Data Preprocessing in Machine Learning

A Comprehensive Guide

Dr. ABDELLAOUI

2024/2025

Outline

Why Data Preprocessing Matters

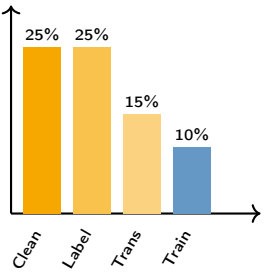
The Reality of ML Projects

- Real-world data is **messy** and **incomplete**
- Raw data rarely meets algorithm requirements
- **"Garbage In, Garbage Out"**
- Quality preprocessing = better performance

Key Insight

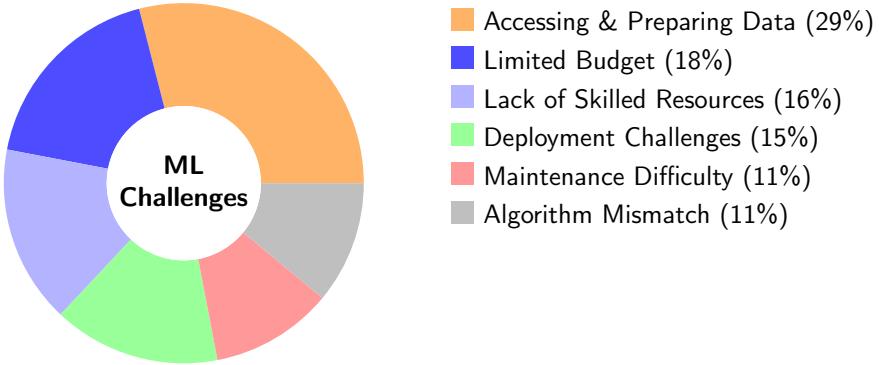
Data preprocessing is the **most time-consuming** but **most impactful** phase!

Time Allocation in ML Projects



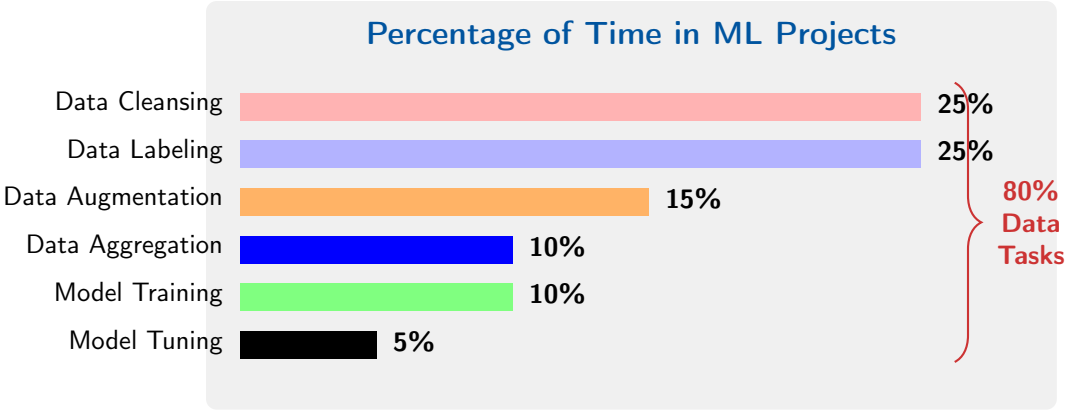
80% on data tasks!

Challenges in Applying Machine Learning



Key Finding: Data-related challenges represent the **#1 obstacle** in ML adoption.

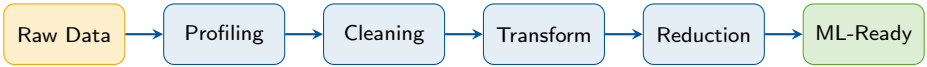
Time Allocation in ML Projects



Industry Statistics

According to multiple surveys, data scientists spend **60-80%** of their time on data preparation

The Data Preprocessing Pipeline



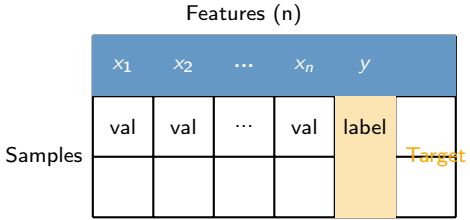
Preprocessing Objectives

1. **Data Quality:** Ensure accuracy and completeness
2. **Data Compatibility:** Match algorithm requirements
3. **Data Efficiency:** Reduce dimensionality
4. **Data Enhancement:** Create meaningful features

What is a Dataset?

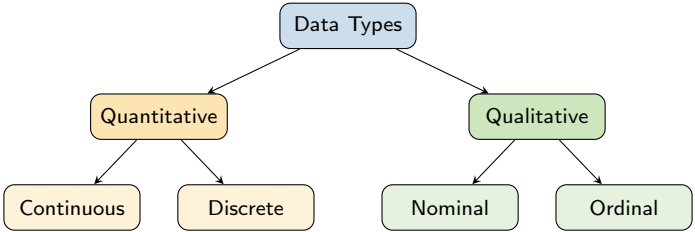
Definition

A **dataset** is an organized collection of data points described by features, formatted for analysis.



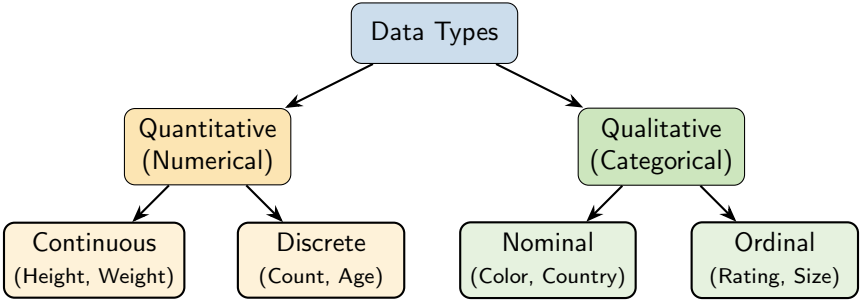
Notation: $\mathcal{D} = \{(\mathrm{x}_i, y_i)\}_{i=1}^{m}$ where $\mathrm{x}_i \in \mathbb{R}^n$

Types of Data



Type	Equality	Order	Add	Ratio
Nominal	✓			
Ordinal	✓	✓		
Interval	✓	✓	✓	
Ratio	✓	✓	✓	✓

Types of Data



Data Type	Equality	Order	Addition	Multiplication
Nominal	✓	×	×	×
Ordinal	✓	✓	×	×
Interval	✓	✓	✓	×
Ratio	✓	✓	✓	✓

Common Data Quality Issues

Issue	Description	Example
Missing Values	NULL, NaN, empty	Age = NULL
Duplicates	Repeated records	Same row twice
Outliers	Extreme values	Age = 999
Noise	Random errors	Typos in data
Inconsistency	Format variations	NY vs New York
Errors	Wrong annotations	Mislabeled data

Real-World Example

A customer database: “NY”, “New York”, “new york”, “NYC” — all the same city!

Handling Missing Values

Detection

- Check for NULL, NaN
- `df.isnull().sum()`
- Visualize with heatmaps

Treatment Strategies

1. **Deletion** - Remove rows/columns
2. **Imputation**
   - Mean/Median/Mode
   - KNN Imputation
   - MICE (Multiple Imputation)
3. **Indicator** - Binary flag

Imputation Methods

Method	Best For
Mean	Numerical, normal dist
Median	Numerical, skewed
Mode	Categorical
KNN	Preserves relations
MICE	Uncertainty modeling

Outlier Detection

Statistical Methods

- **Z-score:** $|z| > 3$ is outlier

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

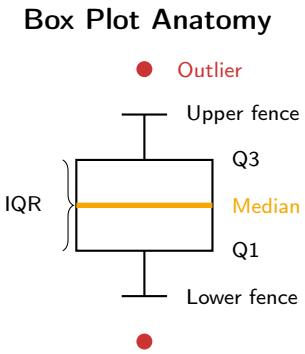
- **IQR Method:**

$$x < Q_1 - 1.5 \cdot IQR$$

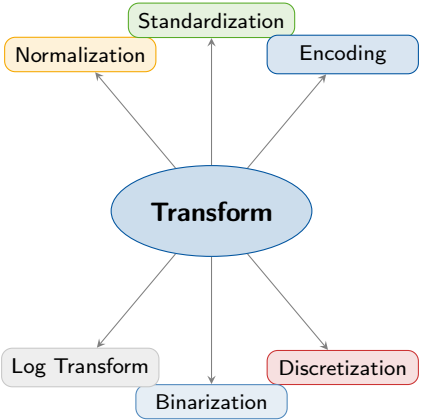
$$x > Q_3 + 1.5 \cdot IQR$$

Treatment Options

Remove, Cap (Winsorize), Transform, Keep



Transformation Overview



Normalization: Min-Max Scaling

Formula

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

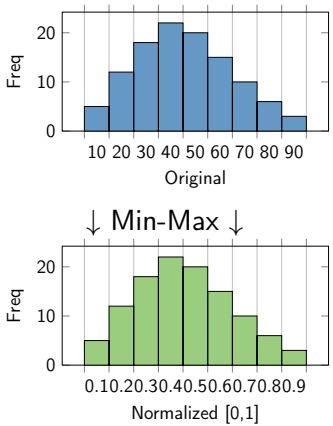
Scales to $[0, 1]$

When to Use

- Bounded data (age 0-100)
- Uniform distribution
- Neural networks, KNN

Limitation

Sensitive to outliers



Standardization: Z-Score

Formula

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

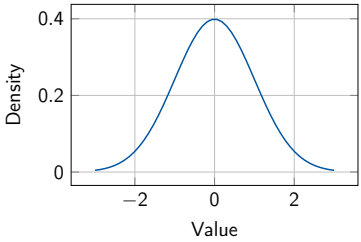
μ=mean, σ=std dev

When to Use

- Data with outliers
- Unknown bounds
- Gaussian assumptions
- SVM, PCA, Gradient descent

Result

Mean=0, Std=1



Standard Normal Distribution

$\mu = 0, \sigma = 1$

Normalization vs. Standardization

Aspect	Normalization	Standardization
Range	$[0, 1]$	Unbounded
Outliers	Sensitive	More robust
Distribution	Preserves	Centers at 0
Use Case	Bounded, NN	Unbounded, SVM
Formula	$\frac{x - x_{min}}{x_{max} - x_{min}}$	$\frac{x - \mu}{\sigma}$

Decision Rule

- **Normalization:** bounded data, no outliers
- **Standardization:** outliers present, Gaussian assumption

Encoding Categorical Variables

One-Hot Encoding

Binary columns per category

Color
Red
Blue
Green

→

R	B	G
1	0	0
0	1	0
0	0	1

Use for **nominal** data

Label Encoding

Integer per category

Size
Small
Medium
Large

→

Size
0
1
2

Use for **ordinal** data

Other Methods

Target Encoding, Frequency Encoding, Binary Encoding

Discretization (Binning)

Definition

Converting continuous → categorical by grouping into bins

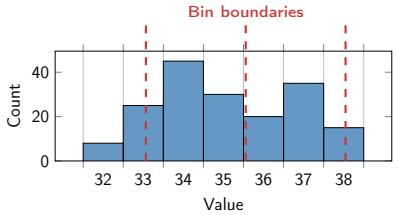
Equal-Width

$$width = \frac{x_{max} - x_{min}}{k}$$

All bins same range

Equal-Frequency (Quantile)

All bins same count



Why Reduce Dimensions?

Curse of Dimensionality

- Data sparse in high dimensions
- Distance metrics lose meaning
- More features = more data needed
- Higher computational cost
- Overfitting risk

Benefits

- Faster training
- Better generalization
- Easier visualization
- Noise reduction

Methods

Feature Selection:

- Filter (correlation, chi-square)
- Wrapper (RFE)
- Embedded (Lasso, trees)

Feature Extraction:

- PCA
- t-SNE
- Autoencoders

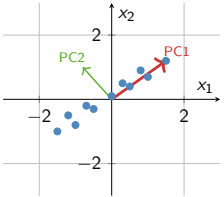
Principal Component Analysis (PCA)

Concept

Find orthogonal directions maximizing variance

Steps

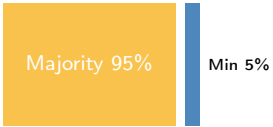
1. Standardize data
2. Compute covariance matrix
3. Get eigenvectors/values
4. Sort by eigenvalue
5. Select top k components
6. Project data



Handling Imbalanced Data

The Problem

Skewed class distribution $\rightarrow$ model favors majority



Degree	Minority
Mild	20-40%
Moderate	1-20%
Extreme	$<1\%$

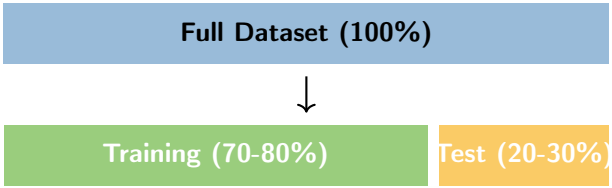
Solutions

- **Undersampling:** Reduce majority
- **Oversampling:** Increase minority (SMOTE)
- **Class weights:** Penalize majority errors more

Train-Test Split

Fundamental Rule

Never train and evaluate on the same data!



Key Points

- Test set must be representative
- Use stratified split for imbalanced data
- Temporal split for time series

Cross-Validation

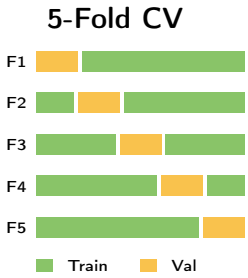
K-Fold CV

1. Split into k folds
2. For each fold: use as validation, rest for training
3. Average k scores

Benefits

- Reliable estimate
- Uses all data
- Essential for small datasets

Common: k=5 or k=10



Python Libraries

Library	Purpose	Key Functions
Pandas	Data manipulation	read_csv, fillna, drop
NumPy	Numerical ops	array, mean, std
Scikit-learn	ML preprocessing	StandardScaler, OneHotEncoder
SciPy	Statistics	zscore, IQR
imbalanced-learn	Resampling	SMOTE, RandomUnderSampler

GUI Tools

Weka, Orange3, RapidMiner, KNIME

Code Example

```python
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline

# Numeric preprocessing
numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# Categorical preprocessing
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('encoder', OneHotEncoder(handle_unknown='ignore'))
])

# Combine
preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_cols),
    ('cat', categorical_transformer, categorical_cols)
])

# Apply
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)  # NO fit!
```

Best Practices

Do's

- Explore data first
- Document transformations
- Use pipelines
- Fit on training only
- Validate assumptions
- Version your data

Don'ts

- Don't leak test data
- Don't over-engineer
- Don't ignore domain
- Don't skip visualization
- Don't forget scaling

Summary

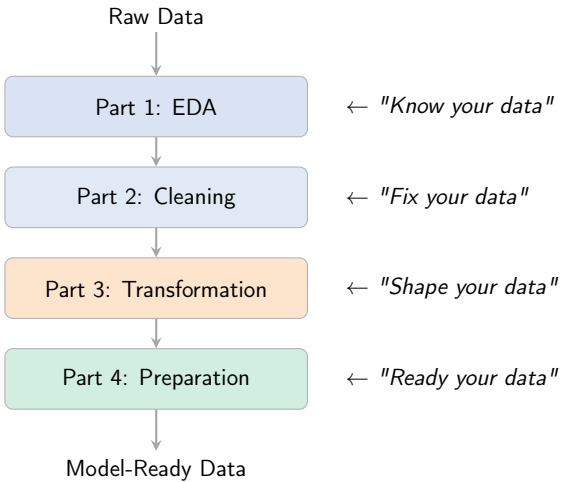
Key Takeaways

1. Data preprocessing is **critical** for ML success
2. **80%** of project time on data tasks
3. Quality data $>$ Complex algorithms
4. Main steps: **Clean, Transform, Reduce, Split**
5. Always **fit on training only**
6. Use **pipelines** for reproducibility

Questions?

Thank you for your attention!

Pre-Processing Pipeline Overview



Part 1: EDA

"Know your data"

Outline

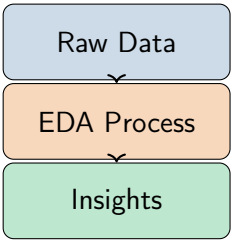
What is Exploratory Data Analysis?

Definition

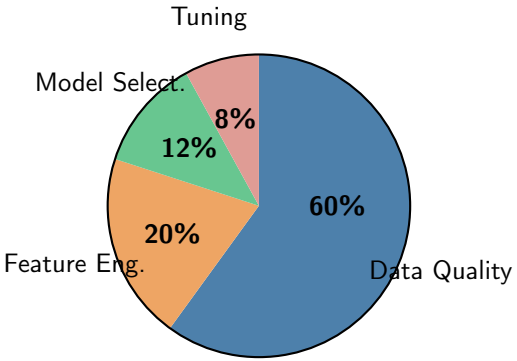
Exploratory Data Analysis (EDA) is an approach to analyzing datasets to summarize their main characteristics, often using statistical graphics and other data visualization methods.

Key Objectives:

- Understand data structure
- Detect patterns and anomalies
- Test hypotheses
- Check assumptions
- Guide feature engineering



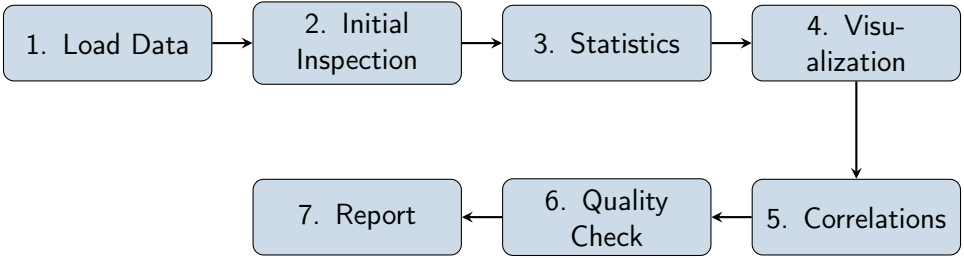
Why is EDA Critical in Machine Learning?



Golden Rule

"Garbage In, Garbage Out" — The quality of your model depends entirely on the quality of your data understanding.

The EDA Workflow



Understanding Data Types

Numerical Data

Continuous:

- Temperature: $T \in \mathbb{R}$
- Price: $P \in \mathbb{R}^+$
- Height: $h \in [0, \infty)$

Discrete:

- Count: $n \in \mathbb{N}$
- Age (years): $a \in \mathbb{Z}^+$

Categorical Data

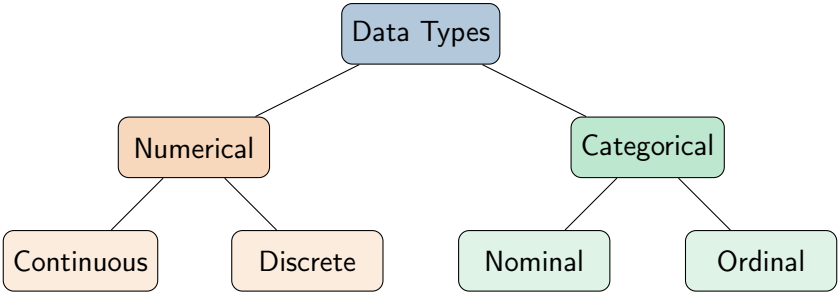
Nominal:

- Color: {Red, Blue, Green}
- Gender: {M, F, Other}

Ordinal:

- Education: {High School $<$ Bachelor $<$ Master $<$ PhD}
- Rating: $\{1 < 2 < 3 < 4 < 5\}$

Data Type Hierarchy



Why It Matters

Different data types require different:

- Statistical measures (mean vs. mode)
- Visualization techniques (histogram vs. bar chart)
- Preprocessing methods (scaling vs. encoding)

Data Structures in ML

The Feature Matrix

A dataset can be represented as a matrix $X \in \mathbb{R}^{n \times p}$:

$$X = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1p} \\ x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{np} \end{pmatrix}$$

Where:

- n = number of samples (rows)
- p = number of features (columns)
- x_{ij} = value of feature j for sample i

Initial Data Inspection Checklist

Questions to Answer:

1. What is the shape? $(n, p) = ?$
2. What are the column names?
3. What are the data types?
4. How much memory is used?
5. Are there missing values?
6. What do the first rows look like?

Python Commands

```
df.shape
df.columns
df.dtypes
df.memory_usage()
df.isnull().sum()
df.head()
```

Measures of Central Tendency

Mean (Arithmetic Average)

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Median

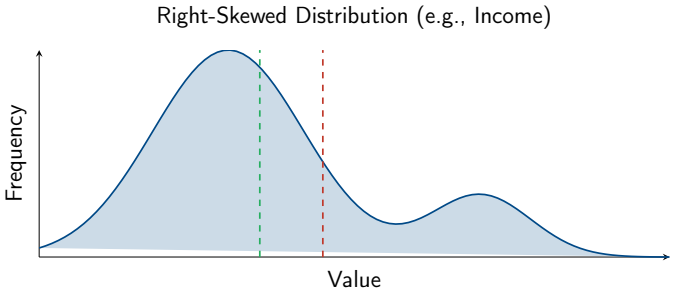
The middle value when data is sorted:

$$\tilde{x} = \begin{cases} x_{\frac{n+1}{2}} & \text{if } n \text{ is odd} \\ \frac{x_{\frac{n}{2}} + x_{\frac{n}{2}+1}}{2} & \text{if } n \text{ is even} \end{cases}$$

Mode

The most frequently occurring value: $\text{Mode} = \arg\max_x f(x)$

Mean vs. Median: When to Use Which?



Key Insight

Median is robust to outliers! For skewed data or data with outliers, median provides a better representation of the "typical" value.

Measures of Dispersion

Variance

Population variance:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Sample variance (unbiased estimator):

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

Standard Deviation

$$\sigma = \sqrt{\sigma^2} \quad \text{or} \quad s = \sqrt{s^2}$$

Standard deviation is in the same units as the original data.

Range and Interquartile Range (IQR)

Range

$$R = x_{\max} - x_{\min}$$

Sensitive to outliers

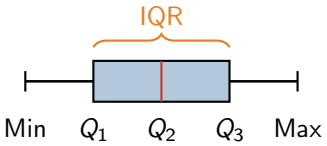
Interquartile Range

$$\text{IQR} = Q_3 - Q_1$$

Where:

- Q_1 = 25th percentile
- Q_3 = 75th percentile

Robust to outliers



Coefficient of Variation (CV)

Definition

The coefficient of variation measures relative variability:

$$CV = \frac{\sigma}{\mu} \times 100\%$$

or for samples:

$$CV = \frac{s}{\bar{x}} \times 100\%$$

Use Case

Comparing variability across different scales:

- Feature A: $\bar{x}_A = 1000$, $s_A = 50 \Rightarrow CV_A = 5\%$
- Feature B: $\bar{x}_B = 10$, $s_B = 2 \Rightarrow CV_B = 20\%$

Feature B has higher *relative* variability despite lower absolute s.

Skewness and Kurtosis

Skewness

Measures asymmetry of distribution:

$$\gamma_1 = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s}\right)^3$$

- $\gamma_1 = 0$: Symmetric
- $\gamma_1 > 0$: Right-skewed
- $\gamma_1 < 0$: Left-skewed

Kurtosis

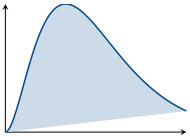
Measures "tailedness":

$$\gamma_2 = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{s}\right)^4 - 3$$

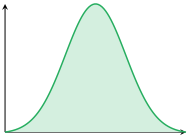
- $\gamma_2 = 0$: Mesokurtic (normal)
- $\gamma_2 > 0$: Leptokurtic (heavy tails)
- $\gamma_2 < 0$: Platykurtic (light tails)

Visualizing Skewness

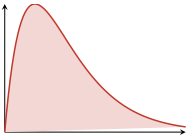
Left-Skewed ($\gamma_1 < 0$)



Symmetric ($\gamma_1 = 0$)



Right-Skewed ($\gamma_1 > 0$)



Summary Statistics Table

Measure	Formula	Robust?
Mean	$\bar{x} = \frac{1}{n}\sum x_i$	No
Median	Middle value	Yes
Mode	Most frequent	Yes
Variance	$s^2 = \frac{1}{n-1}\sum(x_i - \bar{x})^2$	No
Std Dev	$s = \sqrt{s^2}$	No
Range	$R = x_{max} - x_{min}$	No
IQR	$Q_3 - Q_1$	Yes

Tip

Always report both mean/std AND median/IQR when exploring data!

Why Visualize Data?

Anscombe's Quartet

Four datasets with **identical** summary statistics:

- Same mean: $\bar{x} = 9$, $\bar{y} = 7.5$
- Same variance: $s_x^2 = 11$, $s_y^2 = 4.12$
- Same correlation: $r = 0.816$
- Same regression line: $y = 3 + 0.5x$

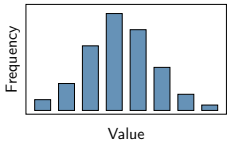
Yet they look completely different when plotted!

Lesson

Never trust statistics alone — always visualize your data!

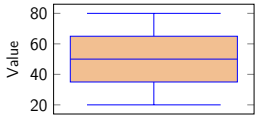
Univariate Visualizations

Histogram



Shows distribution shape, bins continuous data

Box Plot



Shows quartiles, median, outliers

Density Plots and KDE

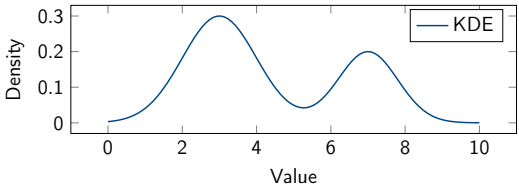
Kernel Density Estimation (KDE)

Smooth estimate of probability density function:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

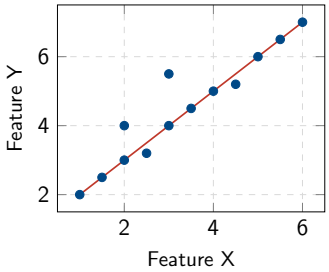
Where:

- $K(\cdot)$ is the kernel function (often Gaussian)
- h is the bandwidth (smoothing parameter)



Bivariate Visualizations

Scatter Plot



Reveals relationships between two variables

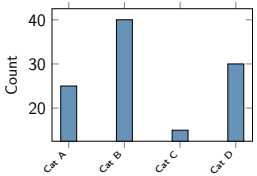
Heatmap (Correlation Matrix)

	A	B	C	D
B	0.7	1.0	0.5	-0.3
C	0.3	0.5	1.0	0.2
D	-0.5	-0.3	0.2	1.0

Shows pairwise correlations

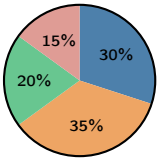
Categorical Data Visualizations

Bar Chart



Compare counts across categories

Pie Chart

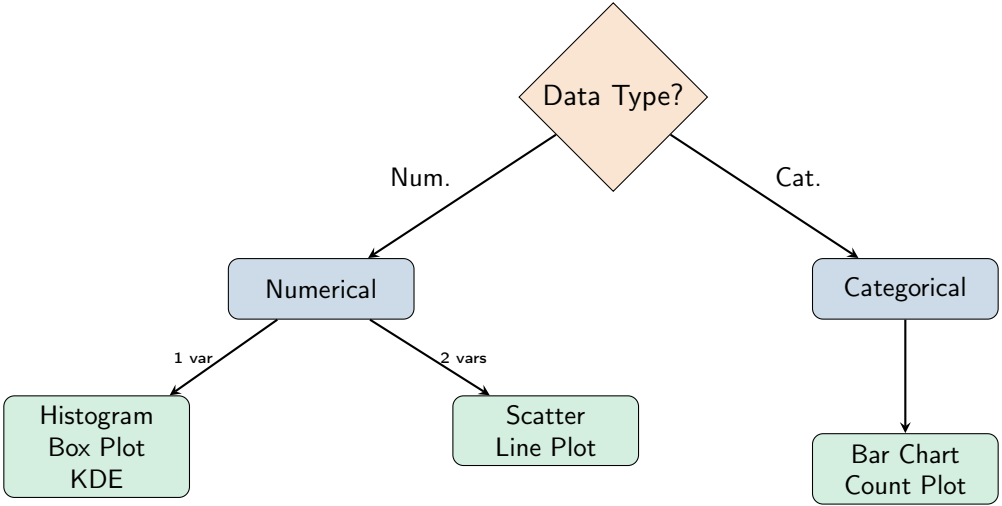


Show proportions (use sparingly!)

Best Practice

Prefer bar charts over pie charts — humans compare lengths better than angles!

Visualization Selection Guide



Pearson Correlation Coefficient

Definition

Measures **linear** relationship between two continuous variables:

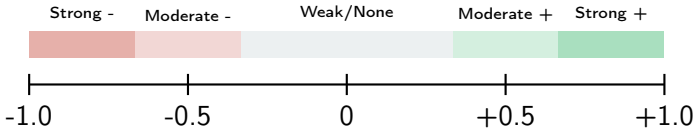
$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2} \cdot \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

Or equivalently:

$$r_{xy} = \frac{\text{Cov}(X, Y)}{\sigma_X \cdot \sigma_Y} = \frac{\mathbb{E}[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

- $r = +1$: Perfect positive linear correlation
- $r = -1$: Perfect negative linear correlation
- $r = 0$: No linear correlation (but may have non-linear relationship!)

