机器学习与 Scikit-learn 实战指南

前言

欢迎来到机器学习的世界！本教程旨在帮助你从零开始，逐步掌握机器学习的基本概念、常用算法以及 Scikit-learn、XGBoost 等强大工具库的使用，最终达到能够理解和编写类似本教程开头示例代码的水平。

机器学习是人工智能的一个分支，它使计算机系统能够从数据中学习并改进，而无需进行显式编程。它已经渗透到我们生活的方方面面，从推荐系统、自然语言处理到金融风控、医疗诊断等。

本教程将结合理论与实践，通过清晰的讲解和 Python 代码示例，带你一步步构建机器学习模型，解决实际问题。

学习目标:

- 理解机器学习的基本概念和工作流程。
- 熟练使用 NumPy、Pandas 进行数据处理。
- 掌握 Scikit-learn 库进行数据预处理、模型训练和评估。
- 了解并应用 XGBoost 等集成学习算法。
- 学会使用 Optuna 进行超参数优化。
- 能够处理不平衡数据、进行特征工程。
- 最终能够独立分析问题，选择合适的模型并进行实现。

目标读者:

- 具备 Python 基础编程知识。
- 对数据科学和机器学习感兴趣的初学者。
- 希望系统学习 Scikit-learn 和 XGBoost 的开发者。

让我们开始这段激动人心的学习之旅吧！

第一章：机器学习基础

1.1 什么是机器学习?

简单来说，机器学习（Machine Learning, ML）是让计算机拥有像人一样的学习能力，通过分析大量数据来发现规律，并利用这些规律对未知数据进行预测或决策。

Arthur Samuel (1959): "机器学习是这样的一个领域，它赋予计算机学习的能力，而这种能力不是通过显式编程获得的。"

Tom Mitchell (1997): "对于某类任务 T 和性能度量 P，如果一个计算机程序在 T 上以 P衡量的性能随着经验 E 自我完善，那么我们称这个计算机程序在从经验 E 学习。"

- **任务 (T):** 程序需要完成的具体工作，例如预测降雨量、识别图片中的猫。
- **经验 (E):** 用于学习的数据，例如历史气象数据、带标签的猫图片。
- **性能度量 (P):** 评估程序在任务 T 上表现好坏的标准，例如预测准确率、均方误差。

1.2 机器学习的类型

机器学习主要分为以下几类：

1. **监督学习 (Supervised Learning):**
    - **特点:** 使用带有标签（答案）的数据进行训练。模型学习从输入特征到输出标签的映射关系。
    - **例子:**
        - **分类 (Classification):** 预测离散的类别标签。例如，判断邮件是否为垃圾邮件（是/否），识别图片中的动物类别（猫/狗/鸟）。
        - **回归 (Regression):** 预测连续的数值。例如，预测房价、预测股票价格、预测降雨量。
    - **本教程重点关注监督学习。**
2. **无监督学习 (Unsupervised Learning):**
    - **特点:** 使用没有标签的数据进行训练。模型需要自己发现数据中的结构和模式。
    - **例子:**
        - **聚类 (Clustering):** 将相似的数据点分组。例如，用户分群、新闻主题分类。
        - **降维 (Dimensionality Reduction):** 减少数据的特征数量，同时保留重要信息。例如，数据可视化、特征提取。
3. **强化学习 (Reinforcement Learning):**
    - **特点:** 模型（智能体 Agent）通过与环境 (Environment) 交互来学习。智能体根据环境的反馈（奖励 Reward 或惩罚 Punishment）调整自己的行为（动作 Action），以最大化累积奖励。
    - **例子:** 游戏 AI（AlphaGo）、机器人控制、自动驾驶策略。

1.3 为何选择 Python 进行机器学习？

Python 已成为机器学习和数据科学领域最流行的语言之一，主要原因在于其强大的生态系统：

- **简洁易学:** Python 语法清晰，上手快。
- **丰富的库:**
    - **NumPy:** 高性能科学计算和多维数组处理的基础库。
    - **Pandas:** 提供高效的数据结构（DataFrame, Series），方便进行数据清洗、处理和分析。
    - **Matplotlib & Seaborn:** 强大的数据可视化库，用于绘制各种图表。
    - **Scikit-learn:** 机器学习的核心库，提供了大量的预处理工具、机器学习算法和评估方法。
    - **XGBoost, LightGBM, CatBoost:** 高性能的梯度提升库，在各种竞赛和实际应用中表现出色。
    - **TensorFlow & PyTorch:** 深度学习框架。
- **活跃的社区:** 遇到问题时容易找到解决方案和支持。

1.4 环境设置

为了顺利进行后续学习，你需要安装 Python 和相关的库。推荐使用 Anaconda 发行版，它包含了 Python 解释器以及常用的科学计算库。

1. **下载并安装 Anaconda:** 访问 Anaconda 官网 下载适合你操作系统的版本并安装。
2. **创建虚拟环境 (推荐):** 打开 Anaconda Prompt (Windows) 或终端 (macOS/Linux)，创建一个独立的环境，避免库版本冲突。

```
conda create -n ml_env python=3.9 # 创建名为 ml_env 的环境，指定 Python 版本
conda activate ml_env             # 激活环境
```

3. **安装核心库:** 在激活的环境中安装必要的库。

```
conda install numpy pandas matplotlib seaborn scikit-learn jupyterlab # 基础
库
conda install -c conda-forge xgboost optuna joblib # XGBoost 和 Optuna
# 或者使用 pip (如果 conda 安装失败或需要特定版本)
# pip install numpy pandas matplotlib seaborn scikit-learn jupyterlab
xgboost optuna joblib
```

4. **启动 Jupyter Lab:** Jupyter Lab 提供了一个交互式的开发环境，非常适合数据分析和模型实验。

```
jupyter lab
```

