Lecture 7: Data Cleaning & Preprocessing: Mathematical Intuition and Practical Implementation

Dr. Ratnesh Srivastava, CSIT, GGV Bilaspur

July 19, 2025

Introduction

Data preprocessing is the critical first step in any data science pipeline. Real-world data is messy, incomplete, and inconsistent. Proper preprocessing ensures data quality and prepares datasets for machine learning algorithms. This lecture covers the mathematical intuition behind key preprocessing steps with practical examples.

1 Why Preprocess Data?

- **Missing values** cause computational errors and bias
- **Outliers** distort statistical measures and model performance
- **Feature scaling** ensures algorithms converge properly
- **Categorical variables** require numerical representation
- **Non-linear relationships** need transformation for linear models

2 Mathematical Foundations

2.1 1. Handling Missing Values

- **Problem:** Algorithms fail on NaN values
- **Median Imputation:** Robust to outliers

$$\text{Imputed Value} = \text{Median}(X)$$

- **Mean Imputation:** For normal distributions

$$\text{Imputed Value} = \bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

- **KNN Imputation:** Uses distance metrics

$$\text{Imputed Value} = \frac{1}{k}\sum_{j=1}^{k} X_j$$

2.2 2. Outlier Detection & Treatment

- **Z-Score:** For Gaussian distributions

$$Z = \frac{X - \mu}{\sigma}, \quad |Z| > 3 \text{ is outlier}$$

- **IQR Method:** Robust for skewed data

$$\text{Lower Bound} = Q1 - 1.5 \times IQR, \quad \text{Upper Bound} = Q3 + 1.5 \times IQR$$

2.3 3. Feature Scaling

- **Standardization:**

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

- **Min-Max Scaling:**

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

- **Robust Scaling:**

$$X_{\text{scaled}} = \frac{X - \text{Median}}{\text{IQR}}$$

2.4 4. Encoding Categorical Variables

- **One-Hot Encoding:** For nominal data

Color	Red	Green	Blue
Red	1	0	0
Green	0	1	0
Blue	0	0	1

- **Label Encoding:** For ordinal data

$$\{\text{Low}, \text{Medium}, \text{High}\} \to \{0, 1, 2\}$$

2.5 5. Feature Transformation

- **Log Transform:** For right-skewed data

$$X_{\text{new}} = \log(X + \epsilon)$$

- **Box-Cox Transform:** Generalizes log transform

$$X_{\text{new}} = \begin{cases} \frac{X^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(X) & \lambda = 0 \end{cases}$$

- **Polynomial Features:** Captures non-linear relationships

$$\text{New Features} = \{X, X^2, X^3, \dots\}$$

3 Comprehensive Example

We'll use a customer dataset to demonstrate all preprocessing steps.

3.1 Original Dataset

Age	Income ($)	Gender	Purchase History	Days Since Last Purchase	Purchased
25	50,000	Male	High	10	Yes
30	NaN	Female	Medium	60	No
22	200,000	Male	Low	5	Yes
35	75,000	Female	High	NaN	No
45	30,000	Other	Medium	365	No
28	1,000,000	Male	High	2	Yes

1lightyellow

3.2 Step 1: Handle Missing Values

- **Income:** Median = 75,000 → Impute NaN with 75,000
- **Days Since Last Purchase:** Mean = 88.4 → Impute NaN with 88.4

3.3 Step 2: Outlier Treatment (IQR Method)

- **Income:** Q1 = 50,000, Q3 = 200,000, IQR = 150,000
  Upper Bound = 200,000 + 1.5 × 150,000 = 425,000
  Cap 1,000,000 → 425,000
- **Days Since Last Purchase:** Q1 = 5, Q3 = 60, IQR = 55
  Upper Bound = 60 + 1.5 × 55 = 142.5
  Cap 365 → 142.5

3.4 Step 3: Feature Transformation

- **Box-Cox for Income:** Optimal = 0.25

$$\text{Income}_{\text{trans}} = \frac{\text{Income}^{0.25} - 1}{0.25}$$

- **Polynomial for Age:** Create Age^2 feature

3.5 Step 4: Encoding Categorical Variables

- **Gender:** One-Hot Encoding (drop first)
- **Purchase History:** Ordinal Encoding
  {Low:0, Medium:1, High:2}

