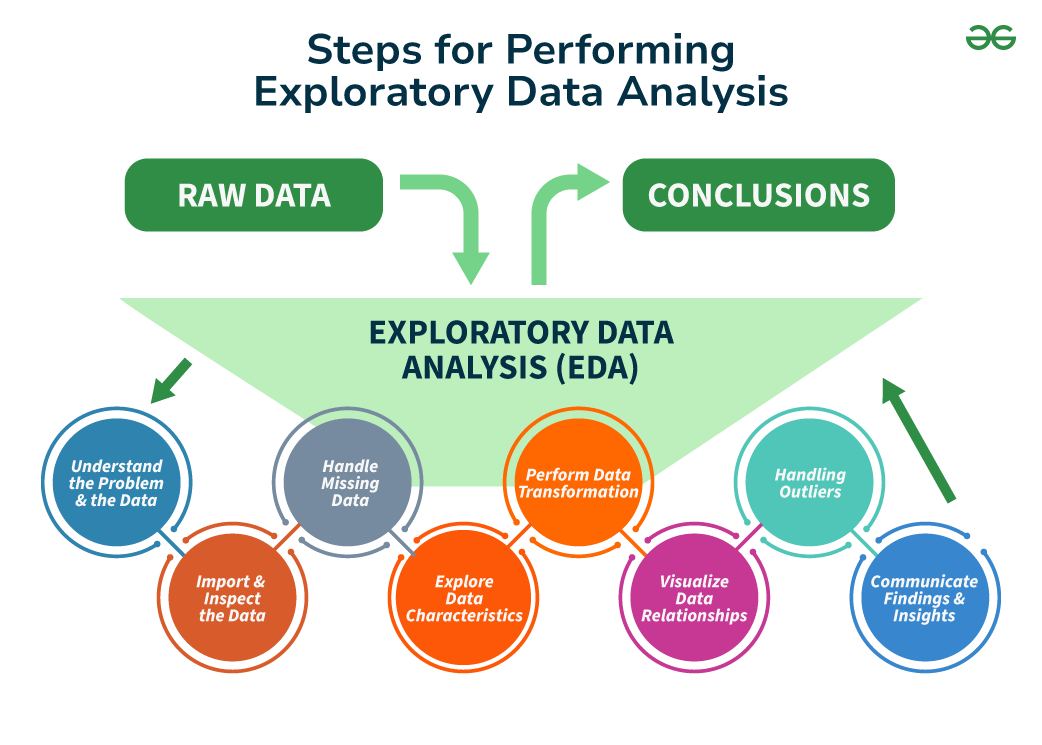
What is Exploratory Data Analysis?

**Exploratory Data Analysis (EDA)** is a important step in data science as it visualizing data to understand its main features, find patterns and discover how different parts of the data are connected. In this article, we will see more about **Exploratory Data Analysis (EDA)**.



2 / 3

**Why Exploratory Data Analysis is Important?**

Exploratory Data Analysis (EDA) is important for several reasons in the context of data science and statistical modeling. Here are some of the key reasons:

1. It helps to understand the dataset by showing how many features it has, what type of data each feature contains and how the data is distributed.
2. It helps to identify hidden patterns and relationships between different data points which help us in and model building.
3. Allows to identify errors or unusual data points (outliers) that could affect our results.
4. The insights gained from EDA help us to identify most important features for building models and guide us on how to prepare them for better performance.
5. By understanding the data it helps us in choosing best modeling techniques and adjusting them for better results.

**Types of Exploratory Data Analysis**

There are various types of EDA based on nature of records. Depending on the number of columns we are analyzing we can divide EDA into three types:

**1. Univariate Analysis**

[Univariate analysis](https://www.geeksforgeeks.org/what-is-univariate-bivariate-multivariate-analysis-in-data-visualisation/)focuses on studying one variable to understand its characteristics. It helps to describe data and find patterns within a single feature. Various common methods like histograms are used to show data distribution, box plots to detect outliers and understand data spread and bar charts for categorical data. Summary statistics like[**mean**,](https://www.geeksforgeeks.org/mean/) [**median**](https://www.geeksforgeeks.org/median/), [**mode**](https://www.geeksforgeeks.org/what-is-mode/),[**variance**](https://www.geeksforgeeks.org/variance/) and [**standard deviation**](https://www.geeksforgeeks.org/standard-deviation-formula/) helps in describing the central tendency and spread of the data

**2. Bivariate Analysis**

[Bivariate Analysis](https://www.geeksforgeeks.org/bivariate-analysis/) focuses on identifying relationship between two variables to find connections, correlations and dependencies. It helps to understand how two variables interact with each other. Some key techniques include:

* Scatter plots which visualize the relationship between two continuous variables.
* **C**orrelation coefficient measures how strongly two variables are related which commonly use [**Pearson's correlation**](https://www.geeksforgeeks.org/pearson-correlation-coefficient/) for linear relationships.
* Cross-tabulation or contingency tables shows the frequency distribution of two categorical variables and help to understand their relationship.
* **Line graphs** are useful for comparing two variables over time in time series data to identify trends or patterns.
* [**Covariance**](https://www.geeksforgeeks.org/mathematics-covariance-and-correlation/) measures how two variables change together but it is paired with the correlation coefficient for a clearer and more standardized understanding of the relationship.

**3. Multivariate Analysis**

[Multivariate Analysis](https://www.geeksforgeeks.org/multivariate-analysis-in-r/) identify relationships between two or more variables in the dataset and aims to understand how variables interact with one another which is important for statistical modeling techniques. It include techniques like:

* [**Pair plots**](https://www.geeksforgeeks.org/python-seaborn-pairplot-method/)which shows the relationships between multiple variables at once and helps in understanding how they interact.
* Another technique is [**Principal Component Analysis (PCA**](https://www.geeksforgeeks.org/principal-component-analysis-pca/)**)** which reduces the complexity of large datasets by simplifying them while keeping the most important information.
* [**Spatial Analysis**](https://www.geeksforgeeks.org/what-is-spatial-analysis/) is used for geographical data by using maps and spatial plotting to understand the geographical distribution of variables.
* [**Time Series Analysis**](https://www.geeksforgeeks.org/time-series-data-visualization-in-python/) is used for datasets that involve time-based data and it involves understanding and modeling patterns and trends over time. Common techniques include line plots, autocorrelation analysis, moving averages and [ARIMA](https://www.geeksforgeeks.org/python-arima-model-for-time-series-forecasting/) models.

**Steps for Performing Exploratory Data Analysis**

It involves a series of steps to help us understand the data, uncover patterns, identify anomalies, test hypotheses and ensure the data is clean and ready for further analysis. It can be done using different tools like:

* In Python, Pandas is used to clean, filter and manipulate data. [Matplotlib](https://www.geeksforgeeks.org/data-visualization-using-matplotlib/)helps to create basic visualizations while [Seaborn](https://www.geeksforgeeks.org/introduction-to-seaborn-python/)makes more attractive plots. For interactive visualizations Plotly is a good choice.
* In R, **ggplot2**is used for creating complex plots, **dplyr**helps with data manipulation and **tidyr**makes sure our data is organized and easy to work with.

Its step includes:

**Step 1: Understanding the Problem and the Data**

The first step in any data analysis project is to fully understand the problem we're solving and the data we have. This includes asking key questions like:

1. What is the business goal or research question?
2. What are the variables in the data and what do they represent?
3. What types of data (numerical, categorical, text, etc.) do you have?
4. Are there any known data quality issues or limitations?
5. Are there any domain-specific concerns or restrictions?

By understanding the problem and the data, we can plan our analysis more effectively, avoid incorrect assumptions and ensure accurate conclusions.

**Step 2: Importing and Inspecting the Data**

After understanding the problem and the data, next step is to import the data into our analysis environment such as Python, R or a spreadsheet tool. It’s important to find data to gain an basic understanding of its structure, variable types and any potential issues. Here’s what we can do:

1. Load the data into our environment carefully to avoid errors or truncations.
2. Check the size of the data like number of rows and columns to understand its complexity.
3. Check for missing values and see how they are distributed across variables since missing data can impact the quality of your analysis.
4. Identify data types for each variable like numerical, categorical, etc which will help in the next steps of data manipulation and analysis.
5. Look for errors or inconsistencies such as invalid values, mismatched units or outliers which could show major issues with the data.

By completing these tasks we'll be prepared to clean and analyze the data more effectively.

**Step 3: Handling Missing Data**

[Missing data](https://www.geeksforgeeks.org/ml-handling-missing-values/)is common in many datasets and can affect the quality of our analysis. During EDA it's important to identify and handle missing data properly to avoid biased or misleading results. Here’s how to handle it:

1. Understand the patterns and possible causes of missing data. Is it missing completely at random (MCAR), missing at random (MAR) or missing not at random (MNAR). Identifying this helps us to find best way to handle the missing data.
2. Decide whether to remove missing data or impute (fill in) the missing values. Removing data can lead to biased outcomes if the missing data isn’t MCAR. Filling values helps to preserve data but should be done carefully.
3. Use appropriate imputation methods like mean or median imputation, [regression](https://www.geeksforgeeks.org/regression-in-machine-learning/)imputation or machine learning techniques like[KNN](https://www.geeksforgeeks.org/k-nearest-neighbours/)or [decision trees](https://www.geeksforgeeks.org/decision-tree/)based on the data’s characteristics.
4. Consider the impact of missing data. Even after imputing, missing data can cause uncertainty and bias so understands the result with caution.

Properly handling of missing data improves the accuracy of our analysis and prevents misleading conclusions.

**Step 4: Exploring Data Characteristics**

After addressing missing data we find the characteristics of our data by checking the distribution, central tendency and variability of our variables and identifying outliers or anomalies. This helps in selecting appropriate analysis methods and finding major data issues. We should calculate summary statistics like mean, median, mode, standard deviation, skewness and kurtosis for numerical variables. These provide an overview of the data’s distribution and helps us to identify any irregular patterns or issues.

**Step 5: Performing Data Transformation**

Data transformation is an important step in EDA as it prepares our data for accurate analysis and modeling. Depending on our data's characteristics and analysis needs, we may need to transform it to ensure it's in the right format. Common transformation techniques include:

1. Scaling or normalizing numerical variables like [min-max scaling](https://www.geeksforgeeks.org/standardscaler-minmaxscaler-and-robustscaler-techniques-ml/) or [standardization](https://www.geeksforgeeks.org/what-is-standardization-in-machine-learning/).
2. Encoding categorical variables for machine learning like [one-hot encoding](https://www.geeksforgeeks.org/ml-one-hot-encoding/) or [label encoding.](https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/)
3. Applying mathematical transformations like [logarithmic square root](https://www.geeksforgeeks.org/square-root-number-using-log/)to correct skewness or non-linearity.
4. Creating new variables from existing ones like calculating ratios or combining variables.
5. Aggregating or grouping data based on specific variables or conditions.

**Step 6: Visualizing Relationship of Data**

Visualization helps to find relationships between variables and identify patterns or trends that may not be seen from summary statistics alone.

1. For categorical variables, create frequency tables, bar plots and pie charts to understand the distribution of categories and identify imbalances or unusual patterns.
2. For numerical variables generate histograms, box plots, violin plots and density plots to visualize distribution, shape, spread and potential outliers.
3. To find relationships between variables use scatter plots, correlation matrices or statistical tests like Pearson’s correlation coefficient or Spearman’s rank correlation.

**Step 7: Handling Outliers**

Outliers are data points that differs from the rest of the data may caused by errors in measurement or data entry. Detecting and handling outliers is important because they can skew our analysis and affect model performance. We can identify outliers using methods like[**interquartile range (IQR)**](https://www.geeksforgeeks.org/interquartile-range-iqr/), [**Z-scores**](https://www.geeksforgeeks.org/z-score-in-statistics/) or domain-specific rules. Once identified it can be removed or adjusted depending on the context. Properly managing outliers shows our analysis is accurate and reliable.

