DATA PREPROCESSING

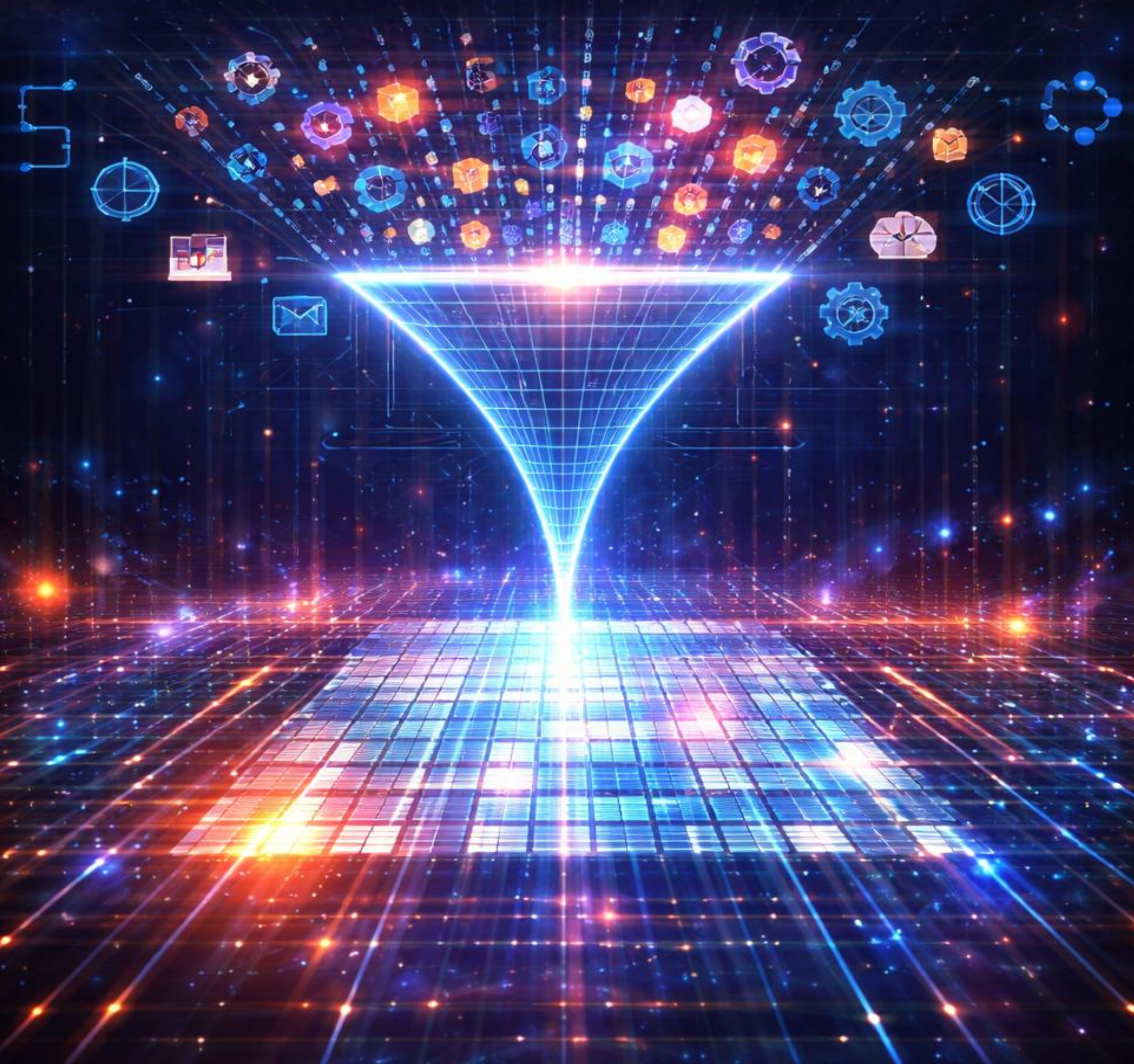

Table of Contents

Introduction

Purpose

The purpose of this document is to provide a clear and detailed roadmap for preparing raw data and applying machine learning models effectively. It aims to bridge the gap between theory and practice by demonstrating the essential preprocessing steps, modeling strategies, and evaluation techniques required to build reliable and interpretable ML solutions.

Scope

This document covers the **end-to-end pipeline** of data preprocessing and machine learning. It spans data understanding, cleaning, encoding, scaling, feature engineering, splitting strategies, handling imbalances, model selection, validation, hyperparameter tuning, and final evaluation. It also includes practical code examples, explanations of when and why to use each technique.

Procedure

The procedure follows a **step-by-step approach**:

1. Understanding data and defining the prediction goal.
2. Cleaning, standardizing, and transforming the dataset.
3. Performing exploratory data analysis (EDA).
4. Handling missing values, duplicates, and outliers.
5. Encoding categorical variables and scaling numerical features.
6. Splitting data into training, validation, and testing sets.
7. Applying techniques to handle imbalanced datasets.
8. Building ML models with pipelines for preprocessing + training.
9. Performing hyperparameter tuning via Grid/Random Search with cross-validation.
10. Evaluating models using multiple metrics and visualizations.

Brief Problem

In real-world scenarios, raw data is rarely clean or directly usable for machine learning. Datasets may contain missing values, duplicates, inconsistent formats, categorical variables, or imbalances that bias predictions. Moreover, choosing the right preprocessing technique, model, and hyperparameters can significantly affect performance. This document addresses these challenges by providing structured guidance and examples, ensuring practitioners can move from messy data to optimized, well-evaluated models.

Data Preprocessing

1. Understanding the Data (Domain Understanding)

Before touching the dataset, the first step in any ML pipeline is to **understand the problem and the data's meaning**. Without this, all preprocessing and modeling will be mechanical, and results may be useless.

What this means

1. **Define the problem clearly**
   - What are we trying to predict?
     - Example: *"Predict whether a customer will churn in the next month"* (Classification).
     - Example: *"Predict house prices based on features"* (Regression).
   - Is this a **supervised problem** (labels available) or **unsupervised** (no labels)?
2. **Understand each feature (column) and its relevance**
   - What does each column represent?
   - Is it directly related to the target?
   - Which features might have high impact, low impact, or be irrelevant?
3. **Research domain knowledge (outside the dataset)**
   - Use **Google, articles, or industry knowledge** to understand which features logically affect the target.
   - Example:
     - If predicting **house price** → important features are location, size, bedrooms, crime rate, and school district.
     - If predicting **heart disease** → relevant features include age, cholesterol, blood pressure, and smoking habits.
4. **Think ahead to feature engineering**
   - Combining related features: e.g., Weight + Height → BMI.
   - Standardizing synonyms: "Los Angeles" vs "LA" → "Los Angeles".
   - Creating interaction features: e.g., Age × Income.

5. **Identify what is noise / unhelpful**
    - Irrelevant features like Customer_ID may not help the prediction.
    - Free text without NLP preprocessing may confuse the model.

Example: Predicting Loan Approval

Suppose we have a dataset about **loan approvals**.

- **Target variable**: Approved (Yes/No)
- **Features**: Age, Income, Credit Score, Loan Amount, Marital Status, City, Education

Domain research tells us:

- Credit Score is the most important predictor.
- Income / Loan Amount ratio is more useful than Income or Loan Amount separately.
- City may matter if lending policies differ per region.
- Marital Status may not be a strong predictor unless combined with dependents.

So before cleaning or preprocessing, we already know:
✓ Which features matter most.
✓ Which features can be engineered.
✓ Which features may be dropped later.

Why this step is critical

- Prevents “blind” feature selection.
- Saves time later in feature engineering.
- Improves model explainability (stakeholders care about *why* a prediction is made).
- Avoids garbage-in-garbage-out problem.

2. Determining the Machine Learning Branch

Once you understand your data and what you're trying to predict, the next step is to decide **what type of ML problem you're solving**. This is crucial because the branch of ML determines:

- Which algorithms are appropriate.
- How the data should be prepared.
- What evaluation metrics to use.

Main Branches of Machine Learning

1. Supervised Learning

We have **input features (X)** and a **labeled target (y)**. The model learns a mapping from X → y.

- **Regression** → Target is **continuous (numeric)**.
  - Example: Predicting house price, predicting temperature.
  - Algorithms: Linear Regression, Random Forest Regressor, XGBoost, Neural Networks.
  - Metrics: MSE, RMSE, MAE, R^2.
- **Classification** → Target is **categorical (discrete labels)**.
  - Example: Spam vs Not Spam, Disease vs No Disease.
  - Algorithms: Logistic Regression, SVM, Decision Trees, Random Forest, Neural Networks.
  - Metrics: Accuracy, Precision, Recall, F1-score, ROC-AUC.

---

2. Unsupervised Learning

We have **input features (X)** but **no target (y)**. The goal is to find patterns or structure in the data.

- **Clustering** → Group similar data points.
  - Example: Customer segmentation, grouping news articles.
  - Algorithms: KMeans, DBSCAN, Hierarchical Clustering.
- **Dimensionality Reduction** → Reduce features while keeping essential information.
  - Example: PCA, t-SNE, UMAP.

---

3. Semi-Supervised Learning

- Dataset has **a few labeled samples + many unlabeled samples**.
- Example: Medical images (labels are expensive to get).
- Methods: Self-training, Label Propagation.

How to Decide the Branch

1. **Check the target variable (y)**:
    - If numeric → **Regression**.
    - If categorical → **Classification**.
    - If absent → **Unsupervised** (clustering, dimensionality reduction).
2. **Check business/research problem**:
    - *Do we want prediction or discovery?*
    - *Do we want grouping or forecasting?*
3. **Check data availability**:
    - Enough labels → Supervised.
    - No labels → Unsupervised.
    - Limited labels → Semi-supervised.

Example: Predicting Employee Attrition

- **Target variable**: Attrition (Yes/No)
- Type → **Categorical** → **Classification**.
- Possible algorithms → Logistic Regression, Random Forest, XGBoost.
- Evaluation → Accuracy, Precision, Recall, ROC-AUC.

Why this step matters

- Wrong branch = Wrong algorithms = Poor results.
- Prevents wasting time testing irrelevant models.
- Clarifies evaluation metrics early in the project.