3.6 Step 5: Feature Scaling (Standardization)

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

3.7 Final Preprocessed Dataset

1lightgreen

Age (sc)	Age² (sc)	Income (BC, sc)	Gen_F	Gen_O	PH	Days (sc)	Purchased
-0.73	0.12	-0.45	0	0	2	-0.78	Yes
-0.10	0.34	-0.20	1	0	1	0.16	No
-1.10	-0.55	0.75	0	0	0	-0.88	Yes
0.52	1.02	-0.20	1	0	2	0.70	No
1.76	2.89	-0.90	0	1	1	1.72	No
-0.35	-0.22	1.00	0	0	2	-0.93	Yes

sc = scaled, BC = Box-Cox, Gen_F = Gender_Female, Gen_O = Gender_Other, PH = Purchase History

Key Insights

- **Missing Values:** Always visualize missing patterns before imputation
- **Outliers:** IQR method is robust for skewed distributions
- **Scaling:** Essential for distance-based algorithms (KNN, SVM)
- **Encoding:** One-hot for nominal, label for ordinal data
- **Transformations:** Box-Cox for optimal normality, polynomials for non-linearity

Questions & Answers

1. **Q: Why use median instead of mean for missing value imputation?**
   A: Median is robust to outliers. If income data has extreme values (e.g., $1,000,000), the mean would be skewed, but median remains representative.

2. **Q: When should we use Box-Cox instead of log transform?**
   A: Box-Cox generalizes log transform by finding optimal . Use it when log transform doesn't fully normalize the distribution.

3. **Q: Why drop the first column in one-hot encoding?**
   A: To avoid the "dummy variable trap" (perfect multicollinearity). The dropped feature becomes the reference category.

4. **Q: How does polynomial features help linear models?**
   A: Linear models assume linear relationships. Polynomial terms (x^2, x^3) allow them to capture non-linear patterns.

5. **Q: Why cap outliers instead of removing them?**
   A: Capping preserves data points while reducing extreme value impact. Removal risks losing valuable information, especially in small datasets.

6. **Q: Should we scale before or after transformation?**
   A: Always transform first! Transformations change distributions, so scaling should be the final step before modeling.

7. **Q: When to use robust scaling vs standardization?**
   A: Use robust scaling when outliers are present. It uses median and IQR instead of mean and standard deviation.

8. **Q: How to handle new categories during one-hot encoding?**
   A: During deployment, new categories should be mapped to zeros in all columns or treated as a separate "unknown" category.

Best Practices

1. Always create a preprocessing pipeline for reproducibility
2. Visualize distributions before and after transformations
3. Split data before preprocessing to avoid data leakage
4. Document all preprocessing decisions
5. Validate preprocessing impact on model performance

Next Class: Implementing these techniques in Python with scikit-learn!

Lecture 7: Data Cleaning & Preprocessing: Mathematical Intuition and Practical Implementation

Dr. Ratnesh Srivastava, CSIT, GGV Bilaspur

July 19, 2025

Introduction

Data preprocessing is the critical first step in any data science pipeline. Real-world data is messy, incomplete, and inconsistent. Proper preprocessing ensures data quality and prepares datasets for machine learning algorithms. This lecture covers the mathematical intuition behind key preprocessing steps with practical examples, guiding you through the essential stages of transforming raw data into an understandable and usable format.

—

1 Why Preprocess Data?

Data preprocessing is fundamental to improving data efficiency and directly affects the outcomes of any analytic algorithm. It addresses common issues such as:

- **Missing values** cause computational errors and bias in models.
- **Outliers** distort statistical measures and negatively impact model performance.
- **Feature scaling** ensures algorithms converge properly and that no single feature dominates due to its magnitude.
- **Categorical variables** require numerical representation for machine learning algorithms to process them.
- **Non-linear relationships** need transformation for linear models to effectively capture patterns.

—

2 Steps in Data Preprocessing

Data preprocessing is generally carried out in a structured sequence of simple steps:

1. Gathering the data
2. Import the dataset & Libraries
3. Dealing with Missing Values
4. Divide the dataset into Dependent & Independent variable
5. Dealing with Categorical values
6. Split the dataset into training and test set
7. Feature Scaling

—

2.1 1. Gathering the Data

Data, in its raw form, is information representing human and machine observations of the world. The specific dataset you gather entirely depends on the type of problem you aim to solve in machine learning, as each problem often demands a unique data approach.

