📌 Handling Data Before Training for ML

Categories:

1. Handling Missing Data
2. Handling Categorical Variables
3. Feature Scaling & Normalization
4. Feature Engineering
5. Outlier Detection and Removal
6. Dimensionality Reduction
7. Handling Imbalanced Data
8. Feature Selection

For each category, I will **deep dive** into the **FOUR** most important methods.

1️⃣ Handling Missing Data

Main Methods

1. Mean/Median Imputation
2. Mode Imputation
3. Forward/Backward Fill
4. KNN Imputation

🔷 Method 1: Mean/Median Imputation

🧠 Intuitive Understanding

Missing values are replaced with the **mean** or **median** of the column.

- **Mean** works best for normally distributed data.
- **Median** is better for skewed distributions.

📌 When to Use

- When data is **numerical** and has missing values.
- Works well when missing data is **randomly** distributed (Missing Completely At Random - MCAR).

📌 Algorithms That Use It

- **Regression-based models (Linear Regression, Logistic Regression, Neural Networks, SVM, k-NN)**
- **Clustering models (K-Means, GMM)**
- **Not needed for tree-based models (Random Forest, XGBoost, Decision Trees)**

🧮 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each column in dataset:
    If column contains missing values:
        If strategy == "mean":
            Replace missing values with column mean
        If strategy == "median":
            Replace missing values with column median
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
import numpy as np

def impute_missing_values(data, strategy="mean"):
    """
    Imputes missing values using mean or median.

    Parameters:
    - data (np.array): Dataset containing NaNs.
    - strategy (str): "mean" or "median".

    Returns:
    - np.array: Dataset with missing values replaced.
    """
    data = np.copy(data)  # Create a copy to avoid modifying original data

    for i in range(data.shape[1]):  # Loop over each column
        col = data[:, i]

        if np.isnan(col).any():  # Check if there are missing values
            if strategy == "mean":
                replacement = np.nanmean(col)  # Compute mean ignoring NaNs
            elif strategy == "median":
                replacement = np.nanmedian(col)  # Compute median ignoring NaNs
            else:
                raise ValueError("Invalid strategy. Use 'mean' or 'median'.")

            # Replace NaNs with computed replacement value
            data[:, i] = np.where(np.isnan(col), replacement, col)

    return data
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.impute import SimpleImputer

def sklearn_impute_missing_values(data, strategy="mean"):
    """
    Imputes missing values using Scikit-Learn's SimpleImputer.

    Parameters:
    - data (np.array): Dataset containing NaNs.
    - strategy (str): "mean" or "median".

    Returns:
    - np.array: Dataset with missing values replaced.
    """
    imputer = SimpleImputer(strategy=strategy)
    return imputer.fit_transform(data)
```

🔷 Method 2: Mode Imputation

🧠 Intuitive Understanding

Missing values in **categorical** data are replaced with the **most frequent value** (mode).

📌 When to Use

- When data is **categorical**.
- Works well when missing values are **randomly distributed**.

📌 Algorithms That Use It

- **Logistic Regression, Naive Bayes, SVM (for categorical data)**
- **Tree-based models (Random Forest, XGBoost, Decision Trees) accept categorical values but benefit from imputation**

🧮 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each categorical column in dataset:
    If column contains missing values:
        Compute the most frequent value (mode)
        Replace missing values with mode
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
def impute_mode(data):
    """
    Replaces missing categorical values with the most frequent value (mode).

    Parameters:
    - data (np.array): Categorical dataset with missing values.

    Returns:
    - np.array: Dataset with missing values replaced.
    """
    data = np.copy(data)  # Create a copy to avoid modifying original data

    for i in range(data.shape[1]):  # Iterate over columns
        col = data[:, i]
        if np.isnan(col).any():  # If there are missing values
            unique, counts = np.unique(col[~np.isnan(col)], return_counts=True)
            mode = unique[np.argmax(counts)]  # Find the most frequent value
            data[:, i] = np.where(np.isnan(col), mode, col)  # Replace NaNs with mode

    return data
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
def sklearn_impute_mode(data):
    """
    Imputes missing values using Scikit-Learn's SimpleImputer with "most_frequent".

    Parameters:
    - data (np.array): Categorical dataset with missing values.

    Returns:
    - np.array: Dataset with missing values replaced.
    """
    imputer = SimpleImputer(strategy="most_frequent")
    return imputer.fit_transform(data)
```

🔷 Method 3: Forward/Backward Fill

🧠 Intuitive Understanding

- **Forward Fill**: Uses the **previous** value to fill missing data.
- **Backward Fill**: Uses the **next** value to fill missing data.