3. Load and Merge Data

Once you know which ML branch your problem belongs to, the next step is to **bring your data into your environment (usually Python)** so you can start preparing it.

Common Ways to Load Data in Python

1. CSV / Excel Files

Most datasets are provided as .csv or .xlsx.

```
import pandas as pd

# Load CSV
df = pd.read_csv("data.csv")

# Load Excel
df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
```

---

2. Databases (SQL)

If your data is stored in MySQL, PostgreSQL, or SQLite:

```
import pandas as pd
import sqlite3

# Example: SQLite
conn = sqlite3.connect("database.db")
df = pd.read_sql("SELECT * FROM employees", conn)
```

For larger databases (MySQL, PostgreSQL), you use SQLAlchemy.

---

3. Web / APIs

```
import pandas as pd

url = "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
df = pd.read_csv(url)
```

For APIs:

```
import requests

response = requests.get("https://api.example.com/data")
data = response.json()
df = pd.DataFrame(data)
```

4. Multiple Datasets

Often you'll have more than one dataset, e.g., sales.csv and customers.csv.
You'll need to **merge** them.

```
# Example DataFrames
customers = pd.DataFrame({
   "customer_id": [1,2,3],
   "name": ["Alice", "Bob", "Charlie"]
})

sales = pd.DataFrame({
   "sale_id": [101,102,103],
   "customer_id": [1,2,3],
   "amount": [250, 150, 400]
})

# Merge on customer_id
merged = pd.merge(sales, customers, on="customer_id", how="inner")
print(merged)
```

```
   sale_id  customer_id  amount     name
0      101            1     250    Alice
1      102            2     150      Bob
2      103            3     400  Charlie
```

how can be:

- "inner" → only matching rows.
- "left" → keep all from left, add matches.
- "right" → keep all from right.
- "outer" → keep everything, fill missing with NaN.

---

5. Concatenating Datasets

When you have multiple files with the **same columns**, you just stack them:

```
df1 = pd.read_csv("data_part1.csv")
df2 = pd.read_csv("data_part2.csv")

# Combine vertically
df = pd.concat([df1, df2], ignore_index=True)
```

4. Inspect and Standardize Data

Once the dataset is loaded, we need to **understand its structure and clean up inconsistencies** before doing deep analysis.

Inspecting the Dataset

1. Shape (Rows × Columns)

```
print(df.shape)
```

👉 Tells us how many samples and features we have.

---

2. Preview Data

```
print(df.head())    # first 5 rows
print(df.tail())    # last 5 rows
```

👉 Quick peek at actual values.

---

3. Column Names

```
print(df.columns)
```

👉 Helps us spot typos like "CustName" vs "Customer Name".

---

4. Data Types

```
print(df.dtypes)
```

👉 Example: int64, float64, object (categorical), datetime64.

---

5. Null / Missing Values

```
print(df.isnull().sum())
```

👉 Identifies columns with missing data.

---

6. Unique Values (for Categorical Columns)

```
for col in df.select_dtypes(include="object").columns:
   print(col, df[col].unique()[:10])  # show first 10 unique values
```

👉 Helps us see inconsistencies like "Yes", "yes", "Y".

7. Descriptive Statistics

```
print(df.describe())        # for numeric
print(df.describe(include='object'))  # for categorical
```

---

2. Standardizing Categorical Data

This is the **basic cleaning step** to ensure consistent representation.

1. Consistent Case (upper/lower)

```
df['City'] = df['City'].str.strip().str.title()   # "los angelos" → "Los Angelos"
df['Response'] = df['Response'].str.strip().str.upper()   # "yes", "Yes" → "YES"
```

---

2. Fix Common Synonyms

```
df['Response'] = df['Response'].replace({
    "Y": "YES", "Yes": "YES", "yes": "YES",
    "N": "NO", "No": "NO", "no": "NO"
})

df['City'] = df['City'].replace({
    "LA": "Los Angeles", "Los Angelos": "Los Angeles",
    "NYC": "New York", "NewYork": "New York"
})
```

---

3. Dates Standardization

```
df['Date'] = pd.to_datetime(df['Date'], errors="coerce")
```

---

4. Removing Whitespace

```
df.columns = df.columns.str.strip()
```

---

5. Rename Columns for Clarity

```
df.rename(columns={"CustName": "Customer_Name"}, inplace=True)
```

✅ After this step, your dataset is **clean, consistent, and standardized** — ready for **EDA (Exploratory Data Analysis)**.

3. Standardizing Numerical Data

1. Check Summary Statistics

```
print(df.describe())
```

👉 Reveals suspicious values: negative ages, income = 0, very high outliers, etc.

---

2. Handling Impossible Values

Some numeric fields have **logical limits**. Example:

- Age must be > 0 and usually < 120.
- Salary should not be negative.
- Quantity of products can't be negative.

```
df = df[df['Age'] > 0]              # remove negative or zero ages
df['Salary'] = df['Salary'].clip(lower=0)   # replace negatives with 0
```

---

3. Replace with NaN for Review

Instead of dropping directly, sometimes we mark invalid values as missing so we can handle them later:

```
import numpy as np

df.loc[df['Age'] < 0, 'Age'] = np.nan
df.loc[df['Salary'] < 0, 'Salary'] = np.nan
```

---

4. Unit Standardization

Different datasets may use different units (e.g., **cm vs inches**, **kg vs pounds**).
We should unify them:

```
# Convert inches to cm
df['Height_cm'] = df['Height_in'] * 2.54
```

---

5. Rounding / Precision

Sometimes decimals don't make sense (e.g., **Age = 23.7**).

```
df['Age'] = df['Age'].round(0).astype(int)
```

4. Dropping Unnecessary Columns

Why Drop Columns?

Some columns do not add value to the prediction task, or may even harm the model. Examples:

- **Identifiers**: ID, Name, SSN, CustomerID → unique for every row, not useful for prediction.
- **High missing values**: Columns with >70–80% nulls (unless the missingness itself is informative).
- **Duplicate information**: Features that are highly correlated or essentially encode the same info (e.g., Height in cm and Height in inches).
- **Irrelevant to target**: Example: if predicting "house price," a "serial number" column is irrelevant.
- **Data leakage**: Columns that directly reveal the target (e.g., "TotalPurchase" when predicting "Purchase").

How to Detect Useless Columns

```
# Drop columns by name
df = df.drop(['CustomerID', 'SerialNo'], axis=1)

# Drop columns with too many missing values
threshold = 0.7 * len(df)   # 70% missing
df = df.dropna(thresh=threshold, axis=1)

# Drop columns with only one unique value
df = df.drop(columns=[col for col in df.columns if df[col].nunique() == 1])
```

Dropping Correlated Columns

Highly correlated columns may not always be harmful, but they can cause **multicollinearity** (bad for linear regression, logistic regression).

```
import seaborn as sns

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

# Example: drop one of two highly correlated features
df = df.drop('Height_in_inches', axis=1)
```

Smart Dropping vs Blind Dropping

- **Don't just delete everything with missing values or correlation** — consider domain knowledge.
- Sometimes a feature looks irrelevant but becomes useful in **feature engineering** (e.g., splitting "Address" into "City" and "Zipcode").

5. Exploratory Data Analysis

1. Univariate Analysis (One Feature at a Time)

For Numerical Features:

- **Histograms / KDE plots** → Show distributions.

```
import seaborn as sns
sns.histplot(df['Age'], kde=True)
```

- **Boxplots** → Identify outliers.

```
sns.boxplot(x=df['Income'])
```

For Categorical Features:

- **Barplots / Countplots** → Frequency of each category.

```
sns.countplot(x='Gender', data=df)
```

- **Pie charts** → Proportions of categories.

```
df['Gender'].value_counts().plot.pie(autopct="%1.1f%%")
```

---

2. Bivariate Analysis (Two Features Together)

- **Numerical vs Numerical**: Scatter plots & correlation.

```
sns.scatterplot(x='Age', y='Income', data=df)
df[['Age','Income']].corr()
```

- **Categorical vs Numerical**: Boxplot / Violin plot.

```
sns.boxplot(x='Gender', y='Income', data=df)
```

- **Categorical vs Categorical**: Crosstab / Heatmap.

```
import pandas as pd
pd.crosstab(df['Gender'], df['Purchased'])
sns.heatmap(pd.crosstab(df['Gender'], df['Purchased']), annot=True)
```

3. Multivariate Analysis (Many Features)

- **Pairplots**: Visualize relationships between all numerical features.

```
sns.pairplot(df[['Age','Income','SpendingScore']], hue='Gender')
```

- **Heatmap of Correlations**:

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

- **3D Plots** (using Plotly):

```
import plotly.express as px
fig = px.scatter_3d(df, x='Age', y='Income', z='SpendingScore', color='Gender')
fig.show()
```

---

4. Detect Patterns & Insights

- Which features are **important predictors**?
- Any **strong correlations**?
- Are there **imbalances** in categories (e.g., Male vs Female)?
- Are there **outliers** that need handling?

6. Handling Missing Values

Why Handle Missing Values?

- **Incomplete data** can bias or break models.
- Some ML models (like XGBoost, LightGBM) handle missing values internally, but many (like scikit-learn's Logistic Regression, SVM) **cannot work with NaNs**.
- The strategy depends on:
    - **Feature type** (numeric, categorical, text, datetime).
    - **Amount of missingness**.
    - **Whether missingness is random or systematic**.

Detecting Missing Values

```
# Count total missing values per column
print(df.isnull().sum())

# Percentage missing
print(df.isnull().mean() * 100)

# Visualize missing data
import seaborn as sns
sns.heatmap(df.isnull(), cbar=False, cmap="viridis")
```

Strategies to Handle Missing Values

1. Drop Missing Values

- When too many values are missing (>70–80%).

```
df = df.dropna()            # Drop rows with any NaN
df = df.dropna(axis=1)        # Drop columns with any NaN
df = df.dropna(thresh=100)      # Keep only rows with >=100 non-null values
```

2. Fill (Impute) Missing Values

Numeric Columns

- **Mean / Median / Mode imputation**

```
df['Age'].fillna(df['Age'].mean(), inplace=True)      # Mean
df['Age'].fillna(df['Age'].median(), inplace=True)    # Median
df['Age'].fillna(df['Age'].mode()[0], inplace=True)   # Mode
```

- **Forward Fill / Backward Fill**

```
df['Sales'].fillna(method='ffill', inplace=True)   # Forward fill
df['Sales'].fillna(method='bfill', inplace=True)   # Backward fill
```

- **Domain-specific filling**
    - Example: Missing **Salary** → fill with 0 (if "not employed").
    - Example: Missing **Temperature** → fill with seasonal averages.