Popular Sources for Datasets:

- **Kaggle:** A widely used platform for data science competitions and a vast repository of datasets. (`https://www.kaggle.com/datasets`)
- **UCI Machine Learning Repository:** One of the oldest and most established sources for machine learning datasets on the web. (`http://mlr.cs.umass.edu/ml/`)
- **Awesome Public Datasets (GitHub):** A curated GitHub repository featuring a collection of high-quality public datasets. (`https://github.com/awesomedata/awesome-public-datasets`)
- **Government Open Data Portals:** Many governments provide open access to their data for public use.
  - Indian Government: `http://data.gov.in`
  - US Government: `https://www.data.gov/`
  - British Government: `https://data.gov.uk/`
  - France Government: `https://www.data.gouv.fr/en/`

—

2.2 2. Importing the Dataset & Libraries

The initial step in any programming-based data preprocessing workflow involves importing the necessary libraries and then loading your dataset. Libraries are collections of pre-written modules and functions that streamline coding tasks.

Importing Libraries (Python Example): Python's `import` keyword is used to bring external libraries into your program. Common libraries for data preprocessing include Pandas for data manipulation and Scikit-learn for machine learning utilities.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder, LabelEncoder
from sklearn.compose import ColumnTransformer
```

Importing the Dataset (Python Example): Data is commonly stored in formats like CSV (Comma Separated Values) format. The Pandas library provides the `read_csv()` method, among others, to load various file types.

```
# Loading data from a CSV file
dataset = pd.read_csv('your_dataset_name.csv')

# Pandas can read many other file formats:
# pd.read_excel('file.xlsx') # For Excel files
# pd.read_json('file.json')  # For JSON files
# pd.read_sql('SELECT * FROM your_table', connection) # For SQL databases
```

—

3 Mathematical Foundations & Practical Implementation

3.1 3. Dealing with Missing Values

Problem: Missing data (often represented as `NaN` - Not a Number) can cause algorithms to fail or produce biased results. If not addressed, missing values can prevent many machine learning models from processing datasets.

Checking for Null Values:

It's essential to first identify the presence and extent of missing values in your dataset. Pandas provides convenient methods for this.

```
# Get a concise summary of the DataFrame, including non-null counts and data types
print(dataset.info())
# This shows total entries and non-null counts for each feature.

# See a boolean mask of null values (True for null, False for not null)
print(dataset.isna())

# Get the count of null values corresponding to each feature
print(dataset.isna().sum())
# Example output:
# Age         2
# Salary      1
# Country     0
# Purchased   0
# (This indicates 'Age' and 'Salary' have missing values)
```

Strategies for Handling Missing Values:

- **Dropping Rows/Columns:** The simplest approach is to remove rows or columns that contain missing data using `dropna()`.

  ```
  # Drop all rows with any missing values
  dataset_cleaned_rows = dataset.dropna()
  # print(dataset_cleaned_rows)
  # Caution: This is not always a good idea, especially for small datasets,
  # as removing entire rows can lead to a significant loss of valuable information.
  ```

- **Replacing Null Values with Strategy (Imputation):** This involves filling in missing values with a calculated approximation (e.g., mean, median, or mode) of the existing data in that feature. This approach helps to retain more data points compared to dropping rows.
  - **Mean Imputation:** For numerical features that are approximately normally distributed.

    $$\text{Imputed Value} = \bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

    ```
    # Example: Impute missing 'Age' values with the mean of the 'Age' column
    dataset['Age'].fillna(dataset['Age'].mean(), inplace=True)
    # 'inplace=True' modifies the DataFrame directly.
    ```

– **Median Imputation:** Preferred for numerical features that are skewed or contain outliers, as the median is less affected by extreme values than the mean.

$$\text{Imputed Value} = \text{Median}(X)$$

```
# Example: Impute missing 'Salary' values with the median of the 'Salary' column
dataset['Salary'].fillna(dataset['Salary'].median(), inplace=True)
```

– **Mode Imputation:** Best for categorical or discrete numerical features.

```
# Replace missing categorical values with the mode (most frequent value)
# dataset['Categorical_Feature'].fillna(dataset['Categorical_Feature'].mode()[0], inplace=True)
```

– **Using Scikit-learn's SimpleImputer:** A more robust way to handle imputation, especially within a preprocessing pipeline.