**Step 8: Communicate Findings and Insights**

The final step in EDA is to communicate our findings clearly. This involves summarizing the analysis, pointing out key discoveries and presenting our results in a clear way.

1. Clearly state the goals and scope of your analysis.
2. Provide context and background to help others understand your approach.
3. Use visualizations to support our findings and make them easier to understand.
4. Highlight key insights, patterns or anomalies discovered.
5. Mention any limitations or challenges faced during the analysis.
6. Suggest next steps or areas that need further investigation.

Effective communication is important to ensure that our EDA efforts make an impact and that stakeholders understand and act on our insights. By following these steps and using the right tools, EDA helps in increasing the quality of our data, leading to more informed decisions and successful outcomes in any data-driven project.

[notes](https://www.geeksforgeeks.org/data-analysis/what-is-exploratory-data-analysis/)

## 🔍 Key Difference:

| **Aspect** | **Data Cleaning** | **Data Preprocessing** |
| --- | --- | --- |
| **Definition** | Fixing or removing incorrect, missing, or bad data | Transforming raw data into a format suitable for modeling |
| **Scope** | Subset of preprocessing | Broader — includes cleaning + transformations |
| **Goal** | Improve data quality | Make data usable for analysis or machine learning |
| **When it's done** | Early stage of EDA | Just before modeling |

## 🔧 **Data Cleaning Includes:**

* Handling **missing values** (e.g., fill with median or drop rows)
* Fixing **inconsistent formatting** (e.g., "Male" vs "male")
* Removing **duplicates**
* Removing or correcting **outliers**
* Detecting **erroneous values** (e.g., negative age)

## 🛠️ **Data Preprocessing Includes:**

* ✅ **Data cleaning** (as one step)
* 🧮 **Encoding** categorical variables (e.g., one-hot encoding)
* 📏 **Scaling/normalizing** features
* 🧹 **Feature selection/extraction**
* 🧱 **Splitting** data into train/test
* 🔁 **Handling imbalanced data**

### 🔁 Example in Titanic Dataset:

| **Task** | **Cleaning or Preprocessing?** |
| --- | --- |
| Fill missing age with median | Cleaning ✅ |
| Drop name, ticket, cabin | Cleaning ✅ |
| Encode sex with get\_dummies | Preprocessing ✅ |
| Scale age and fare | Preprocessing ✅ |

### ✅ Summary:

**Data Cleaning** is a **subset** of **Data Preprocessing**.  
You **clean the data first**, then **transform it** to make it suitable for modeling.

Great question! Feature scaling has a **huge impact** in data preprocessing, especially when using algorithms that are **sensitive to the scale of features**.

## 🎯 What Is Feature Scaling?

Feature scaling transforms values of features to a **common scale**, usually:

* **Standardization** (mean = 0, std = 1)
* **Normalization** (min = 0, max = 1)

## 🚀 Why Feature Scaling Matters (Impacts):

### 1. ✅ ****Improves Model Performance (Speed & Accuracy)****

* Algorithms like **KNN**, **SVM**, **Logistic Regression**, and **Gradient Descent-based models** perform **better and converge faster** when features are scaled.
* Otherwise, **larger-valued features dominate** smaller ones.

### 2. 📏 ****Ensures Fair Weighting of Features****

Without scaling:

| **Feature** | **Range** |
| --- | --- |
| age | 0–100 |
| income | 0–100,000 |

* Models may **assume income is more important**, just because of its scale.
* Scaling brings them to **equal footing**.

### 3. 💡 ****Improves Distance-Based Algorithms****

Algorithms like:

* **K-Nearest Neighbors (KNN)**
* **K-Means Clustering**
* **DBSCAN**

...rely on **Euclidean distance** or similar metrics, which are **scale-sensitive**.

🧠 If one feature has a much larger scale, it can **distort distance calculations**.

### 4. 📉 ****Improves Convergence in Gradient Descent****

* Models like **Logistic Regression** or **Neural Networks** that use **gradient descent** benefit from scaled features because:
  + The **cost function** becomes smoother.
  + The **algorithm converges faster** and more reliably.

## ❌ When Scaling is Not Needed

* **Tree-based models** like:
  + Decision Tree
  + Random Forest
  + XGBoost, LightGBM

These models are **not sensitive to feature scale**. They split data based on thresholds, not distance.

## ✅ Summary:

| **Benefit** | **Explanation** |
| --- | --- |
| Faster training | Especially in gradient-based models |
| More accurate distance metrics | In KNN, K-Means, SVM |
| Fair comparison between features | Prevents dominance by features with larger values |
| Needed for many ML algorithms | Except tree-based ones |

Great question! Choosing between MinMaxScaler and StandardScaler depends on:

* The **distribution** of your data
* The **algorithm** you're using
* The **presence of outliers**

Let’s break it down.

## ⚖️ **Difference Between MinMaxScaler and StandardScaler**

| **Scaler** | **Formula** | **Resulting Range** | **Sensitive to Outliers?** |
| --- | --- | --- | --- |
| **StandardScaler** | (x−μ)/σ(x - \mu) / \sigma | Mean = 0, Std Dev = 1 | ✅ Yes (uses mean/std) |
| **MinMaxScaler** | (x−xmin)/(xmax−xmin)(x - x\_{min}) / (x\_{max} - x\_{min}) | 0 to 1 (or custom range) | ✅ Yes (depends on min/max) |

## ✅ When to Use Each

### 🔹 Use ****StandardScaler**** When:

* Features are **normally distributed** (bell-shaped).
* You're using **algorithms that assume Gaussian distribution**, like:
  + Logistic Regression
  + Linear Regression
  + SVM
  + KNN
  + Neural Networks
* You **don't mind negative values**.

### 🔹 Use ****MinMaxScaler**** When:

* You want features in a **bounded range** (e.g., 0–1), especially for:
  + **Neural Networks** (especially in deep learning)
  + **Image pixel scaling**
* Your data does **not follow normal distribution**.
* You're **not using tree-based models** (which don't require scaling anyway).
* You're preparing data for algorithms **sensitive to magnitude**, like:
  + KNN
  + SVM

### ❗ Be Careful:

Both scalers are **sensitive to outliers**. For **robust scaling**, use:

from sklearn.preprocessing import RobustScaler

Which uses **median and IQR**, and works well with outliers.

## ✅ Summary Table

| **Situation** | **Recommended Scaler** |
| --- | --- |
| Normally distributed features | StandardScaler |
| Bounded feature range (0–1) needed | MinMaxScaler |
| Outliers present | RobustScaler |
| Image pixel data | MinMaxScaler (0–1) |
| Neural networks (inputs) | Often MinMaxScaler |

## 🎯 What is Scaling?

**Scaling** is the process of transforming your data so that all numeric features are on a **comparable scale**, typically:

* ✅ Between 0 and 1 (MinMax scaling)
* ✅ With mean = 0 and standard deviation = 1 (Standard scaling)

## 🔍 Why Is Scaling Important?

Many machine learning models are **sensitive to the scale** of input features. For example:

| **Without Scaling** | **With Scaling** |
| --- | --- |
| Larger values dominate | All features contribute equally |
| Slower convergence | Faster, stable training |
| Poor model accuracy | Better performance in gradient-based models |

## 🤖 Models That **Require** Scaling:

| **Model Type** | **Needs Scaling?** |
| --- | --- |
| KNN (K-Nearest Neighbors) | ✅ Yes |
| SVM (Support Vector Machine) | ✅ Yes |
| Logistic/Linear Regression | ✅ Yes |
| PCA / Clustering (KMeans) | ✅ Yes |
| Neural Networks | ✅ Yes |

## 🧱 Models That **Don't Need** Scaling:

| **Model Type** | **Needs Scaling?** |
| --- | --- |
| Decision Tree | ❌ No |
| Random Forest | ❌ No |
| XGBoost / LightGBM | ❌ No |

## 🔧 Common Scaling Methods

| **Scaler** | **Description** | **Use When** |
| --- | --- | --- |
| StandardScaler | Mean = 0, Std = 1 | General-purpose |
| MinMaxScaler | Scales between 0 and 1 | When bounded range is needed |
| RobustScaler | Uses median and IQR | Data has outliers |
| MaxAbsScaler | Scales by max absolute value | Sparse data (e.g. text vectors) |

## ✅ Example (StandardScaler)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df\_scaled = df.copy()

df\_scaled[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

## 📌 Summary:

| **🔹 Scaling helps:** | **🔹 When to skip:** |
| --- | --- |
| Models using distances | Tree-based models |
| Gradient-based training | When only categorical data |
| Faster convergence | Already normalized data |

# ⚖️ **Types of Feature Scaling in Machine Learning**

Scaling is used to **normalize numerical features** so that they’re on the **same scale**, helping models learn better and faster.

## 🔢 1. **Standardization (Z-score Scaling)**

### ✅ ****Definition****:

Transforms data to have a **mean of 0** and **standard deviation of 1**.

### 🧮 ****Formula****:

z=x−μσz = \frac{x - \mu}{\sigma}

Where:

* μ\mu: mean of the feature
* σ\sigma: standard deviation

### 📌 ****Use when****:

* Data follows a **normal distribution** or is close to it.
* Algorithms that assume normality: **SVM**, **Logistic Regression**, **Linear Regression**, **KNN**, **PCA**

### ✅ ****Code****:

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df\_scaled = scaler.fit\_transform(df)

## 🌈 2. **Min-Max Scaling (Normalization)**

### ✅ ****Definition****:

Rescales values to a **range of 0 to 1**.

### 🧮 ****Formula****:

xscaled=x−xmin⁡xmax⁡−xmin⁡x\_{\text{scaled}} = \frac{x - x\_{\min}}{x\_{\max} - x\_{\min}}

### 📌 ****Use when****:

* Features have **different units** or **magnitudes**
* You want to **bound values** between 0 and 1 (e.g., image pixel values)

### ✅ ****Code****:

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df\_scaled = scaler.fit\_transform(df)

## 🧱 3. **Robust Scaling**

### ✅ ****Definition****:

Uses **median** and **interquartile range (IQR)**, making it robust to **outliers**.

### 🧮 ****Formula****:

xscaled=x−medianIQRx\_{\text{scaled}} = \frac{x - \text{median}}{IQR}

### 📌 ****Use when****:

* Your data contains **outliers**
* You want **less influence** from extreme values

### ✅ ****Code****:

from sklearn.preprocessing import RobustScaler

scaler = RobustScaler()

df\_scaled = scaler.fit\_transform(df)

## ➕ 4. **MaxAbs Scaling**

### ✅ ****Definition****:

Scales each feature by its **maximum absolute value**, keeping **sign** (negative or positive).

### 📌 ****Use when****:

* You're working with **sparse data** (lots of zeros), e.g. text TF-IDF vectors
* You don’t want to shift zero-centered data

### ✅ ****Code****:

from sklearn.preprocessing import MaxAbsScaler

scaler = MaxAbsScaler()

df\_scaled = scaler.fit\_transform(df)

## 🧠 Summary Table

| **Scaling Type** | **Range** | **Sensitive to Outliers?** | **Best For** |
| --- | --- | --- | --- |
| **StandardScaler** | Mean = 0, Std = 1 | Yes | Normal distributions |
| **MinMaxScaler** | [0, 1] | Yes (very sensitive) | Bounded scaling |
| **RobustScaler** | No fixed range | ❌ No | Data with outliers |
| **MaxAbsScaler** | [-1, 1] | Yes | Sparse data (e.g. text) |

You can apply a **log transformation** in Pandas using NumPy’s np.log() or np.log1p() functions.

**✅ 1. Basic log transformation (natural log):**

import numpy as np

df['log\_column'] = np.log(df['your\_column'])

**⚠️ Important:**

* np.log() **only works with positive numbers**. If there are 0 or negative values, it will return -inf or NaN.