- **Interpolation**

```
df['Temperature'] = df['Temperature'].interpolate(method='linear')
```

Categorical Columns

- **Fill with Mode**

```
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
```

- **Fill with Constant**

```
df['City'].fillna('Unknown', inplace=True)
```

- **Domain Knowledge**
    - Example: "Country" missing → look up ZIP code.

Date/Time Columns

- **Forward/Backward Fill**

```
df['Date'] = df['Date'].fillna(method='ffill')
```

- **Special imputation**: If predicting seasonal trends, missing dates may be interpolated.

---

4. Filling with a Condition or Custom Function

Sometimes you need **domain-specific logic** for imputation.

Example 1: Fill based on condition

```
# If Age < 18 and NaN in Salary → fill with 0
df.loc[(df['Age'] < 18) & (df['Salary'].isnull()), 'Salary'] = 0
```

Example 2: Fill with Group Statistics

```
# Fill missing Salary with median salary of each Department
df['Salary'] = df.groupby('Department')['Salary'].transform(
   lambda x: x.fillna(x.median())
)
```

Example 3: Use a Function

```
def fill_salary(row):
   if pd.isnull(row['Salary']):
      if row['Job'] == 'Intern':
         return 1000
      elif row['Job'] == 'Manager':
         return 8000
      else:
         return 3000
   else:
      return row['Salary']

df['Salary'] = df.apply(fill_salary, axis=1)
```

✅ This is often the **most powerful method** because it uses **context and business rules**.

3. Model-Based Imputation

- Train a small model (KNN, regression, ML imputer) to predict missing values.

```
from sklearn.impute import KNNImputer
import numpy as np

imputer = KNNImputer(n_neighbors=3)
df_imputed = imputer.fit_transform(df.select_dtypes(include=np.number))
```

- Use Simple Imputer

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

df = pd.DataFrame({
   "Age": [25, np.nan, 27, 29, np.nan],
   "Salary": [50000, 54000, np.nan, 58000, 60000]
})

# Mean Imputer
imputer_mean = SimpleImputer(strategy="mean")
df_mean = imputer_mean.fit_transform(df)

# Median Imputer
imputer_median = SimpleImputer(strategy="median")
df_median = imputer_median.fit_transform(df)

# Most frequent (mode)
imputer_mode = SimpleImputer(strategy="most_frequent")
df_mode = imputer_mode.fit_transform(df)

# Constant value
imputer_constant = SimpleImputer(strategy="constant", fill_value=-1)
df_constant = imputer_constant.fit_transform(df)
```

When to Drop vs Fill?

- **Drop** if column is mostly NaN or irrelevant.
- **Fill** if the column is important and only partially missing.
- **Keep Missing as a Feature** → sometimes *missingness itself* is informative (e.g., missing salary → unemployed).

Detailed Filling / Imputation

1. Forward Fill (ffill) & Backward Fill (bfill)

- **Forward Fill (ffill)** → takes the last **known value** and carries it forward.
- **Backward Fill (bfill)** → takes the next **known value** and pushes it backward.

Example

```
import pandas as pd

df = pd.DataFrame({"Temperature": [22, None, None, 25, None, 28]})
print(df)

df_ffill = df.fillna(method="ffill")
df_bfill = df.fillna(method="bfill")
```

- ffill result: [22, 22, 22, 25, 25, 28]
- bfill result: [22, 25, 25, 25, 28, 28]

✅ **When to use**:

- Time-series data where the previous/next value makes sense.
- Example: Stock prices, weather recordings, sensor data.

⚠️ **When not to use**:

- If missing values are long gaps (you'd just be duplicating one value).
- If data isn't time-ordered.

2. Interpolation

Interpolation fills missing values by estimating based on surrounding data.

Types of Interpolation in Pandas

```
df['Temperature'].interpolate(method='linear')      # Default: straight line between points
df['Temperature'].interpolate(method='polynomial', order=2)   # Curve fit
df['Temperature'].interpolate(method='spline', order=2)       # Smooth curve
df['Temperature'].interpolate(method='time')       # Use index as time for estimation
```

✅ **When to use**:

- Continuous numerical data (temperature, pressure, signal data).
- When values follow a trend and you don't want to artificially repeat numbers.

⚠️ **When not to use**:

- For categorical data (like gender or city).
- If missingness is at the beginning or end → interpolation fails because no boundary values exist.

3. KNN Imputer (Model-Based)

The **KNN Imputer** uses the values of **nearest neighbors** (similar rows) to estimate missing values.

Example

```
import numpy as np
import pandas as pd
from sklearn.impute import KNNImputer

df = pd.DataFrame({
    "Age": [25, np.nan, 27, 29, np.nan],
    "Salary": [50000, 54000, np.nan, 58000, 60000]
})

imputer = KNNImputer(n_neighbors=2, weights="uniform")
df_imputed = imputer.fit_transform(df)

print(pd.DataFrame(df_imputed, columns=df.columns))
```

🔹 **KNN Imputer Hyperparameters**

- **n_neighbors** (default=5): number of neighbors used to estimate missing value.
    - Larger values = smoother but less precise.
    - Smaller values = more variation but risk of overfitting.
- **weights**:
    - "uniform" → all neighbors equal.
    - "distance" → closer neighbors weighted more.
- **metric**:
    - "nan_euclidean" (default): Euclidean distance that ignores NaNs.

✅ **When to use**:

- When features are related (age ↔ salary ↔ experience).
- Works better on **small to medium datasets** (KNN can be slow on millions of rows).

⚠️ **When not to use**:

- If data is very large (KNN = $O(n^2)$ complexity).
- If features are independent (no relationship to exploit).

4. Mean / Median / Mode Imputation

- **Mean**: good for symmetric distributions.
- **Median**: robust for skewed data or when outliers exist.
- **Mode**: best for categorical data.

✅ Simple, fast, and works when the missingness is small.
⚠️ Distorts variance and correlations if many values are missing.

---

5. Constant Value Filling

```
df['City'].fillna('Unknown', inplace=True)
```

- Useful for **categorical columns** where missing has a special meaning.
- Example: Missing Occupation → fill with "Unemployed".

---

6. Leave Missing as a Category

- Sometimes missingness itself is **informative** (e.g., people not reporting income).

```
df['Income'] = df['Income'].fillna("Missing")
```

---

7. Using SimpleImputer (scikit-learn)

- The SimpleImputer is the most **basic imputation tool** in scikit-learn, but very powerful when combined with ML pipelines.

Parameters

- **strategy**: "mean", "median", "most_frequent", "constant".
- **fill_value**: value to use if strategy="constant".
- **missing_values**: default np.nan, but can be set to any placeholder.
- **add_indicator**: if True, creates extra binary columns marking where missing values were.

✅ Works well in **pipelines** with scikit-learn.

7. Dropping Duplicates

Why?

Duplicate rows (or duplicate records) can:

- Inflate counts → misleading statistics (e.g., average income, churn rate).
- Bias the model → the same sample gets "double weight" in training.
- Waste storage and computation resources.

So, we need to detect and remove duplicates before modeling.

Types of Duplicates

1. **Exact duplicates**
   - Entire row is identical.
   - Example:
   - ID | Name | Age | City
   - 1 | Alice | 25 | Paris
   - 1 | Alice | 25 | Paris ← duplicate
2. **Partial duplicates** (based on a subset of columns)
   - Example: Two rows have the same Name and City but different ID.
3. **Near-duplicates** (typos, formatting differences, inconsistent capitalization)
   - Example:
   - "New York" vs "new york" vs "NY"
   - "Yes" vs "yes" vs "Y"

These need **standardization** before deduplication.

Methods to Handle Duplicates

1. Remove exact duplicates

```
# Drop exact duplicates
df = df.drop_duplicates()
```

2. Remove based on subset of columns

```
# Drop duplicates only based on Name and City
df = df.drop_duplicates(subset=["Name", "City"])
```

3. Keep first or last duplicate

```
# Keep first occurrence, drop others
df = df.drop_duplicates(keep="first")

# Keep last occurrence
df = df.drop_duplicates(keep="last")
```

When NOT to Drop Duplicates

- In **time-series data**: Repeated values may be valid (e.g., sensor readings at the same time).
- In **multi-label data**: Same feature row but multiple target labels may exist.
- In **transaction data**: Two identical transactions may both be legitimate.

Rule of thumb:

- Drop duplicates only if they don't carry unique information.
- Always check domain knowledge before removing.

Helper Functions for all Previous Steps

Statistical Analysis

```
def statistical_analysis(df):
    # 1. General Overview Dashboard
    display(Markdown("### 1. General Overview"))
    overview = pd.DataFrame({
        'Metric': ['Rows', 'Columns', 'Duplicates', 'Total Missing Values'],
        'Value': [df.shape[0], df.shape[1], df.duplicated().sum(), df.isnull().sum().sum()]
    }).set_index('Metric')
    display(overview)

    # 2. Data Types & Missing Values (Combined)
    display(Markdown("### 2. Data Types & Missing Values"))
    info_df = pd.DataFrame({
        'Type': df.dtypes,
        'Null Count': df.isnull().sum(),
        'Null %': (df.isnull().sum() / len(df)) * 100
    })
    # Sort by Null % so you see problems first
    display(info_df.sort_values(by='Null Count', ascending=False))

    # 3. Descriptive Statistics (Transposed for readability)
    display(Markdown("### 3. Numerical Statistics"))
    # .T makes it easier to read if you have many columns
    display(df.describe().T.style.background_gradient(cmap='Blues'))

    # 4. Categorical Breakdown
    display(Markdown("### 4. Categorical Value Counts"))
    cat_columns = df.select_dtypes(include=['object', 'category']).columns.tolist()

    if not cat_columns:
        print("No categorical columns found.")
    else:
        for col in cat_columns:
            display(Markdown(f"**--- {col} ---**"))
            # Display as a horizontal bar chart inside the dataframe
            val_counts = df[col].value_counts().to_frame()
            display(val_counts.style.bar(color='#5fba7d', vmin=0))

    # 5. First 5 Rows
    display(Markdown("### 5. Head"))
    display(df.head())
```

Plotting Histograms

```
def plot_distributions(df, columns=None, n_cols=3):
    """
    Plots histograms with a Kernel Density Estimate (KDE) line for numerical columns.