```
from sklearn.impute import SimpleImputer
# For numerical columns, use 'mean' or 'median' strategy
imputer_numeric = SimpleImputer(missing_values=np.nan, strategy='mean')
% imputer_numeric = SimpleImputer(missing_values=np.nan, strategy='median')
% dataset[['Age', 'Salary']] = imputer_numeric.fit_transform(dataset[['Age', 'Salary']])

% For categorical columns, use 'most_frequent' strategy
% imputer_categorical = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
% dataset[['Categorical_Feature']] = imputer_categorical.fit_transform(dataset[['Categorical_Fe
```

—

3.2 4. Divide the Dataset into Dependent & Independent Variable

After importing and initial handling of missing values, it's crucial to separate your dataset into features (independent variables, usually denoted as X) and the target (dependent variable, usually denoted as Y). This is a standard practice for supervised machine learning problems.

Example Dataset Structure: Consider a dataset for a shopping complex tracking customer purchases:

Country	Age	Salary	Purchased
France	44	72000	No
Spain	27	48000	Yes
Germany	30	54000	No
...	...	...	...

Here, 'Country', 'Age', and 'Salary' are the **independent variables** (features), typically denoted as X. 'Purchased' is the **dependent variable** (target) that we aim to predict, typically denoted as Y.

Separating Variables (Python Example using `iloc`): The Pandas `iloc` indexer is used for integer-location based indexing, allowing you to select rows and columns by their integer positions.

```
# X will contain all rows and all columns EXCEPT the last one (features)
X = dataset.iloc[:, :-1].values

# Y will contain all rows and ONLY the last column (target)
Y = dataset.iloc[:, -1].values

# Note: ':' selects all, using '[]' helps you select multiple columns or rows.
# ':-1' selects all columns from the beginning up to (but not including) the last column.
# '-1' selects only the last column. '.values' converts to a NumPy array.
```

—

3.3 5. Encoding Categorical Variables

Categorical variables (e.g., 'Country', 'Gender', 'Purchase History' as 'Low'/'Medium'/'High') represent qualitative data. Since machine learning models are built upon mathematical equations and calculations, they cannot directly process text-based categories. Therefore, these categorical values must be encoded into numerical representations.

Label Encoding:

Label Encoding assigns a unique integer to each category. It is suitable for **ordinal categorical data**, where there is a clear, inherent order or ranking among the categories (e.g., 'Low' < 'Medium' < 'High').

$$\{\text{Low}, \text{Medium}, \text{High}\} \rightarrow \{0, 1, 2\}$$

Problem with Label Encoding for Nominal Data: If Label Encoding is applied to **nominal categorical data** (where no intrinsic order exists, like 'France', 'Spain', 'Germany'), the machine learning model might incorrectly infer an ordinal relationship. For instance, if 'France' is encoded as 0, 'Germany' as 1, and 'Spain' as 2, the model might mistakenly assume that 'Spain' has a "higher value" than 'Germany' or 'France', which is not true for nominal categories.

Label Encoding (Python Example):

```
from sklearn.preprocessing import LabelEncoder

# Create a LabelEncoder object
le = LabelEncoder()

# Example for a dependent variable (Y) like 'Purchased' (Yes/No)
# Assuming Y is a NumPy array of 'Yes'/'No' strings
Y = le.fit_transform(Y)
% After transformation, Y might be [0, 1, 0, ...] where 0 could be 'No' and 1 'Yes' (or vice-versa)

