Essential Guide to Feature Engineering

Turn raw data into signal: encode, scale, transform, and pipeline with pandas and scikit-learn.

A practical guide to encoding, scaling, transforming, and piping data into model-ready features with pandas and scikit-learn.

rubió
Metabolomics

Ibon Martínez-Arranz | imartinez@labrubiocom
Data Science Manager at Rubió Metabolomics
www.rubiometabolomics.com

Ibon Martínez-Arranz, PhD in Mathematics and Statistics, holds MSc degrees in Applied Statistical Techniques and Mathematical Modeling. He has extensive expertise in statistical modeling and advanced data analysis techniques. Since 2017, he has led the Data Science area at Rubió Metabolomics, driving predictive model development and statistical computing management for metabolomics, data handling, and R&D projects. His doctoral research in Mathematics investigated and adapted genetic algorithms to improve the classification of NAFLD subtypes.

Itziar Mincholé Canals | iminchole@labrubio.com
Data Specialist at Rubió Metabolomics
www.rubiometabolomics.com

Itziar Mincholé Canals has a degree in physical sciences (1999) from the University of Zaragoza (Spain). In 1999 she began developing her professional career as a software analyst and developer, mainly working on web development projects and database programming in different sectors. In 2009 she joined OWL Metabolomics as a bioinformatician, where she has participated in several R&D projects and has developed several software tools for metabolomics. In 2016 she completed the master's degree in applied statistics with R software. After joining the Data Science department, she also supports data analysis.

Essential Guide to Feature Engineering

Ibon Martínez-Arranz

Life Feels Good

rubió Metabolomics

Contents

Introduction and Setup **1**
What is Feature Engineering? 1
Why is Feature Engineering Important? 1
Libraries Used in This Book 2
Example Datasets 3
Quick Overview of the Datasets 4
Iris Dataset 4
California Housing Dataset 6
Titanic Dataset 9
Initial Data Distributions 13
What Comes Next 17

Encoding Categorical Variables **19**
Categorical vs Numerical Features 19
Takeaways 24
Label Encoding (with `sklearn.preprocessing.LabelEncoder`) 25
One-Hot Encoding (pandas & scikit-learn) 26
Ordinal Encoding (with `sklearn.preprocessing.OrdinalEncoder`) 31
High-Cardinality Categories 34
Practical Pitfalls: Dummy Variable Trap & Sparse Matrices 36
Dummy Variable Trap 36
Sparse Matrices 37
Putting It Together: A Mini Encoding Pipeline 38
Summary 43

Scaling Numerical Features **45**
Why scaling is necessary 45
Standardization with `StandardScaler` 45
Min-Max Scaling with `MinMaxScaler` 50
Robust Scaling with `RobustScaler` 52
Comparing scaling methods on the same dataset 55
Practical tips: essential vs optional 58
What comes next 59

Transforming Variables **61**
Detecting skewness and choosing a transformation 61
Log transformations for skewed data 63
Box-Cox and Yeo-Johnson with `PowerTransformer` 67
Quantile transformations for non-Gaussian distributions 69
Binarization with `Binarizer` 71
Combining transformations with pipelines 74
Summary 77

Feature Generation **79**
Creating interaction features (`PolynomialFeatures`) 79
Feature crosses: combining categorical features 84
Extracting date/time features 87
Text preprocessing basics: Bag-of-Words and TF-IDF 93
Domain-driven features (California Housing) 96
Summary 104

Handling Missing Data **107**
Why missing data matters (MCAR, MAR, MNAR) 107
Inspecting missingness 107
Simple imputation (mean/median/most-frequent/constant) 110
Adding missingness indicators 114
KNNImputer (neighbors-based imputation) 117

IterativeImputer (multivariate, model-based) 119
Visual check: distribution before/after imputation (adds value) 121
Leakage-safe evaluation: quick comparison 122
Summary 124

Introduction and Setup

What is Feature Engineering?

Feature engineering is the art and science of transforming raw data into meaningful variables (features) that a machine learning model can effectively use.

A typical dataset is rarely ready to be fed directly into an algorithm.
Categorical variables, missing values, different scales, or skewed distributions can all reduce model performance or even cause failures.

Feature engineering involves tasks such as: - Converting categories into numbers (encoding). - Scaling and normalizing numerical variables. - Applying transformations to improve distributions. - Creating new features from existing ones. - Handling missing values consistently.

In this book, we will go through these steps using practical examples in **Python**, mainly with **pandas** and **scikit-learn**.

Why is Feature Engineering Important?

Machine learning algorithms, no matter how sophisticated, cannot fully compensate for poorly prepared data.

- A strong model with bad features $\rightarrow$ poor results.

- A simple model with well-crafted features → surprisingly good results.

In practice, most machine learning projects spend more than **70% of the time on preprocessing and feature engineering**, not on model tuning.

This document will cover: - Encoding of categorical variables. - Scaling and normalization of numerical variables. - Variable transformations (log, Box-Cox, Yeo-Johnson). - Feature creation. - Missing value imputation. - Pipelines for reproducibility.

Libraries Used in This Book

We will rely on the following libraries:

- **pandas** → handling tabular data (load, clean, transform).
- **numpy** → efficient numerical operations.
- **scikit-learn** → preprocessing tools (`OneHotEncoder`, `StandardScaler`, `PowerTransformer`, `Pipeline`, etc.).
- **matplotlib** and **seaborn** → visualization of distributions and results.

Let's import them and check that everything is working.

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

from sklearn import datasets
```

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

from sklearn import datasets
```

Example Datasets

We will use some classical machine learning datasets, easily available from `scikit-learn` or `seaborn`:

- **Titanic dataset** (survival of passengers).
  Useful to work with categorical variables, missing values, and encoding.
- **Iris dataset** (classification of flowers based on measurements).
  Perfect for scaling, transformations, and simple pipelines.
- **California Housing dataset** (prediction of house prices).
  Good for numeric features, scaling, and feature generation.

Let's load them.

```
# Iris dataset
iris = datasets.load_iris(as_frame=True)
iris_df = iris.frame

# California housing dataset
housing = datasets.fetch_california_housing(as_frame=True)
housing_df = housing.frame

# Titanic dataset (from seaborn)
titanic_df = sns.load_dataset("titanic")

print("Iris shape:", iris_df.shape)
print("Housing shape:", housing_df.shape)
print("Titanic shape:", titanic_df.shape)
```

```
# Iris dataset
iris = datasets.load_iris(as_frame=True)
iris_df = iris.frame

# California housing dataset
housing = datasets.fetch_california_housing(as_frame=True)
```

```
housing_df = housing.frame

# Titanic dataset (from seaborn)
titanic_df = sns.load_dataset("titanic")

print("Iris shape:", iris_df.shape)
print("Housing shape:", housing_df.shape)
print("Titanic shape:", titanic_df.shape)
```

```
Iris shape: (150, 5)
Housing shape: (20640, 9)
Titanic shape: (891, 15)
```

Quick Overview of the Datasets

Iris Dataset

- **150 samples**, 4 numerical features (sepal length/width, petal length/width).
- Target: species of iris flower (Setosa, Versicolor, Virginica).
- Good for classification tasks.

```
# Quick peek at each dataset
display(iris_df.head())
```


```
# Quick peek at each dataset
display(iris_df.head())
```


sepal length (cm)

sepal width (cm)

petal length (cm)

petal width (cm)

target

0

5.1

3.5

1.4

0.2

0

1

4.9

3.0

1.4

0.2

0

2

4.7

3.2

1.3

0.2

0

3

4.6

3.1

1.5

0.2

0

4

5.0

3.6

1.4

0.2

0

California Housing Dataset

- **20,640 samples**, 8 numerical features (median income, average rooms, latitude, longitude…).
- Target: median house value.
- Good for regression tasks.

```
# Quick peek at each dataset
display(housing_df.head())
```


```
# Quick peek at each dataset
display(housing_df.head())
```


MedInc

HouseAge

AveRooms

AveBedrms

Population

AveOccup

Latitude

Longitude

MedHouseVal

0

8.3252

41.0

6.984127

1.023810

322.0

2.555556

37.88

-122.23

4.526

1

8.3014

21.0

6.238137

0.971880

2401.0

2.109842

37.86

-122.22

3.585

2

7.2574

52.0

8.288136

1.073446

496.0

2.802260

37.85

-122.24

3.521

3

5.6431

52.0

5.817352

1.073059

558.0

2.547945

37.85

-122.25

3.413

4

3.8462

52.0

6.281853

1.081081

565.0

2.181467

37.85

-122.25

3.422

Titanic Dataset

- **891 passengers**, mix of categorical and numerical features (sex, age, class, embarked…).
- Target: survival (0 = died, 1 = survived).
- Good for preprocessing demonstrations (encoding, imputation).

```
# Quick peek at each dataset
display(titanic_df.head())
```


```
# Quick peek at each dataset
display(titanic_df.head())
```


survived

pclass

sex

age

sibsp

parch

fare

embarked

class

who

adult_male

deck

embark_town

alive

alone

0

0

3

male

22.0

1

0

7.2500

S

Third

man

True

NaN

Southampton

no

False

1

1

1

female

38.0

1

0

71.2833

C

First

woman

False

C

Cherbourg

yes

False

2

1

3

female

26.0

0

0

7.9250

S

Third

woman

False

NaN

Southampton

yes

True

3

1

1

female

35.0

1

0

53.1000

S

First

woman

False

C

Southampton

yes

False

4

0

3

male

35.0

0

0

8.0500

S

Third

man

True

NaN

Southampton

no

True

Initial Data Distributions

Before diving into encoding, scaling, and transformations, it is useful to visualize the **distributions of the features**.

This helps us see: - Which variables are numeric vs categorical. - If variables have different scales. - If distributions are skewed. - How datasets differ in complexity.

Let's plot some histograms for each dataset.

```
# Histograms for Iris dataset
iris_df.hist(figsize=(3*3,2.5*2), bins = 15, layout =
    (2,3), color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("Iris Dataset - Feature Distributions",
    fontsize=14)
plt.show()
```


```
# Histograms for Iris dataset
iris_df.hist(figsize=(3*3,2.5*2), bins = 15, layout =
    (2,3), color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("Iris Dataset - Feature Distributions",
    fontsize=14)
plt.show()
```

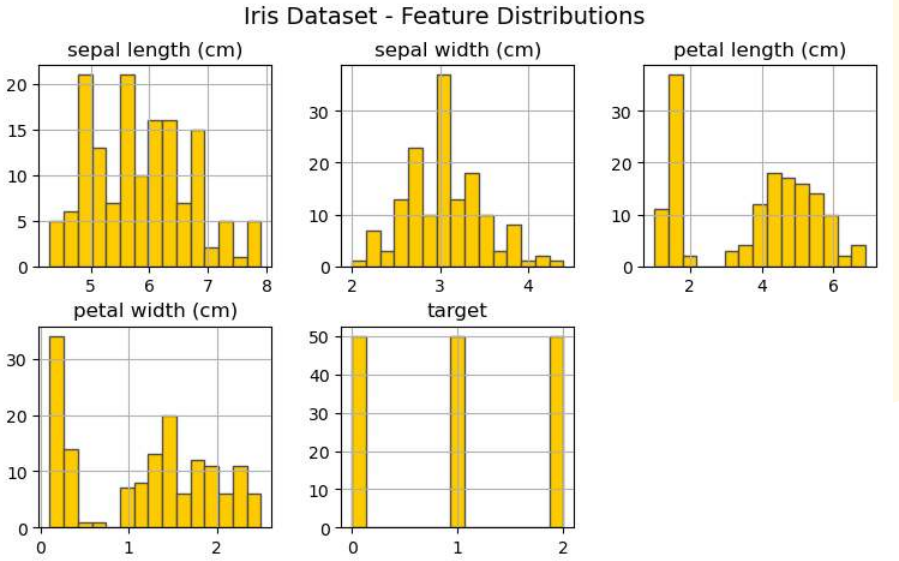

Figure 1: Image generated by the provided code.

```
# Histograms for California Housing dataset
housing_df.hist(figsize=(3*3,2.5*3), bins = 15, layout =
    (3,3), color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("California Housing Dataset - Feature
    Distributions", fontsize=14)
plt.show()
```


```
# Histograms for California Housing dataset
housing_df.hist(figsize=(3*3,2.5*3), bins = 15, layout =
    (3,3), color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("California Housing Dataset - Feature
    Distributions", fontsize=14)
plt.show()
```

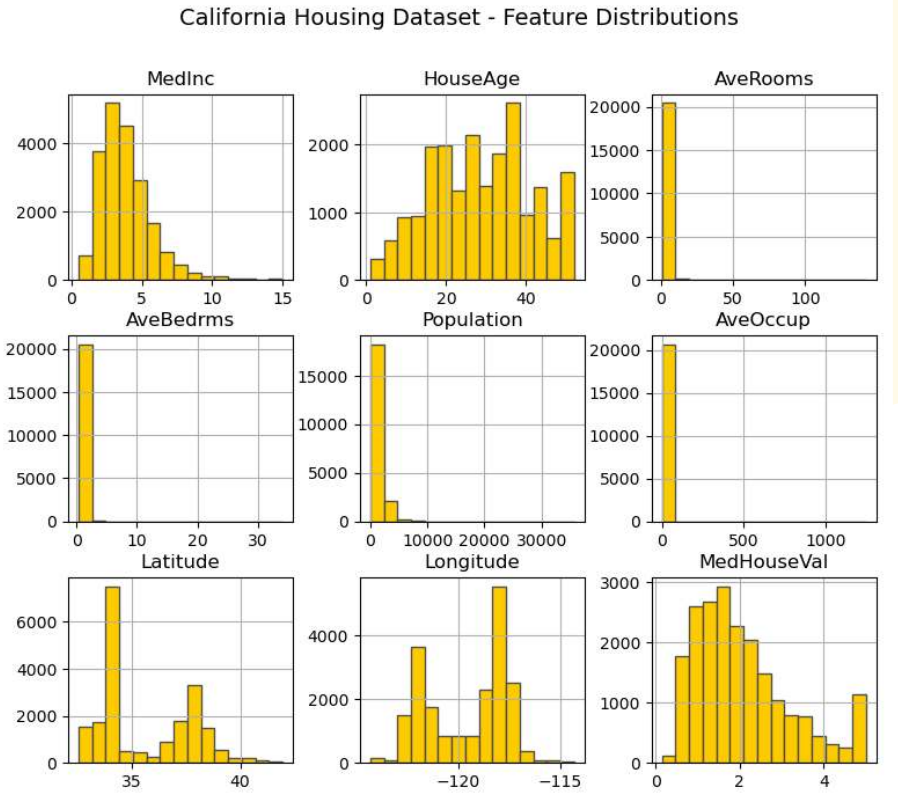

Figure 2: Image generated by the provided code.

```
# Histograms for Titanic dataset (numeric features only)
titanic_df.select_dtypes(include=["float64", "int64"]).
    hist(figsize=(3*3,2.5*2), bins = 15, layout = (2,3),
    color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("Titanic Dataset - Numeric Feature
    Distributions", fontsize=14)
plt.show()
```

```
# Histograms for Titanic dataset (numeric features only)
```

```
titanic_df.select_dtypes(include=["float64", "int64"]).
    hist(figsize=(3*3,2.5*2), bins = 15, layout = (2,3),
    color = "#FECB00", edgecolor = '0.3',)
plt.suptitle("Titanic Dataset - Numeric Feature
    Distributions", fontsize=14)
plt.show()
```

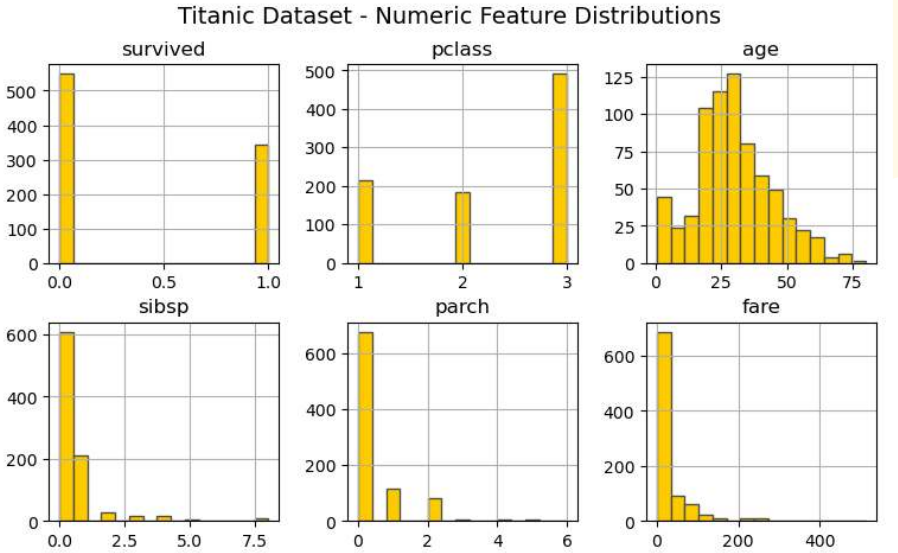

Figure 3: Image generated by the provided code.

