# Essential Guide to Scikit-Learn

**Machine learning without the black box.**

Learn machine learning step by step with scikit-learn. Build, evaluate, and interpret predictive models with ease

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 Scikit-Learn

Ibon Martínez-Arranz

Life
Feels
Good

rubió
Metabolomics

# Contents

# Essential Guide to Scikit-Learn

# Introduction and Setup

In this chapter, we will introduce both the **scikit-learn** library and the basic concepts of **Machine Learning (ML)**. This is the starting point for learning how to apply machine learning algorithms in practice using Python.

## What is Machine Learning?

Machine Learning is a branch of Artificial Intelligence (AI) that allows computers to learn from data without being explicitly programmed to perform specific tasks.

Instead of writing rules, we provide data and let the system learn patterns and relationships to make predictions or decisions.

### When is Machine Learning used?

You will typically apply Machine Learning when:

- You have enough data but no clear rules to solve a task.
- You want to automate tasks involving predictions.
- You need to find hidden patterns in large datasets.

### Main types of Machine Learning

- **Supervised Learning**: The algorithm learns from labeled data (e.g., images labeled as cats or dogs).
- **Unsupervised Learning**: The algorithm finds patterns without labels (e.g., customer segmentation).

## What is scikit-learn?

**scikit-learn** is one of the most popular Python libraries for Machine Learning. It provides simple and efficient tools for:

- Classification
- Regression
- Clustering
- Dimensionality Reduction
- Preprocessing
- Model Selection and Evaluation

It is designed to be simple, consistent, and interoperable with other essential libraries like `numpy`, `scipy`, `pandas`, and `matplotlib`.

## Installation

Using `pip`:

```
pip install scikit-learn
```

Or, if you use Anaconda:

```
conda install scikit-learn
```

## Basic Imports

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
```

# Built-in Datasets

scikit-learn provides several classical datasets that we will use throughout the book:

- **Iris**: Characteristics of iris flowers.
- **Wine**: Chemical analysis of wines.
- **Breast Cancer**: Features computed from breast cancer images.
- **Diabetes**: Data for regression tasks.
- **Digits**: Images of handwritten digits.

## Example: First Look at the Iris Dataset

```python
iris = datasets.load_iris(as_frame=True)
iris_df = iris.frame
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 | 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 |

| | sepal length (cm) | sepal width (cm) | petal length (cm) | petal width (cm) | target |
|---|---|---|---|---|---|
| 4 | 5 | 3.6 | 1.4 | 0.2 | 0 |

This dataset contains 150 samples of iris flowers, classified into three species (`setosa`, `versicolor`, `virginica`) based on four features: sepal length, sepal width, petal length, and petal width.

**Example: First Look at the Wine Dataset**

```python
wine = datasets.load_wine(as_frame=True)
wine_df = wine.frame
wine_df.iloc[:5,:5]
```

| | alcohol | malic_acid | ash | alcalinity_of_ash | magnesium |
|---|---|---|---|---|---|
| 0 | 14.23 | 1.71 | 2.43 | 15.6 | 127 |
| 1 | 13.2 | 1.78 | 2.14 | 11.2 | 100 |
| 2 | 13.16 | 2.36 | 2.67 | 18.6 | 101 |
| 3 | 14.37 | 1.95 | 2.5 | 16.8 | 113 |
| 4 | 13.24 | 2.59 | 2.87 | 21 | 118 |

This dataset contains results from a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. It is often used for classification tasks.

## Example: First Look at the Breast Cancer Dataset

```python
breast_cancer = datasets.load_breast_cancer(as_frame=True
    )
breast_cancer_df = breast_cancer.frame
breast_cancer_df.iloc[:5,:5]
```

| | mean radius | mean texture | mean perimeter | mean area | mean smoothness |
|---|---|---|---|---|---|
| 0 | 17.99 | 10.38 | 122.8 | 1001 | 0.1184 |
| 1 | 20.57 | 17.77 | 132.9 | 1326 | 0.08474 |
| 2 | 19.69 | 21.25 | 130 | 1203 | 0.1096 |
| 3 | 11.42 | 20.38 | 77.58 | 386.1 | 0.1425 |
| 4 | 20.29 | 14.34 | 135.1 | 1297 | 0.1003 |

The dataset provides features computed from digitized images of breast masses. It is commonly used for binary classification (malignant vs benign).

## Example: First Look at the Diabetes Dataset

```python
diabetes = datasets.load_diabetes(as_frame=True)
diabetes_df = diabetes.frame
diabetes_df.iloc[:5,:5]
```

| | age | sex | bmi | bp | s1 |
|---|---|---|---|---|---|
| 0 | 0.0380759 | 0.0506801 | 0.0616962 | 0.0218724 | -0.0442235 |
| 1 | -0.00188202 | -0.0446416 | -0.0514741 | -0.0263275 | -0.00844872 |
| 2 | 0.0852989 | 0.0506801 | 0.0444512 | -0.00567042 | -0.0455995 |

| | age | sex | bmi | bp | s1 |
|---|---|---|---|---|---|
| 3 | -0.0890629 | -0.0446416 | -0.011595 | -0.0366561 | 0.0121906 |
| 4 | 0.00538306 | -0.0446416 | -0.0363847 | 0.0218724 | 0.00393485 |

The diabetes dataset is intended for regression problems. It includes physiological data and disease progression after one year.

## Example: First Look at the Digits Dataset

```python
digits = datasets.load_digits(as_frame=True)
digits_df = digits.frame
digits_df.iloc[:5,:5]
```

| | pixel_0_0 | pixel_0_1 | pixel_0_2 | pixel_0_3 | pixel_0_4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 5 | 13 | 9 |
| 1 | 0 | 0 | 0 | 12 | 13 |
| 2 | 0 | 0 | 0 | 4 | 15 |
| 3 | 0 | 0 | 7 | 15 | 13 |
| 4 | 0 | 0 | 0 | 1 | 11 |

The digits dataset contains images (8x8 pixels) of handwritten digits (0 to 9) and is commonly used for image classification.

## Typical Machine Learning Workflow

The basic steps we will follow throughout the book are:

1. **Data Loading**
   From CSV files, databases, or built-in datasets.
2. **Data Preprocessing**
   Cleaning, transforming, and scaling data.
3. **Model Selection and Training**
   Choosing and fitting an algorithm (classifier, regressor, etc.).
4. **Model Evaluation**
   Measuring model performance.
5. **Hyperparameter Tuning**
   Adjusting parameters to improve results.
6. **Saving and Reusing Models**
   Exporting models for future use.

## scikit-learn and the Data Science Ecosystem

scikit-learn usually works together with:

- `numpy`: numerical operations and arrays.
- `pandas`: tabular data handling.
- `matplotlib` and `seaborn`: data visualization.
- `joblib`: model export and persistence.

## First Complete Example: Classification with Iris Dataset

```python
# Load dataset
iris = datasets.load_iris()
X = iris.data
y = iris.target

# Import a simple classifier
from sklearn.neighbors import KNeighborsClassifier

# Create the model
model = KNeighborsClassifier(n_neighbors=3)

# Train the model
model.fit(X, y)

# Make a prediction
sample = [[5.1, 3.5, 1.4, 0.2]]
prediction = model.predict(sample)
print("Prediction:", iris.target_names[prediction][0])
```

```
Prediction: setosa
```

This simple example shows the basic flow:

1. Data loading
2. Model creation
3. Model training
4. Making predictions

In the following chapters, we will explore each of these steps in more detail.

# Supervised Learning

Supervised learning is the most common type of machine learning. It refers to the task of learning a function that maps input data (features) to known outputs (labels), based on a set of labeled training examples.

## What is Supervised Learning?

In supervised learning, the algorithm is provided with:

- **Inputs** (`X`): A set of features (variables, measurements).
- **Outputs** (`y`): A known label or target for each input.

The objective is to learn a function `f(X)` ≈ `y` that generalizes well to unseen data.

Supervised learning is used when we want to:

- Predict a continuous value = **Regression**.
- Predict a category = **Classification**.

## Workflow in Supervised Learning

The typical workflow is:

1. Data collection and preparation.
2. Data preprocessing.
3. Model selection.
4. Model training (fitting).
5. Model evaluation.
6. Prediction and deployment.

## Classification vs Regression

| Task | Input | Output | Example |
|---|---|---|---|
| **Classification** | Features | Discrete label (category) | Email is spam or not? |
| **Regression** | Features | Continuous value | Predict house prices? |

## Example: Classification with k-Nearest Neighbors (k-NN)

The **k-Nearest Neighbors (k-NN)** algorithm is a simple, yet powerful classification method. It predicts the class of a new sample by looking at the majority class among its *k* closest neighbors in the training data.