📌 When to Use

- When data is **time-series**.
- When missing values follow a **pattern** and adjacent values are useful.

📌 Algorithms That Use It

- **Recurrent Neural Networks (RNNs), LSTMs for sequential data**
- **Time-series forecasting models (ARIMA, Prophet, etc.)**

🧮 Mathematical Breakdown (Pseudocode)

vbnet Copy Edit

```
For each row in dataset:
    If value is missing:
        If forward fill:
            Replace with previous row's value
        If backward fill:
            Replace with next row's value
```

Fully Commented NumPy Implementation

python Copy Edit

```python
def forward_fill(data):
    """
    Replaces missing values using the previous row's value (forward fill).

    Parameters:
    - data (np.array): Dataset with missing values.

    Returns:
    - np.array: Forward-filled dataset.
    """
    for i in range(1, len(data)):  # Start from second row
        mask = np.isnan(data[i])
        data[i, mask] = data[i - 1, mask]  # Replace NaN with previous row value

    return data
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
import pandas as pd

def sklearn_forward_fill(df):
    """
    Forward fills missing values using Pandas.

    Parameters:
    - df (pd.DataFrame): Dataset with missing values.

    Returns:
    - pd.DataFrame: Forward-filled dataset.
    """
    return df.fillna(method='ffill')
```

🔷 Method 4: KNN Imputation

🧠 Intuitive Understanding

K-Nearest Neighbors (KNN) **predicts** missing values using the most similar rows in the dataset. Instead of replacing missing values with a fixed statistic (like mean/median), it **estimates values based on nearby data points**.

📌 When to Use

- When data has a **non-random pattern** of missing values.
- When other features in the dataset can help **predict the missing value**.

📌 Algorithms That Use It

- **Regression models (Linear Regression, Logistic Regression, SVMs, Neural Networks, k-NN, etc.)**
- **Clustering models (K-Means, Gaussian Mixture Models)**
- **Not necessary for tree-based models (Decision Trees, XGBoost, Random Forests).**

🧮 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each row with missing values:
    Find k nearest neighbors based on Euclidean distance.
    Compute the mean/median of the k neighbors for the missing column.
    Replace the missing value with this computed value.
```

python

Copy Edit

```python
from scipy.spatial import distance

def knn_impute(data, k=5):
    """
    Uses k-Nearest Neighbors to impute missing values.

    Parameters:
    - data (np.array): Dataset containing missing values (NaNs).
    - k (int): Number of nearest neighbors to use.

    Returns:
    - np.array: Dataset with missing values imputed.
    """
    data = np.copy(data)  # Avoid modifying original dataset
    rows, cols = data.shape

    for i in range(rows):  # Iterate through each row
        nan_cols = np.isnan(data[i])  # Identify missing values

        if np.any(nan_cols):  # If there are missing values in the row
            valid_rows = ~np.isnan(data).any(axis=1)  # Find complete rows
            complete_data = data[valid_rows]  # Subset without missing values

            # Compute distances to all complete rows
            dists = np.array([distance.euclidean(data[i, ~nan_cols], row[~nan_cols])
                              for row in complete_data])

            # Get k nearest neighbors
            knn_indices = np.argsort(dists)[:k]  # Indices of k nearest neighbors
            knn_values = complete_data[knn_indices][:, nan_cols]  # Get values

            # Replace missing values with mean of k-nearest neighbors
            data[i, nan_cols] = np.nanmean(knn_values, axis=0)

    return data
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.impute import KNNImputer

def sklearn_knn_impute(data, k=5):
    """
    Uses Scikit-Learn's KNNImputer to impute missing values.

    Parameters:
    - data (np.array): Dataset with missing values.
    - k (int): Number of nearest neighbors.