这将在浏览器中打开 Jupyter Lab 界面。

1.5 机器学习的基本工作流程

一个典型的机器学习项目通常遵循以下步骤：

1. **定义问题 (Define Problem):** 明确要解决的任务（分类、回归等）和目标（例如，提高降雨预测的准确率）。
2. **收集数据 (Data Collection):** 获取与问题相关的数据。数据来源可以是数据库、文件（CSV, Excel, NPY）、API 等。
3. **数据准备与预处理 (Data Preparation & Preprocessing):** 这是机器学习中最耗时但至关重要的阶段。
    - **数据加载:** 将数据读入程序（例如，使用 Pandas）。
    - **数据探索与可视化 (EDA):** 理解数据的分布、特征之间的关系、发现异常值等。
    - **数据清洗:** 处理缺失值、异常值、重复值。
    - **特征工程 (Feature Engineering):** 创建新的特征或转换现有特征，以提高模型性能。例如，时间滞后特征、交互特征。
    - **特征缩放:** 将不同范围的特征缩放的相似的尺度。
    - **数据划分:** 将数据分为训练集、验证集（可选）和测试集。
4. **选择模型 (Model Selection):** 根据问题类型和数据特点选择合适的机器学习算法。
5. **模型训练 (Model Training):** 使用训练集数据训练选定的模型，让模型学习数据中的模式。
6. **模型评估 (Model Evaluation):** 使用测试集（或验证集）评估模型的性能，选择合适的评估指标（准确率、AUC、CSI 等）。
7. **超参数调优 (Hyperparameter Tuning):** 调整模型的超参数（例如，学习率、树的深度），以获得最佳性能。
8. **模型部署 (Model Deployment):** 将训练好的模型集成到实际应用中。
9. **模型监控与维护 (Monitoring & Maintenance):** 持续监控模型性能，根据需要重新训练或更新模型。

第二章：数据处理基石 - NumPy 与 Pandas

在进行机器学习之前，我们需要掌握处理数据的利器：NumPy 和 Pandas。

2.1 NumPy: 高性能数值计算

NumPy (Numerical Python) 是 Python 科学计算的基础包。它提供了：

- 强大的 N 维数组对象 (`ndarray`)。
- 成熟的函数库，用于操作数组。
- 线性代数、傅里叶变换和随机数生成等功能。

Scikit-learn 和许多其他库都建立在 NumPy 之上。

2.1.1 创建 NumPy 数组

```
import numpy as np

# 从列表创建
list_data = [1, 2, 3, 4, 5]
arr1d = np.array(list_data)
print(arr1d)
print(f"Shape: {arr1d.shape}, Type: {arr1d.dtype}") # dtype 通常是 int32 或 int64

list2d_data = [[1, 2, 3], [4, 5, 6]]
arr2d = np.array(list2d_data, dtype=np.float32) # 指定数据类型
print(arr2d)
print(f"Shape: {arr2d.shape}, Type: {arr2d.dtype}")

# 创建特殊数组
zeros_arr = np.zeros((2, 3)) # 创建全 0 数组
print("Zeros:\n", zeros_arr)

ones_arr = np.ones((3, 2), dtype=int) # 创建全 1 数组
print("Ones:\n", ones_arr)

range_arr = np.arange(0, 10, 2) # 类似 Python range，步长为 2
print("Range:\n", range_arr)

linspace_arr = np.linspace(0, 1, 5) # 创建等差数列，包含 5 个点
print("Linspace:\n", linspace_arr)

random_arr = np.random.rand(2, 2) # 创建 0-1 之间的随机数数组
print("Random:\n", random_arr)

random_int_arr = np.random.randint(0, 10, size=(3, 3)) # 创建指定范围的随机整数数组
print("Random Int:\n", random_int_arr)
```

2.1.2 数组索引与切片

NumPy 的索引和切片与 Python 列表类似，但功能更强大，尤其是在多维数组上。

```
arr = np.arange(10) # [0 1 2 3 4 5 6 7 8 9]
```

```
# 索引
print(arr[0])    # 0
print(arr[-1])   # 9

# 切片
print(arr[2:5])  # [2 3 4] (不包含索引 5)
print(arr[:3])   # [0 1 2] (从头开始)
print(arr[5:])   # [5 6 7 8 9] (到结尾)
print(arr[::2])  # [0 2 4 6 8] (步长为 2)

# 多维数组索引
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("arr2d:\n", arr2d)

print(arr2d[0, 1]) # 第 0 行, 第 1 列 -> 2
print(arr2d[1])    # 第 1 行 -> [4 5 6]
print(arr2d[:, 1]) # 第 1 列 -> [2 5 8]

# 多维数组切片
print("Slice 1:\n", arr2d[:2, 1:]) # 前 2 行, 第 1 列及之后 -> [[2 3], [5 6]]
print("Slice 2:\n", arr2d[1, :2])   # 第 1 行, 前 2 列 -> [4 5]

# 布尔索引 (非常重要!)
arr = np.array([10, 20, 5, 15, 25])
bool_idx = (arr > 15) # [False False False False  True]
print(arr[bool_idx])  # [25] - 只选择 True 对应的元素

print(arr[arr % 2 == 0]) # 选择所有偶数 -> [10 20]
```

2.1.3 NumPy 运算

NumPy 允许对整个数组执行快速的元素级运算（向量化运算），比 Python 循环快得多。

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# 元素级运算
print("Sum:", arr1 + arr2)      # [5 7 9]
print("Difference:", arr1 - arr2) # [-3 -3 -3]
print("Product:", arr1 * arr2)    # [ 4 10 18]
print("Division:", arr1 / arr2)    # [0.25 0.4  0.5 ]
print("Power:", arr1 ** 2)       # [1 4 9]

# 通用函数 (ufunc)
print("Square root:", np.sqrt(arr1))
print("Exponential:", np.exp(arr1))
print("Sin:", np.sin(arr1))

# 聚合运算
arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Total sum:", np.sum(arr))         # 21
print("Sum along axis 0 (columns):", np.sum(arr, axis=0)) # [5 7 9]
```

```
print("Sum along axis 1 (rows):", np.sum(arr, axis=1))   # [ 6 15]
print("Mean:", np.mean(arr))
print("Standard deviation:", np.std(arr))
print("Min:", np.min(arr))
print("Max:", np.max(arr))
print("Index of max:", np.argmax(arr)) # 展平后的最大值索引
```

2.1.4 数组形状操作

改变数组的形状而不改变其数据。

```
arr = np.arange(12) # [ 0  1  2  3  4  5  6  7  8  9 10 11]