**✅ 2. Safe log transformation (to handle zeros):**

df['log\_column'] = np.log1p(df['your\_column'])

* np.log1p(x) is equal to log(1 + x) → avoids issues with 0 values.
* Especially useful for skewed data with zeros.

**✅ 3. Log base 10 or base 2:**

df['log10\_column'] = np.log10(df['your\_column']) # Base 10

df['log2\_column'] = np.log2(df['your\_column']) # Base 2

**🧪 Example:**

import pandas as pd

import numpy as np

df = pd.DataFrame({

'range\_km': [100, 200, 300, 400, 0, -10]

})

# Safe log transform

df['range\_km\_log'] = np.log1p(df['range\_km'])

print(df)

Let me know if you want to apply log transformation to **multiple columns** at once or **visualize the effect** using histograms.

## 🎯 What is **Standardization**?

**Standardization** (also called **Z-score normalization**) is a **scaling technique** where:

Standardized value=(x−μ)σ\text{Standardized value} = \frac{(x - \mu)}{\sigma}

Where:

* xx = original value
* μ\mu = mean of the feature
* σ\sigma = standard deviation of the feature

## 🔍 What It Does:

It **transforms** data so that:

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

## 🧠 Why Use Standardization?

Standardization is especially useful when:

* Features have **different units or magnitudes**
  + e.g., price in thousands, mileage in tens of thousands
* Algorithms assume **Gaussian (normal)** distribution (e.g., logistic regression, SVM)
* You want to handle **negative values** correctly (which MinMaxScaler doesn’t)

## ✅ Example with StandardScaler in sklearn

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

df[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

## 📊 Before vs After Standardization

| **Original Price** | **Standardized Price** |
| --- | --- |
| 30,000 | 0.25 |
| 20,000 | -0.65 |
| 40,000 | 1.15 |

## ⚠️ Difference Between **Standardization** vs **Normalization**

| **Aspect** | **Standardization (Z-score)** | **Normalization (MinMax)** |
| --- | --- | --- |
| Output Range | Centered around 0 | [0, 1] |
| Formula | x−μσ\frac{x - \mu}{\sigma} | x−min⁡max⁡−min⁡\frac{x - \min}{\max - \min} |
| Sensitive to Outliers | Yes | Yes (more sensitive) |
| When to Use | General-purpose | When scale must be bounded |

## ✅ Summary

📌 **Standardization** is a method of **scaling data** to have a **mean of 0** and a **standard deviation of 1**. It’s very useful in most machine learning models, especially ones that rely on **distance, gradient, or probability**.

Would you like to apply standardization to your dataset now?

## 🎯 What is **Normalization**?

**Normalization** is a scaling technique that **resizes the range** of your numeric features so that they all fall **between 0 and 1** (or sometimes -1 to 1).

### 🔧 Formula (Min-Max Normalization):

xnormalized=x−xminxmax−xminx\_{\text{normalized}} = \frac{x - x\_{\text{min}}}{x\_{\text{max}} - x\_{\text{min}}}

Where:

* xx = original value
* xminx\_{\text{min}} = minimum value of the feature
* xmaxx\_{\text{max}} = maximum value of the feature

## ✅ Example

| **Original Mileage** | **Normalized Mileage** |
| --- | --- |
| 10,000 | 0.0 |
| 55,000 | 0.75 |
| 80,000 | 1.0 |

## 🧠 When to Use Normalization?

Use normalization when:

* You need **values between 0 and 1**
* You're working with **neural networks**
* You're using **distance-based models** like KNN, K-means
* The data is **not normally distributed**

## ⚖️ Difference: Normalization vs Standardization

| **Feature** | **Normalization (Min-Max)** | **Standardization (Z-Score)** |
| --- | --- | --- |
| Output Range | 0 to 1 | Mean = 0, Std = 1 |
| Handles Outliers | ❌ No | ✅ Better |
| Formula | x−min⁡max⁡−min⁡\frac{x - \min}{\max - \min} | x−μσ\frac{x - \mu}{\sigma} |
| When to Use | When scale must be bounded | General-purpose |

## ✅ How to Apply in Python

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

### 📌 Summary:

**Normalization** rescales values to **[0, 1]** — ideal when all features need to be in a fixed range or when models are **sensitive to magnitude differences**.

## 🧠 What is **Encoding** in Machine Learning?

**Encoding** is the process of converting **categorical (text) data** into **numeric format** so that machine learning models can understand and work with it.

🧠 ML models only understand numbers — not text like 'Sedan', 'Diesel', or 'Toyota'.

## 🎯 Why is Encoding Important?

Many columns in your dataset might contain **categories** like:

* fuel: 'Petrol', 'Diesel', 'Electric'
* transmission: 'Automatic', 'Manual'
* body: 'SUV', 'Sedan', 'Truck'

These need to be converted to **numerical values** before feeding them into ML models.

## 🔧 Types of Encoding Methods

### 1️⃣ Label Encoding

* Converts categories to integers (0, 1, 2, ...)

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df['fuel'] = le.fit\_transform(df['fuel'])

| **fuel** | **fuel\_encoded** |
| --- | --- |
| Petrol | 2 |
| Diesel | 1 |
| Electric | 0 |

✅ Simple and memory efficient  
❌ Can mislead models into thinking there's **order** or **distance** between values

### 2️⃣ One-Hot Encoding

* Creates a new binary column for each category (0 or 1)

df = pd.get\_dummies(df, columns=['fuel'], drop\_first=True)

| **fuel\_Diesel** | **fuel\_Electric** | **fuel\_Petrol** |
| --- | --- | --- |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |

✅ No ordinal relationship  
❌ Can lead to **many columns** if there are too many categories

### 3️⃣ Frequency / Count Encoding

* Replaces each category with its frequency in the dataset

freq\_map = df['model'].value\_counts()

df['model\_encoded'] = df['model'].map(freq\_map)

✅ Simple  
✅ Keeps dimensionality low  
❌ May not reflect true semantic meaning

### 4️⃣ Target Encoding (Advanced)

* Replaces category with **mean of target variable** (e.g., average price per car make)

target\_mean = df.groupby('make')['price'].mean()

df['make\_encoded'] = df['make'].map(target\_mean)

✅ Powerful for regression  
❌ Must be done carefully to avoid **data leakage**

## 📌 Summary Table

| **Method** | **Best For** | **Limitation** |
| --- | --- | --- |
| Label Encoding | Low-cardinality categorical | Implies order (not always good) |
| One-Hot Encoding | Nominal (unordered) categories | High dimensionality |
| Frequency Encoding | High-cardinality columns | Doesn't capture meaning |
| Target Encoding | Regression tasks | Risk of leakage |

# 🧠 **When to Use Different Types of Encoding**

## 🔡 What is Encoding?

**Encoding** converts **categorical (textual)** data into **numerical** form, so machine learning models can understand and process it.

## 🔢 1. **Label Encoding**

### ✅ Use When:

* The **categorical variable is ordinal** (has an inherent order)
* The number of unique categories is **small**
* You’re using **tree-based models** (they can handle label encodings)

### 🧾 Example:

| **Size** | **Encoded** |
| --- | --- |
| Small | 0 |
| Medium | 1 |
| Large | 2 |

### ✅ Suitable for:

* **Decision Trees**, **Random Forests**, **XGBoost**

### ❌ Avoid for:

* **Linear Models**, **KNN**, **SVM** → it implies an artificial order if not intended.

### 🔧 Code:

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

df['size'] = le.fit\_transform(df['size'])

## 🟨 2. **One-Hot Encoding**

### ✅ Use When:

* The variable is **nominal** (no order between categories)
* The number of categories is **low to moderate**
* Model requires **no assumption of order**

### 🧾 Example:

| **Fuel** | **Petrol** | **Diesel** | **Electric** |
| --- | --- | --- | --- |
| Petrol | 1 | 0 | 0 |
| Diesel | 0 | 1 | 0 |

### ✅ Suitable for:

* **Linear Regression**, **Logistic Regression**, **Neural Networks**, **KNN**

### ❌ Avoid when:

* Feature has **too many unique categories** → leads to **high dimensionality** (curse of dimensionality)

### 🔧 Code:

df = pd.get\_dummies(df, columns=['fuel'], drop\_first=True)

## 📊 3. **Ordinal Encoding**

### ✅ Use When:

* Categories have a **clear order**, like Low < Medium < High
* You want to **preserve the ranking**

### 🧾 Example:

| **Quality** | **Encoded** |
| --- | --- |
| Poor | 0 |
| Good | 1 |
| Excellent | 2 |

### ✅ Suitable for:

* Models that can interpret or benefit from **ordinal relationships**

### 🔧 Code:

from sklearn.preprocessing import OrdinalEncoder

encoder = OrdinalEncoder(categories=[['Poor', 'Good', 'Excellent']])

df['quality\_encoded'] = encoder.fit\_transform(df[['quality']])

## 🔢 4. **Frequency / Count Encoding**

### ✅ Use When:

* High-cardinality features (e.g., model with 100s of categories)
* Need a **compact representation** without creating many columns

### 🧾 Example:

| **Model** | **Count** |
| --- | --- |
| Corolla | 230 |
| Civic | 150 |
| Mustang | 90 |

### ✅ Suitable for:

* Tree-based models like RandomForest, XGBoost
* Cases with **many unique values**

### 🔧 Code:

df['model\_freq'] = df['model'].map(df['model'].value\_counts())

## 🎯 5. **Target Encoding (Mean Encoding)**

### ✅ Use When:

* You want to encode based on **mean of target variable**
* Useful for **regression problems**
* Needs to be used with **cross-validation** to avoid leakage

### 🧾 Example:

| **Brand** | **Avg\_Price** |
| --- | --- |
| BMW | 30000 |
| Ford | 20000 |

### ✅ Suitable for:

* Tree-based models
* Features that **strongly influence the target**

### ❌ Be careful with:

* **Data leakage** – only use training data when calculating mean

### 🔧 Code:

mean\_price = df.groupby('make')['price'].mean()

df['make\_encoded'] = df['make'].map(mean\_price)

## 📌 Summary Table

| **Encoding Type** | **Best for...** | **Model Type** | **Handles Order** | **Handles High Cardinality** |
| --- | --- | --- | --- | --- |
| **Label Encoding** | Ordinal / low-cardinality | Tree-based models | ✅ Yes | ❌ No |
| **One-Hot** | Nominal, few categories | Linear, NN, KNN | ❌ No | ❌ No |
| **Ordinal** | Ordered categories | Any | ✅ Yes | ❌ No |
| **Frequency** | High-cardinality nominal | Tree-based | ❌ No | ✅ Yes |
| **Target Encoding** | Feature correlated with target | Regression, Trees | ❌ No | ✅ Yes |

Great question! Handling missing data is **crucial** because it directly affects the **accuracy**, **reliability**, and **performance** of your data analysis or machine learning models.

## 📉 Why Handling Missing Data is Crucial

### 1. ✅ ****Preserves Data Integrity****

* Missing values can distort statistical summaries (mean, std, etc.).
* Leads to **inaccurate EDA** and misleading insights.

### 2. ✅ ****Ensures Model Compatibility****

* Most ML models (like LogisticRegression, RandomForest) **do not accept missing values**.
* Leads to errors or failed training if not handled.

### 3. ✅ ****Avoids Biased Models****

* Ignoring missing data or removing rows carelessly can introduce **bias** in the dataset.
* Especially important in **healthcare, finance**, or **sensitive applications**.

### 4. ✅ ****Improves Model Performance****

* Smart imputation can help models generalize better and make better predictions.

## 🛠️ Basic Techniques (Recap)

| **Method** | **Description** | **When to Use** |
| --- | --- | --- |
| Drop rows | Remove rows with missing values | Few missing values |
| Mean/Median | Fill numeric missing with avg/median | Data is symmetric or skewed |
| Mode | For categorical variables | Categorical and low cardinality |

Great question! In Pandas, the .agg() function supports a wide variety of **aggregation functions** — both built-in and custom — for summarizing your data.