In the following example, we will load the classic Iris dataset, split it into training and testing sets, create a k-NN classifier with `k=3`, train it on the training data, and evaluate its performance on the test set by calculating its accuracy.

```python
# Load the iris dataset
from sklearn import datasets
iris = datasets.load_iris()
```

```python
X = iris.data
y = iris.target

# Train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
     test_size=0.2, random_state=21)

# Create and train k-NN model
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate accuracy
from sklearn.metrics import accuracy_score
print("Accuracy:", accuracy_score(y_test, y_pred))
print(f"Accuracy: {accuracy_score(y_test, y_pred):.3f}")
```

```
/home/imartinez/anaconda3/envs/DataScience/lib/python3.9/
    site-packages/scipy/__init__.py:155: UserWarning: A
    NumPy version >=1.18.5 and <1.26.0 is required for
    this version of SciPy (detected version 1.26.4
  warnings.warn(f"A NumPy version >={np_minversion} and
      <{np_maxversion}"


Accuracy: 0.967
```

## Visualizing k-NN Decision Boundaries (optional)

In the following example, we will use **Principal Component Analysis (PCA)** to reduce the dimensionality of the Iris dataset from 4 features to 2, allowing us to visualize the data in a simple 2D scatter plot. This technique is often used for exploratory data analysis, as it helps to reveal patterns, clusters, or structure in the data. After reducing the dimensions, we will plot each data point colored according to its class label, providing a visual representation of how the different species are distributed based on the two main principal components.

```python
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Reduce dimensions for visualization
pca = PCA(n_components=2)
X_reduced = pca.fit_transform(X)

# Plot decision regions
plt.figure(figsize=(8,6))
plt.scatter(X_reduced[:,0], X_reduced[:,1], c=y, cmap="
    viridis", edgecolor="k")
plt.title("Iris dataset (PCA projection)")
plt.xlabel("PCA 1")
plt.ylabel("PCA 2")
plt.show()
```

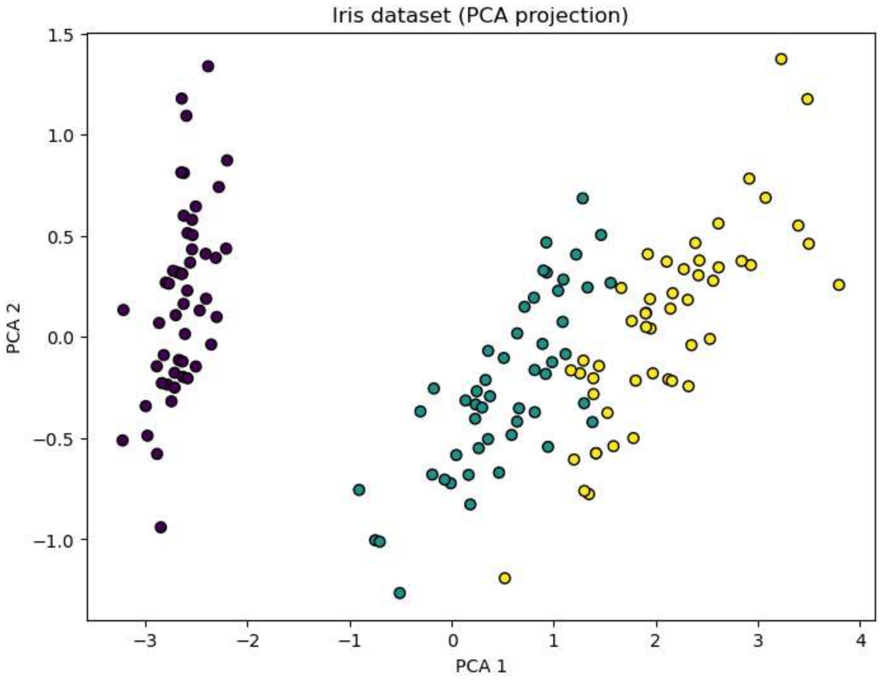

**Figure 1:** Image generated by the provided code.

## Example: Regression with Linear Regression

**Linear Regression** is the most fundamental regression algorithm. It assumes that the relationship between the input variables and the target is linear.

In this example, we will work with the **Diabetes** dataset, which is commonly used for regression tasks. We will split the data into training and testing sets, create a **Linear Regression** model, and train it to predict a continuous target: the disease progression after one year. After fitting the model, we will make predictions on the test set and evaluate its performance using two common regression metrics: the **R**$^2$

**score**, which indicates how well the model explains the variance in the data, and the **Mean Squared Error (MSE)**, which measures the average squared difference between predicted and actual values.

```python
# Load the diabetes dataset
diabetes = datasets.load_diabetes()
X = diabetes.data
y = diabetes.target

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
     test_size=0.2, random_state=42)

# Create and train the model
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate with R^2 score
from sklearn.metrics import r2_score, mean_squared_error
print("R2 Score:", r2_score(y_test, y_pred))
print("MSE:", mean_squared_error(y_test, y_pred))
```

```
R2 Score: 0.4526027629719195
MSE: 2900.193628493482
```

## Visualizing Regression Results (optional)

Visualizing the relationship between the predicted values and the true values is a simple but powerful way to assess the quality of a regression model. A good model should produce predictions that align closely with the true values, resulting in points near the diagonal line. Plotting these values allows us to quickly detect systematic errors, outliers, or if the model tends to underpredict or overpredict in certain regions. This type of plot is especially helpful during exploratory analysis and model evaluation.

In the following example, we will create a scatter plot comparing the predicted values from our linear regression model with the actual target values from the test set. Each point represents a single prediction. Ideally, all points should lie close to the dashed diagonal line, which indicates perfect predictions. This visual inspection complements the numerical evaluation metrics and helps us identify potential patterns, biases, or areas where the model is not performing well.

```python
plt.scatter(y_test, y_pred)
plt.xlabel("True Values")
plt.ylabel("Predicted Values")
plt.title("Linear Regression - True vs Predicted")
plt.plot([y_test.min(), y_test.max()], [y_test.min(),
    y_test.max()], "r--")
plt.show()
```

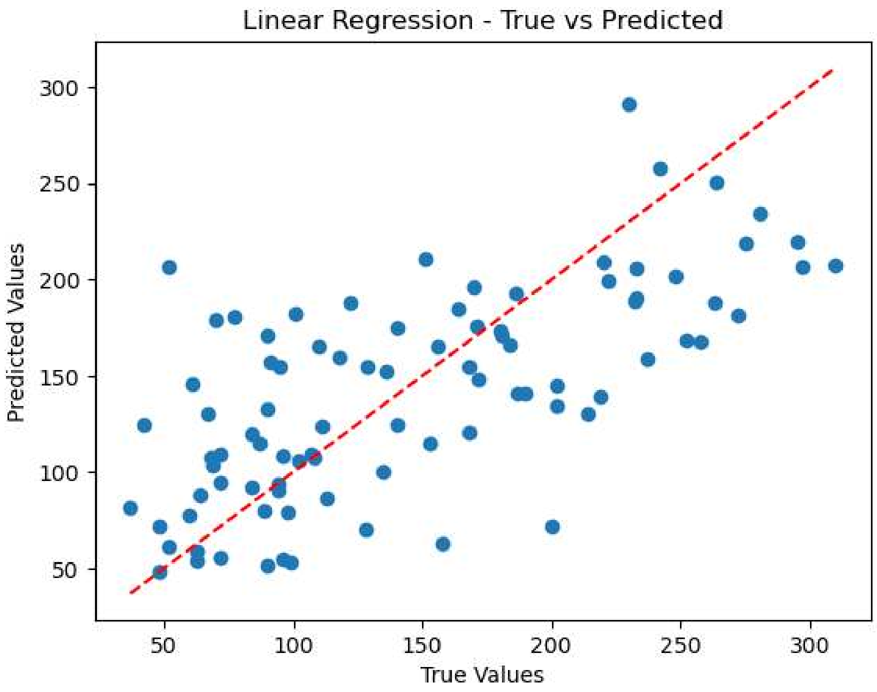

**Figure 2:** Image generated by the provided code.

## Choosing the Right Model

There is no one-size-fits-all. Some common models you will encounter:

## Classification Models

### Logistic Regression

Logistic Regression is one of the simplest and most interpretable classification models. It models the probability of belonging to a class using a logistic (sigmoid) function applied to a linear combination of the input features. **Geometric interpretation**: It creates a straight decision boundary (a line in 2D, a plane in 3D, or a hyperplane in higher dimensions) that separates the classes.

### Decision Trees

Decision Trees classify samples by following a sequence of questions about the features. Each internal node makes a decision based on a feature value, and leaves represent class labels. **Geometric interpretation**: It creates axis-aligned decision boundaries (parallel to feature axes) that split the feature space into rectangular regions.

### k-Nearest Neighbors (k-NN)

The k-NN classifier predicts the class of a new sample by looking at the most common class among its *k* nearest neighbors in the training data. It does not build an explicit model but relies on distance calculations. **Geometric interpretation**: The decision boundaries are formed implicitly by the spatial arrangement of the data points, often resulting in irregular and complex regions.