What Comes Next

Now that the environment is set up and we have a first look at the datasets, in the next chapters we will learn to:

1. Encode categorical variables (label encoding, one-hot, ordinal).
2. Scale and normalize numerical variables.

3. Transform distributions with log, Box-Cox, and Yeo-Johnson.
4. Create new features and handle missing values.
5. Build complete preprocessing pipelines for reproducibility and production.

With this, we are ready to move to **Chapter 2: Encoding Categorical Variables**.

Encoding Categorical Variables

Categorical vs Numerical Features

Numerical features are measured on a numeric scale (e.g., `age`, `fare`, `sepal_length`).
Categorical features represent discrete groups or labels (e.g., `sex`, **`class`**, `embarked`).
Some categorical features are **ordinal** (they have an inherent order, e.g., `low < medium < high`), while others are **nominal** (no order, e.g., `red`, `green`, `blue`).

Why this matters:

- Many Machine Learning algorithms require numeric input. Categorical variables must be encoded.
- The choice of encoding depends on whether the category is **ordinal** or **nominal**.
- For high-cardinality categories (hundreds/thousands of unique values), naive one-hot encoding can explode dimensionality.

Below we'll inspect the Titanic dataset to identify categorical vs numerical columns and look at their distributions.

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

```

```
titanic_df = sns.load_dataset("titanic").copy()

# Basic dtype view
display(titanic_df.dtypes.to_frame("dtype"))

# Split by type
cat_cols = titanic_df.select_dtypes(include=["object", "category", "bool"]).columns.tolist()
num_cols = titanic_df.select_dtypes(include=["number"]).columns.tolist()

print("Categorical columns:", cat_cols)
print("Numerical columns:", num_cols)

# Quick cardinality check
cardinality = titanic_df[cat_cols].nunique().sort_values(ascending=False)
display(cardinality.to_frame("n_unique"))
```

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

titanic_df = sns.load_dataset("titanic").copy()

# Basic dtype view
display(titanic_df.dtypes.to_frame("dtype"))

# Split by type
cat_cols = titanic_df.select_dtypes(include=["object", "category", "bool"]).columns.tolist()
num_cols = titanic_df.select_dtypes(include=["number"]).columns.tolist()

print("Categorical columns:", cat_cols)
print("Numerical columns:", num_cols)

# Quick cardinality check
cardinality = titanic_df[cat_cols].nunique().sort_values(ascending=False)
display(cardinality.to_frame("n_unique"))
```

dtype

survived

int64

pclass

int64

sex

object

age

float64

sibsp

int64

parch

int64

fare

float64

embarked

object

class

category

who

object

adult_male

bool

deck

category

embark_town

object

alive

object

alone

bool

```
Categorical columns: ['sex', 'embarked', 'class', 'who',
    'adult_male', 'deck', 'embark_town', 'alive', 'alone'
    ]
Numerical columns: ['survived', 'pclass', 'age', 'sibsp',
     'parch', 'fare']
```

n_unique

deck

7

embark_town

3

embarked

3

who

3

class

3

sex

2

adult_male

2

alive

2

alone

2

```
# Plot distributions for some common categorical
    variables
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
sns.countplot(data=titanic_df, x="sex", ax=axes[0],
    palette = ['#fecb00ff','#6b99a6ff'])
axes[0].set_title("sex")

sns.countplot(data=titanic_df, x="class", ax=axes[1],
    palette = ['#b7d3dbff','#6b99a6ff','#3a6674ff'])
axes[1].set_title("class")

sns.countplot(data=titanic_df, x="embarked", ax=axes[2],
    palette = ['#ffef80ff','#fecb00ff','#cc9b00ff'])
axes[2].set_title("embarked")
plt.tight_layout()
plt.show()
```

```
# Plot distributions for some common categorical
    variables
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
sns.countplot(data=titanic_df, x="sex", ax=axes[0],
    palette = ['#fecb00ff','#6b99a6ff'])
axes[0].set_title("sex")

```

```
sns.countplot(data=titanic_df, x="class", ax=axes[1],
    palette = ['#b7d3dbff','#6b99a6ff','#3a6674ff'])
axes[1].set_title("class")

sns.countplot(data=titanic_df, x="embarked", ax=axes[2],
    palette = ['#ffef80ff','#fecb00ff','#cc9b00ff'])
axes[2].set_title("embarked")
plt.tight_layout()
plt.show()
```

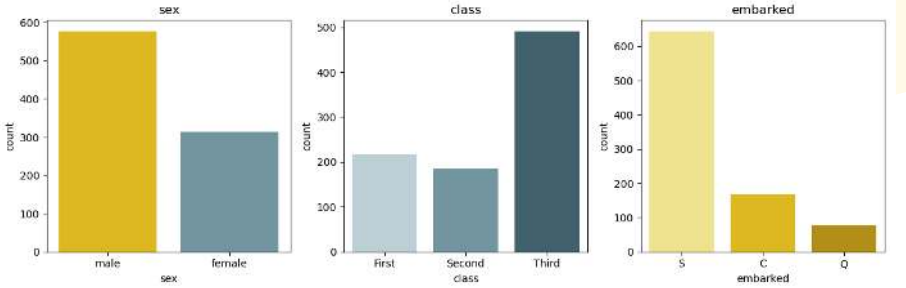

Figure 1: Image generated by the provided code.

Takeaways

- `sex`, **`class`**, and `embarked` are typical **nominal**/mixed categorical features.
- **`class`** is **ordinal** on Titanic (`Third` < `Second` < `First`).
- High-cardinality columns can cause problems for naive one-hot encoding.

Label Encoding (with `sklearn.preprocessing.LabelEncoder`)

Label Encoding converts labels to integers {`class_1: 0, class_2: 1, ...`}.
Best use: encoding the **target variable (y)** in classification problems, or truly **ordinal** features if you understand the implied ordering.

Pitfall: Using LabelEncoder on **nominal feature columns** injects a fake order (e.g., `male`=0, `female`=1), which can mislead linear and distance-based models.

We'll demonstrate it on the **Iris** dataset's target (species names).

```
from sklearn.preprocessing import LabelEncoder

iris_df = sns.load_dataset("iris").copy()  # species is string labels
y_str = iris_df["species"]

le = LabelEncoder()
y_encoded = le.fit_transform(y_str)

print("Classes:", list(le.classes_))
print("Encoded sample:", y_str.head().tolist(), "->", y_encoded[:5].tolist())
```


```
from sklearn.preprocessing import LabelEncoder

iris_df = sns.load_dataset("iris").copy()  # species is string labels
y_str = iris_df["species"]

le = LabelEncoder()
y_encoded = le.fit_transform(y_str)

print("Classes:", list(le.classes_))
print("Encoded sample:", y_str.head().tolist(), "->", y_encoded[:5].tolist())
```

```
Classes: ['setosa', 'versicolor', 'virginica']
Encoded sample: ['setosa', 'setosa', 'setosa', 'setosa',
    'setosa'] -> [0, 0, 0, 0, 0]
```

When not to use it: Do **not** use `LabelEncoder` for nominal features like `sex` or `embarked` unless your algorithm is tree-based (trees are insensitive to monotonic transformations). For general pipelines, prefer **One-Hot** or **OrdinalEncoder** (with explicit order).

One-Hot Encoding (pandas & scikit-learn)

One-Hot Encoding (OHE) creates one binary column per category (e.g., `sex_female`, `sex_male`).

- Pros: safe for nominal data, widely used.
- Cons: dimensionality blow-up for high-cardinality features.

We'll show:

1) `pandas.get_dummies()` — fast and convenient for EDA/quick baselines.
2) `sklearn.preprocessing.OneHotEncoder` — integrates with pipelines/transformers (recommended for production).

```
# One-Hot with pandas.get_dummies()
titanic_small = titanic_df[["survived", "sex", "class", "
    embarked", "age", "fare"]].copy()

# drop_first=True can avoid the "dummy variable trap" for
     linear models with intercept
dummies_pd = pd.get_dummies(
    titanic_small,
    columns=["sex", "class", "embarked"],
```

```
    drop_first=True  # set to False if you want the full
        set of dummies
)
display(dummies_pd.head())
```

```
# One-Hot with pandas.get_dummies()
titanic_small = titanic_df[["survived", "sex", "class", "
    embarked", "age", "fare"]].copy()

# drop_first=True can avoid the "dummy variable trap" for
     linear models with intercept
dummies_pd = pd.get_dummies(
    titanic_small,
    columns=["sex", "class", "embarked"],
    drop_first=True  # set to False if you want the full
        set of dummies
)
display(dummies_pd.head())
```

survived

age

fare

sex_male

class_Second

class_Third

embarked_Q

embarked_S

0

0

22.0

7.2500

1

0

1

0

1

1

1

38.0

71.2833

0

0

0

0

0

2

1

26.0

7.9250

0

0

1

0

1

3

1

35.0

53.1000

0

0

0

0

1

4

0

35.0

8.0500

1

0

1

0

1

`drop_first=True` removes one level per categorical feature to avoid perfect multicollinearity in linear models with intercept.
For tree-based or regularized models, keeping all dummies is usually fine.

```
# One-Hot with scikit-learn's OneHotEncoder +
    ColumnTransformer
from sklearn.compose import ColumnTransformer
```

```
from sklearn.preprocessing import OneHotEncoder

X = titanic_df.drop(columns=["survived"])
y = titanic_df["survived"]

categorical_features = ["sex", "class", "embarked"]
numeric_features = ["age", "fare"]

# Note: For scikit-learn >=1.2 use sparse_output=False; for <1.2 use sparse=False
ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False, drop=None)

ct = ColumnTransformer(
    transformers=[
        ("ohe", ohe, categorical_features),
        ("passthrough", "passthrough", numeric_features),
    ],
    remainder="drop"
)

X_enc = ct.fit_transform(titanic_df)
feature_names = ct.get_feature_names_out()
print("Encoded shape:", X_enc.shape)
print("Sample feature names:", feature_names[:10])
```

```
# One-Hot with scikit-learn's OneHotEncoder + ColumnTransformer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder

X = titanic_df.drop(columns=["survived"])
y = titanic_df["survived"]

categorical_features = ["sex", "class", "embarked"]
numeric_features = ["age", "fare"]

# Note: For scikit-learn >=1.2 use sparse_output=False; for <1.2 use sparse=False
ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False, drop=None)
```

```
13
14 ct = ColumnTransformer(
15     transformers=[
16         ("ohe", ohe, categorical_features),
17         ("passthrough", "passthrough", numeric_features),
18     ],
19     remainder="drop"
20 )
21
22 X_enc = ct.fit_transform(titanic_df)
23 feature_names = ct.get_feature_names_out()
24 print("Encoded shape:", X_enc.shape)
25 print("Sample feature names:", feature_names[:10])
```

```
1 Encoded shape: (891, 11)
2 Sample feature names: ['ohe__sex_female' 'ohe__sex_male'
      'ohe__class_First' 'ohe__class_Second'
3  'ohe__class_Third' 'ohe__embarked_C' 'ohe__embarked_Q' '
       ohe__embarked_S'
4  'ohe__embarked_nan' 'passthrough__age']
```

Notes

- `handle_unknown="ignore"` prevents errors when unseen categories appear at inference.
- Use `drop="if_binary"` (or `drop="first"`) to reduce redundancy for linear models.
- Prefer integrating OHE via `ColumnTransformer` and `Pipeline` so training and inference apply the exact same steps.

Ordinal Encoding (with `sklearn.preprocessing.OrdinalEncoder`)

Ordinal Encoding maps categories to ordered integers **based on a known order**. Use it **only** when the order is meaningful (e.g., `Third < Second < First`

class in Titanic).
For nominal variables, do **not** use ordinal encoding.

We will encode `class` with a custom order.

```
from sklearn.preprocessing import OrdinalEncoder
from numpy import nan

ord_cols = ["class"]
ord_order = [["Third", "Second", "First"]]

ord_enc = OrdinalEncoder(
    categories=ord_order,
    handle_unknown="use_encoded_value",
    unknown_value=npy.nan
)

# Copy to avoid modifying original
titanic_ord = titanic_df[["class"]].copy()
titanic_ord_enc = ord_enc.fit_transform(titanic_ord)

titanic_df["class_ordinal"] = titanic_ord_enc
display(titanic_df[["class", "class_ordinal"]].head(10))
```

```
**Tip:** For multiple ordinal features, pass a list of
    lists to `categories`, one per column.
For missing or unseen categories at inference, consider
    imputation or an explicit "Unknown" category in
    preprocessing.
```

```
from sklearn.preprocessing import OrdinalEncoder
from numpy import nan

ord_cols = ["class"]
ord_order = [["Third", "Second", "First"]]

ord_enc = OrdinalEncoder(
    categories=ord_order,
    handle_unknown="use_encoded_value",
    unknown_value=npy.nan
```

```
)

# Copy to avoid modifying original
titanic_ord = titanic_df[["class"]].copy()
titanic_ord_enc = ord_enc.fit_transform(titanic_ord)

titanic_df["class_ordinal"] = titanic_ord_enc
display(titanic_df[["class", "class_ordinal"]].head(10))
```

class

class_ordinal

0

Third

0.0

1

First

2.0

2

Third

0.0

3

First

2.0

4

Third

0.0

5

Third

0.0

6

First

2.0

7

Third

0.0

8

Third

0.0

9

Second

1.0

Tip: For multiple ordinal features, pass a list of lists to `categories`, one per column. For missing or unseen categories at inference, consider imputation or an explicit "Unknown" category in preprocessing.

High-Cardinality Categories

The seaborn `titanic` dataset does not include truly high-cardinality categorical features (like *ticket* or *cabin* in the Kaggle version). To demonstrate best practices:

- We will illustrate **frequency encoding** on a low-cardinality feature (`embark_town`) to show the mechanics.