### ✅ ****Common Built-in Aggregation Functions****:

You can pass them as strings (e.g., 'mean') or functions (e.g., np.mean):

| **Function Name** | **Description** |
| --- | --- |
| 'mean' | Average value |
| 'median' | Median value |
| 'std' | Standard deviation |
| 'var' | Variance |
| 'min' | Minimum value |
| 'max' | Maximum value |
| 'sum' | Sum of values |
| 'count' | Number of non-null values |
| 'nunique' | Number of unique values |
| 'first' | First value in group |
| 'last' | Last value in group |
| 'prod' | Product of values |
| 'sem' | Standard error of the mean |
| 'size' | Group size (use .size() separately) |

### 🧪 Example with many aggregations:

import numpy as np

x = df.groupby('brand').agg({

'range\_km': ['mean', 'min', 'max', 'std', 'nunique'],

'torque\_nm': ['sum', 'mean', 'median', 'var']

})

### 🧠 ****Custom Aggregation Functions****:

You can also define your own:

x = df.groupby('brand').agg({

'range\_km': [lambda x: x.max() - x.min(), 'mean']

})

You can even **name** them using a dictionary:

x = df.groupby('brand').agg(

range\_km\_range = ('range\_km', lambda x: x.max() - x.min()),

torque\_nm\_mean = ('torque\_nm', 'mean')

)

### 📌 Pro Tip:

To see all valid aggregation functions for a numeric column:

import pandas as pd

print(pd.core.base.SelectionMixin.\_builtin\_table['numeric'])

Or refer to [Pandas documentation on aggregation](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.agg.html)

Let me know if you want to filter by top-N brands, plot the results, or export them to

## 🧠 Advanced Imputation Techniques

Here are more **intelligent** techniques used in practice:

### 1. 📊 ****KNN Imputation****

* Uses **K-Nearest Neighbors** to fill missing values based on similar samples.
* Learns from the **structure of the data**.

from sklearn.impute import KNNImputer

imputer = KNNImputer(n\_neighbors=5)

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

✅ Best for: numeric features, small-to-medium datasets

### 2. 🔍 ****Multivariate Imputation by Chained Equations (MICE)****

* Predicts each missing value using **regression** on other features, **iteratively**.
* Maintains **relationships between features**.

from sklearn.experimental import enable\_iterative\_imputer

from sklearn.impute import IterativeImputer

imputer = IterativeImputer()

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

✅ Best for: multiple missing columns, highly correlated data

### 3. 🧠 ****Model-Based Imputation****

* Train a separate model (e.g., Random Forest) to predict missing values.
* Requires more code and control, but allows **custom logic**.

✅ Best for: critical columns where accuracy matters

### 4. 📈 ****Time-Series Imputation****

* For time-based data: use methods like:
  + **Forward fill / Backward fill**
  + **Interpolation** (linear, spline)
  + **Kalman filtering**, etc.

df['value'] = df['value'].interpolate(method='linear')

✅ Best for: time-indexed data

### 5. 📌 ****Categorical Encoding of Missing****

* Treat missing category as a **separate label**: 'Missing' or 'Unknown'.

df['embarked'].fillna('Missing', inplace=True)

✅ Best for: categorical columns in decision trees

## ✅ Summary

| **Technique** | **Works Well With** | **Avoid If...** |
| --- | --- | --- |
| Mean/Median/Mode | Simple numeric/categorical | Data has strong relationships |
| KNN Imputation | Small/clean numeric datasets | Large or sparse datasets |
| MICE | Correlated multivariate data | Very large datasets |
| Time-Series Fill | Time-indexed data | Non-temporal data |
| Model-Based | Important features | Low-resource settings |

Excellent question! **Class imbalance** is a common and serious issue, especially in **classification problems** like fraud detection, disease prediction, or rare event modeling.

## ⚖️ What Is Class Imbalance?

Class imbalance occurs when the number of samples in one class is **much higher** than in another.

Example: In a dataset for disease detection

* Class 0 (Healthy): 950 samples
* Class 1 (Diseased): 50 samples

This causes models to become biased toward the **majority class**, leading to **poor predictive performance** on the minority class — which is usually more important!

## 📊 How to Deal with Class Imbalance in EDA / Preprocessing

### ✅ ****1. Detect It (EDA)****

Start by inspecting the class distribution:

df['target'].value\_counts(normalize=True) # percentage

You can also visualize:

import seaborn as sns

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

### 🛠️ ****2. Preprocessing Techniques to Handle Imbalance****

#### 🔁 A. **Resampling Techniques**

##### i. **Undersampling (Reduce majority class)**

from imblearn.under\_sampling import RandomUnderSampler

rus = RandomUnderSampler()

X\_res, y\_res = rus.fit\_resample(X, y)

✅ Good when dataset is large  
❌ Risk of losing useful data

##### ii. **Oversampling (Duplicate minority class)**

from imblearn.over\_sampling import RandomOverSampler

ros = RandomOverSampler()

X\_res, y\_res = ros.fit\_resample(X, y)

✅ Good when minority class is small  
❌ Risk of overfitting

##### iii. **SMOTE (Synthetic Minority Over-sampling Technique)**

Generates **synthetic data points** for the minority class.

from imblearn.over\_sampling import SMOTE

smote = SMOTE()

X\_res, y\_res = smote.fit\_resample(X, y)

✅ Smarter than random oversampling  
❌ Doesn’t work well with categorical data

#### 🧠 B. **Use Class Weights in Model**

Some models allow you to give **more importance** to the minority class:

from sklearn.linear\_model import LogisticRegression

model = LogisticRegression(class\_weight='balanced')

✅ Helps avoid resampling  
❌ Might not be as effective as SMOTE for extreme imbalance

#### 📊 C. **Use Appropriate Metrics**

Avoid just relying on accuracy.

Instead, use:

* **Precision / Recall / F1-score**
* **ROC-AUC Score**
* **Confusion Matrix**

from sklearn.metrics import classification\_report

print(classification\_report(y\_test, y\_pred))

### ✅ Summary

| **Stage** | **Technique** | **Purpose** |
| --- | --- | --- |
| EDA | value\_counts() / countplot() | Detect imbalance |
| Resampling | Oversampling, Undersampling, SMOTE | Balance the classes |
| Algorithm tweak | class\_weight='balanced' | Penalize misclassification of minority |
| Evaluation | Use F1, ROC, confusion matrix | Avoid misleading accuracy |

Excellent question! **Multicollinearity** is an important issue in **feature selection**, especially when using linear models like **Linear Regression** or **Logistic Regression**.

## 🎯 What is Multicollinearity?

**Multicollinearity** occurs when **two or more independent variables (features) are highly correlated** with each other.

This means one feature can be linearly predicted from others with a high degree of accuracy.

## 🚨 Why Is It a Problem?

While models **can still make predictions**, multicollinearity can:

| **Issue** | **Impact** |
| --- | --- |
| ❌ Inflates standard errors | Leads to **unstable coefficients** |
| ❌ Makes model interpretation difficult | You can’t tell which variable truly impacts the target |
| ❌ Affects feature importance | Confuses the model about which variable to rely on |
| ❌ May cause overfitting | Especially when multicollinearity is extreme |

## 🧪 How to Detect Multicollinearity

### ✅ 1. ****Correlation Matrix****

Plot a heatmap to see how features are correlated with each other.

import seaborn as sns

import matplotlib.pyplot as plt

corr\_matrix = df.corr(numeric\_only=True)

sns.heatmap(corr\_matrix, annot=True, cmap='coolwarm')

plt.title("Correlation Matrix")

plt.show()

* Correlation values **close to +1 or -1** between **independent variables** suggest multicollinearity.

### ✅ 2. ****Variance Inflation Factor (VIF)****

VIF tells how much the variance of a regression coefficient is inflated due to collinearity.

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

from statsmodels.tools.tools import add\_constant

import pandas as pd

X = add\_constant(df) # Add intercept (bias) term

vif = pd.DataFrame()

vif["feature"] = X.columns

vif["VIF"] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]

print(vif)

### 📌 VIF Interpretation:

| **VIF Value** | **Interpretation** |
| --- | --- |
| 1 | No multicollinearity |
| 1–5 | Moderate, usually acceptable |
| **> 5 or 10** | **Problematic** multicollinearity |

## 🧹 How to Handle Multicollinearity

| **Technique** | **What It Does** |
| --- | --- |
| Drop one of the correlated features | Simplifies model |
| Combine features (e.g., BMI = weight/height^2) | Create new meaningful variables |
| Use PCA (Principal Component Analysis) | Reduces dimensionality while preserving variance |
| Use models not sensitive to multicollinearity | e.g., tree-based models like Random Forest |

## ✅ Summary

| **Concept** | **Explanation** |
| --- | --- |
| Multicollinearity | High correlation between independent variables |
| Detected via | Correlation matrix, VIF |
| Problem for | Linear/logistic regression models |
| Fixed by | Dropping, combining features, or using PCA |

## 🎯 Why Do We Need These?

In **imbalanced datasets**, one class (majority) dominates the other (minority).  
This can lead to models that:

* Predict the **majority class too often**
* Fail to **recognize or learn from the minority class** (which is often more important)

## 🔁 Resampling Strategies:

### 🔻 1. ****Undersampling****

**Reduce the number of samples in the majority class** to match the minority class.

#### 📌 Example:

| **Class** | **Count (Before)** | **Count (After Undersampling)** |
| --- | --- | --- |
| 0 | 900 | 100 |
| 1 | 100 | 100 |

#### ✅ Pros:

* Faster training
* Balanced dataset

#### ❌ Cons:

* You **lose valuable data**
* Might miss patterns in majority class

#### 🧪 Code:

from imblearn.under\_sampling import RandomUnderSampler

rus = RandomUnderSampler()

X\_resampled, y\_resampled = rus.fit\_resample(X, y)

### 🔺 2. ****Oversampling****

**Duplicate or generate new samples** for the minority class to match the majority class.

#### 📌 Example:

| **Class** | **Count (Before)** | **Count (After Oversampling)** |
| --- | --- | --- |
| 0 | 900 | 900 |
| 1 | 100 | 900 |

#### ✅ Pros:

* Keeps all data from majority class
* Gives model more chance to learn from minority

#### ❌ Cons:

* **Risk of overfitting** (if simply duplicating)
* Increases training time

#### 🧪 Code:

from imblearn.over\_sampling import RandomOverSampler

ros = RandomOverSampler()

X\_resampled, y\_resampled = ros.fit\_resample(X, y)

### 🤖 Advanced Oversampling: ****SMOTE****

Instead of duplicating, it **generates synthetic samples** based on existing ones.

from imblearn.over\_sampling import SMOTE

smote = SMOTE()

X\_resampled, y\_resampled = smote.fit\_resample(X, y)

## ✅ Summary:

| **Method** | **How it works** | **Best When** |
| --- | --- | --- |
| Undersampling | Removes data from majority class | Majority class is huge |
| Oversampling | Adds copies or synthetic data to minority | Minority class is small |
| SMOTE | Adds synthetic minority samples | Better than just duplicating |

### ❌ Effects of Multicollinearity:

#### 1. **Regression coefficients become unstable**

* In **linear/logistic regression**, the model struggles to assign meaningful weights.
* Coefficients can become **very large** or **change signs** with small changes in data.

#### 2. **Interpretability is lost**

* You can't trust the **importance** or **sign** of a feature.
* Model might assign credit/blame to the wrong category.

#### 3. **Model performance can degrade**

* While the predictions may not always be wrong, the **training process becomes inefficient**, and the model is more prone to **overfitting**.

## 🔍 How to Detect Multicollinearity

### 1. ****Correlation Matrix****

import seaborn as sns

import matplotlib.pyplot as plt

sns.heatmap(df.corr(), annot=True, cmap='coolwarm')

* Helps identify **high correlation** (near ±1) between numeric variables.

