Data Pre-processing
with Python

A 30-hour practical course

Yaé Ulrich Gaba

AIRINA Labs

2026

Contents

Preface

Data preprocessing is the unglamorous but decisive phase of every data project. Studies consistently show that data scientists spend 60–80% of their time cleaning and preparing data. Yet most courses skip over this phase in a single lecture, eager to reach the “interesting” parts: models, algorithms, predictions.

This course takes the opposite approach. We devote 30 hours to the art and science of turning raw, messy, real-world data into clean, analysis-ready datasets. If you learn to preprocess data well, every model you build afterward will be better.

We use Python because its ecosystem — `pandas`, `scikit-learn`, `missingno`, `nltk` — provides the most complete toolkit for data preprocessing available in any language. Every example uses a real, publicly available dataset with real problems: missing values, inconsistent types, outliers, mixed encodings, and messy text.

What this course is not. This is not a machine learning course (though we prepare data for ML). It is not a statistics course (though we use statistical reasoning). It is a *practical data preparation course* for people who want to master the most time-consuming part of the data pipeline.

Prerequisites. Basic Python (variables, lists, loops, functions). Familiarity with pandas DataFrames is helpful but will be reviewed in Chapter 1.

Software. All code runs in Jupyter notebooks, either locally (Anaconda) or in the cloud (Google Colab — free, no installation required).

Yaé Ulrich Gaba
Cotonou, 2026

Chapter 1

Data landscape and pipelines

> *"Data is the new oil — but like oil, it must be refined before it is useful."* —
> Clive Humby

1.1 What is data preprocessing?

Data preprocessing is the set of transformations applied to raw data before analysis or modeling. It bridges the gap between the data you *have* and the data you *need*.

A typical preprocessing pipeline includes:

1. **Loading** — reading data from files, databases, or APIs.
2. **Inspection** — understanding structure, types, and quality.
3. **Cleaning** — handling missing values, duplicates, and errors.
4. **Transformation** — encoding, scaling, and creating features.
5. **Validation** — confirming the output meets downstream requirements.

Data tip

In industry surveys, data practitioners report spending 60–80% of project time on preprocessing. Mastering this phase is the single highest-leverage skill in data science.

1.2 Data types and structures

1.2.1 Structured, semi-structured, and unstructured data

- **Structured:** tables with rows and columns (CSV, SQL databases, Excel).
- **Semi-structured:** data with flexible schemas (JSON, XML, YAML).
- **Unstructured:** free text, images, audio, video.

This course focuses on structured and semi-structured data, with one chapter (Chapter 7) devoted to text.

CHAPTER 1. DATA LANDSCAPE AND PIPELINES

1.2.2 Variable types

```
import pandas as pd

# Load a real dataset to inspect variable types
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)
print(df.dtypes)
```

Output (abbreviated):

```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Fare           float64
Embarked        object
```

Preprocessing tip

The `object` dtype in pandas usually means string data, but it can also hide mixed types. Always verify with `df[col].apply(type).value_counts()`.

1.2.3 Numerical vs. categorical

```
numerical = df.select_dtypes(include="number").columns.tolist()
categorical = df.select_dtypes(include="object").columns.tolist()

print(f"Numerical columns ({len(numerical)}): {numerical}")
print(f"Categorical columns ({len(categorical)}): {categorical}")
```

1.3 Common data formats

Format	Extension	pandas reader	Notes
CSV	`.csv`	`read_csv`	Most common; watch for delimiter and encoding
Excel	`.xlsx`	`read_excel`	Requires `openpyxl`; multi-sheet support
JSON	`.json`	`read_json`	Nested structures need `json_normalize`

Parquet	`.parquet`	`read_parquet`	Columnar, fast, type-preserving
SQL	—	`read_sql`	Requires `sqlalchemy` connection

1.4 First look at a real dataset

We will use the Melbourne Housing dataset throughout several chapters:

```python
# Melbourne Housing dataset
melb = pd.read_csv("melb_data.csv")

# Shape: rows x columns
print(f"Shape: {melb.shape}")

# First 3 rows
print(melb.head(3))

# Summary statistics
print(melb.describe())

# Missing values per column
print(melb.isnull().sum().sort_values(ascending=False).head(10))
```

```python
# Quick quality report
def data_quality_report(df):
    """Generate a quick quality report for any DataFrame."""
    report = pd.DataFrame({
        "dtype": df.dtypes,
        "non_null": df.notnull().sum(),
        "null_count": df.isnull().sum(),
        "null_pct": (df.isnull().sum() / len(df) * 100).round(1),
        "n_unique": df.nunique(),
        "sample_value": df.iloc[0]
    })
    return report.sort_values("null_pct", ascending=False)

print(data_quality_report(melb))
```

Data tip

Always run a quality report *before* any analysis. The five numbers you need immediately are: shape, dtypes, null counts, unique counts, and summary statistics.

1.5 The preprocessing workflow

```python
# A minimal end-to-end preprocessing workflow
```

CHAPTER 1. DATA LANDSCAPE AND PIPELINES

```
import pandas as pd
import numpy as np

# 1. Load
df = pd.read_csv("melb_data.csv")

# 2. Inspect
print(df.info())

# 3. Clean: drop duplicates, handle missing values
df = df.drop_duplicates()
df["Car"] = df["Car"].fillna(df["Car"].median())
df = df.dropna(subset=["Price"])

# 4. Transform: encode categorical, scale numerical
df["Type_encoded"] = df["Type"].map({"h": 0, "u": 1, "t": 2})

# 5. Validate
assert df.isnull().sum().sum() == 0 or True  # relaxed for demo
print(f"Clean shape: {df.shape}")
```

1.6 Pandas review: essential operations

1.6.1 Selection and filtering

```
# Select columns
prices = melb[["Suburb", "Price", "Rooms"]]

# Filter rows
expensive = melb[melb["Price"] > 2_000_000]
south = melb[melb["Regionname"] == "Southern Metropolitan"]

# Combined
big_south = melb[(melb["Regionname"] == "Southern Metropolitan") &
                 (melb["Rooms"] >= 4)]
print(f"Large southern houses: {len(big_south)}")
```

1.6.2 Groupby and aggregation

```
# Average price by region
region_stats = (melb.groupby("Regionname")["Price"]
                .agg(["mean", "median", "count"])
                .sort_values("median", ascending=False))
print(region_stats)
```

Caution

Pandas `groupby` silently drops `NaN` group keys by default. If your grouping column has missing values, they will be excluded from the result. Use `dropna=False` to include them.

1.7 Tools and libraries overview

```python
# Core preprocessing stack
import pandas as pd                    # data manipulation
import numpy as np                      # numerical operations
import matplotlib.pyplot as plt         # visualization
import seaborn as sns                   # statistical plots
import missingno as msno                # missing data visualization
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
```

1.8 Exercises

Exercise

1. Download the Melbourne Housing dataset. Write a quality report showing: number of rows, columns, dtype of each column, percentage of missing values, and number of unique values.

2. Load the Gapminder dataset (`gapminder.csv` or via `plotly.express`). Identify which columns are numerical, which are categorical, and which have missing values.

3. Write a function `detect_mixed_types(df)` that checks each column for mixed Python types and returns a list of problematic columns.

4. Compare loading the Titanic dataset as CSV vs. Parquet. Measure file size and load time using `%timeit`.

1.9 Chapter summary

- Data preprocessing transforms raw data into analysis-ready data and consumes the majority of project time.

- Data comes in structured, semi-structured, and unstructured formats; pandas handles the first two.

- Every preprocessing workflow follows five steps: load, inspect, clean, transform, validate.

CHAPTER 1. DATA LANDSCAPE AND PIPELINES

- A quick quality report (shape, dtypes, nulls, uniques, statistics) should be your first action on any new dataset.
- The core Python stack for preprocessing is pandas, numpy, scikit-learn, matplotlib, and missingno.

Chapter 2

Data loading

"You can't analyze what you can't load."

2.1 Loading CSV files

CSV (comma-separated values) is the most common data exchange format. Pandas handles it with `read_csv`, which has over 50 parameters for dealing with real-world quirks.

```
import pandas as pd

# Basic load
df = pd.read_csv("titanic.csv")
print(df.shape)
print(df.head())
```

2.1.1 Common CSV problems

```
# Problem 1: Wrong delimiter
# Some European CSVs use semicolons
df_euro = pd.read_csv("data_eu.csv", sep=";", decimal=",")

# Problem 2: Encoding issues
df_fr = pd.read_csv("french_data.csv", encoding="latin-1")

# Problem 3: Messy headers
df = pd.read_csv("messy.csv", header=0, skiprows=[1, 2])
df.columns = df.columns.str.strip().str.lower().str.replace(" ", "_")
print(df.columns.tolist())
```

Preprocessing tip

Always standardize column names immediately after loading: lowercase, underscores, no spaces. This prevents subtle bugs from inconsistent capitalization.

2.1.2 Optimizing CSV loads

```
# Load only needed columns (saves memory)
cols = ["PassengerId", "Survived", "Pclass", "Sex", "Age", "Fare"]
df = pd.read_csv("titanic.csv", usecols=cols)

# Specify dtypes to save memory
dtypes = {"Survived": "int8", "Pclass": "int8", "Sex": "category"}
df = pd.read_csv("titanic.csv", usecols=cols, dtype=dtypes)
print(df.memory_usage(deep=True))

# Load in chunks for large files
chunks = pd.read_csv("large_file.csv", chunksize=50_000)
result = pd.concat([chunk[chunk["Age"] > 30] for chunk in chunks])
```

Caution

Never load a multi-gigabyte CSV into memory all at once. Use `chunksize`, or switch to Parquet/Feather format. A 2 GB CSV typically becomes 400 MB in Parquet with faster load times.

2.2 Loading Excel files

```
# Basic Excel load (requires openpyxl)
df = pd.read_excel("survey_results.xlsx", sheet_name="Sheet1")

# Load all sheets into a dictionary
all_sheets = pd.read_excel("multi_sheet.xlsx", sheet_name=None)
for name, sheet_df in all_sheets.items():
    print(f"Sheet '{name}': {sheet_df.shape}")

# Skip header rows and footers
df = pd.read_excel("report.xlsx", skiprows=3, skipfooter=2,
                   engine="openpyxl")
```

Data tip

Excel files preserve formatting but lose type precision. Dates may load as strings, percentages as floats between 0 and 1, and currency as plain numbers. Always verify dtypes after loading Excel.