# Reshape
reshaped_arr = arr.reshape((3, 4))
print("Reshaped:\n", reshaped_arr)

# Flatten (展平)
flattened_arr = reshaped_arr.flatten() # 或 reshaped_arr.ravel()
print("Flattened:", flattened_arr)

# Transpose (转置)
transposed_arr = reshaped_arr.T
print("Transposed:\n", transposed_arr)
print("Transposed shape:", transposed_arr.shape) # (4, 3)
```

2.1.5 NumPy 与内存

- **内存映射 (Memory Mapping):** 对于非常大的数组，可以使用 `np.load(filename, mmap_mode='r')` 以内存映射模式加载，只在需要时从磁盘读取数据块，节省内存。这在处理示例代码中的大型 `X_flat_features.npy` 时很有用。
- **数据类型:** 选择合适的数据类型（如 `np.float32` 代替 `np.float64`）可以显著减少内存占用。

NumPy 是后续学习的基础，务必熟练掌握其核心操作。

2.2 Pandas: 数据分析利器

Pandas 建立在 NumPy 之上，提供了更高级的数据结构和数据分析工具，特别适合处理表格化数据（类似 Excel 表格或 SQL 数据表）。

核心数据结构:

- **Series:** 一维带标签的数组，可以看作是带有索引的 NumPy 数组。
- **DataFrame:** 二维带标签的数据结构，可以看作是 Series 的容器，拥有行索引和列索引。这是 Pandas 中最常用的数据结构。

2.2.1 创建 Series 和 DataFrame

```
import pandas as pd

# 创建 Series
s = pd.Series([1, 3, 5, np.nan, 6, 8], name='MyNumbers') # np.nan 表示缺失值
print(s)

# 创建 DataFrame
dates = pd.date_range('20230101', periods=6) # 创建日期索引
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list('ABCD'))
print("\nDataFrame from random data:\n", df)

data = {'col1': [1, 2, 3, 4],
        'col2': ['A', 'B', 'C', 'D'],
        'col3': [True, False, True, False]}
df_from_dict = pd.DataFrame(data)
print("\nDataFrame from dict:\n", df_from_dict)
```

2.2.2 数据加载与保存

Pandas 可以方便地读写多种格式的数据文件。

```
# 假设存在 data.csv 文件
# col_a,col_b,col_c
# 1,apple,1.2
# 2,orange,3.5
# 3,banana,2.8

# 读取 CSV
# df_csv = pd.read_csv('data.csv')
# print(df_csv)

# 保存为 CSV
# df.to_csv('output.csv', index=False) # index=False 表示不保存行索引

# 读取 Excel (需要安装 openpyxl 或 xlrd)
# df_excel = pd.read_excel('data.xlsx', sheet_name='Sheet1')

# 保存为 Excel
# df.to_excel('output.xlsx', sheet_name='MySheet')

# 读取 NumPy 文件 (虽然通常用 np.load)
# arr = np.load('features.npy')
# df_npy = pd.DataFrame(arr, columns=[...]) # 需要提供列名
```

2.2.3 数据查看与选择

```
print("\n--- Data Viewing ---")
print("Head:\n", df.head(2)) # 查看前几行
print("Tail:\n", df.tail(2)) # 查看后几行
```

```
print("Index:", df.index)
print("Columns:", df.columns)
print("Describe:\n", df.describe()) # 查看数值列的统计摘要
print("Info:\n")
df.info() # 查看 DataFrame 的简要信息（列名，非空值数量，数据类型）

print("\n--- Data Selection ---")
# 选择列
print("Column A:\n", df['A']) # 选择单列，返回 Series
print("Columns A and C:\n", df[['A', 'C']]) # 选择多列，返回 DataFrame

# 选择行（基于标签 .loc）
print("Row by label:\n", df.loc[dates[0]]) # 选择第一行
print("Rows by label slice:\n", df.loc['20230102':'20230104']) # 选择多行

# 选择行（基于位置 .iloc）
print("Row by position:\n", df.iloc[0]) # 选择第一行
print("Rows by position slice:\n", df.iloc[1:3]) # 选择第 1，2 行（不含 3）
print("Specific elements:\n", df.iloc[[0, 2, 4], [0, 2]]) # 选择特定行和列

# 条件选择（布尔索引）
print("Rows where A > 0:\n", df[df['A'] > 0])
print("Rows where A > 0 and B < 0:\n", df[(df['A'] > 0) & (df['B'] < 0)])
```

2.2.4 处理缺失值

```
df_missing = df.copy()
df_missing.iloc[1, 1] = np.nan # 手动设置缺失值
df_missing.iloc[3, 2] = np.nan
print("\nDataFrame with missing values:\n", df_missing)

# 检查缺失值
print("Is Null:\n", df_missing.isnull())
print("Sum of Nulls per column:\n", df_missing.isnull().sum())

# 删除包含缺失值的行或列
print("Drop NA rows:\n", df_missing.dropna(how='any')) # 默认删除行
# print(df_missing.dropna(axis=1, how='all')) # 删除全为 NA 的列

# 填充缺失值
print("Fill NA with 0:\n", df_missing.fillna(value=0))
print("Fill NA with column mean:\n", df_missing.fillna(df_missing.mean())) # 用列
均值填充
```

2.2.5 数据操作与合并

```
# 应用函数
print("\nApply function (sqrt):\n", df.apply(np.sqrt)) # 对每一列应用函数
print("Apply lambda:\n", df['A'].apply(lambda x: x * 100)) # 对 Series 应用 lambda
```

```
# 排序
print("Sort by index (descending):\n", df.sort_index(axis=0, ascending=False))
print("Sort by values in column B:\n", df.sort_values(by='B'))