Interpreting Correlation Values

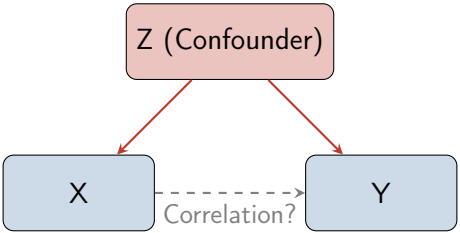


$\lvert r \rvert$	**Interpretation**
0.0 – 0.3	Negligible/Weak
0.3 – 0.5	Moderate
0.5 – 0.7	Strong
0.7 – 1.0	Very Strong

Correlation $\neq$ Causation!

Critical Warning

Correlation does not imply causation!



Example: Ice cream sales and drowning deaths are correlated.
Confounder: Hot weather increases both!

Spearman Rank Correlation

Definition

Measures **monotonic** relationship (not necessarily linear):

$$\rho = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)}$$

Where $d_i = \text{rank}(x_i) - \text{rank}(y_i)$

When to use Spearman:

- Non-linear monotonic relationships
- Ordinal data
- Data with outliers
- Non-normal distributions

When to use Pearson:

- Linear relationships
- Continuous data
- Normally distributed data
- No significant outliers

Correlation for Categorical Variables

Chi-Square Test of Independence

For categorical $\times$ categorical variables:

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

Where:

- O_{ij} = Observed frequency
- $E_{ij} = \frac{(\text{Row Total}_i) \times (\text{Column Total}_j)}{n}$ = Expected frequency

Cramér's V

Strength of association (normalized chi-square):

$$V = \sqrt{\frac{\chi^2}{}}$$

Point-Biserial Correlation

For Binary × Continuous Variables

Special case of Pearson correlation:

$$r_{pb} = \frac{\bar{Y}_1 - \bar{Y}_0}{s_Y} \cdot \sqrt{\frac{n_1 n_0}{n^2}}$$

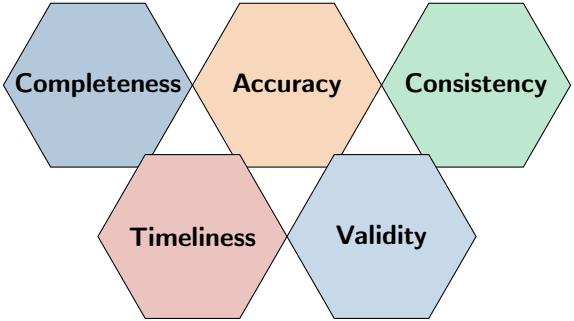
Where:

- $\bar{Y}_1$, $\bar{Y}_0$ = means of Y for each binary group
- n_1, n_0 = sample sizes for each group
- s_Y = standard deviation of Y

Example

Correlation between *Gender* (binary) and *Salary* (continuous)

Dimensions of Data Quality



- **Completeness:** Are there missing values?
- **Accuracy:** Are values correct?
- **Consistency:** Are values uniform across the dataset?
- **Timeliness:** Is the data up-to-date?
- **Validity:** Do values conform to expected formats/ranges?

Missing Data Analysis

Types of Missing Data

- **MCAR** (Missing Completely At Random): Missingness unrelated to any data
- **MAR** (Missing At Random): Missingness depends on observed data
- **MNAR** (Missing Not At Random): Missingness depends on unobserved data

Missing Rate Calculation

$$\text{Missing Rate}_j = \frac{\text{Number of missing values in column } j}{n} \times 100\%$$

Rule of Thumb

- $< 5\%$ missing: Generally safe to impute
- $5 - 20\%$ missing: Careful analysis required
- $> 20\%$ missing: Consider dropping or advanced imputation

Detecting Outliers

IQR Method

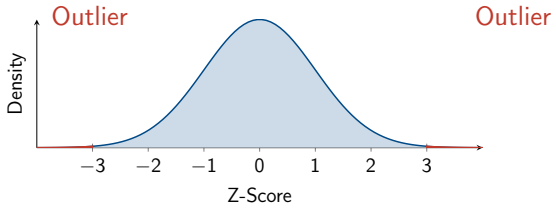
Outlier if: $x < Q_1 - 1.5 \cdot \text{IQR}$

or $x > Q_3 + 1.5 \cdot \text{IQR}$

Z-Score Method

$$z_i = \frac{x_i - \bar{x}}{s}$$

Outlier if: $|z_i| > 3$



Data Quality Checklist

Structural Checks:

- ☐ Correct number of rows/columns
- ☐ Appropriate data types
- ☐ No duplicate records
- ☐ Consistent column names
- ☐ Proper encoding (UTF-8)

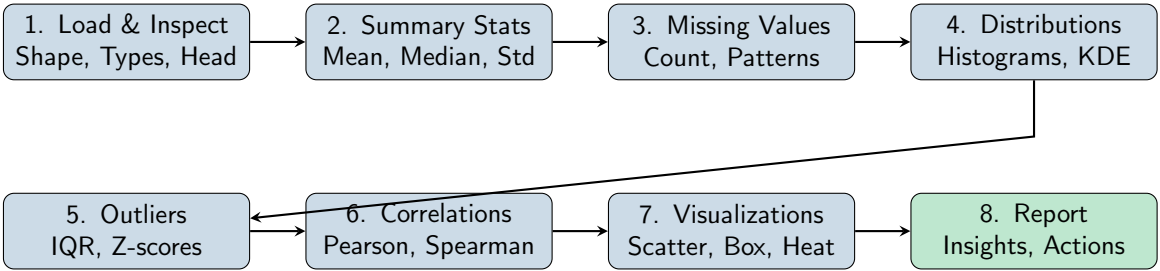
Content Checks:

- ☐ Values in valid ranges
- ☐ Categorical values expected
- ☐ No impossible combinations
- ☐ Dates in valid format
- ☐ Outliers identified

Quality Score

$$\text{Quality Score} = \frac{\text{Valid Records}}{\text{Total Records}} \times 100\%$$

Complete EDA Workflow



EDA Report Template

What to Include in Your EDA Report

1. **Dataset Overview**
   - Source, size, features description
2. **Data Quality Summary**
   - Missing values, duplicates, data types
3. **Univariate Analysis**
   - Key statistics and distributions for each feature
4. **Bivariate Analysis**
   - Important correlations and relationships
5. **Key Findings**
   - Patterns, anomalies, insights discovered
6. **Recommendations**
   - Suggested preprocessing steps

Python Code: Basic EDA Template

Essential EDA Commands

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load and inspect
df = pd.read_csv('data.csv')
print(df.shape, df.dtypes)
print(df.describe())

# Missing values
print(df.isnull().sum())

# Distributions
df.hist(figsize=(12,8))
```

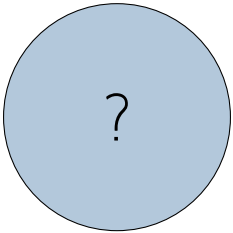
Key Takeaways

1. **Always start with EDA** — never skip this step!
2. **Know your data types** — they determine your analysis approach
3. **Use both statistics AND visualizations** — they complement each other
4. **Correlation $\neq$ Causation** — always remember this
5. **Document your findings** — create reproducible reports
6. **Quality matters** — assess completeness, accuracy, consistency

Next Part

Part 2: Data Cleaning — Handling missing values, duplicates, outliers, and noise.

Questions & Discussion



Contact: professor@university.edu

References

- Tukey, J.W. (1977). *Exploratory Data Analysis*. Addison-Wesley.
- Wickham, H. & Grolemund, G. (2017). *R for Data Science*. O'Reilly.
- McKinney, W. (2017). *Python for Data Analysis*. O'Reilly.
- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly.

References

Books

- Géron, A. (2022). *Hands-On Machine Learning*
- Zheng & Casari (2018). *Feature Engineering for ML*

Online

- `https://scikit-learn.org/stable/modules/preprocessing.html`
- `https://www.kaggle.com/learn`

Part I

Practice for Part 1 : EDA

What We Will Cover

Practical EDA Skills

- Setting up the Python environment
- Loading and inspecting data
- Understanding data types and structure
- Computing descriptive statistics
- Visualizing distributions and relationships
- Correlation analysis
- Identifying data quality issues

Libraries Used

`pandas`, `numpy`, `matplotlib`, `seaborn`, `scipy`

Essential Libraries Import

```python
# Data manipulation
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Statistical functions
from scipy import stats

# Display settings
pd.set_option('display.max_columns', None)
pd.set_option('display.precision', 2)
plt.style.use('seaborn-v0_8-whitegrid')
sns.set_palette("husl")

# Ignore warnings
import warnings
warnings.filterwarnings('ignore')
```

Loading Data

```python
# From CSV file
df = pd.read_csv('data.csv')

# From Excel file
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# From URL
url = 'https://example.com/data.csv'
df = pd.read_csv(url)

# Common parameters
df = pd.read_csv('data.csv',
                 sep=',',           # Delimiter
                 header=0,          # Row for column names
                 index_col=None,    # Column to use as index
                 na_values=['NA', 'missing', '?'],
                 parse_dates=['date_column'],
                 encoding='utf-8')
```

First Look at Data

```python
# Display first/last rows
df.head(10)         # First 10 rows
df.tail(5)          # Last 5 rows

# Random sample
df.sample(5)        # 5 random rows

# Dataset dimensions
print(f"Shape: {df.shape}")
print(f"Rows: {df.shape[0]}, Columns: {df.shape[1]}")

# Column names
print(df.columns.tolist())

# Quick overview
df.info()
```

Understanding Data Types

```python
# Check data types
print(df.dtypes)

# Count by data type
print(df.dtypes.value_counts())

# Identify column types
numerical_cols = df.select_dtypes(
    include=['int64', 'float64']).columns.tolist()
categorical_cols = df.select_dtypes(
    include=['object', 'category']).columns.tolist()
datetime_cols = df.select_dtypes(
    include=['datetime64']).columns.tolist()

print(f"Numerical: {numerical_cols}")
print(f"Categorical: {categorical_cols}")
print(f"Datetime: {datetime_cols}")
```

Memory Usage

```python
# Memory usage summary
print(df.info(memory_usage='deep'))

# Memory per column
memory_usage = df.memory_usage(deep=True)
print(memory_usage)

# Total memory in MB
total_mb = df.memory_usage(deep=True).sum() / 1024**2
print(f"Total Memory: {total_mb:.2f} MB")

# Optimize memory (convert to category for low cardinality)
for col in categorical_cols:
    if df[col].nunique() / len(df) < 0.5:
        df[col] = df[col].astype('category')
```

Basic Statistics for Numerical Data

```python
# Complete statistical summary
df.describe()

# Include all columns (categorical too)
df.describe(include='all')

# Specific percentiles
df.describe(percentiles=[.01, .05, .25, .5, .75, .95, .99])

# Individual statistics
print(f"Mean:\n{df[numerical_cols].mean()}")
print(f"Median:\n{df[numerical_cols].median()}")
print(f"Std Dev:\n{df[numerical_cols].std()}")
print(f"Min:\n{df[numerical_cols].min()}")
print(f"Max:\n{df[numerical_cols].max()}")
```

Advanced Statistics

```python
# Skewness and Kurtosis
from scipy.stats import skew, kurtosis

for col in numerical_cols:
    sk = skew(df[col].dropna())
    kt = kurtosis(df[col].dropna())
    print(f"{col}: Skewness={sk:.3f}, Kurtosis={kt:.3f}")

# Quantiles
print(df[numerical_cols].quantile([0.25, 0.5, 0.75]))

# Range and IQR
for col in numerical_cols:
    q1, q3 = df[col].quantile([0.25, 0.75])
    iqr = q3 - q1
    range_val = df[col].max() - df[col].min()
    print(f"{col}: Range={range_val:.2f}, IQR={iqr:.2f}")
```

Statistics for Categorical Data

```python
# Value counts for each categorical column
for col in categorical_cols:
    print(f"\n{col}:")
    print(df[col].value_counts())
    print(f"Unique values: {df[col].nunique()}")

# Frequency and percentage
for col in categorical_cols:
    freq = df[col].value_counts()
    pct = df[col].value_counts(normalize=True) * 100
    summary = pd.DataFrame({'Count': freq, 'Percent': pct})
    print(f"\n{col}:")
    print(summary)

# Mode (most frequent value)
print(df[categorical_cols].mode().iloc[0])
```

Creating a Summary Report

```python
def eda_summary(df):
    """Generate comprehensive EDA summary"""
    summary = pd.DataFrame({
        'dtype': df.dtypes,
        'non_null': df.count(),
        'null_count': df.isnull().sum(),
        'null_pct': (df.isnull().sum() / len(df) * 100).round(2),
        'unique': df.nunique(),
        'unique_pct': (df.nunique() / len(df) * 100).round(2)
    })

    # Add statistics for numeric columns
    num_stats = df.describe().T[['mean', 'std', 'min', 'max']]
    summary = summary.join(num_stats)

    return summary

print(eda_summary(df))
```

Histogram - Distribution of Single Variable

```python
# Single histogram
plt.figure(figsize=(10, 6))
plt.hist(df['column'], bins=30, edgecolor='black', alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Distribution of Column')
plt.show()

# Multiple histograms with seaborn
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
for idx, col in enumerate(numerical_cols[:6]):
    ax = axes[idx // 3, idx % 3]
    sns.histplot(df[col], kde=True, ax=ax)
    ax.set_title(f'Distribution of {col}')
plt.tight_layout()
plt.show()
```

Box Plot - Identifying Outliers

```python
# Single box plot
plt.figure(figsize=(8, 6))
sns.boxplot(x=df['column'])
plt.title('Box Plot of Column')
plt.show()

# Multiple box plots
plt.figure(figsize=(14, 6))
df[numerical_cols].boxplot()
plt.xticks(rotation=45)
plt.title('Box Plots of All Numerical Variables')
plt.tight_layout()
plt.show()

# Box plot by category
plt.figure(figsize=(10, 6))
sns.boxplot(x='category_col', y='numeric_col', data=df)
plt.title('Numeric by Category')
plt.show()
```

Violin Plot - Distribution Shape

```python
# Violin plot shows distribution shape
plt.figure(figsize=(12, 6))
sns.violinplot(x='category', y='value', data=df)
plt.title('Distribution by Category')
plt.show()

# Combined violin and box plot
plt.figure(figsize=(12, 6))
sns.violinplot(x='category', y='value', data=df, inner='box')
plt.show()

# Split violin plot for binary comparison
plt.figure(figsize=(12, 6))
sns.violinplot(x='category', y='value', hue='binary_col',
               data=df, split=True)
plt.show()
```

Bar Plot - Categorical Data

```
# Count plot for categorical variables
plt.figure(figsize=(10, 6))
sns.countplot(x='category_col', data=df,
              order=df['category_col'].value_counts().index)
plt.xticks(rotation=45)
plt.title('Count by Category')
plt.show()

# Bar plot with mean values
plt.figure(figsize=(10, 6))
sns.barplot(x='category', y='value', data=df,
            estimator=np.mean, ci=95)
plt.title('Mean Value by Category (with 95% CI)')
plt.show()

# Horizontal bar plot
plt.figure(figsize=(8, 10))
df['category'].value_counts().plot(kind='barh')
plt.xlabel('Count')
plt.show()
```

Scatter Plot - Relationships

```python
# Basic scatter plot
plt.figure(figsize=(10, 6))
plt.scatter(df['x'], df['y'], alpha=0.5)
plt.xlabel('X Variable')
plt.ylabel('Y Variable')
plt.title('Scatter Plot: X vs Y')
plt.show()

# Scatter with regression line
plt.figure(figsize=(10, 6))
sns.regplot(x='x', y='y', data=df, scatter_kws={'alpha':0.5})
plt.title('Scatter Plot with Regression Line')
plt.show()

# Scatter colored by category
plt.figure(figsize=(10, 6))
sns.scatterplot(x='x', y='y', hue='category', data=df)
plt.title('Scatter Plot by Category')
plt.show()
```

Pair Plot - Multiple Relationships

```python
# Basic pair plot (all numerical variables)
sns.pairplot(df[numerical_cols[:5]])  # Limit to 5 columns
plt.suptitle('Pair Plot of Numerical Variables', y=1.02)
plt.show()

# Pair plot with hue (colored by category)
sns.pairplot(df, vars=numerical_cols[:4], hue='target')
plt.show()

# Pair plot with different plot types
sns.pairplot(df[numerical_cols[:4]],
             diag_kind='kde',      # KDE on diagonal
             plot_kws={'alpha': 0.5},
             corner=True)          # Only lower triangle
plt.show()
```

Heatmap - Visualizing Matrices

```python
# Correlation heatmap
plt.figure(figsize=(12, 10))
correlation_matrix = df[numerical_cols].corr()
sns.heatmap(correlation_matrix,
            annot=True,           # Show values
            fmt='.2f',            # Format
            cmap='RdBu_r',        # Color map
            center=0,             # Center at 0
            square=True,
            linewidths=0.5)
plt.title('Correlation Heatmap')
plt.tight_layout()
plt.show()

# Missing values heatmap
plt.figure(figsize=(12, 6))
sns.heatmap(df.isnull(), cbar=True, yticklabels=False)
plt.title('Missing Values Pattern')
plt.show()
```

Distribution Comparison (KDE)

```python
# KDE plot
plt.figure(figsize=(10, 6))
sns.kdeplot(df['column'], fill=True, alpha=0.5)
plt.title('Kernel Density Estimation')
plt.show()

# Compare distributions by group
plt.figure(figsize=(10, 6))
for category in df['group'].unique():
    subset = df[df['group'] == category]['value']
    sns.kdeplot(subset, label=category, fill=True, alpha=0.3)
plt.legend()
plt.title('Distribution by Group')
plt.show()

# Compare with normal distribution
from scipy.stats import norm
sns.kdeplot(df['column'], label='Data')
x = np.linspace(df['column'].min(), df['column'].max(), 100)
plt.plot(x, norm.pdf(x, df['column'].mean(), df['column'].std()),
         label='Normal', linestyle='--')
plt.legend()
```

Pearson Correlation

```python
# Correlation matrix (Pearson - default)
corr_matrix = df[numerical_cols].corr(method='pearson')
print(corr_matrix)

# Correlation with specific target
target_corr = df[numerical_cols].corr()['target'].sort_values(
    ascending=False)
print("Correlation with target:")
print(target_corr)

# Statistical significance
from scipy.stats import pearsonr
for col in numerical_cols:
    if col != 'target':
        corr, p_value = pearsonr(df[col].dropna(),
                                 df['target'].dropna())
        sig = "***" if p_value < 0.001 else \
              "**" if p_value < 0.01 else \
              "*" if p_value < 0.05 else ""
        print(f"{col}: r={corr:.3f}, p={p_value:.4f} {sig}")
```

Spearman and Kendall Correlation

```python
# Spearman correlation (rank-based, non-linear monotonic)
spearman_corr = df[numerical_cols].corr(method='spearman')

# Kendall correlation (also rank-based)
kendall_corr = df[numerical_cols].corr(method='kendall')

# Compare all three methods
from scipy.stats import spearmanr, kendalltau

col1, col2 = 'var1', 'var2'

pearson_r, pearson_p = pearsonr(df[col1], df[col2])
spearman_r, spearman_p = spearmanr(df[col1], df[col2])
kendall_r, kendall_p = kendalltau(df[col1], df[col2])

print(f"Pearson:  r={pearson_r:.3f}, p={pearson_p:.4f}")
print(f"Spearman: r={spearman_r:.3f}, p={spearman_p:.4f}")
print(f"Kendall:  r={kendall_r:.3f}, p={kendall_p:.4f}")
```

Correlation for Categorical Variables

```python
# Chi-Square test for categorical vs categorical
from scipy.stats import chi2_contingency

def cramers_v(x, y):
    """Calculate Cramer's V for categorical association"""
    contingency = pd.crosstab(x, y)
    chi2, p, dof, expected = chi2_contingency(contingency)
    n = contingency.sum().sum()
    min_dim = min(contingency.shape) - 1
    return np.sqrt(chi2 / (n * min_dim)), p

# Example usage
v, p = cramers_v(df['cat1'], df['cat2'])
print(f"Cramer's V: {v:.3f}, p-value: {p:.4f}")

# Contingency table
contingency = pd.crosstab(df['cat1'], df['cat2'],
                          margins=True, normalize='all')
print(contingency)
```

Point-Biserial Correlation

```python
# Point-biserial: numerical vs binary categorical
from scipy.stats import pointbiserialr

# Convert binary category to 0/1
df['binary_encoded'] = (df['binary_col'] == 'Yes').astype(int)

# Calculate correlation
for col in numerical_cols:
    corr, p_value = pointbiserialr(df['binary_encoded'], df[col])
    print(f"{col}: r={corr:.3f}, p={p_value:.4f}")

# ANOVA for numerical vs multi-class categorical
from scipy.stats import f_oneway

groups = [group['numeric_col'].values
          for name, group in df.groupby('category_col')]
f_stat, p_value = f_oneway(*groups)
print(f"ANOVA: F={f_stat:.3f}, p={p_value:.4f}")
```

Correlation Visualization

```python
# Clustered heatmap (hierarchical clustering)
sns.clustermap(df[numerical_cols].corr(),
               annot=True, fmt='.2f', cmap='RdBu_r',
               figsize=(12, 12))
plt.show()

# Top correlations (excluding self-correlation)
def get_top_correlations(corr_matrix, n=10):
    pairs = []
    for i in range(len(corr_matrix.columns)):
        for j in range(i+1, len(corr_matrix.columns)):
            pairs.append((corr_matrix.columns[i],
                          corr_matrix.columns[j],
                          corr_matrix.iloc[i, j]))
    return sorted(pairs, key=lambda x: abs(x[2]), reverse=True)[:n]

top_corr = get_top_correlations(corr_matrix, n=10)
for col1, col2, corr in top_corr:
    print(f"{col1} <-> {col2}: {corr:.3f}")
```

Missing Values Analysis

```python
# Count missing values
print("Missing Values:")
print(df.isnull().sum())

# Percentage of missing values
missing_pct = (df.isnull().sum() / len(df) * 100).round(2)
missing_df = pd.DataFrame({
    'Missing Count': df.isnull().sum(),
    'Missing %': missing_pct
}).sort_values('Missing %', ascending=False)
print(missing_df[missing_df['Missing Count'] > 0])

# Visualize missing values
import missingno as msno  # pip install missingno
msno.matrix(df)
plt.show()

msno.bar(df)
plt.show()
```

Missing Values Pattern

```python
# Missing values heatmap (correlation of missingness)
msno.heatmap(df)
plt.show()

# Dendrogram of missing values
msno.dendrogram(df)
plt.show()

# Check if missing values are related to other columns
def missing_analysis(df, col_with_missing, other_cols):
    """Compare statistics when value is missing vs not missing"""
    missing_mask = df[col_with_missing].isnull()

    for col in other_cols:
        if df[col].dtype in ['int64', 'float64']:
            mean_missing = df.loc[missing_mask, col].mean()
            mean_not_missing = df.loc[~missing_mask, col].mean()
            print(f"{col}: Missing={mean_missing:.2f}, "
                  f"Not Missing={mean_not_missing:.2f}")

missing_analysis(df, 'col_with_na', numerical_cols)
```

Duplicate Detection

```python
# Check for duplicate rows
print(f"Duplicate rows: {df.duplicated().sum()}")

# View duplicate rows
duplicates = df[df.duplicated(keep=False)]
print(duplicates.sort_values(by=df.columns.tolist()))

# Duplicates based on specific columns
key_cols = ['id', 'date', 'category']
print(f"Duplicates by key: {df.duplicated(subset=key_cols).sum()}")

# Identify which rows are duplicates
df['is_duplicate'] = df.duplicated(keep=False)
df['duplicate_group'] = df.groupby(list(df.columns[:-1])).ngroup()

# Count duplicates per group
dup_counts = df.groupby('duplicate_group').size()
print(dup_counts[dup_counts > 1])
```

Outlier Detection - IQR Method

```python
def detect_outliers_iqr(df, columns):
    """Detect outliers using IQR method"""
    outlier_info = {}

    for col in columns:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        outliers = df[(df[col] < lower_bound) |
                      (df[col] > upper_bound)]

        outlier_info[col] = {
            'count': len(outliers),
            'percentage': len(outliers) / len(df) * 100,
            'lower_bound': lower_bound,
            'upper_bound': upper_bound
        }

    return pd.DataFrame(outlier_info).T

print(detect_outliers_iqr(df, numerical_cols))
```

Outlier Detection - Z-Score Method

```python
from scipy.stats import zscore

def detect_outliers_zscore(df, columns, threshold=3):
    """Detect outliers using Z-score method"""
    outlier_info = {}

    for col in columns:
        z_scores = np.abs(zscore(df[col].dropna()))
        outlier_mask = z_scores > threshold

        outlier_info[col] = {
            'count': outlier_mask.sum(),
            'percentage': outlier_mask.sum() / len(df) * 100,
            'max_zscore': z_scores.max()
        }

    return pd.DataFrame(outlier_info).T

print(detect_outliers_zscore(df, numerical_cols))

# Mark outliers in dataframe
for col in numerical_cols:
    df[f'{col}_zscore'] = np.abs(zscore(df[col].fillna(df[col].mean())))
    df[f'{col}_is_outlier'] = df[f'{col}_zscore'] > 3
```

Outlier Visualization

```python
# Box plots for all numerical columns
fig, axes = plt.subplots(2, 3, figsize=(15, 10))
for idx, col in enumerate(numerical_cols[:6]):
    ax = axes[idx // 3, idx % 3]
    sns.boxplot(x=df[col], ax=ax)
    ax.set_title(f'{col}')
plt.tight_layout()
plt.show()

# Scatter plot highlighting outliers
plt.figure(figsize=(10, 6))
outlier_mask = np.abs(zscore(df['column'])) > 3
plt.scatter(df.index[~outlier_mask], df.loc[~outlier_mask, 'column'],
            alpha=0.5, label='Normal')
plt.scatter(df.index[outlier_mask], df.loc[outlier_mask, 'column'],
            color='red', alpha=0.7, label='Outlier')
plt.legend()
plt.title('Outlier Detection')
plt.show()
```

Pandas Profiling / YData Profiling

```python
# Install: pip install ydata-profiling

from ydata_profiling import ProfileReport

# Generate comprehensive report
profile = ProfileReport(df,
                        title="EDA Report",
                        explorative=True,
                        minimal=False)

# Save to HTML file
profile.to_file("eda_report.html")

# Display in Jupyter notebook
profile.to_notebook_iframe()

# Minimal report (faster for large datasets)
profile_minimal = ProfileReport(df, minimal=True)
profile_minimal.to_file("eda_report_minimal.html")
```

Sweetviz

```python
# Install: pip install sweetviz

import sweetviz as sv

# Analyze single dataset
report = sv.analyze(df)
report.show_html("sweetviz_report.html")

# Compare two datasets (e.g., train vs test)
report = sv.compare([df_train, "Train"], [df_test, "Test"])
report.show_html("comparison_report.html")

# Analyze with target variable
report = sv.analyze(df, target_feat='target_column')
report.show_html("sweetviz_target_report.html")

# Compare subsets
report = sv.compare_intra(df, df['category'] == 'A',
                          names=['Group A', 'Others'])
report.show_html("intra_comparison.html")
```

D-Tale (Interactive EDA)

```python
# Install: pip install dtale

import dtale

# Launch interactive dashboard
d = dtale.show(df)

# Opens in browser automatically
# Or get the URL
print(d._url)

# Features available in D-Tale:
# - Column analysis
# - Correlation matrix
# - Charts builder
# - Missing values analysis
# - Duplicate detection
# - Code export

# For Jupyter notebooks
d.notebook()
```

EDA Function Template (Part 1)

```python
def complete_eda(df, target=None):
    """Comprehensive EDA function"""
    print("="*50)
    print("DATASET OVERVIEW")
    print("="*50)
    print(f"Shape: {df.shape}")
    print(f"\nData Types:\n{df.dtypes.value_counts()}")
    print(f"\nMemory Usage: {df.memory_usage(deep=True).sum()/1024**2:.2f} MB")

    # Identify column types
    num_cols = df.select_dtypes(include=np.number).columns.tolist()
    cat_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()

    print(f"\nNumerical columns ({len(num_cols)}): {num_cols}")
    print(f"Categorical columns ({len(cat_cols)}): {cat_cols}")

    # Missing values
    print("\n" + "="*50)
    print("MISSING VALUES")
    print("="*50)
    missing = df.isnull().sum()
    missing_pct = (missing / len(df) * 100).round(2)
    print(pd.DataFrame({'Count': missing, '%': missing_pct}
          ).query('Count > 0').sort_values('%', ascending=False))
```

EDA Function Template (Part 2)

```
    # Continued from previous slide...