% Example for an independent variable (X) column (e.g., 'Purchase History' if ordinal)
% X[:, index_of_column] = le.fit_transform(X[:, index_of_column])
```

One-Hot Encoding (Dummy Variables):

One-Hot Encoding is the preferred method for **nominal categorical data** where there is no inherent order. It transforms each categorical value into a new binary (0 or 1) column. This prevents the model from assuming any artificial ordinal relationship.

Concept of Dummy Variables: Instead of a single column for the categorical feature, One-Hot Encoding creates N new columns if there are N unique categories. A '1' in a new column indicates the presence of that specific category, and '0' indicates its absence.

Original	Red	Green	Blue
Red	1	0	0
Green	0	1	0
Blue	0	0	1

One-Hot Encoding (Python Example): Scikit-learn's `OneHotEncoder` is used, often in conjunction with `ColumnTransformer` to apply transformations to specific columns within your feature matrix X.

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
import numpy as np % Ensure numpy is imported

% Create a ColumnTransformer.
% 'onehot_country' is a name for this transformer.
```

```
% OneHotEncoder() is the transformer to apply.
% [0] is the index of the column(s) to apply it to (e.g., 'Country' is at index 0 in X).
% 'remainder='passthrough'' ensures that other columns not specified are kept as they are.
ct = ColumnTransformer(
    transformers=[('onehot_country', OneHotEncoder(), [0])],
    remainder='passthrough'
)

% Apply the transformation to X. fit_transform learns the categories and then transforms.
X = ct.fit_transform(X)

% The output of ColumnTransformer is a NumPy array.
% You can check its type: print(type(X))
% To visualize (optional, for understanding):
% import pandas as pd
% X_df = pd.DataFrame(X)
% print(X_df.head())
% The first few columns will now represent the one-hot encoded 'Country'.
% Example: X might start with columns like [1.0, 0.0, 0.0, 44, 72000] for France.
```

Why Drop the First Column (Dummy Variable Trap): When performing One-Hot Encoding, it is a common practice to drop one of the newly created dummy variables (e.g., if you encode 'Country', you might drop 'Country_France'). This is done to avoid the "dummy variable trap," which leads to perfect multicollinearity. Multicollinearity occurs when two or more independent variables in a regression model are highly correlated, making it difficult for the model to distinguish their individual effects. If you have N categories, you only need $N-1$ dummy variables to uniquely represent all categories. The omitted category serves as the reference category, and its presence is implied when all other dummy variables for that feature are 0.

—

3.4 6. Split the Dataset into Training and Test Set

Before training any machine learning model, it is crucial to split your dataset into two distinct subsets: a training set and a test set. This separation is vital for evaluating your model's ability to generalize to new, unseen data.

- **Training Set ($X_{\text{train}}, Y_{\text{train}}$):** This subset of the data is used to train the machine learning model. The model learns patterns, relationships, and parameters from this data.
- **Test Set ($X_{\text{test}}, Y_{\text{test}}$):** This subset is reserved and not used during the training phase. It is used to evaluate the performance of the trained model on data it has never seen before. This provides an unbiased estimate of how well the model will perform in real-world scenarios.

Common Split Ratios: Typical split ratios for training and testing data include 70:30, 80:20, or 75:25 (training percentage : testing percentage). The choice often depends on the size of your dataset (larger datasets can afford smaller test sets) and the specific problem requirements. An 80:20 split is a frequently used default.

Splitting the Dataset (Python Example): The `train_test_split` function from Scikit-learn's `model_selection` module is the standard tool for this task. It takes your features (X) and target (Y) and returns four arrays: `X_train`, `X_test`, `Y_train`, and `Y_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,      % Allocate 20% of the data for the test set
```

```
    random_state = 42     % Set a random state for reproducibility of the split
)
% X_train: Training features
% X_test: Test features
% Y_train: Training target values (corresponding to X_train)
% Y_test: Test target values (corresponding to X_test)
```

—

3.5 7. Feature Scaling

Why Scaling: Most machine learning algorithms, especially those that rely on distance calculations (e.g., K-Nearest Neighbors, Support Vector Machines, K-Means Clustering, PCA, Neural Networks, Regression models), are sensitive to the magnitude and range of features. If features have vastly different scales, the feature with a larger range might disproportionately influence the model's performance, leading to biased results or slow convergence.

Feature scaling standardizes or normalizes independent features so they fall within a fixed range or follow a specific distribution.

Types of Feature Scaling:

- **Standardization (Z-score Normalization):** This technique transforms data such that it has a mean (μ) of 0 and a standard deviation (σ) of 1. It is suitable for features that follow a Gaussian (normal) distribution or when algorithms assume a zero mean and unit variance. Standardization does not bound values to a specific range, so outliers can still exist.

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

Standardization (Python Example):

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

% IMPORTANT: Fit the scaler ONLY on the training data (X_train) to avoid data leakage.
% This learns the mean and standard deviation from the training data.
X_train_scaled = scaler.fit_transform(X_train)

% Transform both the training and test data using the *fitted* scaler.
% The test data is transformed using the mean/std learned from the training data.
X_test_scaled = scaler.transform(X_test)
```