### 2. ****Variance Inflation Factor (VIF)****

from statsmodels.stats.outliers\_influence import variance\_inflation\_factor

from statsmodels.tools.tools import add\_constant

X = add\_constant(df)

vif = pd.DataFrame()

vif["feature"] = X.columns

vif["VIF"] = [variance\_inflation\_factor(X.values, i) for i in range(X.shape[1])]

* If VIF > **5 or 10**, multicollinearity is high.

## 🛠️ Methods to Handle Multicollinearity

### ✅ 1. ****Drop One of the Correlated Features****

* **Simplest method** when two features are highly correlated.

df.drop('one\_of\_the\_columns', axis=1, inplace=True)

### ✅ 2. ****Use**** drop\_first=True ****in One-Hot Encoding****

* Prevents dummy variable trap when encoding categoricals.

pd.get\_dummies(df, drop\_first=True)

### ✅ 3. ****Combine Correlated Features (Feature Engineering)****

* Create a single feature using correlated ones.

df['combined'] = df['feature1'] + df['feature2']

df.drop(['feature1', 'feature2'], axis=1, inplace=True)

### ✅ 4. ****Principal Component Analysis (PCA)****

* Transforms correlated features into uncorrelated **principal components**.

from sklearn.decomposition import PCA

pca = PCA(n\_components=2)

X\_pca = pca.fit\_transform(df)

* Useful when you want to **retain information** while reducing dimensionality.

### ✅ 5. ****Ridge Regression (L2 Regularization)****

* Shrinks coefficients; **penalizes large weights**.

from sklearn.linear\_model import Ridge

model = Ridge(alpha=1.0)

### ✅ 6. ****Lasso Regression (L1 Regularization)****

* Can **zero out** some coefficients → helps in feature selection too.

from sklearn.linear\_model import Lasso

model = Lasso(alpha=0.1)

### ✅ 7. ****Use Tree-Based Models****

* Models like Random Forest, XGBoost, or Decision Trees are **not affected** by multicollinearity — they select features automatically.

## 🧠 Summary Table:

| **Method** | **When to Use** |
| --- | --- |
| Drop a feature | Simple correlation between 2+ vars |
| drop\_first=True | Handling categorical vars |
| Combine features | Logical or domain knowledge-based |
| PCA | Many correlated features |
| Ridge/Lasso | Regression tasks with collinearity |
| Tree-based models | Avoid collinearity issues entirely |

## 🧠 What is Feature Engineering?

**Feature Engineering** is the process of **creating, transforming, or selecting features (columns)** in your dataset to help machine learning models perform better.

In short: It's making your data smarter.

## 🎯 Why Feature Engineering?

Because **better features = better models**.

Even if you use a simple algorithm, **smart features** can drastically improve accuracy, speed, and interpretability.

## 🔧 What Can You Do in Feature Engineering?

### 1️⃣ ****Create New Features****

You derive new features from existing ones:

| **Example** | **New Feature** |
| --- | --- |
| year of vehicle | vehicle\_age = current\_year - year |
| mileage and vehicle\_age | mileage\_per\_year = mileage / age |
| name like "2021 Ford F150" | brand = 'Ford' |

### 2️⃣ ****Transform Existing Features****

| **Original** | **Transformation** |
| --- | --- |
| Skewed price | Apply log(price) |
| Long text | Extract length or keyword presence |
| Categories | Apply encoding (label, one-hot) |

### 3️⃣ ****Group or Bin Features****

| **Column** | **New Feature** |
| --- | --- |
| price | price\_range = High / Medium / Low |
| mileage | Bucketed into groups (e.g., low, medium, high) |

### 4️⃣ ****Handle Outliers or Missing Data****

Feature engineering may also involve:

* Imputing missing values
* Replacing outliers with group medians

### 5️⃣ ****Combine Features****

Sometimes, you combine multiple features to extract patterns:

df['price\_per\_mile'] = df['price'] / df['mileage']

## 🛠 Example in Vehicle Dataset:

| **Raw Feature** | **Engineered Feature** |
| --- | --- |
| year | vehicle\_age |
| model | model\_frequency |
| mileage | mileage\_per\_year |
| name | brand, variant |
| description | desc\_length |

## ✅ Why It Matters:

| **Feature Engineering Helps...** |
| --- |
| Reveal hidden patterns |
| Improve model performance |
| Reduce overfitting |
| Work better with linear models |

### 📌 Summary:

**Feature engineering** is about **adding intelligence to your data** before you train a machine learning model.

It’s often where the **real magic happens** in data science! 🌟

## 🔍 Step 1: What Are Outliers?

**Outliers** are extreme values that are significantly different from most other values.

| **Example:** |
| --- |
| Vehicle with 1,000,000 miles 🚗 |
| Price = $1,000,000 for a Honda Civic 💸 |
| Mileage = 0 on a 10-year-old car 🕙 |

Outliers can:

* Skew model results
* Mislead insights
* Affect regression and distance-based models

## ✅ Step 2: Common Outlier Detection Methods

We'll use:

| **Method** | **Description** |
| --- | --- |
| **Boxplot** | Visual detection using IQR |
| **IQR** | Statistical rule (Q1/Q3 range) |
| **Z-Score** | Standard deviation-based |
| **Isolation Forest** | ML-based (for large data) |

### 🧪 Let's Start with IQR Method (Most Common)

# Select only numeric columns

numeric\_cols = df.select\_dtypes(include=['int64', 'float64']).columns.tolist()

# Create a copy to work on

df\_iqr = df.copy()

# Detect and remove outliers using IQR

for col in numeric\_cols:

Q1 = df\_iqr[col].quantile(0.25)

Q3 = df\_iqr[col].quantile(0.75)

IQR = Q3 - Q1

lower = Q1 - 1.5 \* IQR

upper = Q3 + 1.5 \* IQR

# Filter values within range

df\_iqr = df\_iqr[(df\_iqr[col] >= lower) & (df\_iqr[col] <= upper)]

print("Shape after IQR-based outlier removal:", df\_iqr.shape)

### 📈 Optional: Visual Boxplots Before and After

import seaborn as sns

import matplotlib.pyplot as plt

for col in numeric\_cols:

plt.figure(figsize=(10, 2))

sns.boxplot(x=df[col])

plt.title(f"Boxplot Before Removing Outliers: {col}")

plt.show()

plt.figure(figsize=(10, 2))

sns.boxplot(x=df\_iqr[col])

plt.title(f"Boxplot After Removing Outliers: {col}")

plt.show()

### 🧮 Z-Score Method

from scipy.stats import zscore

import numpy as np

df\_z = df.copy()

# Calculate Z-score and remove rows with |z| > 3

for col in numeric\_cols:

z = np.abs(zscore(df\_z[col]))

df\_z = df\_z[(z < 3)]

print("Shape after Z-score based removal:", df\_z.shape)

### 🤖 Isolation Forest (For large datasets)

from sklearn.ensemble import IsolationForest

iso = IsolationForest(contamination=0.05, random\_state=42)

outlier\_flag = iso.fit\_predict(df[numeric\_cols])

df\_iso = df.copy()

df\_iso['outlier'] = outlier\_flag

# -1 = outlier, 1 = normal

df\_iso = df\_iso[df\_iso['outlier'] == 1].drop(columns='outlier')

print("Shape after Isolation Forest outlier removal:", df\_iso.shape)

## 🧹 Step 3: Handling Outliers

You can either:

| **Option** | **Description** |
| --- | --- |
| 🗑 Drop | Remove outlier rows (as done above) |
| 🧮 Cap | Replace with min/max percentile |
| ⚖️ Transform | Use log/sqrt to reduce impact |

### Example: Capping Outliers (a.k.a. Winsorizing)

# Cap outliers at 5th and 95th percentiles

for col in numeric\_cols:

lower = df[col].quantile(0.05)

upper = df[col].quantile(0.95)

df[col] = np.clip(df[col], lower, upper)

## ✅ Summary

| **You’ve now covered:** |
| --- |
| ✅ Outlier detection (IQR, Z-score, ML) |
| ✅ Visualization with boxplots |
| ✅ Outlier handling (removal & capping) |

# 📘 **Outlier Detection and Handling – Theoretical Notes**

## 🔍 What is an Outlier?

An **outlier** is a data point that is **significantly different** from other observations in a dataset.  
It **deviates from the overall pattern**, and may indicate variability, error, or rare event.

🎯 **Definition**:  
An outlier is an observation that lies an abnormal distance from other values in a dataset.

## 🎯 Why Detect and Handle Outliers?

Outliers can:

* Skew the **mean**, **standard deviation**, and **model performance**
* Mislead **visualizations** and **insights**
* Lead to **overfitting** or **biased models**

## 🧠 Causes of Outliers

* **Data entry errors** (e.g., extra zeros)
* **Measurement error** (e.g., faulty sensors)
* **Natural variation** (e.g., luxury car prices)
* **Rare events** (e.g., limited edition vehicles)

## 📊 Examples of Outliers

| **Example** | **Why it's an outlier** |
| --- | --- |
| A car with 1,000,000 miles | Unusually high mileage |
| A 5-year-old car priced at $5 | Data entry error |
| A brand-new Ferrari for $2M | Rare luxury outlier |

## 🚦 Outlier Detection Methods

### 1️⃣ ****Boxplot Method / IQR Rule (Interquartile Range)****

* **Definition**: Based on the spread of the middle 50% of the data.
* **Formula**:
  + IQR = Q3 – Q1
  + Outliers are points below Q1 – 1.5×IQR or above Q3 + 1.5×IQR

🔸 Example:  
If Q1 = 10000, Q3 = 30000  
→ IQR = 20000  
→ Lower Bound = 10000 – (1.5×20000) = –20000  
→ Upper Bound = 30000 + (1.5×20000) = 60000  
→ Any price > 60,000 is an outlier

### 2️⃣ ****Z-Score Method****

* **Definition**: Measures how many standard deviations a data point is from the mean.
* **Formula**:

Z=x−μσZ = \frac{x - \mu}{\sigma}

* If |Z| > 3 → outlier

🔸 Example:  
If price = 80,000, mean = 30,000, std = 10,000  
→ Z = (80000 – 30000) / 10000 = 5 → outlier

### 3️⃣ ****Percentile / Capping****

* Use fixed **upper and lower percentile thresholds** (e.g., 5th, 95th percentile).
* Replace extreme values with percentile cutoffs (known as **Winsorizing**).

### 4️⃣ ****Isolation Forest (ML-based)****

* An **unsupervised machine learning algorithm**.
* Detects anomalies by isolating observations in random trees.
* Effective for **large, high-dimensional** datasets.

## 🧹 Outlier Handling Methods

| **Method** | **Description** |
| --- | --- |
| ❌ **Remove** | Delete rows with outliers |
| ⚖️ **Cap (Clip)** | Limit values to a threshold or percentile |
| 🔁 **Impute** | Replace with mean/median (not ideal for extreme values) |
| 🔁 **Transform** | Apply log, sqrt, Box-Cox to reduce skew |

## ✅ Example in Python (IQR-based Removal)

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

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

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

df\_filtered = df[(df['price'] >= lower\_bound) & (df['price'] <= upper\_bound)]

## 📌 Summary

| **Topic** | **Summary** |
| --- | --- |
| What is an Outlier? | Data point far from most others |
| Why handle it? | Improves model performance & data quality |
| Detection Methods | IQR, Z-Score, Percentile, Isolation Forest |
| Handling Methods | Remove, Cap, Transform, Impute |
| Key Tip | Always explore **why** a value is an outlier |

# 📊 **EDA Plots – Definitions and Insights**

## 1️⃣ **Histogram**

### ✅ ****Definition****:

A histogram is a **distribution plot** that shows the **frequency** of a numeric feature across intervals (bins).

### 🔍 ****What it tells you****:

* The **shape** of the data (normal, skewed, bimodal, etc.)
* Where values are **concentrated**
* Presence of **outliers** or **gaps**

### 📘 Example Insight:

* If price histogram is **right-skewed**, most vehicles are **lower-priced**, and a few luxury ones stretch the tail.