    # Descriptive statistics
    print("\n" + "="*50)
    print("DESCRIPTIVE STATISTICS")
    print("="*50)
    print(df[num_cols].describe().T)

    # Duplicates
    print(f"\nDuplicate rows: {df.duplicated().sum()}")

    # Target analysis (if provided)
    if target and target in df.columns:
        print("\n" + "="*50)
        print(f"TARGET ANALYSIS: {target}")
        print("="*50)
        if df[target].dtype in ['object', 'category']:
            print(df[target].value_counts())
            print(f"\nClass balance: {df[target].value_counts(normalize=True)}")
        else:
            print(df[target].describe())

    return num_cols, cat_cols
```

EDA Visualization Function

```python
def plot_eda(df, num_cols, cat_cols, target=None):
    """Generate EDA visualizations"""
    # 1. Distribution of numerical variables
    n_num = len(num_cols)
    fig, axes = plt.subplots((n_num+2)//3, 3, figsize=(15, 4*((n_num+2)//3)))
    axes = axes.flatten()
    for i, col in enumerate(num_cols):
        sns.histplot(df[col], kde=True, ax=axes[i])
        axes[i].set_title(f'{col}')
    plt.tight_layout()
    plt.savefig('numerical_distributions.png', dpi=150)
    plt.show()

    # 2. Correlation heatmap
    plt.figure(figsize=(12, 10))
    sns.heatmap(df[num_cols].corr(), annot=True, fmt='.2f', cmap='RdBu_r')
    plt.title('Correlation Matrix')
    plt.tight_layout()
    plt.savefig('correlation_matrix.png', dpi=150)
    plt.show()
```

Running Complete EDA

```python
# Load data
df = pd.read_csv('your_data.csv')

# Run EDA
num_cols, cat_cols = complete_eda(df, target='target_column')

# Generate plots
plot_eda(df, num_cols, cat_cols, target='target_column')

# Or use automated tools
from ydata_profiling import ProfileReport
profile = ProfileReport(df, title="Complete EDA Report")
profile.to_file("complete_eda_report.html")

# Save summary to Excel
with pd.ExcelWriter('eda_summary.xlsx') as writer:
    df.describe().to_excel(writer, sheet_name='Statistics')
    df.corr().to_excel(writer, sheet_name='Correlations')
    df.isnull().sum().to_excel(writer, sheet_name='Missing')
```

Summary: EDA Checklist

Data Understanding:

- ☐ `df.shape`, `df.info()`
- ☐ `df.head()`, `df.sample()`
- ☐ `df.dtypes`
- ☐ `df.describe()`

Data Quality:

- ☐ `df.isnull().sum()`
- ☐ `df.duplicated().sum()`
- ☐ Outlier detection (IQR/Z-score)

Visualizations:

- ☐ Histograms / KDE
- ☐ Box plots
- ☐ Scatter plots
- ☐ Correlation heatmap
- ☐ Pair plot

Correlations:

- ☐ Pearson (numerical)
- ☐ Spearman (non-linear)
- ☐ Chi-square (categorical)

Key Python Functions Reference

Task	Function
Load data	`pd.read_csv()`, `pd.read_excel()`
First look	`df.head()`, `df.info()`, `df.shape`
Statistics	`df.describe()`, `df.mean()`, `df.std()`
Missing values	`df.isnull().sum()`, `msno.matrix()`
Duplicates	`df.duplicated()`, `df.drop_duplicates()`
Value counts	`df['col'].value_counts()`
Correlation	`df.corr()`, `pearsonr()`, `spearmanr()`
Histogram	`sns.histplot()`, `plt.hist()`
Box plot	`sns.boxplot()`, `df.boxplot()`
Scatter	`sns.scatterplot()`, `plt.scatter()`
Heatmap	`sns.heatmap()`
Pair plot	`sns.pairplot()`
Auto EDA	`ProfileReport()`, `sv.analyze()`

Questions & Practice

Next: Part 2 - Data Cleaning with Python

Part II

Part 2: Data Cleaning

"Fix your data"

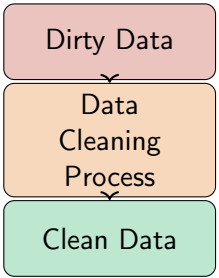
What is Data Cleaning?

Definition

Data Cleaning (or Data Cleansing) is the process of detecting and correcting (or removing) corrupt, inaccurate, or irrelevant records from a dataset.

Common Data Problems:

- Missing values
- Duplicate records
- Outliers and anomalies
- Inconsistent formatting
- Noisy data
- Invalid entries



Why is Data Cleaning Important?

60-80%
of ML project time

Poor data quality
costs $15M/year avg.

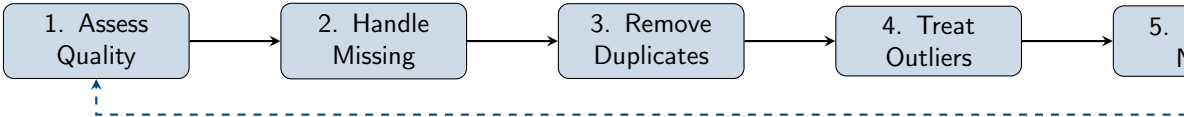
Clean data
improves accuracy 40%+

The Data Quality Principle

"Your model is only as good as your data."

- Missing values → Biased models
- Duplicates → Overfitting
- Outliers → Skewed predictions
- Noise → Poor generalization

Data Cleaning Workflow



Iterative Process

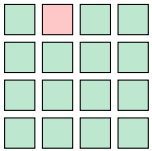
Data cleaning is often **iterative** — you may need to revisit earlier steps as you discover new issues.

Types of Missing Data

Classification by Rubin (1976)

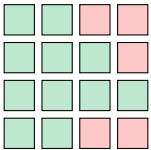
Missing data mechanisms determine appropriate handling strategies:

MCAR
Missing Completely At Random



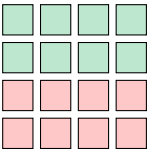
Random pattern

MAR
Missing At Random



Depends on observed

MNAR
Missing Not At Random



Depends on missing value

Mathematical Definition of Missingness

Formal Notation

Let X be the complete data matrix and M be the missingness indicator matrix:

$$M_{ij} = \begin{cases} 1 & \text{if } X_{ij} \text{ is missing} \\ 0 & \text{if } X_{ij} \text{ is observed} \end{cases}$$

Missing Data Mechanisms

- **MCAR:** $P(\mathrm{M}|\mathrm{X}) = P(\mathrm{M})$
- **MAR:** $P(\mathrm{M}|\mathrm{X}) = P(\mathrm{M}|\mathrm{X}_{obs})$
- **MNAR:** $P(\mathrm{M}|\mathrm{X}) = P(\mathrm{M}|\mathrm{X}_{obs}, \mathrm{X}_{miss})$

Key Insight

MCAR and MAR are *ignorable* — standard methods work. MNAR requires specialized techniques.

Detecting Missing Values

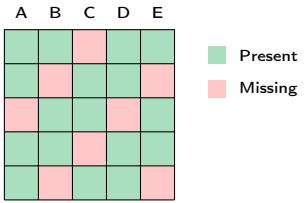
Missing Rate per Feature

$$r_j = \frac{\sum_{i=1}^{n} M_{ij}}{n} \times 100\%$$

Total Missing Rate

$$r_{total} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{p} M_{ij}}{n \times p} \times 100\%$$

Missing Value Heatmap



Strategy 1: Deletion Methods

Listwise Deletion (Complete Case)

Remove entire rows with any missing values:

$$X_{clean} = \{x_i : \sum_{j=1}^{p} M_{ij} = 0\}$$

Pros:

- Simple to implement
- Works if MCAR

Cons:

- Loses data (reduces n)
- Biased if not MCAR

Pairwise Deletion

Use available data for each analysis:

$\text{Corr}(X_j, X_k)$ uses all rows where both present

Pros:

- Preserves more data
- Flexible

Cons:

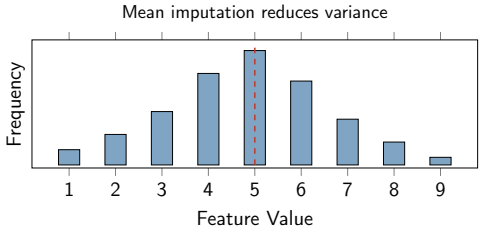
- Different n per calculation
- Can produce inconsistent results

Strategy 2: Simple Imputation Methods

Mean/Median/Mode Imputation

Replace missing values with central tendency measure:

$$\hat{x}_{ij} = \begin{cases} \bar{x}_j = \frac{1}{n_{obs}} \sum_{i:M_{ij}=0} x_{ij} & \text{(Mean)} \\ \tilde{x}_j & \text{(Median)} \\ \text{Mode}(x_j) & \text{(Mode, for categorical)} \end{cases}$$

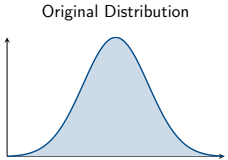


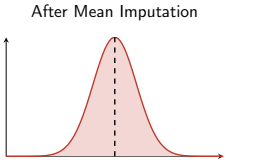
Problems with Simple Imputation

Variance Reduction Problem

Mean imputation artificially reduces variance:

$$\text{Var}(\hat{X}) = \text{Var}(X_{obs}) \cdot \frac{n_{obs}}{n} < \text{Var}(X_{true})$$





Strategy 3: K-Nearest Neighbors (KNN) Imputation

Algorithm

For each missing value x_{ij}:

1. Find k nearest neighbors based on other features
2. Impute using weighted average of neighbors' values

$$\hat{x}_{ij} = \frac{\sum_{l \in N_k(i)} w_{il} \cdot x_{lj}}{\sum_{l \in N_k(i)} w_{il}}$$

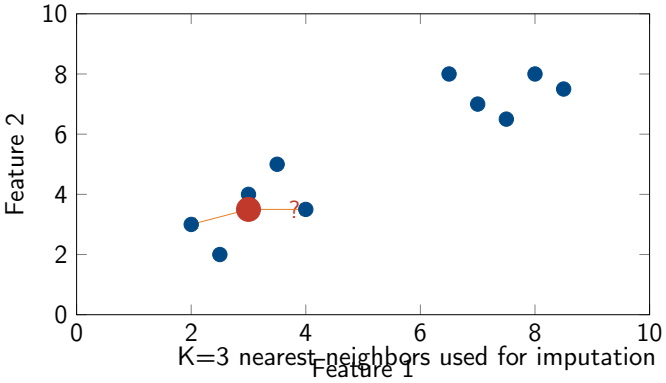
Where weights are typically inverse distance:

$$w_{il} = \frac{1}{d(x_i, x_l) + \epsilon}$$

Distance Metric (Euclidean)

$$d(x_i, x_l) = \sqrt{\sum_{u} (x_{iu} - x_{lu})^2}$$

KNN Imputation Visualization



Strategy 4: Regression Imputation

Linear Regression Imputation

Predict missing values using regression on observed features:

$$\hat{x}_{ij} = \beta_0 + \sum_{k \neq j} \beta_k x_{ik}$$

Where $\boldsymbol{\beta}$ is estimated from complete cases:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}_{obs}^T \mathbf{X}_{obs})^{-1} \mathbf{X}_{obs}^T \mathbf{y}_{obs}$$

Stochastic Regression Imputation

Add random error to preserve variance:

$$\hat{x}_{ij} = \beta_0 + \sum_{k \neq j} \beta_k x_{ik} + \epsilon_i$$

Multivariate Imputation by Chained Equations

Create m complete datasets, analyze each, combine results:

Rubin's Combining Rules

$$\bar{Q} = \frac{1}{m} \sum_{i=1}^{m} \hat{Q}_i \quad \text{(Combined estimate)}$$

Imputation Method Comparison

Method	Preserves Variance	Handles MNAR	Compu-tation	Best For
Deletion	N/A	✗	Low	MCAR, low %
Mean/Median	✗	✗	Low	Quick baseline
KNN	✓	✗	Medium	Local patterns
Regression	Partial	✗	Medium	Linear relations
MICE	✓	Partial	High	Best practice

Rule of Thumb

- $< 5\%$ missing: Mean/median usually fine
- $5 - 20\%$ missing: KNN or regression
- $> 20\%$ missing: MICE or consider dropping feature

Types of Duplicate Records

Exact Duplicates

All values identical across all columns:

$$x_i = x_j \Leftrightarrow \forall k : x_{ik} = x_{jk}$$

John	25	NYC
Jane	30	LA
John	25	NYC
Bob	22	CHI

Fuzzy/Near Duplicates

Similar but not identical (typos, variations):

John Smith	NYC
Jon Smith	New York
Jane Doe	LA

Different spellings, same entity!

Impact of Duplicates

- Inflate dataset size artificially
- Cause overfitting (same samples in train/test)

Detecting Exact Duplicates

Duplicate Detection Methods

1. Full Row Comparison:

$$\text{isDuplicate}(i, j) = \prod_{k=1}^{p} 1[x_{ik} = x_{jk}]$$

2. Key-Based Detection:

$$\text{isDuplicate}(i, j) = 1[\text{key}(x_i) = \text{key}(x_j)]$$

Where key might be (ID, timestamp) or composite columns.

Duplicate Rate

$$\text{Duplicate Rate} = \frac{n - n_{unique}}{n} \times 100\%$$

Fuzzy Matching: String Similarity Metrics

Levenshtein Distance (Edit Distance)

Minimum number of single-character edits (insert, delete, substitute):

$$\text{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0 \\ |b| & \text{if } |a| = 0 \\ \text{lev}(\text{tail}(a), \text{tail}(b)) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \text{lev}(\text{tail}(a), b) \\ \text{lev}(a, \text{tail}(b)) \\ \text{lev}(\text{tail}(a), \text{tail}(b)) \end{cases} & \text{otherwise} \end{cases}$$

Example

“kitten” $\rightarrow$ “sitting”: Distance = 3

1. kitten $\rightarrow$ sitten (substitute k$\rightarrow$s)

More Similarity Metrics

Jaro-Winkler Similarity

$$sim_j = \begin{cases} 0 & \text{if } m = 0 \\ \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right) & \text{otherwise} \end{cases}$$

Where m = matching characters, t = transpositions

$$sim_{jw} = sim_j + \ell \cdot p \cdot (1 - sim_j)$$

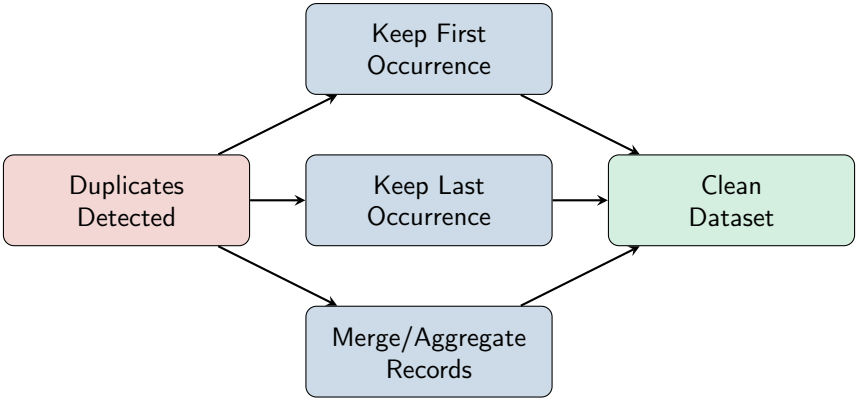
Where ℓ = common prefix length, p = scaling factor (usually 0.1)

Jaccard Similarity (for sets/tokens)

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Example: {John, Smith} vs {Jon, Smith}

Duplicate Handling Strategies



Aggregation Functions for Merging

For numeric columns: mean, median, sum, min, max

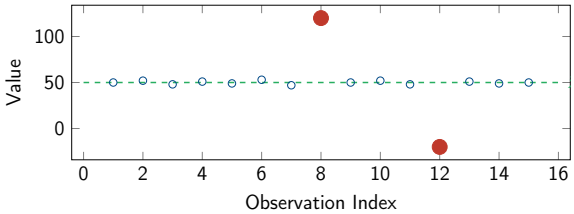
For categorical: mode, concatenate unique values

What are Outliers?

Definition

An **outlier** is a data point that differs significantly from other observations. It may indicate:

- Measurement/data entry errors
- Experimental errors
- Natural variation (rare but valid)
- Fraudulent behavior



Statistical Methods: Z-Score

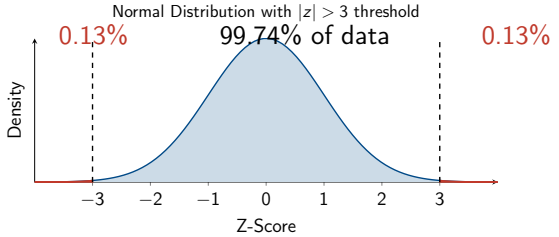
Z-Score Method

Measures how many standard deviations a point is from the mean:

$$z_i = \frac{x_i - \bar{x}}{s}$$

Outlier criterion:

$$|z_i| > \tau \quad \text{(typically } \tau = 3\text{)}$$



Modified Z-Score (Robust)

Problem with Standard Z-Score

Mean and standard deviation are sensitive to outliers themselves!

Modified Z-Score (Iglewicz & Hoaglin)

Uses median and MAD (Median Absolute Deviation):

$$M_i = \frac{0.6745 \cdot (x_i - \tilde{x})}{\text{MAD}}$$

Where:

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

Outlier criterion: $|M_i| > 3.5$

Why 0.6745?

This constant makes MAD consistent with standard deviation for normal distributions:

$\approx 1.4826 \times$ MAD

IQR Method (Tukey's Fences)

Interquartile Range Method

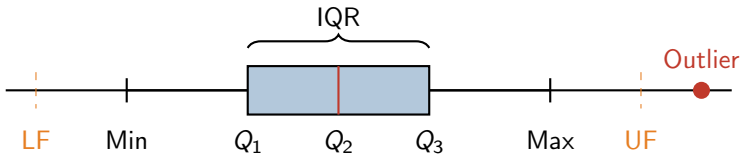
$$\text{IQR} = Q_3 - Q_1$$

Outlier boundaries:

$$\text{Lower Fence} = Q_1 - k \cdot \text{IQR}$$

$$\text{Upper Fence} = Q_3 + k \cdot \text{IQR}$$

Where $k = 1.5$ (outlier) or $k = 3$ (extreme outlier)



Multivariate Outlier Detection: Mahalanobis Distance

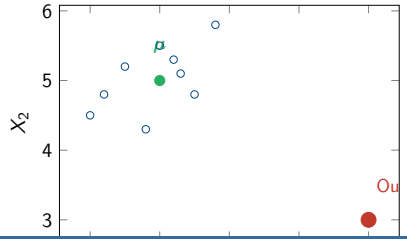
Mahalanobis Distance

Accounts for correlations between variables:

$$D_M(x_i) = \sqrt{(x_i - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (x_i - \boldsymbol{\mu})}$$

Where:

- $\boldsymbol{\mu}$ = mean vector
- $\boldsymbol{\Sigma}$ = covariance matrix



Machine Learning Methods: Isolation Forest

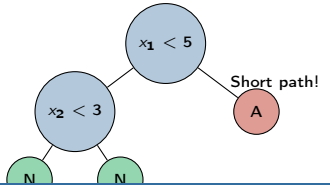
Isolation Forest Algorithm

Anomalies are easier to isolate (fewer splits needed):

$$s(x, n) = 2^{-\frac{E[h(x)]}{c(n)}}$$

Where:

- $h(x)$ = path length to isolate point x
- $c(n)$ = average path length in unsuccessful search in BST
- $s \approx 1$: anomaly, $s \approx 0.5$: normal, $s \approx 0$: very normal



Local Outlier Factor (LOF)

LOF Definition

Compares local density of a point to its neighbors:

$$\text{LOF}_k(x) = \frac{\sum_{o \in N_k(x)} \frac{\text{lrd}_k(o)}{\text{lrd}_k(x)}}{|N_k(x)|}$$

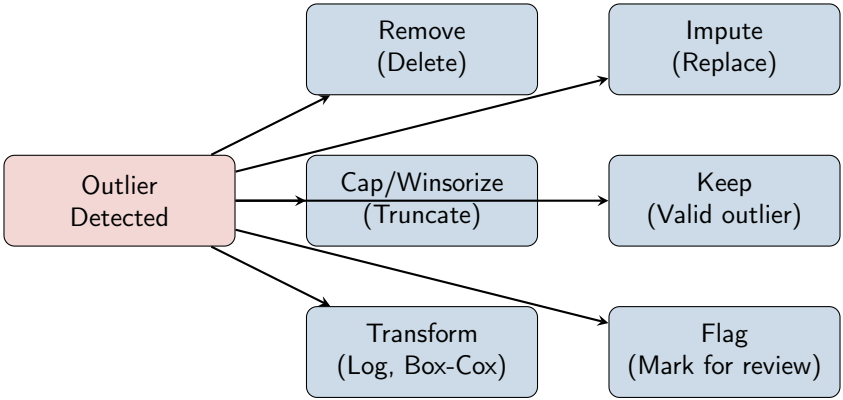
Where local reachability density:

$$\text{lrd}_k(x) = \frac{1}{\frac{\sum_{o \in N_k(x)} \text{reach-dist}_k(x,o)}{|N_k(x)|}}$$

Interpretation

- LOF ≈ 1: Similar density to neighbors (normal)
- LOF > 1: Lower density than neighbors (potential outlier)
- LOF $>> 1$: Much lower density (likely outlier)

Outlier Treatment Strategies



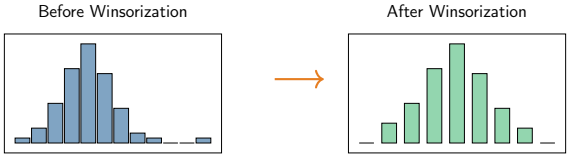
Winsorization (Capping)

Definition

Replace extreme values with specified percentiles:

$$x_i^{winsor} = \begin{cases} P_\alpha & \text{if } x_i < P_\alpha \\ P_{1-\alpha} & \text{if } x_i > P_{1-\alpha} \\ x_i & \text{otherwise} \end{cases}$$

Typically $\alpha = 0.01$ or $\alpha = 0.05$



Outlier Detection Summary

Method	Univariate	Multivariate	Best For
Z-Score	✓	✗	Normal distributions
Modified Z	✓	✗	Non-normal, robust
IQR/Tukey	✓	✗	Any distribution
Mahalanobis	✗	✓	Correlated features
Isolation Forest	✓	✓	High dimensions
LOF	✓	✓	Local anomalies
DBSCAN	✗	✓	Cluster-based

Important Consideration

Always investigate outliers before removing! They might be:

- Data entry errors (remove/correct)
- Genuine rare events (keep, possibly flag)
- Important signals (keep, they matter!)

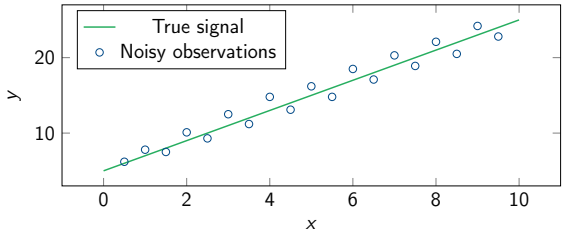
What is Noise in Data?

Definition

Noise is random error or variance in measured variables that obscures the true underlying pattern.

$$x_{observed} = x_{true} + \epsilon$$

Where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ (random noise)



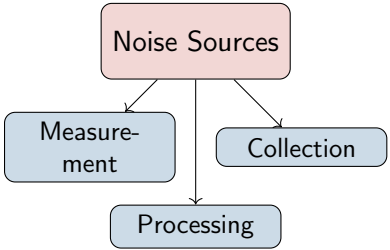
Sources of Noise

Measurement Noise

- Sensor inaccuracy
- Rounding errors
- Timing variations
- Environmental factors

Data Collection Noise

- Human errors
- Inconsistent processes
- Sampling bias



Binning (Discretization)

Smoothing by Binning

Partition data into bins and replace values with bin representative:

1. **Bin Means:**

$$x_i^{smooth} = \bar{x}_{bin(i)} = \frac{1}{|B_k|} \sum_{j \in B_k} x_j$$

2. **Bin Medians:**

$$x_i^{smooth} = \tilde{x}_{bin(i)}$$

3. **Bin Boundaries:**

$$x_i^{smooth} = \begin{cases} \min(B_k) & \text{if } x_i < \text{midpoint}(B_k) \\ \max(B_k) & \text{otherwise} \end{cases}$$

Binning Example

Example: Smoothing by Bin Means

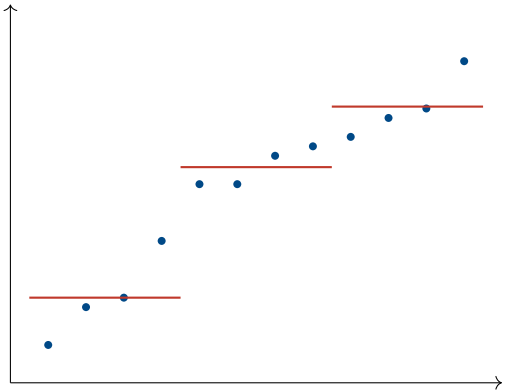
Original sorted data: 4, 8, 9, 15, 21, 21, 24, 25, 26, 28, 29, 34

Partition into bins (size 4):

- Bin 1: 4, 8, 9, 15 $\rightarrow$ Mean $= 9$
- Bin 2: 21, 21, 24, 25 $\rightarrow$ Mean $= 22.75$
- Bin 3: 26, 28, 29, 34 $\rightarrow$ Mean $= 29.25$

Smoothed data: 9, 9, 9, 9, 22.75, 22.75, 22.75, 22.75, 29.25, 29.25, 29.25, 29.25

Binning Example



Moving Average (Rolling Mean)

Simple Moving Average (SMA)

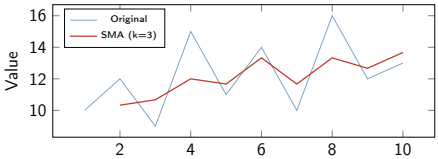
$$\text{SMA}_t = \frac{1}{k} \sum_{i=0}^{k-1} x_{t-i}$$

Where k is the window size.

Exponential Moving Average (EMA)

$$\text{EMA}_t = \alpha \cdot x_t + (1 - \alpha) \cdot \text{EMA}_{t-1}$$

Where $\alpha = \frac{2}{k+1}$ (smoothing factor)



Regression-Based Smoothing

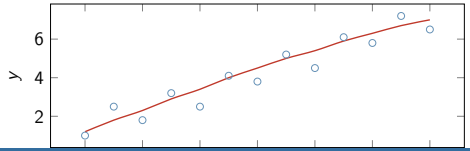
LOWESS/LOESS (Locally Weighted Scatterplot Smoothing)

Fits local weighted regression at each point:

$$\hat{y}_i = \sum_{j=1}^{n} w_{ij} \cdot y_j$$

Where weights decrease with distance (tricube kernel):

$$w_{ij} = \left(1 - \left|\frac{x_i - x_j}{d_i}\right|^3\right)^3$$



Signal-to-Noise Ratio (SNR)

Definition

Measure of signal strength relative to background noise:

$$\text{SNR} = \frac{\mu_{signal}}{\sigma_{noise}}$$

In decibels:

$$\text{SNR}_{dB} = 10 \cdot \log_{10}\left(\frac{P_{signal}}{P_{noise}}\right) = 20 \cdot \log_{10}\left(\frac{A_{signal}}{A_{noise}}\right)$$

Quality Interpretation

- SNR $>$ 20 dB: Excellent signal quality
- SNR 10-20 dB: Good signal quality
- SNR $<$ 10 dB: Poor, may need filtering

Data Validation Rules

Types of Validation

1. **Type Validation:** Data matches expected type
2. **Range Validation:** Values within acceptable bounds
3. **Format Validation:** Correct format (dates, emails, etc.)
4. **Consistency Validation:** Cross-field logic checks
5. **Uniqueness Validation:** Primary keys are unique

Example Rules

- Age: $0 \leq \text{age} \leq 150$ (range)
- Email: matches regex pattern (format)
- Birth date $<$ Hire date (consistency)
- Employee ID is unique (uniqueness)

Data Quality Metrics

Key Metrics to Track

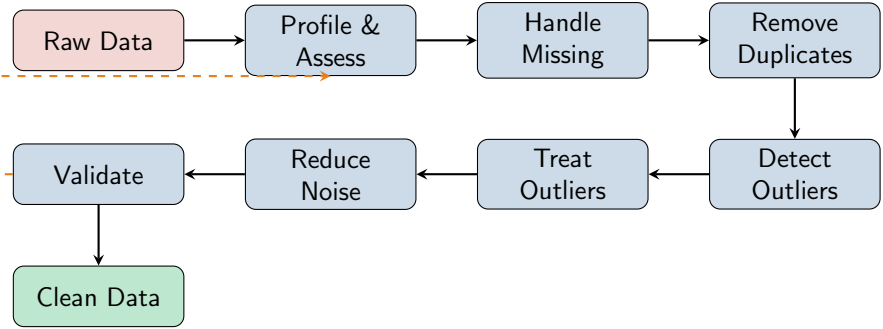
$$\text{Completeness} = \frac{\text{Non-null values}}{\text{Total values}} \times 100\%$$

$$\text{Validity} = \frac{\text{Values passing rules}}{\text{Total values}} \times 100\%$$

$$\text{Uniqueness} = \frac{\text{Distinct values}}{\text{Total values}} \times 100\%$$

$$\text{Consistency} = \frac{\text{Consistent records}}{\text{Total records}} \times 100\%$$

Data Cleaning Pipeline



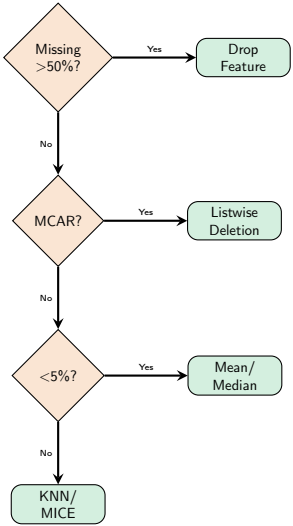
Remember

Data cleaning is **iterative**! Always validate results and be prepared to revisit earlier steps.

Data Cleaning Best Practices

1. **Document everything** — Keep logs of all changes
2. **Preserve original data** — Never modify source files
3. **Automate with scripts** — Reproducible pipelines
4. **Validate continuously** — Check quality at each step
5. **Understand the domain** — Context matters for decisions
6. **Handle edge cases** — Plan for unexpected values
7. **Test on samples first** — Verify before full processing
8. **Monitor data drift** — Ongoing quality checks

Decision Flowchart: Missing Values



Key Takeaways

Missing Values:

- Understand mechanism (MCAR/MAR/MNAR)
- Choose appropriate imputation
- MICE for best results

Duplicates:

- Check exact & fuzzy
- Use similarity metrics
- Document removal decisions

Outliers:

- Multiple detection methods
- Investigate before removing
- Consider domain context

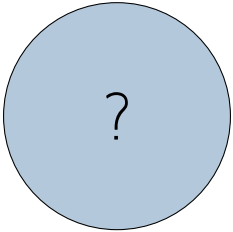
Noise:

- Smoothing techniques
- Balance signal preservation
- Monitor SNR

Next Part

Part 3: Data Transformation — Scaling, Encoding, and Feature Engineering

Questions & Discussion



Contact: professor@university.edu

References

- Rubin, D.B. (1976). *Inference and Missing Data*. Biometrika.
- Little, R.J.A. & Rubin, D.B. (2019). *Statistical Analysis with Missing Data*. Wiley.
- Aggarwal, C.C. (2017). *Outlier Analysis*. Springer.
- Han, J., Kamber, M., & Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.
- Wickham, H. (2014). *Tidy Data*. Journal of Statistical Software.

Part 3: Transformation

"Shape your data"

Part III

Practice for Part 2 : Practice for Data Cleaning

What We Will Cover

Practical Data Cleaning Skills

- Detecting and handling missing values
- Identifying and removing duplicates
- Detecting and treating outliers
- Correcting data types
- Cleaning text and string data
- Handling inconsistent data
- Building cleaning pipelines

Libraries Used

`pandas`, `numpy`, `scikit-learn`, `scipy`, `fuzzywuzzy`

Essential Libraries Import