- We will also create a **synthetic high-cardinality** feature (`synthetic_id`) with a long-tail distribution, to show why frequency encoding and the **hashing trick** are useful when one-hot encoding is impractical.

```
# Load seaborn titanic
titanic_df = sns.load_dataset("titanic").copy()

## Create a synthetic high-cardinality feature (e.g., ~600 unique IDs, Zipf-like long tail) ---
rng = npy.random.default_rng(42)
n = len(titanic_df)

# Generate category indices with a Zipf distribution (skewed, many rare categories)
# Cap very large values to keep the number of unique categories reasonable
zipf_raw = rng.zipf(a=2.0, size=n)
zipf_capped = npy.minimum(zipf_raw, 600)  # at most 600 categories

titanic_df["synthetic_id"] = pd.Series(zipf_capped).map(lambda i: f"ID_{i:03d}")

## Quick sanity check
print("Unique synthetic_id:", titanic_df["synthetic_id"].nunique())
print(titanic_df[["synthetic_id"]].head())
```

```
# Load seaborn titanic
titanic_df = sns.load_dataset("titanic").copy()

## Create a synthetic high-cardinality feature (e.g., ~600 unique IDs, Zipf-like long tail) ---
rng = npy.random.default_rng(42)
n = len(titanic_df)

# Generate category indices with a Zipf distribution (skewed, many rare categories)
# Cap very large values to keep the number of unique categories reasonable
```

```
zipf_raw = rng.zipf(a=2.0, size=n)
zipf_capped = npy.minimum(zipf_raw, 600)  # at most 600 categories

titanic_df["synthetic_id"] = pd.Series(zipf_capped).map(lambda i: f"ID_{i:03d}")

## Quick sanity check
print("Unique synthetic_id:", titanic_df["synthetic_id"].nunique())
print(titanic_df[["synthetic_id"]].head())
```

```
Unique synthetic_id: 40
  synthetic_id
0       ID_004
1       ID_001
2       ID_001
3       ID_001
4       ID_001
```

Pros of frequency encoding: keeps dimensionality to 1 column; often effective for tree-based models.
Cons: can leak target information if computed improperly (always compute frequencies on **train only**, then map to valid sets in validation/test).

Practical Pitfalls: Dummy Variable Trap & Sparse Matrices

Dummy Variable Trap

Creating one dummy column for **every** category + an intercept can yield **perfect multicollinearity** in linear models.
Solutions: - Drop one level per feature: `drop_first=True` in `get_dummies()` or `drop="first"` in `OneHotEncoder`. - Use regularization (Ridge/Lasso) or

algorithms robust to collinearity (trees).

Sparse Matrices

Encoders like `OneHotEncoder` and `FeatureHasher` often produce **sparse matrices**.
- Pros: huge memory savings when most entries are zero.
- Cons: not all libraries accept sparse inputs; sometimes you'll need `X.toarray()` (careful with memory!).

Let's compare memory usage roughly for a dense vs sparse representation.

```
import sys
from scipy import sparse

# Example: One-hot encode three categorical cols
ohe = OneHotEncoder(handle_unknown="ignore")  # default returns sparse
X_ohe_sparse = ohe.fit_transform(titanic_df[["sex", "class", "embarked"]])

# Convert to dense (for comparison ONLY)
X_ohe_dense = X_ohe_sparse.toarray()

# Rough memory comparison
sparse_mem = (X_ohe_sparse.data.nbytes + X_ohe_sparse.indptr.nbytes + X_ohe_sparse.indices.nbytes) / 1024
dense_mem = X_ohe_dense.nbytes / 1024

print(f"Sparse approx memory: {sparse_mem:,.1f} KB")
print(f"Dense  approx memory: {dense_mem:,.1f} KB")
```

```
import sys
from scipy import sparse

# Example: One-hot encode three categorical cols
ohe = OneHotEncoder(handle_unknown="ignore")  # default returns sparse
```

```
X_ohe_sparse = ohe.fit_transform(titanic_df[["sex", "
    class", "embarked"]])

# Convert to dense (for comparison ONLY)
X_ohe_dense = X_ohe_sparse.toarray()

# Rough memory comparison
sparse_mem = (X_ohe_sparse.data.nbytes + X_ohe_sparse.
    indptr.nbytes + X_ohe_sparse.indices.nbytes) / 1024
dense_mem = X_ohe_dense.nbytes / 1024

print(f"Sparse approx memory: {sparse_mem:,.1f} KB")
print(f"Dense  approx memory: {dense_mem:,.1f} KB")
```

```
Sparse approx memory: 34.8 KB
Dense  approx memory: 62.6 KB
```

Guideline: keep representations **sparse** as long as your downstream model supports it (e.g., linear models in scikit-learn do). Convert to dense only if strictly necessary.

Putting It Together: A Mini Encoding Pipeline

Below, we combine: - One-hot for nominal features, - Ordinal encoding for **`class`**, - And pass through a couple of numeric features.

This illustrates how to **compose encodings cleanly** before modeling.

```
# Import required classes and utilities
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder,
    OrdinalEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.impute import SimpleImputer
import numpy as npy

```

```
# Use the Titanic DataFrame from seaborn as the working dataset
df = titanic_df.copy()

# Define feature matrix X and target vector y
X = df[["sex", "class", "embarked", "age", "fare"]]
y = df["survived"]

# Group columns by semantic type: nominal, ordinal, numeric
nominal = ["sex", "embarked"]
ordinal = ["class"]
numeric = ["age", "fare"]

# Specify the explicit order for the ordinal categorical feature
ordinal_order = [["Third", "Second", "First"]]

# OneHotEncoder compatibility shim: prefer sparse_output=False, fall back to sparse=False on older scikit-learn
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

# Build the ColumnTransformer so column selection happens on the original pandas DataFrame
preprocessor = ColumnTransformer(
    transformers=[
        # Nominal features: impute missing values with the most frequent category, then one-hot encode
        ("nominal_ohe",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="most_frequent")),
             ("ohe", ohe),
         ]),
```

```
            nominal),
        # Ordinal feature: impute missing values with the most frequent category, then apply OrdinalEncoder with explicit category order
        ("class_ord",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="most_frequent")),
             ("ord", OrdinalEncoder(categories=ordinal_order)),
         ]),
         ordinal),
        # Numeric features: impute missing values with the median
        ("num",
         Pipeline(steps=[
             ("imputer", SimpleImputer(strategy="median")),
         ]),
         numeric),
    ],
    # Drop any remaining columns and keep concise feature names in the output
    remainder="drop",
    verbose_feature_names_out=False
)

# Assemble the final Pipeline: preprocessing followed by a simple Logistic Regression classifier
pipe = Pipeline(steps=[
    ("prep", preprocessor),
    ("clf", LogisticRegression(max_iter=1000))
])

# Fit the end-to-end pipeline on the training data
pipe.fit(X, y)
print("Pipeline ready. Sample prediction:", pipe.predict(X.head(5)))

# (Optional) Inspect the transformed feature names and the final feature count
```

```
feat_names = pipe.named_steps["prep"].
    get_feature_names_out()
print("n_features:", len(feat_names))
print("first 10 features:", feat_names[:10])
```

```
# Import required classes and utilities
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder,
    OrdinalEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.impute import SimpleImputer
import numpy as npy

# Use the Titanic DataFrame from seaborn as the working
    dataset
df = titanic_df.copy()

# Define feature matrix X and target vector y
X = df[["sex", "class", "embarked", "age", "fare"]]
y = df["survived"]

# Group columns by semantic type: nominal, ordinal,
    numeric
nominal = ["sex", "embarked"]
ordinal = ["class"]
numeric = ["age", "fare"]

# Specify the explicit order for the ordinal categorical
    feature
ordinal_order = [["Third", "Second", "First"]]

# OneHotEncoder compatibility shim: prefer sparse_output=
    False, fall back to sparse=False on older scikit-
    learn
try:
    ohe = OneHotEncoder(handle_unknown="ignore",
        sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=
        False)
```

```
29
30  # Build the ColumnTransformer so column selection happens
        on the original pandas DataFrame
31  preprocessor = ColumnTransformer(
32      transformers=[
33          # Nominal features: impute missing values with
                the most frequent category, then one-hot
                encode
34          ("nominal_ohe",
35           Pipeline(steps=[
36               ("imputer", SimpleImputer(strategy="
                    most_frequent")),
37               ("ohe", ohe),
38           ]),
39           nominal),
40          # Ordinal feature: impute missing values with the
                most frequent category, then apply
                OrdinalEncoder with explicit category order
41          ("class_ord",
42           Pipeline(steps=[
43               ("imputer", SimpleImputer(strategy="
                    most_frequent")),
44               ("ord", OrdinalEncoder(categories=
                    ordinal_order)),
45           ]),
46           ordinal),
47          # Numeric features: impute missing values with
                the median
48          ("num",
49           Pipeline(steps=[
50               ("imputer", SimpleImputer(strategy="median")
                    ),
51           ]),
52           numeric),
53      ],
54      # Drop any remaining columns and keep concise feature
            names in the output
55      remainder="drop",
56      verbose_feature_names_out=False
57  )
58
```

```python
# Assemble the final Pipeline: preprocessing followed by a simple Logistic Regression classifier
pipe = Pipeline(steps=[
    ("prep", preprocessor),
    ("clf", LogisticRegression(max_iter=1000))
])

# Fit the end-to-end pipeline on the training data
pipe.fit(X, y)
print("Pipeline ready. Sample prediction:", pipe.predict(X.head(5)))

# (Optional) Inspect the transformed feature names and the final feature count
feat_names = pipe.named_steps["prep"].get_feature_names_out()
print("n_features:", len(feat_names))
print("first 10 features:", feat_names[:10])
```

```
Pipeline ready. Sample prediction: [0 1 1 1 0]
n_features: 8
first 10 features: ['sex_female' 'sex_male' 'embarked_C' 'embarked_Q' 'embarked_S' 'class'
 'age' 'fare']
```

Summary

- **Categorical vs numerical**: determine type and cardinality early.
- **LabelEncoder**: typically for **targets** or **true ordinal** features; avoid for nominal features in general.
- **One-Hot**: safe default for nominal data; watch dimensionality.
- **OrdinalEncoder**: only when order is meaningful (pass explicit category order).
- **High-cardinality**: prefer **frequency encoding** or **hashing** to limit dimensions.

- **Pitfalls**: avoid the **dummy variable trap** in linear models; keep **sparse** encodings when possible.

Next, we'll move to **Scaling Numerical Features**, comparing standardization, min–max scaling, and robust scaling with practical guidance on when each is appropriate.

Scaling Numerical Features

Why scaling is necessary

Several algorithms assume features live on comparable scales or use distances/gradients that are sensitive to scale: - **Distance-based models**: k-Nearest Neighbors, SVM with RBF/poly kernels, K-Means. - **Gradient-based models**: Linear/Logistic Regression with regularization, Neural Networks (faster, more stable convergence). - **Regularization**: L1/L2 penalties depend on coefficient magnitudes; standardized features make penalties comparable.

When scaling is often **not** needed: - **Tree-based models** (Decision Trees, Random Forests, Gradient Boosted Trees) are invariant to monotonic feature scaling. - **Naive Bayes** (in many variants) is mostly unaffected by linear rescaling.

Below we compare the main scalers and show when each one shines.

Standardization with `StandardScaler`

What it does: subtract the mean and divide by the standard deviation $\rightarrow$ each feature roughly **mean 0, std 1**.

When to use: default choice for many linear models and distance-based models when distributions are not heavy-tailed.

```
1  import numpy as npy
```

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load Iris (all-numeric, clean)
iris = load_iris(as_frame=True)
X_iris = iris.frame.drop(columns=["target"]) if "target" in iris.frame else iris.data
cols = X_iris.columns

scaler_std = StandardScaler()
X_std = pd.DataFrame(scaler_std.fit_transform(X_iris), columns=cols)

# Quick sanity: means ~0, stds ~1
summary = pd.DataFrame({
    "mean_before": X_iris.mean(),
    "std_before": X_iris.std(),
    "mean_after": X_std.mean(),
    "std_after": X_std.std()
}).round(3)
display(summary)

# Plot histograms AFTER scaling (adds value: shows standardized spread)
fig, axes = plt.subplots(2, 2, figsize=(9, 6))
axes = axes.ravel()
for ax, c in zip(axes, cols):
    ax.hist(X_std[c], bins=20)
    ax.set_title(f"{c} (standardized)")
plt.tight_layout()
plt.show()
```

```
import numpy as npy
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler

# Load Iris (all-numeric, clean)
```

```
iris = load_iris(as_frame=True)
X_iris = iris.frame.drop(columns=["target"]) if "target" in iris.frame else iris.data
cols = X_iris.columns

scaler_std = StandardScaler()
X_std = pd.DataFrame(scaler_std.fit_transform(X_iris), columns=cols)

# Quick sanity: means ~0, stds ~1
summary = pd.DataFrame({
    "mean_before": X_iris.mean(),
    "std_before": X_iris.std(),
    "mean_after": X_std.mean(),
    "std_after": X_std.std()
}).round(3)
display(summary)
```

mean_before

std_before

mean_after

std_after

sepal length (cm)

5.843

0.828

-0.0

1.003

sepal width (cm)

3.057

0.436

-0.0

1.003

petal length (cm)

3.758

1.765

-0.0

1.003

petal width (cm)

1.199

0.762

-0.0

1.003

```
import numpy as npy
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler


# Plot histograms AFTER scaling (adds value: shows
    standardized spread)
fig, axes = plt.subplots(2, 2, figsize=(9, 6))
axes = axes.ravel()
for ax, c in zip(axes, cols):
    ax.hist(X_std[c], bins=20)
    ax.set_title(f"{c} (standardized)")
plt.tight_layout()
plt.show()
```

```
# Plot histograms AFTER scaling (adds value: shows
    standardized spread)
fig, axes = plt.subplots(2, 2, figsize=(9, 6))
axes = axes.ravel()
```

```
for ax, c in zip(axes, cols):
    ax.hist(X_std[c], bins=20, color = "#FECB00",
        edgecolor = '0.3',)
    ax.set_title(f"{c} (standardized)")
plt.tight_layout()
plt.show()
```

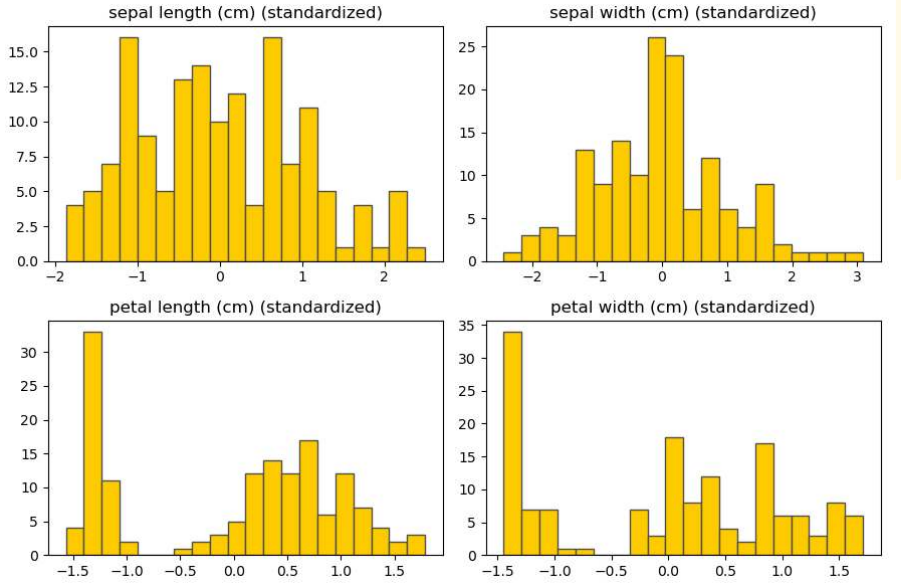

Figure 1: Image generated by the provided code.

Reading the plot/table: After standardization, each feature centers around zero with unit variance. This makes gradient steps comparable across coordinates and distances more meaningful across features.

Min-Max Scaling with `MinMaxScaler`

- **What it does:** linearly maps each feature to a bounded range, typically **[0, 1]**.
- **When to use:** algorithms that assume bounded inputs (e.g., some neural nets/activation setups), or when you need to preserve the original distribution shape but compress to a fixed interval.
- **Caveat:** very sensitive to **outliers** (min and max are extreme points).