### Support Vector Machines (SVM)

SVMs try to find the hyperplane that maximizes the margin between classes. Only a few data points (the support vectors) are crucial to defining this boundary. SVMs can also use kernels to project data into higher dimensions and separate non-linearly separable classes. **Geometric interpretation**: The model defines a hyperplane (or curved boundary with kernels) that separates classes with the largest possible margin.

### Random Forest

Random Forest is an ensemble of multiple decision trees trained on different subsets of the data and features. The final prediction is usually made by majority voting among the trees. It reduces variance and improves generalization compared to a single decision tree. **Geometric interpretation**: Similar to decision trees, it creates complex axis-aligned decision regions, but the combination of many trees smooths the boundaries.

### Gradient Boosting

Gradient Boosting is an ensemble method that builds a sequence of weak learners (usually shallow decision trees) where each tree tries to correct the errors of the previous ones. It is powerful and often used in winning solutions in machine learning competitions. **Geometric interpretation**: Like Random Forest, it creates complex decision boundaries by combining many simple trees, but with an iterative error-correcting approach.

## Regression Models

### Linear Regression

Linear Regression is the most basic regression technique. It models the relationship between the input features and the target variable by fitting a straight line (or a hyperplane in higher dimensions) that minimizes the squared differences between predicted and actual values. **Geometric interpretation**: It fits a straight line in 2D, a plane in 3D, or a hyperplane in higher dimensions.

### Ridge and Lasso Regression

Both Ridge and Lasso are extensions of Linear Regression that introduce regularization to prevent overfitting:

- **Ridge Regression** adds an L2 penalty (squared magnitude of coefficients).
- **Lasso Regression** adds an L1 penalty (absolute value of coefficients) and can shrink some coefficients exactly to zero, performing variable selection.

**Geometric interpretation**: They still fit linear models (lines, planes, hyperplanes) but the regularization influences the shape and magnitude of the coefficients, effectively adjusting the orientation and steepness of the regression plane.

### Decision Trees for Regression

Decision Trees for regression predict continuous values by splitting the feature space into regions and assigning the mean target value of the samples within each region. **Geometric interpretation**: The model partitions the input space into axis-aligned rectangles (in 2D) or hyperrectangles (in higher dimensions) and assigns a constant prediction within each.

### Random Forest Regressor

Random Forest Regressor is an ensemble of decision trees trained on different subsets of data and features. Predictions are averaged across all trees to produce a more stable and robust estimate. **Geometric interpretation**: It creates a combination of piecewise constant prediction surfaces resulting from averaging the outputs of multiple decision trees, which reduces variance compared to a single tree.

### Gradient Boosting Regressor

Gradient Boosting Regressor builds a sequence of shallow decision trees, where each new tree tries to predict the residual errors made by the ensemble so far. The final prediction is the sum of the predictions of all trees. **Geometric interpretation**: Like Random Forest, it produces a complex, piecewise constant approximation of the target function, but it builds it sequentially by focusing on improving errors iteratively.

## Common Evaluation Metrics

Evaluating machine learning models is a crucial step to determine how well they perform on unseen data. The goal is not only to assess the quality of a single model but also to be able to compare different models objectively and select the most suitable one for the problem at hand.

Choosing the right evaluation metric is essential, as it should reflect the nature of the task and the aspects that matter most in your specific application. For example, in classification tasks, accuracy may not always be enough if the classes are imbalanced, and metrics like precision or recall could be more informative. In regression problems, it is common to compare models using error-based metrics such as **MSE**, **MAE**, or **R**$^2$.

Throughout this book, we will make use of different metrics depending on the context, and we will see how they help us:

- Evaluate a model's ability to make accurate predictions.
- Detect problems like bias, variance, or imbalance.
- Compare different models quantitatively and fairly.

In the following table, you will find some of the most common evaluation metrics, grouped by task.

| Task | Metric | Description | scikit-learn Function |
|---|---|---|---|
| Classification | Accuracy | Percentage of correct predictions | `sklearn.metrics.accuracy_score()` |
| Classification | Precision | How many predicted positives are real? | `sklearn.metrics.precision_score()` |

| Task | Metric | Description | scikit-learn Function |
|---|---|---|---|
| Classification | Recall | How many real positives are detected? | `sklearn.metrics.recall_score()` |
| Classification | F1-score | Harmonic mean of precision and recall | `sklearn.metrics.f1_score()` |
| Regression | MSE | Mean Squared Error | `sklearn.metrics.mean_squared_error()` |
| Regression | RMSE | Root Mean Squared Error (square root of MSE) | `np.sqrt(mean_squared_error())` |
| Regression | MAE | Mean Absolute Error | `sklearn.metrics.mean_absolute_error()` |
| Regression | $R^2$ | Coefficient of determination | `sklearn.metrics.r2_score()` |

We will go deeper into these metrics in the **Model Evaluation** chapter.

---

In the next chapters, we will explore these models, metrics, and techniques in detail, always with practical examples.

# Unsupervised Learning

Unsupervised learning is a branch of machine learning where the model tries to find hidden patterns or structure in the data without having access to labeled outputs. Unlike supervised learning, there are no explicit labels or targets; instead, the algorithm must identify relationships within the data itself.

## What is Unsupervised Learning?

The goal of unsupervised learning is to extract useful information about the structure of the data. Typical tasks include:

- **Clustering**: Grouping similar samples together.
- **Dimensionality Reduction**: Reducing the number of variables while retaining the most important information.

### When do we use Unsupervised Learning?

- When labels are not available.
- When we want to explore or visualize the data structure.
- When we need to preprocess or transform data for further analysis.

## Workflow in Unsupervised Learning

1. Data preparation and preprocessing.
2. Model selection.
3. Model training.
4. Evaluation and interpretation (usually qualitative).
5. Application (clustering, visualization, anomaly detection, etc.).

## Main Unsupervised Learning Techniques

- **Clustering**
  - k-Means
  - DBSCAN
  - Agglomerative Clustering
- **Dimensionality Reduction**
  - Principal Component Analysis (PCA)
  - t-SNE (will be briefly mentioned)

## Example: Clustering with k-Means

The **k-Means** algorithm tries to partition the dataset into *k* clusters, grouping samples that are close together in feature space. The algorithm iteratively assigns samples to the nearest centroid and then recalculates the centroids.

In the following example, we will use the iris dataset (without its labels) to create 3 clusters and compare them to the actual classes for illustration purposes.

```python
from sklearn import datasets
from sklearn.cluster import KMeans
```

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

# Load the dataset
iris = datasets.load_iris()
X = iris.data

# Apply k-Means clustering
kmeans = KMeans(n_clusters=3, random_state=42, n_init=3)
y_kmeans = kmeans.fit_predict(X)

# Scatter plot of the clusters
plt.figure(figsize=(8,6))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=y_kmeans, palette
    ="viridis", edgecolor="k")
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title("k-Means clustering on Iris dataset")
plt.legend(title="Cluster")
plt.show()
```

In this example, we cluster the iris samples into 3 groups using k-Means. We then plot the first two features (sepal length and sepal width) to visualize the clusters. Although we do not use the true labels, the clusters often resemble the actual classes to some extent.

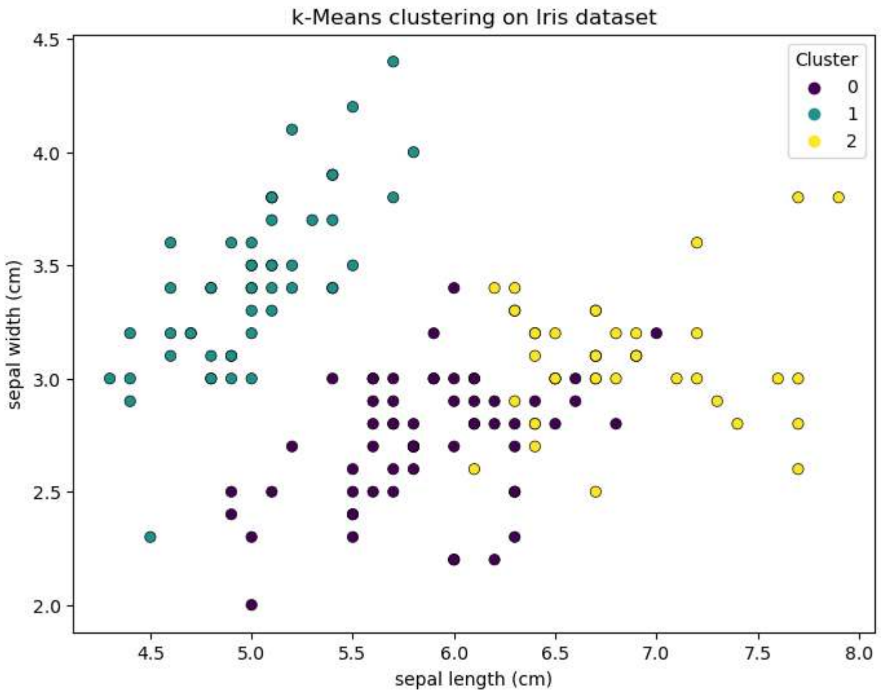

**Figure 1:** Image generated by the provided code.

## Example: Density-Based Clustering with DBSCAN

**DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) groups samples based on the density of points in the feature space. It is especially useful when the data has irregular shapes or contains noise.

In the next example, we will apply DBSCAN to the iris dataset.