```python
# Data manipulation
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn imputers
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Statistical functions
from scipy import stats

# String matching (install: pip install fuzzywuzzy python-Levenshtein)
from fuzzywuzzy import fuzz, process

# Display settings
pd.set_option('display.max_columns', None)
```

Detecting Missing Values

```python
# Check for missing values
print(df.isnull().sum())          # Count per column
print(df.isna().sum())            # Same as isnull()

# Total missing values
print(f"Total missing: {df.isnull().sum().sum()}")

# Percentage missing per column
missing_pct = (df.isnull().sum() / len(df) * 100).round(2)
print(missing_pct.sort_values(ascending=False))

# Rows with any missing value
rows_with_missing = df[df.isnull().any(axis=1)]
print(f"Rows with missing: {len(rows_with_missing)}")

# Columns with missing values
cols_with_missing = df.columns[df.isnull().any()].tolist()
print(f"Columns with missing: {cols_with_missing}")
```

Missing Values Summary Function

```python
def missing_summary(df):
    """Generate detailed missing values summary"""
    missing = df.isnull().sum()
    missing_pct = (missing / len(df) * 100).round(2)

    summary = pd.DataFrame({
        'Missing Count': missing,
        'Missing %': missing_pct,
        'Dtype': df.dtypes
    })

    # Filter only columns with missing values
    summary = summary[summary['Missing Count'] > 0]
    summary = summary.sort_values('Missing %', ascending=False)

    print(f"Total columns: {len(df.columns)}")
    print(f"Columns with missing: {len(summary)}")
    print(f"Total missing values: {missing.sum()}")

    return summary

print(missing_summary(df))
```

Visualizing Missing Values

```python
# Using missingno library (pip install missingno)
import missingno as msno

# Matrix visualization
msno.matrix(df, figsize=(12, 6))
plt.title('Missing Values Matrix')
plt.show()

# Bar chart of missing values
msno.bar(df, figsize=(12, 6))
plt.show()

# Heatmap of missing value correlations
msno.heatmap(df, figsize=(10, 8))
plt.show()

# Dendrogram showing missing value patterns
msno.dendrogram(df, figsize=(12, 6))
plt.show()
```

Custom Missing Values Heatmap

```python
# Create missing values heatmap with seaborn
plt.figure(figsize=(14, 8))
sns.heatmap(df.isnull(), cbar=True, yticklabels=False,
            cmap='viridis')
plt.title('Missing Values Heatmap')
plt.tight_layout()
plt.show()

# Missing values by row (identify problematic rows)
row_missing = df.isnull().sum(axis=1)
plt.figure(figsize=(12, 4))
plt.hist(row_missing, bins=50, edgecolor='black')
plt.xlabel('Number of Missing Values per Row')
plt.ylabel('Frequency')
plt.title('Distribution of Missing Values per Row')
plt.show()

# Rows with too many missing values
threshold = len(df.columns) * 0.5  # 50% missing
bad_rows = df[row_missing > threshold]
print(f"Rows with >50% missing: {len(bad_rows)}")
```

Deletion Methods

```python
# Drop rows with ANY missing value
df_clean = df.dropna()

# Drop rows with ALL missing values
df_clean = df.dropna(how='all')

# Drop rows with missing in specific columns
df_clean = df.dropna(subset=['col1', 'col2'])

# Drop rows with minimum non-null values (threshold)
df_clean = df.dropna(thresh=5)  # Keep rows with at least 5 non-null

# Drop columns with missing values
df_clean = df.dropna(axis=1)

# Drop columns with too many missing values
threshold = len(df) * 0.5  # 50% missing threshold
cols_to_drop = df.columns[df.isnull().sum() > threshold]
df_clean = df.drop(columns=cols_to_drop)
print(f"Dropped columns: {cols_to_drop.tolist()}")
```

Simple Imputation - Numerical

```python
# Fill with constant value
df['col'] = df['col'].fillna(0)

# Fill with mean
df['col'] = df['col'].fillna(df['col'].mean())

# Fill with median (robust to outliers)
df['col'] = df['col'].fillna(df['col'].median())

# Fill with mode (most frequent)
df['col'] = df['col'].fillna(df['col'].mode()[0])

# Fill multiple columns at once
numerical_cols = df.select_dtypes(include=np.number).columns
df[numerical_cols] = df[numerical_cols].fillna(
    df[numerical_cols].median()
)

# Fill with column-specific values
fill_values = {'col1': 0, 'col2': df['col2'].mean(), 'col3': -1}
df = df.fillna(fill_values)
```

Simple Imputation - Categorical

```python
# Fill with mode (most frequent category)
df['category'] = df['category'].fillna(df['category'].mode()[0])

# Fill with a specific category
df['category'] = df['category'].fillna('Unknown')
df['category'] = df['category'].fillna('Missing')

# Fill all categorical columns with mode
categorical_cols = df.select_dtypes(include=['object', 'category']).columns
for col in categorical_cols:
    df[col] = df[col].fillna(df[col].mode()[0])

# Or fill with 'Unknown'
df[categorical_cols] = df[categorical_cols].fillna('Unknown')

# Conditional filling
df['category'] = df['category'].fillna(
    df.groupby('other_col')['category'].transform(
        lambda x: x.mode()[0] if len(x.mode()) > 0 else 'Unknown'
    )
)
```

Forward and Backward Fill

```
# Forward fill (use previous value)
df['col'] = df['col'].fillna(method='ffill')
# Or using newer syntax:
df['col'] = df['col'].ffill()

# Backward fill (use next value)
df['col'] = df['col'].fillna(method='bfill')
# Or:
df['col'] = df['col'].bfill()

# Limit the fill (max consecutive NaNs to fill)
df['col'] = df['col'].ffill(limit=2)

# Combine forward and backward fill
df['col'] = df['col'].ffill().bfill()

# Useful for time series data
df = df.sort_values('date')
df['value'] = df['value'].ffill()
```

Interpolation Methods

```python
# Linear interpolation (default)
df['col'] = df['col'].interpolate(method='linear')

# Polynomial interpolation
df['col'] = df['col'].interpolate(method='polynomial', order=2)

# Spline interpolation (smooth curves)
df['col'] = df['col'].interpolate(method='spline', order=3)

# Time-based interpolation (for datetime index)
df['col'] = df['col'].interpolate(method='time')

# Index-based interpolation
df['col'] = df['col'].interpolate(method='index')

# Limit interpolation direction
df['col'] = df['col'].interpolate(limit=2, limit_direction='forward')

# Only interpolate inside (not at edges)
df['col'] = df['col'].interpolate(limit_area='inside')
```

Scikit-learn SimpleImputer

```python
from sklearn.impute import SimpleImputer

# Mean imputation
imputer_mean = SimpleImputer(strategy='mean')
df[numerical_cols] = imputer_mean.fit_transform(df[numerical_cols])

# Median imputation
imputer_median = SimpleImputer(strategy='median')
df[numerical_cols] = imputer_median.fit_transform(df[numerical_cols])

# Most frequent (mode) - works for categorical too
imputer_mode = SimpleImputer(strategy='most_frequent')
df[categorical_cols] = imputer_mode.fit_transform(df[categorical_cols])

# Constant value imputation
imputer_const = SimpleImputer(strategy='constant', fill_value=0)
df[numerical_cols] = imputer_const.fit_transform(df[numerical_cols])

# For categorical with constant
imputer_cat = SimpleImputer(strategy='constant', fill_value='Missing')
df[categorical_cols] = imputer_cat.fit_transform(df[categorical_cols])
```

KNN Imputation

```python
from sklearn.impute import KNNImputer

# KNN imputer (uses k nearest neighbors)
knn_imputer = KNNImputer(n_neighbors=5, weights='uniform')
df[numerical_cols] = knn_imputer.fit_transform(df[numerical_cols])

# Weighted by distance (closer neighbors have more influence)
knn_imputer = KNNImputer(n_neighbors=5, weights='distance')
df[numerical_cols] = knn_imputer.fit_transform(df[numerical_cols])

# Custom number of neighbors
knn_imputer = KNNImputer(n_neighbors=10)

# Note: KNNImputer only works with numerical data
# For mixed data, encode categoricals first or use separate imputers

# Scale data before KNN imputation for better results
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df[numerical_cols])
df_imputed = knn_imputer.fit_transform(df_scaled)
df[numerical_cols] = scaler.inverse_transform(df_imputed)
```

Iterative Imputer (MICE)

```python
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Basic iterative imputer (MICE - Multiple Imputation by Chained Equations)
iterative_imputer = IterativeImputer(max_iter=10, random_state=42)
df[numerical_cols] = iterative_imputer.fit_transform(df[numerical_cols])

# With specific estimator
from sklearn.linear_model import BayesianRidge
iterative_imputer = IterativeImputer(
    estimator=BayesianRidge(),
    max_iter=10,
    random_state=42
)
df[numerical_cols] = iterative_imputer.fit_transform(df[numerical_cols])

# With Random Forest for non-linear relationships
from sklearn.ensemble import RandomForestRegressor
iterative_imputer = IterativeImputer(
    estimator=RandomForestRegressor(n_estimators=10, random_state=42),
    max_iter=10,
    random_state=42
)
```

Group-Based Imputation

```python
# Impute based on group statistics
# Example: Fill missing salary with median salary by department

# Using groupby transform
df['salary'] = df.groupby('department')['salary'].transform(
    lambda x: x.fillna(x.median())
)

# Multiple group columns
df['salary'] = df.groupby(['department', 'level'])['salary'].transform(
    lambda x: x.fillna(x.median())
)

# Fallback to global median if group has all NaN
def group_impute(group):
    if group.isnull().all():
        return group.fillna(df['salary'].median())
    return group.fillna(group.median())

df['salary'] = df.groupby('department')['salary'].transform(group_impute)

# Custom function for complex logic
df['value'] = df.groupby('category')['value'].apply(
    lambda x: x.fillna(x.median() if x.notna().sum() > 0 else 0)
)
```

Imputation with Indicator

```python
# Create indicator column for missing values (useful for modeling)
df['col_was_missing'] = df['col'].isnull().astype(int)

# Then impute
df['col'] = df['col'].fillna(df['col'].median())

# Function to add missing indicators
def add_missing_indicators(df, columns):
    for col in columns:
        if df[col].isnull().sum() > 0:
            df[f'{col}_missing'] = df[col].isnull().astype(int)
    return df

df = add_missing_indicators(df, numerical_cols)

# Using SimpleImputer with add_indicator
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median', add_indicator=True)
result = imputer.fit_transform(df[numerical_cols])
# This returns both imputed values and indicator columns
```

Detecting Duplicates

```python
# Check for duplicate rows
print(f"Duplicate rows: {df.duplicated().sum()}")

# Check duplicates based on specific columns
print(f"Duplicate keys: {df.duplicated(subset=['id', 'date']).sum()}")

# View duplicate rows
duplicates = df[df.duplicated(keep=False)]  # keep=False shows all duplicates
print(duplicates.sort_values(by=df.columns.tolist()))

# First occurrence vs duplicates
first_occurrence = df[df.duplicated(keep='first') == False]
duplicate_rows = df[df.duplicated(keep='first') == True]

# Count duplicates per group
dup_counts = df.groupby(df.columns.tolist()).size().reset_index(name='count')
dup_counts = dup_counts[dup_counts['count'] > 1]
print(f"Groups with duplicates: {len(dup_counts)}")
```

Removing Duplicates

```python
# Remove exact duplicate rows (keep first occurrence)
df_clean = df.drop_duplicates()

# Keep last occurrence instead
df_clean = df.drop_duplicates(keep='last')

# Remove all duplicates (don't keep any)
df_clean = df.drop_duplicates(keep=False)

# Remove duplicates based on specific columns
df_clean = df.drop_duplicates(subset=['id', 'date'])

# Remove duplicates and reset index
df_clean = df.drop_duplicates().reset_index(drop=True)

# In-place removal
df.drop_duplicates(inplace=True)

# Keep row with most non-null values among duplicates
df_clean = df.loc[df.groupby(['id'])['col'].transform('count').idxmax()]
```

Handling Duplicates with Aggregation

```python
# Instead of dropping, aggregate duplicate rows
# Keep first value for some columns, mean for others

agg_rules = {
    'name': 'first',            # Keep first name
    'value': 'mean',            # Average the values
    'count': 'sum',             # Sum the counts
    'date': 'max',              # Keep latest date
    'category': lambda x: x.mode()[0]  # Most frequent
}

df_agg = df.groupby('id').agg(agg_rules).reset_index()

# Custom aggregation for complex cases
def resolve_duplicates(group):
    return pd.Series({
        'name': group['name'].iloc[0],
        'value': group['value'].mean(),
        'latest_date': group['date'].max(),
        'all_categories': ','.join(group['category'].unique())
    })

df_resolved = df.groupby('id').apply(resolve_duplicates).reset_index()
```

Fuzzy Duplicate Detection

```python
# For near-duplicates (e.g., "John Smith" vs "Jon Smith")
from fuzzywuzzy import fuzz, process

# Compare two strings
similarity = fuzz.ratio("John Smith", "Jon Smith")
print(f"Similarity: {similarity}%")  # Output: 91%

# Find potential duplicates in a column
def find_fuzzy_duplicates(df, column, threshold=85):
    duplicates = []
    values = df[column].unique()

    for i, val1 in enumerate(values):
        for val2 in values[i+1:]:
            score = fuzz.ratio(str(val1), str(val2))
            if score >= threshold:
                duplicates.append((val1, val2, score))

    return pd.DataFrame(duplicates, columns=['Value1', 'Value2', 'Score'])

fuzzy_dups = find_fuzzy_duplicates(df, 'name', threshold=85)
print(fuzzy_dups)
```

Standardizing Fuzzy Duplicates

```python
# Standardize similar values to one canonical form
from fuzzywuzzy import process

def standardize_column(df, column, threshold=85):
    """Standardize similar values to most frequent form"""
    value_counts = df[column].value_counts()
    mapping = {}

    for value in value_counts.index:
        if value not in mapping:
            # Find best match among more frequent values
            choices = [v for v in value_counts.index if v not in mapping.values()]
            match, score = process.extractOne(str(value), choices)
            if score >= threshold:
                mapping[value] = match
            else:
                mapping[value] = value

    return df[column].map(mapping)

df['name_clean'] = standardize_column(df, 'name', threshold=85)

# Check the mapping
print(df[['name', 'name_clean']].drop_duplicates())
```

IQR Method

```python
def detect_outliers_iqr(df, column, multiplier=1.5):
    """Detect outliers using IQR method"""
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - multiplier * IQR
    upper_bound = Q3 + multiplier * IQR

    outliers = (df[column] < lower_bound) | (df[column] > upper_bound)

    return outliers, lower_bound, upper_bound

# Apply to column
outlier_mask, lower, upper = detect_outliers_iqr(df, 'value')
print(f"Outliers: {outlier_mask.sum()}")
print(f"Bounds: [{lower:.2f}, {upper:.2f}]")

# View outliers
print(df[outlier_mask])

# Apply to all numerical columns
for col in numerical_cols:
    mask, _, _ = detect_outliers_iqr(df, col)
    print(f"{col}: {mask.sum()} outliers ({mask.mean()*100:.1f}%)")
```

Z-Score Method

```python
from scipy import stats

def detect_outliers_zscore(df, column, threshold=3):
    """Detect outliers using Z-score method"""
    z_scores = np.abs(stats.zscore(df[column].dropna()))
    outliers = z_scores > threshold
    return outliers

# Apply to column
outlier_mask = detect_outliers_zscore(df, 'value', threshold=3)
print(f"Outliers (|z| > 3): {outlier_mask.sum()}")

# Calculate z-scores for all rows
df['z_score'] = np.abs(stats.zscore(df['value']))
df['is_outlier'] = df['z_score'] > 3

# Modified Z-score (more robust, uses median)
def modified_zscore(x):
    median = np.median(x)
    mad = np.median(np.abs(x - median))
    return 0.6745 * (x - median) / mad

df['modified_z'] = np.abs(modified_zscore(df['value']))
df['is_outlier_robust'] = df['modified_z'] > 3.5
```

Isolation Forest

```python
from sklearn.ensemble import IsolationForest

# Isolation Forest for multivariate outlier detection
iso_forest = IsolationForest(
    n_estimators=100,
    contamination=0.05,  # Expected proportion of outliers
    random_state=42
)

# Fit and predict (-1 for outliers, 1 for inliers)
df['outlier_if'] = iso_forest.fit_predict(df[numerical_cols])
df['is_outlier_if'] = df['outlier_if'] == -1

print(f"Outliers detected: {(df['outlier_if'] == -1).sum()}")

# Get anomaly scores (lower = more anomalous)
df['anomaly_score'] = iso_forest.decision_function(df[numerical_cols])

# Visualize
plt.figure(figsize=(10, 6))
plt.scatter(df.index, df['anomaly_score'], c=df['outlier_if'], cmap='RdYlGn')
plt.xlabel('Index')
plt.ylabel('Anomaly Score')
plt.title('Isolation Forest Anomaly Scores')
plt.show()
```

Local Outlier Factor (LOF)

```python
from sklearn.neighbors import LocalOutlierFactor

# LOF for density-based outlier detection
lof = LocalOutlierFactor(
    n_neighbors=20,
    contamination=0.05
)

# Fit and predict
df['outlier_lof'] = lof.fit_predict(df[numerical_cols])
df['is_outlier_lof'] = df['outlier_lof'] == -1

print(f"LOF outliers: {(df['outlier_lof'] == -1).sum()}")

# Get LOF scores (negative_outlier_factor_)
df['lof_score'] = -lof.negative_outlier_factor_

# Visualize 2D projection with outliers
plt.figure(figsize=(10, 8))
colors = ['red' if x == -1 else 'blue' for x in df['outlier_lof']]
plt.scatter(df['col1'], df['col2'], c=colors, alpha=0.5)
plt.xlabel('Column 1')
plt.ylabel('Column 2')
plt.title('LOF Outlier Detection')
plt.show()
```

DBSCAN for Outlier Detection

```python
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Scale data first
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df[numerical_cols])

# DBSCAN clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
df['cluster'] = dbscan.fit_predict(X_scaled)

# Points labeled -1 are outliers (noise)
df['is_outlier_dbscan'] = df['cluster'] == -1

print(f"DBSCAN outliers: {(df['cluster'] == -1).sum()}")
print(f"Number of clusters: {df['cluster'].max() + 1}")

# Visualize
plt.figure(figsize=(10, 8))
scatter = plt.scatter(df['col1'], df['col2'], c=df['cluster'], cmap='viridis')
plt.colorbar(scatter)
plt.title('DBSCAN Clustering (-1 = Outliers)')
plt.show()
```

Removing Outliers

```python
# Remove outliers detected by IQR
outlier_mask, lower, upper = detect_outliers_iqr(df, 'value')
df_clean = df[~outlier_mask]
print(f"Removed {outlier_mask.sum()} outliers")

# Remove outliers from multiple columns
def remove_outliers_iqr(df, columns, multiplier=1.5):
    mask = pd.Series([False] * len(df))
    for col in columns:
        outliers, _, _ = detect_outliers_iqr(df, col, multiplier)
        mask = mask | outliers
    return df[~mask]

df_clean = remove_outliers_iqr(df, numerical_cols)

# Remove based on Z-score
z_scores = np.abs(stats.zscore(df[numerical_cols]))
df_clean = df[(z_scores < 3).all(axis=1)]

# Remove based on percentiles
lower = df['value'].quantile(0.01)
upper = df['value'].quantile(0.99)
df_clean = df[(df['value'] >= lower) & (df['value'] <= upper)]
```

Capping / Winsorization

```python
# Cap outliers to boundary values (Winsorization)
def cap_outliers_iqr(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - multiplier * IQR
    upper = Q3 + multiplier * IQR

    df[column] = df[column].clip(lower=lower, upper=upper)
    return df

df = cap_outliers_iqr(df, 'value')

# Cap using percentiles
def cap_percentile(df, column, lower_pct=0.01, upper_pct=0.99):
    lower = df[column].quantile(lower_pct)
    upper = df[column].quantile(upper_pct)
    df[column] = df[column].clip(lower=lower, upper=upper)
    return df

# Using scipy's winsorize
from scipy.stats.mstats import winsorize
df['value_winsorized'] = winsorize(df['value'], limits=[0.05, 0.05])
```

Transformation Methods

```python
# Log transformation (for right-skewed data)
df['value_log'] = np.log1p(df['value'])  # log(1+x) handles zeros

# Square root transformation
df['value_sqrt'] = np.sqrt(df['value'])

# Box-Cox transformation (requires positive values)
from scipy.stats import boxcox
df['value_boxcox'], lambda_param = boxcox(df['value'] + 1)
print(f"Optimal lambda: {lambda_param}")

# Yeo-Johnson transformation (handles negative values)
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer(method='yeo-johnson')
df['value_yeojohnson'] = pt.fit_transform(df[['value']])

# Compare distributions
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
df['value'].hist(ax=axes[0], bins=30)
axes[0].set_title('Original')
df['value_log'].hist(ax=axes[1], bins=30)
axes[1].set_title('Log Transform')
df['value_boxcox'].hist(ax=axes[2], bins=30)
axes[2].set_title('Box-Cox Transform')
plt.tight_layout()
```

Imputing Outliers

```python
# Replace outliers with median
def impute_outliers_median(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - multiplier * IQR
    upper = Q3 + multiplier * IQR
    median = df[column].median()

    df[column] = df[column].apply(
        lambda x: median if (x < lower or x > upper) else x
    )
    return df

# Replace outliers with NaN, then impute
def outliers_to_nan(df, column, multiplier=1.5):
    mask, lower, upper = detect_outliers_iqr(df, column, multiplier)
    df.loc[mask, column] = np.nan
    return df

df = outliers_to_nan(df, 'value')
df['value'] = df['value'].fillna(df['value'].median())
```

Converting Data Types

```python
# Check current data types
print(df.dtypes)

# Convert to numeric (coerce errors to NaN)
df['col'] = pd.to_numeric(df['col'], errors='coerce')

# Convert to integer (must handle NaN first)
df['col'] = df['col'].fillna(0).astype(int)

# Nullable integer type (handles NaN)
df['col'] = df['col'].astype('Int64')  # Note: capital I

# Convert to float
df['col'] = df['col'].astype(float)

# Convert to string
df['col'] = df['col'].astype(str)

# Convert to category (memory efficient)
df['category_col'] = df['category_col'].astype('category')

# Convert to boolean
df['bool_col'] = df['bool_col'].astype(bool)
df['bool_col'] = df['col'].map({'Yes': True, 'No': False})
```

Converting Datetime

```python
# Convert to datetime
df['date'] = pd.to_datetime(df['date'])

# Specify format for faster parsing
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')

# Handle different formats
df['date'] = pd.to_datetime(df['date'], format='mixed', dayfirst=True)

# Handle errors
df['date'] = pd.to_datetime(df['date'], errors='coerce')  # Invalid -> NaT

# Extract components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['dayofweek'] = df['date'].dt.dayofweek
df['hour'] = df['date'].dt.hour

# Convert Unix timestamp to datetime
df['date'] = pd.to_datetime(df['timestamp'], unit='s')

# Convert datetime to string
df['date_str'] = df['date'].dt.strftime('%Y-%m-%d')
```

Automatic Type Inference

```python
# Infer better dtypes automatically
df = df.infer_objects()

# Convert object columns to best possible types
df = df.convert_dtypes()

# Custom type conversion function
def optimize_dtypes(df):
    for col in df.columns:
        col_type = df[col].dtype

        if col_type == 'object':
            # Try to convert to numeric
            try:
                df[col] = pd.to_numeric(df[col])
            except:
                # Try to convert to datetime
                try:
                    df[col] = pd.to_datetime(df[col])
                except:
                    # Convert to category if low cardinality
                    if df[col].nunique() / len(df) < 0.5:
                        df[col] = df[col].astype('category')
    return df

df = optimize_dtypes(df)
```

Basic String Cleaning

```python
# Remove leading/trailing whitespace
df['col'] = df['col'].str.strip()

# Remove extra whitespace
df['col'] = df['col'].str.replace(r'\s+', ' ', regex=True)

# Convert to lowercase/uppercase
df['col'] = df['col'].str.lower()
df['col'] = df['col'].str.upper()
df['col'] = df['col'].str.title()  # Title Case

# Remove specific characters
df['col'] = df['col'].str.replace('[^\w\s]', '', regex=True)  # Keep only alphanumeric

# Remove digits
df['col'] = df['col'].str.replace(r'\d+', '', regex=True)

# Remove specific strings
df['col'] = df['col'].str.replace('unwanted', '', regex=False)

# Replace multiple values
df['col'] = df['col'].replace({'old1': 'new1', 'old2': 'new2'})
```

Advanced String Operations

```python
# Extract patterns with regex
df['phone'] = df['text'].str.extract(r'(\d{3}-\d{3}-\d{4})')
df['email'] = df['text'].str.extract(r'([\w.-]+@[\w.-]+)')

# Split strings
df[['first_name', 'last_name']] = df['full_name'].str.split(' ', n=1, expand=True)

# Check for patterns
df['has_email'] = df['text'].str.contains(r'@', regex=True)
df['starts_with_a'] = df['col'].str.startswith('A')

# Pad strings
df['code'] = df['code'].str.zfill(5)  # Pad with zeros: '42' -> '00042'

# Slice strings
df['first_3'] = df['col'].str[:3]
df['last_4'] = df['col'].str[-4:]

# Get string length
df['length'] = df['col'].str.len()

# Count occurrences
df['word_count'] = df['text'].str.count(r'\w+')
```

Standardizing Categorical Values

```python
# Map inconsistent values to standard forms
mapping = {
    'yes': 'Yes', 'YES': 'Yes', 'y': 'Yes', 'Y': 'Yes', '1': 'Yes',
    'no': 'No', 'NO': 'No', 'n': 'No', 'N': 'No', '0': 'No'
}
df['response'] = df['response'].map(mapping).fillna(df['response'])

# Using replace with regex
df['col'] = df['col'].replace({
    r'^[Yy]es.*': 'Yes',
    r'^[Nn]o.*': 'No'
}, regex=True)

# Standardize case and strip
def standardize_text(text):
    if pd.isna(text):
        return text
    return str(text).strip().lower().title()

df['name'] = df['name'].apply(standardize_text)

# Fix common typos
typo_fixes = {'teh': 'the', 'recieve': 'receive', 'seperate': 'separate'}
for typo, fix in typo_fixes.items():
    df['text'] = df['text'].str.replace(typo, fix, case=False)
```

Complete Cleaning Function

```python
def clean_dataframe(df):
    """Comprehensive data cleaning pipeline"""
    df = df.copy()

    # 1. Remove duplicate rows
    initial_rows = len(df)
    df = df.drop_duplicates()
    print(f"Removed {initial_rows - len(df)} duplicate rows")

    # 2. Handle column names
    df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

    # 3. Identify column types
    num_cols = df.select_dtypes(include=np.number).columns.tolist()
    cat_cols = df.select_dtypes(include='object').columns.tolist()

    # 4. Clean string columns
    for col in cat_cols:
        df[col] = df[col].str.strip()

    # 5. Handle missing values
    # Numerical: median imputation
    for col in num_cols:
        df[col] = df[col].fillna(df[col].median())

    return df
```

Sklearn Pipeline for Cleaning

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Define preprocessing for different column types
numerical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='Missing')),
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

# Combine pipelines
preprocessor = ColumnTransformer([
    ('num', numerical_pipeline, numerical_cols),
    ('cat', categorical_pipeline, categorical_cols)
])

# Fit and transform
X_clean = preprocessor.fit_transform(df)
```

Custom Transformer

```python
from sklearn.base import BaseEstimator, TransformerMixin

class OutlierRemover(BaseEstimator, TransformerMixin):
    def __init__(self, multiplier=1.5):
        self.multiplier = multiplier
        self.bounds_ = {}

    def fit(self, X, y=None):
        for col in X.columns:
            Q1, Q3 = X[col].quantile([0.25, 0.75])
            IQR = Q3 - Q1
            self.bounds_[col] = (Q1 - self.multiplier * IQR,
                                 Q3 + self.multiplier * IQR)
        return self

    def transform(self, X):
        X = X.copy()
        for col in X.columns:
            lower, upper = self.bounds_[col]
            X[col] = X[col].clip(lower=lower, upper=upper)
        return X

# Use in pipeline
pipeline = Pipeline([
    ('outlier', OutlierRemover(multiplier=1.5)),
    ('imputer', SimpleImputer(strategy='median'))
])
```

Saving and Loading Clean Data

```python
# Save cleaned data
df_clean.to_csv('data_cleaned.csv', index=False)
df_clean.to_parquet('data_cleaned.parquet', index=False)  # Better for large files
df_clean.to_pickle('data_cleaned.pkl')  # Preserves dtypes

# Save with compression
df_clean.to_csv('data_cleaned.csv.gz', compression='gzip', index=False)

# Load cleaned data
df = pd.read_csv('data_cleaned.csv')
df = pd.read_parquet('data_cleaned.parquet')
df = pd.read_pickle('data_cleaned.pkl')

# Save preprocessing pipeline (for production)
import joblib
joblib.dump(preprocessor, 'preprocessor.pkl')

# Load and apply to new data
preprocessor = joblib.load('preprocessor.pkl')
X_new_clean = preprocessor.transform(df_new)
```

Data Cleaning Checklist

Missing Values:

- ☐ Identify missing patterns
- ☐ Decide: delete or impute
- ☐ Apply appropriate imputation
- ☐ Add missing indicators if needed

Duplicates:

- ☐ Check exact duplicates
- ☐ Check key-based duplicates
- ☐ Check fuzzy duplicates
- ☐ Remove or aggregate

Outliers:

- ☐ Detect with IQR/Z-score
- ☐ Verify if real or errors
- ☐ Remove, cap, or transform

Data Types & Text:

- ☐ Fix incorrect types
- ☐ Parse dates properly
- ☐ Clean string columns
- ☐ Standardize categories

Key Python Functions Reference

Task	Function
Detect missing	`df.isnull().sum()`, `msno.matrix()`
Drop missing	`df.dropna()`, `df.dropna(subset=[...])`
Fill missing	`df.fillna()`, `df.interpolate()`
Impute (sklearn)	`SimpleImputer`, `KNNImputer`
Detect duplicates	`df.duplicated()`, `fuzz.ratio()`
Remove duplicates	`df.drop_duplicates()`
Detect outliers	`zscore()`, `IsolationForest`
Cap outliers	`df.clip()`, `winsorize()`
Convert types	`astype()`, `pd.to_numeric()`
Clean strings	`str.strip()`, `str.replace()`

Questions & Practice

Next: Part 3 - Data Transformation with Python

Part IV

Part 3 : Data Transformation

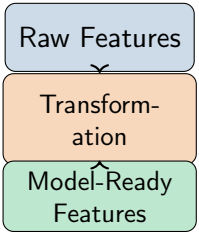
What is Data Transformation?