```
from sklearn.preprocessing import MinMaxScaler

scaler_mm = MinMaxScaler(feature_range=(0, 1))
X_mm = pd.DataFrame(scaler_mm.fit_transform(X_iris),
    columns=cols)

# Show min/max after scaling
mm_bounds = pd.DataFrame({
    "min_after": X_mm.min(),
    "max_after": X_mm.max()
}).round(3)
display(mm_bounds)
```


```
from sklearn.preprocessing import MinMaxScaler

scaler_mm = MinMaxScaler(feature_range=(0, 1))
X_mm = pd.DataFrame(scaler_mm.fit_transform(X_iris),
    columns=cols)

# Show min/max after scaling
mm_bounds = pd.DataFrame({
    "min_after": X_mm.min(),
    "max_after": X_mm.max()
}).round(3)
display(mm_bounds)
```


min_after

max_after

sepal length (cm)

0.0

1.0

sepal width (cm)

0.0

1.0

petal length (cm)

0.0

1.0

petal width (cm)

0.0

1.0

```
# 2D scatter before vs after (adds value: visual bound to [0,1])
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].scatter(X_iris[cols[0]], X_iris[cols[1]], s=12)
ax[0].set_title(f"Raw: {cols[0]} vs {cols[1]}")
ax[1].scatter(X_mm[cols[0]], X_mm[cols[1]], s=12)
ax[1].set_title(f"Min-Max: {cols[0]} vs {cols[1]}")
plt.tight_layout()
plt.show()
```

```
# 2D scatter before vs after (adds value: visual bound to [0,1])
fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].scatter(X_iris[cols[0]], X_iris[cols[1]], s = 50, color = '#6b99a6ff')
ax[0].set_title(f"Raw: {cols[0]} vs {cols[1]}")
ax[1].scatter(X_mm[cols[0]], X_mm[cols[1]], s = 50, color = '#6b99a6ff')
ax[1].set_title(f"Min-Max: {cols[0]} vs {cols[1]}")
```

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

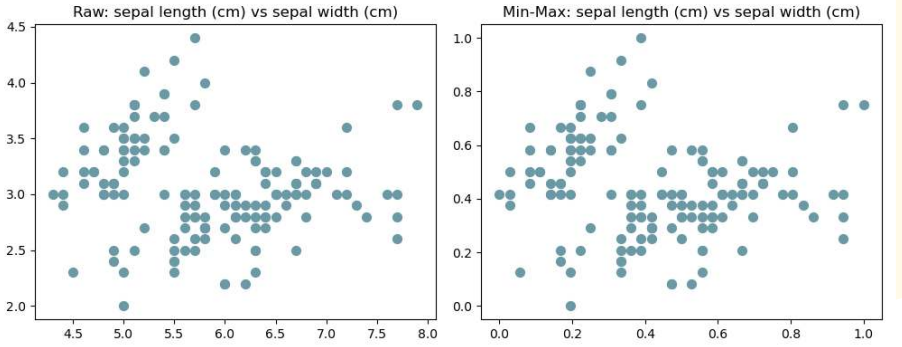

Figure 2: Image generated by the provided code.

Reading the plot/table: All features are squeezed into [0, 1]. Relative positions are preserved, but outliers can dominate the mapping.

Robust Scaling with `RobustScaler`

- **What it does:** centers using the **median** and scales by the **interquartile range (IQR)**.
- **When to use:** data with **outliers** or heavy tails; keeps the bulk of the distribution comparable without being distorted by extremes.

```
from sklearn.preprocessing import RobustScaler

# Create a synthetic feature with clear outliers to show the benefit
rng = np.random.default_rng(7)
x_main = rng.normal(loc=50, scale=10, size=400)
```

```
x_out = rng.normal(loc=120, scale=5, size=10)   # outliers
x = np.concatenate([x_main, x_out])

X_synth = pd.DataFrame({"feature": x})

scaler_std_s = StandardScaler()
scaler_rob = RobustScaler()

X_std_s = pd.DataFrame(scaler_std_s.fit_transform(X_synth), columns=["feature"])
X_rob = pd.DataFrame(scaler_rob.fit_transform(X_synth), columns=["feature"])

# Boxplots: standard vs robust (adds value: robust reduces outlier distortion)
fig, ax = plt.subplots(1, 2, figsize=(8, 4))
ax[0].boxplot(X_std_s["feature"], vert=True)
ax[0].set_title("StandardScaler")
ax[1].boxplot(X_rob["feature"], vert=True)
ax[1].set_title("RobustScaler")
plt.tight_layout()
plt.show()
```

```
from sklearn.preprocessing import RobustScaler

# Create a synthetic feature with clear outliers to show the benefit
rng = npy.random.default_rng(7)
x_main = rng.normal(loc=50, scale=10, size=400)
x_out = rng.normal(loc=120, scale=5, size=10)   # outliers
x = npy.concatenate([x_main, x_out])

X_synth = pd.DataFrame({"feature": x})

scaler_std_s = StandardScaler()
scaler_rob = RobustScaler()

X_std_s = pd.DataFrame(scaler_std_s.fit_transform(X_synth), columns=["feature"])
```

```
X_rob = pd.DataFrame(scaler_rob.fit_transform(X_synth),
    columns=["feature"])

# Boxplots: standard vs robust (adds value: robust
    reduces outlier distortion)
fig, ax = plt.subplots(1, 2, figsize=(8, 4))
ax[0].boxplot(X_std_s["feature"], vert=True)
ax[0].set_title("StandardScaler")
ax[1].boxplot(X_rob["feature"], vert=True)
ax[1].set_title("RobustScaler")
plt.tight_layout()
plt.show()
```

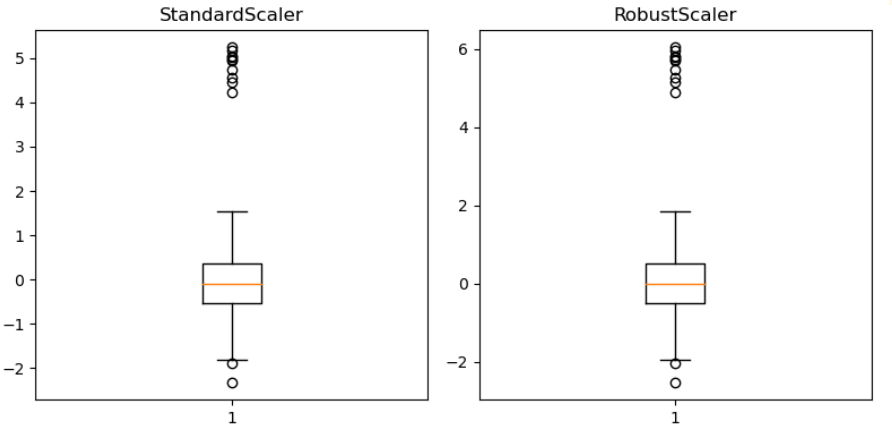

Figure 3: Image generated by the provided code.

Reading the plot: Robust scaling compresses the central mass similarly across features while preventing a few extreme values from stretching the scale.

Comparing scaling methods on the same dataset

We quantify the impact on a **distance-based model**: k-Nearest Neighbors (kNN) regression on the California Housing dataset.
We compare **no scaling**, **StandardScaler**, and **MinMaxScaler** using cross-validated RMSE.
(We subsample to keep runtime reasonable.)

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import Pipeline

# Load data
housing = fetch_california_housing(as_frame=True)
X_h = housing.frame.drop(columns=["MedHouseVal"])
y_h = housing.frame["MedHouseVal"]

# Subsample to speed up (adjust as needed)
sample = X_h.sample(n=5000, random_state=42)
target = y_h.loc[sample.index]

def rmse(scores):
    return np.sqrt(-scores.mean())

pipelines = {
    "No Scaling": Pipeline([("model", KNeighborsRegressor
        (n_neighbors=5))]),
    "StandardScaler": Pipeline([("scaler", StandardScaler
        ()), ("model", KNeighborsRegressor(n_neighbors=5)
        )]),
    "MinMaxScaler": Pipeline([("scaler", MinMaxScaler()),
         ("model", KNeighborsRegressor(n_neighbors=5))]),
}

results = {}
for name, pipe in pipelines.items():
    scores = cross_val_score(pipe, sample, target,
        scoring="neg_mean_squared_error", cv=5, n_jobs
```

```
        =-1)
    results[name] = [rmse(scores)]

results_df = pd.DataFrame(results).T
results_df.columns = ["RMSE"]
results_df = results_df.sort_values("RMSE")

# Bar plot (adds value: quick visual comparison)
fig, ax = plt.subplots(figsize=(6, 4))
ax.bar(results_df.index, results_df["RMSE"])
ax.set_ylabel("RMSE (lower is better)")
ax.set_title("kNN Regression - Effect of Scaling")
plt.tight_layout()
plt.show()
```

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsRegressor
from sklearn.pipeline import Pipeline

# Load data
housing = fetch_california_housing(as_frame=True)
X_h = housing.frame.drop(columns=["MedHouseVal"])
y_h = housing.frame["MedHouseVal"]

# Subsample to speed up (adjust as needed)
sample = X_h.sample(n=5000, random_state=42)
target = y_h.loc[sample.index]

def rmse(scores):
    return npy.sqrt(-scores.mean())

pipelines = {
    "No Scaling": Pipeline([("model", KNeighborsRegressor
        (n_neighbors=5))]),
    "StandardScaler": Pipeline([("scaler", StandardScaler
        ()), ("model", KNeighborsRegressor(n_neighbors=5)
        )]),
    "MinMaxScaler": Pipeline([("scaler", MinMaxScaler()),
         ("model", KNeighborsRegressor(n_neighbors=5))]),
}
```

```
23
24 results = {}
25 for name, pipe in pipelines.items():
26     scores = cross_val_score(pipe, sample, target,
           scoring="neg_mean_squared_error", cv=5, n_jobs
           =-1)
27     results[name] = [rmse(scores)]
28
29 results_df = pd.DataFrame(results).T
30 results_df.columns = ["RMSE"]
31 results_df = results_df.sort_values("RMSE")
32
33 # Bar plot (adds value: quick visual comparison)
34 fig, ax = plt.subplots(figsize=(6, 4))
35 ax.bar(results_df.index, results_df["RMSE"], color = ['#
       b7d3dbff','#6b99a6ff','#3a6674ff'])
36 ax.set_ylabel("RMSE (lower is better)")
37 ax.set_title("kNN Regression - Effect of Scaling")
38 plt.tight_layout()
39 plt.show()
```

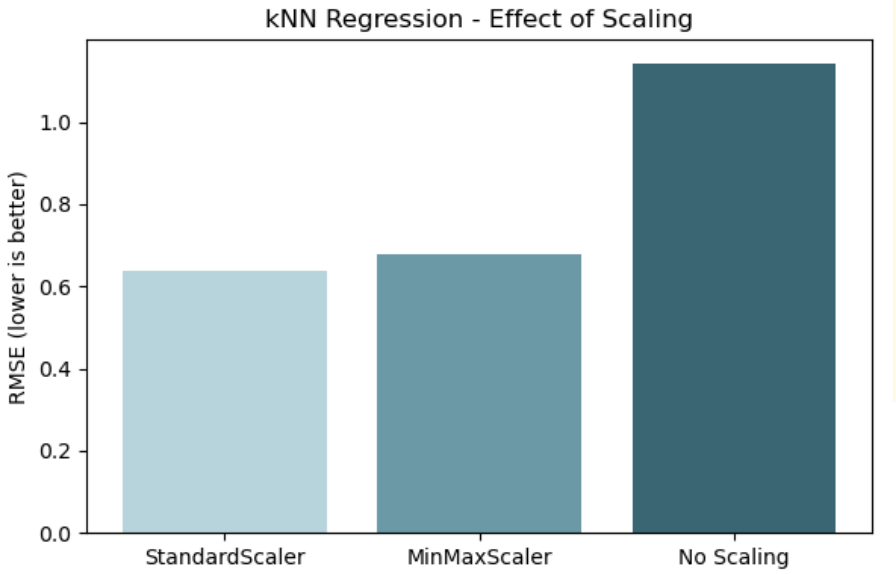

Figure 4: Image generated by the provided code.

Interpreting the numbers: Distance-based models typically benefit from scaling. Which scaler wins can depend on feature distributions; standardization is a robust default, MinMax can help if you want all features in a fixed range.

Practical tips: essential vs optional

- **Essential / highly recommended**
  - kNN, K-Means, SVM with kernels, PCA/UMAP/t-SNE, Linear/Logistic Regression with regularization, Neural Networks.
  - Mixed-unit features (e.g., `income` in thousands vs `rooms` count) almost always need scaling.

- **Useful but situational**
  - When features have very different variances; standardization evens their influence.
  - When optimization is unstable or slow to converge.
- **Usually optional**
  - Tree-based methods (Random Forests, Gradient Boosted Trees, XGBoost/LightGBM/CatBoost).
  - Models that do not use distances or gradients in ways sensitive to scale.

Rule of thumb: start with **StandardScaler**; consider **RobustScaler** if outliers dominate; use **MinMaxScaler** when a bounded range is required or when feeding into methods expecting $[0, 1]$.

What comes next

We have the core scalers under our belt. Next chapter, we'll look at **transformations** to tame skewed distributions: **log transforms**, **Box-Cox**, and **Yeo-Johnson**, and how to combine them inside **pipelines**.

Transforming Variables

Detecting skewness and choosing a transformation

Many real-world features are **skewed** (long tails). Skewness can harm models that assume near-normal inputs or use distances/gradients. Typical fixes:

- **Log transform**: for strictly positive, right-skewed variables.
- **Box-Cox** / **Yeo-Johnson**: learn power parameters; Box-Cox needs strictly positive values; Yeo-Johnson handles zero/negative.
- **Quantile transforms**: reshape distributions to **normal** or **uniform** via rank mapping.
- **Binarization**: convert to indicator features when only a threshold matters.

We'll:

1) measure skewness,
2) apply several transforms,
3) compare before/after on representative features.

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import FunctionTransformer,
    PowerTransformer, QuantileTransformer, Binarizer

# Load California Housing (all numeric, several skewed)
cal = fetch_california_housing(as_frame=True)
```

```
X = cal.frame.drop(columns=["MedHouseVal"])  # predictors only
cols = X.columns

# Compute skewness
skew_before = X.skew(numeric_only=True).sort_values(ascending=False)
skew_before.name = "skew_before"

display(pd.DataFrame(skew_before).T)  # quick overview (top row sorted)
```

```
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import FunctionTransformer, PowerTransformer, QuantileTransformer, Binarizer

# Load California Housing (all numeric, several skewed)
cal = fetch_california_housing(as_frame=True)
X = cal.frame.drop(columns=["MedHouseVal"])  # predictors only
cols = X.columns

# Compute skewness
skew_before = X.skew(numeric_only=True).sort_values(ascending=False)
skew_before.name = "skew_before"

display(pd.DataFrame(skew_before).T)  # quick overview (top row sorted)
```

AveOccup

AveBedrms

AveRooms

Population

MedInc

Latitude

HouseAge

Longitude

skew_before

97.639561

31.316956

20.697869

4.935858

1.646657

0.465953

0.060331

-0.297801

Log transformations for skewed data

When to use: strictly positive, right-skewed features (counts, incomes, areas). We will use `npy.log1p(x)` (log(1+x)) which is stable near zero.