# 合并 (Concat, Merge, Join) - 类似 SQL 操作
df1 = pd.DataFrame({'A': ['A0', 'A1'], 'B': ['B0', 'B1']}, index=[0, 1])
df2 = pd.DataFrame({'A': ['A2', 'A3'], 'B': ['B2', 'B3']}, index=[2, 3])
concat_df = pd.concat([df1, df2]) # 沿 axis=0 (行) 连接
print("\nConcatenated DataFrame:\n", concat_df)

left = pd.DataFrame({'key': ['K0', 'K1', 'K2'], 'A': ['A0', 'A1', 'A2']})
right = pd.DataFrame({'key': ['K0', 'K1', 'K3'], 'B': ['B0', 'B1', 'B3']})
merged_df = pd.merge(left, right, on='key', how='inner') # 内连接
print("\nMerged DataFrame (inner):\n", merged_df)
# how 可以是 'left', 'right', 'outer'
```

Pandas 是数据预处理和探索性数据分析的核心工具。熟练使用 Pandas 将极大地提高你处理和理解数据的效率。

第三章：数据可视化 - Matplotlib 与 Seaborn

数据可视化是理解数据、发现模式、沟通结果的关键步骤。Python 的 Matplotlib 和 Seaborn 库提供了强大的可视化功能。

- **Matplotlib:** 是 Python 最基础、最广泛使用的绘图库。它提供了底层的绘图接口，可以绘制各种静态、动态、交互式的图表。
- **Seaborn:** 基于 Matplotlib 构建，提供了更高级的接口，专注于统计数据可视化，可以轻松绘制更美观、更复杂的图表。

通常我们会结合使用这两个库。

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

# 设置 Seaborn 风格 (可选，会让图表更好看)
sns.set_theme(style="whitegrid")

# 准备一些示例数据
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)

data = pd.DataFrame({
    'x': np.random.rand(100) * 10,
    'y': np.random.rand(100) * 10,
    'value': np.random.randn(100) * 10,
```

```
    'category': np.random.choice(['A', 'B', 'C', 'D'], 100)
})
```

3.1 Matplotlib 基础

3.1.1 常用图表

```
# 1. 折线图 (Line Plot) - 常用于显示趋势
plt.figure(figsize=(8, 4)) # 创建画布，指定大小
plt.plot(x, y1, label='Sin(x)', color='blue', linestyle='-')
plt.plot(x, y2, label='Cos(x)', color='red', linestyle='--')
plt.title('Sine and Cosine Waves') # 添加标题
plt.xlabel('X-axis')               # 添加 X 轴标签
plt.ylabel('Y-axis')               # 添加 Y 轴标签
plt.legend()                       # 显示图例
plt.grid(True)                     # 显示网格
plt.show()                         # 显示图表

# 2. 散点图 (Scatter Plot) - 常用于显示两个变量的关系
plt.figure(figsize=(6, 6))
plt.scatter(data['x'], data['y'], c=data['value'], cmap='viridis', alpha=0.7,
label='Data Points')
plt.title('Scatter Plot of X vs Y')
plt.xlabel('X Value')
plt.ylabel('Y Value')
plt.colorbar(label='Value') # 添加颜色条
plt.legend()
plt.show()

# 3. 柱状图 (Bar Chart) - 常用于比较类别数据
category_counts = data['category'].value_counts().sort_index()
plt.figure(figsize=(7, 4))
plt.bar(category_counts.index, category_counts.values, color=['skyblue',
'lightcoral', 'lightgreen', 'gold'])
plt.title('Counts per Category')
plt.xlabel('Category')
plt.ylabel('Count')
plt.show()

# 4. 直方图 (Histogram) - 常用于显示数据分布
plt.figure(figsize=(7, 4))
plt.hist(data['value'], bins=20, color='purple', alpha=0.7) # bins 控制条柱数量
plt.title('Distribution of Value')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.show()
```

3.1.2 子图 (Subplots)

在一个画布上绘制多个图表。

```
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(12, 4)) # 创建 1 行 2 列的子图

# 第一个子图
axes[0].plot(x, y1, label='Sin(x)', color='blue')
axes[0].set_title('Sine Wave')
axes[0].set_xlabel('X')
axes[0].set_ylabel('Sin(X)')
axes[0].legend()
axes[0].grid(True)

# 第二个子图
axes[1].scatter(data['x'], data['y'], label='Data', color='green', alpha=0.5)
axes[1].set_title('Scatter Plot')
axes[1].set_xlabel('X Value')
axes[1].set_ylabel('Y Value')
axes[1].legend()
axes[1].grid(True)

plt.tight_layout() # 调整子图布局，防止重叠
plt.show()
```

3.2 Seaborn 统计可视化

Seaborn 让绘制统计图表更加便捷美观。

3.2.1 分布图

```
# 1. 直方图与核密度估计 (Histogram + KDE)
plt.figure(figsize=(8, 5))
sns.histplot(data=data, x='value', kde=True, bins=20, color='teal') # kde=True 添加核密度曲线
plt.title('Distribution of Value (Seaborn)')
plt.show()

# 2. 核密度估计图 (KDE Plot) - 平滑的分布曲线
plt.figure(figsize=(8, 5))
sns.kdeplot(data=data, x='value', fill=True, color='magenta') # fill=True 填充曲线下方区域
plt.title('Kernel Density Estimate of Value')
plt.show()

# 3. 多变量分布 (Joint Plot) - 显示两个变量的联合分布和各自的边际分布
sns.jointplot(data=data, x='x', y='y', kind='scatter', color='orange') # kind 可以是 'kde', 'hist', 'reg'
plt.suptitle('Joint Distribution of X and Y', y=1.02) # 主标题
plt.show()
```

3.2.2 类别图

```
# 1. 箱线图 (Box Plot) - 显示类别数据的分布、中位数、四分位数和异常值
plt.figure(figsize=(8, 5))
sns.boxplot(data=data, x='category', y='value', palette='pastel')
plt.title('Value Distribution by Category (Box Plot)')
plt.show()

# 2. 小提琴图 (Violin Plot) - 箱线图与核密度估计的结合
plt.figure(figsize=(8, 5))
sns.violinplot(data=data, x='category', y='value', palette='muted')
plt.title('Value Distribution by Category (Violin Plot)')
plt.show()

# 3. 计数图 (Count Plot) - 显示类别数据的频数
plt.figure(figsize=(7, 4))
sns.countplot(data=data, x='category', palette='viridis')
plt.title('Category Counts (Seaborn)')
plt.show()
```

3.2.3 关系图

```
# 1. 散点图 (增强版) - 可以按类别着色或改变点的大小/形状
plt.figure(figsize=(8, 6))
sns.scatterplot(data=data, x='x', y='y', hue='category', size='value',
palette='bright', alpha=0.8)
plt.title('Scatter Plot with Category Hue and Value Size')
plt.show()

# 2. 配对图 (Pair Plot) - 显示数据集中多个数值变量两两之间的关系以及单个变量的分布
# 注意: 如果特征很多, 这个图会很大且计算耗时
# sns.pairplot(data[['x', 'y', 'value', 'category']].dropna(), hue='category',
palette='Set2') # dropna() 避免 NaN 问题
# plt.suptitle('Pair Plot of Numerical Features', y=1.02)
# plt.show()
```