```python
from sklearn.cluster import DBSCAN

```

```
# Apply DBSCAN
dbscan = DBSCAN(eps=0.5, min_samples=5)
y_dbscan = dbscan.fit_predict(X)

# Plot DBSCAN clustering
plt.figure(figsize=(8,6))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=y_dbscan, palette
    ="Set1", edgecolor="k")
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title("DBSCAN clustering on Iris dataset")
plt.legend(title="Cluster")
plt.show()
```

In this example, DBSCAN detects clusters based on point density. Notice that DBSCAN may assign some points to the *noise* cluster (usually labeled as $-1$), indicating that they don't belong to any dense region.

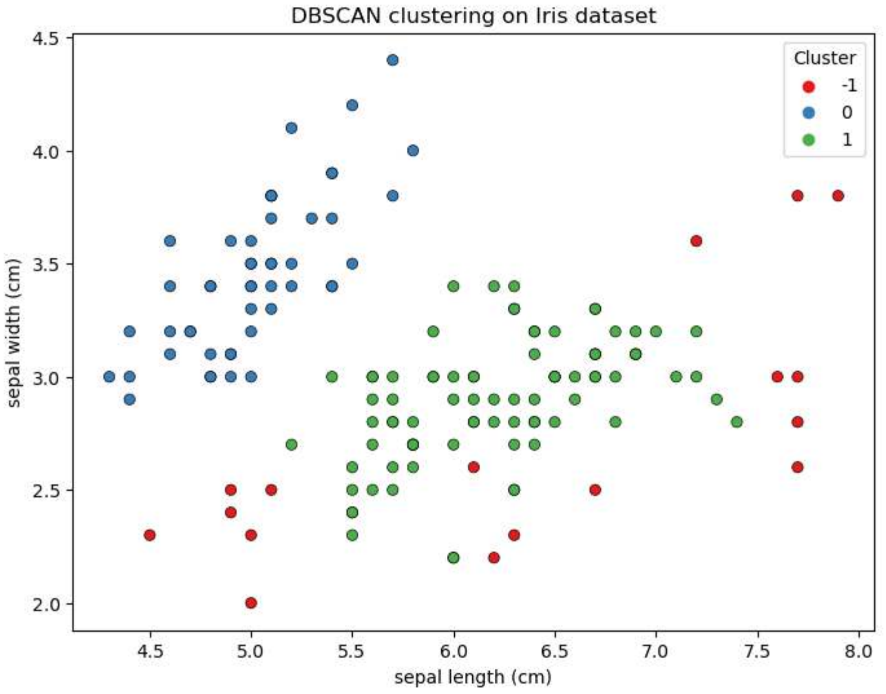

**Figure 2:** Image generated by the provided code.

## Example: Agglomerative Clustering

**Agglomerative Clustering** is a hierarchical clustering technique. It starts by treating each sample as its own cluster and iteratively merges the closest clusters until a predefined number of clusters is obtained.

```python
from sklearn.cluster import AgglomerativeClustering

# Apply Agglomerative Clustering
agg = AgglomerativeClustering(n_clusters=3)
```

```python
y_agg = agg.fit_predict(X)

# Plot Agglomerative clustering
plt.figure(figsize=(8,6))
sns.scatterplot(x=X[:,0], y=X[:,1], hue=y_agg, palette="
    Set2", edgecolor="k")
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title("Agglomerative clustering on Iris dataset")
plt.legend(title="Cluster")
plt.show()
```

Agglomerative Clustering produces a hierarchy of clusters and, in this example, results in 3 groups. Its flexibility allows for different linkage criteria, which we will explore later.

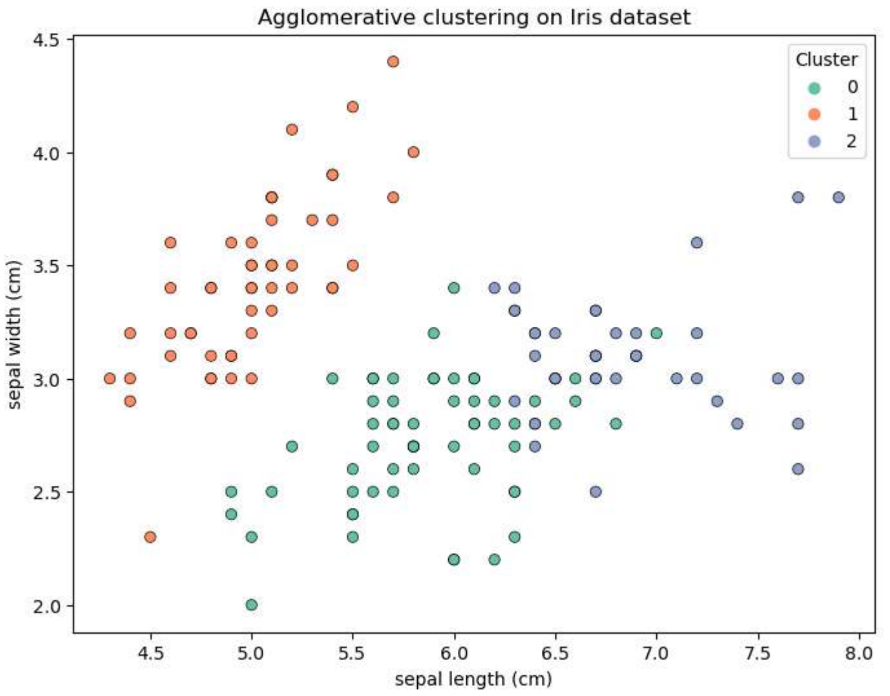

**Figure 3:** Image generated by the provided code.

## Example: Dimensionality Reduction with PCA

**Principal Component Analysis (PCA)** reduces the number of features by projecting the data onto a smaller set of dimensions that capture most of its variance. It is widely used for visualization.

In the following example, we reduce the iris dataset from 4 dimensions to 2 to visualize it.

```python
from sklearn.decomposition import PCA
```

```python

# Apply PCA
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Scatter plot
plt.figure(figsize=(8,6))
sns.scatterplot(x=X_pca[:,0], y=X_pca[:,1], hue=iris.
    target, palette="viridis", edgecolor="k")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("PCA projection of the Iris dataset")
plt.legend(title="Class")
plt.show()
```

In this example, PCA finds the two directions that best explain the variance in the data and projects the samples into this 2D space. You can clearly see how classes tend to form separate groups, even though PCA is unsupervised and does not use the labels during the transformation.

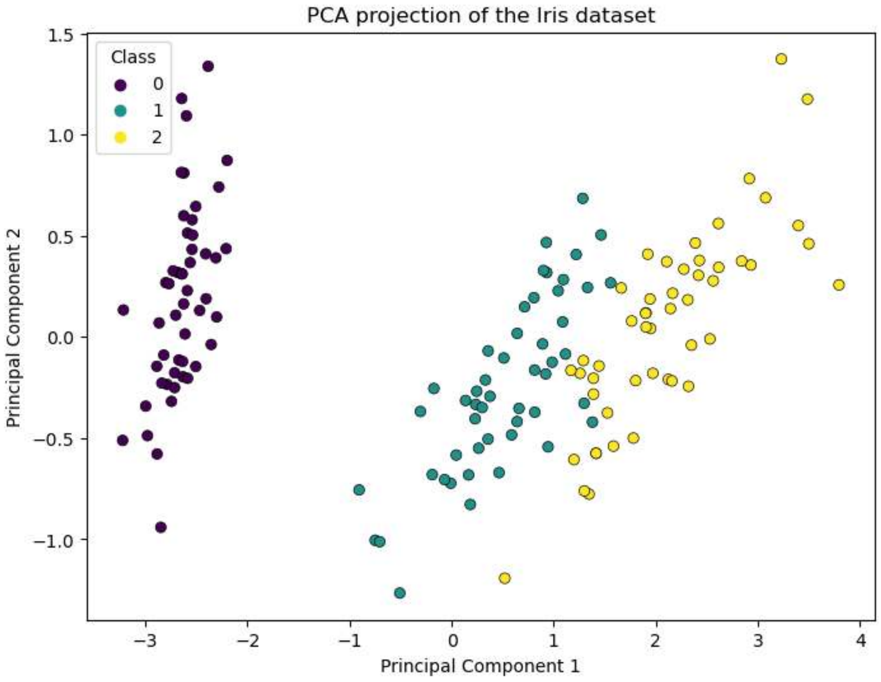

**Figure 4:** Image generated by the provided code.

## Remarks

Unsupervised learning models are powerful tools to:

- Discover structure and relationships in data.
- Perform data compression.
- Preprocess data for further analysis (e.g., feeding into supervised models).

In the next chapters, we will deepen into model evaluation, pipelines, and more advanced methods.

# Model Evaluation and Selection

Once a model is trained, it is essential to evaluate its performance and compare it to alternative models. Evaluation allows us to assess whether the model is learning correctly, how well it generalizes to unseen data, and whether its predictions are reliable.

Model selection, on the other hand, refers to the process of systematically comparing multiple models or configurations to choose the most suitable one for a given task. In this chapter, we will introduce key evaluation metrics, cross-validation techniques, and pipelines to streamline and standardize workflows.

## Evaluation Metrics

Selecting an appropriate metric is as important as choosing the model itself. Depending on the problem (classification or regression), different metrics highlight different aspects of the model's performance.

### Classification Metrics

In classification problems, predictions are discrete class labels. Some of the most common metrics include:

| Metric | Description |
|---|---|
| Accuracy | Proportion of correct predictions. Suitable for balanced datasets. |
| Precision | How many predicted positives are actually positive. Important when false positives are costly. |
| Recall | How many real positives are correctly detected. Important when false negatives are costly. |
| F1-score | Harmonic mean of precision and recall. Useful when dealing with imbalanced datasets. |

**Example: Compute classification metrics**

In the following example, we will compute common classification metrics using a k-NN classifier on the Iris dataset.