We will illustrate on two commonly skewed features from this dataset:

- `MedInc` (median income)
- `Population`

We will verify positivity and then compare histograms before/after.

```
1  def plot_hist_pair(series_raw, series_tx, title_raw,
       title_tx):
2      fig, axes = plt.subplots(1, 2, figsize=(10, 4))
```

```
    axes[0].hist(series_raw, bins = 30, color = "#FECB00"
        , edgecolor = '0.3',)
    axes[0].set_title(title_raw)
    axes[1].hist(series_tx, bins = 30, color = "#FECB00",
         edgecolor = '0.3',)
    axes[1].set_title(title_tx)
    plt.tight_layout()
    plt.show()

# Choose two features
feat_log_candidates = ["MedInc", "Population"]
for feat in feat_log_candidates:
    s = X[feat].dropna()
    # Ensure strictly positive for log
    s_pos = s[s > 0]
    s_log = npy.log1p(s_pos)

    plot_hist_pair(
        s_pos, s_log,
        f"{feat} (raw)",
        f"{feat} (log1p)"
    )

# Skewness comparison table for selected features
log_rows = []
for feat in feat_log_candidates:
    s = X[feat].dropna()
    s_pos = s[s > 0]
    s_log = npy.log1p(s_pos)
    log_rows.append({
        "feature": feat,
        "skew_raw": float(s_pos.skew()),
        "skew_log1p": float(pd.Series(s_log).skew()),
    })
```

```
def plot_hist_pair(series_raw, series_tx, title_raw,
    title_tx):
    fig, axes = plt.subplots(1, 2, figsize=(10, 4))
    axes[0].hist(series_raw, bins = 30, color = "#FECB00"
        , edgecolor = '0.3',)
    axes[0].set_title(title_raw)
```

```
    axes[1].hist(series_tx, bins = 30, color = "#FECB00",
        edgecolor = '0.3',)
    axes[1].set_title(title_tx)
    plt.tight_layout()
    plt.show()

# Choose two features
feat_log_candidates = ["MedInc", "Population"]
for feat in feat_log_candidates:
    s = X[feat].dropna()
    # Ensure strictly positive for log
    s_pos = s[s > 0]
    s_log = npy.log1p(s_pos)

    plot_hist_pair(
        s_pos, s_log,
        f"{feat} (raw)",
        f"{feat} (log1p)"
    )

# Skewness comparison table for selected features
log_rows = []
for feat in feat_log_candidates:
    s = X[feat].dropna()
    s_pos = s[s > 0]
    s_log = npy.log1p(s_pos)
    log_rows.append({
        "feature": feat,
        "skew_raw": float(s_pos.skew()),
        "skew_log1p": float(pd.Series(s_log).skew()),
    })
```

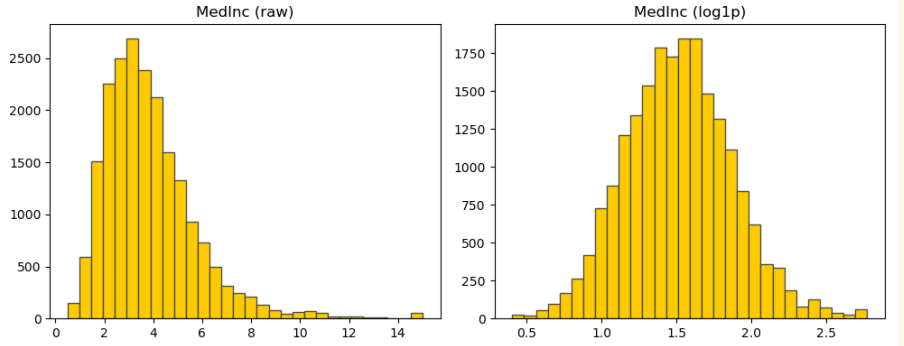

Figure 1: Image generated by the provided code.

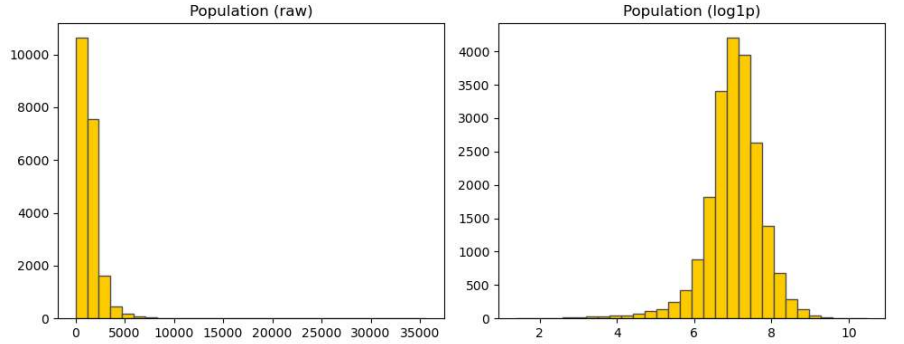

Figure 2: Image generated by the provided code.

Reading the plots/table: The log transform typically reduces right skewness and compresses long tails, making gradients and distances more stable.

Box-Cox and Yeo-Johnson with `PowerTransformer`

PowerTransformer estimates a power parameter per feature to make the data more Gaussian-like.

- **Box-Cox** works only for **strictly positive** data.
- **Yeo-Johnson** works for **zero/negative** as well and is often a safe default.

We will apply both where possible and compare skewness reductions.

```
from sklearn.compose import ColumnTransformer

# Identify strictly positive columns for Box-Cox (min > 0)
positive_cols = [c for c in cols if (X[c] > 0).all()]
non_positive_cols = [c for c in cols if c not in positive_cols]

pt_boxcox = PowerTransformer(method="box-cox", standardize=True)
pt_yeojohn = PowerTransformer(method="yeo-johnson", standardize=True)

# Fit/transform separate copies for clarity
X_box = X[positive_cols].copy()
X_box_tx = pd.DataFrame(pt_boxcox.fit_transform(X_box), columns=positive_cols, index=X_box.index)

X_yj = X.copy()
X_yj_tx = pd.DataFrame(pt_yeojohn.fit_transform(X_yj), columns=cols, index=X.index)

# Skewness summaries (median across features as a compact signal)
summary_skew = pd.DataFrame({
    "median_skew_raw": [X.skew().median()],
    "median_skew_boxcox_(pos_only)": [X_box_tx.skew().median() if len(positive_cols) else npy.nan],
    "median_skew_yeojohnson": [X_yj_tx.skew().median()]
}).round(3)
```

```
display(summary_skew)

# Visual example on MedInc with Yeo-Johnson (adds value if different from log)
feat = "MedInc"
s_raw = X[feat]
s_yj = X_yj_tx[feat]
plot_hist_pair(s_raw, s_yj, f"{feat} (raw)", f"{feat} (Yeo-Johnson)", "fig_medinc_yj_hist.png")
```

```
from sklearn.compose import ColumnTransformer

# Identify strictly positive columns for Box-Cox (min > 0)
positive_cols = [c for c in cols if (X[c] > 0).all()]
non_positive_cols = [c for c in cols if c not in positive_cols]

pt_boxcox = PowerTransformer(method="box-cox", standardize=True)
pt_yeojohn = PowerTransformer(method="yeo-johnson", standardize=True)

# Fit/transform separate copies for clarity
X_box = X[positive_cols].copy()
X_box_tx = pd.DataFrame(pt_boxcox.fit_transform(X_box), columns=positive_cols, index=X_box.index)

X_yj = X.copy()
X_yj_tx = pd.DataFrame(pt_yeojohn.fit_transform(X_yj), columns=cols, index=X.index)

# Skewness summaries (median across features as a compact signal)
summary_skew = pd.DataFrame({
    "median_skew_raw": [X.skew().median()],
    "median_skew_boxcox_(pos_only)": [X_box_tx.skew().median() if len(positive_cols) else npy.nan],
    "median_skew_yeojohnson": [X_yj_tx.skew().median()]
}).round(3)

```

```
# Visual example on MedInc with Yeo-Johnson (adds value if different from log)
feat = "MedInc"
s_raw = X[feat]
s_yj = X_yj_tx[feat]
plot_hist_pair(s_raw, s_yj, f"{feat} (raw)", f"{feat} (Yeo-Johnson)", )
```

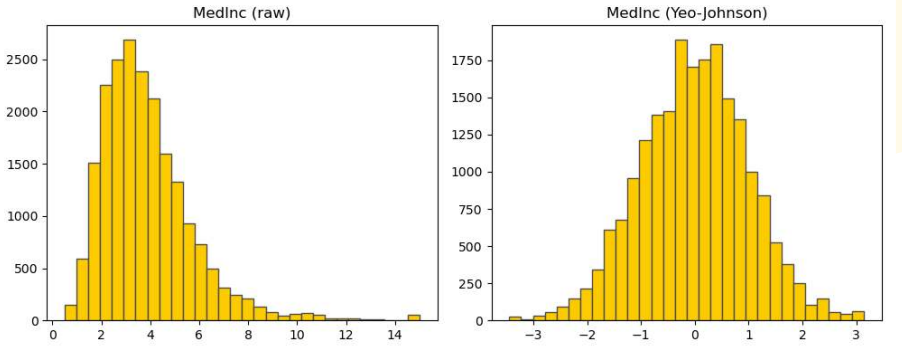

Figure 3: Image generated by the provided code.

Interpretation: Both Box-Cox and Yeo-Johnson usually reduce skewness. Yeo-Johnson is more flexible and avoids the "strictly positive" constraint.

Quantile transformations for non-Gaussian distributions

QuantileTransformer maps values to their empirical quantiles and then to a target distribution:

- `output_distribution="normal"` → approximately standard normal.

- `output_distribution="uniform"` → approximately uniform on [0, 1].

This can dramatically stabilize highly non-Gaussian features, at the cost of non-linear distortions.

```
qt_norm = QuantileTransformer(output_distribution="normal
    ", random_state=42)
qt_unif = QuantileTransformer(output_distribution="
    uniform", random_state=42)

feat = "Population"  # typically heavy-tailed
s = X[feat].values.reshape(-1, 1)

s_norm = qt_norm.fit_transform(s).ravel()
s_unif = qt_unif.fit_transform(s).ravel()

# Histograms to show target distributions
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].hist(s.ravel(), bins=30)
axes[0].set_title(f"{feat} (raw)")
axes[1].hist(s_norm, bins=30)
axes[1].set_title(f"{feat} (Quantile -> Normal)")
axes[2].hist(s_unif, bins=30)
axes[2].set_title(f"{feat} (Quantile -> Uniform)")
plt.tight_layout()
plt.show()
```

```
qt_norm = QuantileTransformer(output_distribution="normal
    ", random_state=42)
qt_unif = QuantileTransformer(output_distribution="
    uniform", random_state=42)

feat = "Population"  # typically heavy-tailed
s = X[feat].values.reshape(-1, 1)

s_norm = qt_norm.fit_transform(s).ravel()
s_unif = qt_unif.fit_transform(s).ravel()

# Histograms to show target distributions
```

```
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].hist(s.ravel(), bins=30, color = "#FECB00",
    edgecolor = '0.3',)
axes[0].set_title(f"{feat} (raw)")
axes[1].hist(s_norm, bins=30, color = "#FECB00",
    edgecolor = '0.3',)
axes[1].set_title(f"{feat} (Quantile -> Normal)")
axes[2].hist(s_unif, bins=30, color = "#FECB00",
    edgecolor = '0.3',)
axes[2].set_title(f"{feat} (Quantile -> Uniform)")
plt.tight_layout()
plt.show()
```

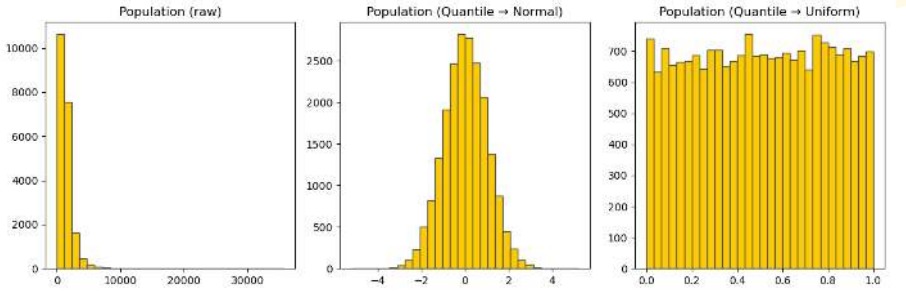

Figure 4: Image generated by the provided code.

Trade-off: Quantile transforms can greatly help algorithms needing near-Gaussian inputs but may warp relationships. Consider applying them within **pipelines** and validate their impact on your metrics.

Binarization with `Binarizer`

Binarizer converts a numeric feature into a binary indicator using a threshold. Use it when a **decision boundary** (e.g., above/below median or a domain-specific cutoff) carries most of the predictive signal.

We will create a binary indicator for **high income** using the median of `MedInc`.

```
from sklearn.preprocessing import Binarizer

threshold = float(X["MedInc"].median())
bin_tx = Binarizer(threshold=threshold)  # > threshold becomes 1.0
medinc_bin = bin_tx.fit_transform(X[["MedInc"]]).ravel()

# Show proportion of 1s
prop_high = medinc_bin.mean()
print(f"Threshold used: {threshold:.3f}; share above threshold: {prop_high:.2%}")

# Simple before/after visualization (adds value: shows the discretization)
fig, axes = plt.subplots(1, 2, figsize=(9, 4))
axes[0].hist(X["MedInc"], bins=30, color = "#FECB00", edgecolor = '0.3',)
axes[0].set_title("MedInc (raw)")
axes[1].hist(medinc_bin, bins=2, color = "#FECB00", edgecolor = '0.3',)
axes[1].set_xticks([0, 1])
axes[1].set_title("MedInc (binarized)")
plt.tight_layout()
plt.show()
```

```
from sklearn.preprocessing import Binarizer

threshold = float(X["MedInc"].median())
bin_tx = Binarizer(threshold=threshold)  # > threshold becomes 1.0
medinc_bin = bin_tx.fit_transform(X[["MedInc"]]).ravel()

# Show proportion of 1s
prop_high = medinc_bin.mean()
print(f"Threshold used: {threshold:.3f}; share above threshold: {prop_high:.2%}")

# Simple before/after visualization (adds value: shows the discretization)
```

```
fig, axes = plt.subplots(1, 2, figsize=(9, 4))
axes[0].hist(X["MedInc"], bins=30, color = "#FECB00",
    edgecolor = '0.3',)
axes[0].set_title("MedInc (raw)")
axes[1].hist(medinc_bin, bins=2, color = "#FECB00",
    edgecolor = '0.3',)
axes[1].set_xticks([0, 1])
axes[1].set_title("MedInc (binarized)")
plt.tight_layout()
plt.show()
```

```
Threshold used: 3.535; share above threshold: 50.00%
```

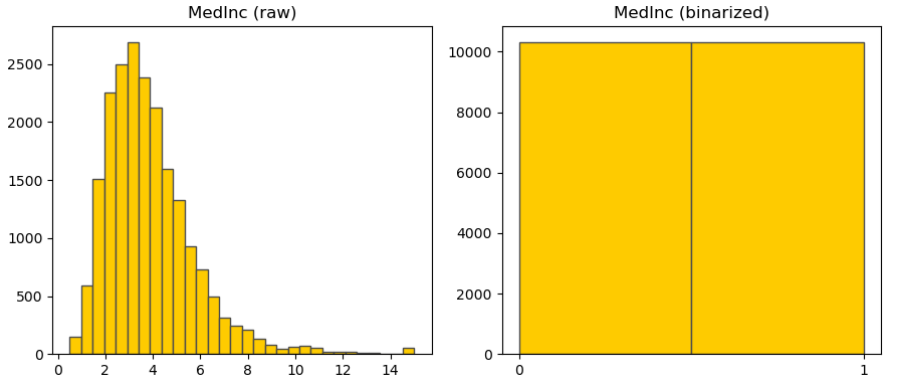

Figure 5: Image generated by the provided code.

Caution: Binarization discards within-bin variation. It can be powerful for rule-like signals but may reduce information for models that benefit from continuous structure.

Combining transformations with pipelines

We will:

- Apply **Yeo-Johnson** to skewed features,
- Leave approximately symmetric features as-is (or standardize),
- Add a **binarized** version of `MedInc` via a separate pipeline branch,
- Train a simple **Ridge** regressor on the California Housing target to illustrate end-to-end usage.

We will compare RMSE with and without transformations.