    Parameters:
    - df: The dataframe
    - columns: List of specific columns to plot (optional). If None, plots all numerical cols.
    - n_cols: Number of graphs per row.
    """
    # Auto-select numerical columns if none provided
    if columns is None:
        columns = df.select_dtypes(include=[np.number]).columns.tolist()

    if len(columns) == 0:
        print("No numerical columns to plot.")
        return

    # Calculate layout
    n_rows = (len(columns) - 1) // n_cols + 1
    plt.figure(figsize=(n_cols * 5, n_rows * 4))

    for i, col in enumerate(columns):
        plt.subplot(n_rows, n_cols, i + 1)
        sns.histplot(df[col], kde=True, color='teal', edgecolor='black')
        plt.title(f'Distribution of {col}')
        plt.xlabel(col)
        plt.ylabel('Frequency')

    plt.tight_layout()
    plt.show()
```

Plotting Box Plots for Outliers

```
def plot_outliers(df, columns=None, n_cols=4):
    """
    Plots box plots for numerical columns to identify outliers.
    """
    if columns is None:
        columns = df.select_dtypes(include=[np.number]).columns.tolist()

    if len(columns) == 0:
        print("No numerical columns to plot.")
        return

    n_rows = (len(columns) - 1) // n_cols + 1
    plt.figure(figsize=(n_cols * 4, n_rows * 3))

    for i, col in enumerate(columns):
        plt.subplot(n_rows, n_cols, i + 1)
        sns.boxplot(x=df[col], color='seagreen')
        plt.title(f'Outliers in {col}')

    plt.tight_layout()
    plt.show()
```

Plotting Count Plots for Categorical Data

```
def plot_categoricals(df, columns=None, n_cols=3):
    """
    Plots bar charts for categorical columns to show frequency.
    """
    if columns is None:
        # Select object types and categories
        columns = df.select_dtypes(include=['object', 'category']).columns.tolist()

    if len(columns) == 0:
        print("No categorical columns to plot.")
        return

    n_rows = (len(columns) - 1) // n_cols + 1
    plt.figure(figsize=(n_cols * 5, n_rows * 4))

    for i, col in enumerate(columns):
        plt.subplot(n_rows, n_cols, i + 1)
        sns.countplot(x=df[col], palette='magma')
        plt.title(f'Counts of {col}')
        plt.xticks(rotation=45)

    plt.tight_layout()
    plt.show()
```

Plotting Correlation Heat Map

```
def plot_correlation(df, target_col=None):
    """
    Plots a heatmap of correlations between numerical features.
    If target_col is provided, prints the correlation of features with the target.
    """
    # Select only numerical columns for correlation
    numeric_df = df.select_dtypes(include=[np.number])

    if numeric_df.empty:
        print("No numerical data for correlation.")
        return

    plt.figure(figsize=(20, 15))
    corr = numeric_df.corr()

    # Mask the upper triangle (since it's a mirror image)
    mask = np.triu(np.ones_like(corr, dtype=bool))

    sns.heatmap(corr, mask=mask, annot=True, fmt=".2f", cmap='coolwarm',
                linewidths=0.5, vmin=-1, vmax=1)
    plt.title('Correlation Matrix Heatmap')
    plt.show()

    if target_col and target_col in numeric_df.columns:
        print(f"\n--- Correlation with Target: {target_col} ---")
        target_corr = corr[target_col].sort_values(ascending=False)
        print(target_corr)
```

Plotting Features vs Target

```
def plot_feature_vs_target(df, target, isClassification, feature_cols=None):
    """
    Plots relationships between features and the target.
    - If target is Categorical (Classification): Uses Box Plots (Numerical Feature vs Target Class)
    - If target is Numerical (Regression): Uses Scatter Plots (Numerical Feature vs Target Value)
    """
    if feature_cols is None:
        # Default to top 5 numerical columns
        feature_cols = df.select_dtypes(include=[np.number]).columns.tolist()[:5]

    # Determine problem type based on target uniqueness
    is_classification = isClassification

    plt.figure(figsize=(15, 4 * len(feature_cols)))

    for i, col in enumerate(feature_cols):
        if col == target: continue

        plt.subplot(len(feature_cols), 1, i + 1)

        if is_classification:
            # Classification: Box Plot separating classes
            sns.boxplot(data=df, x=target, y=col, palette='Set2')
            plt.title(f'{col} distribution by {target}')
        else:
            # Regression: Scatter Plot
            sns.scatterplot(data=df, x=col, y=target, alpha=0.6)
            plt.title(f'{col} vs {target}')

    plt.tight_layout()
    plt.show()
```


Made by Youssef Amgad Elkhatib

8. Handling Outliers

What is an Outlier?

An **outlier** is a data point that significantly deviates from the rest of the dataset.

- Example: In a dataset of people's heights, values like 2.5m or 0.5m may be outliers.

Outliers can:

- Distort mean, variance, and correlation.
- Mislead ML models (especially linear regression, clustering).
- Cause longer training time and overfitting.

Causes of Outliers

- **Measurement errors** → faulty sensor, manual input error.
- **Natural variation** → rare but valid values (e.g., tallest person).
- **Wrong entry/data corruption**.
- **Sampling errors** → mixed populations in one dataset.

Methods to Handle Outliers

1. Dropping Outliers

- Directly remove rows with extreme values.
- Works if:
  - Outliers are clearly **errors**, not meaningful data.
  - Dataset is **large enough** (removing few rows won't affect training).

```
# Drop rows where Age > 100
df = df[df["Age"] <= 100]
```

2. Z-Score Method

- Based on **Standard Deviation**.
- Formula:

$$Z = (x - \mu) / \sigma$$

  - If $|Z| > 3$, the point is usually considered an outlier.

✅ Best when data is **normally distributed**.

```
from scipy import stats
import numpy as np

z_scores = np.abs(stats.zscore(df["Age"]))
df_no_outliers = df[z_scores < 3]
```

---

3. IQR Method (Interquartile Range)

- More **robust to non-normal data**.
- Steps:
  - Q1 = 25th percentile
  - Q3 = 75th percentile
  - IQR = Q3 - Q1
  - Outlier if value < (min = Q1 - 1.5 * IQR) or > (max = Q3 + 1.5 * IQR).

✅ Works well with **skewed data**.

```
Q1 = df["Age"].quantile(0.25)
Q3 = df["Age"].quantile(0.75)
IQR = Q3 - Q1

df_no_outliers = df[(df["Age"] >= Q1 - 1.5*IQR) & (df["Age"] <= Q3 + 1.5*IQR)]
```

4. Transformation

- Apply mathematical transformations to **reduce the effect of outliers**.
- Examples:
    - **Log** → strong right skew, positive-only values (income, salaries).
    - **Square Root** → moderate skew, count data, positive-only.
    - **Cube Root** → very skewed data, can handle negatives.

✅ Works well with heavily **skewed data** (e.g., financial data).

```
df["Income_log"] = np.log1p(df["Income"])
```

⚠️ **Important Note**: Not all outliers are bad. For example, in fraud detection, **outliers ARE the signal**.

9. Feature Engineering

Feature engineering is the process of transforming raw data into meaningful features that improve the performance of machine learning models. It's often the most important step in building high-quality ML pipelines.

---

1. Feature Creation (Generating New Features)

You create new variables that may capture more information from raw data.

- **Date/Time features**

```
import pandas as pd

df = pd.DataFrame({"date": pd.to_datetime(["2023-01-01", "2023-05-15", "2023-08-20"])})
df["year"] = df["date"].dt.year
df["month"] = df["date"].dt.month
df["day_of_week"] = df["date"].dt.dayofweek
```

- **Ratios / Rates**

```
df["price_per_unit"] = df["total_price"] / df["quantity"]
```

- **Text features**

```
df["review_length"] = df["review"].apply(len)
df["word_count"] = df["review"].apply(lambda x: len(x.split()))
```

---

2. Feature Combination

Combining multiple features to add more predictive power.

- **Interaction features**

```
df["rooms_bath_ratio"] = df["rooms"] / df["bathrooms"]
df["total_size"] = df["height"] * df["width"]
```

10. Splitting Data

After splitting Data, we can apply feature selection and dimensionality reduction before encoding, scaling.

1. Train / Validation / Test Splitting

There are 3 sets of data:

- **Training set** → Model *learns patterns* from this.
- **Validation set** → Used to *tune hyperparameters* and decide which model is best.
- **Test set** → *Final evaluation only*, simulating real-world unseen data.

```
from sklearn.model_selection import train_test_split
import pandas as pd

# Example dataset
df = pd.read_csv("data.csv")
X = df.drop("target", axis=1)
y = df["target"]

# Train (70%) / Validation (15%) / Test (15%)
X_train_full, X_test, y_train_full, y_test = train_test_split(
   X, y, test_size=0.15, random_state=42, stratify=y
)
X_train, X_val, y_train, y_val = train_test_split(
   X_train_full, y_train_full, test_size=0.176, random_state=42, stratify=y_train_full
)  # 0.176 ~ 15% of original

# 3. Validate
y_val_pred = model.predict(X_val)
print("Validation Accuracy:", accuracy_score(y_val, y_val_pred))

# (tune hyperparameters manually here...)

# 4. Retrain best model on Train+Val
X_train_final = np.concatenate([X_train, X_val])
y_train_final = np.concatenate([y_train, y_val])
best_model = RandomForestClassifier(n_estimators=200, max_depth=10, random_state=42)
best_model.fit(X_train_final, y_train_final)

# 5. Test evaluation
y_test_pred = best_model.predict(X_test)
print("Test Accuracy:", accuracy_score(y_test, y_test_pred))
```

- **test_size: how much data goes into test.**
- **random_state: makes split reproducible.**
- **stratify=y: keeps class distribution consistent.**

2. Cross-Validation (CV) Instead of Fixed Validation

Instead of manually splitting into Train/Validation, we can use **k-Fold Cross Validation**:

- Split train data into *k* folds (e.g., 5).
- Train on (k-1) folds, validate on 1.
- Repeat k times, average performance.

This ensures stability, since results won't depend on one lucky split.