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

## 2️⃣ **Boxplot**

### ✅ ****Definition****:

A boxplot displays the **distribution** of a variable using its **quartiles**, highlighting **outliers**.

### 🔍 ****What it tells you****:

* **Median**, **Q1**, **Q3**
* Range of typical values
* Any **extreme outliers**

### 📘 Example Insight:

* A boxplot of mileage may show a few cars with abnormally high mileage — easy outlier detection.

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

## 3️⃣ **Countplot**

### ✅ ****Definition****:

A bar plot that shows **counts** of each unique category in a **categorical feature**.

### 🔍 ****What it tells you****:

* **Frequency distribution** of categories
* Imbalanced features (e.g., more Automatics than Manuals)

### 📘 Example Insight:

* A countplot of body may show SUV dominates your dataset.

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

## 4️⃣ **Bar Plot (Grouped Mean/Count)**

### ✅ ****Definition****:

A plot that shows **aggregate values** (like mean, count) for each category.

### 🔍 ****What it tells you****:

* Compare **average price** by make or fuel type
* Group-wise patterns

### 📘 Example Insight:

* Diesel cars may have a higher average price than Petrol.

sns.barplot(x='fuel', y='price', data=df)

## 5️⃣ **Scatter Plot**

### ✅ ****Definition****:

A plot showing the **relationship between two continuous variables**.

### 🔍 ****What it tells you****:

* Is there a **positive or negative correlation**?
* Are there **clusters** or **trends**?
* Spot **outliers** visually

### 📘 Example Insight:

* A scatterplot of mileage vs price may show that higher mileage leads to lower price.

sns.scatterplot(x='mileage', y='price', data=df)

## 6️⃣ **Pairplot**

### ✅ ****Definition****:

A grid of scatterplots for **pairwise comparisons** between features, along with **histograms** on the diagonal.

### 🔍 ****What it tells you****:

* Relationships across many features at once
* Clusters, patterns, outliers

sns.pairplot(df[numeric\_cols])

## 7️⃣ **Heatmap (Correlation Matrix)**

### ✅ ****Definition****:

A color-coded matrix that shows **correlation coefficients** between numeric variables.

### 🔍 ****What it tells you****:

* Which variables are **strongly correlated** (+ or –)
* Identify **multicollinearity** (important for regression)

### 📘 Example Insight:

* If vehicle\_age and price have strong negative correlation → Older cars are cheaper.

corr = df.corr()

sns.heatmap(corr, annot=True, cmap='coolwarm')

## 8️⃣ **KDE Plot (Kernel Density Estimate)**

### ✅ ****Definition****:

A **smoothed curve** version of a histogram that estimates the probability distribution.

### 🔍 ****What it tells you****:

* More elegant view of **distribution shape**
* Useful to compare **multiple groups**

sns.kdeplot(df['price'], shade=True)

## Summary Table:

| **📊 Plot** | **🧠 What it Shows** | **🧾 Use Case Example** |
| --- | --- | --- |
| Histogram | Distribution, skewness, peaks | Price or mileage spread |
| Boxplot | Quartiles, IQR, outliers | Outlier detection for numeric features |
| Countplot | Frequency of categories | Most common vehicle body type |
| Barplot | Mean/aggregate per category | Avg price per make |
| Scatter Plot | Relationship between 2 variables | Mileage vs Price |
| Pairplot | Pairwise numeric relationships | Multiple regression prep |
| Heatmap | Correlation matrix | Check multicollinearity |
| KDE Plot | Smoothed distribution curves | Price shape comparison across groups |

## 🧠 What is Vectorization?

**Vectorization** is the process of **rewriting code to use vector or array operations instead of loops**. It allows you to **perform operations on entire arrays or matrices at once**, instead of element by element.

### 🚀 Why Use Vectorization?

| **Benefit** | **Explanation** |
| --- | --- |
| ✅ **Speed** | Vectorized operations are **much faster** than loops |
| ✅ **Cleaner Code** | Fewer lines of code, easier to read |
| ✅ **Leverages Hardware** | Takes advantage of **SIMD** (Single Instruction, Multiple Data) and **NumPy/Pandas optimizations** |

### 🧪 Example: Without vs With Vectorization (NumPy)

#### ❌ Without Vectorization (Using Loops):

import numpy as np

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

b = np.array([4, 5, 6])

result = []

for i in range(len(a)):

result.append(a[i] + b[i])

print(result) # Output: [5, 7, 9]

#### ✅ With Vectorization:

import numpy as np

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

b = np.array([4, 5, 6])

result = a + b

print(result) # Output: [5 7 9]

## 📊 Vectorization in Machine Learning

In ML, vectorization helps speed up:

* **Feature transformations**
* **Matrix multiplications**
* **Gradient calculations**

For example, instead of calculating each sample's prediction manually in logistic regression, we use:

y\_pred = sigmoid(np.dot(X, W) + b)

This line computes predictions for **all samples at once** using vectorized operations.

## 🧠 Vectorization vs. Looping (Performance)

Let’s compare in terms of speed:

| **Approach** | **Speed** | **Readability** | **Scalability** |
| --- | --- | --- | --- |
| Loops | Slower | Okay | Poor |
| Vectorized Code | Much faster | Clean | Excellent |

## 🛠 Vectorization Tools in Python

| **Library** | **Use Case** |
| --- | --- |
| **NumPy** | Array/matrix math |
| **Pandas** | Series/DataFrame operations |
| **TensorFlow / PyTorch** | Deep learning models |
| **Sklearn** | ML pipelines, models |

### 🎯 Summary

* **Vectorization** = replacing loops with efficient array operations
* Makes your code **faster**, **cleaner**, and **more efficient**
* It's a key concept in **numerical computing** and **machine learning**

In the context of **EDA (Exploratory Data Analysis)**, the concept of **performance** typically refers to how the variables in your dataset relate to a **target outcome** — especially in tasks involving **classification, regression, or predictive modeling**.

Let’s break it down:

## ⚡ What Does "Performance" Mean in EDA?

In **EDA**, **performance** relates to:

🔎 **How well certain features (columns) influence or predict the outcome variable** (e.g., sales, price, success, etc.)

It's about analyzing:

* Which variables are **strong predictors**
* How different **classes or values** perform against the target
* Where there are **outliers**, **imbalances**, or **patterns** that affect model quality

## ✅ Examples of Performance Analysis in EDA

### 1. 📊 ****Target Variable Distribution****

Understanding how your outcome variable is distributed.

* For classification: Are classes balanced?
* For regression: Is the target skewed?

### 2. 📈 ****Feature vs Target Analysis****

Understanding how each input variable affects the target.

* **Box plots**: e.g., Income vs Purchase amount
* **Groupby mean**: e.g., Average test score per education level
* **Correlation heatmap**: To find strong linear relations

### 3. 🚨 ****Outlier Impact****

Extreme values can affect model performance:

* Use plots like box plots, scatter plots
* Use Z-score or IQR to detect and decide whether to remove or transform

### 4. 🏷️ ****Feature Importance Estimation (Optional Pre-Modeling)****

Using techniques like:

* RandomForestClassifier().feature\_importances\_
* chi2, mutual\_info\_classif from sklearn

To see which features impact the target most.

## 🔬 Real Example: Vehicle Price Prediction

In EDA for predicting vehicle price:

* **Performance perspective** checks:
  + How does engine\_size, fuel\_type, or brand affect price?
  + Are high-priced cars from specific brands only?
  + Does horsepower linearly relate to price?

import seaborn as sns

sns.scatterplot(data=df, x='engine\_size', y='price')

This helps evaluate the **performance relationship** between engine\_size and price.

## ⚖️ Performance ≠ Model Metrics (But Related)

* During EDA, you're **not calculating accuracy or RMSE** — that comes later in modeling.
* But you **analyze what might help improve model performance**:
  + Check **feature correlation**
  + Spot **data quality issues**
  + Understand **variance** and **bias** in variables

## 🧠 Summary: Performance in EDA Involves

| **Task** | **Purpose** |
| --- | --- |
| Analyze variable relationships | Find strong/weak predictors |
| Detect outliers and skew | Avoid model distortion |
| Check feature-target trends | Inform model feature selection |
| Assess class balance | Prevent bias in classification models |

## ⚙️ What is Optimization in EDA?

While EDA is mostly about **exploration**, not **training or tuning models**, **optimization in EDA** refers to:

🔍 **Making your data cleaner, more informative, and ready for efficient modeling** by improving how features are structured, transformed, and selected.

### 🧠 Think of it this way:

You're not optimizing a model, you're optimizing the **data** to make your future model perform better.

## ✅ Key Optimization Concepts in EDA

### 1. 🧹 ****Data Cleaning Optimization****

* Remove unnecessary columns (ID, unnamed columns)
* Handle missing values smartly (drop, fill with median/mean/mode, or model-based imputation)
* Remove or cap **outliers** (e.g., using IQR or z-score)

### 2. 🧠 ****Feature Optimization****

| **Action** | **Optimization Goal** |
| --- | --- |
| **Feature selection** | Drop irrelevant/redundant columns |
| **Feature engineering** | Create more meaningful columns (e.g., age bins) |
| **Encoding categories** | Use label encoding or one-hot efficiently |
| **Scaling** | Normalize or standardize numerical features |

### 3. 🧮 ****Dimensionality Reduction****

Reduce number of features to make data simpler and faster to process.

* Techniques: PCA (Principal Component Analysis), t-SNE (for visualization)
* Example: Reducing 1000 features from text/NLP to 100 using TF-IDF + PCA

### 4. 📊 ****Data Distribution Optimization****

* Transform **skewed** data (e.g., log/sqrt transformations)
* Normalize ranges (especially for ML models that assume normality)

import numpy as np

df['income'] = np.log1p(df['income']) # Log transform for right-skewed data

### 5. 🔄 ****Memory Optimization****

In large datasets, reducing memory usage matters.

df['column'] = df['column'].astype('int32') # reduce from int64

Or downcast numeric types using pd.to\_numeric(..., downcast='integer').

## 💡 Example: Optimizing a Titanic Dataset for Modeling

| **Raw Data Step** | **Optimization Step** |
| --- | --- |
| Cabin mostly null | Drop it |
| Age has missing values | Fill with median |
| Fare is right-skewed | Apply log1p() |
| Sex is categorical | Use LabelEncoder or OneHotEncoder |
| Name, Ticket are noisy | Drop or extract useful parts like title |

## 🧠 Summary

**Optimization in EDA** means:

Preparing and transforming your dataset so it's cleaner, smaller, and more useful for analysis and modeling.

### 🔁 EDA Optimization Checklist

✅ Remove useless features  
✅ Handle missing values smartly  
✅ Encode categorical variables  
✅ Normalize / scale where needed  
✅ Eliminate outliers  
✅ Reduce skewness  
✅ Reduce memory usage if needed

**✅ Goal:**

If two or more rows have the **same make and model**, and some are missing price, fill the missing price with the **average or median** price of that make-model combination.

**💡 Step-by-Step Strategy**

# Step 1: Group by make and model and compute the median price

model\_price\_map = df.groupby(['make', 'model'])['price'].median()

# Step 2: Define a function to fill missing price based on group

def fill\_price(row):

if pd.isnull(row['price']):

return model\_price\_map.get((row['make'], row['model']), np.nan)

else:

return row['price']

# Step 3: Apply the function row-wise

df['price'] = df.apply(fill\_price, axis=1)

**✅ Explanation:**

* groupby(['make', 'model'])['price'].median() computes the median price per make-model group.
* .get((row['make'], row['model']), np.nan) fetches the median for that group, or leaves it as NaN if no match is found.
* apply() is used to go row-by-row and impute missing values.