2.3 Loading JSON data

```
import json

# Flat JSON
```

```
df = pd.read_json("flat_data.json")

# Nested JSON (common from APIs)
with open("nested_data.json") as f:
    raw = json.load(f)

# Use json_normalize to flatten
from pandas import json_normalize

df = json_normalize(raw["results"],
                    record_path="measurements",
                    meta=["station_id", "station_name"])
print(df.head())
```

```
# Real example: loading Gapminder-style JSON
import requests

url =
↪  "https://api.worldbank.org/v2/country/BEN/indicator/SP.POP.TOTL?format=json&per_page=60'
response = requests.get(url)
data = response.json()

# The actual data is in data[1]
df_wb = pd.DataFrame(data[1])
print(df_wb[["date", "value"]].head(10))
```

2.4 Loading from SQL databases

```
from sqlalchemy import create_engine

# SQLite (file-based, no server needed)
engine = create_engine("sqlite:///local_data.db")

# Load entire table
df = pd.read_sql("SELECT * FROM patients", engine)

# Load with filtering (push computation to the database)
query = """
SELECT patient_id, age, diagnosis, lab_value
FROM patients
WHERE age > 40
  AND diagnosis IS NOT NULL
ORDER BY lab_value DESC
LIMIT 10000
"""
df = pd.read_sql(query, engine)
print(df.shape)
```

Preprocessing tip

When loading from SQL, always push as much filtering as possible into the query. Loading 10 million rows into pandas and then filtering is much slower than letting the database filter first.

2.5 Loading from APIs

```
import requests
import pandas as pd

# REST API example: Open Meteo weather API
url = "https://archive-api.open-meteo.com/v1/archive"
params = {
    "latitude": 6.36,     # Cotonou
    "longitude": 2.43,
    "start_date": "2024-01-01",
    "end_date": "2024-12-31",
    "daily": "temperature_2m_mean,precipitation_sum",
}

response = requests.get(url, params=params)
data = response.json()

df_weather = pd.DataFrame({
    "date": data["daily"]["time"],
    "temp_mean": data["daily"]["temperature_2m_mean"],
    "precip_mm": data["daily"]["precipitation_sum"],
})
df_weather["date"] = pd.to_datetime(df_weather["date"])
print(df_weather.head())
```

```
# Pagination: many APIs return data in pages
all_records = []
page = 1

while True:
    resp = requests.get(f"{base_url}?page={page}&per_page=100")
    records = resp.json()["results"]
    if not records:
        break
    all_records.extend(records)
    page += 1

df = pd.DataFrame(all_records)
print(f"Total records: {len(df)}")
```

2.6 Web scraping basics

```
import requests
from bs4 import BeautifulSoup
import pandas as pd

# Scrape an HTML table
url = "https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)"
response = requests.get(url)
soup = BeautifulSoup(response.content, "html.parser")

# pandas can read HTML tables directly
tables = pd.read_html(url)
print(f"Found {len(tables)} tables on the page")
gdp_df = tables[2]  # pick the right table by index
print(gdp_df.head())
```

Caution

Web scraping may violate a website's terms of service. Always check `robots.txt` and the site's ToS before scraping. For research, prefer official APIs and open data portals.

2.7 Comparing formats: performance

```
import time

# Compare CSV vs Parquet load times
# First, create a Parquet version
df = pd.read_csv("large_dataset.csv")
df.to_parquet("large_dataset.parquet")

# Time CSV load
start = time.time()
df_csv = pd.read_csv("large_dataset.csv")
csv_time = time.time() - start

# Time Parquet load
start = time.time()
df_pq = pd.read_parquet("large_dataset.parquet")
pq_time = time.time() - start

print(f"CSV load:     {csv_time:.2f}s")
print(f"Parquet load: {pq_time:.2f}s")
print(f"Speedup:      {csv_time / pq_time:.1f}x")
```

2.8 Exercises

Exercise

1. Load the Titanic CSV with optimized dtypes. Compare memory usage before and after dtype optimization.
2. Load data from the World Bank API for three African countries (Benin, Nigeria, Ghana) for the indicator `SP.POP.TOTL` (total population). Combine into a single DataFrame.
3. Load an Excel file with multiple sheets. Write a function that returns a dictionary mapping sheet names to their shapes.
4. Scrape an HTML table from Wikipedia and clean the column names (lowercase, underscores, no special characters).
5. Convert the Adult Census UCI dataset from CSV to Parquet. Compare file sizes and load times.

2.9 Chapter summary

- Pandas provides readers for CSV, Excel, JSON, SQL, Parquet, HTML, and more.
- CSV loading requires attention to delimiters, encodings, headers, and dtypes.
- JSON from APIs often needs `json_normalize` to flatten nested structures.
- SQL loads should push filtering into the query, not into pandas.
- APIs require handling pagination, rate limits, and authentication.
- Parquet is 3–5x faster than CSV for loading and uses 5x less disk space.

Chapter 3

Missing data

> *"The data you don't have is often more important than the data you do."* — Anonymous statistician

3.1 Why data goes missing

Missing data is not a nuisance — it is information. The *reason* data is missing determines how you should handle it. Three mechanisms were formalized by Rubin (1976):

Definition 3.1 (Missing data mechanisms).

- ***MCAR*** *(Missing Completely At Random): missingness is unrelated to any variable, observed or unobserved. Example: a lab sample is accidentally dropped.*
- ***MAR*** *(Missing At Random): missingness depends on observed variables but not on the missing value itself. Example: younger patients skip a depression questionnaire.*
- ***MNAR*** *(Missing Not At Random): missingness depends on the unobserved value. Example: high-income individuals refuse to report income.*

> ⚠ Caution
>
> You cannot statistically prove MNAR — it requires domain knowledge. But the distinction matters enormously: simple imputation works for MCAR, is reasonable for MAR, and can be badly biased for MNAR.

3.2 Detecting missing values

```python
import pandas as pd
import numpy as np

# Load Titanic data
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)

# Count missing values
```

```
print(df.isnull().sum())
print()
print(f"Total missing cells: {df.isnull().sum().sum()}")
print(f"Percentage missing: {df.isnull().mean().mean() * 100:.1f}%")
```

```
# Detailed missing report
def missing_report(df):
    """Return a DataFrame summarizing missing data."""
    missing = df.isnull().sum()
    pct = (missing / len(df)) * 100
    report = pd.DataFrame({"missing": missing, "pct": pct.round(1)})
    return report[report["missing"] > 0].sort_values("pct", ascending=False)

print(missing_report(df))
```

3.2.1 Hidden missing values

```
# Missing values are not always NaN
# Common disguises: "?", "N/A", "", -1, 999, -999, "unknown"
df_adult = pd.read_csv(

    ↪ "https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
    header=None,
    na_values=["?", " ?"],  # Adult dataset uses " ?" for missing
    names=["age", "workclass", "fnlwgt", "education", "education_num",
           "marital_status", "occupation", "relationship", "race",
           "sex", "capital_gain", "capital_loss", "hours_per_week",
           "native_country", "income"]
)
print(missing_report(df_adult))
```

Data tip

Always scan for sentinel values: `-1`, `999`, `0` in numeric columns; empty strings, “unknown”, “N/A” in text columns. Replace them with `np.nan` before any analysis.

3.3 Visualizing missing data

```
import missingno as msno
import matplotlib.pyplot as plt

# Matrix plot: white lines = missing
msno.matrix(df, figsize=(10, 5), sparkline=False)
plt.title("Missing data pattern - Titanic")
plt.tight_layout()
plt.savefig("missing_matrix.png", dpi=150)
```

```
plt.show()
```

```
# Bar chart: count of non-null values per column
msno.bar(df, figsize=(10, 4))
plt.tight_layout()
plt.show()

# Heatmap: correlation of missingness between columns
# Values near +1 mean columns tend to be missing together
msno.heatmap(df, figsize=(8, 6))
plt.tight_layout()
plt.show()
```

```
# Dendrogram: hierarchical clustering of missingness patterns
msno.dendrogram(df, figsize=(10, 5))
plt.tight_layout()
plt.show()
```

Preprocessing tip

The missingness correlation heatmap is extremely useful: if two columns have correlated missingness, they may share a common cause (e.g., both come from the same survey section that some respondents skipped).

3.4 Deletion strategies

3.4.1 Listwise deletion (dropping rows)

```
# Drop rows with ANY missing value
df_complete = df.dropna()
print(f"Before: {len(df)}, After: {len(df_complete)}")
print(f"Lost: {len(df) - len(df_complete)} rows "
      f"({(len(df) - len(df_complete)) / len(df) * 100:.1f}%)")
```

3.4.2 Column deletion

```
# Drop columns with more than 50% missing
threshold = 0.5
cols_to_drop = df.columns[df.isnull().mean() > threshold].tolist()
print(f"Dropping columns: {cols_to_drop}")
df_reduced = df.drop(columns=cols_to_drop)
```

⚠ Caution

Listwise deletion is only safe under MCAR. Under MAR or MNAR, it introduces bias. For the Titanic dataset, older passengers with higher-class tickets are more likely to have recorded ages — dropping missing ages skews the dataset toward lower-class passengers.

3.5 Simple imputation

3.5.1 Mean, median, and mode imputation

```
from sklearn.impute import SimpleImputer

# Numerical: median imputation (robust to outliers)
num_imputer = SimpleImputer(strategy="median")
df["Age"] = num_imputer.fit_transform(df[["Age"]])

# Categorical: mode (most frequent) imputation
cat_imputer = SimpleImputer(strategy="most_frequent")
df["Embarked"] = cat_imputer.fit_transform(df[["Embarked"]]).ravel()

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

3.5.2 Constant imputation

```
# Fill with a specific value
df["Cabin"] = df["Cabin"].fillna("Unknown")

# Fill numerical with 0 (use with caution)
df["some_col"] = df["some_col"].fillna(0)
```

3.6 Advanced imputation

3.6.1 KNN imputation

```
from sklearn.impute import KNNImputer

# KNN uses similar rows to estimate missing values
# Only works on numerical data
num_cols = ["Age", "Fare", "SibSp", "Parch"]
knn_imputer = KNNImputer(n_neighbors=5, weights="distance")
df[num_cols] = knn_imputer.fit_transform(df[num_cols])

print(f"Missing after KNN: {df[num_cols].isnull().sum().sum()}")
```

3.6.2 Iterative imputation (MICE)

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

# Iterative imputer models each feature as a function of others
iter_imputer = IterativeImputer(max_iter=10, random_state=42)
df[num_cols] = iter_imputer.fit_transform(df[num_cols])

print("Iterative imputation complete.")
print(df[num_cols].describe())
```

Preprocessing tip

KNN imputation respects local structure (similar passengers get similar ages). Iterative imputation captures linear relationships across features. Both outperform mean/median imputation when data is MAR.

3.7 Imputation with an indicator

```
# Create a binary indicator BEFORE imputing
df["Age_was_missing"] = df["Age"].isnull().astype(int)

# Then impute
df["Age"] = df["Age"].fillna(df["Age"].median())

# The indicator preserves the information that the value was missing
print(df[["Age", "Age_was_missing"]].head(10))
```

Data tip

Adding a missing indicator column is one of the most underused techniques. It lets downstream models learn whether the missingness itself is predictive. In the Titanic dataset, having a missing cabin number correlates with lower survival.

3.8 Choosing an imputation strategy

Method	Best for	Weakness	Mechanism
Drop rows	<5% missing	Loses data	MCAR only
Mean/Median	Quick baseline	Reduces variance	MCAR
Mode	Categorical	Ignores structure	MCAR

KNN	Clustered data	Slow on large data	MAR
Iterative (MICE)	Linear relationships	Assumes linearity	MAR
Domain constant	Known meaning	May introduce bias	Any

3.9 Exercises

Exercise

1. Load the Melbourne Housing dataset. Create a missing data report and a `missingno` matrix plot. Which columns have correlated missingness?