    Returns:
    - np.array: Dataset with imputed values.
    """
    imputer = KNNImputer(n_neighbors=k)
    return imputer.fit_transform(data)
```

2️⃣ Handling Categorical Variables

Main Methods

1. **Label Encoding**
2. **One-Hot Encoding**
3. **Ordinal Encoding**
4. **Binary Encoding**

🔷 Method 1: Label Encoding

🧠 Intuitive Understanding

Label Encoding assigns each unique categorical value an **integer**.

📌 When to Use

- When categories have **no natural ordering** (e.g., "red", "blue", "green").
- **Only for tree-based models** (other models may be misled by numerical ordering).

📌 Algorithms That Use It

- **Decision Trees, Random Forest, XGBoost (work well with label encoding).**
- **Do NOT use it for Linear Regression or Neural Networks** (they might assume numerical relationships).

🧱 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each unique category:
    Assign a unique integer value.
    Replace categorical values in the dataset with corresponding integers.
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
def label_encode(data):
    """
    Encodes categorical data using label encoding.

    Parameters:
    - data (np.array): Categorical column.

    Returns:
    - np.array: Label encoded values.
    - dict: Mapping of original categories to integers.
    """
    unique_vals = np.unique(data)
    encoding_map = {val: idx for idx, val in enumerate(unique_vals)}
    return np.vectorize(encoding_map.get)(data), encoding_map
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import LabelEncoder

def sklearn_label_encode(data):
    """
    Performs label encoding using Scikit-Learn.

    Parameters:
    - data (np.array): Categorical column.

    Returns:
    - np.array: Encoded values.
    - dict: Mapping of categories to integers.
    """
    le = LabelEncoder()
    return le.fit_transform(data), le.classes_
```

🔷 Method 2: One-Hot Encoding

🧠 Intuitive Understanding

Converts categorical values into **binary vectors**, with one-hot encoding **each category gets its own column**.

📌 When to Use

- When categorical features have **no ordinal relationship**.
- Used for **Neural Networks, Linear Models, SVMs**.

📌 Algorithms That Use It

- **Neural Networks, Logistic Regression, SVMs, k-NN, Naive Bayes**
- **Not needed for tree-based models (Decision Trees, Random Forests, XGBoost).**

🧮 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each unique category:
    Create a new column (one per category).
    Set 1 if the category is present, 0 otherwise.
```

Fully Commented NumPy Implementation

python Copy Edit

```python
def one_hot_encode(data):
    """
    Performs one-hot encoding on a categorical column.

    Parameters:
    - data (np.array): Categorical column.

    Returns:
    - np.array: One-hot encoded representation.
    - dict: Mapping of categories to column indices.
    """
    unique_vals = np.unique(data)
    encoding_map = {val: i for i, val in enumerate(unique_vals)}
    encoded = np.zeros((len(data), len(unique_vals)))

    for i, val in enumerate(data):
        encoded[i, encoding_map[val]] = 1  # Set correct position to 1

    return encoded, encoding_map
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import OneHotEncoder

def sklearn_one_hot_encode(data):
    """
    Performs one-hot encoding using Scikit-Learn.

    Parameters:
    - data (np.array): Categorical column.

    Returns:
    - np.array: One-hot encoded representation.
    """
    ohe = OneHotEncoder(sparse=False)
    return ohe.fit_transform(data.reshape(-1, 1)), ohe.categories_
```

3️⃣ Feature Scaling & Normalization

Feature scaling ensures that numerical features are on the **same scale** to prevent some features from dominating the learning process.

Main Methods

1. **Standardization (Z-Score Normalization)**
2. **Min-Max Scaling (Normalization to [0,1])**
3. **Robust Scaling (Using IQR - robust to outliers)**
4. **Log Transform (Handles skewed distributions)**

🔷 Method 1: Standardization (Z-Score Scaling)

🧠 Intuitive Understanding

- **Centers data around 0** with a **standard deviation of 1**.
- Makes the data **normally distributed**.
- Formula:

$$x' = \frac{x - \mu}{\sigma}$$

where **μ** = mean, **σ** = standard deviation.

📌 When to Use

- When **features have different scales**.
- When data **follows a normal distribution**.

📌 Algorithms That Use It

- **Linear models (Linear Regression, Logistic Regression, SVM, PCA, k-NN, Neural Networks)**
- **Not needed for tree-based models (Decision Trees, Random Forests, XGBoost, etc.)**

🧱 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each column in dataset:
    Compute mean (μ) and standard deviation (σ).
    For each value x:
        Compute z-score: (x - μ) / σ.
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
def standardize_data(data):
    """
    Standardizes data using Z-score normalization.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Standardized data.
    """
    mean = np.mean(data, axis=0)  # Compute mean for each column
    std = np.std(data, axis=0)  # Compute standard deviation for each column

    return (data - mean) / std  # Apply Z-score normalization
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import StandardScaler

def sklearn_standardize(data):
    """
    Standardizes data using Scikit-Learn's StandardScaler.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Standardized data.
    """
    scaler = StandardScaler()
    return scaler.fit_transform(data)
```