3.2.4 热力图 (Heatmap)

常用于显示相关系数矩阵或混淆矩阵。

```
# 计算相关系数矩阵 (仅数值列)
correlation_matrix = data[['x', 'y', 'value']].corr()

plt.figure(figsize=(6, 5))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f") #
annot=True 显示数值, fmt 控制格式
plt.title('Correlation Matrix Heatmap')
plt.show()

# 示例: 绘制混淆矩阵 (在模型评估部分会用到)
# from sklearn.metrics import confusion_matrix
```

```
# y_true_example = [0, 1, 0, 1, 0, 1, 1, 0]
# y_pred_example = [0, 1, 1, 1, 0, 0, 1, 0]
# cm_example = confusion_matrix(y_true_example, y_pred_example)
# plt.figure(figsize=(5, 4))
# sns.heatmap(cm_example, annot=True, fmt='d', cmap='Blues', xticklabels=['Pred No Rain', 'Pred Rain'], yticklabels=['True No Rain', 'True Rain'])
# plt.title('Example Confusion Matrix')
# plt.ylabel('Actual Label')
# plt.xlabel('Predicted Label')
# plt.show()
```

3.3 可视化的重要性

- **探索性数据分析 (EDA):**
  - 理解特征分布（偏态、峰态）。
  - 发现特征之间的关系（线性、非线性）。
  - 识别异常值或离群点。
  - 比较不同类别的数据分布。
- **模型评估:**
  - 可视化混淆矩阵（如上例）。
  - 绘制 ROC 曲线和计算 AUC（将在评估章节讲解）。
  - 可视化特征重要性（如 XGBoost 示例代码所示）。
  - 绘制学习曲线，判断过拟合或欠拟合。
- **结果展示:**
  - 向他人清晰地传达模型的性能和数据的洞察。

熟练掌握数据可视化是进行有效机器学习的关键技能之一。

第四章：Scikit-learn 核心概念与数据预处理

Scikit-learn (sklearn) 是 Python 中最流行的通用机器学习库。它建立在 NumPy、SciPy 和 Matplotlib 之上，提供了大量用于数据预处理、模型选择、模型训练和评估的工具。

4.1 Scikit-learn API 设计理念

Scikit-learn 的 API 设计简洁且一致，主要围绕以下核心概念：

1. **估计器 (Estimator):**
   - 任何可以从数据中学习参数的对象都是估计器。例如，`LinearRegression`、`RandomForestClassifier`、`StandardScaler`。
   - 学习过程通过 `fit(X, y)` 方法触发，其中 `X` 是特征数据（通常是 NumPy 数组或稀疏矩阵），`y` 是目标标签（对于监督学习）。
   - 所有估计器的超参数（Hyperparameters）都可以在实例化时设置，或者通过 `set_params()` 方法修改。
   - 学习到的参数（例如，线性回归的系数）存储在估计器对象的以 `_` 结尾的属性中（例如，`model.coef_`）。
2. **转换器 (Transformer):**

- 一种特殊的估计器，用于对数据进行转换（预处理）。
- 除了 `fit(X, [y])` 方法（用于学习转换所需的参数，如均值和标准差），还必须实现 `transform(X)` 方法，用于应用转换。
- 通常还提供一个便捷方法 `fit_transform(X, [y])`，它等同于先调用 `fit` 再调用 `transform`，但有时更高效。
- 常见的转换器包括 `StandardScaler`、`MinMaxScaler`、`SimpleImputer`、`OneHotEncoder`。

3. **预测器 (Predictor):**

- 一种特殊的估计器，可以根据学习到的模型对新数据进行预测。
- 除了 `fit(X, y)`，还必须实现 `predict(X)` 方法，返回对输入 `X` 的预测结果。
- 分类器通常还提供 `predict_proba(X)` 方法，返回每个类别的概率。
- 回归器通常还提供 `score(X, y)` 方法，返回评估指标（如 R^2 分数）。
- 常见的预测器包括 `LogisticRegression`、`SVC`、`XGBClassifier`、`LinearRegression`、`RandomForestRegressor`。

这种一致的 API 使得组合不同的算法和预处理步骤变得非常容易。

4.2 数据划分：训练集与测试集

在训练模型之前，必须将数据集划分为训练集和测试集。

- **训练集 (Training Set):** 用于训练模型，让模型学习数据中的模式。
- **测试集 (Test Set):** 用于评估最终训练好的模型在**未见过**的数据上的性能。**测试集绝对不能用于训练或调优过程**，否则评估结果会过于乐观，无法反映模型的泛化能力。

Scikit-learn 提供了 `train_test_split` 函数来方便地进行数据划分。

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

# 假设我们有特征 X 和目标 y
# 创建一些示例数据
X_example = np.arange(100).reshape(50, 2) # 50 个样本，2 个特征
y_example = np.random.randint(0, 2, 50)   # 50 个二元标签

print("Original X shape:", X_example.shape)
print("Original y shape:", y_example.shape)

# 划分数据，通常测试集占 20%-30%
# random_state 保证每次划分结果一致，便于复现
# stratify=y 保证训练集和测试集中类别比例与原始数据大致相同（对分类问题很重要）
X_train, X_test, y_train, y_test = train_test_split(
    X_example, y_example,
    test_size=0.2,       # 测试集比例
    random_state=42,     # 随机种子
    stratify=y_example   # 按 y 进行分层抽样
)

print("\nAfter splitting:")
```

```
print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)

print("\nTrain set label distribution:", np.bincount(y_train))
print("Test set label distribution:", np.bincount(y_test))
```

注意: 在示例代码 `xgboost1.py` 和 `1_generate_base_predictions.py` 中，数据划分是基于 `X_flat` 和 `Y_binary` 进行的，使用了 `stratify=Y_binary` 来处理类别不平衡问题。

4.3 数据预处理 (Preprocessing)

原始数据通常不能直接用于训练机器学习模型，需要进行预处理。Scikit-learn 的 `sklearn.preprocessing` 模块提供了多种预处理工具。

重要原则: 预处理步骤（如计算均值、标准差、最大最小值等）**必须仅在训练集上 `fit`**，然后用学习到的参数**同时 `transform` 训练集和测试集**。这可以防止测试集的信息泄露到训练过程中。

4.3.1 处理缺失值 (`SimpleImputer`)

机器学习算法通常无法处理包含缺失值（NaN）的数据。`SimpleImputer` 可以用指定策略填充缺失值。