- **Normalization (Min-Max Scaling):** This method scales features to a fixed range, typically between 0 and 1. It is useful when you need features to be within a specific bounded range, for instance, for algorithms that expect input features in a specific range (e.g., some neural network activation functions).

$$X_{\text{scaled}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Normalization (Python Example):

```
from sklearn.preprocessing import MinMaxScaler

minmax_scaler = MinMaxScaler()
```

```
X_train_minmax = minmax_scaler.fit_transform(X_train)
X_test_minmax = minmax_scaler.transform(X_test)
```

- **Robust Scaling:** This method scales features using the median and Interquartile Range (IQR). It is particularly useful when your data contains many outliers, as the median and IQR are less sensitive to extreme values compared to the mean and standard deviation.

$$X_{\text{scaled}} = \frac{X - \text{Median}}{\text{IQR}}$$

Robust Scaling (Python Example):

```
from sklearn.preprocessing import RobustScaler

robust_scaler = RobustScaler()
X_train_robust = robust_scaler.fit_transform(X_train)
X_test_robust = robust_scaler.transform(X_test)
```

Should you apply Feature Scaling to the Dependent Variable (Y)?

- For **classification problems** where Y is a categorical variable (e.g., 0 or 1), scaling the dependent variable is generally **not required**. The target variable represents distinct classes, not magnitudes that need standardization.
- For **regression problems** where Y is a continuous variable, scaling the dependent variable **might be beneficial** for certain models (e.g., Neural Networks) or for interpreting coefficients. However, it's less universally required than scaling features. If the dependent variable is scaled, remember to inverse transform the model's predictions to get results back in the original scale for meaningful interpretation.

—

4 Mathematical Foundations (Advanced Topics)

4.1 Outlier Detection & Treatment

Outliers are data points that significantly deviate from other observations. They can skew statistical measures (like mean and standard deviation) and negatively impact model training by distorting the learned relationships.

- **Z-Score Method:** This method is suitable for distributions that are approximately Gaussian (normal). A data point is considered an outlier if its Z-score (which measures how many standard deviations away from the mean a data point is) exceeds a certain threshold (commonly ± 2 or ± 3).

$$Z = \frac{X - \mu}{\sigma}, \quad |Z| > 3 \text{ is often considered an outlier}$$

- **IQR Method (Interquartile Range):** This method is robust for skewed data or non-Gaussian distributions. Outliers are typically defined as values falling below a calculated lower bound or above a calculated upper bound.

$$\text{Lower Bound} = Q1 - 1.5 \times IQR, \quad \text{Upper Bound} = Q3 + 1.5 \times IQR$$

Treatment Options for Outliers:

- **Removal:** Delete the outlier data points from the dataset. This should be used with caution, especially for small datasets, as it can lead to loss of valuable information.
- **Capping (Winsorization):** Replace outlier values with a specified percentile value. For example, values above the 99th percentile might be replaced with the value at the 99th percentile. This preserves the data point but reduces its extreme impact on the model.

—

4.2 Feature Transformation

Feature transformation is used to change the distribution or scale of a feature. This is often done to meet the assumptions of certain machine learning models (e.g., linearity, normality), improve model performance, or reduce skewness.

- **Log Transform:** Commonly applied to right-skewed data to make its distribution more symmetric, often closer to a normal distribution.

$$X_{\text{new}} = \log(X + \epsilon)$$

  Where ϵ is a small constant (e.g., 1) to handle zero or negative values if they exist, as $\log(0)$ is undefined.

- **Box-Cox Transform:** This is a more generalized power transformation that can handle both positive and negative values. It finds the optimal λ (lambda) value to transform the data to be as close to a normal distribution as possible.

$$X_{\text{new}} = \begin{cases} \frac{X^\lambda - 1}{\lambda} & \lambda \neq 0 \\ \log(X) & \lambda = 0 \end{cases}$$

- **Polynomial Features:** Creates new features by raising existing features to a certain power or by creating interaction terms between features. This allows linear models to capture non-linear relationships.

$$\text{New Features} = \{X, X^2, X^3, \ldots, X_1X_2, X_1^2X_2, \ldots\}$$

—

5 Comprehensive Example

We'll use a customer dataset to demonstrate all preprocessing steps.