🔷 Method 2: Min-Max Scaling (Normalization to [0,1])

🧠 Intuitive Understanding

- Scales values between 0 and 1.
- Useful for data that has a fixed range.
- Formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

📌 When to Use

- When features have different scales but should remain proportional.
- When data does not follow a normal distribution.

📌 Algorithms That Use It

- Neural Networks (especially for deep learning)
- K-Nearest Neighbors (k-NN)
- Distance-based algorithms (e.g., K-Means, PCA, SVMs)

🧱 Mathematical Breakdown (Pseudocode)

arduino Copy Edit

```
For each column in dataset:
    Compute min and max.
    For each value x:
        Compute scaled value: (x - min) / (max - min).
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
def normalize_data(data):
    """
    Normalizes data using Min-Max scaling.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Normalized data between 0 and 1.
    """
    min_val = np.min(data, axis=0)  # Compute min for each column
    max_val = np.max(data, axis=0)  # Compute max for each column

    return (data - min_val) / (max_val - min_val)  # Apply Min-Max scaling
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import MinMaxScaler

def sklearn_normalize(data):
    """
    Normalizes data using Scikit-Learn's MinMaxScaler.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Normalized data.
    """
    scaler = MinMaxScaler()
    return scaler.fit_transform(data)
```

🔷 Method 3: Robust Scaling (Using IQR)

🧠 Intuitive Understanding

- **Robust to outliers** because it **scales using median and IQR** instead of mean and standard deviation.
- Formula:

$$x' = \frac{x - \text{median}}{\text{IQR}}$$

where **IQR** = 75th percentile - 25th percentile.

📌 When to Use

- When **data has extreme outliers** that would distort standardization or Min-Max scaling.

📌 Algorithms That Use It

- **Logistic Regression, SVMs, k-NN, Clustering Models**
- **Works well when feature distributions are highly skewed.**

🧮 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each column in dataset:
    Compute median and IQR.
    For each value x:
        Compute scaled value: (x - median) / IQR.
```

Copy Edit

```python
def robust_scale(data):
    """
    Scales data using the IQR method (Robust Scaling).

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Robustly scaled data.
    """
    median = np.median(data, axis=0)  # Compute median for each column
    q75, q25 = np.percentile(data, [75, 25], axis=0)  # Compute quartiles
    iqr = q75 - q25  # Compute IQR

    return (data - median) / iqr  # Apply robust scaling
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import RobustScaler

def sklearn_robust_scale(data):
    """
    Scales data using Scikit-Learn's RobustScaler.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Robustly scaled data.
    """
    scaler = RobustScaler()
    return scaler.fit_transform(data)
```

🔷 Method 4: Log Transform (Handles Skewed Data)

🧠 Intuitive Understanding

- **Reduces the impact of large outliers.**
- **Useful for skewed distributions** (e.g., income, population).
- Formula:

$$x' = \log(x + 1)$$

📌 When to Use

- When **data is highly skewed (e.g., exponential distributions)**.
- When **reducing variance in features** improves model performance.

📌 Algorithms That Use It

- **Linear models (Logistic Regression, SVMs, k-NN, PCA)**
- **Helps improve normality assumptions in regression models.**

🧮 Mathematical Breakdown (Pseudocode)

```perl
For each value x in dataset:
    Compute log(x + 1).
```

🛠️ Fully Commented NumPy Implementation

python Copy Edit

```python
def log_transform(data):
    """
    Applies log transformation to data.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Log-transformed data.
    """
    return np.log1p(data)  # log(x + 1) transformation
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
import numpy as np

def sklearn_log_transform(data):
    """
    Applies log transformation using NumPy.