```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(random_state=42)

# 5-Fold Cross-Validation (Accuracy score)
cv_scores = cross_val_score(rf, X, y, cv=5, scoring="accuracy")

print("Cross-validation scores:", cv_scores)
print("Mean CV accuracy:", cv_scores.mean())
```

11. Encoding

Categorical data must be converted to numeric before feeding into ML models.

---

1. Label Encoding(Target)

- **Each unique category is mapped to an integer.**
- **Example: ["Red", "Green", "Blue"] → [0,1,2].**
- ⚠️ **Problem: Implies an order that may not exist (bad for linear models).**
- ✅ **Good for tree-based models (Random Forest, XGBoost).**

```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
df["color_encoded"] = le.fit_transform(df["color"])
```

---

2. One-Hot Encoding

- **Creates binary columns for each category.**
- **Example: Color = ["Red", "Blue"] → Red:[1,0], Blue:[0,1].**
- ✅ **Best when categories are nominal and small in number.**
- ⚠️ **Not good when categories are large in number.**

```
pd.get_dummies(df, columns=["color"])
```

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

# Example
df = pd.DataFrame({"Color": ["Red", "Blue", "Green", "Blue"]})

encoder = OneHotEncoder(sparse=False, drop="first")  # drop first to avoid dummy trap
encoded = encoder.fit_transform(df[["Color"]])

df_encoded = pd.DataFrame(encoded, columns=encoder.get_feature_names_out(["Color"]))
print(df_encoded)
```

3. Frequency / Count Encoding

- Replace categories with their frequency.
- Example: ["A","B","A","C"] → [2,1,2,1].
- ✅ Useful when there are many categories (like city names).
- ⚠️ Bad For high-cardinality categorical features with small samples → noise dominates.

```
freq = df["city"].value_counts()
df["city_freq"] = df["city"].map(freq)
```

---

4. Target Encoding (Mean Encoding)

- Replace categories with mean of target variable.
- Example: if target=churn, encode city as churn probability.
- ⚠️ Risk of data leakage — use with CV.
- ✅ Good for high-cardinality features.

```
df["city_te"] = df.groupby("city")["target"].transform("mean")
```

---

5. Ordinal Encoding

- Maps categories to integers based on order.
- Example: Education = {Primary:1, Secondary:2, Bachelor:3, Master:4, PhD:5}.
- ✅ Best for ordinal data (with order but not exact numeric meaning).
- ⚠️ Bad when Categories are nominal (unordered), like ["Red", "Blue", "Green"]

```
from sklearn.preprocessing import OrdinalEncoder

df = pd.DataFrame({"Education": ["Bachelor", "Master", "PhD", "Secondary"]})

encoder = OrdinalEncoder(categories=[["Primary","Secondary","Bachelor","Master","PhD"]])
encoded = encoder.fit_transform(df[["Education"]])
df["Education_encoded"] = encoded
```

12. Feature Selection

Feature Selection chooses a **subset of the original features** that are most relevant to your target variable.

It **does not create new features**, unlike PCA.

Benefits:

- Reduces overfitting
- Improves model accuracy
- Shortens training time
- Enhances interpretability

(a) Information Gain (Entropy-Based)

Measures how much information a feature gives about the class (used in Decision Trees).

Information gain calculates the reduction in entropy from the transformation of a dataset. It can be used for feature selection by evaluating the Information gain of each variable in the context of the target variable.

Code Example:

```
from sklearn.feature_selection import mutual_info_classif
from sklearn.datasets import load_iris
import pandas as pd

X, y = load_iris(return_X_y=True)
mi = mutual_info_classif(X, y)

# Rank features by Information Gain
pd.DataFrame({'Feature': range(X.shape[1]), 'Information Gain': mi}).sort_values(by='Information Gain',
ascending=False)
```

High information gain → strong predictor.

(b) Chi-Square Test (χ^2 Test)

We calculate Chi-square between each feature and the target and select the desired number of features with the best Chi-square scores. In order to correctly apply the chi-squared to test the relation between various features in the dataset and the target variable, the following conditions have to be met: the variables have to be categorical, sampled independently, and values should have an expected frequency greater than 5.
It tests the independence between each feature and the target.

Code Example:

```
from sklearn.feature_selection import chi2, SelectKBest
k_best = SelectKBest(chi2, k=5)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=42)
x_train_chi = k_best.fit_transform(x_train, y_train)
x_test_chi = k_best.transform(x_test)
k_best.get_support(indices=True)
```

High chi2 score → more dependence → more important feature.

(c) Fisher's Score

Used for classification; measures the separation between classes for each feature:

It returns the ranks of the variables based on the fisher's score in descending order. We can then select the variables as per the case

$$\text{Score}(i) = \frac{(\mu_1 - \mu_2)^2}{\sigma_1^2 + \sigma_2^2}$$

Code Example:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
fisher = LinearDiscriminantAnalysis(n_components=1)
x_train_fisher = fisher.fit_transform(x_train, y_train)
x_test_fisher = fisher.transform(x_test)
```

High Fisher's score → better separation between classes.

(d) Variance Threshold

It removes all features whose variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e., features with the same value in all samples. We assume that features with a higher variance may contain more useful information but note that we are not taking the relationship between feature variables or feature and target variables into account.

Code Example:

```
from sklearn.feature_selection import VarianceThreshold
import numpy as np

X = np.array([[0, 2, 0, 3],
              [0, 1, 4, 3],
              [0, 1, 1, 3]])

sel = VarianceThreshold(threshold=0.2)
X_new = sel.fit_transform(X)

print("Remaining features shape:", X_new.shape)
```

Removes features with variance below 0.2.

(e) Median Absolute Deviation (MAD)

Like variance threshold but **more robust to outliers**.

$$MAD = \text{median}(|\ x_i - \text{median}(x)\ |)$$

Code Example:

```
def mad(x):
   mean = np.mean(x)
   return np.mean(np.abs(x - mean))
mad_vales = np.apply_along_axis(mad, axis=0, arr=x)
threshold = 4
selected_features = np.where(mad_vales > threshold)[0]
```

Keeps features with meaningful variability (robust to outliers).

13. Scaling Types

Scaling ensures that features are comparable and don't dominate learning due to their units/magnitude.

---

1. Standardization (Z-score scaling)

- **Formula:**

$$z=(x-\mu) / \sigma$$

- **Mean = 0, Std = 1.**
- ✅ **Best for algorithms assuming normal distribution (Logistic Regression, Linear Regression, SVM).**
- ⚠️ **Data is not normally distributed and heavily skewed (mean/std don't describe distribution well).**

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

---

2. Min-Max Normalization

- **Formula:**

$$x'=(x-x_{min}) / (x_{max}-x_{min})$$

- **Scales values to range [0,1].**
- ✅ **Best for distance-based models (KNN, Neural Networks).**
- ⚠️ **Very Sensitive to Outliers**

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_norm = scaler.fit_transform(X)
```

3. Robust Scaling

- **Uses median & IQR instead of mean & std.**
- ⚠️ **Data has no or few outliers. Then, using median/IQR unnecessarily downplays natural variation.**
- ✅ **Best for Data with outliers**

```
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()
X_robust = scaler.fit_transform(X)
```

---

4. Transformation

- Apply **mathematical transformations**.
- Examples:
  - **Log** → strong right skew, positive-only values (income, salaries).
  - **Square Root** → moderate skew, count data, positive-only.
  - **Cube Root** → very skewed data, can handle negatives.

✅ Works well with heavily **skewed data** (e.g., financial data).

14. Dimensionality Reduction

Dimensionality Reduction means reducing the number of **input variables** (features) while keeping as much **information (variance)** as possible.

Why do it?

- To **reduce overfitting** (fewer features → simpler model)
- To **speed up training**
- To **visualize** high-dimensional data (2D/3D projection)
- To **remove correlated or redundant features**

PCA (Principal Component Analysis)

Concept

PCA finds **new axes (principal components)** that:

- capture the **maximum variance** in the data,
- are **uncorrelated** with each other.

It projects your original data into a smaller space, e.g.:

- From 10 features → 2 or 3 "principal components"
- These components are **linear combinations** of your original features

Mathematical Idea

PCA computes the **eigenvectors** and **eigenvalues** of the covariance matrix of your data.

The top $k$eigenvectors (largest eigenvalues) form your new feature space.

Python Example (PCA)

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
import pandas as pd

# Load dataset
X, y = load_iris(return_X_y=True)

# Standardize data (important for PCA)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

print("Explained variance ratio:", pca.explained_variance_ratio_)
```

Output Example:

Explained variance ratio: [0.9246, 0.0530]

This means the first two components explain ~97% of the variance → enough to represent the data efficiently.

When to Use PCA

- When features are **highly correlated**
- When you have **many continuous variables**
- When you need **visualization in 2D/3D**
- Before clustering or other distance-based models (e.g., K-Means)

15. Pipelines (Best Practice)

Instead of manually repeating, use **ColumnTransformer + Pipeline**:

```
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# Define preprocessors
numeric_features = ["age", "salary"]
categorical_features = ["city"]

preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numeric_features),
        ("cat", OneHotEncoder(handle_unknown="ignore"), categorical_features),
        ('pca', PCA(n_components=5))
    ]
)

# Full pipeline
clf = Pipeline(steps=[
    ("preprocess", preprocessor),
    ("model", LogisticRegression())
])

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

This way:

- Scaling & encoding are **automatically applied** correctly.
- Avoids leakage.
- Cleaner code.

What each piece means

1. preprocessor

- A **ColumnTransformer** object.
- Its job: apply **different transformations** to **different columns**.
- Example: scale numbers, encode categories, leave IDs untouched.

---

2. ("num", StandardScaler(), numeric_features)

- "num": Just a **name/label** you give to this transformer.
- StandardScaler(): The **transformer** applied to numeric columns.
- numeric_features: List of columns to apply it to.