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

target = cal.frame["MedHouseVal"]

# Heuristic: mark skewed if |skew| >= 1.0
skews = X.skew().abs()
skewed_cols = skews[skews >= 1.0].index.tolist()
other_cols = [c for c in cols if c not in skewed_cols]

# Pipelines for branches
yj_pipe = Pipeline(steps=[
    ("yeojohn", PowerTransformer(method="yeo-johnson",
        standardize=True))
])

# "Passthrough" for other columns (you could add
    StandardScaler if desired)
pass_pipe = "passthrough"

# Binarized MedInc as an extra feature (single-column
    pipeline)
bin_col = ["MedInc"]
bin_threshold = float(X["MedInc"].median())
bin_pipe = Pipeline(steps=[
```

```
    ("bin", Binarizer(threshold=bin_threshold))
])

# ColumnTransformer combining branches (note: binarized MedInc will appear alongside raw/other features)
preprocessor = ColumnTransformer(
    transformers=[
        ("yj", yj_pipe, skewed_cols),
        ("other", pass_pipe, other_cols),
        ("medinc_bin", bin_pipe, bin_col),
    ],
    remainder="drop",
    verbose_feature_names_out=False
)

model = Ridge(alpha=1.0, random_state=42)

pipe_tx = Pipeline(steps=[
    ("prep", preprocessor),
    ("model", model)
])

pipe_raw = Pipeline(steps=[
    ("model", Ridge(alpha=1.0, random_state=42))
])

def rmse(cv_scores):
    return npy.sqrt(-cv_scores.mean())

scores_tx = cross_val_score(pipe_tx, X, target, scoring="neg_mean_squared_error", cv=5, n_jobs=-1)
scores_raw = cross_val_score(pipe_raw, X, target, scoring="neg_mean_squared_error", cv=5, n_jobs=-1)

print("RMSE (no transforms):", rmse(scores_raw))
print("RMSE (with transforms):", rmse(scores_tx))
```

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score
```

```
5
6  target = cal.frame["MedHouseVal"]
7
8  # Heuristic: mark skewed if |skew| >= 1.0
9  skews = X.skew().abs()
10 skewed_cols = skews[skews >= 1.0].index.tolist()
11 other_cols = [c for c in cols if c not in skewed_cols]
12
13 # Pipelines for branches
14 yj_pipe = Pipeline(steps=[
15     ("yeojohn", PowerTransformer(method="yeo-johnson",
       standardize=True))
16 ])
17
18 # "Passthrough" for other columns (you could add
       StandardScaler if desired)
19 pass_pipe = "passthrough"
20
21 # Binarized MedInc as an extra feature (single-column
       pipeline)
22 bin_col = ["MedInc"]
23 bin_threshold = float(X["MedInc"].median())
24 bin_pipe = Pipeline(steps=[
25     ("bin", Binarizer(threshold=bin_threshold))
26 ])
27
28 # ColumnTransformer combining branches (note: binarized
       MedInc will appear alongside raw/other features)
29 preprocessor = ColumnTransformer(
30     transformers=[
31         ("yj", yj_pipe, skewed_cols),
32         ("other", pass_pipe, other_cols),
33         ("medinc_bin", bin_pipe, bin_col),
34     ],
35     remainder="drop",
36     verbose_feature_names_out=False
37 )
38
39 model = Ridge(alpha=1.0, random_state=42)
40
41 pipe_tx = Pipeline(steps=[
```

```
    ("prep", preprocessor),
    ("model", model)
])

pipe_raw = Pipeline(steps=[
    ("model", Ridge(alpha=1.0, random_state=42))
])

def rmse(cv_scores):
    return npy.sqrt(-cv_scores.mean())

scores_tx = cross_val_score(pipe_tx, X, target, scoring="
    neg_mean_squared_error", cv=5, n_jobs=-1)
scores_raw = cross_val_score(pipe_raw, X, target, scoring
    ="neg_mean_squared_error", cv=5, n_jobs=-1)

print("RMSE (no transforms):", rmse(scores_raw))
print("RMSE (with transforms):", rmse(scores_tx))
```

```
RMSE (no transforms): 0.7471805175916729
RMSE (with transforms): 0.7315995010450792
```

How to read the comparison: If RMSE improves with transformations, the preprocessing meaningfully stabilized the feature space for this model. If not, revisit which columns you transform and consider StandardScaler for non-skewed columns.

Summary

- **Log** is a simple, effective fix for positive, right-skewed data.
- **PowerTransformer** (Box-Cox / Yeo-Johnson) learns power parameters; Yeo-Johnson is more general.
- **QuantileTransformer** can enforce near-normal or uniform outputs but is highly non-linear; validate its impact.
- **Binarizer** is useful when a threshold carries most of the signal.
- Combine these steps using **pipelines** so training and inference stay consistent

and reproducible.

Feature Generation

Creating interaction features (`PolynomialFeatures`)

Interaction features capture multiplicative relationships between variables (e.g., `x1 * x2`).
They allow **linear** models to approximate non-linear boundaries without changing the model class.

We'll illustrate with two numerical predictors from California Housing and show how to create:

- *Interactions only* (no squares),
- Full *polynomial* terms up to degree 2 (including squares and interactions).

Caution: Polynomial expansion can explode dimensionality—validate with cross-validation and regularization.

```
import numpy as npy
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import PolynomialFeatures

cal = fetch_california_housing(as_frame=True)
Xh = cal.frame.drop(columns=["MedHouseVal"])
y = cal.frame["MedHouseVal"]
```

```
num_pair = ["MedInc", "HouseAge"]
X_pair = Xh[num_pair].copy()

# Interactions only
poly_int = PolynomialFeatures(degree=2, include_bias=
    False, interaction_only=True)
X_int = pd.DataFrame(poly_int.fit_transform(X_pair),
    columns=poly_int.get_feature_names_out(num_pair))

# Full quadratic (squares + interactions)
poly_full = PolynomialFeatures(degree=2, include_bias=
    False, interaction_only=False)
X_quad = pd.DataFrame(poly_full.fit_transform(X_pair),
    columns=poly_full.get_feature_names_out(num_pair))

print("Interaction-only columns:", X_int.columns.tolist()
    )
print("Quadratic columns:", X_quad.columns.tolist())
display(X_int.head())
display(X_quad.head())
```

```
import numpy as npy
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_california_housing
from sklearn.preprocessing import PolynomialFeatures

cal = fetch_california_housing(as_frame=True)
Xh = cal.frame.drop(columns=["MedHouseVal"])
y = cal.frame["MedHouseVal"]

num_pair = ["MedInc", "HouseAge"]
X_pair = Xh[num_pair].copy()

# Interactions only
poly_int = PolynomialFeatures(degree=2, include_bias=
    False, interaction_only=True)
X_int = pd.DataFrame(poly_int.fit_transform(X_pair),
    columns=poly_int.get_feature_names_out(num_pair))
```

```
18
19  # Full quadratic (squares + interactions)
20  poly_full = PolynomialFeatures(degree=2, include_bias=
        False, interaction_only=False)
21  X_quad = pd.DataFrame(poly_full.fit_transform(X_pair),
        columns=poly_full.get_feature_names_out(num_pair))
22
23  print("Interaction-only columns:", X_int.columns.tolist()
        )
24  print("Quadratic columns:", X_quad.columns.tolist())
25  display(X_int.head())
26  display(X_quad.head())
```

```
1  Interaction-only columns: ['MedInc', 'HouseAge', 'MedInc
       HouseAge']
2  Quadratic columns: ['MedInc', 'HouseAge', 'MedInc^2', '
       MedInc HouseAge', 'HouseAge^2']
```

MedInc

HouseAge

MedInc HouseAge

0

8.3252

41.0

341.3332

1

8.3014

21.0

174.3294

2

7.2574

52.0

377.3848

3

5.6431

52.0

293.4412

4

3.8462

52.0

200.0024

MedInc

HouseAge

MedInc^2

MedInc HouseAge

HouseAge^2

0

8.3252

41.0

69.308955

341.3332

1681.0

1

8.3014

21.0

68.913242

174.3294

441.0

2

7.2574

52.0

52.669855

377.3848

2704.0

3

5.6431

52.0

31.844578

293.4412

2704.0

4

3.8462

52.0

14.793254

200.0024

2704.0

When it helps: If target $\approx$ smooth function of inputs with cross-effects (e.g., income $\times$ house age), interactions let linear models weight those effects directly.

Feature crosses: combining categorical features

Feature crosses join categories to capture **co-occurrence patterns** (e.g., `sex` × `class` in Titanic).
Two common approaches:

1) **Manual cross in pandas**, then one-hot encode.

2) **OneHot → PolynomialFeatures(interaction_only=True)**: create interactions *after* one-hot (watch out for dimensionality).

We'll demonstrate the manual route for clarity and control.

```
import seaborn as sns
import numpy as npy
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.impute import SimpleImputer

titanic = sns.load_dataset("titanic").copy()
X = titanic[["sex", "class", "embarked", "age", "fare"]]
y = titanic["survived"]

# Manual cross of categorical features
X = X.assign(sex_class=X["sex"].astype(str) + "_" + X["
    class"].astype(str))

# Column groups
nominal = ["sex", "embarked", "sex_class"]
numeric = ["age", "fare"]

# OneHotEncoder compatibility shim (sklearn >=1.2 uses
    sparse_output)
try:
```

```
    ohe = OneHotEncoder(handle_unknown="ignore",
        sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=
        False)

# Build preprocessing: impute first (train-only inside CV
    ), then encode / passthrough
pre = ColumnTransformer(
    transformers=[
        # Nominal features: impute most frequent category
            , then one-hot encode
        ("nominal", Pipeline([
            ("imputer", SimpleImputer(strategy="
                most_frequent")),
            ("ohe", ohe),
        ]), nominal),

        # Numeric features: impute median (robust to
            outliers)
        ("numeric", Pipeline([
            ("imputer", SimpleImputer(strategy="median"))
                ,
        ]), numeric),
    ],
    verbose_feature_names_out=False
)

# End-to-end pipeline: preprocessing + classifier
pipe = Pipeline([
    ("prep", pre),
    ("clf", LogisticRegression(max_iter=1000))
])

# Cross-validated accuracy
scores = cross_val_score(pipe, X, y, cv=5, scoring="
    accuracy", n_jobs=-1)
print("Cross-validated accuracy (with crossed feature +
    imputation):", scores.mean().round(3))
```

```
import seaborn as sns
```

```
import numpy as npy
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.impute import SimpleImputer

titanic = sns.load_dataset("titanic").copy()
X = titanic[["sex", "class", "embarked", "age", "fare"]]
y = titanic["survived"]

# Manual cross of categorical features
X = X.assign(sex_class=X["sex"].astype(str) + "_" + X["
    class"].astype(str))

# Column groups
nominal = ["sex", "embarked", "sex_class"]
numeric = ["age", "fare"]

# OneHotEncoder compatibility shim (sklearn >=1.2 uses
    sparse_output)
try:
    ohe = OneHotEncoder(handle_unknown="ignore",
        sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=
        False)

# Build preprocessing: impute first (train-only inside CV
    ), then encode / passthrough
pre = ColumnTransformer(
    transformers=[
        # Nominal features: impute most frequent category
            , then one-hot encode
        ("nominal", Pipeline([
            ("imputer", SimpleImputer(strategy="
                most_frequent")),
            ("ohe", ohe),
        ]), nominal),
```

```
36
37          # Numeric features: impute median (robust to
                outliers)
38          ("numeric", Pipeline([
39              ("imputer", SimpleImputer(strategy="median"))
                    ,
40          ]), numeric),
41      ],
42      verbose_feature_names_out=False
43  )
44
45  # End-to-end pipeline: preprocessing + classifier
46  pipe = Pipeline([
47      ("prep", pre),
48      ("clf", LogisticRegression(max_iter=1000))
49  ])
50
51  # Cross-validated accuracy
52  scores = cross_val_score(pipe, X, y, cv=5, scoring="
        accuracy", n_jobs=-1)
53  print("Cross-validated accuracy (with crossed feature +
        imputation):", scores.mean().round(3))
```

```
1  Cross-validated accuracy (with crossed feature +
       imputation): 0.797
```

Tip: Crosses are powerful when certain pairs of categories behave differently than each one alone (e.g., `male` in `First` class vs `Third` class).

Extracting date/time features

Timestamps encode **periodic** and **calendar** signals (seasonality, weekday vs weekend, rush hours).
Typical extractions:

- `year`, `month`, `day`, `weekday`, `hour`, `weekofyear`.
- **Cyclical encodings** for periodic variables (e.g., hour-of-day) using `sin`/`cos`.

Below we create a small example DataFrame with timestamps and extract informative features.

```
import pandas as pd
import numpy as npy
import matplotlib.pyplot as plt

# Synthetic event timestamps (hourly for ~2 weeks)
ts = pd.date_range("2023-05-01", periods=24*14, freq="H")
events = pd.DataFrame({
    "timestamp": ts,
    "value": npy.random.default_rng(0).normal(loc=0,
        scale=1, size=len(ts))
})

# Basic calendar features
events["year"] = events["timestamp"].dt.year
events["month"] = events["timestamp"].dt.month
events["day"] = events["timestamp"].dt.day
events["weekday"] = events["timestamp"].dt.weekday  # 0=
    Mon
events["hour"] = events["timestamp"].dt.hour
events["is_weekend"] = events["weekday"].isin([5, 6]).
    astype(int)

# Cyclical encodings for hour-of-day
events["hour_sin"] = npy.sin(2*npy.pi*events["hour"]/24)
events["hour_cos"] = npy.cos(2*npy.pi*events["hour"]/24)

display(events.head())

# (Adds value) Quick weekday count plot to show extracted
     structure
weekday_counts = events["weekday"].value_counts().
    sort_index()
plt.figure(figsize=(5,3))
plt.bar(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"],
    weekday_counts)
plt.title("Event counts by weekday")
plt.tight_layout()
plt.savefig("fig_events_weekday_counts.png", dpi=150,
```

```
    bbox_inches="tight")
plt.show()
```

```
import pandas as pd
import numpy as npy
import matplotlib.pyplot as plt

# Synthetic event timestamps (hourly for ~2 weeks)
ts = pd.date_range("2023-05-01", periods=24*14, freq="H")
events = pd.DataFrame({
    "timestamp": ts,
    "value": npy.random.default_rng(0).normal(loc=0,
        scale=1, size=len(ts))
})

# Basic calendar features
events["year"] = events["timestamp"].dt.year
events["month"] = events["timestamp"].dt.month
events["day"] = events["timestamp"].dt.day
events["weekday"] = events["timestamp"].dt.weekday  # 0=
    Mon
events["hour"] = events["timestamp"].dt.hour
events["is_weekend"] = events["weekday"].isin([5, 6]).
    astype(int)

# Cyclical encodings for hour-of-day
events["hour_sin"] = npy.sin(2*npy.pi*events["hour"]/24)
events["hour_cos"] = npy.cos(2*npy.pi*events["hour"]/24)

display(events.head())

# (Adds value) Quick weekday count plot to show extracted
     structure
weekday_counts = events["weekday"].value_counts().
    sort_index()
plt.figure(figsize=(5,3))
plt.bar(["Mon","Tue","Wed","Thu","Fri","Sat","Sun"],
    weekday_counts)
plt.title("Event counts by weekday")
plt.tight_layout()
plt.show()
```

timestamp

value

year

month

day

weekday

hour

is_weekend

hour_sin

hour_cos

0

2023-05-01 00:00:00

0.125730

2023

5

1

0

0

0

0.000000

1.000000

1

2023-05-01 01:00:00

-0.132105

2023

5

1

0

1

0

0.258819

0.965926

2

2023-05-01 02:00:00

0.640423

2023

5

1

0

2

0

0.500000

0.866025

3

2023-05-01 03:00:00

0.104900

2023

5

1

0

3

0

0.707107

0.707107

4

2023-05-01 04:00:00

-0.535669

2023

5

1

0

4

0

0.866025

0.500000

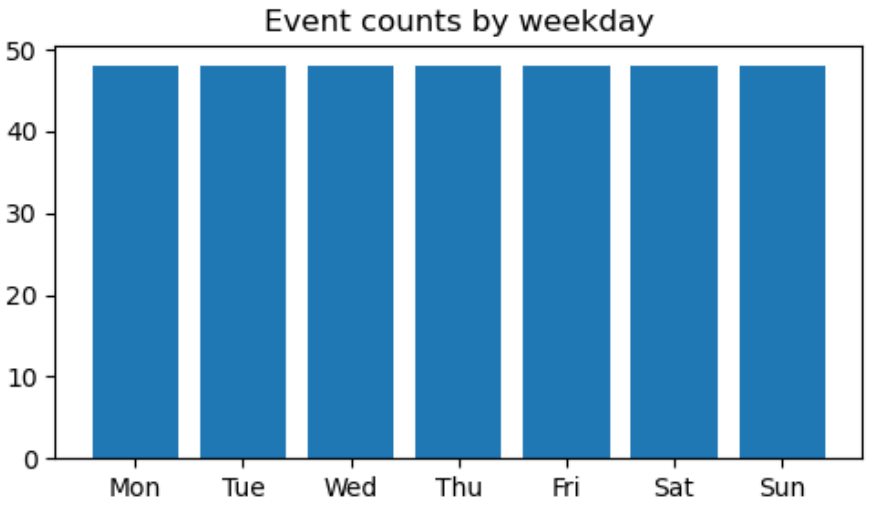

Figure 1: Image generated by the provided code.