```python
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Load data and split
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data,
                                                    iris.target,
                                                    test_size=0.2,
                                                    random_state
                                                       =21,
                                                    stratify
                                                       =
                                                       iris
                                                       .
                                                       target
                                                       )

# Train k-NN
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Evaluate
print(f"Accuracy: {accuracy_score(y_test, y_pred):.3f}")
print(f"Precision (macro): {precision_score(y_test,
    y_pred, average='macro'):.3f}")
print(f"Recall (macro): {recall_score(y_test, y_pred,
    average='macro'):.3f}")
print(f"F1-score (macro): {f1_score(y_test, y_pred,
    average='macro'):.3f}")
```

```
Accuracy: 0.967
Precision (macro): 0.970
Recall (macro): 0.967
F1-score (macro): 0.967
```

This example shows how to obtain basic classification metrics using scikit-learn. We use `average='macro'` since the Iris dataset has more than two classes.

## Regression Metrics

In regression problems, the model predicts continuous numerical values instead of discrete class labels. To evaluate regression models, we measure the difference between predicted and actual values. Several metrics are commonly used depending on the application and the type of errors we want to penalize.

| Metric | Description |
|---|---|
| MSE | Mean Squared Error, penalizes large errors more strongly. |
| RMSE | Root Mean Squared Error, the square root of MSE, expressed in the same units as the target variable. |
| MAE | Mean Absolute Error, the average of the absolute errors. Less sensitive to outliers than MSE. |
| $R^2$ | Coefficient of determination, measures the proportion of variance explained by the model. Values closer to 1 indicate better models. |

**Example: Regression Metrics with the California Housing Dataset**

To illustrate the use of these metrics, we will use the **California Housing** dataset, which contains information about districts in California and their median house prices. This dataset is a common benchmark for regression problems.

In the following example, we will: 1. Load the dataset. 2. Split it into training and testing sets. 3. Train a simple linear regression model. 4. Evaluate it using the most common regression metrics.

```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error,
    mean_absolute_error, r2_score

# Load the California Housing dataset
housing = fetch_california_housing()
X = housing.data
y = housing.target
```

```python

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
     test_size=0.2, random_state=42)

# Train a linear regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
print(f"MSE: {mean_squared_error(y_test, y_pred):.3f}")
print(f"RMSE: {mean_squared_error(y_test, y_pred, squared
    =False):.3f}")
print(f"MAE: {mean_absolute_error(y_test, y_pred):.3f}")
print(f"R2: {r2_score(y_test, y_pred):.3f}")
```

```
MSE: 0.556
RMSE: 0.746
MAE: 0.533
R2: 0.576
```

In this example, the model tries to predict the median house value (in units of 100,000 dollars) based on socio-economic and geographical features. We report several metrics to assess the model's performance.

### Visualizing Regression Results

It is often useful to visualize the relationship between the predicted values and the true target values. Ideally, the points should lie close to the diagonal line, which indicates perfect predictions.

```python
import matplotlib.pyplot as plt

plt.scatter(y_test, y_pred, alpha=0.5)
plt.xlabel("True Median House Value")
plt.ylabel("Predicted Median House Value")
plt.title("Linear Regression - True vs Predicted")
plt.plot([y_test.min(), y_test.max()], [y_test.min(),
    y_test.max()], "r--")
plt.show()
```

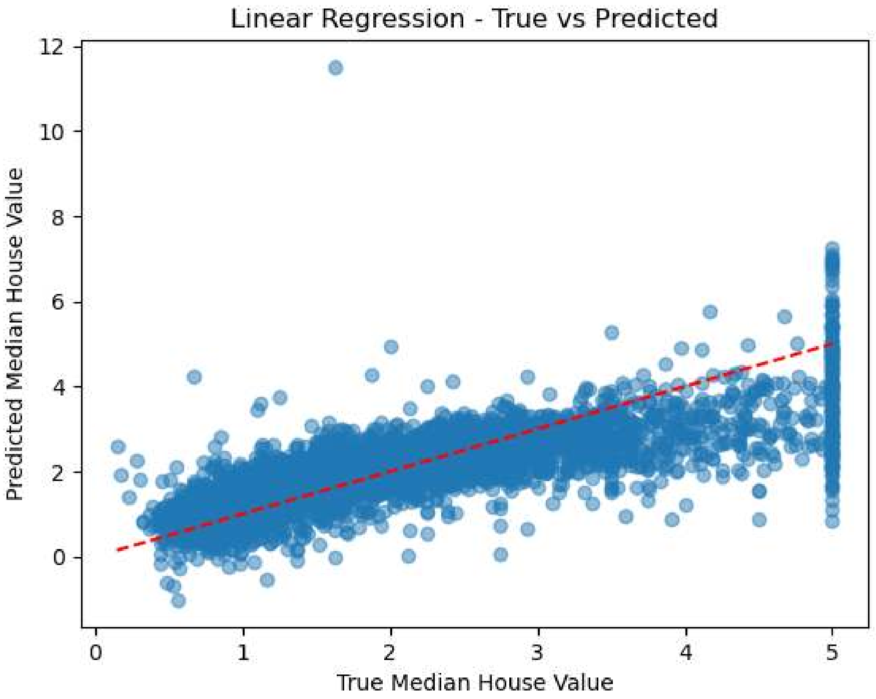

**Figure 1:** Image generated by the provided code.

This plot helps to visually detect biases, patterns, or systematic errors that may not be obvious from the numeric metrics alone.

## Cross-Validation

Cross-validation is a crucial technique to evaluate the generalization capability of a model. Instead of relying on a single train/test split, cross-validation partitions the training data into *k* folds and performs multiple train/validation cycles.

The most popular approach is **k-Fold Cross-Validation**.

### Example: k-Fold Cross-Validation on the Housing Dataset

```python
from sklearn.model_selection import cross_val_score

# Cross-validation with R2 as the scoring metric
scores = cross_val_score(model, X_train, y_train, cv=5,
    scoring='r2')

print("Cross-validated R2 scores:", scores)
print("Mean R2:", scores.mean())
```

```
Cross-validated R2 scores: [0.62011512 0.61298876
    0.6134416  0.61069973 0.60017477]
Mean R2: 0.6114839952560993
```

This example evaluates the linear regression model using 5-fold cross-validation and computes the average $R^2$ across the folds.

## Model Selection with GridSearchCV and RandomizedSearchCV

Hyperparameter tuning helps improve model performance by searching for the best combination of parameters.

- **GridSearchCV** tests all combinations in a predefined grid.

- **RandomizedSearchCV** samples combinations randomly, which is useful for large search spaces.

## Example: Grid Search for k-NN Regression

We now illustrate model selection with **k-Nearest Neighbors Regression**, which has a hyperparameter `n_neighbors`.

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import GridSearchCV

# Define hyperparameter grid
param_grid = {'n_neighbors': [3, 5, 7, 9, 11, 13, 15]}

# Grid search with 5-fold CV
grid = GridSearchCV(KNeighborsRegressor(), param_grid, cv
    =5, scoring='r2')
grid.fit(X_train, y_train)

print("Best parameters:", grid.best_params_)
print("Best cross-validated R2:", grid.best_score_)
```

```
Best parameters: {'n_neighbors': 9}
Best cross-validated R2: 0.142318599273422
```

```
pd.DataFrame(grid.cv_results_)[["param_n_neighbors", "
    mean_test_score", "std_test_score"]]
```

param_n_neighbors

mean_test_score

std_test_score

0

3

0.095089

0.010594

1

5

0.130695

0.010877

2

7

0.139132

0.009678

3

9

0.142319

0.005315

4

11

0.137519

0.005391

5

13

0.133094

0.005110

6

15

0.126079

0.004753

This example searches for the optimal number of neighbors for the `KNeighborsRegressor` using cross-validation.