2. For the Titanic dataset, compare the distribution of `Age` before and after mean imputation vs. KNN imputation. Plot both distributions on the same histogram.

3. Load the Adult Census dataset (with `na_values=[" ?"]`) and impute the three categorical columns with missing values using mode imputation. Verify no missing values remain.

4. Implement a function `smart_impute(df)` that: (a) drops columns with >70% missing, (b) imputes numerical columns with median, (c) imputes categorical columns with mode, (d) adds missing indicators for all originally-missing columns.

5. Investigate whether `Age` in the Titanic dataset is MCAR by comparing the survival rate of passengers with and without a recorded age. What do you conclude?

3.10 Chapter summary

- Missing data has three mechanisms: MCAR, MAR, and MNAR. The mechanism determines which imputation strategy is valid.
- Always scan for hidden missing values (sentinel values like `?`, `-1`, `999`).
- The `missingno` library provides matrix, bar, heatmap, and dendrogram visualizations for missing patterns.
- Simple imputation (mean, median, mode) is fast but reduces variance and only works well under MCAR.
- KNN and iterative (MICE) imputation capture relationships between features and work under MAR.
- Adding a missing indicator column preserves the information content of missingness.

Chapter 4

Outlier detection and treatment

"An outlier is not noise until you have proven it is noise."

4.1 What is an outlier?

An outlier is an observation that deviates substantially from the pattern established by the majority of the data. Outliers can be:

- **Data entry errors**: a patient weight of 800 kg (meant 80).
- **Measurement artifacts**: a sensor returning -999 during calibration.
- **Genuine extreme values**: a house selling for $10 million in a suburb where the median is $500k.

> **Caution**
>
> Never remove outliers automatically. Each outlier must be investigated. Removing genuine extreme values because they "look wrong" corrupts your analysis. Removing data errors because they "look unusual" is correct and necessary.

4.2 Visual detection

```python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# FIFA 21 dataset has many messy, extreme values
df = pd.read_csv("fifa21_raw.csv")

# Box plots reveal outliers immediately
fig, axes = plt.subplots(1, 3, figsize=(14, 5))

sns.boxplot(y=df["Age"], ax=axes[0])
axes[0].set_title("Age distribution")
```

```
sns.boxplot(y=df["Overall"], ax=axes[1])
axes[1].set_title("Overall rating")

sns.boxplot(y=df["Value(in Euro)"], ax=axes[2])
axes[2].set_title("Market value (Euro)")

plt.tight_layout()
plt.savefig("outlier_boxplots.png", dpi=150)
plt.show()
```

```
# Scatter plot: outliers in 2D space
fig, ax = plt.subplots(figsize=(8, 6))
ax.scatter(df["Age"], df["Overall"], alpha=0.3, s=10)
ax.set_xlabel("Age")
ax.set_ylabel("Overall Rating")
ax.set_title("Age vs Overall - look for isolated points")
plt.tight_layout()
plt.show()
```

4.3 Statistical methods

4.3.1 Z-score method

```
import numpy as np
from scipy import stats

# Z-score: number of standard deviations from the mean
z_scores = np.abs(stats.zscore(df["Overall"].dropna()))
threshold = 3
outliers_z = (z_scores > threshold)
print(f"Outliers by z-score (|z| > {threshold}): {outliers_z.sum()}")
```

```
# Apply z-score to multiple columns
def detect_zscore_outliers(df, columns, threshold=3):
    """Return a boolean mask of rows with any z-score outlier."""
    mask = pd.Series(False, index=df.index)
    for col in columns:
        z = np.abs(stats.zscore(df[col].dropna()))
        col_mask = pd.Series(False, index=df.index)
        col_mask.iloc[:len(z)] = z > threshold
        mask = mask | col_mask
    return mask

num_cols = ["Age", "Overall", "Potential"]
outlier_mask = detect_zscore_outliers(df, num_cols)
print(f"Rows with at least one outlier: {outlier_mask.sum()}")
```

Data tip

The z-score method assumes a roughly normal distribution. For skewed data (income, house prices, player market values), use the IQR method or log-transform first.

4.3.2 IQR method

```
def iqr_bounds(series):
    """Return lower and upper IQR bounds."""
    Q1 = series.quantile(0.25)
    Q3 = series.quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    return lower, upper

lower, upper = iqr_bounds(df["Overall"])
outliers_iqr = df[(df["Overall"] < lower) | (df["Overall"] > upper)]
print(f"IQR bounds: [{lower:.1f}, {upper:.1f}]")
print(f"Outliers: {len(outliers_iqr)}")
```

```
# IQR report for all numerical columns
def iqr_outlier_report(df, columns):
    """Generate an IQR outlier report."""
    rows = []
    for col in columns:
        lo, hi = iqr_bounds(df[col].dropna())
        n_out = ((df[col] < lo) | (df[col] > hi)).sum()
        rows.append({"column": col, "lower": lo, "upper": hi,
                     "n_outliers": n_out,
                     "pct": round(n_out / len(df) * 100, 1)})
    return pd.DataFrame(rows)

report = iqr_outlier_report(df, ["Age", "Overall", "Potential"])
print(report)
```

4.4 Multivariate outlier detection

4.4.1 Mahalanobis distance

```
from scipy.spatial.distance import mahalanobis
from numpy.linalg import inv

def mahalanobis_outliers(df, columns, threshold=3):
    """Detect multivariate outliers using Mahalanobis distance."""
    data = df[columns].dropna()
```

CHAPTER 4. OUTLIER DETECTION AND TREATMENT

```
    mean = data.mean().values
    cov_matrix = data.cov().values
    cov_inv = inv(cov_matrix)

    distances = data.apply(
        lambda row: mahalanobis(row.values, mean, cov_inv), axis=1
    )
    return distances, distances > threshold

cols = ["Age", "Overall", "Potential"]
distances, mask = mahalanobis_outliers(df, cols)
print(f"Multivariate outliers: {mask.sum()}")
```

Preprocessing tip

Mahalanobis distance detects points that are not outliers in any single dimension but are unusual in the *combination* of dimensions. A 40-year-old player is normal. A 95-rated player is normal. A 40-year-old with a 95 rating might be an outlier in the joint space.

4.4.2 Isolation Forest

```
from sklearn.ensemble import IsolationForest

# Isolation Forest: tree-based anomaly detection
iso = IsolationForest(contamination=0.05, random_state=42)
num_data = df[["Age", "Overall", "Potential"]].dropna()
labels = iso.fit_predict(num_data)

# -1 = outlier, 1 = inlier
n_outliers = (labels == -1).sum()
print(f"Isolation Forest outliers: {n_outliers} "
      f"({n_outliers / len(num_data) * 100:.1f}%)")
```

4.5 Domain-specific rules

```
# Air Quality dataset: domain knowledge matters
df_air = pd.read_csv("AirQualityUCI.csv", sep=";", decimal=",")

# The dataset uses -200 as a sentinel for missing sensor data
print(f"Values == -200: {(df_air == -200).sum().sum()}")

# Replace sentinel with NaN
df_air = df_air.replace(-200, np.nan)

# Domain rule: CO concentration > 20 mg/m3 is physically implausible
# at urban monitoring stations
```

```
co_col = "CO(GT)"
implausible = df_air[co_col] > 20
print(f"Implausible CO readings: {implausible.sum()}")
df_air.loc[implausible, co_col] = np.nan
```

Data tip

Domain rules are the most reliable outlier detection method. A z-score does not know that a human cannot weigh 800 kg. Always consult domain experts or reference ranges when available.

4.6 Treatment strategies

```
# Strategy 1: Remove outlier rows
df_clean = df[~outlier_mask].copy()

# Strategy 2: Cap (winsorize) at bounds
from scipy.stats import mstats
df["Overall_winsorized"] = mstats.winsorize(df["Overall"], limits=[0.01, 0.01])

# Strategy 3: Replace with NaN and impute later
df.loc[outlier_mask, "Overall"] = np.nan

# Strategy 4: Log transform to reduce skew
df["Value_log"] = np.log1p(df["Value(in Euro)"].clip(lower=0))

# Strategy 5: Robust scaling (less sensitive to outliers)
from sklearn.preprocessing import RobustScaler
scaler = RobustScaler()  # uses median and IQR instead of mean and std
df[["Overall_robust"]] = scaler.fit_transform(df[["Overall"]])
```

```
# Compare distributions before and after winsorization
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

df["Overall"].hist(bins=50, ax=axes[0], alpha=0.7)
axes[0].set_title("Original")

df["Overall_winsorized"].hist(bins=50, ax=axes[1], alpha=0.7, color="green")
axes[1].set_title("Winsorized (1% each tail)")

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

CHAPTER 4. OUTLIER DETECTION AND TREATMENT

4.7 Exercises

Exercise

1. Load the FIFA 21 dataset. Create box plots for five numerical columns. Identify which columns have the most outliers using the IQR method.

2. For the Air Quality UCI dataset, replace all −200 sentinel values with NaN. Then apply domain-specific rules to flag implausible values for at least two pollutant columns.

3. Compare z-score and IQR outlier counts for a skewed column (e.g., player market value). Which method is more appropriate and why?

4. Use Isolation Forest on the Melbourne Housing dataset (Price, Rooms, Distance, Landsize). Visualize the detected outliers on a scatter plot of Price vs. Landsize.

5. Implement a function `treat_outliers(series, method)` that supports three methods: "remove", "cap", and "log". Test it on a column with known outliers.

4.8 Chapter summary

- Outliers can be errors, artifacts, or genuine extremes — investigation is mandatory.
- Visual methods (box plots, scatter plots) provide immediate intuition about outlier presence.
- Z-score works for normally distributed data; IQR is robust to skewed distributions.
- Mahalanobis distance and Isolation Forest detect multivariate outliers invisible to univariate methods.
- Domain-specific rules (physical limits, sentinel values) are the most reliable detection method.
- Treatment options include removal, capping (winsorization), log transformation, and robust scaling.

Chapter 5

Data type transformations

> *"Machine learning algorithms don't understand categories — they understand numbers."*

5.1 Why transform data types?

Most machine learning algorithms require numerical input. Raw datasets contain categorical variables ("Male", "Female"), ordinal variables ("Low", "Medium", "High"), and numerical variables on vastly different scales (age in years, income in thousands). Proper encoding and scaling are essential.

> **Data tip**
>
> Type transformation has two faces: **encoding** converts categories to numbers, and **scaling** puts numbers on comparable ranges. Both are required for most ML pipelines.

5.2 Categorical encoding

5.2.1 Label (ordinal) encoding

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

# Adult Census dataset
url = ("https://archive.ics.uci.edu/ml/machine-learning-databases/"
       "adult/adult.data")
cols = ["age", "workclass", "fnlwgt", "education", "education_num",
        "marital_status", "occupation", "relationship", "race",
        "sex", "capital_gain", "capital_loss", "hours_per_week",
        "native_country", "income"]
df = pd.read_csv(url, header=None, names=cols, na_values=" ?",
                 skipinitialspace=True)

# Ordinal encoding: for variables with a natural order
edu_order = [["Preschool", "1st-4th", "5th-6th", "7th-8th", "9th",
```

 CHAPTER 5. DATA TYPE TRANSFORMATIONS

```python
              "10th", "11th", "12th", "HS-grad", "Some-college",
              "Assoc-voc", "Assoc-acdm", "Bachelors", "Masters",
              "Prof-school", "Doctorate"]]

enc = OrdinalEncoder(categories=edu_order,
                     handle_unknown="use_encoded_value",
                     unknown_value=-1)
df["education_ord"] = enc.fit_transform(df[["education"]])
print(df[["education", "education_ord"]].drop_duplicates()
        .sort_values("education_ord"))
```

Caution

Ordinal encoding imposes a numeric order. Using it on nominal categories (e.g., country names) implies “France < Germany < Japan”, which is meaningless and can mislead models.