Why cyclical features? `23:00` and `00:00` are adjacent in time but far as integers. Sine/cosine preserve circular adjacency for models that read numbers literally.

Text preprocessing basics: Bag-of-Words and TF-IDF

For short texts, two foundations:

- **Bag-of-Words (CountVectorizer)**: counts token occurrences per document.
- **TF-IDF (TfidfVectorizer)**: down-weights ubiquitous tokens, up-weights distinctive ones.

We'll use a tiny corpus and inspect the top-weighted terms.

```python
from sklearn.feature_extraction.text import
    CountVectorizer, TfidfVectorizer
import numpy as npy

corpus = [
    "Feature engineering turns data into signal",
    "Good features help simple models compete",
    "Pipelines keep preprocessing reproducible",
    "Scaling and encoding are essential steps",
]

# Bag of Words
cv = CountVectorizer(stop_words="english")
X_bow = cv.fit_transform(corpus)
vocab = npy.array(cv.get_feature_names_out())
print("BOW shape:", X_bow.shape)

# TF-IDF
tfidf = TfidfVectorizer(stop_words="english")
X_tfidf = tfidf.fit_transform(corpus)
terms = npy.array(tfidf.get_feature_names_out())

# Show top-5 terms by weight for each doc
for i, doc in enumerate(corpus):
    row = X_tfidf.getrow(i).toarray().ravel()
    top_idx = row.argsort()[-5:][::-1]
    print(f"\nDoc {i}:", doc)
    print("Top terms:", terms[top_idx].tolist(), "weights
        :", row[top_idx].round(3).tolist())
```

```python
from sklearn.feature_extraction.text import
    CountVectorizer, TfidfVectorizer
import numpy as npy

corpus = [
    "Feature engineering turns data into signal",
    "Good features help simple models compete",
    "Pipelines keep preprocessing reproducible",
    "Scaling and encoding are essential steps",
]

```

```
# Bag of Words
cv = CountVectorizer(stop_words="english")
X_bow = cv.fit_transform(corpus)
vocab = npy.array(cv.get_feature_names_out())
print("BOW shape:", X_bow.shape)

# TF-IDF
tfidf = TfidfVectorizer(stop_words="english")
X_tfidf = tfidf.fit_transform(corpus)
terms = npy.array(tfidf.get_feature_names_out())

# Show top-5 terms by weight for each doc
for i, doc in enumerate(corpus):
    row = X_tfidf.getrow(i).toarray().ravel()
    top_idx = row.argsort()[-5:][::-1]
    print(f"\nDoc {i}:", doc)
    print("Top terms:", terms[top_idx].tolist(), "weights
        :", row[top_idx].round(3).tolist())
```

```
BOW shape: (4, 18)

Doc 0: Feature engineering turns data into signal
Top terms: ['turns', 'signal', 'engineering', 'data', '
    feature'] weights: [0.447, 0.447, 0.447, 0.447,
    0.447]

Doc 1: Good features help simple models compete
Top terms: ['help', 'simple', 'features', 'compete', '
    models'] weights: [0.408, 0.408, 0.408, 0.408, 0.408]

Doc 2: Pipelines keep preprocessing reproducible
Top terms: ['preprocessing', 'reproducible', 'pipelines',
     'turns', 'steps'] weights: [0.577, 0.577, 0.577,
    0.0, 0.0]

Doc 3: Scaling and encoding are essential steps
Top terms: ['steps', 'scaling', 'encoding', 'essential',
    'turns'] weights: [0.5, 0.5, 0.5, 0.5, 0.0]
```

Guidance: For real projects, add tokenization/normalization (lowercasing, stem-

ming/lemmatization), n-grams, character models for misspellings, or domain vocabularies.

Domain-driven features (California Housing)

Domain knowledge often beats blind expansion. Examples for this dataset:

- **Bedrooms ratio**: `AveBedrms / AveRooms` (how many bedrooms per room).
- **Rooms per occupant**: `AveRooms / AveOccup` (space per person).
- **Income per occupant**: `MedInc / AveOccup` (ability to pay).
- **Coarse geospatial bins**: latitude/longitude buckets to approximate regions.

We'll implement a small transformer that **adds** these features, and show how to integrate it into a pipeline.

```
from sklearn.base import BaseEstimator, TransformerMixin

class AddHousingFeatures(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.new_cols_ = ["bedrooms_ratio", "rooms_per_occup", "income_per_occup", "lat_bin", "lon_bin"]
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X = X.copy()
        # Guard against divide-by-zero
        eps = 1e-9
        X["bedrooms_ratio"] = (X["AveBedrms"] / (X["AveRooms"] + eps)).clip(0, None)
```

```
        X["rooms_per_occup"] = (X["AveRooms"] / (X["
            AveOccup"] + eps)).clip(0, None)
        X["income_per_occup"] = (X["MedInc"] / (X["
            AveOccup"] + eps)).clip(0, None)
        # Coarse bins (tune bin edges to your domain/
            region)
        X["lat_bin"] = pd.cut(X["Latitude"], bins=[32,
            34, 36, 38, 40, 42], labels=False,
            include_lowest=True)
        X["lon_bin"] = pd.cut(X["Longitude"], bins=[-125,
             -123, -121, -119, -117, -115], labels=False,
             include_lowest=True)
        return X

# Demo: transform a copy and preview
demo = AddHousingFeatures().fit_transform(Xh.copy())
display(demo[["AveRooms","AveBedrms","AveOccup","MedInc",
    "bedrooms_ratio","rooms_per_occup","income_per_occup"
    ,"lat_bin","lon_bin"]].head())
```

```
from sklearn.base import BaseEstimator, TransformerMixin

class AddHousingFeatures(BaseEstimator, TransformerMixin)
    :
    def __init__(self):
        self.new_cols_ = ["bedrooms_ratio", "
            rooms_per_occup", "income_per_occup", "
            lat_bin", "lon_bin"]
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        X = X.copy()
        # Guard against divide-by-zero
        eps = 1e-9
        X["bedrooms_ratio"] = (X["AveBedrms"] / (X["
            AveRooms"] + eps)).clip(0, None)
        X["rooms_per_occup"] = (X["AveRooms"] / (X["
            AveOccup"] + eps)).clip(0, None)
        X["income_per_occup"] = (X["MedInc"] / (X["
            AveOccup"] + eps)).clip(0, None)
        # Coarse bins (tune bin edges to your domain/
```

```
            region)
        X["lat_bin"] = pd.cut(X["Latitude"], bins=[32,
            34, 36, 38, 40, 42], labels=False,
            include_lowest=True)
        X["lon_bin"] = pd.cut(X["Longitude"], bins=[-125,
             -123, -121, -119, -117, -115], labels=False,
             include_lowest=True)
        return X

# Demo: transform a copy and preview
demo = AddHousingFeatures().fit_transform(Xh.copy())
display(demo[["AveRooms","AveBedrms","AveOccup","MedInc",
    "bedrooms_ratio","rooms_per_occup","income_per_occup"
    ,"lat_bin","lon_bin"]].head())
```

AveRooms

AveBedrms

AveOccup

MedInc

bedrooms_ratio

rooms_per_occup

income_per_occup

lat_bin

lon_bin

0

6.984127

1.023810

2.555556

8.3252

0.146591

2.732919

3.257687

2

1.0

1

6.238137

0.971880

2.109842

8.3014

0.155797

2.956685

3.934608

2

1.0

2

8.288136

1.073446

2.802260

7.2574

0.129516

2.957661

2.589838

2

1.0

3

5.817352

1.073059

2.547945

5.6431

0.184458

2.283154

2.214765

2

1.0

4

6.281853

1.081081

2.181467

3.8462

0.172096

2.879646

1.763125

2

1.0

Pipelines with generators: Place the feature generator at the **front** of your pipeline so downstream scalers/encoders see both original and derived columns consistently.

```
import numpy as npy
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# 1) Carga y define X/y del *mismo* dataset (California Housing)
cal = fetch_california_housing(as_frame=True)
Xh = cal.frame.drop(columns=["MedHouseVal"]).copy()
y_h = cal.frame["MedHouseVal"].copy()

# 2) Columnas esperadas tras el generador AddHousingFeatures (definido previamente)
num_cols = [
    "MedInc","HouseAge","AveRooms","AveBedrms","Population","AveOccup","Latitude","Longitude",
    "bedrooms_ratio","rooms_per_occup","income_per_occup"
]
cat_cols = ["lat_bin","lon_bin"]

# 3) OneHotEncoder: compatibilidad (>=1.2 usa sparse_output)
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

# 4) Preprocesado: escala numéricas y one-hot para bins geográficos
```

```
pre = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
    verbose_feature_names_out=False
)

# 5) Pipeline completo con el generador de *domain features*
pipe_domain = Pipeline([
    ("gen", AddHousingFeatures()),   # añade bedrooms_ratio, rooms_per_occup, income_per_occup, lat_bin, lon_bin
    ("prep", pre),
    ("model", Ridge(alpha=1.0, random_state=42))
])

# 6) Baseline en crudo (sin features de dominio ni escalado)
baseline = Ridge(alpha=1.0, random_state=42)

# 7) Comparativa CV (métrica: RMSE)
def rmse(scores):
    return npy.sqrt(-scores.mean())

scores_raw = cross_val_score(baseline, Xh, y_h, cv=5, scoring="neg_mean_squared_error", n_jobs=-1)
scores_domain = cross_val_score(pipe_domain, Xh, y_h, cv=5, scoring="neg_mean_squared_error", n_jobs=-1)

print("RMSE (raw features):    ", rmse(scores_raw))
print("RMSE (domain features): ", rmse(scores_domain))
```

```
import numpy as npy
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler,
```

```
    OneHotEncoder
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# 1) Carga y define X/y del *mismo* dataset (California Housing)
cal = fetch_california_housing(as_frame=True)
Xh = cal.frame.drop(columns=["MedHouseVal"]).copy()
y_h = cal.frame["MedHouseVal"].copy()

# 2) Columnas esperadas tras el generador AddHousingFeatures (definido previamente)
num_cols = [
    "MedInc","HouseAge","AveRooms","AveBedrms","Population","AveOccup","Latitude","Longitude",
    "bedrooms_ratio","rooms_per_occup","income_per_occup"
]
cat_cols = ["lat_bin","lon_bin"]

# 3) OneHotEncoder: compatibilidad (>=1.2 usa sparse_output)
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

# 4) Preprocesado: escala numéricas y one-hot para bins geográficos
pre = ColumnTransformer(
    transformers=[
        ("num", StandardScaler(), num_cols),
        ("cat", ohe, cat_cols),
    ],
    remainder="drop",
    verbose_feature_names_out=False
)

# 5) Pipeline completo con el generador de *domain features*
```

```
pipe_domain = Pipeline([
    ("gen", AddHousingFeatures()),   # añade bedrooms_ratio, rooms_per_occup, income_per_occup, lat_bin, lon_bin
    ("prep", pre),
    ("model", Ridge(alpha=1.0, random_state=42))
])

# 6) Baseline en crudo (sin features de dominio ni escalado)
baseline = Ridge(alpha=1.0, random_state=42)

# 7) Comparativa CV (métrica: RMSE)
def rmse(scores):
    return npy.sqrt(-scores.mean())

scores_raw = cross_val_score(baseline, Xh, y_h, cv=5, scoring="neg_mean_squared_error", n_jobs=-1)
scores_domain = cross_val_score(pipe_domain, Xh, y_h, cv=5, scoring="neg_mean_squared_error", n_jobs=-1)

print("RMSE (raw features):    ", rmse(scores_raw))
print("RMSE (domain features): ", rmse(scores_domain))
```

```
RMSE (raw features):     0.7471805175916729
RMSE (domain features):  0.697073317297543
```

Reading the comparison: If RMSE drops with domain features, you've added meaningful signal. If it doesn't, revisit definitions and consider interactions or non-linear models.

Summary

- **Interactions** (`PolynomialFeatures`) let linear models capture cross-effects; control degree to avoid feature explosion.
- **Feature crosses** combine categories to capture co-occurrence patterns; en-

code after crossing.

- **Date/time** features should include both calendar fields and **cyclical encodings** for periodic variables.
- **Text** basics: Bag-of-Words and TF-IDF provide strong baselines; enrich with n-grams and normalization for real workloads.
- **Domain-driven** features leverage the problem's structure and often yield outsized gains with simple models.

Handling Missing Data

Why missing data matters (MCAR, MAR, MNAR)

Real-world datasets almost always have missing values. Ignoring them can bias estimates or break models.

- **MCAR** (Missing Completely At Random): missingness unrelated to any variable → least harmful.
- **MAR** (Missing At Random): missingness depends on observed variables (e.g., `age` missing more often for a given **`class`**).
- **MNAR** (Missing Not At Random): missingness depends on the unobserved value itself (hardest case).

Key rules: - **Never** impute using information from the validation/test split → use **pipelines** so imputers learn from *train* only. - Consider adding **missingness indicators** to help the model learn patterns in what is missing.