Definition

Data Transformation is the process of converting data from one format, structure, or value range to another to make it suitable for machine learning algorithms.

Main Objectives:

- Normalize feature scales
- Encode categorical variables
- Create new informative features
- Reduce dimensionality
- Handle non-linear relationships



Why Transform Data?

Scale Difference
Age: 0-100
Income: 0-1M

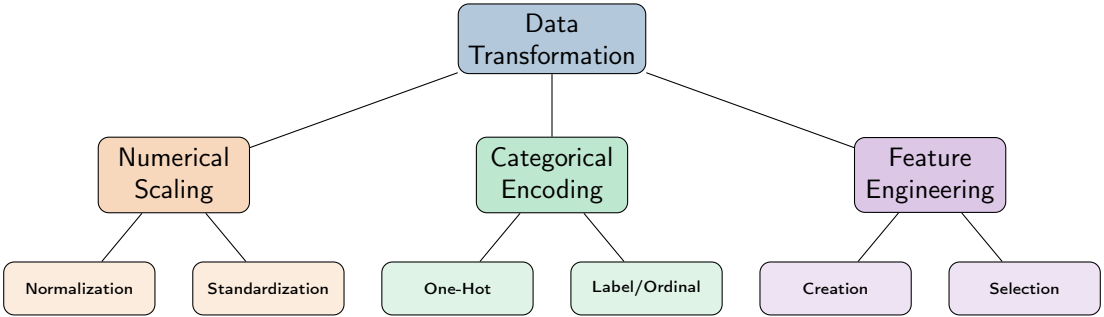
Categorical Data
Color: Red, Blue
ML needs numbers

Non-linearity
Exponential growth
Linear models fail

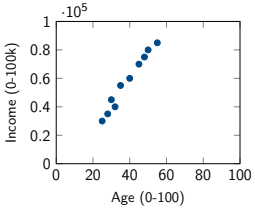
Impact on ML Algorithms

- **Gradient-based methods** (Neural Networks, Logistic Regression): Sensitive to feature scales
- **Distance-based methods** (KNN, K-Means, SVM): Dominated by large-scale features
- **Tree-based methods** (Random Forest, XGBoost): Generally scale-invariant

Transformation Categories Overview

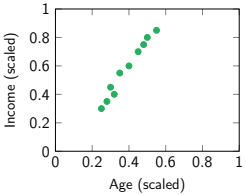


Why Scale Numerical Features?



Before Scaling

Income dominates distance!



After Scaling

Equal contribution!

Euclidean Distance Example

$$d = \sqrt{(100 - 0)^2 + (100000 - 0)^2} \approx 100000 \quad \text{(Income dominates!)}$$

Min-Max Normalization (Rescaling)

Formula

Scales features to a fixed range $[a, b]$ (typically $[0, 1]$):

$$x_i' = \frac{x_i - x_{min}}{x_{max} - x_{min}} \cdot (b - a) + a$$

For $[0, 1]$ range:

$$x_i' = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

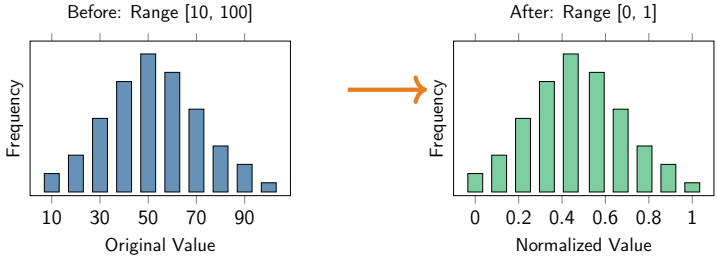
Properties:

- Output range: $[0, 1]$ or $[a, b]$
- Preserves zero values (if $a = 0$)
- Preserves relationships

Limitations:

- Sensitive to outliers
- New data may fall outside $[0, 1]$
- Doesn't center data

Min-Max Normalization: Visual Example



Calculation Example

$x = 55$, $x_{min} = 10$, $x_{max} = 100$

$$x' = \frac{55 - 10}{100 - 10} = \frac{45}{90} = 0.5$$

Z-Score Standardization

Formula

Transforms data to have mean $\mu = 0$ and standard deviation $\sigma = 1$:

$$z_i = \frac{x_i - \mu}{\sigma} = \frac{x_i - \bar{x}}{s}$$

Where:

- $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$ (sample mean)
- $s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ (sample std dev)

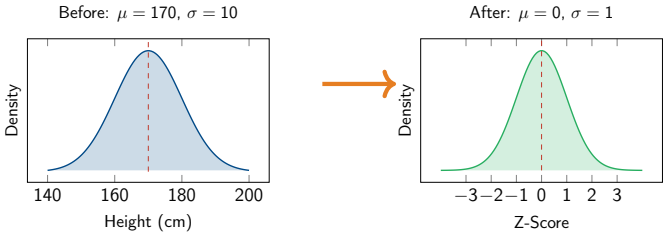
Properties:

- Centered at zero
- Unit variance
- Unbounded range

Best for:

- Gaussian-like distributions
- Algorithms assuming $\mathcal{N}(0, 1)$
- When outliers present

Z-Score Standardization: Visual



Interpretation

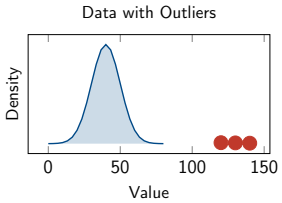
A z-score of $z = 2$ means the value is 2 standard deviations above the mean.

Robust Scaling

Formula

Uses median and IQR (robust to outliers):

$$x_i' = \frac{x_i - \text{median}(x)}{\text{IQR}} = \frac{x_i - Q_2}{Q_3 - Q_1}$$



Z-Score: Affected by outliers
σ inflated $\rightarrow$ poor scaling

Robust: Uses median/IQR
Outliers have minimal impact

MaxAbs Scaling

Formula

Scales by maximum absolute value (preserves sparsity):

$$x_i' = \frac{x_i}{\max(|x|)} = \frac{x_i}{|x_{max}|}$$

Result range: $[-1, 1]$

Use Case: Sparse Data

- Preserves zero entries (important for sparse matrices)
- No centering (doesn't destroy sparsity)
- Useful for data already centered at zero

Example

Data: $[-10, -5, 0, 3, 8]$, $\max(|x|) = 10$

Power Transformations: Box-Cox

Box-Cox Transformation

Makes data more Gaussian-like:

$$x_i' = \begin{cases} \frac{x_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \ln(x_i) & \text{if } \lambda = 0 \end{cases}$$

Constraint: $x_i > 0$ (all values must be positive)

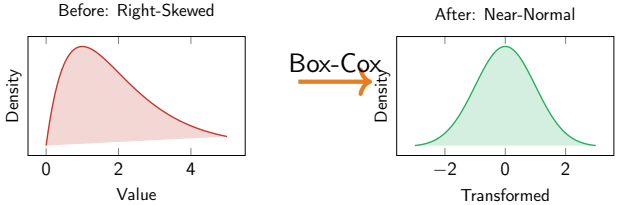
Special Cases:

- $\lambda = 1$: No transformation
- $\lambda = 0$: Log transform
- $\lambda = 0.5$: Square root
- $\lambda = -1$: Reciprocal

λ Selection:

- Maximum Likelihood Estimation
- Optimizes normality
- Typically $\lambda \in [-5, 5]$

Box-Cox Transformation: Visual



When to Use

- Linear regression (assumes normal residuals)
- Algorithms sensitive to distribution shape
- Reducing heteroscedasticity

Yeo-Johnson Transformation

Formula (handles negative values)

$$x_i' = \begin{cases} \frac{(x_i+1)^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, x_i \geq 0 \\ \ln(x_i + 1) & \text{if } \lambda = 0, x_i \geq 0 \\ \frac{-((-x_i+1)^{2-\lambda} - 1)}{2-\lambda} & \text{if } \lambda \neq 2, x_i < 0 \\ -\ln(-x_i + 1) & \text{if } \lambda = 2, x_i < 0 \end{cases}$$

Advantage over Box-Cox

- Works with zero and negative values
- No need to shift data first
- More flexible for real-world data

Log and Square Root Transformations

Log Transformation

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

Where c ensures $x + c > 0$

Use for:

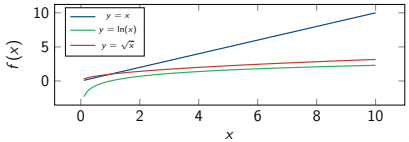
- Right-skewed data
- Multiplicative relationships
- Data spanning orders of magnitude

Square Root Transformation

$$x' = \sqrt{x}$$

Use for:

- Count data
- Moderate right skew
- Variance stabilization



Caution

Non-parametric, so doesn't preserve relationships between features. Best for single features.

Definition

Maps data to a specified distribution (uniform or normal) using quantiles:

$$x_i' = \Phi^{-1}(F(x_i))$$

Where:

- $F(x_i)$ = empirical CDF (rank / n)
- Φ^{-1} = inverse CDF of target distribution

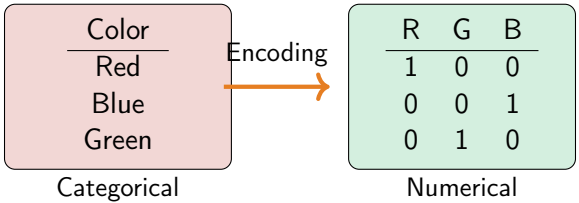
Scaling Methods Comparison

Method	Range	Centered	Outlier Robust	Sparse Safe	Use Case
Min-Max	$[0, 1]$	No	✗	✗	Bounded data
Z-Score	$(-\infty, \infty)$	Yes	✗	✗	Gaussian data
Robust	$(-\infty, \infty)$	Yes	✓	✗	Outliers present
MaxAbs	$[-1, 1]$	No	✗	✓	Sparse matrices
Box-Cox	Varies	Varies	Partial	✗	Non-normal
Quantile	Varies	Varies	✓	✗	Any distribution

Why Encode Categorical Features?

The Problem

Machine learning algorithms work with **numerical data**. Categorical variables must be converted to numbers.



Important

The encoding method matters! Wrong encoding can introduce false ordinal relationships.

Label Encoding (Integer Encoding)

Definition

Assigns a unique integer to each category:

$$f : \mathcal{C} \to \{0, 1, 2, ..., k-1\}$$

Where k = number of unique categories

Example

Category	Encoded
Red	0
Blue	1
Green	2

Problem!

Implies false ordering:

$$\text{Red} < \text{Blue} < \text{Green}$$

And false distances:

$$d(\text{Red}, \text{Green}) = 2$$

Ordinal Encoding

Definition

Like label encoding, but respects natural ordering:

$$f : \mathcal{C}_{ordered} \rightarrow \{0, 1, 2, ..., k-1\}$$

Where ordering is meaningful

Appropriate Examples

Education Level:

Level	Code
High School	0
Bachelor's	1
Master's	2
PhD	3

Size:

Size	Code
Small	0
Medium	1
Large	2
XL	3

One-Hot Encoding (Dummy Variables)

Definition

Creates binary columns for each category:

$$x \in \mathcal{C} \rightarrow \mathrm{e}_x \in \{0,1\}^k$$

Where e_x is a one-hot vector with 1 at position x

Color	is_Blue	
Red	0	
Blue	1	
Green	0	
Red	0	

One-Hot Encoding: Pros and Cons

Advantages

- No ordinal assumption
- Equal distance between categories:

$$d(\mathrm{e}_i, \mathrm{e}_j) = \sqrt{2} \quad \forall i \neq j$$

- Works with all algorithms
- Interpretable coefficients

Disadvantages

- **Curse of dimensionality:**
  k categories $\rightarrow$ k (or $k-1$) columns
- High cardinality problems
- Sparse matrices
- Multicollinearity (dummy trap)

Dummy Variable Trap

Drop one column to avoid perfect multicollinearity:

$$\text{is\_Red} + \text{is\_Blue} + \text{is\_Green} = 1 \quad \text{(always)}$$

Binary Encoding

Definition

Encode categories as binary numbers, then split into columns:

$$\text{Category} \xrightarrow{\text{integer}} n \xrightarrow{\text{binary}} (b_1, b_2, ..., b_m)$$

Where $m = \lceil \log_2(k) \rceil$ columns needed for k categories

Example with 5 categories

Category	Integer	Binary	b_1	b_2	b_3
A	0	000	0	0	0
B	1	001	0	0	1
C	2	010	0	1	0
D	3	011	0	1	1
E	4	100	1	0	0

5 categories → 3 columns (vs 5 for one-hot)

Target Encoding (Mean Encoding)

Definition

Replace category with mean of target variable for that category:

$$x'_c = \frac{1}{n_c} \sum_{i:x_i=c} y_i = \bar{y}_c$$

Where n_c = count of samples with category c

Example: Predicting Salary

Department	Avg Salary	Encoded
Engineering	\$95,000	95000
Marketing	\$75,000	75000
Sales	\$65,000	65000

Danger: Data Leakage!

Target Encoding: Smoothing

Smoothed Target Encoding

Blend category mean with global mean to handle rare categories:

$$x'_c = \lambda(n_c) \cdot \bar{y}_c + (1 - \lambda(n_c)) \cdot \bar{y}_{global}$$

Where smoothing factor:

$$\lambda(n) = \frac{n}{n + m}$$

m = smoothing parameter (e.g., 10-100)

Interpretation

- Large n_c: $\lambda \approx 1$, trust category mean
- Small n_c: $\lambda \approx 0$, use global mean

Frequency Encoding

Definition

Replace category with its frequency (count or proportion):

$$x'_c = \frac{n_c}{n} \quad \text{or} \quad x'_c = n_c$$

Example

City	Count	Freq
NYC	500	0.50
LA	300	0.30
Chicago	200	0.20

Pros:

- No target leakage
- Single column output
- Captures rarity

Cons:

- Same encoding for different categories with same frequency

Encoding Methods Comparison

Method	Output Dims	High Cardinality	Target Leakage	Best For
Label	1	✓	✓	Trees, Ordinal
One-Hot	k or $k-1$	✗	✓	Low cardinality
Binary	$\log_2(k)$	✓	✓	Medium cardinality
Target	1	✓	✗	Supervised (with CV)
Frequency	1	✓	✓	Any

Rule of Thumb

- $k < 10$: One-Hot encoding
- $10 \leq k < 100$: Binary or Target encoding
- $k \geq 100$: Target or Frequency encoding

Andrew Ng's Quote

"Coming up with features is difficult, time-consuming, requires expert knowledge. Applied machine learning is basically feature engineering."

Definition

Feature Engineering is the process of using domain knowledge to create new features from raw data that better represent the underlying problem to the model.

Mathematical Feature Creation

Arithmetic Operations

Combine features using math operations:

$$x_{new} = f(x_1, x_2, ..., x_n)$$

Common Operations

- **Sum:** $x_1 + x_2$
- **Difference:** $x_1 - x_2$
- **Product:** $x_1 \times x_2$
- **Ratio:** $\frac{x_1}{x_2}$
- **Power:** x^2, x^3
- **Root:** $\sqrt{x}$
- **Log:** $\log(x)$
- **Exponential:** e^x

Real Example: BMI Calculation

$$\text{BMI} = \frac{\text{Weight (kg)}}{\text{Height (m)}^2}$$

Polynomial Features

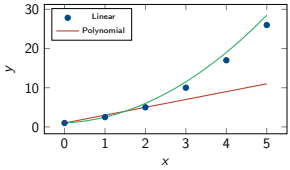
Definition

Generate polynomial and interaction terms up to degree d:

$$\phi(x_1, x_2) = (1, x_1, x_2, x_1^2, x_1x_2, x_2^2, x_1^3, ...)$$

Example: Degree 2 with 2 features

Input: (x_1, x_2)
Output: $(1, x_1, x_2, x_1^2, x_1x_2, x_2^2)$
Number of features: $\binom{n+d}{d} = \binom{2+2}{2} = 6$



Datetime Feature Engineering

Extracting Components from Datetime

$$\text{datetime} \rightarrow \{year, month, day, hour, minute, dayofweek, ...\}$$

Common Datetime Features

Direct Extraction:

- Year, Month, Day
- Hour, Minute, Second
- Day of week (0-6)
- Day of year (1-365)
- Week of year
- Quarter

Derived Features:

- Is weekend (binary)
- Is holiday (binary)
- Time since event
- Part of day (morning/afternoon)
- Season
- Is month start/end

Cyclical Encoding for Time Features

Problem with Linear Encoding

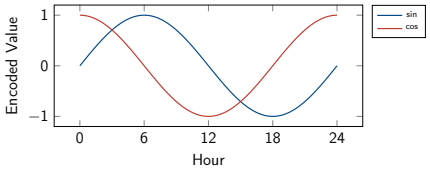
Hour 23 and Hour 0 should be close, but:

$$|23 - 0| = 23 \quad \text{(Wrong! They're 1 hour apart)}$$

Solution: Sine-Cosine Encoding

$$x_{sin} = \sin\left(\frac{2\pi \cdot t}{T}\right), \quad x_{cos} = \cos\left(\frac{2\pi \cdot t}{T}\right)$$

Where T = period (24 for hours, 7 for days, 12 for months)



Binning (Discretization)

Definition

Convert continuous features into discrete bins/categories:

$$x \in \mathbb{R} \rightarrow b \in \{1, 2, ..., k\}$$

Equal-Width Binning

$$\text{width} = \frac{x_{max} - x_{min}}{k}$$

Bin boundaries evenly spaced

Equal-Frequency (Quantile)

Each bin has $\approx \frac{n}{k}$ samples

Uses percentiles as boundaries

Example: Age Binning

Age	Equal-Width	Domain-Based
25	20-40	Young Adult
45	40-60	Middle Age

Text Feature Engineering

Basic Text Features

- **Length:** Character count, word count
- **Statistics:** Average word length, unique word ratio
- **Patterns:** Number of digits, special chars, uppercase ratio

Bag of Words (BoW)

Count occurrences of each word:

$$\text{BoW}(d) = (c_1, c_2, ..., c_V)$$

Where c_i = count of word i in document d

TF-IDF (Term Frequency - Inverse Document Frequency)

$$\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$$

Aggregation Features

Group-Based Statistics

Compute statistics within groups:

$$x_{agg} = \text{Agg}_{group}(x)$$

Common Aggregations

- Mean: $\bar{x}_{group}$
- Median: $\tilde{x}_{group}$
- Sum: $\sum x_{group}$
- Count: n_{group}
- Min/Max
- Standard deviation
- Skewness, Kurtosis
- Range

Example: Customer Features

Raw	**Engineered**
Transaction amounts	Avg transaction per customer
Purchase dates	Days since last purchase

Aggregation Features

Example: Customer Features

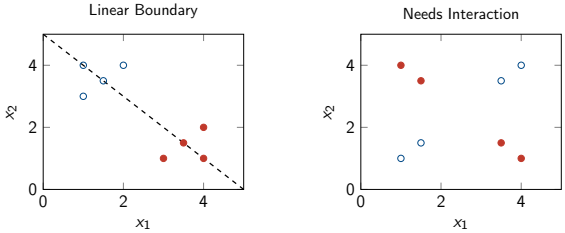
Raw	Engineered
Transaction amounts	Avg transaction per customer
Purchase dates	Days since last purchase
Product categories	Most frequent category

Interaction Features

Definition

Capture relationships between features:

$$x_{interaction} = x_i \times x_j \quad \text{or} \quad x_{interaction} = f(x_i, x_j)$$



XOR pattern requires $x_1 \times x_2$ interaction term to separate linearly

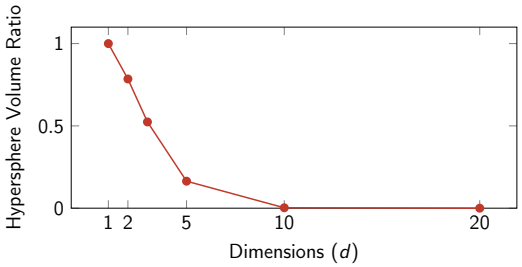
The Curse of Dimensionality

Problem

As dimensions increase, data becomes increasingly sparse:

$$\text{Volume of unit hypercube} = 1^d = 1$$

$$\text{Volume of inscribed hypersphere} = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)} \cdot \left(\frac{1}{2}\right)^d \xrightarrow{d\to\infty} 0$$



Feature Selection vs Feature Extraction

Feature Selection

Select subset of original features:

$$X' = X[:, S]$$

Where $S \subset \{1, ..., p\}$

Methods:

- Filter (correlation, variance)
- Wrapper (RFE, forward selection)
- Embedded (Lasso, tree importance)

Feature Extraction

Create new features from combinations:

$$X' = X \cdot W$$

Where $W \in \mathbb{R}^{p \times k}$, $k < p$

Methods:

- PCA
- LDA
- Autoencoders
- t-SNE, UMAP

Principal Component Analysis (PCA) Overview

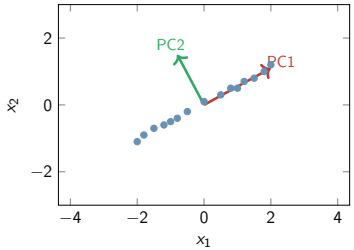
Goal

Find orthogonal directions of maximum variance:

$$z = W^T(x - \boldsymbol{\mu})$$

Where columns of W are eigenvectors of covariance matrix:

$$\boldsymbol{\Sigma} w_i = \lambda_i w_i$$



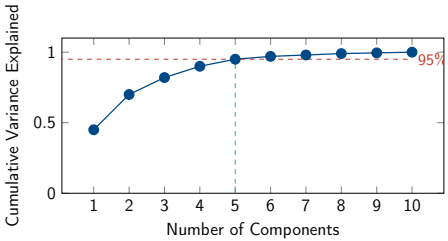
Variance Explained by PCA

Explained Variance Ratio

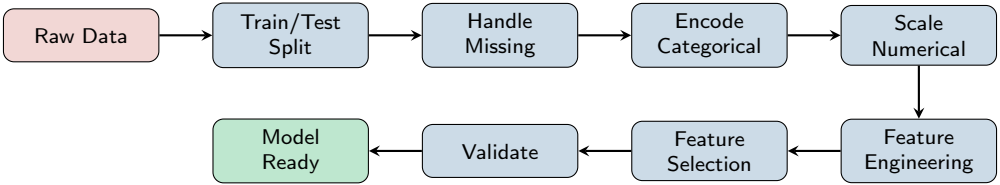
$$\text{EVR}_i = \frac{\lambda_i}{\sum_{j=1}^{p} \lambda_j}$$

Cumulative:

$$\text{CEVR}_k = \sum_{i=1}^{k} \text{EVR}_i$$



Complete Transformation Pipeline



Critical Rule: Fit on Train Only!

$$\text{scaler.fit}(X_{train}) \quad \text{then} \quad \text{scaler.transform}(X_{test})$$

Never fit transformers on test data — this causes **data leakage**!

Avoiding Data Leakage

What is Data Leakage?

Using information from test set during training, leading to overly optimistic performance estimates.

Wrong (Leakage)

1. Scale ALL data
2. Split into train/test
3. Train model

Test statistics leak into scaling!

Correct

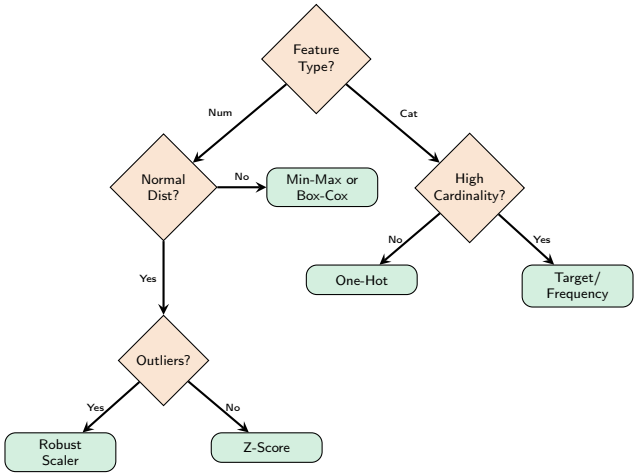
1. Split into train/test
2. Fit scaler on train
3. Transform both sets
4. Train model

Test data remains unseen!

Use Pipelines!

Sklearn Pipeline ensures correct order and prevents leakage automatically.

Transformation Decision Guide



Key Takeaways

1. **Scaling matters** for distance-based and gradient-based algorithms
2. **Choose encoding wisely** — wrong encoding creates false relationships
3. **Feature engineering** often has the biggest impact on model performance
4. **Avoid data leakage** — always fit on training data only
5. **Use pipelines** for reproducible and correct transformations
6. **Domain knowledge** is crucial for effective feature engineering

Next Part

Part 4: Data Preparation for Modeling — Train/Test Split, Cross-Validation, Handling Imbalanced Data

References

- Pedregosa et al. (2011). *Scikit-learn: Machine Learning in Python*. JMLR.
- Zheng, A. & Casari, A. (2018). *Feature Engineering for Machine Learning*. O'Reilly.
- Box, G.E.P. & Cox, D.R. (1964). *An Analysis of Transformations*. JRSS.
- Kuhn, M. & Johnson, K. (2019). *Feature Engineering and Selection*. CRC Press.
- Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly.