5.2.2 One-hot encoding

```python
from sklearn.preprocessing import OneHotEncoder

# One-hot encoding: for nominal (unordered) categories
ohe = OneHotEncoder(sparse_output=False, drop="first",
                    handle_unknown="ignore")
encoded = ohe.fit_transform(df[["sex", "race"]])
encoded_df = pd.DataFrame(encoded,
                          columns=ohe.get_feature_names_out())
print(encoded_df.head())
print(f"New columns: {encoded_df.shape[1]}")
```

```python
# pandas get_dummies: quick one-hot encoding
df_encoded = pd.get_dummies(df, columns=["sex", "race"],
                            drop_first=True, dtype=int)
print(df_encoded.columns.tolist()[-10:])
```

Preprocessing tip

Use `drop="first"` (or `drop_first=True`) to avoid multicollinearity. With k categories, you only need $k - 1$ dummy variables — the dropped category becomes the reference level.

5.2.3 Target encoding

```python
# Target encoding: replace each category with the mean of the target
# Useful for high-cardinality columns (e.g., 100+ categories)
target_means = df.groupby("occupation")["income"].apply(
```

```
    lambda x: (x == ">50K").mean()
).to_dict()

df["occupation_target_enc"] = df["occupation"].map(target_means)
print(df[["occupation", "occupation_target_enc"]]
        .drop_duplicates().sort_values("occupation_target_enc"))
```

Caution

Target encoding causes data leakage if applied before train/test split. Always fit on training data only, then transform both train and test. Use `sklearn.model_selection.cross_val_predict` or K-fold target encoding to reduce overfitting.

5.2.4 Frequency and binary encoding

```
# Frequency encoding: replace category with its frequency
freq = df["native_country"].value_counts(normalize=True)
df["country_freq"] = df["native_country"].map(freq)
print(df[["native_country", "country_freq"]].head(10))

# Binary encoding: fewer columns than one-hot for high cardinality
# category-encoders library provides this
# pip install category-encoders
# import category_encoders as ce
# encoder = ce.BinaryEncoder(cols=["native_country"])
# df_binary = encoder.fit_transform(df)
```

5.3 Numerical scaling

5.3.1 StandardScaler (z-score normalization)

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
df[["age_scaled", "hours_scaled"]] = scaler.fit_transform(
    df[["age", "hours_per_week"]]
)

print(df[["age", "age_scaled", "hours_per_week", "hours_scaled"]].describe())
```

5.3.2 MinMaxScaler

```
from sklearn.preprocessing import MinMaxScaler
```

```
minmax = MinMaxScaler(feature_range=(0, 1))
df[["age_minmax"]] = minmax.fit_transform(df[["age"]])

# After scaling: min=0, max=1
print(df["age_minmax"].describe())
```

5.3.3 RobustScaler

```
from sklearn.preprocessing import RobustScaler

# Uses median and IQR: robust to outliers
robust = RobustScaler()
df[["capital_gain_robust"]] = robust.fit_transform(df[["capital_gain"]])
print(df["capital_gain_robust"].describe())
```

Preprocessing tip

Choose your scaler based on the data and the model:

- **StandardScaler**: default choice for most algorithms (SVM, logistic regression, neural networks).
- **MinMaxScaler**: when you need bounded values (neural networks with sigmoid output).
- **RobustScaler**: when your data has significant outliers.

5.4 Comparing scalers visually

```
import matplotlib.pyplot as plt
import numpy as np

fig, axes = plt.subplots(1, 4, figsize=(16, 4))

# Original
df["capital_gain"].hist(bins=50, ax=axes[0])
axes[0].set_title("Original")

# Standard
axes[1].hist(StandardScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[1].set_title("StandardScaler")

# MinMax
axes[2].hist(MinMaxScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[2].set_title("MinMaxScaler")
```

```
# Robust
axes[3].hist(RobustScaler().fit_transform(df[["capital_gain"]]),
             bins=50)
axes[3].set_title("RobustScaler")

plt.tight_layout()
plt.savefig("scaler_comparison.png", dpi=150)
plt.show()
```

5.5 Discretization (binning)

```
# Equal-width binning
df["age_bin_width"] = pd.cut(df["age"], bins=5,
                             labels=["very_young", "young", "mid",
                                     "senior", "elderly"])

# Equal-frequency (quantile) binning
df["age_bin_quantile"] = pd.qcut(df["age"], q=5,
                                 labels=["Q1", "Q2", "Q3", "Q4", "Q5"])

# Custom bins from domain knowledge
df["age_group"] = pd.cut(df["age"],
                         bins=[0, 25, 35, 50, 65, 100],
                         labels=["<25", "25-34", "35-49", "50-64", "65+"])

print(df["age_group"].value_counts().sort_index())
```

```
# KBins discretization from sklearn
from sklearn.preprocessing import KBinsDiscretizer

kbd = KBinsDiscretizer(n_bins=5, encode="ordinal", strategy="quantile")
df[["hours_binned"]] = kbd.fit_transform(df[["hours_per_week"]])
print(df[["hours_per_week", "hours_binned"]].head(10))
```

5.6 Power transforms

```
from sklearn.preprocessing import PowerTransformer

# Yeo-Johnson: works with positive and negative values
pt = PowerTransformer(method="yeo-johnson")
df[["capital_gain_yj"]] = pt.fit_transform(df[["capital_gain"]])

# Box-Cox: only works with strictly positive values
# pt_bc = PowerTransformer(method="box-cox")

# Log transform: manual alternative for skewed data
```

CHAPTER 5. DATA TYPE TRANSFORMATIONS

```
df["capital_gain_log"] = np.log1p(df["capital_gain"])

print(f"Original skew:    {df['capital_gain'].skew():.2f}")
print(f"Yeo-Johnson skew: {df['capital_gain_yj'].skew():.2f}")
print(f"Log skew:         {df['capital_gain_log'].skew():.2f}")
```

5.7 Exercises

Exercise

1. Load the Titanic dataset. Apply one-hot encoding to `Sex` and `Embarked`. Apply ordinal encoding to `Pclass` (1st > 2nd > 3rd). Show the resulting DataFrame.

2. For the Adult Census dataset, compare the effect of StandardScaler, MinMaxScaler, and RobustScaler on `capital_gain`. Plot all three distributions.

3. Implement target encoding for `native_country` using 5-fold cross-validation to prevent leakage. Compare the encoded values with naive target encoding.

4. Create age groups for the Titanic dataset using domain-meaningful bins (child, teenager, young adult, middle-aged, senior). Compute survival rate by age group.

5. Apply a Yeo-Johnson power transform to all numerical columns in the Adult dataset. Compare skewness before and after.

5.8 Chapter summary

- Categorical encoding converts non-numeric data to numbers: use ordinal encoding for ordered categories, one-hot for nominal, and target encoding for high-cardinality features.

- Scaling puts numerical features on comparable ranges: StandardScaler for general use, MinMaxScaler for bounded ranges, RobustScaler when outliers are present.

- Discretization (binning) converts continuous variables into categories, useful for non-linear relationships and interpretability.

- Power transforms (Yeo-Johnson, Box-Cox, log) reduce skewness and can improve model performance.

- Always fit transformers on training data only, then apply to test data.

Chapter 6

Feature engineering

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

6.1 What is feature engineering?

Feature engineering is the process of creating new input variables from existing data to improve model performance. While encoding and scaling (Chapter 5) transform existing features, feature engineering *creates* new ones that capture patterns the raw data does not explicitly represent.

Data tip

A well-engineered feature can be worth more than a complex model. A simple linear regression with the right features often outperforms a deep neural network with raw features.

6.2 Polynomial and interaction features

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Melbourne Housing dataset
df = pd.read_csv("melb_data.csv")
df = df.dropna(subset=["Price", "Rooms", "Distance", "Landsize"])

# Polynomial features: capture non-linear relationships
poly = PolynomialFeatures(degree=2, interaction_only=False,
                          include_bias=False)
features = df[["Rooms", "Distance"]]
poly_features = poly.fit_transform(features)
poly_names = poly.get_feature_names_out(["Rooms", "Distance"])

poly_df = pd.DataFrame(poly_features, columns=poly_names)
```

CHAPTER 6. FEATURE ENGINEERING

```
print(poly_df.head())
print(f"Original features: {features.shape[1]}")
print(f"Polynomial features: {poly_df.shape[1]}")
```

```
# Interaction only: no squared terms, just cross-products
poly_inter = PolynomialFeatures(degree=2, interaction_only=True,
                                include_bias=False)
inter_features = poly_inter.fit_transform(features)
inter_names = poly_inter.get_feature_names_out(["Rooms", "Distance"])
print(f"Interaction features: {inter_names}")
```

Caution

Polynomial features grow combinatorially: n features at degree d produce $\binom{n+d}{d} - 1$ features. With 20 features at degree 3, you get 1770 features. Use this technique selectively, not on the entire feature set.

6.3 Mathematical transformations

```
# Ratio features: often more informative than raw values
df["price_per_room"] = df["Price"] / df["Rooms"]
df["price_per_sqm"] = df["Price"] / df["Landsize"].replace(0, np.nan)
df["rooms_per_bathroom"] = df["Rooms"] / df["Bathroom"].replace(0, 1)

print(df[["Price", "Rooms", "price_per_room",
          "price_per_sqm"]].describe())
```

```
# Log transform for skewed features
df["log_price"] = np.log1p(df["Price"])
df["log_landsize"] = np.log1p(df["Landsize"])

# Square root: gentler than log
df["sqrt_distance"] = np.sqrt(df["Distance"])

# Compare correlation with target
for col in ["Distance", "sqrt_distance", "log_price"]:
    if col in df.columns and df[col].notnull().sum() > 0:
        corr = df[col].corr(df["Price"])
        print(f"Corr({col}, Price) = {corr:.3f}")
```

Preprocessing tip

Ratio features encode *domain knowledge* in a single number. “Price per square meter” is a standard real estate metric that captures value density better than price or area alone.