```
from sklearn.impute import SimpleImputer

# 创建包含缺失值的示例数据
X_missing = np.array([[1.0, 2.0, np.nan],
                      [4.0, np.nan, 6.0],
                      [7.0, 8.0, 9.0],
                      [np.nan, 11.0, 12.0]])
print("Original data with missing values:\n", X_missing)

# 1. 使用均值填充
imputer_mean = SimpleImputer(strategy='mean')
# fit 在训练集上计算均值
imputer_mean.fit(X_missing) # 假设这是训练数据
# transform 应用填充
X_filled_mean = imputer_mean.transform(X_missing)
print("\nFilled with mean:\n", X_filled_mean)
print("Learned means:", imputer_mean.statistics_)

# 2. 使用中位数填充（对异常值更鲁棒）
imputer_median = SimpleImputer(strategy='median')
X_filled_median = imputer_median.fit_transform(X_missing) # fit 和 transform 一步
完成
print("\nFilled with median:\n", X_filled_median)
print("Learned medians:", imputer_median.statistics_)

# 3. 使用众数填充（适用于类别特征）
# imputer_mode = SimpleImputer(strategy='most_frequent')
```

```
# 4. 使用常数填充
imputer_constant = SimpleImputer(strategy='constant', fill_value=-1)
X_filled_constant = imputer_constant.fit_transform(X_missing)
print("\nFilled with constant -1:\n", X_filled_constant)
```

注意: 在 `turn1.py` 中，使用了 `np.nan_to_num(X_flat, nan=0.0, posinf=0.0, neginf=0.0)` 来处理 NaN，这是一种简单的方法，相当于用 0 填充。使用 `SimpleImputer` 提供了更多策略选择。

4.3.2 特征缩放 (`StandardScaler`, `MinMaxScaler`)

许多机器学习算法（如 SVM、逻辑回归、神经网络，以及基于距离的算法如 KNN）对特征的尺度敏感。如果特征具有不同的范围（例如，年龄在 0-100 之间，收入在 0-1,000,000 之间），范围较大的特征可能会主导模型训练。特征缩放将所有特征调整到相似的范围。

- **StandardScaler (标准化/Z-score 标准化):**
    - 将特征缩放为均值为 0，标准差为 1。
    - 计算公式: `z = (x - mean) / std_dev`
    - 适用于数据近似高斯分布的情况，对异常值敏感。
- **MinMaxScaler (归一化):**
    - 将特征缩放到指定的范围（默认为 [0, 1]）。
    - 计算公式: `x_scaled = (x - min) / (max - min)`
    - 适用于数据范围已知或需要特定范围的情况，对异常值非常敏感。

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler

X_scale_example = np.array([[1., -1., 2.],
                            [2., 0., 0.],
                            [0., 1., -1.]])
print("Original data for scaling:\n", X_scale_example)

# 1. StandardScaler
scaler_std = StandardScaler()
# fit 计算均值和标准差
scaler_std.fit(X_train) # !! 仅在训练集上 fit !!
# transform 应用缩放
X_train_scaled_std = scaler_std.transform(X_train)
X_test_scaled_std = scaler_std.transform(X_test) # !! 用训练集的参数 transform 测试集 !!

print("\nStandardScaler - Learned mean:", scaler_std.mean_)
print("StandardScaler - Learned scale (std dev):", scaler_std.scale_)
# print("Scaled training data (StandardScaler):\n", X_train_scaled_std) # 输出会比较多
print("Mean of scaled train data (approx 0):", X_train_scaled_std.mean(axis=0))
print("Std dev of scaled train data (approx 1):", X_train_scaled_std.std(axis=0))

# 2. MinMaxScaler
scaler_minmax = MinMaxScaler(feature_range=(0, 1)) # 默认范围 [0, 1]
```

```
# fit 计算最小值和最大值
scaler_minmax.fit(X_train) # !! 仅在训练集上 fit !!
# transform 应用缩放
X_train_scaled_minmax = scaler_minmax.transform(X_train)
X_test_scaled_minmax = scaler_minmax.transform(X_test) # !! 用训练集的参数
transform 测试集 !!

print("\nMinMaxScaler - Learned min:", scaler_minmax.min_)
print("MinMaxScaler - Learned max:", scaler_minmax.data_max_)
# print("Scaled training data (MinMaxScaler):\n", X_train_scaled_minmax)
print("Min of scaled train data (approx 0):", X_train_scaled_minmax.min(axis=0))
print("Max of scaled train data (approx 1):", X_train_scaled_minmax.max(axis=0))
```

注意: 像 XGBoost 这样的树模型通常对特征缩放不敏感，因为树的分裂是基于阈值的。但在 `xgboost1.py` 中，虽然没有显式使用 `StandardScaler` 或 `MinMaxScaler`，但在特征工程 `turn1.py` 中计算的一些特征（如差分、标准化统计量）实际上隐式地进行了某种形式的缩放或归一化。

4.3.3 编码类别特征 (`OneHotEncoder`, `LabelEncoder`)