Part V

Part 3 : Practice Data Transformation

What We Will Cover

Practical Data Cleaning Skills

- Detecting and handling missing values
- Identifying and removing duplicates
- Detecting and treating outliers
- Correcting data types
- Cleaning text and string data
- Handling inconsistent data
- Building cleaning pipelines

Libraries Used

`pandas`, `numpy`, `scikit-learn`, `scipy`, `fuzzywuzzy`

Essential Libraries Import

```python
# Data manipulation
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Scikit-learn imputers
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Statistical functions
from scipy import stats

# String matching (install: pip install fuzzywuzzy python-Levenshtein)
from fuzzywuzzy import fuzz, process

# Display settings
pd.set_option('display.max_columns', None)
```

Detecting Missing Values

```python
# Check for missing values
print(df.isnull().sum())          # Count per column
print(df.isna().sum())            # Same as isnull()

# Total missing values
print(f"Total missing: {df.isnull().sum().sum()}")

# Percentage missing per column
missing_pct = (df.isnull().sum() / len(df) * 100).round(2)
print(missing_pct.sort_values(ascending=False))

# Rows with any missing value
rows_with_missing = df[df.isnull().any(axis=1)]
print(f"Rows with missing: {len(rows_with_missing)}")

# Columns with missing values
cols_with_missing = df.columns[df.isnull().any()].tolist()
print(f"Columns with missing: {cols_with_missing}")
```

Missing Values Summary Function

```python
def missing_summary(df):
    """Generate detailed missing values summary"""
    missing = df.isnull().sum()
    missing_pct = (missing / len(df) * 100).round(2)

    summary = pd.DataFrame({
        'Missing Count': missing,
        'Missing %': missing_pct,
        'Dtype': df.dtypes
    })

    # Filter only columns with missing values
    summary = summary[summary['Missing Count'] > 0]
    summary = summary.sort_values('Missing %', ascending=False)

    print(f"Total columns: {len(df.columns)}")
    print(f"Columns with missing: {len(summary)}")
    print(f"Total missing values: {missing.sum()}")

    return summary

print(missing_summary(df))
```

Visualizing Missing Values

```python
# Using missingno library (pip install missingno)
import missingno as msno

# Matrix visualization
msno.matrix(df, figsize=(12, 6))
plt.title('Missing Values Matrix')
plt.show()

# Bar chart of missing values
msno.bar(df, figsize=(12, 6))
plt.show()

# Heatmap of missing value correlations
msno.heatmap(df, figsize=(10, 8))
plt.show()

# Dendrogram showing missing value patterns
msno.dendrogram(df, figsize=(12, 6))
plt.show()
```

Custom Missing Values Heatmap

```python
# Create missing values heatmap with seaborn
plt.figure(figsize=(14, 8))
sns.heatmap(df.isnull(), cbar=True, yticklabels=False,
            cmap='viridis')
plt.title('Missing Values Heatmap')
plt.tight_layout()
plt.show()

# Missing values by row (identify problematic rows)
row_missing = df.isnull().sum(axis=1)
plt.figure(figsize=(12, 4))
plt.hist(row_missing, bins=50, edgecolor='black')
plt.xlabel('Number of Missing Values per Row')
plt.ylabel('Frequency')
plt.title('Distribution of Missing Values per Row')
plt.show()

# Rows with too many missing values
threshold = len(df.columns) * 0.5  # 50% missing
bad_rows = df[row_missing > threshold]
print(f"Rows with >50% missing: {len(bad_rows)}")
```

Deletion Methods

```python
# Drop rows with ANY missing value
df_clean = df.dropna()

# Drop rows with ALL missing values
df_clean = df.dropna(how='all')

# Drop rows with missing in specific columns
df_clean = df.dropna(subset=['col1', 'col2'])

# Drop rows with minimum non-null values (threshold)
df_clean = df.dropna(thresh=5)  # Keep rows with at least 5 non-null

# Drop columns with missing values
df_clean = df.dropna(axis=1)

# Drop columns with too many missing values
threshold = len(df) * 0.5  # 50% missing threshold
cols_to_drop = df.columns[df.isnull().sum() > threshold]
df_clean = df.drop(columns=cols_to_drop)
print(f"Dropped columns: {cols_to_drop.tolist()}")
```

Simple Imputation - Numerical

```python
# Fill with constant value
df['col'] = df['col'].fillna(0)

# Fill with mean
df['col'] = df['col'].fillna(df['col'].mean())

# Fill with median (robust to outliers)
df['col'] = df['col'].fillna(df['col'].median())

# Fill with mode (most frequent)
df['col'] = df['col'].fillna(df['col'].mode()[0])

# Fill multiple columns at once
numerical_cols = df.select_dtypes(include=np.number).columns
df[numerical_cols] = df[numerical_cols].fillna(
    df[numerical_cols].median()
)

# Fill with column-specific values
fill_values = {'col1': 0, 'col2': df['col2'].mean(), 'col3': -1}
df = df.fillna(fill_values)
```

Simple Imputation - Categorical

```python
# Fill with mode (most frequent category)
df['category'] = df['category'].fillna(df['category'].mode()[0])

# Fill with a specific category
df['category'] = df['category'].fillna('Unknown')
df['category'] = df['category'].fillna('Missing')

# Fill all categorical columns with mode
categorical_cols = df.select_dtypes(include=['object', 'category']).columns
for col in categorical_cols:
    df[col] = df[col].fillna(df[col].mode()[0])

# Or fill with 'Unknown'
df[categorical_cols] = df[categorical_cols].fillna('Unknown')

# Conditional filling
df['category'] = df['category'].fillna(
    df.groupby('other_col')['category'].transform(
        lambda x: x.mode()[0] if len(x.mode()) > 0 else 'Unknown'
    )
)
```

Forward and Backward Fill

```
# Forward fill (use previous value)
df['col'] = df['col'].fillna(method='ffill')
# Or using newer syntax:
df['col'] = df['col'].ffill()

# Backward fill (use next value)
df['col'] = df['col'].fillna(method='bfill')
# Or:
df['col'] = df['col'].bfill()

# Limit the fill (max consecutive NaNs to fill)
df['col'] = df['col'].ffill(limit=2)

# Combine forward and backward fill
df['col'] = df['col'].ffill().bfill()

# Useful for time series data
df = df.sort_values('date')
df['value'] = df['value'].ffill()
```

Interpolation Methods

```python
# Linear interpolation (default)
df['col'] = df['col'].interpolate(method='linear')

# Polynomial interpolation
df['col'] = df['col'].interpolate(method='polynomial', order=2)

# Spline interpolation (smooth curves)
df['col'] = df['col'].interpolate(method='spline', order=3)

# Time-based interpolation (for datetime index)
df['col'] = df['col'].interpolate(method='time')

# Index-based interpolation
df['col'] = df['col'].interpolate(method='index')

# Limit interpolation direction
df['col'] = df['col'].interpolate(limit=2, limit_direction='forward')

# Only interpolate inside (not at edges)
df['col'] = df['col'].interpolate(limit_area='inside')
```

Scikit-learn SimpleImputer

```python
from sklearn.impute import SimpleImputer

# Mean imputation
imputer_mean = SimpleImputer(strategy='mean')
df[numerical_cols] = imputer_mean.fit_transform(df[numerical_cols])

# Median imputation
imputer_median = SimpleImputer(strategy='median')
df[numerical_cols] = imputer_median.fit_transform(df[numerical_cols])

# Most frequent (mode) - works for categorical too
imputer_mode = SimpleImputer(strategy='most_frequent')
df[categorical_cols] = imputer_mode.fit_transform(df[categorical_cols])

# Constant value imputation
imputer_const = SimpleImputer(strategy='constant', fill_value=0)
df[numerical_cols] = imputer_const.fit_transform(df[numerical_cols])

# For categorical with constant
imputer_cat = SimpleImputer(strategy='constant', fill_value='Missing')
df[categorical_cols] = imputer_cat.fit_transform(df[categorical_cols])
```

KNN Imputation

```python
from sklearn.impute import KNNImputer

# KNN imputer (uses k nearest neighbors)
knn_imputer = KNNImputer(n_neighbors=5, weights='uniform')
df[numerical_cols] = knn_imputer.fit_transform(df[numerical_cols])

# Weighted by distance (closer neighbors have more influence)
knn_imputer = KNNImputer(n_neighbors=5, weights='distance')
df[numerical_cols] = knn_imputer.fit_transform(df[numerical_cols])

# Custom number of neighbors
knn_imputer = KNNImputer(n_neighbors=10)

# Note: KNNImputer only works with numerical data
# For mixed data, encode categoricals first or use separate imputers

# Scale data before KNN imputation for better results
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df_scaled = scaler.fit_transform(df[numerical_cols])
df_imputed = knn_imputer.fit_transform(df_scaled)
df[numerical_cols] = scaler.inverse_transform(df_imputed)
```

Iterative Imputer (MICE)

```python
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

# Basic iterative imputer (MICE - Multiple Imputation by Chained Equations)
iterative_imputer = IterativeImputer(max_iter=10, random_state=42)
df[numerical_cols] = iterative_imputer.fit_transform(df[numerical_cols])

# With specific estimator
from sklearn.linear_model import BayesianRidge
iterative_imputer = IterativeImputer(
    estimator=BayesianRidge(),
    max_iter=10,
    random_state=42
)
df[numerical_cols] = iterative_imputer.fit_transform(df[numerical_cols])

# With Random Forest for non-linear relationships
from sklearn.ensemble import RandomForestRegressor
iterative_imputer = IterativeImputer(
    estimator=RandomForestRegressor(n_estimators=10, random_state=42),
    max_iter=10,
    random_state=42
)
```

Group-Based Imputation

```python
# Impute based on group statistics
# Example: Fill missing salary with median salary by department

# Using groupby transform
df['salary'] = df.groupby('department')['salary'].transform(
    lambda x: x.fillna(x.median())
)

# Multiple group columns
df['salary'] = df.groupby(['department', 'level'])['salary'].transform(
    lambda x: x.fillna(x.median())
)

# Fallback to global median if group has all NaN
def group_impute(group):
    if group.isnull().all():
        return group.fillna(df['salary'].median())
    return group.fillna(group.median())

df['salary'] = df.groupby('department')['salary'].transform(group_impute)

# Custom function for complex logic
df['value'] = df.groupby('category')['value'].apply(
    lambda x: x.fillna(x.median() if x.notna().sum() > 0 else 0)
)
```

Imputation with Indicator

```python
# Create indicator column for missing values (useful for modeling)
df['col_was_missing'] = df['col'].isnull().astype(int)

# Then impute
df['col'] = df['col'].fillna(df['col'].median())

# Function to add missing indicators
def add_missing_indicators(df, columns):
    for col in columns:
        if df[col].isnull().sum() > 0:
            df[f'{col}_missing'] = df[col].isnull().astype(int)
    return df

df = add_missing_indicators(df, numerical_cols)

# Using SimpleImputer with add_indicator
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median', add_indicator=True)
result = imputer.fit_transform(df[numerical_cols])
# This returns both imputed values and indicator columns
```

Detecting Duplicates

```python
# Check for duplicate rows
print(f"Duplicate rows: {df.duplicated().sum()}")

# Check duplicates based on specific columns
print(f"Duplicate keys: {df.duplicated(subset=['id', 'date']).sum()}")

# View duplicate rows
duplicates = df[df.duplicated(keep=False)]  # keep=False shows all duplicates
print(duplicates.sort_values(by=df.columns.tolist()))

# First occurrence vs duplicates
first_occurrence = df[df.duplicated(keep='first') == False]
duplicate_rows = df[df.duplicated(keep='first') == True]

# Count duplicates per group
dup_counts = df.groupby(df.columns.tolist()).size().reset_index(name='count')
dup_counts = dup_counts[dup_counts['count'] > 1]
print(f"Groups with duplicates: {len(dup_counts)}")
```

Removing Duplicates

```python
# Remove exact duplicate rows (keep first occurrence)
df_clean = df.drop_duplicates()

# Keep last occurrence instead
df_clean = df.drop_duplicates(keep='last')

# Remove all duplicates (don't keep any)
df_clean = df.drop_duplicates(keep=False)

# Remove duplicates based on specific columns
df_clean = df.drop_duplicates(subset=['id', 'date'])

# Remove duplicates and reset index
df_clean = df.drop_duplicates().reset_index(drop=True)

# In-place removal
df.drop_duplicates(inplace=True)

# Keep row with most non-null values among duplicates
df_clean = df.loc[df.groupby(['id'])['col'].transform('count').idxmax()]
```

Handling Duplicates with Aggregation

```python
# Instead of dropping, aggregate duplicate rows
# Keep first value for some columns, mean for others

agg_rules = {
    'name': 'first',            # Keep first name
    'value': 'mean',            # Average the values
    'count': 'sum',             # Sum the counts
    'date': 'max',              # Keep latest date
    'category': lambda x: x.mode()[0]  # Most frequent
}

df_agg = df.groupby('id').agg(agg_rules).reset_index()

# Custom aggregation for complex cases
def resolve_duplicates(group):
    return pd.Series({
        'name': group['name'].iloc[0],
        'value': group['value'].mean(),
        'latest_date': group['date'].max(),
        'all_categories': ','.join(group['category'].unique())
    })

df_resolved = df.groupby('id').apply(resolve_duplicates).reset_index()
```

Fuzzy Duplicate Detection

```python
# For near-duplicates (e.g., "John Smith" vs "Jon Smith")
from fuzzywuzzy import fuzz, process

# Compare two strings
similarity = fuzz.ratio("John Smith", "Jon Smith")
print(f"Similarity: {similarity}%")  # Output: 91%

# Find potential duplicates in a column
def find_fuzzy_duplicates(df, column, threshold=85):
    duplicates = []
    values = df[column].unique()

    for i, val1 in enumerate(values):
        for val2 in values[i+1:]:
            score = fuzz.ratio(str(val1), str(val2))
            if score >= threshold:
                duplicates.append((val1, val2, score))

    return pd.DataFrame(duplicates, columns=['Value1', 'Value2', 'Score'])

fuzzy_dups = find_fuzzy_duplicates(df, 'name', threshold=85)
print(fuzzy_dups)
```

Standardizing Fuzzy Duplicates

```python
# Standardize similar values to one canonical form
from fuzzywuzzy import process

def standardize_column(df, column, threshold=85):
    """Standardize similar values to most frequent form"""
    value_counts = df[column].value_counts()
    mapping = {}

    for value in value_counts.index:
        if value not in mapping:
            # Find best match among more frequent values
            choices = [v for v in value_counts.index if v not in mapping.values()]
            match, score = process.extractOne(str(value), choices)
            if score >= threshold:
                mapping[value] = match
            else:
                mapping[value] = value

    return df[column].map(mapping)

df['name_clean'] = standardize_column(df, 'name', threshold=85)

# Check the mapping
print(df[['name', 'name_clean']].drop_duplicates())
```

IQR Method

```python
def detect_outliers_iqr(df, column, multiplier=1.5):
    """Detect outliers using IQR method"""
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1

    lower_bound = Q1 - multiplier * IQR
    upper_bound = Q3 + multiplier * IQR

    outliers = (df[column] < lower_bound) | (df[column] > upper_bound)

    return outliers, lower_bound, upper_bound

# Apply to column
outlier_mask, lower, upper = detect_outliers_iqr(df, 'value')
print(f"Outliers: {outlier_mask.sum()}")
print(f"Bounds: [{lower:.2f}, {upper:.2f}]")

# View outliers
print(df[outlier_mask])

# Apply to all numerical columns
for col in numerical_cols:
    mask, _, _ = detect_outliers_iqr(df, col)
    print(f"{col}: {mask.sum()} outliers ({mask.mean()*100:.1f}%)")
```

Z-Score Method

```python
from scipy import stats

def detect_outliers_zscore(df, column, threshold=3):
    """Detect outliers using Z-score method"""
    z_scores = np.abs(stats.zscore(df[column].dropna()))
    outliers = z_scores > threshold
    return outliers

# Apply to column
outlier_mask = detect_outliers_zscore(df, 'value', threshold=3)
print(f"Outliers (|z| > 3): {outlier_mask.sum()}")

# Calculate z-scores for all rows
df['z_score'] = np.abs(stats.zscore(df['value']))
df['is_outlier'] = df['z_score'] > 3

# Modified Z-score (more robust, uses median)
def modified_zscore(x):
    median = np.median(x)
    mad = np.median(np.abs(x - median))
    return 0.6745 * (x - median) / mad

df['modified_z'] = np.abs(modified_zscore(df['value']))
df['is_outlier_robust'] = df['modified_z'] > 3.5
```

Isolation Forest

```python
from sklearn.ensemble import IsolationForest

# Isolation Forest for multivariate outlier detection
iso_forest = IsolationForest(
    n_estimators=100,
    contamination=0.05,  # Expected proportion of outliers
    random_state=42
)

# Fit and predict (-1 for outliers, 1 for inliers)
df['outlier_if'] = iso_forest.fit_predict(df[numerical_cols])
df['is_outlier_if'] = df['outlier_if'] == -1

print(f"Outliers detected: {(df['outlier_if'] == -1).sum()}")

# Get anomaly scores (lower = more anomalous)
df['anomaly_score'] = iso_forest.decision_function(df[numerical_cols])

# Visualize
plt.figure(figsize=(10, 6))
plt.scatter(df.index, df['anomaly_score'], c=df['outlier_if'], cmap='RdYlGn')
plt.xlabel('Index')
plt.ylabel('Anomaly Score')
plt.title('Isolation Forest Anomaly Scores')
plt.show()
```

Local Outlier Factor (LOF)

```python
from sklearn.neighbors import LocalOutlierFactor

# LOF for density-based outlier detection
lof = LocalOutlierFactor(
    n_neighbors=20,
    contamination=0.05
)

# Fit and predict
df['outlier_lof'] = lof.fit_predict(df[numerical_cols])
df['is_outlier_lof'] = df['outlier_lof'] == -1

print(f"LOF outliers: {(df['outlier_lof'] == -1).sum()}")

# Get LOF scores (negative_outlier_factor_)
df['lof_score'] = -lof.negative_outlier_factor_

# Visualize 2D projection with outliers
plt.figure(figsize=(10, 8))
colors = ['red' if x == -1 else 'blue' for x in df['outlier_lof']]
plt.scatter(df['col1'], df['col2'], c=colors, alpha=0.5)
plt.xlabel('Column 1')
plt.ylabel('Column 2')
plt.title('LOF Outlier Detection')
plt.show()
```

DBSCAN for Outlier Detection

```python
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

# Scale data first
scaler = StandardScaler()
X_scaled = scaler.fit_transform(df[numerical_cols])

# DBSCAN clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
df['cluster'] = dbscan.fit_predict(X_scaled)

# Points labeled -1 are outliers (noise)
df['is_outlier_dbscan'] = df['cluster'] == -1

print(f"DBSCAN outliers: {(df['cluster'] == -1).sum()}")
print(f"Number of clusters: {df['cluster'].max() + 1}")

# Visualize
plt.figure(figsize=(10, 8))
scatter = plt.scatter(df['col1'], df['col2'], c=df['cluster'], cmap='viridis')
plt.colorbar(scatter)
plt.title('DBSCAN Clustering (-1 = Outliers)')
plt.show()
```

Removing Outliers

```python
# Remove outliers detected by IQR
outlier_mask, lower, upper = detect_outliers_iqr(df, 'value')
df_clean = df[~outlier_mask]
print(f"Removed {outlier_mask.sum()} outliers")

# Remove outliers from multiple columns
def remove_outliers_iqr(df, columns, multiplier=1.5):
    mask = pd.Series([False] * len(df))
    for col in columns:
        outliers, _, _ = detect_outliers_iqr(df, col, multiplier)
        mask = mask | outliers
    return df[~mask]

df_clean = remove_outliers_iqr(df, numerical_cols)

# Remove based on Z-score
z_scores = np.abs(stats.zscore(df[numerical_cols]))
df_clean = df[(z_scores < 3).all(axis=1)]

# Remove based on percentiles
lower = df['value'].quantile(0.01)
upper = df['value'].quantile(0.99)
df_clean = df[(df['value'] >= lower) & (df['value'] <= upper)]
```

Capping / Winsorization

```python
# Cap outliers to boundary values (Winsorization)
def cap_outliers_iqr(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - multiplier * IQR
    upper = Q3 + multiplier * IQR

    df[column] = df[column].clip(lower=lower, upper=upper)
    return df

df = cap_outliers_iqr(df, 'value')

# Cap using percentiles
def cap_percentile(df, column, lower_pct=0.01, upper_pct=0.99):
    lower = df[column].quantile(lower_pct)
    upper = df[column].quantile(upper_pct)
    df[column] = df[column].clip(lower=lower, upper=upper)
    return df

# Using scipy's winsorize
from scipy.stats.mstats import winsorize
df['value_winsorized'] = winsorize(df['value'], limits=[0.05, 0.05])
```

Transformation Methods

```python
# Log transformation (for right-skewed data)
df['value_log'] = np.log1p(df['value'])  # log(1+x) handles zeros

# Square root transformation
df['value_sqrt'] = np.sqrt(df['value'])

# Box-Cox transformation (requires positive values)
from scipy.stats import boxcox
df['value_boxcox'], lambda_param = boxcox(df['value'] + 1)
print(f"Optimal lambda: {lambda_param}")

# Yeo-Johnson transformation (handles negative values)
from sklearn.preprocessing import PowerTransformer
pt = PowerTransformer(method='yeo-johnson')
df['value_yeojohnson'] = pt.fit_transform(df[['value']])

# Compare distributions
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
df['value'].hist(ax=axes[0], bins=30)
axes[0].set_title('Original')
df['value_log'].hist(ax=axes[1], bins=30)
axes[1].set_title('Log Transform')
df['value_boxcox'].hist(ax=axes[2], bins=30)
axes[2].set_title('Box-Cox Transform')
plt.tight_layout()
```

Imputing Outliers

```python
# Replace outliers with median
def impute_outliers_median(df, column, multiplier=1.5):
    Q1 = df[column].quantile(0.25)
    Q3 = df[column].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - multiplier * IQR
    upper = Q3 + multiplier * IQR
    median = df[column].median()

    df[column] = df[column].apply(
        lambda x: median if (x < lower or x > upper) else x
    )
    return df

# Replace outliers with NaN, then impute
def outliers_to_nan(df, column, multiplier=1.5):
    mask, lower, upper = detect_outliers_iqr(df, column, multiplier)
    df.loc[mask, column] = np.nan
    return df

df = outliers_to_nan(df, 'value')
df['value'] = df['value'].fillna(df['value'].median())
```

Converting Data Types

```python
# Check current data types
print(df.dtypes)

# Convert to numeric (coerce errors to NaN)
df['col'] = pd.to_numeric(df['col'], errors='coerce')

# Convert to integer (must handle NaN first)
df['col'] = df['col'].fillna(0).astype(int)

# Nullable integer type (handles NaN)
df['col'] = df['col'].astype('Int64')  # Note: capital I

# Convert to float
df['col'] = df['col'].astype(float)

# Convert to string
df['col'] = df['col'].astype(str)

# Convert to category (memory efficient)
df['category_col'] = df['category_col'].astype('category')

# Convert to boolean
df['bool_col'] = df['bool_col'].astype(bool)
df['bool_col'] = df['col'].map({'Yes': True, 'No': False})
```

Converting Datetime

```python
# Convert to datetime
df['date'] = pd.to_datetime(df['date'])

# Specify format for faster parsing
df['date'] = pd.to_datetime(df['date'], format='%Y-%m-%d')

# Handle different formats
df['date'] = pd.to_datetime(df['date'], format='mixed', dayfirst=True)

# Handle errors
df['date'] = pd.to_datetime(df['date'], errors='coerce')  # Invalid -> NaT

# Extract components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['dayofweek'] = df['date'].dt.dayofweek
df['hour'] = df['date'].dt.hour

# Convert Unix timestamp to datetime
df['date'] = pd.to_datetime(df['timestamp'], unit='s')

# Convert datetime to string
df['date_str'] = df['date'].dt.strftime('%Y-%m-%d')
```

Automatic Type Inference

```python
# Infer better dtypes automatically
df = df.infer_objects()

# Convert object columns to best possible types
df = df.convert_dtypes()

# Custom type conversion function
def optimize_dtypes(df):
    for col in df.columns:
        col_type = df[col].dtype

        if col_type == 'object':
            # Try to convert to numeric
            try:
                df[col] = pd.to_numeric(df[col])
            except:
                # Try to convert to datetime
                try:
                    df[col] = pd.to_datetime(df[col])
                except:
                    # Convert to category if low cardinality
                    if df[col].nunique() / len(df) < 0.5:
                        df[col] = df[col].astype('category')
    return df

df = optimize_dtypes(df)
```

Basic String Cleaning

```python
# Remove leading/trailing whitespace
df['col'] = df['col'].str.strip()

# Remove extra whitespace
df['col'] = df['col'].str.replace(r'\s+', ' ', regex=True)

# Convert to lowercase/uppercase
df['col'] = df['col'].str.lower()
df['col'] = df['col'].str.upper()
df['col'] = df['col'].str.title()  # Title Case

# Remove specific characters
df['col'] = df['col'].str.replace('[^\w\s]', '', regex=True)  # Keep only alphanumeric

# Remove digits
df['col'] = df['col'].str.replace(r'\d+', '', regex=True)

# Remove specific strings
df['col'] = df['col'].str.replace('unwanted', '', regex=False)

# Replace multiple values
df['col'] = df['col'].replace({'old1': 'new1', 'old2': 'new2'})
```

Advanced String Operations

```python
# Extract patterns with regex
df['phone'] = df['text'].str.extract(r'(\d{3}-\d{3}-\d{4})')
df['email'] = df['text'].str.extract(r'([\w.-]+@[\w.-]+)')

# Split strings
df[['first_name', 'last_name']] = df['full_name'].str.split(' ', n=1, expand=True)

# Check for patterns
df['has_email'] = df['text'].str.contains(r'@', regex=True)
df['starts_with_a'] = df['col'].str.startswith('A')

# Pad strings
df['code'] = df['code'].str.zfill(5)  # Pad with zeros: '42' -> '00042'

# Slice strings
df['first_3'] = df['col'].str[:3]
df['last_4'] = df['col'].str[-4:]

# Get string length
df['length'] = df['col'].str.len()

# Count occurrences
df['word_count'] = df['text'].str.count(r'\w+')
```

Standardizing Categorical Values

```python
# Map inconsistent values to standard forms
mapping = {
    'yes': 'Yes', 'YES': 'Yes', 'y': 'Yes', 'Y': 'Yes', '1': 'Yes',
    'no': 'No', 'NO': 'No', 'n': 'No', 'N': 'No', '0': 'No'
}
df['response'] = df['response'].map(mapping).fillna(df['response'])

# Using replace with regex
df['col'] = df['col'].replace({
    r'^[Yy]es.*': 'Yes',
    r'^[Nn]o.*': 'No'
}, regex=True)

# Standardize case and strip
def standardize_text(text):
    if pd.isna(text):
        return text
    return str(text).strip().lower().title()

df['name'] = df['name'].apply(standardize_text)

# Fix common typos
typo_fixes = {'teh': 'the', 'recieve': 'receive', 'seperate': 'separate'}
for typo, fix in typo_fixes.items():
    df['text'] = df['text'].str.replace(typo, fix, case=False)
```

Complete Cleaning Function

```python
def clean_dataframe(df):
    """Comprehensive data cleaning pipeline"""
    df = df.copy()

    # 1. Remove duplicate rows
    initial_rows = len(df)
    df = df.drop_duplicates()
    print(f"Removed {initial_rows - len(df)} duplicate rows")

    # 2. Handle column names
    df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

    # 3. Identify column types
    num_cols = df.select_dtypes(include=np.number).columns.tolist()
    cat_cols = df.select_dtypes(include='object').columns.tolist()

    # 4. Clean string columns
    for col in cat_cols:
        df[col] = df[col].str.strip()

    # 5. Handle missing values
    # Numerical: median imputation
    for col in num_cols:
        df[col] = df[col].fillna(df[col].median())

    return df
```

Sklearn Pipeline for Cleaning

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Define preprocessing for different column types
numerical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

categorical_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='Missing')),
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

# Combine pipelines
preprocessor = ColumnTransformer([
    ('num', numerical_pipeline, numerical_cols),
    ('cat', categorical_pipeline, categorical_cols)
])

# Fit and transform
X_clean = preprocessor.fit_transform(df)
```

Custom Transformer

```python
from sklearn.base import BaseEstimator, TransformerMixin

class OutlierRemover(BaseEstimator, TransformerMixin):
    def __init__(self, multiplier=1.5):
        self.multiplier = multiplier
        self.bounds_ = {}

    def fit(self, X, y=None):
        for col in X.columns:
            Q1, Q3 = X[col].quantile([0.25, 0.75])
            IQR = Q3 - Q1
            self.bounds_[col] = (Q1 - self.multiplier * IQR,
                                 Q3 + self.multiplier * IQR)
        return self

    def transform(self, X):
        X = X.copy()
        for col in X.columns:
            lower, upper = self.bounds_[col]
            X[col] = X[col].clip(lower=lower, upper=upper)
        return X

# Use in pipeline
pipeline = Pipeline([
    ('outlier', OutlierRemover(multiplier=1.5)),
    ('imputer', SimpleImputer(strategy='median'))
])
```

Saving and Loading Clean Data

```python
# Save cleaned data
df_clean.to_csv('data_cleaned.csv', index=False)
df_clean.to_parquet('data_cleaned.parquet', index=False)  # Better for large files
df_clean.to_pickle('data_cleaned.pkl')  # Preserves dtypes

# Save with compression
df_clean.to_csv('data_cleaned.csv.gz', compression='gzip', index=False)

# Load cleaned data
df = pd.read_csv('data_cleaned.csv')
df = pd.read_parquet('data_cleaned.parquet')
df = pd.read_pickle('data_cleaned.pkl')