**Optional: Still Missing?**

After this, if you want to **drop the remaining** rows that couldn’t be filled:

df = df.dropna(subset=['price'])

#### Optional: Drop partial duplicates (if needed):

If you think vehicles with same name, year, and mileage are essentially the same listing, try:

python

CopyEdit

df = df.drop\_duplicates(subset=['name', 'year', 'mileage'])

This helps clean up cases where the same car may be listed twice but has different colors or descriptions.

**✅ 🔧 Encoding Code (Ready for Jupyter Notebook)**

import pandas as pd

from sklearn.preprocessing import LabelEncoder

# Assuming your cleaned DataFrame is named df

# 1️⃣ Label Encoding for low-cardinality categorical features

label\_cols = ['make', 'trim', 'interior\_color', 'exterior\_color']

le = LabelEncoder()

for col in label\_cols:

df[col] = le.fit\_transform(df[col])

# 2️⃣ Frequency Encoding for high-cardinality column like 'model'

model\_freq = df['model'].value\_counts()

df['model\_encoded'] = df['model'].map(model\_freq)

# Optional: drop the original 'model' column if no longer needed

df.drop(columns=['model'], inplace=True)

# 3️⃣ One-Hot Encoding for nominal categorical features

one\_hot\_cols = ['fuel', 'transmission', 'body', 'drivetrain']

df = pd.get\_dummies(df, columns=one\_hot\_cols, drop\_first=True)

# ✅ Done: Encoded DataFrame is ready

df.head()

**🧠 Notes:**

* drop\_first=True in pd.get\_dummies() avoids the dummy variable trap (good for linear models).
* You can choose to keep or drop name, description, and engine depending on whether you’ll process them further (like TF-IDF or dropping).
* You can rename the encoded DataFrame or export it if needed.

Let me know if you want the code to:

* Export this as a new CSV
* Visualize encoding effects
* Start model training afterward

Now that your data is encoded and clean, let's move to **Feature Engineering** — a crucial step to boost your model's predictive power.

### ✅ Key Feature Engineering Ideas for Vehicle Price Prediction

Here are meaningful, domain-specific features you can engineer from your current columns:

### 1️⃣ ****Age of Vehicle****

Older vehicles generally have lower prices.

from datetime import datetime

current\_year = datetime.now().year

df['vehicle\_age'] = current\_year - df['year']

### 2️⃣ ****Mileage per Year****

Gives insight into how heavily a car has been used.

df['mileage\_per\_year'] = df['mileage'] / df['vehicle\_age'].replace(0, 1) # avoid division by zero

### 3️⃣ ****Is New****

If mileage is near 0 and age is 0, it’s likely brand new.

df['is\_new'] = ((df['vehicle\_age'] == 0) & (df['mileage'] < 100)).astype(int)

### 4️⃣ ****Cylinders Category****

Binning cylinders into categories (e.g., 4, 6, 8+).

df['cyl\_category'] = pd.cut(df['cylinders'], bins=[0, 4, 6, 12], labels=['Low', 'Medium', 'High'])

### 5️⃣ ****Luxury Brands****

Add a binary flag for premium makes.

luxury\_brands = ['BMW', 'Mercedes-Benz', 'Audi', 'Lexus', 'Tesla']

df['is\_luxury'] = df['make'].apply(lambda x: 1 if x in luxury\_brands else 0)

⚠️ If make is already label encoded, you'll need a reverse mapping to get brand names.

### 6️⃣ ****Trim Simplification****

If trim has lots of unique values, simplify to common keywords.

df['trim\_level'] = df['trim'].astype(str).str.extract(r'(\bLX|EX|Sport|Limited|Laramie|Denali|Platinum|XLT\b)', expand=False).fillna('Other')

### ✅ Optional: Drop Less Useful Columns

These are mostly text-based or IDs:

df.drop(columns=['name', 'description', 'engine'], inplace=True)

### ✅ Final Touch: Convert New Categorical Columns

df = pd.get\_dummies(df, columns=['cyl\_category', 'trim\_level'], drop\_first=True)

**✅ Why Scaling?**

Scaling is crucial for models that are sensitive to feature magnitudes, like:

* **Linear Regression**
* **KNN**
* **SVM**
* **Gradient Descent–based models**

Not strictly necessary for:

* **Tree-based models** (Random Forest, XGBoost, LightGBM)

**🎯 Common Scalers**

| **Scaler** | **Use Case** | **Description** |
| --- | --- | --- |
| StandardScaler | Most common | Mean = 0, Std = 1 |
| MinMaxScaler | Range-based models (e.g., Neural Nets) | Scales between 0 and 1 |
| RobustScaler | Outlier-resistant | Uses median and IQR |

**✅ Let’s Scale Numeric Features**

Here’s code for **Standard Scaling** (the most commonly used):

from sklearn.preprocessing import StandardScaler

# Select only numeric features for scaling

numeric\_cols = df.select\_dtypes(include=['int64', 'float64']).columns.tolist()

# Optionally exclude the target column (e.g., 'price') from scaling

numeric\_cols.remove('price')

# Initialize and fit the scaler

scaler = StandardScaler()

df[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

# ✅ Done: Scaled features are now normalized

df.head()

**🔄 If You Prefer MinMaxScaler:**

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[numeric\_cols] = scaler.fit\_transform(df[numeric\_cols])

Here's a full set of EDA visualization code blocks for your **Jupyter Notebook**, covering:

* ✅ Univariate Analysis
* ✅ Bivariate Analysis
* ✅ Multivariate Analysis

We'll use **Matplotlib**, **Seaborn**, and **Pandas**.

## 📦 Step 1: Import Libraries

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

# Optional: for cleaner plots

sns.set(style="whitegrid")

%matplotlib inline

## 📊 1️⃣ Univariate Analysis

### 🔸 Numeric Columns

# Histogram of price

plt.figure(figsize=(8, 5))

sns.histplot(df['price'], bins=30, kde=True)

plt.title('Distribution of Vehicle Prices')

plt.xlabel('Price')

plt.ylabel('Count')

plt.show()

### 🔸 Categorical Columns

# Count plot of body type (if not one-hot encoded already)

plt.figure(figsize=(10, 5))

sns.countplot(x='body', data=df\_original) # use original if you dropped this in encoding

plt.title('Count of Vehicles by Body Type')

plt.xticks(rotation=45)

plt.show()

📝 If you have one-hot encoded body types, you can do .sum() to visualize count:

df.filter(like='body\_').sum().sort\_values().plot(kind='barh', figsize=(8,5), title="Count of Body Types (One-hot)")

## 📊 2️⃣ Bivariate Analysis

### 🔹 Price vs Categorical (Boxplot)

plt.figure(figsize=(10, 5))

sns.boxplot(x='fuel', y='price', data=df\_original) # if fuel wasn't one-hot encoded

plt.title('Price by Fuel Type')

plt.xticks(rotation=45)

plt.show()

### 🔹 Price vs Numeric (Scatterplot)

plt.figure(figsize=(8, 5))

sns.scatterplot(x='mileage', y='price', data=df)

plt.title('Price vs Mileage')

plt.xlabel('Mileage')

plt.ylabel('Price')

plt.show()

## 📊 3️⃣ Multivariate Analysis

### 🔸 Correlation Heatmap

plt.figure(figsize=(14, 10))

corr = df.corr()

sns.heatmap(corr, annot=False, cmap='coolwarm', linewidths=0.5)

plt.title('Correlation Heatmap')

plt.show()

### 🔸 Pairplot (subset of features)

sns.pairplot(df[['price', 'mileage', 'vehicle\_age', 'cylinders']])

plt.suptitle('Pairplot of Selected Features', y=1.02)

plt.show()

### ✅ Optional: Grouped Barplot

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

grouped = df\_original.groupby('make')['price'].mean().sort\_values(ascending=False).head(10)

sns.barplot(x=grouped.index, y=grouped.values)

plt.xticks(rotation=45)

plt.title('Top 10 Car Makes by Average Price')

plt.ylabel('Average Price')

plt.show()

Let’s now perform **outlier detection** and **skewness analysis** — these steps help you improve data quality and model performance, especially for regression tasks like vehicle price prediction.

## 🔍 Step 1: Outlier Detection (Multiple Methods)

We'll apply **4 methods**:

### 1️⃣ Boxplot (Visual method)

### 2️⃣ IQR (Interquartile Range Rule)

### 3️⃣ Z-Score

### 4️⃣ Isolation Forest (advanced ML-based outlier detection)

## 📦 Prerequisite: Identify numeric columns

# Select only numeric columns

numeric\_cols = df.select\_dtypes(include=['int64', 'float64']).columns.tolist()

## 🔸 1. Boxplot (Visual)

import matplotlib.pyplot as plt

import seaborn as sns

for col in numeric\_cols:

plt.figure(figsize=(6, 2))

sns.boxplot(x=df[col])

plt.title(f'Boxplot for {col}')

plt.show()

## 🔸 2. IQR Method (Programmatic)

# Detect outliers based on IQR for each column

outlier\_iqr = {}

for col in numeric\_cols:

Q1 = df[col].quantile(0.25)

Q3 = df[col].quantile(0.75)

IQR = Q3 - Q1

lower\_bound = Q1 - 1.5 \* IQR

upper\_bound = Q3 + 1.5 \* IQR

outliers = df[(df[col] < lower\_bound) | (df[col] > upper\_bound)]

outlier\_iqr[col] = len(outliers)

# Show number of outliers detected by IQR

pd.Series(outlier\_iqr).sort\_values(ascending=False)

## 🔸 3. Z-Score Method

from scipy.stats import zscore

import numpy as np

z\_outlier\_count = {}

# Z-score threshold

threshold = 3

for col in numeric\_cols:

z\_scores = zscore(df[col])

outliers = df[np.abs(z\_scores) > threshold]

z\_outlier\_count[col] = outliers.shape[0]

# Show Z-score outliers

pd.Series(z\_outlier\_count).sort\_values(ascending=False)

## 🔸 4. Isolation Forest (ML-Based)

from sklearn.ensemble import IsolationForest

iso = IsolationForest(contamination=0.05, random\_state=42)

outliers = iso.fit\_predict(df[numeric\_cols])

# Add a column showing outliers (-1 = outlier, 1 = normal)

df['outlier\_iso'] = outliers

# Count how many outliers were found

df['outlier\_iso'].value\_counts()

## 📈 Step 2: Skewness Check

# Check skewness of each numeric column

skewness = df[numeric\_cols].skew().sort\_values(ascending=False)

print("Highly skewed features (>|1|):\n")

print(skewness[abs(skewness) > 1])

### Optional: Visualize Skewed Distributions

skewed\_cols = skewness[abs(skewness) > 1].index.tolist()

for col in skewed\_cols:

plt.figure(figsize=(6, 3))

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

plt.title(f'Distribution of {col} (Skewness = {skewness[col]:.2f})')

plt.show()

## 💡 Want to Fix Skewed Data?

Use **log or square root transformation**:

df['price\_log'] = np.log1p(df['price']) # log(1 + x) to avoid log(0)

Let me know if you’d like to:

* Remove outliers based on any of the above methods
* Fix highly skewed features
* Move on to model training!

## 🎯 What is Kurtosis?

* **Kurtosis** measures the **tailedness** of a distribution (how extreme the outliers are).
* It complements **skewness** which measures asymmetry.

### 📊 Types of Kurtosis:

| **Type** | **Value Range** | **Description** |
| --- | --- | --- |
| **Mesokurtic** | ≈ 3 | Normal distribution |
| **Leptokurtic** | > 3 | Heavy tails (more extreme outliers) |
| **Platykurtic** | < 3 | Light tails (fewer outliers) |

## ✅ How to Calculate Kurtosis in Your DataFrame

# Import from scipy

from scipy.stats import kurtosis

# Calculate kurtosis for numeric columns

kurt = df[numeric\_cols].kurtosis().sort\_values(ascending=False)

print("Kurtosis of numeric features:\n")

print(kurt)

## ✅ Optional: Highlight Extreme Kurtosis

print("Highly peaked distributions (kurtosis > 3):\n")

print(kurt[kurt > 3])

## 📈 Optional: Visualize High Kurtosis

high\_kurt\_cols = kurt[kurt > 3].index.tolist()

for col in high\_kurt\_cols:

plt.figure(figsize=(6, 3))

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

plt.title(f'{col} (Kurtosis = {kurt[col]:.2f})')

plt.show()

# This is How I pushed my code to already existing git hub :

### ✅ Steps to Add and Push Your New Folder via CMD

Assuming you're already inside your project folder:

cd path\to\Machine-Learning

### 🔹 1. **Check Git status**

git status

This shows you the new untracked EDA/ folder and its files.