So, this line = *"Apply StandardScaler to columns age and salary."*

---

3. ("cat", OneHotEncoder(...), categorical_features)

- "cat": Name for categorical transformer.
- OneHotEncoder(): Turns categories into 0/1 dummy variables.
- categorical_features: List of categorical columns.

So, this line = *"Apply OneHotEncoder to column city."*

4. What if I want different scalings/encodings?

You just add more transformers in the list.

Example:

```
preprocessor = ColumnTransformer(
    transformers=[
        ("standard", StandardScaler(), ["age"]),
        ("robust", RobustScaler(), ["salary"]),
        ("ordinal", OrdinalEncoder(), ["education"]),
        ("onehot", OneHotEncoder(handle_unknown="ignore"), ["city"])
    ]
)
```

This way:

- age → StandardScaler
- salary → RobustScaler
- education → OrdinalEncoder
- city → OneHotEncoder

---

5. What is a Transformer?

A **transformer** = any object that can:

- .fit() on training data (learn parameters like mean/std for scaling, category mappings for encoding).
- .transform() to apply those learned rules to train/test data.

Examples: StandardScaler, OneHotEncoder, MinMaxScaler.

---

6. Pipeline

- A **sequence of steps** that happen in order.
- Each step = (name, object).
- Ensures preprocessing happens **before** model training.

7. What is preprocess and model here?

- "preprocess": step name (you chose it). Refers to the ColumnTransformer.
- "model": step name for the ML model (Logistic Regression here).

---

8. Where does random_state go?

- Not inside pipeline.
- Goes into the **model** (if it supports randomness) or into train_test_split.

Examples:

```
LogisticRegression(random_state=42)   # reproducible model
train_test_split(X, y, test_size=0.2, random_state=42)  # reproducible split
```

---

9. clf.fit()

This does automatically:

1. **preprocess.fit(X_train) →** learns scaling parameters (mean/std, encodings).
2. **preprocess.transform(X_train) →** applies them to training data.
3. **model.fit() →** trains LogisticRegression on the processed features.

---

10. y_pred = clf.predict(X_test)

This does automatically:

1. **preprocess.transform(X_test) →** applies the same transformations learned from training.
2. **model.predict() →** makes predictions.

16. Handling Imbalanced Data

Handling **imbalanced data in training set** is one of the most important steps in preprocessing because if you skip it, your model will get biased towards the **majority class** and fail badly on the minority class (which is often the class you care about most, e.g. fraud, disease).

Detecting Imbalance

- Check **class distribution**:

```
import pandas as pd
print(y.value_counts(normalize=True))
```

- If one class <10–20% → dataset is **imbalanced**.

Solutions

1. Random Oversampling

- Duplicate minority class samples randomly.
- Pros: Simple, preserves information.
- Cons: Risk of overfitting (duplicates).

```
from imblearn.over_sampling import RandomOverSampler
ros = RandomOverSampler(sampling_strategy='minority', random_state=42)
X_res, y_res = ros.fit_resample(X_train, y_train)
```

---

2. Random Undersampling

- Remove majority class samples randomly.
- Pros: Fast, reduces dataset size.
- Cons: May lose valuable info.

```
from imblearn.under_sampling import RandomUnderSampler
rus = RandomUnderSampler(sampling_strategy='majority', random_state=42)
X_res, y_res = rus.fit_resample(X_train, y_train)
```

3. SMOTE (Synthetic Minority Oversampling Technique)

- Generates synthetic samples for minority class using nearest neighbors.
- Pros: Creates more diverse data than oversampling.
- Cons: Can create noisy samples, works best with continuous features.

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(sampling_strategy='minority', random_state=42, k_neighbors=5)
X_res, y_res = smote.fit_resample(X_train, y_train)
```

17. Model Building and Evaluation

When building, you should evaluate using multiple metrics (not just accuracy, which is misleading in imbalanced datasets).

---

1. Metrics to Compute

- **Classification**:
  - Accuracy → good for balanced datasets → **Higher = better**
  - Precision → correctness of positives → **Higher = better**
  - Recall → coverage of positives → **Higher = better**
  - F1-score → balance of precision & recall → **Higher = better**
  - ROC-AUC → how well classes are separated → **Higher = better**

- **Regression**:
  - Mean Absolute Error (MAE) → Measures *average absolute difference* between predicted and actual values → **Lower is better.**
  - Mean Squared Error (MSE) → Measures the *average of squared errors* → **Lower is better.**
  - Root Mean Squared Error (RMSE) → Square root of MSE, same unit as target → **Lower is better.**
  - R^2 (Coefficient of Determination) → Measures how much variance in the target is explained by the model → **Closer to 1 = better fit.**
  - Adjusted R^2 → adjusted for the number of predictors → **Higher is better.**
  - Mean Absolute Percentage Error (MAPE) → Measures average *percentage* error → **Lower % is better.**
  - Mean Squared Logarithmic Error (MSLE) → Measures squared error in log space → **Lower is better.**

- **Clustering Internal Metrics:**
  - **Silhouette Score** → measures cohesion & separation → **Higher = better**
  - **Davies–Bouldin Index (DBI)** → ratio of within to between cluster distances → **Lower = better**
  - **Calinski–Harabasz Index (CH)** → between / within variance ratio → **Higher = better**
  - **Dunn Index** → min inter-cluster / max intra-cluster distance → **Higher = better**
  - **Elbow Method** → visual method for choosing best k → **Look for the "elbow" point**

- **External Clustering Metrics (with true labels)**
  - **Adjusted Rand Index (ARI)** → pairwise agreement between true & predicted → **Higher = better**
  - **Normalized Mutual Information (NMI)** → shared info between true & predicted → **Higher = better**
  - **Homogeneity Score** → each cluster has one class → **Higher = better**
  - **Completeness Score** → all samples of a class in one cluster → **Higher = better**
  - **V-Measure** → harmonic mean of homogeneity & completeness → **Higher = better**
  - **Fowlkes–Mallows Index (FMI)** → balance of precision & recall → **Higher = better**

2. Hyperparameter Tuning

Hyperparameters are **not learned automatically** (like learning rate, depth, regularization). They must be tuned.

- **Approaches:**

  - **Manual tuning** → try values manually.
  - **Grid Search** (GridSearchCV) → tries *all combinations*.
  - **Randomized Search** (RandomizedSearchCV) → tries *random subset of combinations*.

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
import numpy as np
param_grid = {
    "n_estimators": [100, 200, 500],
    "max_depth": [None, 10, 20],
    "min_samples_split": [2, 5, 10]
}

# Grid Search
grid_search = GridSearchCV(
    estimator=rf,
    param_grid=param_grid,
    cv=5,
    scoring="f1",
    n_jobs=-1,
    verbose=2
)
grid_search.fit(X_train, y_train)
print("Best Params (Grid):", grid_search.best_params_)
param_dist = {
    "n_estimators": np.arange(100, 1000, 100),
    "max_depth": [None, 5, 10, 20, 30],
    "min_samples_split": np.arange(2, 11)
}
random_search = RandomizedSearchCV( #  Randomized Search
    estimator=rf,
    param_distributions=param_dist,
    n_iter=20,
    cv=5,
    scoring="f1",
    n_jobs=-1,
    random_state=42,
    verbose=2
)
random_search.fit(X_train, y_train)
print("Best Params (Random):", random_search.best_params_)
```

Grid and Random Search Parameters

Parameter	Description	Possible Values
estimator	The model (e.g., RandomForestClassifier())	Any scikit-learn model
param_grid	Dict of hyperparameters for tuning	Example: {"n_estimators": [100,200], "max_depth": [5,10]}
param_distributions (RandomizedSearchCV)	Like param_grid but allows distributions instead of fixed values	scipy.stats.randint(10,100), lists, np.arange
cv	Number of folds in cross-validation	int (e.g., 5, 10)
scoring	Metric to optimize	"accuracy", "f1", "roc_auc", "neg_mean_squared_error", etc.
n_jobs	Number of parallel jobs	-1 = use all CPUs
verbose	Logging level	0 (silent), 1, 2 (more detail)
n_iter (RandomizedSearchCV)	Number of parameter settings sampled	int (e.g., 20, 50)
refit	Retrain best model on full training data	True (default)
random_state	Ensures reproducibility	int (e.g., 42)

3. Recommended ML Workflow

1. **Initial Data Split**
    - Always split your dataset into **training** and a **holdout test set**.
    - The test set is only used once at the very end for unbiased evaluation.
2. **Hyperparameter Tuning (with Cross-Validation)**
    - Use **Random Search + CV** first → to quickly explore a wide hyperparameter space.
    - Then use **Grid Search + CV** in a narrower range → to precisely fine-tune the best parameters.
3. **Final Evaluation**
    - Retrain the best model on the **entire training set**.
    - Test once on the **unseen holdout set** for the final performance score.

Adaptations by Dataset Size

- **Small Datasets (hundreds–thousands samples)**
    - Use **Cross-Validation** instead of a single split (more reliable).
    - Grid Search is feasible, Random Search also effective.
    - Prefer **simple models** (linear, SVM, Naive Bayes). Avoid deep learning (overfits easily).
- **Medium Datasets (10k–few million samples)**
    - Fixed **train/validation/test split** works well. CV still possible.
    - **Random Search** preferred for efficiency
    - Good models: **tree-based (RF, XGBoost, GBM), SVMs, medium-sized neural nets**.
- **Large Datasets (millions+ samples)**
    - Standard **train/validation/test split** (no need for CV).
    - Hyperparameter search: **Random Search, Bayesian Optimization, AutoML**.
    - Best models: **Deep learning**, which thrives on massive data.