机器学习模型通常只能处理数值数据。类别特征（例如，“天气”={晴, 阴, 雨}，“城市”={北京, 上海, 广州}）需要转换为数值表示。

- **OneHotEncoder (独热编码):**
  - 为每个类别创建一个新的二元（0 或 1）特征。
  - 例如，“天气”={晴, 阴, 雨} 会变成三个特征：is_晴, is_阴, is_雨。如果样本是“阴”，则 is_阴=1，其他两个为 0。
  - 优点：避免了类别间的虚假顺序关系。
  - 缺点：如果类别数量非常多，会导致特征维度急剧增加（维度灾难）。
  - 通常是处理名义类别特征（Nominal Features）的首选方法。
- **LabelEncoder (标签编码):**
  - 为每个类别分配一个唯一的整数（例如，晴=0, 阴=1, 雨=2）。
  - 优点：简单，不增加维度。
  - 缺点：引入了虚假的顺序关系（例如，模型可能会认为 雨 > 阴 > 晴）。
  - **通常只适用于目标变量 `y` 的编码，或者当类别特征本身具有明确顺序（Ordinal Features）时才用于 `X`。**

```
from sklearn.preprocessing import OneHotEncoder, LabelEncoder

# 示例类别数据
X_categorical = pd.DataFrame({
    'Color': ['Red', 'Green', 'Blue', 'Green'],
    'Size': ['M', 'L', 'S', 'M']
})
print("Original categorical data:\n", X_categorical)

# 1. OneHotEncoder
# handle_unknown='ignore' 使得在 transform 时遇到训练时未见的类别时，所有独热列都为 0
```

```
# sparse_output=False 返回 NumPy 数组，否则返回稀疏矩阵（当类别多时节省内存）
onehot_encoder = OneHotEncoder(handle_unknown='ignore', sparse_output=False)
onehot_encoder.fit(X_categorical) # 学习类别
X_onehot = onehot_encoder.transform(X_categorical)
print("\nOneHot Encoded data:\n", X_onehot)
print("Learned categories:", onehot_encoder.categories_)
print("Feature names:", onehot_encoder.get_feature_names_out(['Color', 'Size']))

# 2. LabelEncoder (通常用于 y)
y_labels = np.array(['Cat', 'Dog', 'Cat', 'Bird', 'Dog'])
print("\nOriginal labels (y):", y_labels)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y_labels)
print("Label Encoded labels (y):", y_encoded)
print("Learned classes:", label_encoder.classes_)
# 可以反向转换
print("Decoded labels:", label_encoder.inverse_transform(y_encoded))

# LabelEncoder 用于 X (仅当有序时推荐)
size_encoder = LabelEncoder()
X_categorical['Size_Encoded'] = size_encoder.fit_transform(X_categorical['Size'])
print("\nDataFrame with Label Encoded 'Size':\n", X_categorical)
print("Size classes:", size_encoder.classes_) # 注意顺序可能不是期望的 S, M, L
```

注意: 在 `turn1.py` 中，`season_onehot` 特征就是手动实现的独热编码。`intensity_bins` 也是一种将连续特征（product_mean）离散化并进行类似独热编码的方式。

4.4 Pipeline: 串联预处理与模型

通常，一个完整的机器学习流程包含多个预处理步骤和一个最终的模型。Scikit-learn 的 `Pipeline` 可以将这些步骤串联起来，形成一个单一的估计器。

优点:

- **代码简洁:** 将多个步骤封装成一个对象。
- **方便交叉验证和网格搜索:** 可以直接对整个 Pipeline 进行超参数调优。
- **防止数据泄露:** 确保预处理步骤在交叉验证的每一折中都只在训练部分 `fit`。

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# 假设我们有一个包含数值和类别特征的数据集
# 以及一个包含缺失值的数值特征
X_pipe_example = pd.DataFrame({
    'numeric1': [1, 2, np.nan, 4, 5],
    'numeric2': [10, 11, 12, 13, 14],
    'category': ['A', 'B', 'A', 'C', 'B']
})
y_pipe_example = np.array([0, 1, 0, 1, 1])

# 划分数据
```

```
X_train_p, X_test_p, y_train_p, y_test_p = train_test_split(
    X_pipe_example, y_pipe_example, test_size=0.2, random_state=42,
stratify=y_pipe_example
)

# 创建 Pipeline
# 步骤是（名称，转换器/估计器）的列表
# 注意：最后一个步骤必须是估计器（模型）
# 这里我们先只处理数值特征
numeric_pipeline = Pipeline([
    ('imputer', SimpleImputer(strategy='mean')), # 步骤1：填充缺失值
    ('scaler', StandardScaler()),                # 步骤2：标准化
    ('classifier', LogisticRegression(random_state=42)) # 步骤3：模型
])

print("\nPipeline definition:\n", numeric_pipeline)

# 训练 Pipeline（只用数值特征）
numeric_features = ['numeric1', 'numeric2']
numeric_pipeline.fit(X_train_p[numeric_features], y_train_p)

# 预测
y_pred_p = numeric_pipeline.predict(X_test_p[numeric_features])
print("\nPipeline predictions on test set:", y_pred_p)

# 评估
accuracy = numeric_pipeline.score(X_test_p[numeric_features], y_test_p)
print("Pipeline accuracy on test set:", accuracy)

# 可以访问 Pipeline 中的步骤
print("\nAccessing steps in pipeline:")
print("Imputer learned means:",
numeric_pipeline.named_steps['imputer'].statistics_)
print("Scaler learned means:", numeric_pipeline.named_steps['scaler'].mean_)
print("Classifier coefficients:",
numeric_pipeline.named_steps['classifier'].coef_)
```

4.5 ColumnTransformer: 对不同列应用不同转换

通常，数据集包含不同类型的列（数值、类别），需要应用不同的预处理步骤。`ColumnTransformer` 允许我们将不同的转换器应用到不同的列子集上，并将结果合并。

```
from sklearn.compose import ColumnTransformer

# 定义数值特征的处理流程
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])

# 定义类别特征的处理流程
```

```
categorical_transformer = Pipeline(steps=[
    ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
])

# 使用 ColumnTransformer 将转换器应用到对应的列
# (名称，转换器，列名列表或索引)
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, ['numeric1', 'numeric2']), # 应用
numeric_transformer 到这两列
        ('cat', categorical_transformer, ['category'])         # 应用
categorical_transformer 到这一列
    ],
    remainder='passthrough' # 'passthrough' 保留未指定的列，'drop' 丢弃未指定的列
)

# 将预处理器和最终模型组合成一个完整的 Pipeline
full_pipeline = Pipeline(steps=[
    ('preprocessor', preprocessor),
    ('classifier', LogisticRegression(random_state=42))
])

print("\nFull pipeline with ColumnTransformer:\n", full_pipeline)