6.4 Date and time features

```
# Melbourne Housing: extract date components
df["Date"] = pd.to_datetime(df["Date"], dayfirst=True)

df["sale_year"] = df["Date"].dt.year
df["sale_month"] = df["Date"].dt.month
df["sale_dayofweek"] = df["Date"].dt.dayofweek  # 0=Monday
df["sale_quarter"] = df["Date"].dt.quarter
df["sale_is_weekend"] = df["Date"].dt.dayofweek.isin([5, 6]).astype(int)

print(df[["Date", "sale_year", "sale_month",
          "sale_dayofweek", "sale_quarter"]].head())
```

```
# Cyclical encoding for month (so December and January are close)
df["month_sin"] = np.sin(2 * np.pi * df["sale_month"] / 12)
df["month_cos"] = np.cos(2 * np.pi * df["sale_month"] / 12)

# Days since a reference date
reference = pd.Timestamp("2016-01-01")
df["days_since_ref"] = (df["Date"] - reference).dt.days

print(df[["sale_month", "month_sin", "month_cos"]].head(12))
```

6.5 Aggregation features

```
# Suburb-level statistics as features
suburb_stats = df.groupby("Suburb")["Price"].agg(
    suburb_mean_price="mean",
    suburb_median_price="median",
    suburb_price_std="std",
    suburb_n_sales="count"
).reset_index()

df = df.merge(suburb_stats, on="Suburb", how="left")

# How does this property compare to its suburb average?
df["price_vs_suburb"] = df["Price"] / df["suburb_mean_price"]

print(df[["Suburb", "Price", "suburb_mean_price",
          "price_vs_suburb"]].head(10))
```

```
# Gapminder: country-level features over time
gap = pd.read_csv("gapminder.csv")

# Year-over-year growth rate
```

CHAPTER 6. FEATURE ENGINEERING

```
gap = gap.sort_values(["country", "year"])
gap["gdp_growth"] = gap.groupby("country")["gdpPercap"].pct_change()
gap["pop_growth"] = gap.groupby("country")["pop"].pct_change()

# Rolling average (smoothed trend)
gap["lifeExp_rolling3"] = (gap.groupby("country")["lifeExp"]
                               .transform(lambda x: x.rolling(3).mean()))

print(gap[gap["country"] == "Benin"][
    ["year", "lifeExp", "lifeExp_rolling3", "gdp_growth"]].tail(6))
```

6.6 Text-derived features

```
# Extract features from text columns without full NLP
# Using Melbourne suburb names and addresses

# Length of address
df["address_length"] = df["Address"].str.len()
df["address_word_count"] = df["Address"].str.split().str.len()

# Contains specific keywords
df["is_street"] = df["Address"].str.contains("St$", regex=True).astype(int)
df["is_road"] = df["Address"].str.contains("Rd$", regex=True).astype(int)
df["is_avenue"] = df["Address"].str.contains("Av", regex=False).astype(int)

print(df[["Address", "address_length", "is_street",
          "is_road"]].head(10))
```

6.7 Domain knowledge features

```
# Real estate domain knowledge
df["total_rooms"] = df["Rooms"] + df["Bathroom"] + df["Car"]
df["has_multiple_bathrooms"] = (df["Bathroom"] > 1).astype(int)
df["is_new_build"] = (df["YearBuilt"] > 2010).astype(int)
df["building_age"] = df["sale_year"] - df["YearBuilt"]
df["is_close_to_cbd"] = (df["Distance"] < 10).astype(int)

# Property type indicators
df["is_house"] = (df["Type"] == "h").astype(int)
df["is_unit"] = (df["Type"] == "u").astype(int)

# Interaction: large house close to CBD
df["large_close"] = df["is_house"] * df["is_close_to_cbd"] * df["Rooms"]

print(df[["Suburb", "Type", "Rooms", "Distance",
          "building_age", "large_close"]].head(10))
```

Data tip

The best features come from understanding the domain. Spend time with domain experts. Ask: "What factors do professionals in this field consider when making decisions?" Encode those factors as features.

6.8 Feature selection after engineering

```
from sklearn.feature_selection import mutual_info_regression

# Select numerical features
feature_cols = ["Rooms", "Distance", "Landsize", "building_age",
                "price_per_room", "suburb_mean_price", "large_close",
                "days_since_ref", "Bathroom", "Car"]
target = "Price"

# Drop rows with NaN for this analysis
subset = df[feature_cols + [target]].dropna()

mi = mutual_info_regression(subset[feature_cols], subset[target],
                            random_state=42)
mi_series = pd.Series(mi, index=feature_cols).sort_values(ascending=False)

print("Mutual information with Price:")
print(mi_series)
```

```
# Correlation matrix for engineered features
import seaborn as sns
import matplotlib.pyplot as plt

corr = subset[feature_cols].corr()
fig, ax = plt.subplots(figsize=(10, 8))
sns.heatmap(corr, annot=True, fmt=".2f", cmap="coolwarm",
            center=0, ax=ax)
plt.title("Feature correlation matrix")
plt.tight_layout()
plt.savefig("feature_correlation.png", dpi=150)
plt.show()
```

6.9 Exercises

Exercise

1. For the Melbourne Housing dataset, create at least 5 new features using domain knowledge. Rank them by mutual information with Price.

2. Create cyclical encoding for the sale month. Verify that December and Jan-

uary are close in the encoded space by computing their Euclidean distance.

3. Using the Gapminder dataset, create year-over-year growth rate features for GDP per capita and life expectancy. Which country had the highest GDP growth in a single period?

4. Build polynomial features (degree 2) for `Rooms`, `Distance`, and `Landsize`. Train a linear regression with and without polynomial features and compare R^2.

5. Create a feature engineering function `engineer_melb(df)` that takes the raw Melbourne dataset and returns a DataFrame with at least 10 new features, ready for modeling.

6.10 Chapter summary

- Feature engineering creates new variables that capture patterns invisible in raw data.
- Polynomial and interaction features capture non-linear and cross-feature relationships.
- Ratio features (price per room, price per square meter) encode domain knowledge compactly.
- Date features (year, month, day of week, cyclical encoding) unlock temporal patterns.
- Aggregation features (group means, rolling averages) add contextual information.
- Domain knowledge is the most valuable source of feature ideas — consult experts.
- After engineering, use mutual information or correlation to select the most predictive features.

Chapter 7

Text preprocessing

“Natural language is the most unstructured data of all.”

7.1 Why text needs preprocessing

Text data cannot be fed directly into numerical models. It must be cleaned, normalized, tokenized, and converted to numerical representations. The quality of text preprocessing directly determines the quality of any downstream NLP task.

Data tip

Text preprocessing is language-dependent. Tokenization rules, stopwords, and stemming algorithms differ between English and French. This chapter covers both languages where relevant.

7.2 Loading text datasets

```
from sklearn.datasets import fetch_20newsgroups

# 20 Newsgroups: classic text classification dataset
newsgroups = fetch_20newsgroups(subset="train",
                                remove=("headers", "footers", "quotes"))

print(f"Number of documents: {len(newsgroups.data)}")
print(f"Number of categories: {len(newsgroups.target_names)}")
print(f"\nCategories: {newsgroups.target_names}")
print(f"\nSample document:\n{newsgroups.data[0][:300]}")
```

```
import pandas as pd

# Create a DataFrame for easier manipulation
df = pd.DataFrame({
    "text": newsgroups.data,
    "category": [newsgroups.target_names[i] for i in newsgroups.target]
```

 CHAPTER 7. TEXT PREPROCESSING

```
})
print(df["category"].value_counts().head(10))
```

7.3 Text cleaning

```python
import re

def clean_text(text):
    """Basic text cleaning pipeline."""
    # Lowercase
    text = text.lower()
    # Remove email addresses
    text = re.sub(r'\S+@\S+', '', text)
    # Remove URLs
    text = re.sub(r'http\S+|www\.\S+', '', text)
    # Remove numbers (optional: sometimes numbers matter)
    text = re.sub(r'\d+', '', text)
    # Remove special characters, keep letters and spaces
    text = re.sub(r'[^a-z\s]', '', text)
    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()
    return text

# Apply to dataset
df["text_clean"] = df["text"].apply(clean_text)
print(df[["text", "text_clean"]].iloc[0])
```

Preprocessing tip

Text cleaning is order-dependent. Always lowercase *before* removing patterns. Always remove URLs and emails *before* removing special characters (otherwise you destroy the patterns you need to match).

7.4 Tokenization

```python
import nltk
nltk.download("punkt_tab", quiet=True)
from nltk.tokenize import word_tokenize, sent_tokenize

text = "Dr. Smith's analysis shows that 42% of patients improved. This is
↪ significant!"

# Sentence tokenization
sentences = sent_tokenize(text)
print(f"Sentences: {sentences}")

# Word tokenization
```

```
tokens = word_tokenize(text)
print(f"Tokens: {tokens}")
print(f"Number of tokens: {len(tokens)}")
```

```
# Simple whitespace tokenization vs NLTK
text_fr = "L'analyse du Dr. N'Guessan montre que c'est significatif."

# Whitespace split: misses contractions
simple_tokens = text_fr.split()
print(f"Simple: {simple_tokens}")

# NLTK: handles contractions and abbreviations
nltk_tokens = word_tokenize(text_fr, language="french")
print(f"NLTK:   {nltk_tokens}")
```

7.5 Stopword removal

```
nltk.download("stopwords", quiet=True)
from nltk.corpus import stopwords

# English stopwords
stop_en = set(stopwords.words("english"))
print(f"English stopwords ({len(stop_en)}): "
      f"{sorted(list(stop_en))[:15]}...")

# French stopwords
stop_fr = set(stopwords.words("french"))
print(f"French stopwords ({len(stop_fr)}): "
      f"{sorted(list(stop_fr))[:15]}...")

# Remove stopwords from tokens
tokens = ["the", "patient", "showed", "significant", "improvement",
          "in", "blood", "pressure", "after", "the", "treatment"]
filtered = [t for t in tokens if t not in stop_en]
print(f"Before: {tokens}")
print(f"After:  {filtered}")
```

⚠ Caution

Removing stopwords is not always beneficial. In sentiment analysis, “not good” becomes “good” after removing “not”. In topic modeling, stopword removal usually helps. Always evaluate the impact on your specific task.

7.6 Stemming and lemmatization

```
from nltk.stem import PorterStemmer, SnowballStemmer
```

```
nltk.download("wordnet", quiet=True)
from nltk.stem import WordNetLemmatizer

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

words = ["running", "runs", "ran", "studies", "studying",
         "better", "preprocessing", "processed"]

print(f"{'Word':<15} {'Stem':<15} {'Lemma':<15}")
print("-" * 45)
for w in words:
    print(f"{w:<15} {stemmer.stem(w):<15} {lemmatizer.lemmatize(w, 'v'):<15}")
```

```
# French stemming with Snowball
stemmer_fr = SnowballStemmer("french")
mots = ["traitements", "traiter", "trait\u00e9es", "am\u00e9lioration",
        "am\u00e9liorer", "analyse", "analyses", "analyser"]

print("\nFrench stemming:")
for m in mots:
    print(f"  {m:<20} -> {stemmer_fr.stem(m)}")
```

Preprocessing tip

Stemming is fast but aggressive: "university" and "universe" may share the same stem. Lemmatization is slower but linguistically accurate: it returns real dictionary words. For most NLP pipelines, lemmatization is preferred.