### 🔹 2. **Stage the new folder**

git add notebooks/EDA

Or to stage **everything**, just:

git add .

### 🔹 3. **Commit the changes**

git commit -m "Added EDA folder with analysis notebooks"

### 🔹 4. **Push to GitHub**

If you're already connected to a remote GitHub repo:

git push origin main

(replace *main* with *master* or your branch name if different)

### 💡 Optional: If You Haven’t Set the Remote Yet

git remote add origin https://github.com/your-username/Machine-Learning.git

git push -u origin main

### 🧠 Recap of All Commands:

cd path\to\Machine-Learning

git status

git add .

git commit -m "Added EDA folder with analysis notebooks"

git push origin main

Let me know if you want a shortcut script or you're getting any errors during push!

Great — that error is clear and easy to fix!

### ✅ Solution: Initialize Git First

In the folder C:\Users\USER\Documents\Brototype\Brocamp, run:

git init

This command **initializes an empty Git repository** in your local folder by creating a hidden .git directory.

### 🔁 Then proceed with the steps:

#### 1. **Initialize Git**

git init

#### 2. **Add remote origin**

git remote add origin https://github.com/exassaro/Datascience\_Workshop.git

#### 3. **Add your files**

git add .

#### 4. **Commit your changes**

git commit -m "Initial commit for datascience workshop"

#### 5. **Push to GitHub**

git branch -M main

git push -u origin main

### ✅ Result:

Your local folder Brocamp will now be connected to the GitHub repo Datascience\_Workshop, and your files will be uploaded without renaming the local folder.

## 🔄 What is VCS (Version Control System)?

A **Version Control System (VCS)** is a **tool that helps track changes** to files over time. It allows **multiple people** to collaborate on a project and provides a **history** of every change made, by whom, and why.

### 🧠 Why Use a VCS?

* ✅ You can **revert** to earlier versions of a project if needed.
* ✅ Enables **team collaboration** without overwriting others’ work.
* ✅ Helps you track **what changed, when, and by whom**.
* ✅ Encourages **experimentation** using branches.

### 📂 Two Main Types of VCS

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Centralized VCS** | One central server. Everyone commits to it. | SVN (Subversion), CVS |
| **Distributed VCS** | Every user has a full copy of the repository. | **Git**, Mercurial |

**Git is a distributed VCS.**

## 🧩 Core Concepts Related to VCS (especially Git)

### 1. ✅ ****Repository (Repo)****

A project directory tracked by Git. Contains all files and history.

* git init → create a repo
* .git/ folder stores metadata and history

### 2. ✅ ****Commit****

A snapshot of the repo at a point in time.

* git commit -m "message"
* Think of it like saving your progress in a game.

### 3. ✅ ****Staging Area (Index)****

Temporary area to prepare files before committing.

* git add moves changes to the staging area.

### 4. ✅ ****Branch****

A separate line of development. Useful for working on features independently.

* git branch → list branches
* git checkout -b feature-x → create & switch to new branch

### 5. ✅ ****Merge****

Combines changes from one branch into another.

* git merge feature-x
* Brings feature-x into your main branch.

### 6. ✅ ****Conflict****

Occurs when changes in different branches affect the same part of a file.  
Git will ask you to manually resolve it.

### 7. ✅ ****Clone****

Copy a remote repository to your local machine.

* git clone <repo-url>

### 8. ✅ ****Pull / Push****

* git pull: Get updates from remote to local repo
* git push: Send your commits to remote repo (e.g., GitHub)

### 9. ✅ ****Tag****

Mark specific commits (e.g., version releases)

* git tag v1.0

## 🛠 Example Workflow (Simple)

git init # create a repo

git add . # stage all files

git commit -m "Initial commit" # snapshot

git branch feature-x # create new branch

git checkout feature-x # switch to it

# make changes

git add . && git commit -m "Add feature"

git checkout main

git merge feature-x # merge feature into main

git push origin main # push to GitHub

**🧠 What is Git?**

**Git** is a **version control system** — a tool that helps you track, manage, and collaborate on changes to code or any set of files over time.

**🔑 Key Features of Git**

| **Feature** | **Description** |
| --- | --- |
| **Version Control** | Tracks changes you make to files, and lets you go back to previous versions. |
| **Collaboration** | Multiple people can work on the same project without overwriting each other’s work. |
| **Branching & Merging** | You can work on different features or bug fixes in separate branches and merge them when ready. |
| **Distributed System** | Every developer has a full copy of the project history, not just the latest version. |

**📦 Think of Git like...**

Imagine you're writing a book.  
With Git, you can:

* Save versions (like snapshots) of your book.
* Go back to older versions if you mess up.
* Let your friend write another chapter in parallel (in a separate branch).
* Merge your friend's work when it's ready.

**⚙️ Common Git Commands**

| **Command** | **What It Does** |
| --- | --- |
| git init | Start a new Git repo |
| git clone <url> | Download a repo from GitHub |
| git status | Show changes since last commit |
| git add . | Stage changes for commit |
| git commit -m "message" | Save a snapshot (commit) |
| git push | Send changes to remote repo |
| git pull | Bring in updates from remote repo |

**💡 Real-Life Example**

Let’s say you’re building a website:

* Day 1: You create index.html → git add ., git commit -m "Add homepage"
* Day 2: You change the layout → git commit -m "Improve layout"
* You mess up? → You can go back to the previous version.
* Your teammate adds a new feature → Git helps merge their code with yours.

**🧩 Bonus: Git vs GitHub**

| **Git** | **GitHub** |
| --- | --- |
| Local tool | Cloud hosting platform |
| Tracks code | Stores repositories online |
| Open source | Owned by Microsoft |

[githubreponotes](https://github.com/arjunravi26/Learn-Git)

**similar Git commands** that look alike but have **different purposes** — knowing these differences can help avoid common mistakes.

### 🔁 git add . vs git add -A vs git add -u

| **Command** | **What it does** |
| --- | --- |
| git add . | Adds **new and modified** files in the **current directory and subdirs** |
| git add -A | Adds **all new, modified, and deleted** files (from entire repo) |
| git add -u | Adds **modified and deleted** files, but **not new** (untracked) files |

### ✏️ git commit -m vs git commit vs git commit --amend

| **Command** | **Description** |
| --- | --- |
| git commit -m "msg" | Commits staged changes with a message in one line |
| git commit | Opens default editor (like Vim) to write a detailed commit message |
| git commit --amend | Rewrites the **last commit**, letting you change the message or add changes |

### 📥 git pull vs git fetch vs git fetch --all

| **Command** | **Description** |
| --- | --- |
| git pull | **Fetches + merges** from remote into your current branch |
| git fetch | Only **fetches** remote changes — doesn't merge them |
| git fetch --all | Fetches updates from **all remotes**, not just origin |

### 📤 git push vs git push origin main vs git push -f

| **Command** | **Description** |
| --- | --- |
| git push | Pushes changes to the **default remote and branch** |
| git push origin main | Pushes explicitly to the main branch on origin |
| git push -f | **Force pushes** (overwrites remote history) — ⚠️ dangerous if not careful |

### 📄 git log vs git log --oneline vs git log --graph --all

| **Command** | **Description** |
| --- | --- |
| git log | Full commit history with details |
| git log --oneline | Compact view: one commit per line (good for quick look) |
| git log --graph --all | ASCII graph of all branches and commits |

### 🧹 git clean -n vs git clean -f

| **Command** | **Description** |
| --- | --- |
| git clean -n | **Preview** what untracked files would be deleted |
| git clean -f | Actually **removes** untracked files |

Absolutely! Here’s a list of **core Git concepts** with simple explanations to help you understand Git better:

**🔧 1. Version Control System (VCS)**

A tool that records changes to files over time so you can recall specific versions later. Git is a **distributed VCS**.

**🧱 2. Repository (Repo)**

A project folder that Git tracks. It contains all your project files and the .git folder (which stores history and config).

* **Local Repository**: On your machine.
* **Remote Repository**: On a server (like GitHub, GitLab, Bitbucket).

**🌳 3. Commit**

A snapshot of changes. Think of it as a save point in your project.

git commit -m "Your message"

**📌 4. Branch**

A separate line of development. Useful for working on features or fixes without affecting the main code.

git branch feature-login

git checkout feature-login

**🔀 5. Merge**

Combines changes from one branch into another (usually into main or master).

git merge feature-login

**🧪 6. Clone**

Copy an existing remote repository to your local machine.

git clone <repository-url>

**🛠️ 7. Staging Area (Index)**

An intermediate area where Git prepares changes before a commit.

git add filename.py

**🚚 8. Push and Pull**

* **Push**: Upload commits to the remote repo.
* **Pull**: Download and merge changes from remote to local.

git push origin main

git pull origin main

**🧭 9. Checkout**

Switch between branches or restore working files.

git checkout main

**🧾 10. Log**

Shows a history of commits.

git log

**🧽 11. Reset**

Undo commits or changes.

git reset --soft HEAD~1 # Keeps changes

git reset --hard HEAD~1 # Discards changes

**🔁 12. Rebase**

Re-applies commits on top of another base tip. Used for cleaner history.

git rebase main

**🧳 13. Remote**

A link to an external repo (like on GitHub).

git remote add origin <url>

**🚨 14. Conflict**

Occurs when Git can’t automatically merge changes. You must resolve them manually.

**📄 15. .gitignore**

A file listing the files/folders Git should ignore (e.g., .env, node\_modules/).

Sure! Here are some essential **Version Control System (VCS)** concepts that are foundational to understanding Git and other VCS tools like SVN or Mercurial:

**🔁 1. Version Control**

A system that records changes to files over time so you can:

* Revert to earlier versions
* Track changes
* Collaborate with others

**🧱 2. Repository**

A storage space for your project’s files and its version history.

* **Local repository**: Stored on your computer
* **Remote repository**: Stored on a server (e.g., GitHub)

**📌 3. Commit**

A snapshot of your project at a specific point in time. Each commit has a:

* Unique ID (hash)
* Message
* Author and timestamp

**🧪 4. Working Directory**

The directory on your machine where you edit files.

**📥 5. Staging Area (Index)**

An intermediate area where you list files you want to include in the next commit.

**🔀 6. Branch**

A parallel version of your project, allowing you to work on features without affecting the main code.

**🔧 7. Merge**

Combines changes from different branches into one.

**🧭 8. Checkout**

Switches between versions or branches of your codebase.

**🚧 9. Conflict**

Occurs when two versions of the same part of a file are modified and Git can’t auto-merge.

**🔁 10. Rebase**

Moves or reapplies commits on top of another base commit (used to keep history linear).

**🧳 11. Remote**

A reference to a shared repo (usually hosted online like GitHub, GitLab).

**🌍 12. Push & Pull**

* **Push**: Send your local commits to a remote repo.
* **Pull**: Fetch and merge remote commits into your local repo.

**📂 13. Clone**

Creates a local copy of a remote repository.

**🧾 14. Log**

A history of commits made in the repository.

**📄 15. .vcsignore / .gitignore**

Tells the VCS which files/folders to ignore (e.g., build files, temp files).

**🔐 16. Access Control & Permissions**

Defines who can read, write, or manage the repository (especially important in team environments).

**🧠 17. Diff**

Shows the difference between file versions or commits.

git diff

**📅 18. Tag**

Marks specific points in history as important — often used for release versions (e.g., v1.0).