## Pipelines

When building real-world models, it is common to apply preprocessing steps before fitting a model. **Pipelines** allow us to chain preprocessing and modeling steps into a single object, making workflows cleaner and less error-prone.

### Example: Pipeline with StandardScaler + Linear Regression

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

# Create pipeline: scaling + regression
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('regressor', LinearRegression())
])

# Train and evaluate
pipe.fit(X_train, y_train)
print("Pipeline R2 on test set:", pipe.score(X_test,
    y_test))
```

```
Pipeline R2 on test set: 0.575787706032451
```

In this example, the pipeline first scales the features using `StandardScaler` and then fits a linear regression model, all in one step.

## Example: ColumnTransformer + Pipeline (Regression)

If your data contains both numerical and categorical variables, you can use `ColumnTransformer` to apply different transformations to each subset of columns.

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

# Example dataset
import pandas as pd
data = pd.DataFrame({
    'feature_num': [1.0, 2.0, 3.0, 4.0],
    'feature_cat': ['A', 'B', 'A', 'B'],
    'target': [100, 200, 150, 250]
})

X = data[['feature_num', 'feature_cat']]
y = data['target']

# Preprocessing: scale numeric, encode categorical
preprocessor = ColumnTransformer([
    ('num', StandardScaler(), ['feature_num']),
    ('cat', OneHotEncoder(), ['feature_cat'])
])

# Full pipeline
pipe = Pipeline([
    ('prep', preprocessor),
    ('regressor', LinearRegression())
])

pipe.fit(X, y)
print("Pipeline fitted successfully.")
```

```
Pipeline fitted successfully.
```

This example shows how to preprocess numerical and categorical features automatically and integrate them into a regression pipeline.

In the next chapter, we will focus on **Preprocessing Techniques**, covering common transformations, feature engineering, and handling of missing data.

# Preprocessing

Before training a machine learning model, it is often necessary to **transform the raw data** into a form that is more suitable for learning. Preprocessing ensures that: - Features are in the right scale and format. - Missing values are handled properly. - Categorical variables are encoded numerically.

This chapter introduces common preprocessing techniques using scikit-learn transformers. Each section shows how data looks **before and after** each transformation to illustrate its effect.

## Feature Scaling

Feature scaling is used to **normalize the range of independent variables**, especially for algorithms that are sensitive to the scale of the data (e.g., k-NN, SVM, Gradient Boosting).

### StandardScaler

The `StandardScaler` standardizes features by removing the mean and scaling to unit variance (z-score normalization).

In the following example, we create a small DataFrame with numeric features, and observe the effect of scaling.

```python
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Example DataFrame
df = pd.DataFrame({
    'feature1': [10, 20, 30, 40, 50],
    'feature2': [1, 2, 3, 4, 5]
})

scaler = StandardScaler()
scaled = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled, columns=df.columns)

# Display original and scaled side by side
pd.concat([df, scaled_df.add_suffix('_scaled')], axis=1)
```

feature1

feature2

feature1_scaled

feature2_scaled

0

10

1

-1.414214

-1.414214

1

20

2

-0.707107

-0.707107

2

30

3

0.000000

0.000000

3

40

4

0.707107

0.707107

4

50

5

1.414214

1.414214

This table shows how `StandardScaler` transforms each value by subtracting the mean and dividing by the standard deviation. The resulting features have mean 0 and standard deviation 1.

## MinMaxScaler

`MinMaxScaler` rescales features to a fixed range, typically [0, 1]. This is useful when we want all features to be strictly positive or within a specific range.

```python
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaled = scaler.fit_transform(df)
scaled_df = pd.DataFrame(scaled, columns=df.columns)

pd.concat([df, scaled_df.add_suffix('_scaled')], axis=1)
```

feature1

feature2

feature1_scaled

feature2_scaled

0

10

1

0.00

0.00

1

20

2

0.25

0.25

2

30

3

0.50

0.50

3

40

4

0.75

0.75

4

50

5

1.00

1.00

As shown, each feature is scaled so that the minimum value becomes 0 and the maximum becomes 1.

## Visualizing the Effect of Scaling

To better understand how feature scaling affects the data, the following boxplot compares the distribution of two numeric features under three different conditions:

- **Original**: raw, unscaled data
- **StandardScaler**: scaled to zero mean and unit variance
- **MinMaxScaler**: rescaled to the [0, 1] range

Each box represents the distribution of one feature under a specific transformation. This visualization highlights how:

- `StandardScaler` centers the data around 0 and normalizes the spread, which is particularly useful for algorithms sensitive to scale.
- `MinMaxScaler` compresses all values into the [0, 1] interval, preserving the relative differences between values but adjusting their absolute range.

Using boxplots makes it easy to observe the shift in central tendency, the change in variance, and how scaling affects the presence or absence of outliers. This kind of visual check is very helpful to confirm that the transformations are behaving as expected before applying them to real models.

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler,
    MinMaxScaler

# Original Data
df = pd.DataFrame({
    'feature1': [0, 5, 10, 15, 20],
    'feature2': [1, 2, 3, 4, 5]
})

df_standard = pd.DataFrame(StandardScaler().fit_transform
    (df), columns=df.columns)
df_minmax = pd.DataFrame(MinMaxScaler().fit_transform(df)
    , columns=df.columns)

df_original = df.copy()
df_original['type'] = 'Original'

df_standard['type'] = 'StandardScaler'
df_minmax['type'] = 'MinMaxScaler'

df_all = pd.concat([df_original, df_standard, df_minmax],
     axis=0)

df_melted = df_all.melt(id_vars='type', var_name='feature
    ', value_name='value')

```

```python
# Plot
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_melted, x='type', y='value', hue='
    feature')
plt.title("Effect of Scaling Methods on Feature
    Distributions")
plt.legend(title='Scaling Method')
plt.show()
```

```python
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler,
    MinMaxScaler

# Datos originales
df = pd.DataFrame({
    'feature1': [0, 5, 10, 15, 20],
    'feature2': [1, 2, 3, 4, 5]
})

# Escalado
df_standard = pd.DataFrame(StandardScaler().fit_transform
    (df), columns=df.columns)
df_minmax = pd.DataFrame(MinMaxScaler().fit_transform(df)
    , columns=df.columns)

# Añadimos una columna para el tipo de escalado
df_original = df.copy()
df_original['type'] = 'Original'

df_standard['type'] = 'StandardScaler'
df_minmax['type'] = 'MinMaxScaler'

# Unimos todo en un solo DataFrame
df_all = pd.concat([df_original, df_standard, df_minmax],
     axis=0)

# Reorganizamos para Seaborn
df_melted = df_all.melt(id_vars='type', var_name='feature
    ', value_name='value')
```

```

# Plot
plt.figure(figsize=(10, 6))
sns.boxplot(data=df_melted, x='type', y='value', hue='
    feature')
plt.title("Effect of Scaling Methods on Feature
    Distributions")
plt.legend(title='Scaling Method')
plt.show()
```

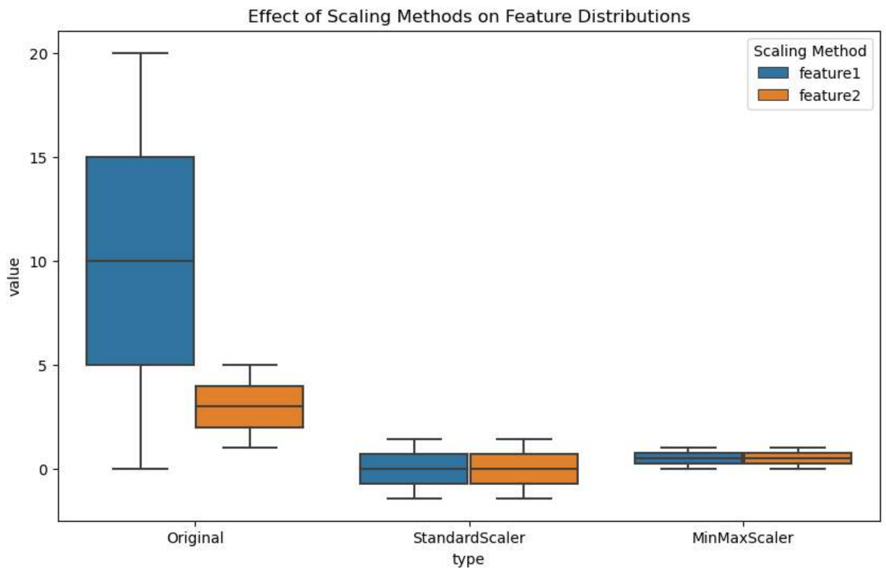

**Figure 1:** Image generated by the provided code.

## Encoding Categorical Variables

Many machine learning algorithms require numerical inputs. Encoding converts **categorical variables** into numeric format.

## OneHotEncoder

`OneHotEncoder` creates a binary column for each category. This is a safe and commonly used method for unordered categories.