    Parameters:
    - data (np.array): Numerical dataset.

    Returns:
    - np.array: Log-transformed data.
    """
    return np.log1p(data)
```

4️⃣ Feature Engineering

Feature Engineering is the process of creating **new features** from existing ones to improve model performance.

Main Methods

1. **Polynomial Features**
2. **Binning (Discretization)**
3. **Feature Interaction (Multiplicative & Additive Interactions)**
4. **Feature Extraction (Principal Component Analysis - PCA)**

🔷 Method 1: Polynomial Features

🧠 Intuitive Understanding

- **Expands numerical features** by adding polynomial terms.
- If a feature is x, then we add x^2, x^3, ... up to a degree d.
- Helps capture **non-linear relationships**.

📌 When to Use

- When **linear models are underfitting** and **higher-order terms improve performance**.
- When **relationships between variables are curved (quadratic, cubic, etc.)**.

📌 Algorithms That Use It

- **Linear Regression, Logistic Regression, SVM, k-NN, Neural Networks**
- **Not needed for tree-based models (Decision Trees, XGBoost, Random Forests).**

Mathematical Breakdown (Pseudocode)

vbnet Copy Edit

```
For each numerical feature in dataset:
    For degree d from 2 to max_degree:
        Compute x^d and add it as a new feature.
```

Fully Commented NumPy Implementation

python Copy Edit

```python
import numpy as np

def generate_polynomial_features(data, degree=2):
    """
    Generates polynomial features up to a given degree.

    Parameters:
    - data (np.array): Input numerical data (2D array).
    - degree (int): Maximum polynomial degree.

    Returns:
    - np.array: Data with additional polynomial features.
    """
    original_features = data.shape[1]
    poly_features = [data]  # Store original features

    for d in range(2, degree + 1):  # Generate powers of features
        poly_features.append(np.power(data, d))

    return np.hstack(poly_features)  # Stack all generated features horizontally
```

📌 Scikit-Learn Implementation

python Copy Edit

```python
from sklearn.preprocessing import PolynomialFeatures

def sklearn_generate_polynomial_features(data, degree=2):
    """
    Generates polynomial features using Scikit-Learn.

    Parameters:
    - data (np.array): Input numerical data.
    - degree (int): Maximum polynomial degree.

    Returns:
    - np.array: Data with additional polynomial features.
    """
    poly = PolynomialFeatures(degree)
    return poly.fit_transform(data)
```

🔷 Method 2: Binning (Discretization)

🧠 Intuitive Understanding

- **Groups continuous values into discrete bins** (e.g., age 0-20, 20-40, etc.).
- Reduces noise and improves interpretability.

📌 When to Use

- When features **vary over a large range** but categorical grouping makes sense.
- When **handling skewed distributions**.

📌 Algorithms That Use It

- **Decision Trees, Naive Bayes, Rule-Based Models**
- **Not recommended for deep learning or k-NN** (loses fine-grained information).

🧱 Mathematical Breakdown (Pseudocode)

sql Copy Edit

```
For each feature in dataset:
    Divide feature values into k bins.
    Replace values with corresponding bin labels.
```

```python
def bin_data(data, num_bins=5):
    """
    Performs binning on numerical features.

    Parameters:
    - data (np.array): Input numerical feature.
    - num_bins (int): Number of bins.

    Returns:
    - np.array: Binned categorical representation.
    """
    min_val = np.min(data)
    max_val = np.max(data)
    bins = np.linspace(min_val, max_val, num_bins + 1)  # Create bin edges
    return np.digitize(data, bins)  # Assign values to bins
```

📌 Scikit-Learn Implementation

```python
from sklearn.preprocessing import KBinsDiscretizer

def sklearn_bin_data(data, num_bins=5):
    """
    Performs binning using Scikit-Learn.

    Parameters:
    - data (np.array): Input numerical feature.
    - num_bins (int): Number of bins.

    Returns:
    - np.array: Binned categorical representation.
    """
    binning = KBinsDiscretizer(n_bins=num_bins, encode='ordinal', strategy='uniform
    return binning.fit_transform(data.reshape(-1, 1))
```