4. Full Classification ML Model Code Example with Metrics

```
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, learning_curve
from sklearn.metrics import (
    classification_report, confusion_matrix,
    roc_curve, auc, precision_recall_curve
)
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.ensemble import RandomForestClassifier
import pandas as pd

# Example dataset
df = pd.read_csv("data.csv")
X = df.drop("target", axis=1)
y = df["target"]

# Split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# Preprocessing
numeric_features = ["age", "salary"]
categorical_features = ["city"]

preprocessor = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), numeric_features),
        ("cat", OneHotEncoder(handle_unknown="ignore"), categorical_features)
    ]
)

# Pipeline
pipe = Pipeline([
    ("preprocess", preprocessor),
    ("model", RandomForestClassifier(random_state=42))
])

# Grid Search
param_grid = {
    "model__n_estimators": [100, 200, 500],
    "model__max_depth": [None, 10, 20],
    "model__min_samples_split": [2, 5, 10]
}
```

```python
grid = GridSearchCV(pipe, param_grid, cv=5, scoring="accuracy", n_jobs=-1, verbose=2)
grid.fit(X_train, y_train)

best_model = grid.best_estimator_
print("Best Params:", grid.best_params_)

# Predictions
y_pred = best_model.predict(X_test)
y_proba = best_model.predict_proba(X_test)[:,1]

# 1. Classification Report
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# 2. Confusion Matrix Heatmap
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.show()

# 3. ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_proba)
roc_auc = auc(fpr, tpr)
plt.plot(fpr, tpr, label=f"AUC={roc_auc:.2f}")
plt.plot([0,1], [0,1], "k--")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()

# 4. Precision-Recall Curve
prec, rec, _ = precision_recall_curve(y_test, y_proba)
plt.plot(rec, prec, label="PR Curve")
plt.xlabel("Recall")
plt.ylabel("Precision")
plt.title("Precision-Recall Curve")
plt.legend()
plt.show()
```

```
# 5. Learning Curves
train_sizes, train_scores, val_scores = learning_curve(
    best_model, X_train, y_train, cv=5, scoring="accuracy",
    train_sizes=np.linspace(0.1, 1.0, 5), n_jobs=-1
)

train_mean = train_scores.mean(axis=1)
val_mean = val_scores.mean(axis=1)

plt.plot(train_sizes, train_mean, label="Training Accuracy")
plt.plot(train_sizes, val_mean, label="Validation Accuracy")
plt.xlabel("Training Size")
plt.ylabel("Accuracy")
plt.title("Learning Curve")
plt.legend()
plt.show()
```

5. Full Regression ML Model Code Example with Metrics

```
# Import libraries
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import (
    mean_absolute_error, mean_squared_error, r2_score, mean_absolute_percentage_error,
    mean_squared_log_error
)
import numpy as np
import pandas as pd
from math import sqrt

# 1. Generate sample regression dataset
X, y = make_regression(n_samples=300, n_features=3, noise=15, random_state=42)

# Split data into train/test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 2. Train model
model = LinearRegression()
model.fit(X_train, y_train)

# 3. Make predictions
y_pred = model.predict(X_test)

# 4. Compute metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = sqrt(mse)
r2 = r2_score(y_test, y_pred)

# Adjusted R² formula
n = len(y_test)
p = X_test.shape[1]
adj_r2 = 1 - (1 - r2) * ((n - 1) / (n - p - 1))

# MAPE (in %)
mape = mean_absolute_percentage_error(y_test, y_pred) * 100

# MSLE — requires non-negative predictions and targets
y_true_pos = np.maximum(y_test, 0)
y_pred_pos = np.maximum(y_pred, 0)
msle = mean_squared_log_error(y_true_pos + 1, y_pred_pos + 1)
```

```python
# 5. Store all metrics in a DataFrame
metrics = {
    'MAE': mae,
    'MSE': mse,
    'RMSE': rmse,
    'R²': r2,
    'Adjusted R²': adj_r2,
    'MAPE (%)': mape,
    'MSLE': msle
}

results_df = pd.DataFrame(metrics, index=['Linear Regression']).T
print("\n=== Regression Model Evaluation ===\n")
print(results_df.round(4))
```

6. Full Clustering ML Model Code Example with Metrics

```
# Import libraries
from sklearn.datasets import make_moons
from sklearn.cluster import DBSCAN, KMeans
from sklearn.metrics import (
    silhouette_score, davies_bouldin_score, calinski_harabasz_score,
    adjusted_rand_score, normalized_mutual_info_score,
    homogeneity_score, completeness_score, v_measure_score, fowlkes_mallows_score
)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 1. Create dataset (with ground truth for external metrics)
X, y_true = make_moons(n_samples=500, noise=0.05, random_state=42)

# 2. Apply DBSCAN
dbscan = DBSCAN(eps=0.2, min_samples=5)
y_db = dbscan.fit_predict(X)

# 3. Apply KMeans (for comparison)
kmeans = KMeans(n_clusters=2, random_state=42, n_init='auto')
y_km = kmeans.fit_predict(X)

# 4. Function to compute metrics safely (avoids issues with single cluster)
def compute_metrics(X, labels, y_true=None):
    mask = labels != -1  # remove noise for DBSCAN
    if len(set(labels[mask])) <= 1:
        return {'Silhouette': np.nan, 'Davies-Bouldin': np.nan, 'Calinski-Harabasz': np.nan,
                'ARI': np.nan, 'NMI': np.nan, 'Homogeneity': np.nan, 'Completeness': np.nan,
                'V-Measure': np.nan, 'FMI': np.nan}

    results = {
        # Internal metrics
        'Silhouette': silhouette_score(X[mask], labels[mask]),
        'Davies-Bouldin': davies_bouldin_score(X[mask], labels[mask]),
        'Calinski-Harabasz': calinski_harabasz_score(X[mask], labels[mask]),
    }

    # External metrics if true labels provided
    if y_true is not None:
        results.update({
            'ARI': adjusted_rand_score(y_true[mask], labels[mask]),
            'NMI': normalized_mutual_info_score(y_true[mask], labels[mask]),
            'Homogeneity': homogeneity_score(y_true[mask], labels[mask]),
            'Completeness': completeness_score(y_true[mask], labels[mask]),
            'V-Measure': v_measure_score(y_true[mask], labels[mask]),
            'FMI': fowlkes_mallows_score(y_true[mask], labels[mask])
        })
```

```python
    return results

# 5. Compute metrics for both models
metrics_db = compute_metrics(X, y_db, y_true)
metrics_km = compute_metrics(X, y_km, y_true)

# 6. Display metrics in a table
results_df = pd.DataFrame([metrics_db, metrics_km], index=['DBSCAN', 'KMeans'])
print("\n=== Clustering Evaluation Metrics ===\n")
print(results_df.round(3))

# 7. Visualize clusters
fig, axs = plt.subplots(1, 2, figsize=(10, 4))
axs[0].scatter(X[:, 0], X[:, 1], c=y_db, cmap='plasma', s=30)
axs[0].set_title('DBSCAN Clustering')
axs[1].scatter(X[:, 0], X[:, 1], c=y_km, cmap='plasma', s=30)
axs[1].set_title('KMeans Clustering')
plt.show()
```

18. Model Visualization

Code

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression, make_classification, make_blobs
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.decomposition import PCA

# --- Configuration ---
plt.style.use('seaborn-v0_8-darkgrid')
np.random.seed(42)


def reduce_features(X, n_components=2):
    """
    Reduce X to n_components using PCA if needed.
    Returns transformed data and the fitted PCA (or None if not applied).
    """
    if X.shape[1] > n_components:
        pca = PCA(n_components=n_components, random_state=42)
        return pca.fit_transform(X), pca
    else:
        return X, None


def plot_model_fit(model_type, X_train, y_train=None, X_test=None, y_test=None, model=None, title=""):
    """
    Unified visualization for Regression, Classification, and Clustering.
    - Regression: plots fit curve on reduced 1D features
    - Classification: plots decision boundaries on reduced 2D features
    - Clustering: plots clusters on reduced 2D features
    """
    plt.figure(figsize=(10, 6))

    # ================= REGRESSION =================
    if model_type == 'regression':
        X_train_red, pca = reduce_features(X_train, n_components=1)
        X_test_red = pca.transform(X_test) if pca and X_test is not None else X_test

        model.fit(X_train_red, y_train)
```

```
    # Generate smooth prediction
    X_range = np.linspace(X_train_red.min(), X_train_red.max(), 500).reshape(-1, 1)
    y_pred_range = model.predict(X_range)
    y_test_pred = model.predict(X_test_red) if X_test_red is not None else None

    # Plot
    plt.scatter(X_train_red, y_train, color='blue', label='Train Data', alpha=0.6)
    if X_test_red is not None:
        plt.scatter(X_test_red, y_test, color='red', label='Test Data', alpha=0.8, marker='X')
    plt.plot(X_range, y_pred_range, color='green', linewidth=3,
             label=f'Model Prediction ({model.__class__.__name__})')

    if X_test_red is not None:
        for i in range(len(X_test_red)):
            plt.plot([X_test_red[i], X_test_red[i]], [y_test[i], y_test_pred[i]],
                     color='orange', linestyle='--', alpha=0.5)

    plt.title(f'{title} (Regression, reduced to 1D)')
    plt.xlabel('Reduced Feature')
    plt.ylabel('Target y')

# ================= CLASSIFICATION =================
elif model_type == 'classification':
    X_train_red, pca = reduce_features(X_train, n_components=2)
    X_test_red = pca.transform(X_test) if pca and X_test is not None else X_test

    model.fit(X_train_red, y_train)

    # Meshgrid for decision boundary
    x_min, x_max = X_train_red[:, 0].min() - 0.5, X_train_red[:, 0].max() + 0.5
    y_min, y_max = X_train_red[:, 1].min() - 0.5, X_train_red[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
                         np.arange(y_min, y_max, 0.02))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8, cmap=plt.cm.RdBu)
    plt.scatter(X_train_red[:, 0], X_train_red[:, 1], c=y_train, cmap=plt.cm.RdBu,
                edgecolor='k', s=80, label='Train Data')

    if X_test_red is not None:
        y_test_pred = model.predict(X_test_red)
        plt.scatter(X_test_red[:, 0], X_test_red[:, 1], c=y_test_pred, cmap=plt.cm.RdBu,
                    marker='P', edgecolor='black', linewidth=1.5, s=150, label='Test Predictions (P)')

    plt.title(f'{title} (Classification, reduced to 2D)')
    plt.xlabel('PC1')
    plt.ylabel('PC2')
```

```
# ================= CLUSTERING =================
elif model_type == 'clustering':
    X_red, pca = reduce_features(X_train, n_components=2)

    model.fit(X_red)
    labels = model.labels_
    centroids = getattr(model, "cluster_centers_", None)

    plt.scatter(X_red[:, 0], X_red[:, 1], c=labels, cmap='viridis',
                s=100, alpha=0.8, edgecolor='k')

    if centroids is not None:
        plt.scatter(centroids[:, 0], centroids[:, 1], marker='*', s=350,
                    c=np.unique(labels), cmap='viridis',
                    edgecolor='red', linewidth=2, label='Centroids')

    plt.title(f'{title} (Clustering, reduced to 2D)')
    plt.xlabel('PC1')
    plt.ylabel('PC2')

plt.legend()
plt.tight_layout()
```

Usage Example

```
# --- Main Execution for Demonstrations ---
if __name__ == '__main__':
    print("Generating visualizations for Regression, Classification, and Clustering...")