```python
from sklearn.preprocessing import OneHotEncoder

# Categorical data
df_cat = pd.DataFrame({'color': ['red', 'green', 'blue',
    'green', 'red']})

encoder = OneHotEncoder(sparse_output=False)
encoded = encoder.fit_transform(df_cat)
encoded_df = pd.DataFrame(encoded, columns=encoder.
    get_feature_names_out())

# Display original and encoded
pd.concat([df_cat, encoded_df], axis=1)
```

color

color_blue

color_green

color_red

0

red

0.0

0.0

1.0

1

green

0.0

1.0

0.0

2

blue

1.0

0.0

0.0

3

green

0.0

1.0

0.0

4

red

0.0

0.0

1.0

This encoding expands the single 'color' column into three binary columns, one for each category.

## OrdinalEncoder

`OrdinalEncoder` assigns an integer to each category. It is useful when the categories have a meaningful order (e.g., low, medium, high), but should be avoided when the order is arbitrary.

```python
from sklearn.preprocessing import OrdinalEncoder

df_ordinal = pd.DataFrame({'size': ['small', 'medium', '
    large', 'medium', 'small']})

encoder = OrdinalEncoder()
encoded = encoder.fit_transform(df_ordinal)
encoded_df = pd.DataFrame(encoded, columns=['size_encoded
    '])

# Show transformation
pd.concat([df_ordinal, encoded_df], axis=1)
```

```python
from sklearn.preprocessing import OrdinalEncoder

df_ordinal = pd.DataFrame({'size': ['small', 'medium', '
    large', 'medium', 'small']})

encoder = OrdinalEncoder()
encoded = encoder.fit_transform(df_ordinal)
encoded_df = pd.DataFrame(encoded, columns=['size_encoded
    '])

# Show transformation
pd.concat([df_ordinal, encoded_df], axis=1)
```

size

size_encoded

0

small

2.0

1

medium

1.0

2

large

0.0

3

medium

1.0

4

small

2.0

In this example, ‘small’, ‘medium’, and ‘large’ are assigned integer values, typically based on their appearance order unless specified.

## Handling Missing Values

Missing values are common in real-world data. scikit-learn provides strategies to **impute** missing values automatically.

### SimpleImputer

The `SimpleImputer` replaces missing values using a chosen strategy, such as the mean, median, or most frequent value.

```
import numpy as npy
from sklearn.impute import SimpleImputer

df_missing = pd.DataFrame({
    'feature1': [1, 2, npy.nan, 4, 5],
    'feature2': [npy.nan, 1, 1, 2, 2]
```

```python
})

imputer = SimpleImputer(strategy='mean')
imputed = imputer.fit_transform(df_missing)
imputed_df = pd.DataFrame(imputed, columns=df_missing.
    columns)

# Compare original and imputed
pd.concat([df_missing, imputed_df.add_suffix('_imputed')
    ], axis=1)
```

feature1

feature2

feature1_imputed

feature2_imputed

0

1.0

NaN

1.0

1.5

1

2.0

1.0

2.0

1.0

2

NaN

1.0

3.0

1.0

3

4.0

2.0

4.0

2.0

4

5.0

2.0

5.0

2.0

This example fills missing values with the **mean** of each column. You can change the strategy to `"median"` or `"most_frequent"` if needed.

## KNNImputer

`KNNImputer` estimates missing values using the average of the nearest neighbors. It can capture more complex relationships than `SimpleImputer`.

```python
from sklearn.impute import KNNImputer

imputer = KNNImputer(n_neighbors=2)
imputed = imputer.fit_transform(df_missing)
imputed_df = pd.DataFrame(imputed, columns=df_missing.
    columns)

pd.concat([df_missing, imputed_df.add_suffix('_knn')],
    axis=1)
```

feature1

feature2

feature1_knn

feature2_knn

0

1.0

NaN

1.0

1.5

1

2.0

1.0

2.0

1.0

2

NaN

1.0

3.0

1.0

3

4.0

2.0

4.0

2.0

4

5.0

2.0

5.0

2.0

This method replaces missing values using the values of the nearest rows, according to the other available features. It is more flexible but computationally more expensive.

## Summary

| Transformation Type | Tool | Description |
|---|---|---|
| Scaling | `StandardScaler` | Normalize data to zero mean and unit variance |
| Scaling | `MinMaxScaler` | Scale data to a fixed range (usually [0, 1]) |
| Encoding | `OneHotEncoder` | Convert categories to binary indicators |
| Encoding | `OrdinalEncoder` | Convert categories to integers (ordered) |
| Imputation | `SimpleImputer` | Fill missing values with mean, median, or mode |
| Imputation | `KNNImputer` | Use nearest neighbors to fill missing values |

In the next chapters, we will combine these transformations using **Pipelines** and apply them to real datasets.

# Model Deployment Basics

Once a machine learning model has been trained and evaluated, the next step is to **deploy** it for real-world use. This means saving the model in a way that allows us to load it later and use it to make predictions — without having to retrain it every time.

In this chapter, we will focus on:

- Exporting trained models using `joblib` and `pickle`
- Loading and using those models for inference
- Good practices for simple model deployment scenarios

## Saving and Loading Models with `joblib`

The `joblib` library is commonly used for serializing scikit-learn models. It is efficient and designed to handle large numpy arrays, making it ideal for saving trained models.

### Example: Exporting a model with `joblib`

```python
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import joblib

```

```python
# Load data and train a simple model
X, y = load_iris(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
    random_state=42)

model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)

# Save the model to disk
joblib.dump(model, "knn_model.joblib")
```

This code trains a simple k-NN classifier and saves it to a file called `knn_model.joblib`. The model can now be stored and reused without retraining.

### Example: Loading and using a saved model

```python
# Load the model from disk
loaded_model = joblib.load("knn_model.joblib")

# Make a prediction
sample = [[5.1, 3.5, 1.4, 0.2]]
prediction = loaded_model.predict(sample)
```

After loading the model, we can directly use it to make predictions, exactly as if it had just been trained. This is the basic workflow behind many production systems.

## Alternative: Using `pickle` to Save Models

Although `joblib` is preferred for scikit-learn models, the `pickle` module can also serialize any Python object. It is more general-purpose and widely supported.

### Example: Saving and loading with `pickle`

```python
import pickle

# Save the model using pickle
with open("knn_model.pkl", "wb") as f:
    pickle.dump(model, f)

# Load the model back
with open("knn_model.pkl", "rb") as f:
    loaded_model = pickle.load(f)

# Use the model
prediction = loaded_model.predict(sample)
```

The `pickle` approach works similarly to `joblib`, but may be slightly slower or less efficient when dealing with large numpy arrays.

## Good Practices

Here are some good practices to follow when exporting and using models in production environments:

- **Always store the model together with the pre-processing steps** (e.g. scaling, encoding) using a pipeline.
- **Test the loaded model** before using it to ensure compatibility and avoid silent failures.
- **Document the model version and training metadata**: which data, algorithm, parameters, and evaluation scores were used.
- **Keep track of dependencies and versions** (e.g. scikit-learn version) to ensure reproducibility.

## Summary

| Task | Tool | Description |
|---|---|---|
| Save model | `joblib.dump()` | Fast and efficient for numpy-based models |
| Load model | `joblib.load()` | Restore model for use in predictions |
| Save with pickle | `pickle.dump()` | General-purpose object serialization |
| Load with pickle | `pickle.load()` | Restore any Python object |
| Use model | `.predict()` | Make predictions with trained model |

In future chapters, we will explore how to integrate saved models into full applications, web APIs, or batch prediction workflows.

# Advanced Topics

This chapter introduces several advanced but highly practical techniques that can improve the performance, robustness, and automation of machine learning workflows using scikit-learn and compatible libraries.

We will cover:

- Feature selection techniques
- Custom pipelines with transformers
- Workflow automation
- Integrations with external libraries like `scikit-optimize`, `imblearn`, and `mlxtend`

## Feature Selection

Feature selection helps reduce overfitting, improves model interpretability, and can reduce training time by selecting the most relevant input variables.

### Example: Feature selection using `SelectKBest`

```
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectKBest,
    f_classif
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

```python
from sklearn.pipeline import Pipeline

# Load dataset
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
     random_state=42)

# Create a pipeline with feature selection + model
pipeline = Pipeline([
    ('select', SelectKBest(score_func=f_classif, k=10)),
    ('clf', LogisticRegression(max_iter=1000))
])

pipeline.fit(X_train, y_train)
print("Test accuracy:", pipeline.score(X_test, y_test))
```

```python
from sklearn.datasets import load_breast_cancer
from sklearn.feature_selection import SelectKBest,
    f_classif
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

# Load dataset
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y,
     random_state=42)

# Create a pipeline with feature selection + model
pipeline = Pipeline([
    ('select', SelectKBest(score_func=f_classif, k=10)),
    ('clf', LogisticRegression(max_iter=1000))
])

pipeline.fit(X_train, y_train)
print("Test accuracy:", pipeline.score(X_test, y_test))
```

```
Test accuracy: 0.986013986013986
```

In this example, we select the 10 best features using ANOVA F-statistics before fitting a logistic regression model. Feature selection is done inside the pipeline to avoid data leakage.