5.1 Original Dataset

	Age	Income ($)	Gender	Purchase History	Days Since Last Purchase	Purchased
	25	50,000	Male	High	10	Yes
	30	NaN	Female	Medium	60	No
1lightyellow	22	200,000	Male	Low	5	Yes
	35	75,000	Female	High	NaN	No
	45	30,000	Other	Medium	365	No
	28	1,000,000	Male	High	2	Yes

5.2 Step 1: Handle Missing Values

- **Income:** Median = 75,000 → Impute NaN with 75,000
- **Days Since Last Purchase:** Mean = 88.4 → Impute NaN with 88.4

5.3 Step 2: Outlier Treatment (IQR Method)

- **Income:** Q1 = 50,000, Q3 = 200,000, IQR = 150,000
  Upper Bound = 200,000 + 1.5 × 150,000 = 425,000
  Cap 1,000,000 → 425,000
- **Days Since Last Purchase:** Q1 = 5, Q3 = 60, IQR = 55
  Upper Bound = 60 + 1.5 × 55 = 142.5
  Cap 365 → 142.5

5.4 Step 3: Feature Transformation

- **Box-Cox for Income:** Optimal $\lambda = 0.25$

$$\text{Income}_{\text{trans}} = \frac{\text{Income}^{0.25} - 1}{0.25}$$

- **Polynomial for Age:** Create Age^2 feature

5.5 Step 4: Encoding Categorical Variables

- **Gender:** One-Hot Encoding (drop first)
- **Purchase History:** Ordinal Encoding
  {Low:0, Medium:1, High:2}

5.6 Step 5: Feature Scaling (Standardization)

$$X_{\text{scaled}} = \frac{X - \mu}{\sigma}$$

5.7 Final Preprocessed Dataset

1lightgreen

Age (sc)	Age^2 (sc)	Income (BC, sc)	Gen_F	Gen_O	PH	Days (sc)	Purchased
-0.73	0.12	-0.45	0	0	2	-0.78	Yes
-0.10	0.34	-0.20	1	0	1	0.16	No
-1.10	-0.55	0.75	0	0	0	-0.88	Yes
0.52	1.02	-0.20	1	0	2	0.70	No
1.76	2.89	-0.90	0	1	1	1.72	No
-0.35	-0.22	1.00	0	0	2	-0.93	Yes

sc = scaled, BC = Box-Cox, Gen_F = Gender_Female, Gen_O = Gender_Other, PH = Purchase History

—

Key Insights

- **Missing Values:** Always visualize missing patterns before imputation.
- **Outliers:** IQR method is robust for skewed distributions.
- **Scaling:** Essential for distance-based algorithms (KNN, SVM).
- **Encoding:** One-hot for nominal, label for ordinal data.
- **Transformations:** Box-Cox for optimal normality, polynomials for non-linearity.

—

Questions & Answers

1. **Q: Why use median instead of mean for missing value imputation?**
   A: Median is robust to outliers. If income data has extreme values (e.g., $1,000,000), the mean would be skewed, but median remains representative.

2. **Q: When should we use Box-Cox instead of log transform?**
   A: Box-Cox generalizes log transform by finding optimal λ. Use it when log transform doesn't fully normalize the distribution.

3. **Q: Why drop the first column in one-hot encoding?**
   A: To avoid the "dummy variable trap" (perfect multicollinearity). The dropped feature becomes the reference category.

4. **Q: How does polynomial features help linear models?**
   A: Linear models assume linear relationships. Polynomial terms (x^2, x^3) allow them to capture non-linear patterns.

5. **Q: Why cap outliers instead of removing them?**
   A: Capping preserves data points while reducing extreme value impact. Removal risks losing valuable information, especially in small datasets.

6. **Q: Should we scale before or after transformation?**
   A: Always transform first! Transformations change distributions, so scaling should be the final step before modeling.

7. **Q: When to use robust scaling vs standardization?**
   A: Use robust scaling when outliers are present. It uses median and IQR instead of mean and standard deviation.

8. **Q: How to handle new categories during one-hot encoding?**
   A: During deployment, new categories should be mapped to zeros in all columns or treated as a separate "unknown" category.

—

Best Practices

1. Always create a preprocessing pipeline for reproducibility
2. Visualize distributions before and after transformations
3. Split data before preprocessing to avoid data leakage
4. Document all preprocessing decisions
5. Validate preprocessing impact on model performance