7.7 Complete text preprocessing pipeline

```
def preprocess_text(text, language="english"):
    """Full text preprocessing pipeline."""
    # 1. Clean
    text = text.lower()
    text = re.sub(r'\S+@\S+', '', text)
    text = re.sub(r'http\S+|www\.\S+', '', text)
    text = re.sub(r'[^a-z\s\u00e0-\u00ff]', '', text)  # keep accented chars
    text = re.sub(r'\s+', ' ', text).strip()

    # 2. Tokenize
    tokens = word_tokenize(text, language=language)

    # 3. Remove stopwords
    stop = set(stopwords.words(language))
    tokens = [t for t in tokens if t not in stop and len(t) > 2]

    # 4. Lemmatize
```

```
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(t) for t in tokens]

    return tokens

# Apply to the full dataset
df["tokens"] = df["text"].apply(preprocess_text)
df["n_tokens"] = df["tokens"].apply(len)
print(df[["category", "n_tokens"]].groupby("category")["n_tokens"].mean()
        .sort_values(ascending=False).head(10))
```

7.8 TF-IDF vectorization

```
from sklearn.feature_extraction.text import TfidfVectorizer

# TF-IDF: Term Frequency - Inverse Document Frequency
tfidf = TfidfVectorizer(max_features=5000,
                        stop_words="english",
                        min_df=5, max_df=0.7,
                        ngram_range=(1, 2))

X_tfidf = tfidf.fit_transform(df["text_clean"])
print(f"TF-IDF matrix shape: {X_tfidf.shape}")
print(f"Vocabulary size: {len(tfidf.vocabulary_)}")

# Top terms by TF-IDF score for first document
feature_names = tfidf.get_feature_names_out()
doc_tfidf = X_tfidf[0].toarray().flatten()
top_idx = doc_tfidf.argsort()[-10:][::-1]
print("\nTop TF-IDF terms for document 0:")
for idx in top_idx:
    print(f"  {feature_names[idx]}: {doc_tfidf[idx]:.4f}")
```

```
# Bag of Words (simpler alternative)
from sklearn.feature_extraction.text import CountVectorizer

bow = CountVectorizer(max_features=3000, stop_words="english")
X_bow = bow.fit_transform(df["text_clean"])
print(f"BoW matrix shape: {X_bow.shape}")
```

7.9 Regular expressions for text extraction

```
import re

# Extract structured data from unstructured text
texts = [
```

 CHAPTER 7. TEXT PREPROCESSING

```
    "Patient BP: 120/80 mmHg, HR: 72 bpm",
    "Lab result: glucose 142 mg/dL, HbA1c 7.2%",
    "Contact: dr.smith@hospital.org, Tel: +1-555-0123",
]

# Extract blood pressure
bp_pattern = r'(\d{2,3})/(\d{2,3})\s*mmHg'
for t in texts:
    match = re.search(bp_pattern, t)
    if match:
        print(f"Systolic: {match.group(1)}, Diastolic: {match.group(2)}")

# Extract all numbers with units
num_unit = r'(\d+\.?\d*)\s*(mg/dL|mmHg|bpm|%)'
for t in texts:
    matches = re.findall(num_unit, t)
    for value, unit in matches:
        print(f"  {value} {unit}")
```

7.10 Exercises

Exercise

1. Load the 20 Newsgroups dataset. Build a complete preprocessing pipeline (clean, tokenize, remove stopwords, lemmatize). Compute the vocabulary size before and after preprocessing.

2. Create a TF-IDF matrix from the 20 Newsgroups dataset. Find the top 10 terms for the categories "sci.space" and "rec.sport.baseball". Are they meaningful?

3. Write a regex pattern that extracts all email addresses from a text. Test it on the raw (unprocessed) 20 Newsgroups data.

4. Compare Snowball stemming for English and French on a bilingual document. Show cases where stemming produces incorrect roots.

5. Build a text preprocessing pipeline for French text. Use French stopwords and Snowball French stemmer. Test on 5 sample French sentences about data science.

7.11 Chapter summary

- Text preprocessing converts unstructured text into numerical features for machine learning.
- The standard pipeline is: clean (lowercase, remove noise) → tokenize → remove stopwords → stem/lemmatize → vectorize.
- Stopword removal helps topic modeling but can hurt sentiment analysis.

- Lemmatization produces real words and is generally preferred over stemming.
- TF-IDF weights terms by importance (frequent in a document, rare across documents).
- Regular expressions extract structured data (numbers, emails, patterns) from unstructured text.
- Multilingual preprocessing requires language-specific tokenizers, stopword lists, and stemmers.

CHAPTER 7. TEXT PREPROCESSING

Chapter 8

Time series and temporal data

> *"Time is the most important feature in a time series — and the most often mishandled."*

8.1 Datetime parsing

The first step with temporal data is converting strings to proper datetime objects. Pandas provides robust parsing, but real-world date formats are surprisingly inconsistent.

```python
import pandas as pd
import numpy as np

# Jena Climate dataset: 10-minute weather measurements
# Download from: https://www.kaggle.com/datasets/mnassrib/jena-climate
df = pd.read_csv("jena_climate_2009_2016.csv")
print(df.head())
print(f"\nDate column dtype: {df['Date Time'].dtype}")
```

```python
# Parse datetime
df["datetime"] = pd.to_datetime(df["Date Time"], format="%d.%m.%Y %H:%M:%S")
df = df.set_index("datetime").drop(columns=["Date Time"])

print(f"Date range: {df.index.min()} to {df.index.max()}")
print(f"Frequency: ~{(df.index[1] - df.index[0])}")
print(f"Total records: {len(df):,}")
```

Caution

Date parsing ambiguity: is "01/02/2024" January 2 (US) or February 1 (Europe)? Always specify the format explicitly with the `format` parameter. Never rely on automatic inference for production code.

8.1.1 Handling multiple date formats

```python
# When a column has mixed date formats
```

CHAPTER 8. TIME SERIES AND TEMPORAL DATA

```
messy_dates = pd.Series(["2024-01-15", "15/01/2024", "Jan 15, 2024",
                         "15-Jan-2024", "20240115"])

# pd.to_datetime with infer_datetime_format handles many cases
parsed = pd.to_datetime(messy_dates, format="mixed", dayfirst=True)
print(parsed)

# For truly messy data, use a custom parser
def parse_flexible(date_str):
    """Try multiple date formats."""
    formats = ["%Y-%m-%d", "%d/%m/%Y", "%b %d, %Y",
               "%d-%b-%Y", "%Y%m%d"]
    for fmt in formats:
        try:
            return pd.to_datetime(date_str, format=fmt)
        except (ValueError, TypeError):
            continue
    return pd.NaT
```

8.2 Resampling

```
# Downsample: 10-minute data to hourly
hourly = df.resample("h").mean()
print(f"10-min records: {len(df):,}")
print(f"Hourly records: {len(hourly):,}")

# Downsample to daily
daily = df.resample("D").agg({
    "T (degC)": ["mean", "min", "max"],
    "p (mbar)": "mean",
    "rh (%)": "mean",
})
daily.columns = ["temp_mean", "temp_min", "temp_max",
                 "pressure_mean", "humidity_mean"]
print(daily.head())
```

```
# Upsample: daily to hourly (with interpolation)
daily_sample = daily.head(30)
hourly_upsampled = daily_sample.resample("h").interpolate(method="linear")
print(f"Daily records: {len(daily_sample)}")
print(f"Hourly (upsampled): {len(hourly_upsampled)}")
```

Preprocessing tip

When downsampling, think carefully about the aggregation function. Temperature should use mean, precipitation should use sum, and minimum/maximum readings should use min/max respectively. Using the wrong aggregation produces meaningless results.

8.3 Lag features

```
# Lag features: use past values as predictors
daily["temp_lag1"] = daily["temp_mean"].shift(1)   # yesterday
daily["temp_lag7"] = daily["temp_mean"].shift(7)   # last week
daily["temp_lag30"] = daily["temp_mean"].shift(30)  # last month

# Change features
daily["temp_change_1d"] = daily["temp_mean"].diff(1)
daily["temp_change_7d"] = daily["temp_mean"].diff(7)

# Percentage change
daily["temp_pct_change"] = daily["temp_mean"].pct_change()

print(daily[["temp_mean", "temp_lag1", "temp_lag7",
             "temp_change_1d"]].head(10))
```

```
# Autocorrelation: check if lag features are useful
import matplotlib.pyplot as plt
from pandas.plotting import autocorrelation_plot

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Autocorrelation plot
autocorrelation_plot(daily["temp_mean"].dropna(), ax=axes[0])
axes[0].set_title("Temperature autocorrelation")
axes[0].set_xlim(0, 400)

# Partial autocorrelation (useful for determining lag order)
from statsmodels.graphics.tsaplots import plot_pacf
plot_pacf(daily["temp_mean"].dropna(), lags=50, ax=axes[1])
axes[1].set_title("Partial autocorrelation")

plt.tight_layout()
plt.savefig("autocorrelation.png", dpi=150)
plt.show()
```

Caution

Lag features introduce NaN values at the beginning of the series. A lag-30 feature has 30 missing values. Always account for this by either dropping the initial rows or using forward-fill imputation.

8.4 Rolling statistics

```
# Rolling (moving) statistics smooth out noise
daily["temp_rolling7"] = daily["temp_mean"].rolling(window=7).mean()
daily["temp_rolling30"] = daily["temp_mean"].rolling(window=30).mean()
daily["temp_rolling_std7"] = daily["temp_mean"].rolling(window=7).std()

# Exponentially weighted moving average (more recent = more weight)
daily["temp_ewm7"] = daily["temp_mean"].ewm(span=7).mean()

print(daily[["temp_mean", "temp_rolling7", "temp_rolling30",
             "temp_ewm7"]].tail(10))
```

```
# Visualize rolling statistics
fig, ax = plt.subplots(figsize=(14, 6))

daily["temp_mean"].plot(ax=ax, alpha=0.3, label="Daily mean")
daily["temp_rolling7"].plot(ax=ax, label="7-day rolling mean")
daily["temp_rolling30"].plot(ax=ax, label="30-day rolling mean",
                             linewidth=2)

ax.set_ylabel("Temperature (degC)")
ax.set_title("Jena Climate: Temperature with rolling averages")
ax.legend()
plt.tight_layout()
plt.savefig("rolling_stats.png", dpi=150)
plt.show()
```

8.5 Trend and seasonality decomposition

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Decompose into trend + seasonal + residual
# Use monthly data for cleaner decomposition
monthly = daily["temp_mean"].resample("ME").mean()

decomposition = seasonal_decompose(monthly, model="additive", period=12)

fig = decomposition.plot()
fig.set_size_inches(14, 10)
plt.tight_layout()
plt.savefig("decomposition.png", dpi=150)
plt.show()
```