## Custom Pipelines

Sometimes you need to define your own preprocessing logic. scikit-learn allows you to create custom transformers by extending `BaseEstimator` and `TransformerMixin`.

### Example: Custom transformer that logs shape of data

```
from sklearn.base import BaseEstimator, TransformerMixin

class ShapeLogger(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        print(f"[fit] Shape: {X.shape}")
        return self
    def transform(self, X):
        print(f"[transform] Shape: {X.shape}")
        return X

# Use in a pipeline
pipeline = Pipeline([
    ('logger', ShapeLogger()),
    ('model', LogisticRegression(max_iter=5000))
])

pipeline.fit(X_train, y_train)
```

```
from sklearn.base import BaseEstimator, TransformerMixin

class ShapeLogger(BaseEstimator, TransformerMixin):
    def fit(self, X, y=None):
        print(f"[fit] Shape: {X.shape}")
        return self
```

```python
    def transform(self, X):
        print(f"[transform] Shape: {X.shape}")
        return X

# Use in a pipeline
pipeline = Pipeline([
    ('logger', ShapeLogger()),
    ('model', LogisticRegression(max_iter=5000))
])

pipeline.fit(X_train, y_train)
```

```
[fit] Shape: (426, 30)
[transform] Shape: (426, 30)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook. On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

Pipeline

ShapeLogger

LogisticRegression

Custom transformers can be inserted at any point in the pipeline, allowing you to debug, transform or inject domain-specific logic.

## Workflow Automation

Automation helps you build reproducible workflows that can scale. This includes automatic preprocessing, model selection, and hyperparameter tuning.

### Example: Pipeline with preprocessing, selection, and tuning

```python
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV

pipe = Pipeline([
    ('scale', StandardScaler()),
    ('select', SelectKBest(score_func=f_classif)),
    ('clf', LogisticRegression(max_iter=5000))
])

# Grid search over number of features and regularization strength
param_grid = {
    'select__k': [5, 10, 15],
    'clf__C': [0.1, 1.0, 10.0]
}

search = GridSearchCV(pipe, param_grid, cv=5)
search.fit(X_train, y_train)

print("Best parameters:", search.best_params_)
print("Best cross-validated score:", search.best_score_)
```

```python
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV

pipe = Pipeline([
    ('scale', StandardScaler()),
    ('select', SelectKBest(score_func=f_classif)),
    ('clf', LogisticRegression(max_iter=5000))
])

# Grid search over number of features and regularization strength
param_grid = {
    'select__k': [5, 10, 15],
    'clf__C': [0.1, 1.0, 10.0]
}

search = GridSearchCV(pipe, param_grid, cv=5)
search.fit(X_train, y_train)
```

```python

print("Best parameters:", search.best_params_)
print("Best cross-validated score:", search.best_score_)
```

```
Best parameters: {'clf__C': 1.0, 'select__k': 15}
Best cross-validated score: 0.9507523939808482
```

This shows a full automated flow: scaling → feature selection → modeling → tuning. It keeps everything clean, traceable, and ready for deployment.

## Integration with Other Libraries

scikit-learn integrates well with other libraries that extend its functionality.

### `mlxtend`: Model Stacking

`mlxtend` offers stacking and blending strategies that combine multiple classifiers into a meta-model.

```python
from mlxtend.classifier import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

meta_model = LogisticRegression()
stack = StackingClassifier(
    classifiers=[RandomForestClassifier(), SVC(
        probability=True)],
    meta_classifier=meta_model
)

stack.fit(X_train, y_train)
print("Stacking model accuracy:", stack.score(X_test,
    y_test))
```

```python
from mlxtend.classifier import StackingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

meta_model = LogisticRegression()
stack = StackingClassifier(
    classifiers=[RandomForestClassifier(), SVC(
        probability=True)],
    meta_classifier=meta_model
)

stack.fit(X_train, y_train)
print("Stacking model accuracy:", stack.score(X_test,
    y_test))
```

```
Stacking model accuracy: 0.965034965034965
```

Stacking can improve performance by combining the strengths of different models.

## Summary

| Technique | Tool | Description |
|---|---|---|
| Feature Selection | `SelectKBest` | Keep top features based on scoring function |
| Custom Pipelines | `TransformerMixin` | Add custom logic inside a scikit-learn pipeline |
| Automation | `GridSearchCV`, `Pipeline` | Automate tuning, selection, and preprocessing |
| Bayesian Search | `BayesSearchCV` | Efficient hyperparameter tuning |

| Technique | Tool | Description |
|---|---|---|
| Resampling | `SMOTE`, `imblearn` | Handle imbalanced datasets during training |
| Stacking | `mlxtend` | Combine multiple models into a stronger meta-model |

These advanced techniques will help you design more powerful and scalable machine learning systems.

# Reference Section

This final chapter provides a curated list of resources to help you go deeper into scikit-learn and related tools. Whether you're looking for official documentation, quick reference guides, datasets, tutorials, or advanced tools, you'll find plenty of links here to continue your learning journey.

## Official Documentation

- **scikit-learn main site**
  https://scikit-learn.org/stable/
- **User Guide** (step-by-step explanations of all topics)
  https://scikit-learn.org/stable/user_guide.html
- **API Reference** (classes, functions, parameters)
  https://scikit-learn.org/stable/modules/classes.html
- **Release Notes / Changelog**
  https://scikit-learn.org/stable/whats_new.html
- **Frequently Asked Questions**
  https://scikit-learn.org/stable/faq.html

## Datasets

- **Built-in Datasets in scikit-learn**
  https://scikit-learn.org/stable/datasets/toy_dataset.html
- **OpenML Integration** (load real-world datasets from OpenML)
  https://scikit-learn.org/stable/datasets/real_world.html
- **UCI Machine Learning Repository**
  https://archive.ics.uci.edu/ml/index.php
- **Kaggle Datasets**
  https://www.kaggle.com/datasets

## Learning Resources

- **Official Tutorials (Jupyter Notebooks)**
  https://scikit-learn.org/stable/tutorial/index.html
- **scikit-learn YouTube Channel**
  https://www.youtube.com/@scikitlearn
- **Course: Introduction to Machine Learning with scikit-learn** (free from DataCamp)
  https://www.datacamp.com/courses/supervised-learning-with-scikit-learn
- **Book: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (Aurélien Géron)**
  https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/

## Cheatsheets and Quick References

- **scikit-learn Algorithm Cheat-Sheet**
  https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html
- **scikit-learn Official Cheatsheet (PDF)**
  https://github.com/scikit-learn/scikit-learn/raw/main/doc/cheat_sheet/cheat_sheet.pdf
- **Pandas Cheatsheet (by DataCamp)**
  https://assets.datacamp.com/blog_assets/PandasPythonForDataScience.pdf
- **NumPy Cheatsheet (by DataCamp)**
  https://assets.datacamp.com/blog_assets/Numpy_Python_Cheat_Sheet.pdf
- **Matplotlib Cheatsheet (by DataCamp)**
  https://github.com/matplotlib/cheatsheets/blob/main/cheatsheets-1.pdf
- **Python Cheatsheet for Data Science**
  https://www.datacamp.com/community/blog/python-data-science-cheat-sheet

## Tools and Extensions

- **scikit-optimize**
  https://scikit-optimize.github.io/stable/
- **imbalanced-learn (imblearn)**
  https://imbalanced-learn.org/stable/
- **mlxtend**
  http://rasbt.github.io/mlxtend/

- **sklearn-pandas** (for DataFrame support in pipelines)
  https://github.com/scikit-learn-contrib/sklearn-pandas
- **Yellowbrick (visual diagnostic tools)**
  https://www.scikit-yb.org/en/latest/

## Additional Resources

- **Awesome scikit-learn (GitHub curated list)**
  https://github.com/EthicalML/awesome-scikit-learn
- **Awesome Machine Learning**
  https://github.com/josephmisiti/awesome-machine-learning
- **Kaggle Learn: Machine Learning with scikit-learn**
  https://www.kaggle.com/learn/intro-to-machine-learning
- **Scikit-learn GitHub Repository**
  https://github.com/scikit-learn/scikit-learn
- **Stack Overflow (tag: scikit-learn)**
  https://stackoverflow.com/questions/tagged/scikit-learn

## What’s Next?

Now that you’ve completed this guide, consider:

- Applying the concepts to a personal or work-related project.
- Contributing to an open-source project like scikit-learn.
- Learning about deployment (e.g. Flask, FastAPI, Docker).
- Exploring deep learning with libraries like PyTorch or TensorFlow.

Happy coding!

rubió
Metabolomics