    # ========================================
    # 1. REGRESSION (n features)
    # ========================================
    X, y, coef = make_regression(n_samples=100, n_features=5, noise=20, coef=True, random_state=42)
    y = y + np.sum(X[:, :2]**2, axis=1) * 5  # add nonlinearity
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

    linear_model = LinearRegression()
    plot_model_fit('regression', X_train, y_train, X_test, y_test, linear_model,
                   "Linear Regression with 5 Features")

    poly_model = make_pipeline(PolynomialFeatures(degree=3), LinearRegression())
    plot_model_fit('regression', X_train, y_train, X_test, y_test, poly_model,
                   "Polynomial Regression with 5 Features")
```

```
    # ========================================
    # 2. CLASSIFICATION (n features)
    # ========================================

X_cls, y_cls = make_classification(n_samples=200, n_features=5, n_informative=3,
                                   n_redundant=0, n_classes=2, random_state=42)
    X_cls_train, X_cls_test, y_cls_train, y_cls_test = train_test_split(X_cls, y_cls, test_size=0.3,
random_state=42)

    log_reg_model = LogisticRegression()
    plot_model_fit('classification', X_cls_train, y_cls_train, X_cls_test, y_cls_test,
                   log_reg_model, "Logistic Regression with 5 Features")

    knn_model = KNeighborsClassifier(n_neighbors=5)
    plot_model_fit('classification', X_cls_train, y_cls_train, X_cls_test, y_cls_test,
                   knn_model, "KNN with 5 Features")

    # ========================================
    # 3. CLUSTERING (n features)
    # ========================================
    X_clu, _ = make_blobs(n_samples=300, n_features=5, centers=4, cluster_std=1.5, random_state=42)
    kmeans_model = KMeans(n_clusters=4, random_state=42, n_init=10)
    plot_model_fit('clustering', X_clu, model=kmeans_model,
                   title="KMeans Clustering with 5 Features")

    plt.show()
```

19. Model Saving

After training, your model's parameters (weights, structure, preprocessing pipeline, etc.) represent the learned knowledge. You don't want to lose that or retrain each time you want to make predictions — especially for large models that take hours to train.
So, saving the model allows you to **store it permanently** and **reload it instantly** later.

What Exactly Gets Saved

You can save different components depending on your workflow:

1. **Model Weights / Parameters** – The learned values during training.
    - Example: coefficients in linear regression, decision tree splits, or neural network weights.
2. **Complete Pipeline** – Combining preprocessing + model ensures consistent input handling.

Tools for Saving Models (in Python)

1. Using Pickle (General Python Method)

Pickle can serialize any Python object, including ML models.

```
import pickle

# Save model
with open('model.pkl', 'wb') as f:
   pickle.dump(model, f)

# Load model
with open('model.pkl', 'rb') as f:
   loaded_model = pickle.load(f)
```

⚠️ *Note:* Use carefully — Pickle files can be unsafe if loaded from untrusted sources.

2. Using Joblib (Optimized for Large Models)

Joblib is preferred for scikit-learn models because it handles large numpy arrays efficiently.

```
import joblib

# Save model
joblib.dump(model, 'model.joblib')

# Load model
loaded_model = joblib.load('model.joblib')
```

---

3. Saving the Full Pipeline

If you used a Pipeline from scikit-learn:

```
from sklearn.pipeline import Pipeline
import joblib

# Assume 'pipeline' includes preprocessing + model
joblib.dump(pipeline, 'final_pipeline.joblib')

# Load it back
pipeline_loaded = joblib.load('final_pipeline.joblib')
```

That’s ideal for deployment, since it ensures the same data transformations occur during inference.

20. Model Deployment through Streamlit

After training and saving your ML model, the next step is to make it **accessible to others** — so users can input new data and instantly get predictions **without needing to understand the code**.

Deployment is the process of integrating your model into a usable application — such as:

- A **web app** (Streamlit, Flask, FastAPI)
- A **mobile app** (via APIs)
- A **cloud service** (AWS, GCP, Azure, Hugging Face, etc.)

Why Use Streamlit for Deployment

Streamlit is one of the most popular and beginner-friendly tools for ML deployment because:

- It turns **Python scripts into web apps** in minutes.
- No frontend knowledge (HTML/CSS/JS) is required.
- It supports **interactive widgets**, **live model inference**, and **visualization** directly from Python.
- You can **share your model online** with one command (streamlit share or via Streamlit Cloud).

General Deployment Workflow with Streamlit

1️⃣ Import Streamlit and other libraries

2️⃣ Load the **saved model** (e.g., .joblib, .pkl)

3️⃣ Design the **user interface** using Streamlit widgets

4️⃣ Collect user input

5️⃣ Preprocess input (same as during training)

6️⃣ Make predictions using the model

7️⃣ Display output (numerical result, graph, message, etc.)

8️⃣ (Optional) Deploy to Streamlit Cloud or another host

Streamlit Function Table (Most Useful Functions)

Category	**Function**	**Description**
Page Setup	st.set_page_config(title, layout)	Configure page title, icon, and layout
Text Display	st.title(), st.header(), st.subheader(), st.write(), st.markdown()	Display titles, formatted text, and markdown
Input Widgets	st.text_input(), st.number_input(), st.slider(), st.selectbox(), st.radio(), st.checkbox()	Capture user input
Buttons & Actions	st.button(), st.download_button()	Trigger actions or allow file downloads
File Uploading	st.file_uploader()	Allow users to upload files (CSV, image, etc.)
Data Display	st.dataframe(), st.table()	Display data or model outputs
Charts	st.line_chart(), st.bar_chart(), st.pyplot(), st.altair_chart()	Visualize results
Media	st.image(), st.audio(), st.video()	Display media files
Layout	st.sidebar, st.columns(), st.tabs()	Organize the UI layout
Feedback	st.progress(), st.spinner(), st.success(), st.error(), st.warning(), st.info()	Display app status and messages
Caching	@st.cache_data, @st.cache_resource	Speed up loading (e.g., caching models)
Session State	st.session_state	Store values persistently between interactions

General ML Deployment App

Let's put it all together into a **Streamlit app** for a general machine learning model.

```
# app.py
import streamlit as st
import joblib
import numpy as np
import pandas as pd

# --- PAGE CONFIG ---
st.set_page_config(page_title="ML Model Deployment", layout="centered")

# --- TITLE ---
st.title("🏠 Machine Learning Model Deployment App")
st.write("This web app allows you to input features and get model predictions instantly!")

# --- LOAD MODEL ---
@st.cache_resource
def load_model():
   model = joblib.load('model.joblib')
   return model

model = load_model()

# --- SIDEBAR INPUTS ---
st.sidebar.header("Enter Input Features")

feature1 = st.sidebar.number_input("Feature 1", min_value=0.0, max_value=100.0, value=10.0)
feature2 = st.sidebar.slider("Feature 2", 0, 50, 25)
feature3 = st.sidebar.selectbox("Feature 3", ["A", "B", "C"])

# Convert categorical input to numeric (example)
feature3_encoded = {"A": 0, "B": 1, "C": 2}[feature3]

# Create DataFrame for model
input_data = pd.DataFrame([[feature1, feature2, feature3_encoded]],
                          columns=['Feature1', 'Feature2', 'Feature3'])

# --- PREDICTION BUTTON ---
if st.button("🔮 Predict"):
   with st.spinner("Predicting..."):
      prediction = model.predict(input_data)
      st.success(f"✅ Model Prediction: {prediction[0]:.2f}")

# --- OPTIONAL: Display Input Data ---
with st.expander("Show Input Data"):
   st.dataframe(input_data)
```

▶️ **Run It Locally**

```
streamlit run app.py
```

Then visit **http://localhost:8501** in your browser.

Deploy Online

1. Push your app to GitHub.
2. Add requirement.txt **(includes libraries used)** to the repo.
3. Go to streamlit.io/cloud.
4. Click **“New app”** → select your repo → deploy.

Conclusion

This document has provided a comprehensive walkthrough of the **data preprocessing pipeline** and **machine learning workflow**, starting from understanding raw data, cleaning, and transforming it, all the way to handling missing values, encoding categorical variables, scaling, splitting datasets, and dealing with imbalanced data. It has also covered feature engineering, outlier handling, exploratory data analysis, and detailed approaches to model building. Special emphasis was placed on **model validation techniques** such as manual splits, cross-validation, and hyperparameter tuning using Grid Search and Randomized Search. Finally, evaluation metrics and visualization tools-including classification reports, confusion matrices, ROC and precision-recall curves, and learning curves-were highlighted to assess and improve model performance.

By following the structured steps and best practices described here, learners and practitioners can build robust, interpretable, and high-performing machine learning models. The goal of this document is not only to guide the reader through the **"how"** of preprocessing and modeling but also to strengthen their understanding of the **"why"**, enabling them to confidently apply these techniques in real-world projects.

Feedback & Contribution

This is a living document. If you have feedback, suggestions, or additional insights that could improve it, I welcome your input. Sharing ideas helps keep this material accurate, relevant, and valuable for everyone.

Copyright & Usage

© 2025 [Youssef Amgad Elkhatib].
This document is intended **for learning and reference purposes only**.
You are free to read, use, and share the content for personal or educational use, but:

- **Do not copy or republish this document as your own.**
- Always provide proper credit when referencing.
- Commercial use or redistribution without permission is not allowed.

You may not reproduce this document in whole or in part without attribution. Suggestions and feedback are welcome, and contributors will be acknowledged, but copyright remains with the author.

Acknowledgments

This document is authored and copyrighted by **[Youssef Amgad Elkhatib]**.
Special thanks to the contributors who provided valuable feedback and suggestions for improvements:

Name	Contribution