# Save preprocessing pipeline (for production)
import joblib
joblib.dump(preprocessor, 'preprocessor.pkl')

# Load and apply to new data
preprocessor = joblib.load('preprocessor.pkl')
X_new_clean = preprocessor.transform(df_new)
```

Data Cleaning Checklist

Missing Values:

- ☐ Identify missing patterns
- ☐ Decide: delete or impute
- ☐ Apply appropriate imputation
- ☐ Add missing indicators if needed

Duplicates:

- ☐ Check exact duplicates
- ☐ Check key-based duplicates
- ☐ Check fuzzy duplicates
- ☐ Remove or aggregate

Outliers:

- ☐ Detect with IQR/Z-score
- ☐ Verify if real or errors
- ☐ Remove, cap, or transform

Data Types & Text:

- ☐ Fix incorrect types
- ☐ Parse dates properly
- ☐ Clean string columns
- ☐ Standardize categories

Key Python Functions Reference

Task	Function
Detect missing	`df.isnull().sum()`, `msno.matrix()`
Drop missing	`df.dropna()`, `df.dropna(subset=[...])`
Fill missing	`df.fillna()`, `df.interpolate()`
Impute (sklearn)	`SimpleImputer`, `KNNImputer`
Detect duplicates	`df.duplicated()`, `fuzz.ratio()`
Remove duplicates	`df.drop_duplicates()`
Detect outliers	`zscore()`, `IsolationForest`
Cap outliers	`df.clip()`, `winsorize()`
Convert types	`astype()`, `pd.to_numeric()`
Clean strings	`str.strip()`, `str.replace()`

Questions & Practice

Next: Part 3 - Data Transformation with Python

Part VI

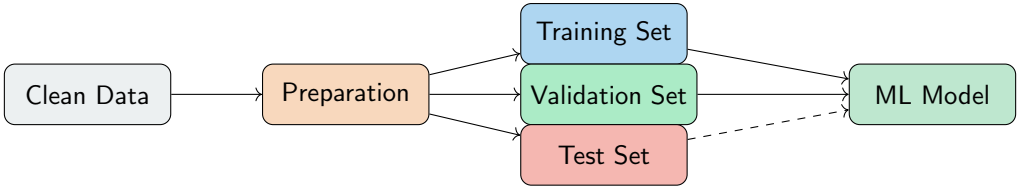
Part 4: Preparation

"Ready your data"

The Final Step: Data Preparation

Definition

Data Preparation for Modeling is the final preprocessing phase where clean, transformed data is organized into appropriate subsets for training, validation, and testing machine learning models.



Why Proper Data Preparation Matters

Without Proper Preparation

- **Data Leakage:** Test info leaks into training
- **Overfitting:** Model memorizes noise
- **Biased Evaluation:** Unrealistic performance
- **Poor Generalization:** Fails on new data

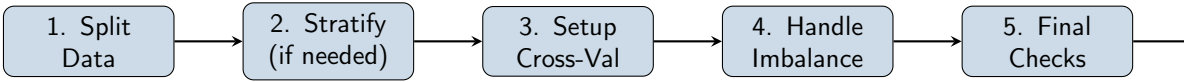
With Proper Preparation

- **Honest Evaluation:** True performance estimate
- **Robust Models:** Generalizes well
- **Reliable Comparisons:** Fair model selection
- **Production Ready:** Works on real data

Key Principle

The test set must simulate truly unseen data — never use it for any decision during model development!

Data Preparation Workflow



Topics Covered

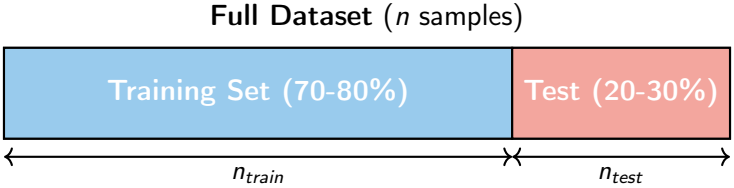
1. Train/Test/Validation Splitting
2. Cross-Validation Techniques
3. Stratified Sampling
4. Handling Imbalanced Datasets
5. Time Series Considerations
6. Final Validation Checklist

The Fundamental Split

Purpose

Divide data into separate sets to:

- **Train:** Learn patterns from data
- **Test:** Evaluate generalization to unseen data



Mathematical Notation

$$\mathcal{D} = \mathcal{D}_{train} \cup \mathcal{D}_{test}, \quad \mathcal{D}_{train} \cap \mathcal{D}_{test} = \emptyset$$

Common Split Ratios

Dataset Size	Train	Validation	Test
Small ($< 1,000$)	60%	20%	20%
Medium ($1,000 - 100,000$)	70%	15%	15%
Large ($100,000 - 1M$)	80%	10%	10%
Very Large ($> 1M$)	98%	1%	1%

Two-Way Split

$$\mathcal{D} \rightarrow \mathcal{D}_{train}, \mathcal{D}_{test}$$

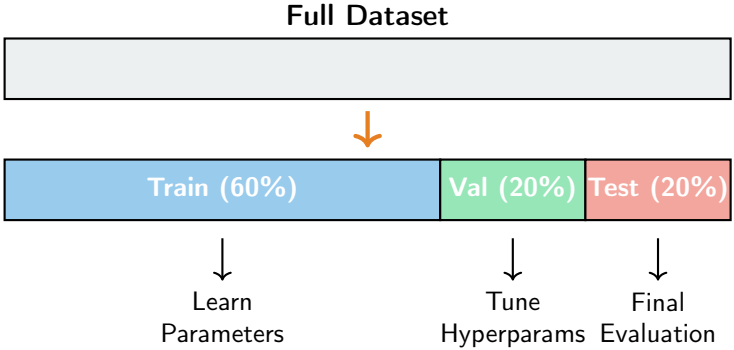
Use cross-validation on train set

Three-Way Split

$$\mathcal{D} \rightarrow \mathcal{D}_{train}, \mathcal{D}_{val}, \mathcal{D}_{test}$$

Explicit validation set

Train-Validation-Test Split



Role of Each Set

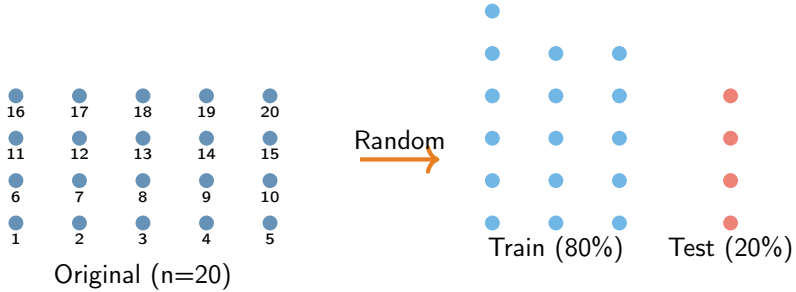
- **Training:** Fit model parameters (θ)
- **Validation:** Tune hyperparameters (λ, learning rate, etc.)

Random Sampling: The Basics

Simple Random Split

Each sample has equal probability of being in test set:

$$P(x_i \in \mathcal{D}_{test}) = \frac{n_{test}}{n} = 1 - \frac{n_{train}}{n}$$



Importance of Random State (Reproducibility)

Setting Random Seed

$$\text{random\_state} = s \implies \text{Same split every time}$$

Without Fixed Seed

- Different splits each run
- Non-reproducible results
- Can't compare experiments
- Debugging nightmare

With Fixed Seed

- Reproducible experiments
- Fair model comparisons
- Easier debugging
- Scientific rigor

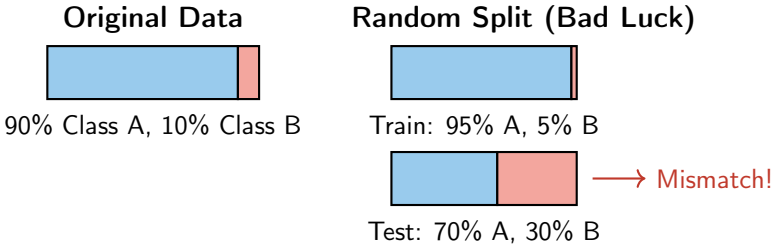
Best Practice

Always set `random_state=42` (or any fixed integer) and document it!

The Problem with Random Sampling

Class Imbalance Issue

Random sampling may not preserve class distribution, especially with imbalanced data!



Consequence

Model trained on different distribution than it's tested on → unreliable evaluation!

Stratified Sampling: The Solution

Definition

Stratified sampling ensures each split has the same proportion of classes as the original dataset:

$$\frac{n_{c,train}}{n_{train}} = \frac{n_{c,test}}{n_{test}} = \frac{n_c}{n} \quad \forall \text{ class } c$$

Original (90%/10%) **Train (90%/10%)** **Test (90%/10%)**

Result

Both train and test maintain the original 90%/10% class ratio!

When to Use Stratified Sampling

Always Stratify When:

- Classification problems (especially imbalanced)
- Small datasets (random variation is high)
- Multi-class problems
- Target variable has skewed distribution

Stratification for Regression

For continuous targets, bin the target into quantiles and stratify:

$$y \rightarrow \text{bins}(y) = \{Q_1, Q_2, Q_3, Q_4\}$$

Then stratify on bins to ensure similar target distributions.

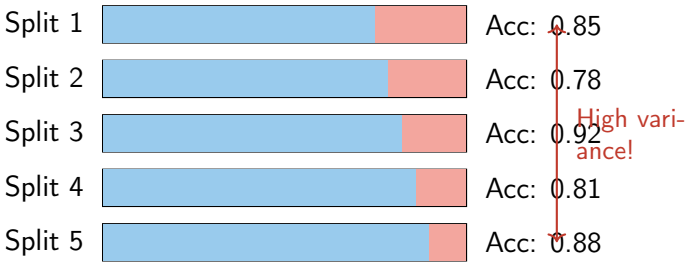
Multi-Label Stratification

For multi-label problems, use iterative stratification algorithms (more complex).

Why Cross-Validation?

Problem with Single Split

A single train/test split may not be representative — performance depends on which samples end up in each set.



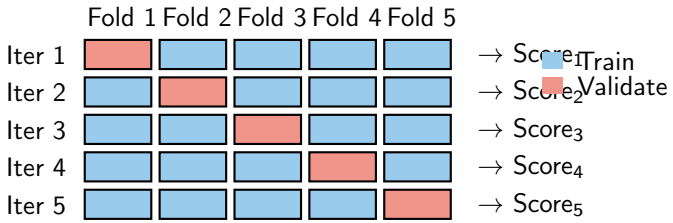
Solution

Cross-validation: Use multiple splits and average the results for a robust estimate.

K-Fold Cross-Validation

Algorithm

1. Divide data into K equal-sized folds
2. For each fold $k = 1, ..., K$:
   - Use fold k as validation set
   - Use remaining $K - 1$ folds as training set
   - Train model and record performance
3. Average performance across all K folds



K-Fold: Mathematical Formulation

Performance Estimation

Let $\mathcal{L}(\hat{f}^{(-k)}, \mathcal{D}_k)$ be the loss on fold k using model trained without fold k:

$$\text{CV}(K) = \frac{1}{K} \sum_{k=1}^{K} \mathcal{L}(\hat{f}^{(-k)}, \mathcal{D}_k)$$

Variance of CV Estimate

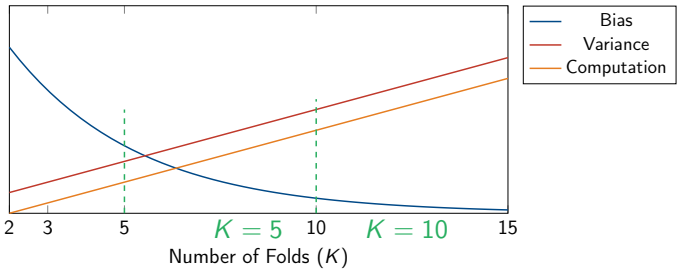
$$\text{Var}[\text{CV}(K)] = \frac{1}{K}\sigma^2 + \frac{K-1}{K}\rho\sigma^2$$

Where:

- σ^2 = variance of individual fold estimates
- ρ = correlation between fold estimates (due to overlapping training sets)

Typical Values

Choosing K: Trade-offs



Common Choices

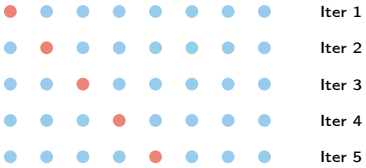
- $K = 5$: Fast, good for large datasets
- $K = 10$: Standard choice, good balance
- $K = n$ (LOOCV): Minimum bias, maximum variance, expensive

Leave-One-Out Cross-Validation (LOOCV)

Definition

Special case where $K = n$ (each sample is its own fold):

$$\text{LOOCV} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{f}^{(-i)}, (x_i, y_i))$$



...

n iterations total

Pros:

Cons:

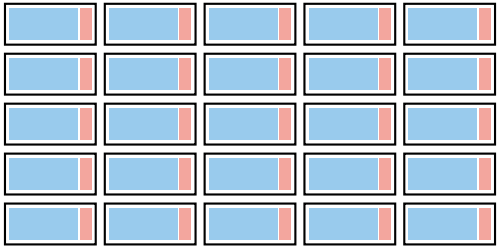
Stratified K-Fold Cross-Validation

Definition

K-Fold CV that preserves class distribution in each fold:

$$\frac{n_{c,k}}{n_k} \approx \frac{n_c}{n} \quad \forall \text{ fold } k, \text{ class } c$$

Each fold maintains 80%/20% ratio



Always Use Stratified K-Fold for Classification!

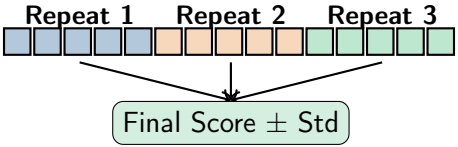
Repeated K-Fold Cross-Validation

Definition

Repeat K-Fold CV multiple times with different random splits:

$$\text{Repeated-CV} = \frac{1}{R \cdot K} \sum_{r=1}^{R} \sum_{k=1}^{K} \mathcal{L}(\hat{f}^{(r,-k)}, \mathcal{D}_k^{(r)})$$

Where R = number of repetitions



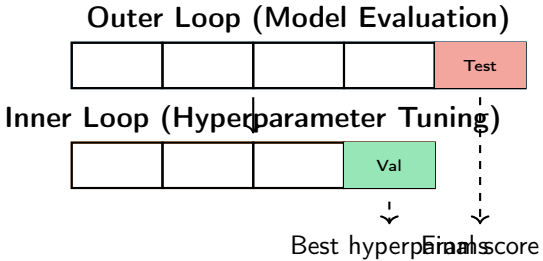
Common Configuration

5-Fold repeated 10 times = 50 total model fits → robust estimate!

Nested Cross-Validation

Purpose

Unbiased evaluation when both model selection AND hyperparameter tuning are needed.



Process

- **Inner loop:** Tune hyperparameters using CV on training folds
- **Outer loop:** Evaluate best model on held-out test fold

Cross-Validation Methods Comparison

Method	Bias	Variance	Cost	Best For
Hold-out	High	High	Low	Quick testing
5-Fold CV	Medium	Medium	Medium	Large datasets
10-Fold CV	Low	Medium	Medium	Standard choice
LOOCV	Very Low	High	High	Small datasets
Stratified K-Fold	Low	Medium	Medium	Classification
Repeated K-Fold	Very Low	Low	High	Final evaluation
Nested CV	Very Low	Low	Very High	Model selection

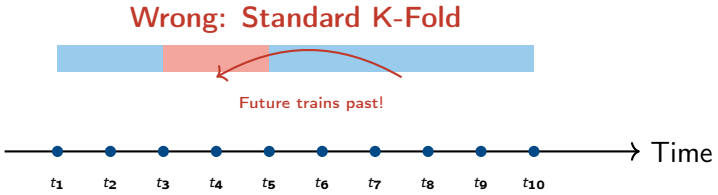
Recommendation

- Default: Stratified 10-Fold CV
- Final reporting: Repeated Stratified K-Fold
- With hyperparameter tuning: Nested CV

Why Standard CV Fails for Time Series

Temporal Leakage Problem

Standard K-Fold uses future data to predict the past — this is **data leakage**!



Key Principle

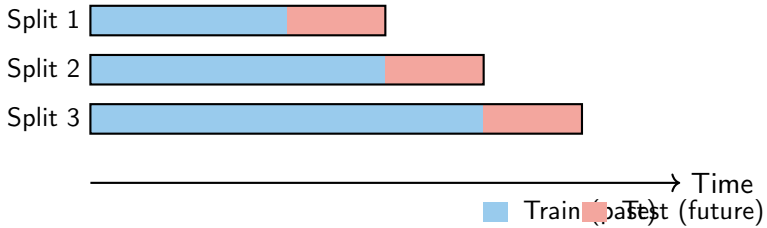
In time series: **Always train on past, test on future!**

$$t_{train} < t_{test} \quad \text{(strictly)}$$

Time Series Split (Walk-Forward Validation)

Definition

Expanding window where training set grows over time:



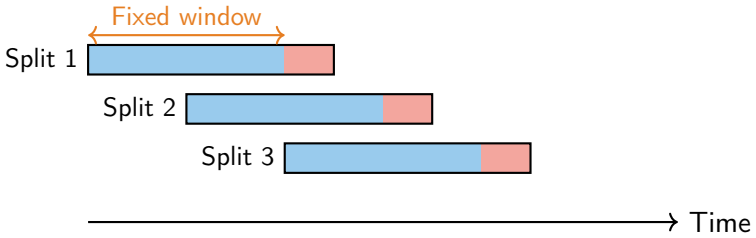
Properties

- Training set always precedes test set
- Training set grows with each split
- Mimics real-world forecasting scenario

Sliding Window Cross-Validation

Definition

Fixed-size training window that slides through time:



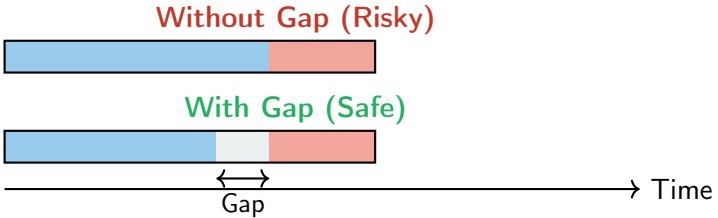
When to Use

- When recent data is more relevant than old data
- Non-stationary time series
- Concept drift scenarios

Gap Between Train and Test

Forecast Horizon Gap

Add gap to prevent information leakage from lag features:



Gap Size

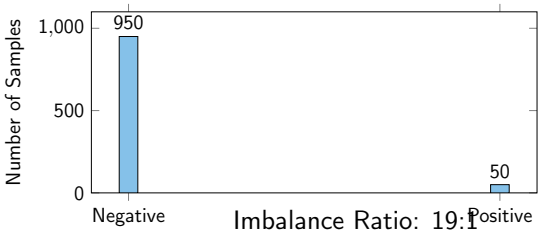
$$\text{Gap} \geq \max(\text{lag features}, \text{forecast horizon})$$

The Class Imbalance Problem

Definition

A dataset where classes have significantly different frequencies:

$$\text{Imbalance Ratio} = \frac{n_{majority}}{n_{minority}} >> 1$$



Real-World Examples

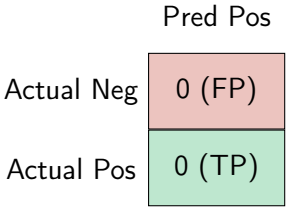
Fraud detection (0.1%), disease diagnosis (1-5%), spam (10-20%)

Why Imbalance is Problematic

Model Behavior on Imbalanced Data

A model can achieve 95% accuracy by simply predicting the majority class!

$$\text{Accuracy} = \frac{950 + 0}{950 + 50} = 95\% \quad \text{(useless model!)}$$



Precision = 0/0 = undefined
Recall = 0/50 = 0%
F1 = 0
Acc = 95% but useless!

Better Metrics for Imbalanced Data

Precision, Recall, F1-Score

$$\text{Precision} = \frac{TP}{TP + FP}, \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2TP}{2TP + FP + FN}$$

F-beta Score (Weighted)

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{\beta^2 \cdot \text{Precision} + \text{Recall}}$$

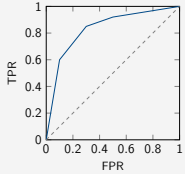
- $\beta = 1$: Equal weight (standard F1)
- $\beta = 2$: Recall is 2x more important than precision
- $\beta = 0.5$: Precision is 2x more important than recall

ROC-AUC and PR-AUC

ROC Curve

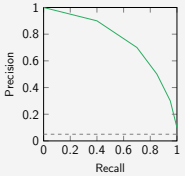
$$\text{TPR} = \frac{TP}{TP + FN}$$

$$\text{FPR} = \frac{FP}{FP + TN}$$



Precision-Recall Curve

Better for imbalanced data!



Baseline $= \frac{n_{pos}}{n}$ (class ratio)

Resampling: Oversampling Techniques

Random Oversampling

Duplicate minority class samples randomly:

$$n'_{minority} = n_{majority} \quad \text{or} \quad n'_{minority} = k \cdot n_{minority}$$

SMOTE (Synthetic Minority Oversampling Technique)

Generate synthetic samples along line segments between minority samples:

$$x_{new} = x_i + \lambda \cdot (x_{nn} - x_i), \quad \lambda \in [0, 1]$$

Where x_{nn} is a nearest neighbor of x_i

SMOTE Variants

SMOTE Variants for Different Scenarios

- **Borderline-SMOTE:** Only oversample near decision boundary
- **ADASYN:** Generate more samples where density is low
- **SMOTE-NC:** Handles mixed numerical and categorical features
- **SVMSMOTE:** Uses SVM to identify support vectors

Borderline-SMOTE focuses here

Resampling: Undersampling Techniques

Random Undersampling

Remove majority class samples randomly:

$$n'_{majority} = n_{minority} \quad \text{or} \quad n'_{majority} = k \cdot n_{minority}$$

Risk: May lose important information!

Informed Undersampling

- **Tomek Links:** Remove majority samples that form Tomek links
- **ENN (Edited Nearest Neighbors):** Remove samples misclassified by KNN
- **NearMiss:** Keep majority samples closest to minority
- **Cluster Centroids:** Replace majority samples with cluster centers

Tomek Link

Pair (x_i, x_j) where they are each other's nearest neighbor but different classes.

SMOTEENN

Combine SMOTE oversampling with ENN cleaning:

$$\mathcal{D}' = \text{ENN}(\text{SMOTE}(\mathcal{D}))$$

SMOTETomek

Combine SMOTE with Tomek links removal:

$$\mathcal{D}' = \text{Tomek}(\text{SMOTE}(\mathcal{D}))$$

Class Weights

Cost-Sensitive Learning

Assign higher weight to minority class in loss function:

$$\mathcal{L}_{weighted} = \sum_{i=1}^{n} w_{y_i} \cdot \ell(y_i, \hat{y}_i)$$

Where weights are typically:

$$w_c = \frac{n}{k \cdot n_c}$$

n = total samples, k = number of classes, n_c = samples in class c

Example

- 950 negative, 50 positive samples
- $w_{neg} = \frac{1000}{2 \times 950} = 0.53$
- $w_{pos} = \frac{1000}{2 \times 50} = 10.0$

Threshold Adjustment

Default Threshold

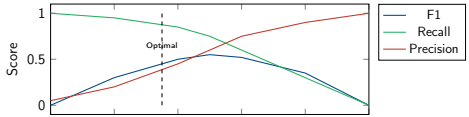
Most classifiers use threshold $\tau = 0.5$:

$$\hat{y} = \begin{cases} 1 & \text{if } P(y = 1|x) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

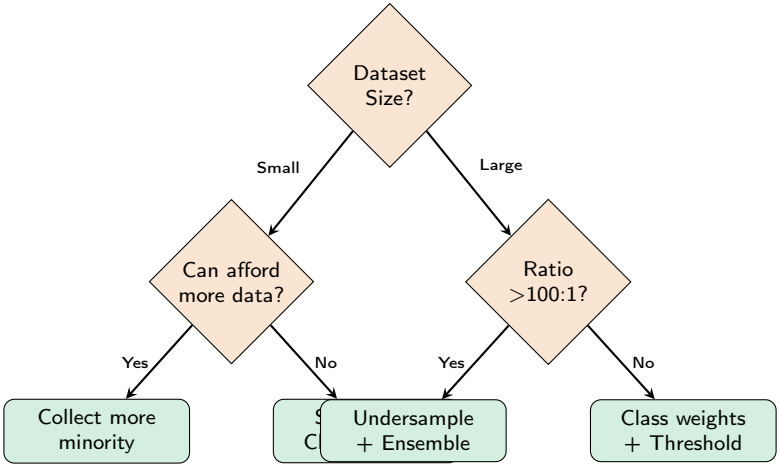
Adjusting Threshold for Imbalance

Lower threshold to increase recall for minority class:

$$\hat{y} = \begin{cases} 1 & \text{if } P(y = 1|x) \geq \tau \\ 0 & \text{otherwise} \end{cases} \quad \text{where } \tau < 0.5$$



Imbalanced Data: Strategy Summary



Important Reminder

Apply resampling ONLY to training data! Test set must remain untouched to reflect

Pre-Modeling Validation Checklist

Data Quality:

- ☐ No missing values (or handled)
- ☐ No duplicate records
- ☐ Outliers addressed
- ☐ Data types correct
- ☐ No invalid values

Transformations:

- ☐ Numerical features scaled
- ☐ Categorical features encoded
- ☐ Features engineered
- ☐ Transformations fitted on train only

Splitting:

- ☐ Train/val/test split done
- ☐ Stratification applied (if needed)
- ☐ No data leakage
- ☐ Random state documented

Imbalance:

- ☐ Class distribution checked
- ☐ Appropriate metrics chosen
- ☐ Resampling applied (train only)
- ☐ Class weights considered

Data Leakage: Final Check

Common Sources of Data Leakage

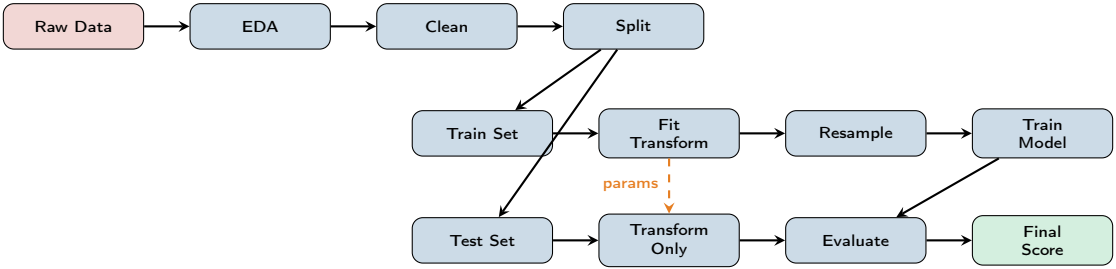
1. **Preprocessing on full data:** Scaling, imputation, encoding
2. **Feature selection on full data:** Using test info to select features
3. **Time series mistakes:** Using future data to predict past
4. **Duplicate records:** Same record in train and test
5. **Target leakage:** Features that contain target information

Prevention Strategy

$$\text{Pipeline} = \text{Preprocessing} \rightarrow \text{Model}$$

Use sklearn Pipeline to ensure all transformations are fit only on training data!

Complete ML Preparation Pipeline



Key Points

- EDA and cleaning on full data (before split)
- All transformers fitted on training data only
- Resampling applied to training data only
- Test set touched only for final evaluation

Key Takeaways

1. **Split first, transform second** — avoid data leakage
2. **Stratify for classification** — preserve class distribution
3. **Use cross-validation** — robust performance estimates
4. **Choose appropriate metrics** — accuracy is not always best
5. **Handle imbalance properly** — resample training data only
6. **Time series is special** — always respect temporal order
7. **Document everything** — reproducibility is key

Course Complete!

You've now covered all four parts of ML Pre-Processing:

1. EDA ✓
2. Data Cleaning ✓
3. Data Transformation ✓
4. Data Preparation ✓

Part VII

Part 4: Practice Preparation

"Ready your data"

What We Will Cover

Practical Data Preparation Skills

- Train-test splitting strategies
- Stratified sampling techniques
- Cross-validation methods (K-Fold, Stratified, LOOCV)
- Time series splitting
- Handling imbalanced datasets (SMOTE, undersampling)
- Evaluation metrics for imbalanced data
- Complete preprocessing pipelines
- Avoiding data leakage

Libraries Used

`scikit-learn`, `imbalanced-learn`, `pandas`, `numpy`

Essential Libraries Import

```python
# Data manipulation
import pandas as pd
import numpy as np

# Train-test splitting
from sklearn.model_selection import (
    train_test_split, KFold, StratifiedKFold, LeaveOneOut,
    RepeatedKFold, RepeatedStratifiedKFold, TimeSeriesSplit,
    cross_val_score, cross_validate, GridSearchCV
)

# Imbalanced data handling (pip install imbalanced-learn)
from imblearn.over_sampling import SMOTE, ADASYN, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler, TomekLinks
from imblearn.combine import SMOTETomek, SMOTEENN
from imblearn.pipeline import Pipeline as ImbPipeline

# Metrics
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    classification_report, confusion_matrix, roc_auc_score
)
```

Basic Train-Test Split

```python
from sklearn.model_selection import train_test_split

# Basic split (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42
)

print(f"Training set: {X_train.shape[0]} samples")
print(f"Test set: {X_test.shape[0]} samples")

# Alternative: specify train_size
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    train_size=0.8,
    random_state=42
)

# Shuffle data (default is True)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    shuffle=True,
    random_state=42
)
```

Train-Validation-Test Split

```python
# Three-way split: 60% train, 20% validation, 20% test

# Method 1: Two sequential splits
X_train_full, X_test, y_train_full, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
X_train, X_val, y_train, y_val = train_test_split(
    X_train_full, y_train_full, test_size=0.25, random_state=42  # 0.25 * 0.8 = 0.2
)

print(f"Train: {len(X_train)} ({len(X_train)/len(X)*100:.0f}%)")
print(f"Val: {len(X_val)} ({len(X_val)/len(X)*100:.0f}%)")
print(f"Test: {len(X_test)} ({len(X_test)/len(X)*100:.0f}%)")

# Method 2: Using numpy split
indices = np.arange(len(X))
np.random.seed(42)
np.random.shuffle(indices)
train_idx = indices[:int(0.6*len(X))]
val_idx = indices[int(0.6*len(X)):int(0.8*len(X))]
test_idx = indices[int(0.8*len(X)):]

X_train, X_val, X_test = X[train_idx], X[val_idx], X[test_idx]
y_train, y_val, y_test = y[train_idx], y[val_idx], y[test_idx]
```

Stratified Split (Classification)

```python
# Stratified split: preserves class distribution
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,          # Key parameter!
    random_state=42
)

# Verify class distribution is preserved
print("Original distribution:")
print(pd.Series(y).value_counts(normalize=True))

print("\nTraining distribution:")
print(pd.Series(y_train).value_counts(normalize=True))

print("\nTest distribution:")
print(pd.Series(y_test).value_counts(normalize=True))

# Stratified three-way split
X_temp, X_test, y_temp, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.25, stratify=y_temp, random_state=42
)
```

Stratified Split for Regression