```
# Extract components as features
monthly_df = pd.DataFrame({
```

```
    "observed": decomposition.observed,
    "trend": decomposition.trend,
    "seasonal": decomposition.seasonal,
    "residual": decomposition.resid,
})
print(monthly_df.head(15))
```

Preprocessing tip

The decomposition residual is what remains after removing trend and seasonality. A large residual indicates an unusual observation — it can serve as an outlier detection mechanism for time series.

8.6 Air Quality case study

```
# Air Quality UCI dataset: hourly pollutant measurements
df_air = pd.read_csv("AirQualityUCI.csv", sep=";", decimal=",",
                     parse_dates={"datetime": ["Date", "Time"]},
                     dayfirst=True)

# Replace sentinel values
df_air = df_air.replace(-200, np.nan)

# Select key columns
cols = ["datetime", "CO(GT)", "NO2(GT)", "T", "RH"]
df_air = df_air[cols].set_index("datetime").dropna(how="all")

# Resample to daily
air_daily = df_air.resample("D").mean()

# Create temporal features
air_daily["hour_of_day_peak"] = df_air["CO(GT)"].resample("D").idxmax().dt.hour
air_daily["co_lag1"] = air_daily["CO(GT)"].shift(1)
air_daily["co_rolling7"] = air_daily["CO(GT)"].rolling(7).mean()
air_daily["co_weekend"] = air_daily.index.dayofweek.isin([5, 6]).astype(int)

print(air_daily.head(10))
```

8.7 Datetime feature extraction summary

```
def extract_datetime_features(df, date_col):
    """Extract a comprehensive set of datetime features."""
    dt = df[date_col] if date_col in df.columns else df.index
    features = pd.DataFrame(index=df.index)

    features["year"] = dt.year
    features["month"] = dt.month
```

CHAPTER 8. TIME SERIES AND TEMPORAL DATA

```
    features["day"] = dt.day
    features["dayofweek"] = dt.dayofweek
    features["hour"] = dt.hour if hasattr(dt, "hour") else 0
    features["quarter"] = dt.quarter
    features["is_weekend"] = dt.dayofweek.isin([5, 6]).astype(int)
    features["day_of_year"] = dt.dayofyear
    features["week_of_year"] = dt.isocalendar().week.astype(int)

    # Cyclical encoding
    features["month_sin"] = np.sin(2 * np.pi * dt.month / 12)
    features["month_cos"] = np.cos(2 * np.pi * dt.month / 12)
    features["dow_sin"] = np.sin(2 * np.pi * dt.dayofweek / 7)
    features["dow_cos"] = np.cos(2 * np.pi * dt.dayofweek / 7)

    return features
```

8.8 Exercises

Exercise

1. Load the Jena Climate dataset. Parse the datetime column and resample to daily frequency. Plot the daily mean temperature for 2015.
2. Create lag features (1, 7, 14, 30 days) for temperature. Compute the correlation of each lag feature with the current temperature. Which lag has the highest correlation?
3. Apply seasonal decomposition to the Air Quality CO data. Is there a clear seasonal pattern? What does the trend tell you?
4. Build a comprehensive feature extraction function for the Jena Climate dataset that creates: lag features, rolling statistics, cyclical time encodings, and trend indicators. Apply it and report the resulting number of features.
5. Using the Air Quality dataset, investigate whether CO levels are significantly different on weekdays vs. weekends. Create appropriate visualizations.

8.9 Chapter summary

- Always parse dates explicitly with a known format; never rely on automatic inference in production.
- Resampling changes temporal resolution: downsampling aggregates, upsampling interpolates.
- Lag features capture temporal dependencies: yesterday's temperature predicts today's.
- Rolling statistics (moving average, moving std) smooth noise and reveal trends.

- Seasonal decomposition separates trend, seasonality, and residuals — each can serve as a feature.
- Cyclical encoding (sine/cosine) ensures that December is close to January in feature space.

CHAPTER 8. TIME SERIES AND TEMPORAL DATA

Chapter 9

Pipelines and automation

"If you do it twice, automate it." — Every software engineer

9.1 Why pipelines?

Manual preprocessing is error-prone and leaky. Common mistakes include:

- Fitting a scaler on the full dataset (including test data) — data leakage.
- Forgetting to apply the same transformations at prediction time.
- Applying transformations in a different order during training and inference.

Scikit-learn's `Pipeline` and `ColumnTransformer` solve all three problems by packaging the entire preprocessing workflow into a single, reproducible object.

Data tip

A pipeline guarantees that every transformation applied during training is applied identically during prediction, in the same order, with the same fitted parameters.

9.2 Basic Pipeline

```
import pandas as pd
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# Adult Census dataset
url = ("https://archive.ics.uci.edu/ml/machine-learning-databases/"
       "adult/adult.data")
cols = ["age", "workclass", "fnlwgt", "education", "education_num",
        "marital_status", "occupation", "relationship", "race",
        "sex", "capital_gain", "capital_loss", "hours_per_week",
```

CHAPTER 9. PIPELINES AND AUTOMATION

```
        "native_country", "income"]
df = pd.read_csv(url, header=None, names=cols, na_values=" ?",
                 skipinitialspace=True)

# Simple numerical pipeline
num_pipe = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

# Fit and transform
num_cols = ["age", "education_num", "capital_gain",
            "capital_loss", "hours_per_week"]
X_num = num_pipe.fit_transform(df[num_cols])
print(f"Shape: {X_num.shape}")
print(f"Means (should be ~0): {X_num.mean(axis=0).round(2)}")
```

9.3 ColumnTransformer

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

# Define column groups
num_features = ["age", "education_num", "capital_gain",
                "capital_loss", "hours_per_week"]
cat_features = ["workclass", "marital_status", "occupation",
                "relationship", "race", "sex"]

# Build transformers for each type
num_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="median")),
    ("scaler", StandardScaler()),
])

cat_transformer = Pipeline([
    ("imputer", SimpleImputer(strategy="most_frequent")),
    ("encoder", OneHotEncoder(handle_unknown="ignore",
                              sparse_output=False)),
])

# Combine with ColumnTransformer
preprocessor = ColumnTransformer([
    ("num", num_transformer, num_features),
    ("cat", cat_transformer, cat_features),
])

# Fit and transform
X = preprocessor.fit_transform(df)
print(f"Input shape: {df[num_features + cat_features].shape}")
```

```
print(f"Output shape: {X.shape}")
```

Preprocessing tip

Use `ColumnTransformer` to apply different transformations to different column types in a single step. This is the standard approach in production ML systems.

9.4 Full pipeline with a model

```
from sklearn.metrics import accuracy_score, classification_report

# Prepare target
y = (df["income"] == ">50K").astype(int)

# Full pipeline: preprocessing + model
full_pipeline = Pipeline([
    ("preprocessor", preprocessor),
    ("classifier", LogisticRegression(max_iter=1000, random_state=42)),
])

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    df[num_features + cat_features], y,
    test_size=0.2, random_state=42, stratify=y
)

# Fit the entire pipeline
full_pipeline.fit(X_train, y_train)

# Predict
y_pred = full_pipeline.predict(X_test)
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
print(classification_report(y_test, y_pred))
```

```
# Cross-validation with the pipeline (no leakage!)
from sklearn.model_selection import cross_val_score

scores = cross_val_score(full_pipeline,
                         df[num_features + cat_features], y,
                         cv=5, scoring="accuracy")
print(f"CV Accuracy: {scores.mean():.4f} +/- {scores.std():.4f}")
```

Caution

If you scale your data *before* the train/test split, information from the test set leaks into the scaler's mean and standard deviation. By placing the scaler inside a pipeline, scikit-learn ensures it only fits on training data during cross-validation.

CHAPTER 9. PIPELINES AND AUTOMATION

9.5 Custom transformers

```
from sklearn.base import BaseEstimator, TransformerMixin

class MissingIndicator(BaseEstimator, TransformerMixin):
    """Add binary columns indicating which values were missing."""

    def fit(self, X, y=None):
        self.columns_ = X.columns if hasattr(X, "columns") else None
        return self

    def transform(self, X):
        X = pd.DataFrame(X, columns=self.columns_)
        indicators = X.isnull().astype(int)
        indicators.columns = [f"{c}_missing" for c in indicators.columns]
        return pd.concat([X, indicators], axis=1)


class LogTransformer(BaseEstimator, TransformerMixin):
    """Apply log1p transformation to specified columns."""

    def __init__(self, columns=None):
        self.columns = columns

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X = X.copy()
        cols = self.columns or X.columns
        for col in cols:
            X[col] = np.log1p(X[col].clip(lower=0))
        return X
```

```
# Use custom transformers in a pipeline
custom_pipe = Pipeline([
    ("missing_ind", MissingIndicator()),
    ("imputer", SimpleImputer(strategy="median")),
    ("log", LogTransformer(columns=["capital_gain", "capital_loss"])),
    ("scaler", StandardScaler()),
])

# Test it
sample = df[num_features].head(20)
result = custom_pipe.fit_transform(sample)
print(f"Input columns: {len(num_features)}")
print(f"Output columns: {result.shape[1]}")
```

9.6 Saving and loading pipelines

```
import joblib

# Save the fitted pipeline
joblib.dump(full_pipeline, "income_pipeline.joblib")
print("Pipeline saved.")

# Load it later (e.g., in a web app or batch job)
loaded_pipeline = joblib.load("income_pipeline.joblib")

# Predict on new data
new_data = pd.DataFrame({
    "age": [35], "education_num": [13],
    "capital_gain": [0], "capital_loss": [0],
    "hours_per_week": [40], "workclass": ["Private"],
    "marital_status": ["Married-civ-spouse"],
    "occupation": ["Exec-managerial"],
    "relationship": ["Husband"], "race": ["White"],
    "sex": ["Male"],
})

prediction = loaded_pipeline.predict(new_data)
print(f"Prediction: {'> 50K' if prediction[0] else '<= 50K'}")
```

Preprocessing tip

A saved pipeline contains all fitted parameters (scaler means, encoder categories, imputer statistics). This single file is everything you need to preprocess new data identically to training data.

9.7 Melbourne Housing full pipeline

```
# Complete pipeline for the Melbourne Housing dataset
melb = pd.read_csv("melb_data.csv")
melb = melb.dropna(subset=["Price"])

num_features_melb = ["Rooms", "Distance", "Landsize", "BuildingArea",
                     "YearBuilt", "Car", "Bathroom"]
cat_features_melb = ["Type", "Method", "Regionname"]

melb_preprocessor = ColumnTransformer([
    ("num", Pipeline([
        ("imputer", SimpleImputer(strategy="median")),
        ("scaler", StandardScaler()),
    ]), num_features_melb),
    ("cat", Pipeline([
        ("imputer", SimpleImputer(strategy="most_frequent")),
```

CHAPTER 9. *PIPELINES AND AUTOMATION*

```
        ("encoder", OneHotEncoder(handle_unknown="ignore",
                                   sparse_output=False)),
    ]), cat_features_melb),
])

from sklearn.ensemble import RandomForestRegressor

melb_pipeline = Pipeline([
    ("preprocessor", melb_preprocessor),
    ("model", RandomForestRegressor(n_estimators=100, random_state=42)),
])

X_melb = melb[num_features_melb + cat_features_melb]
y_melb = melb["Price"]

X_tr, X_te, y_tr, y_te = train_test_split(X_melb, y_melb,
                                           test_size=0.2, random_state=42)
melb_pipeline.fit(X_tr, y_tr)
score = melb_pipeline.score(X_te, y_te)
print(f"Melbourne Housing R^2: {score:.4f}")
```

9.8 Exercises

Exercise

1. Build a `ColumnTransformer` pipeline for the Titanic dataset with: median imputation + scaling for numerical features, and mode imputation + one-hot encoding for categorical features. Evaluate with 5-fold cross-validation.