Inspecting missingness

We will look at the seaborn `titanic` dataset and quantify missing values by column.
A simple **bar chart** of missing counts is often more actionable than a heatmap.

```
1  import pandas as pd
```

```
import matplotlib.pyplot as plt
import seaborn as sns

titanic = sns.load_dataset("titanic").copy()
missing_counts = titanic.isna().sum().sort_values(
    ascending=False)
missing_pct = (missing_counts / len(titanic)).round(3)

miss_tbl = pd.DataFrame({"missing_count": missing_counts,
     "missing_pct": missing_pct})
display(miss_tbl[miss_tbl["missing_count"] > 0])

# Bar chart (adds value: shows which cols matter)
plt.figure(figsize=(8,4))
missing_counts[missing_counts>0].plot(kind="bar")
plt.ylabel("count of NaN")
plt.title("Titanic: missing values by column")
plt.tight_layout()
plt.show()
```

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

titanic = sns.load_dataset("titanic").copy()
missing_counts = titanic.isna().sum().sort_values(
    ascending=False)
missing_pct = (missing_counts / len(titanic)).round(3)

miss_tbl = pd.DataFrame({"missing_count": missing_counts,
     "missing_pct": missing_pct})
display(miss_tbl[miss_tbl["missing_count"] > 0])

# Bar chart (adds value: shows which cols matter)
plt.figure(figsize=(8,4))
missing_counts[missing_counts>0].plot(kind="bar")
plt.ylabel("count of NaN")
plt.title("Titanic: missing values by column")
plt.tight_layout()
plt.show()
```

missing_count

missing_pct

deck

688

0.772

age

177

0.199

embarked

2

0.002

embark_town

2

0.002

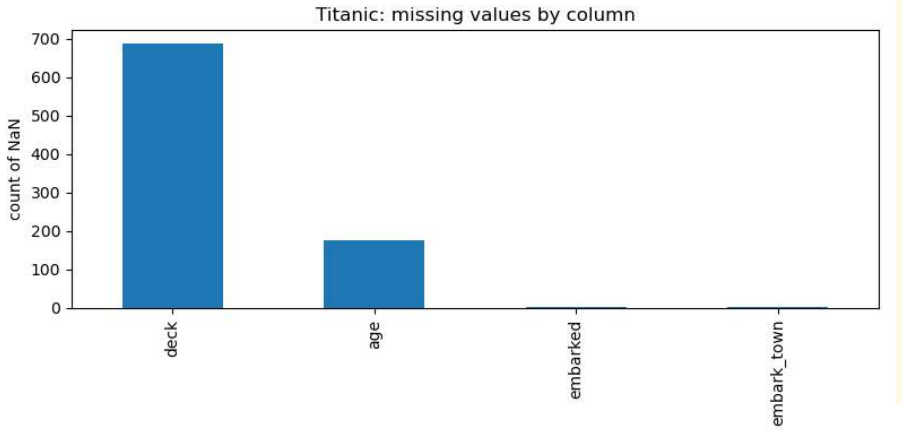

Figure 1: Image generated by the provided code.

Reading the table/plot: `age`, `embarked`, and especially `deck` often have missing values. We'll focus on features used in earlier chapters (`sex`, **`class`**, `embarked`, `age`, `fare`) to keep the pipeline consistent.

Simple imputation (mean/median/most-frequent/constant)

SimpleImputer is a strong baseline:

- **Numeric**: `median` is robust to outliers; `mean` if distribution is symmetric.
- **Categorical**: `most_frequent` or a **constant** like `"Unknown"`.

We will build a pipeline for Titanic classification with **imputation + encoding + logistic regression**.

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

```
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

X = titanic[["sex", "class", "embarked", "age", "fare"]].copy()
y = titanic["survived"].copy()

nominal = ["sex", "embarked"]
ordinal = ["class"]
numeric = ["age", "fare"]

# OneHotEncoder version-compatibility shim
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)

pre_simple = ColumnTransformer(
    transformers=[
        ("nominal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ]), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            # encode 'class' as ordinal  Third < Second < First
            ("ord", OneHotEncoder(handle_unknown="ignore", drop=None, sparse_output=False)  # simple OHE for class
             if hasattr(OneHotEncoder, "sparse_output")
             else OneHotEncoder(handle_unknown="ignore", drop=None, sparse_output=False)),
        ]), ordinal),
        ("numeric", Pipeline([
```

```
            ("imp", SimpleImputer(strategy="median")),
        ]), numeric),
    ],
    verbose_feature_names_out=False
)

pipe_simple = Pipeline([
    ("prep", pre_simple),
    ("clf", LogisticRegression(max_iter=1000))
])

scores = cross_val_score(pipe_simple, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with simple imputation:", scores.mean().round(3))
```

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

X = titanic[["sex", "class", "embarked", "age", "fare"]].copy()
y = titanic["survived"].copy()

nominal = ["sex", "embarked"]
ordinal = ["class"]
numeric = ["age", "fare"]

# OneHotEncoder version-compatibility shim
try:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)
except TypeError:
    ohe = OneHotEncoder(handle_unknown="ignore", sparse_output=False)

pre_simple = ColumnTransformer(
    transformers=[
```

```
        ("nominal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ]), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            # encode 'class' as ordinal  Third < Second < First
            ("ord", OneHotEncoder(handle_unknown="ignore", drop=None, sparse_output=False)  # simple OHE for class
             if hasattr(OneHotEncoder, "sparse_output")
             else OneHotEncoder(handle_unknown="ignore", drop=None, sparse_output=False)),
        ]), ordinal),
        ("numeric", Pipeline([
            ("imp", SimpleImputer(strategy="median")),
        ]), numeric),
    ],
    verbose_feature_names_out=False
)

pipe_simple = Pipeline([
    ("prep", pre_simple),
    ("clf", LogisticRegression(max_iter=1000))
])

scores = cross_val_score(pipe_simple, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with simple imputation:", scores.mean().round(3))
```

```
Accuracy with simple imputation: 0.789
```

Adding missingness indicators

Sometimes **whether a value is missing is predictive** (e.g., people without recorded `age` differ systematically).
We can append indicator columns. The simplest path is `SimpleImputer(add_indicator=True)` (if supported) or `MissingIndicator`.

We'll attempt `add_indicator=True` and fall back to `MissingIndicator` if needed.

```
from sklearn.impute import MissingIndicator

def nominal_block_with_indicator():
    try:
        # SimpleImputer can append indicators in recent scikit-learn
        return Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent", add_indicator=True)),
            ("ohe", ohe),
        ])
    except TypeError:
        # Fallback: separate indicator and stack via ColumnTransformer
        return Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ])

def numeric_block_with_indicator():
    try:
        return Pipeline([
            ("imp", SimpleImputer(strategy="median", add_indicator=True)),
        ])
    except TypeError:
        return Pipeline([
            ("imp", SimpleImputer(strategy="median")),
```

```
        ])

pre_with_ind = ColumnTransformer(
    transformers=[
        ("nominal", nominal_block_with_indicator(), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ]), ordinal),
        ("numeric", numeric_block_with_indicator(), numeric),
    ],
    verbose_feature_names_out=False
)

pipe_with_ind = Pipeline([
    ("prep", pre_with_ind),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_ind = cross_val_score(pipe_with_ind, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with indicators:", scores_ind.mean().round(3))
```

```
from sklearn.impute import MissingIndicator

def nominal_block_with_indicator():
    try:
        # SimpleImputer can append indicators in recent scikit-learn
        return Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent", add_indicator=True)),
            ("ohe", ohe),
        ])
    except TypeError:
        # Fallback: separate indicator and stack via ColumnTransformer
```

```
        return Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ])

def numeric_block_with_indicator():
    try:
        return Pipeline([
            ("imp", SimpleImputer(strategy="median", add_indicator=True)),
        ])
    except TypeError:
        return Pipeline([
            ("imp", SimpleImputer(strategy="median")),
        ])

pre_with_ind = ColumnTransformer(
    transformers=[
        ("nominal", nominal_block_with_indicator(), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent")),
            ("ohe", ohe),
        ]), ordinal),
        ("numeric", numeric_block_with_indicator(), numeric),
    ],
    verbose_feature_names_out=False
)

pipe_with_ind = Pipeline([
    ("prep", pre_with_ind),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_ind = cross_val_score(pipe_with_ind, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with indicators:", scores_ind.mean().round(3))
```

```
Accuracy with indicators: 0.799
```

Interpreting the result: If accuracy improves, the model is benefiting from the information contained in the pattern of missingness.

KNNImputer (neighbors-based imputation)

KNNImputer imputes a value from **similar rows** (nearest neighbors).
Pros: uses multivariate structure. Cons: slower, sensitive to scale and sparse categories.

We will:

- One-hot encode categoricals first (so distance is meaningful),
- Scale numeric columns (optional but recommended),
- Then apply KNNImputer on the **combined** feature space.

```
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler

pre_knn = ColumnTransformer(
    transformers=[
        ("nominal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent
                ")),
            ("ohe", ohe),
        ]), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent
                ")),
            ("ohe", ohe),
        ]), ordinal),
        ("numeric", Pipeline([
            ("imp", SimpleImputer(strategy="median")),
            ("sc", StandardScaler()),
        ]), numeric),
```

```
    ],
    verbose_feature_names_out=False
)

pipe_knn = Pipeline([
    ("prep", pre_knn),
    ("knnimp", KNNImputer(n_neighbors=5, weights="
        distance")),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_knn = cross_val_score(pipe_knn, X, y, cv=5,
    scoring="accuracy", n_jobs=-1)
print("Accuracy with KNNImputer:", scores_knn.mean().
    round(3))
```

```
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler

pre_knn = ColumnTransformer(
    transformers=[
        ("nominal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent
                ")),
            ("ohe", ohe),
        ]), nominal),
        ("ordinal", Pipeline([
            ("imp", SimpleImputer(strategy="most_frequent
                ")),
            ("ohe", ohe),
        ]), ordinal),
        ("numeric", Pipeline([
            ("imp", SimpleImputer(strategy="median")),
            ("sc", StandardScaler()),
        ]), numeric),
    ],
    verbose_feature_names_out=False
)

pipe_knn = Pipeline([
    ("prep", pre_knn),
```

```
    ("knnimp", KNNImputer(n_neighbors=5, weights="
        distance")),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_knn = cross_val_score(pipe_knn, X, y, cv=5,
    scoring="accuracy", n_jobs=-1)
print("Accuracy with KNNImputer:", scores_knn.mean().
    round(3))
```

```
Accuracy with KNNImputer: 0.789
```

Notes:

- KNNImputer occurs **after** encoding/scaling so distances reflect all features.
- Tune `n_neighbors`; `weights="distance"` can help when neighbors vary in similarity.

IterativeImputer (multivariate, model-based)

IterativeImputer models each feature with missing values as a function of other features (round-robin).
Pros: powerful when relationships are approximately linear/non-linear depending on estimator.
Cons: heavier, more hyperparameters.

We'll use a linear estimator (default BayesianRidge) for speed and stability.

```
# Enable IterativeImputer if needed in older versions
try:
    from sklearn.impute import IterativeImputer
except Exception:
    from sklearn.experimental import
        enable_iterative_imputer  # noqa: F401
    from sklearn.impute import IterativeImputer

```

```
pre_iter = pre_knn  # reuse encoding/scaling block from above

pipe_iter = Pipeline([
    ("prep", pre_iter),
    ("iterimp", IterativeImputer(max_iter=10, random_state=42, sample_posterior=False)),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_iter = cross_val_score(pipe_iter, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with IterativeImputer:", scores_iter.mean().round(3))
```

```
# Enable IterativeImputer if needed in older versions
try:
    from sklearn.impute import IterativeImputer
except Exception:
    from sklearn.experimental import enable_iterative_imputer  # noqa: F401
    from sklearn.impute import IterativeImputer

pre_iter = pre_knn  # reuse encoding/scaling block from above

pipe_iter = Pipeline([
    ("prep", pre_iter),
    ("iterimp", IterativeImputer(max_iter=10, random_state=42, sample_posterior=False)),
    ("clf", LogisticRegression(max_iter=1000))
])

scores_iter = cross_val_score(pipe_iter, X, y, cv=5, scoring="accuracy", n_jobs=-1)
print("Accuracy with IterativeImputer:", scores_iter.mean().round(3))
```

```
Accuracy with IterativeImputer: 0.789
```

Visual check: distribution before/after imputation (adds value)

We will compare the **age** distribution before imputation vs after simple median imputation.
This helps verify that imputation did not create unrealistic spikes or distortions.

```
# Extract 'age' before and after a simple median imputation for plotting
age_raw = X["age"]

median_imp = SimpleImputer(strategy="median")
age_imp = pd.Series(median_imp.fit_transform(X[["age"]]).ravel(), name="age_imputed")

plt.figure(figsize=(8,4))
plt.hist(age_raw.dropna(), bins=20, alpha=0.6, label="age (raw)")
plt.hist(age_imp, bins=20, alpha=0.6, label="age (median-imputed)")
plt.legend()
plt.title("Age distribution: raw vs median-imputed")
plt.tight_layout()
plt.show()
```


```
# Extract 'age' before and after a simple median imputation for plotting
age_raw = X["age"]

median_imp = SimpleImputer(strategy="median")
age_imp = pd.Series(median_imp.fit_transform(X[["age"]]).ravel(), name="age_imputed")

plt.figure(figsize=(8,4))
plt.hist(age_raw.dropna(), bins=20, alpha=0.6, label="age (raw)")
plt.hist(age_imp, bins=20, alpha=0.6, label="age (median-imputed)")
```

```
plt.legend()
plt.title("Age distribution: raw vs median-imputed")
plt.tight_layout()
plt.show()
```

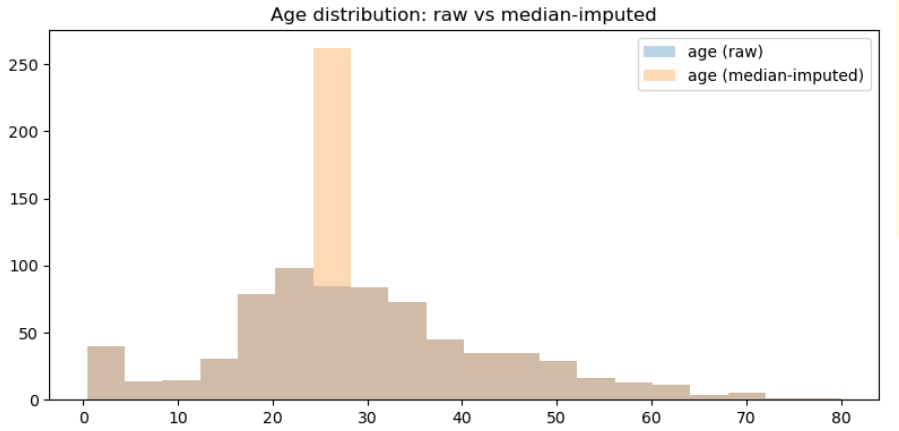

Figure 2: Image generated by the provided code.

Reading the plot: Median imputation fills gaps without changing the central tendency, but can **inflate** the median bin. Advanced imputers (KNN/Iterative) can preserve shape better—validate with metrics.

Leakage-safe evaluation: quick comparison

Let's summarize the **CV accuracy** for the different strategies we tried.
Because each imputer lives *inside* the pipeline, there is **no leakage** from validation folds.

```
import pandas as pd

```

```
acc_table = pd.DataFrame({
    "SimpleImputer": [scores.mean() for scores in [scores]][0],
    "WithIndicators": [scores_ind.mean() for scores_ind in [scores_ind]][0],
    "KNNImputer": [scores_knn.mean() for scores_knn in [scores_knn]][0],
    "IterativeImputer": [scores_iter.mean() for scores_iter in [scores_iter]][0],
}, index=["CV Accuracy"]).T.sort_values(by="CV Accuracy", ascending=False)

display(acc_table.round(3))
```

```
import pandas as pd

acc_table = pd.DataFrame({
    "SimpleImputer": [scores.mean() for scores in [scores]][0],
    "WithIndicators": [scores_ind.mean() for scores_ind in [scores_ind]][0],
    "KNNImputer": [scores_knn.mean() for scores_knn in [scores_knn]][0],
    "IterativeImputer": [scores_iter.mean() for scores_iter in [scores_iter]][0],
}, index=["CV Accuracy"]).T.sort_values(by="CV Accuracy", ascending=False)

display(acc_table.round(3))
```

CV Accuracy

WithIndicators

0.799

SimpleImputer

0.789

KNNImputer

0.789

IterativeImputer

0.789

Summary

- Start with **SimpleImputer** (median for numeric, most-frequent for categorical). It's robust and fast.
- If performance suffers and you suspect structure in missingness, try **missingness indicators**.
- Consider **KNNImputer** or **IterativeImputer** when variables have strong relationships and simple strategies underperform.
- Keep imputers **inside pipelines** to avoid leakage. Tune imputer hyperparameters with cross-validation as you would any model.
- For **MNAR**, modeling missingness explicitly (two-stage models, selection models) or collecting more data may be necessary.


rubió
Metabolomics