```python
# For regression: stratify by binned target

# Create bins from continuous target
n_bins = 5
y_binned = pd.qcut(y, q=n_bins, labels=False, duplicates='drop')

# Stratified split using binned target
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y_binned,
    random_state=42
)

# Verify distribution
print("Original target stats:")
print(f"Mean: {y.mean():.2f}, Std: {y.std():.2f}")

print("\nTrain target stats:")
print(f"Mean: {y_train.mean():.2f}, Std: {y_train.std():.2f}")

print("\nTest target stats:")
print(f"Mean: {y_test.mean():.2f}, Std: {y_test.std():.2f}")
```

Group-Based Split

```python
from sklearn.model_selection import GroupShuffleSplit

# When data has groups that should stay together
# Example: multiple samples from same patient, same customer, etc.

groups = df['patient_id'].values  # Group identifier

# Group shuffle split
gss = GroupShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
train_idx, test_idx = next(gss.split(X, y, groups))

X_train, X_test = X[train_idx], X[test_idx]
y_train, y_test = y[train_idx], y[test_idx]

# Verify no group appears in both sets
train_groups = set(groups[train_idx])
test_groups = set(groups[test_idx])
print(f"Groups in train: {len(train_groups)}")
print(f"Groups in test: {len(test_groups)}")
print(f"Overlap: {len(train_groups & test_groups)}")  # Should be 0

# Alternative: GroupKFold for cross-validation
from sklearn.model_selection import GroupKFold
gkf = GroupKFold(n_splits=5)
for train_idx, test_idx in gkf.split(X, y, groups):
    # Each fold has different groups
    pass
```

K-Fold Cross-Validation

```python
from sklearn.model_selection import KFold, cross_val_score
from sklearn.ensemble import RandomForestClassifier

# Create K-Fold splitter
kfold = KFold(n_splits=5, shuffle=True, random_state=42)

# Method 1: Manual iteration
model = RandomForestClassifier(random_state=42)
scores = []
for fold, (train_idx, val_idx) in enumerate(kfold.split(X)):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

    model.fit(X_train, y_train)
    score = model.score(X_val, y_val)
    scores.append(score)
    print(f"Fold {fold+1}: {score:.4f}")

print(f"\nMean: {np.mean(scores):.4f} (+/- {np.std(scores)*2:.4f})")

# Method 2: Using cross_val_score (simpler)
scores = cross_val_score(model, X, y, cv=5, scoring='accuracy')
print(f"CV Accuracy: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")
```

Stratified K-Fold Cross-Validation

```python
from sklearn.model_selection import StratifiedKFold, cross_val_score

# Stratified K-Fold: preserves class distribution in each fold
skfold = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Manual iteration
for fold, (train_idx, val_idx) in enumerate(skfold.split(X, y)):
    y_train_fold = y[train_idx]
    y_val_fold = y[val_idx]

    # Check distribution
    train_dist = pd.Series(y_train_fold).value_counts(normalize=True)
    val_dist = pd.Series(y_val_fold).value_counts(normalize=True)
    print(f"Fold {fold+1} - Train: {dict(train_dist)}, Val: {dict(val_dist)}")

# Using cross_val_score with stratified CV
model = RandomForestClassifier(random_state=42)
scores = cross_val_score(model, X, y, cv=skfold, scoring='accuracy')
print(f"\nStratified CV Accuracy: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")

# Shortcut: cv=5 with classifier automatically uses StratifiedKFold
scores = cross_val_score(model, X, y, cv=5)  # Auto-stratified for classifiers
```

Leave-One-Out Cross-Validation (LOOCV)

```python
from sklearn.model_selection import LeaveOneOut, cross_val_score

# LOOCV: each sample is used once as validation
loo = LeaveOneOut()

print(f"Number of splits: {loo.get_n_splits(X)}")  # = n_samples

# Using cross_val_score (can be slow for large datasets!)
model = RandomForestClassifier(n_estimators=10, random_state=42)
scores = cross_val_score(model, X, y, cv=loo)

print(f"LOOCV Accuracy: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")

# Manual iteration (for more control)
predictions = []
for train_idx, val_idx in loo.split(X):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

    model.fit(X_train, y_train)
    pred = model.predict(X_val)
    predictions.append(pred[0])

accuracy = np.mean(np.array(predictions) == y)
print(f"LOOCV Accuracy (manual): {accuracy:.4f}")
```

Repeated K-Fold Cross-Validation

```python
from sklearn.model_selection import RepeatedKFold, RepeatedStratifiedKFold

# Repeated K-Fold: run K-Fold multiple times with different shuffles
rkfold = RepeatedKFold(n_splits=5, n_repeats=10, random_state=42)

print(f"Total iterations: {rkfold.get_n_splits(X)}")  # 5 * 10 = 50

model = RandomForestClassifier(random_state=42)
scores = cross_val_score(model, X, y, cv=rkfold, scoring='accuracy')

print(f"Repeated CV: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")

# Repeated Stratified K-Fold (recommended for classification)
rskfold = RepeatedStratifiedKFold(n_splits=5, n_repeats=10, random_state=42)

scores = cross_val_score(model, X, y, cv=rskfold, scoring='accuracy')
print(f"Repeated Stratified CV: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")

# Detailed results per fold
for i, score in enumerate(scores):
    repeat = i // 5 + 1
    fold = i % 5 + 1
    print(f"Repeat {repeat}, Fold {fold}: {score:.4f}")
```

Cross-Validation with Multiple Metrics

```python
from sklearn.model_selection import cross_validate

# Get multiple metrics at once
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro', 'roc_auc_ovr']

model = RandomForestClassifier(random_state=42)
cv_results = cross_validate(
    model, X, y,
    cv=5,
    scoring=scoring,
    return_train_score=True
)

# Print results
for metric in scoring:
    train_key = f'train_{metric}'
    test_key = f'test_{metric}'
    print(f"\n{metric}:")
    print(f"  Train: {cv_results[train_key].mean():.4f} (+/- {cv_results[train_key].std()*2:.4f})")
    print(f"  Test:  {cv_results[test_key].mean():.4f} (+/- {cv_results[test_key].std()*2:.4f})")

# Also returns fit_time and score_time
print(f"\nAverage fit time: {cv_results['fit_time'].mean():.2f}s")
print(f"Average score time: {cv_results['score_time'].mean():.2f}s")
```

Cross-Validation with Predictions

```python
from sklearn.model_selection import cross_val_predict

# Get predictions for each sample (when it was in validation set)
model = RandomForestClassifier(random_state=42)
y_pred = cross_val_predict(model, X, y, cv=5)

# Now can compute any metric
from sklearn.metrics import classification_report, confusion_matrix

print("Classification Report (CV predictions):")
print(classification_report(y, y_pred))

print("\nConfusion Matrix:")
print(confusion_matrix(y, y_pred))

# Get probabilities instead of predictions
y_proba = cross_val_predict(model, X, y, cv=5, method='predict_proba')
print(f"\nProbability shape: {y_proba.shape}")

# Compute ROC-AUC from probabilities
from sklearn.metrics import roc_auc_score
if len(np.unique(y)) == 2:  # Binary classification
    auc = roc_auc_score(y, y_proba[:, 1])
    print(f"ROC-AUC: {auc:.4f}")
```

Time Series Split

```python
from sklearn.model_selection import TimeSeriesSplit

# Time series split: train on past, test on future
tscv = TimeSeriesSplit(n_splits=5)

# Visualize splits
for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    print(f"Fold {fold+1}:")
    print(f"  Train: indices {train_idx[0]} to {train_idx[-1]} ({len(train_idx)} samples)")
    print(f"  Val:   indices {val_idx[0]} to {val_idx[-1]} ({len(val_idx)} samples)")

# Use with cross_val_score
from sklearn.ensemble import GradientBoostingRegressor

model = GradientBoostingRegressor(random_state=42)
scores = cross_val_score(model, X, y, cv=tscv, scoring='neg_mean_squared_error')

rmse_scores = np.sqrt(-scores)
print(f"\nTime Series CV RMSE: {rmse_scores.mean():.4f} (+/- {rmse_scores.std()*2:.4f})")
```

Time Series Split with Gap

```python
from sklearn.model_selection import TimeSeriesSplit

# Add gap between train and test to prevent leakage from lag features
tscv = TimeSeriesSplit(n_splits=5, gap=10)  # 10 samples gap

for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    print(f"Fold {fold+1}:")
    print(f"  Train ends at: {train_idx[-1]}")
    print(f"  Gap: {val_idx[0] - train_idx[-1] - 1} samples")
    print(f"  Val starts at: {val_idx[0]}")

# With maximum train size (sliding window)
tscv = TimeSeriesSplit(n_splits=5, max_train_size=1000)

for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    print(f"Fold {fold+1}: Train size = {len(train_idx)}")

# Custom time series split with test size
tscv = TimeSeriesSplit(n_splits=5, test_size=100)

for fold, (train_idx, val_idx) in enumerate(tscv.split(X)):
    print(f"Fold {fold+1}: Val size = {len(val_idx)}")
```

Custom Sliding Window Split

```python
def sliding_window_split(X, window_size, test_size, step=1):
    """Custom sliding window time series split"""
    n_samples = len(X)
    indices = np.arange(n_samples)

    splits = []
    start = 0
    while start + window_size + test_size <= n_samples:
        train_idx = indices[start:start + window_size]
        test_idx = indices[start + window_size:start + window_size + test_size]
        splits.append((train_idx, test_idx))
        start += step

    return splits

# Usage
splits = sliding_window_split(X, window_size=100, test_size=20, step=20)
print(f"Number of splits: {len(splits)}")

scores = []
for train_idx, test_idx in splits:
    X_train, X_test = X[train_idx], X[test_idx]
    y_train, y_test = y[train_idx], y[test_idx]

    model.fit(X_train, y_train)
    scores.append(model.score(X_test, y_test))

print(f"Sliding Window CV: {np.mean(scores):.4f} (+/- {np.std(scores)*2:.4f})")
```

Blocking Time Series Split

```python
from sklearn.model_selection import TimeSeriesSplit

class BlockingTimeSeriesSplit:
    """Purged time series split to prevent leakage"""
    def __init__(self, n_splits=5, purge_gap=0, embargo_gap=0):
        self.n_splits = n_splits
        self.purge_gap = purge_gap  # Gap before test set
        self.embargo_gap = embargo_gap  # Gap after test set

    def split(self, X, y=None, groups=None):
        n_samples = len(X)
        fold_size = n_samples // (self.n_splits + 1)

        for i in range(self.n_splits):
            test_start = (i + 1) * fold_size
            test_end = test_start + fold_size

            train_end = test_start - self.purge_gap
            train_idx = np.arange(0, train_end)
            test_idx = np.arange(test_start, min(test_end, n_samples))

            yield train_idx, test_idx

# Usage
btscv = BlockingTimeSeriesSplit(n_splits=5, purge_gap=5)
scores = cross_val_score(model, X, y, cv=btscv)
```

Checking Class Imbalance

```python
# Check class distribution
print("Class distribution:")
print(pd.Series(y).value_counts())
print(f"\nImbalance ratio: {pd.Series(y).value_counts().max() / pd.Series(y).value_counts().min():.2f}")

# Visualize
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 5))
pd.Series(y).value_counts().plot(kind='bar')
plt.title('Class Distribution')
plt.xlabel('Class')
plt.ylabel('Count')
plt.tight_layout()
plt.show()

# Calculate class weights
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight('balanced', classes=np.unique(y), y=y)
weight_dict = dict(zip(np.unique(y), class_weights))
print(f"\nComputed class weights: {weight_dict}")
```

Random Over-Sampling

```python
from imblearn.over_sampling import RandomOverSampler

# Random over-sampling: duplicate minority class samples
ros = RandomOverSampler(random_state=42)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)

print(f"Original: {pd.Series(y_train).value_counts().to_dict()}")
print(f"Resampled: {pd.Series(y_resampled).value_counts().to_dict()}")

# With specific sampling strategy
ros = RandomOverSampler(
    sampling_strategy='minority',  # Only oversample minority
    random_state=42
)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)

# Custom ratio
ros = RandomOverSampler(
    sampling_strategy=0.5,  # Minority will be 50% of majority
    random_state=42
)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)

# Specific counts per class
ros = RandomOverSampler(
    sampling_strategy={0: 1000, 1: 1000},  # Both classes to 1000
    random_state=42
)
```

SMOTE (Synthetic Minority Over-sampling)

```python
from imblearn.over_sampling import SMOTE

# SMOTE: create synthetic samples
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)

print(f"Original: {pd.Series(y_train).value_counts().to_dict()}")
print(f"After SMOTE: {pd.Series(y_resampled).value_counts().to_dict()}")

# SMOTE with parameters
smote = SMOTE(
    sampling_strategy='auto',  # Resample all classes but majority
    k_neighbors=5,             # Number of neighbors for synthesis
    random_state=42
)

# Borderline-SMOTE: focus on borderline samples
from imblearn.over_sampling import BorderlineSMOTE
bsmote = BorderlineSMOTE(random_state=42, kind='borderline-1')
X_resampled, y_resampled = bsmote.fit_resample(X_train, y_train)

# SVMSMOTE: use SVM to identify samples
from imblearn.over_sampling import SVMSMOTE
svmsmote = SVMSMOTE(random_state=42)
X_resampled, y_resampled = svmsmote.fit_resample(X_train, y_train)
```

ADASYN (Adaptive Synthetic Sampling)

```python
from imblearn.over_sampling import ADASYN

# ADASYN: more synthetic samples where density is low
adasyn = ADASYN(random_state=42)
X_resampled, y_resampled = adasyn.fit_resample(X_train, y_train)

print(f"Original: {pd.Series(y_train).value_counts().to_dict()}")
print(f"After ADASYN: {pd.Series(y_resampled).value_counts().to_dict()}")

# Note: ADASYN may not produce exact target ratio
# because it adapts to local data density

# ADASYN parameters
adasyn = ADASYN(
    sampling_strategy='minority',
    n_neighbors=5,
    random_state=42
)

# Comparison: SMOTE vs ADASYN
# SMOTE: uniform synthetic samples
# ADASYN: more samples in harder-to-learn regions
```

Random Under-Sampling

```python
from imblearn.under_sampling import RandomUnderSampler

# Random under-sampling: remove majority class samples
rus = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = rus.fit_resample(X_train, y_train)

print(f"Original: {pd.Series(y_train).value_counts().to_dict()}")
print(f"After undersampling: {pd.Series(y_resampled).value_counts().to_dict()}")

# Warning: may lose important information!

# With replacement (bootstrap)
rus = RandomUnderSampler(replacement=True, random_state=42)

# Specific ratio
rus = RandomUnderSampler(
    sampling_strategy=0.5,  # Majority will be 2x minority
    random_state=42
)

# NearMiss: keep majority samples close to minority
from imblearn.under_sampling import NearMiss
nm = NearMiss(version=1)  # version 1, 2, or 3
X_resampled, y_resampled = nm.fit_resample(X_train, y_train)
```

Tomek Links and ENN

```python
from imblearn.under_sampling import TomekLinks, EditedNearestNeighbours

# Tomek Links: remove majority samples that form Tomek links
tomek = TomekLinks()
X_resampled, y_resampled = tomek.fit_resample(X_train, y_train)
print(f"Removed {len(X_train) - len(X_resampled)} samples (Tomek)")

# ENN: remove samples misclassified by KNN
enn = EditedNearestNeighbours(n_neighbors=3)
X_resampled, y_resampled = enn.fit_resample(X_train, y_train)
print(f"Removed {len(X_train) - len(X_resampled)} samples (ENN)")

# Repeated ENN
from imblearn.under_sampling import RepeatedEditedNearestNeighbours
renn = RepeatedEditedNearestNeighbours()
X_resampled, y_resampled = renn.fit_resample(X_train, y_train)

# These methods clean the data by removing noisy samples
# Often used in combination with SMOTE
```

Combined Sampling: SMOTE + Tomek/ENN

```python
from imblearn.combine import SMOTETomek, SMOTEENN

# SMOTE + Tomek Links: oversample then clean
smote_tomek = SMOTETomek(random_state=42)
X_resampled, y_resampled = smote_tomek.fit_resample(X_train, y_train)

print(f"Original: {pd.Series(y_train).value_counts().to_dict()}")
print(f"After SMOTE-Tomek: {pd.Series(y_resampled).value_counts().to_dict()}")

# SMOTE + ENN: oversample then clean with ENN
smote_enn = SMOTEENN(random_state=42)
X_resampled, y_resampled = smote_enn.fit_resample(X_train, y_train)

print(f"After SMOTE-ENN: {pd.Series(y_resampled).value_counts().to_dict()}")

# Custom combination
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import TomekLinks
from imblearn.pipeline import Pipeline

resampling_pipeline = Pipeline([
    ('smote', SMOTE(random_state=42)),
    ('tomek', TomekLinks())
])
X_resampled, y_resampled = resampling_pipeline.fit_resample(X_train, y_train)
```

Using Class Weights

```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression

# Method 1: class_weight='balanced' (auto-compute weights)
model = RandomForestClassifier(class_weight='balanced', random_state=42)
model.fit(X_train, y_train)

# Method 2: Custom class weights
class_weights = {0: 1, 1: 10}  # Minority class 10x more important
model = RandomForestClassifier(class_weight=class_weights, random_state=42)

# Method 3: Compute balanced weights
from sklearn.utils.class_weight import compute_class_weight
weights = compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)
class_weights = dict(zip(np.unique(y_train), weights))
print(f"Computed weights: {class_weights}")

# For Logistic Regression
model = LogisticRegression(class_weight='balanced', random_state=42)

# Sample weights (per-sample, not per-class)
sample_weights = np.where(y_train == 1, 10, 1)  # Minority samples weight 10
model.fit(X_train, y_train, sample_weight=sample_weights)
```

Imbalanced-Learn Pipeline

```python
from imblearn.pipeline import Pipeline as ImbPipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier
from imblearn.over_sampling import SMOTE

# Pipeline that includes resampling
# Note: use imblearn's Pipeline, not sklearn's!
pipeline = ImbPipeline([
    ('scaler', StandardScaler()),
    ('smote', SMOTE(random_state=42)),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Fit the pipeline
pipeline.fit(X_train, y_train)

# Predictions
y_pred = pipeline.predict(X_test)

# Cross-validation with resampling
from sklearn.model_selection import cross_val_score
scores = cross_val_score(pipeline, X, y, cv=5, scoring='f1')
print(f"CV F1: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")

# Important: Resampling is applied only during fit, not predict!
```

Classification Report

```python
from sklearn.metrics import classification_report, confusion_matrix

# Full classification report
print(classification_report(y_test, y_pred))

# With target names
print(classification_report(y_test, y_pred,
                            target_names=['Negative', 'Positive']))

# As dictionary (for programmatic access)
report = classification_report(y_test, y_pred, output_dict=True)
print(f"Precision (class 1): {report['1']['precision']:.4f}")
print(f"Recall (class 1): {report['1']['recall']:.4f}")
print(f"F1 (class 1): {report['1']['f1-score']:.4f}")

# Confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("\nConfusion Matrix:")
print(cm)

# As DataFrame for better visualization
cm_df = pd.DataFrame(cm, index=['Actual 0', 'Actual 1'],
                     columns=['Pred 0', 'Pred 1'])
print(cm_df)
```

Precision, Recall, F1-Score

```python
from sklearn.metrics import precision_score, recall_score, f1_score

# Binary classification
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")

# Multi-class: specify averaging method
precision_macro = precision_score(y_test, y_pred, average='macro')
precision_weighted = precision_score(y_test, y_pred, average='weighted')
precision_micro = precision_score(y_test, y_pred, average='micro')

print(f"\nMacro Precision: {precision_macro:.4f}")  # Unweighted mean
print(f"Weighted Precision: {precision_weighted:.4f}")  # Weighted by support
print(f"Micro Precision: {precision_micro:.4f}")  # Global calculation

# F-beta score (custom beta)
from sklearn.metrics import fbeta_score
f2 = fbeta_score(y_test, y_pred, beta=2)  # Recall 2x more important
f05 = fbeta_score(y_test, y_pred, beta=0.5)  # Precision 2x more important
```

ROC-AUC Score

```python
from sklearn.metrics import roc_auc_score, roc_curve

# Get probabilities
y_proba = model.predict_proba(X_test)[:, 1]

# ROC-AUC score
auc = roc_auc_score(y_test, y_proba)
print(f"ROC-AUC: {auc:.4f}")

# ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_proba)

# Plot ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, label=f'ROC (AUC = {auc:.4f})')
plt.plot([0, 1], [0, 1], 'k--', label='Random')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()

# Multi-class ROC-AUC
y_proba_multi = model.predict_proba(X_test)
auc_ovr = roc_auc_score(y_test, y_proba_multi, multi_class='ovr')
auc_ovo = roc_auc_score(y_test, y_proba_multi, multi_class='ovo')
```

Precision-Recall Curve (Better for Imbalanced)

```python
from sklearn.metrics import precision_recall_curve, average_precision_score

# Get probabilities
y_proba = model.predict_proba(X_test)[:, 1]

# Average Precision (Area under PR curve)
ap = average_precision_score(y_test, y_proba)
print(f"Average Precision: {ap:.4f}")

# Precision-Recall curve
precision, recall, thresholds = precision_recall_curve(y_test, y_proba)

# Plot PR curve
plt.figure(figsize=(8, 6))
plt.plot(recall, precision, label=f'PR (AP = {ap:.4f})')
plt.axhline(y=y_test.mean(), color='r', linestyle='--', label='Baseline')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()
plt.show()

# Find best threshold (maximize F1)
f1_scores = 2 * (precision * recall) / (precision + recall + 1e-8)
best_idx = np.argmax(f1_scores[:-1])  # Last element has no threshold
best_threshold = thresholds[best_idx]
print(f"Best threshold for F1: {best_threshold:.4f}")
```

Threshold Tuning

```python
from sklearn.metrics import precision_score, recall_score, f1_score

# Get probabilities
y_proba = model.predict_proba(X_test)[:, 1]

# Test different thresholds
thresholds = np.arange(0.1, 0.9, 0.05)
results = []

for thresh in thresholds:
    y_pred_thresh = (y_proba >= thresh).astype(int)
    results.append({
        'threshold': thresh,
        'precision': precision_score(y_test, y_pred_thresh),
        'recall': recall_score(y_test, y_pred_thresh),
        'f1': f1_score(y_test, y_pred_thresh)
    })

results_df = pd.DataFrame(results)
print(results_df)

# Find optimal threshold
best_f1_idx = results_df['f1'].idxmax()
best_threshold = results_df.loc[best_f1_idx, 'threshold']
print(f"\nBest threshold (F1): {best_threshold}")

# Apply best threshold
y_pred_optimal = (y_proba >= best_threshold).astype(int)
print(f"F1 with optimal threshold: {f1_score(y_test, y_pred_optimal):.4f}")
```

Confusion Matrix Visualization

```python
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# Method 1: ConfusionMatrixDisplay
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(cm, display_labels=['Negative', 'Positive'])
disp.plot(cmap='Blues')
plt.title('Confusion Matrix')
plt.show()

# Method 2: Seaborn heatmap (more customizable)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Pred Neg', 'Pred Pos'],
            yticklabels=['Actual Neg', 'Actual Pos'])
plt.title('Confusion Matrix')
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()

# Normalized confusion matrix (percentages)
cm_normalized = confusion_matrix(y_test, y_pred, normalize='true')
sns.heatmap(cm_normalized, annot=True, fmt='.2%', cmap='Blues')
plt.title('Normalized Confusion Matrix')
plt.show()
```

Full Pipeline with Preprocessing

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import RandomForestClassifier

# Define column types
numerical_cols = ['age', 'income', 'balance']
categorical_cols = ['gender', 'education']

# Preprocessing pipelines
num_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

cat_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='constant', fill_value='Missing')),
    ('encoder', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

# Column transformer
preprocessor = ColumnTransformer([
    ('num', num_pipeline, numerical_cols),
    ('cat', cat_pipeline, categorical_cols)
])

# Full pipeline
full_pipeline = Pipeline([
    ('preprocessor', preprocessor)
```

Pipeline with Resampling

```python
from imblearn.pipeline import Pipeline as ImbPipeline
from imblearn.over_sampling import SMOTE

# Full pipeline with SMOTE
full_pipeline = ImbPipeline([
    ('preprocessor', preprocessor),
    ('smote', SMOTE(random_state=42)),
    ('classifier', RandomForestClassifier(random_state=42))
])

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    df[numerical_cols + categorical_cols], df['target'],
    test_size=0.2, stratify=df['target'], random_state=42
)

# Fit pipeline
full_pipeline.fit(X_train, y_train)

# Evaluate
y_pred = full_pipeline.predict(X_test)
print(classification_report(y_test, y_pred))

# Cross-validation
scores = cross_val_score(full_pipeline, X_train, y_train, cv=5, scoring='f1')
print(f"CV F1: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")
```

Hyperparameter Tuning with GridSearchCV

```python
from sklearn.model_selection import GridSearchCV

# Parameter grid
param_grid = {
    'preprocessor__num__imputer__strategy': ['mean', 'median'],
    'smote__k_neighbors': [3, 5, 7],
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [5, 10, None],
    'classifier__class_weight': [None, 'balanced']
}

# Grid search with stratified CV
grid_search = GridSearchCV(
    full_pipeline,
    param_grid,
    cv=5,
    scoring='f1',
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train, y_train)

print(f"Best parameters: {grid_search.best_params_}")
print(f"Best CV F1: {grid_search.best_score_:.4f}")

# Evaluate best model on test set
y_pred = grid_search.predict(X_test)
print(f"Test F1: {f1_score(y_test, y_pred):.4f}")
```

Avoiding Data Leakage

```python
# WRONG: Preprocessing before split (DATA LEAKAGE!)
# scaler.fit(X)  # Sees test data!
# X_scaled = scaler.transform(X)
# X_train, X_test = train_test_split(X_scaled, ...)

# CORRECT: Split first, then preprocess
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Fit on training data only
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # fit_transform on train
X_test_scaled = scaler.transform(X_test)  # transform only on test

# BEST: Use Pipeline (handles this automatically)
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', RandomForestClassifier())
])
pipeline.fit(X_train, y_train)  # Scaler sees only training data
predictions = pipeline.predict(X_test)

# Same for resampling: ONLY on training data
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)
# Never apply SMOTE to test data!
```

Saving and Loading Complete Pipeline

```python
import joblib

# Save the complete pipeline
joblib.dump(full_pipeline, 'model_pipeline.joblib')

# Save with compression
joblib.dump(full_pipeline, 'model_pipeline.joblib.gz', compress=3)

# Load pipeline
loaded_pipeline = joblib.load('model_pipeline.joblib')

# Make predictions with loaded pipeline
new_data = pd.DataFrame({
    'age': [35, 45],
    'income': [50000, 75000],
    'balance': [10000, 25000],
    'gender': ['M', 'F'],
    'education': ['Bachelor', 'Master']
})

predictions = loaded_pipeline.predict(new_data)
probabilities = loaded_pipeline.predict_proba(new_data)

print(f"Predictions: {predictions}")
print(f"Probabilities: {probabilities}")

# Save grid search results
joblib.dump(grid_search.best_estimator_, 'best_model.joblib')
```

Data Preparation Checklist

Splitting:

- ☐ Train/test split done
- ☐ Stratification applied
- ☐ Random state documented
- ☐ Groups kept together (if applicable)

Cross-Validation:

- ☐ CV method chosen
- ☐ Stratified for classification
- ☐ Time-aware for time series
- ☐ Multiple metrics computed

Imbalanced Data:

- ☐ Class distribution checked
- ☐ Resampling applied (train only!)
- ☐ Appropriate metrics chosen
- ☐ Threshold tuned

Pipeline:

- ☐ No data leakage
- ☐ Pipeline tested
- ☐ Model saved
- ☐ Reproducible

Key Python Functions Reference

Task	Function/Class
Train-test split	`train_test_split(stratify=y)`
K-Fold CV	`KFold`, `StratifiedKFold`
Repeated CV	`RepeatedStratifiedKFold`
Time series CV	`TimeSeriesSplit`
CV scores	`cross_val_score`, `cross_validate`
CV predictions	`cross_val_predict`
Oversampling	`SMOTE`, `ADASYN`, `RandomOverSampler`
Undersampling	`RandomUnderSampler`, `TomekLinks`
Combined	`SMOTETomek`, `SMOTEENN`
Class weights	`class_weight='balanced'`
Metrics	`f1_score`, `roc_auc_score`
Pipeline	`Pipeline`, `imblearn.pipeline.Pipeline`

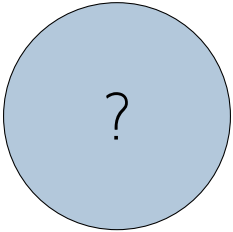
Course Complete!

Congratulations!

You have completed all 4 parts of the
Machine Learning Pre-Processing Course

Questions?

Questions & Discussion



Contact: abdellaoui.e@gmail.com

References

- Kohavi, R. (1995). *A Study of Cross-Validation and Bootstrap*. IJCAI.
- Chawla, N.V. et al. (2002). *SMOTE: Synthetic Minority Over-sampling Technique*. JAIR.
- He, H. & Garcia, E.A. (2009). *Learning from Imbalanced Data*. IEEE TKDE.
- Lemaître, G. et al. (2017). *Imbalanced-learn: A Python Toolbox*. JMLR.
- Bergstra, J. & Bengio, Y. (2012). *Random Search for Hyper-Parameter Optimization*. JMLR.
- Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.

Thank You!

Good luck with your Machine Learning projects!