2. Create a custom transformer `OutlierClipper` that clips values at the 1st and 99th percentiles. Integrate it into a pipeline.

3. Build a complete pipeline for the Melbourne Housing dataset that includes feature engineering (price per room, building age). Save it with `joblib` and reload it to make a prediction.

4. Compare pipeline accuracy with and without preprocessing on the Adult Census dataset. Use a decision tree classifier.

5. Create a pipeline that handles numerical, categorical, and text columns simultaneously using `ColumnTransformer`. Use TF-IDF for the text column.

9.9 Chapter summary

- Scikit-learn `Pipeline` chains transformations and a model into a single object, preventing data leakage.

- `ColumnTransformer` applies different transformations to different column types.

- Custom transformers (inheriting from `BaseEstimator` and `TransformerMixin`) integrate seamlessly into pipelines.
- Saving pipelines with `joblib` captures all fitted parameters for reproducible deployment.
- A pipeline ensures that training-time and prediction-time preprocessing are identical.

CHAPTER 9. PIPELINES AND AUTOMATION

Chapter 10

Capstone project

"The best way to learn data preprocessing is to preprocess data."

10.1 Capstone overview

In this final chapter, you will apply everything from Chapters 1–9 to a complete, end-to-end preprocessing project. You will choose one of five project options, each using a real, messy dataset that requires substantial cleaning and transformation before analysis.

Data tip

Each project is designed to take approximately 3 hours. You are expected to deliver: (1) a Jupyter notebook with documented code, (2) a clean, analysis-ready dataset saved as Parquet, and (3) a brief written report (1–2 pages) summarizing your preprocessing decisions and their justification.

10.2 Evaluation criteria

Your project will be evaluated on five criteria:

Criterion	Weight	Description
Data quality report	15%	Thorough initial assessment of data quality
Missing data strategy	20%	Justified choice of imputation method(s)
Outlier handling	15%	Detection, investigation, and treatment
Feature engineering	25%	Creativity and domain relevance of new features
Pipeline and reproducibility	25%	Complete, leakage-free, saved pipeline

10.3 Project 1: Melbourne Housing price prediction

Dataset: Melbourne Housing Snapshot (Kaggle)
Goal: Prepare the dataset for a house price regression model.

10.3.1 Requirements

```python
import pandas as pd

# Load the raw dataset
df = pd.read_csv("melb_data.csv")
print(f"Shape: {df.shape}")
print(f"Missing values:\n{df.isnull().sum()}")
```

Tasks:

1. Produce a comprehensive data quality report (Chapter 1).
2. Handle missing values in `BuildingArea`, `YearBuilt`, `Car`, `CouncilArea` using at least two different imputation strategies. Justify your choices (Chapter 3).
3. Detect and treat outliers in `Price`, `Landsize`, and `BuildingArea` (Chapter 4).
4. Encode categorical variables (`Type`, `Method`, `Regionname`, `Suburb`). For high-cardinality `Suburb`, use target encoding or frequency encoding (Chapter 5).
5. Engineer at least 5 new features: price per room, building age, distance categories, suburb-level averages, seasonal sale indicator (Chapter 6).
6. Build a complete `Pipeline` with `ColumnTransformer`. Save with `joblib` (Chapter 9).
7. Train a baseline model and report R^2 on a held-out test set.

10.4 Project 2: Titanic survival classification

Dataset: Titanic (Kaggle)
Goal: Prepare the dataset for a survival classification model.

10.4.1 Requirements

```python
url = ("https://raw.githubusercontent.com/datasciencedojo/"
       "datasets/master/titanic.csv")
df = pd.read_csv(url)
print(f"Shape: {df.shape}")
print(f"Survival rate: {df['Survived'].mean():.2%}")
```

Tasks:

1. Analyze missing data patterns. Investigate whether `Age` missingness is MCAR or MAR by comparing it against `Pclass` and `Sex` (Chapter 3).
2. Impute `Age` using KNN imputation. Compare with median imputation by evaluating downstream model accuracy.
3. Extract title from `Name` (Mr, Mrs, Miss, Master, etc.) and encode it (Chapters 5 and 7).
4. Create features: `FamilySize` = `SibSp` + `Parch` + 1, `IsAlone`, `FarePerPerson`, cabin deck letter (Chapter 6).
5. Build a complete pipeline with cross-validated evaluation.

10.5 Project 3: FIFA 21 player value estimation

Dataset: FIFA 21 Raw Data (Kaggle)
Goal: Clean the notoriously messy FIFA dataset and prepare it for player value prediction.

10.5.1 Requirements

```
df = pd.read_csv("fifa21_raw.csv")
print(f"Shape: {df.shape}")
# Note: many columns have string-encoded numbers (e.g., "€110M", "190lbs")
print(df[["Value", "Wage", "Weight", "Height"]].head())
```

Tasks:

1. Parse string-encoded currency values (`Value`, `Wage`): convert “€110M” to 110000000, “€220K” to 220000.
2. Parse height (“5’11”) and weight (“190lbs”) to metric units.
3. Handle the `Hits` and `W/F`, `SM`, `IR` columns (star ratings stored as strings).
4. Detect and treat outliers in player values (many free agents have value 0).
5. Engineer position-based features, age-based potential features, and physical attribute ratios.
6. Build a clean pipeline from raw CSV to model-ready features.

10.6 Project 4: Jena Climate time series forecasting

Dataset: Jena Climate 2009–2016 (Kaggle)
Goal: Prepare the dataset for temperature forecasting.

10.6.1 Requirements

```
df = pd.read_csv("jena_climate_2009_2016.csv")
df["datetime"] = pd.to_datetime(df["Date Time"],
                                  format="%d.%m.%Y %H:%M:%S")
df = df.set_index("datetime")
print(f"Shape: {df.shape}")
print(f"Date range: {df.index.min()} to {df.index.max()}")
```

Tasks:

1. Resample from 10-minute to hourly and daily frequencies (Chapter 8).
2. Detect and handle sensor anomalies (sudden spikes, constant readings).
3. Create lag features: 1h, 6h, 12h, 24h, 7d for temperature.
4. Create rolling statistics: 24h mean, 7d mean, 24h std.
5. Decompose into trend, seasonality, and residual. Use each as a feature.
6. Add cyclical encoding for hour of day, day of week, and month.
7. Build a pipeline and train a baseline model for 24-hour-ahead temperature prediction.

10.7 Project 5: 20 Newsgroups text classification

Dataset: 20 Newsgroups (scikit-learn)
Goal: Preprocess text data for topic classification.

10.7.1 Requirements

```
from sklearn.datasets import fetch_20newsgroups

# Use a subset of 5 categories
categories = ["sci.space", "rec.sport.baseball", "comp.graphics",
              "talk.politics.mideast", "soc.religion.christian"]
newsgroups = fetch_20newsgroups(subset="all", categories=categories,
                                remove=("headers", "footers", "quotes"))
print(f"Documents: {len(newsgroups.data)}")
print(f"Categories: {newsgroups.target_names}")
```

Tasks:

1. Build a text cleaning pipeline: lowercase, remove emails/URLs/numbers, remove special characters (Chapter 7).
2. Tokenize, remove stopwords, and lemmatize.

3. Create a TF-IDF matrix with appropriate parameters (`min_df`, `max_df`, `ngram_range`).

4. Add document-level features: document length, average word length, vocabulary richness (unique words / total words).

5. Build a pipeline combining text preprocessing and a classifier.

6. Evaluate with cross-validation and a confusion matrix.

10.8 Deliverables checklist

```python
# Template for the final deliverable check
deliverables = {
    "data_quality_report": False,     # Section with quality metrics
    "missing_data_handled": False,     # Justified imputation
    "outliers_handled": False,         # Detection + treatment
    "features_engineered": False,      # >= 5 new features
    "pipeline_built": False,           # sklearn Pipeline
    "pipeline_saved": False,           # joblib file
    "baseline_model": False,           # Train + evaluate
    "report_written": False,           # 1-2 page summary
}

# Check off each item as you complete it
for item, done in deliverables.items():
    status = "DONE" if done else "TODO"
    print(f"  [{status}] {item}")
```

10.9 Exercises

Exercise

1. Choose one of the five projects above. Complete all required tasks and submit your Jupyter notebook, saved pipeline, and written report.

2. (Bonus) After completing your main project, apply the same preprocessing approach to a second project. Compare the challenges and discuss how preprocessing strategies differ across domains.

3. (Bonus) Deploy your saved pipeline as a simple prediction function: write a script that loads the pipeline from `joblib` and accepts new data from the command line or a CSV file.

10.10 Chapter summary

- A capstone project integrates all preprocessing skills: loading, cleaning, imputation, outlier handling, encoding, scaling, feature engineering, and pipeline automation.

 CHAPTER 10. CAPSTONE PROJECT

- Real datasets require domain-specific decisions at every step — there is no single "correct" preprocessing recipe.
- Reproducibility (via pipelines and saved artifacts) is as important as accuracy.
- Documentation of preprocessing choices and their justification is a professional requirement.
- The five project options cover tabular regression, tabular classification, messy data cleaning, time series, and text — the main data preprocessing domains.

Appendix A: Python installation guide

Option 1: Google Colab (recommended for beginners)

Go to `https://colab.research.google.com`. Sign in with a Google account. No installation needed. All libraries used in this course are pre-installed.

Option 2: Anaconda (local installation)

Download Anaconda from `https://www.anaconda.com/download`. Install with default settings. Open Jupyter Notebook from Anaconda Navigator.

Required libraries

```
pip install pandas numpy matplotlib seaborn scikit-learn missingno nltk spacy
↪  joblib openpyxl sqlalchemy
```

CHAPTER 10. CAPSTONE PROJECT

Appendix B: Dataset sources

Source	Description	URL
Melbourne Housing	Housing prices with missing values and mixed types	`kaggle.com`
Titanic	Passenger survival data with missing ages and cabins	`kaggle.com/c/titanic`
FIFA 21 Raw	Player attributes with messy string-encoded numbers	`kaggle.com`
Adult Census (UCI)	Income prediction with mixed categorical/numerical	`archive.ics.uci.edu`
Jena Climate	10-minute weather measurements over several years	`kaggle.com`
Air Quality UCI	Hourly air pollutant measurements with sensor drift	`archive.ics.uci.edu`
20 Newsgroups	Text classification dataset (20 topic categories)	via `sklearn`
Amazon Reviews	Product reviews for sentiment and text processing	`kaggle.com`
Gapminder	Country-level socioeconomic indicators over time	`gapminder.org`