# 训练完整的 Pipeline
full_pipeline.fit(X_train_p, y_train_p)

# 预测
y_pred_full = full_pipeline.predict(X_test_p)
print("\nFull pipeline predictions:", y_pred_full)

# 评估
accuracy_full = full_pipeline.score(X_test_p, y_test_p)
print("Full pipeline accuracy:", accuracy_full)

# 查看预处理后的特征名称 (如果 OneHotEncoder 使用了 get_feature_names_out)
# feature_names_out =
full_pipeline.named_steps['preprocessor'].get_feature_names_out()
# print("\nFeature names after preprocessing:", feature_names_out)
```

`Pipeline` 和 `ColumnTransformer` 是构建健壮、可维护的机器学习工作流的强大工具，强烈推荐在实际项目中使用。

第五章：常用机器学习算法 (分类)

在数据准备好之后，下一步是选择并训练一个合适的模型。本章介绍几种 Scikit-learn 中常用的分类算法。我们将使用前面章节准备的 `X_train`, `X_test`, `y_train`, `y_test` (或其缩放版本) 作为示例数据。

```
# 假设我们已经有了划分好的训练集和测试集
# X_train, X_test, y_train, y_test
```

```
# 以及可能经过缩放的数据
# X_train_scaled_std, X_test_scaled_std
# X_train_scaled_minmax, X_test_scaled_minmax

# 为了代码示例，我们重新生成一些简单数据
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

X_clf, y_clf = make_classification(n_samples=1000, n_features=20,
n_informative=15, n_redundant=5,
                                   n_classes=2, random_state=42, flip_y=0.1)

# 划分数据
X_train_clf, X_test_clf, y_train_clf, y_test_clf = train_test_split(
    X_clf, y_clf, test_size=0.2, random_state=42, stratify=y_clf
)

# 特征缩放（对于某些模型很重要）
scaler = StandardScaler()
X_train_clf_scaled = scaler.fit_transform(X_train_clf)
X_test_clf_scaled = scaler.transform(X_test_clf)

# 引入评估函数
from sklearn.metrics import accuracy_score, classification_report, roc_auc_score
```

5.1 逻辑回归 (Logistic Regression)

虽然名字带有"回归"，但逻辑回归是一种广泛使用的**线性分类**算法。

- **核心思想:** 它通过一个 Sigmoid (Logistic) 函数将线性回归的输出映射到 (0, 1) 区间，表示样本属于正类的概率。然后根据概率和一个阈值（通常是 0.5）进行分类。
- **模型:** `P(y=1 | X) = 1 / (1 + exp(-(w*X + b)))`
- **特点:**
  - 模型简单，易于理解和实现。
  - 输出概率，解释性好。
  - 对特征尺度敏感，通常需要进行特征缩放。
  - 本质上是线性的，对于非线性问题需要结合特征工程或核技巧。
- **主要超参数:**
  - `C`: 正则化强度的倒数。较小的值表示更强的正则化（防止过拟合）。
  - `penalty`: 正则化类型 ('l1', 'l2', 'elasticnet', 'none')。L1 正则化可以产生稀疏解（特征选择）。
  - `solver`: 用于优化的算法 ('liblinear', 'lbfgs', 'saga' 等)。不同的 solver 支持不同的 penalty。

```
from sklearn.linear_model import LogisticRegression

# 1. 初始化模型
# 使用 L2 正则化，C=1.0 是默认值
log_reg = LogisticRegression(C=1.0, penalty='l2', solver='liblinear',
random_state=42)
```

```
# 2. 训练模型 (使用缩放后的数据)
log_reg.fit(X_train_clf_scaled, y_train_clf)

# 3. 预测
y_pred_log_reg = log_reg.predict(X_test_clf_scaled)
y_pred_proba_log_reg = log_reg.predict_proba(X_test_clf_scaled)[:, 1] # 获取正类的
概率

# 4. 评估
print("--- Logistic Regression Evaluation ---")
print("Accuracy:", accuracy_score(y_test_clf, y_pred_log_reg))
print("AUC:", roc_auc_score(y_test_clf, y_pred_proba_log_reg))
# print(classification_report(y_test_clf, y_pred_log_reg))
```

5.2 支持向量机 (Support Vector Machine, SVM)

SVM 是一种强大的分类器，特别擅长处理高维数据和非线性问题。

- **核心思想:**
  - **线性 SVM:** 寻找一个能够将不同类别分开的**最大间隔超平面 (Maximum Margin Hyperplane)**。离超平面最近的那些点被称为**支持向量 (Support Vectors)**。
  - **非线性 SVM:** 使用**核技巧 (Kernel Trick)**（如 RBF 核、多项式核）将数据隐式地映射到更高维空间，使得原本线性不可分的数据在高维空间中变得线性可分。
- **特点:**
  - 在高维空间中表现良好。
  - 内存效率高（只使用支持向量）。
  - 核技巧使其能够处理复杂的非线性关系。
  - 对特征尺度非常敏感，**必须进行特征缩放**。
  - 对核函数和超参数的选择比较敏感。
  - 对于非常大的数据集，训练时间可能较长。
  - 默认不直接输出概率，但可以通过设置 `probability=True` (会增加计算时间)。
- **主要超参数 (SVC - Support Vector Classifier):**
  - `C`: 正则化参数。与逻辑回归类似，控制错误分类的惩罚。值越大，间隔越小，容忍的错误越少。
  - `kernel`: 核函数类型 ('linear', 'poly', 'rbf', 'sigmoid')。'rbf' (径向基函数/高斯核) 是常用的默认选择。
  - `gamma`: RBF、poly、sigmoid 核的系数。控制单个样本影响的范围。值越大，影响范围越小，模型越复杂，可能导致过拟合。
  - `degree`: 多项式核 (poly) 的阶数。

```
from sklearn.svm import SVC

# 1. 初始化模型 (使用 RBF 核)
# probability=True 可以调用 predict_proba, 但会增加训练时间
svm_clf = SVC(C=1.0, kernel='rbf', gamma='scale', probability=True,
random_state=42)
# gamma='scale' 是一个常用的默认值: 1 / (n_features * X.var())

# 2. 训练模型 (使用缩放后的数据)
```

```
svm_clf.fit(X_train_clf_scaled, y_train_clf)

# 3. 预测
y_pred_svm = svm_clf.predict(X_test_clf_scaled)
y_pred_proba_svm = svm_clf.predict_proba(X_test_clf_scaled)[:, 1]

# 4. 评估
print("\n--- Support Vector Machine (SVC) Evaluation ---")
print("Accuracy:", accuracy_score(y_test_clf, y_pred_svm))
print("AUC:", roc_auc_score(y_test_clf, y_pred_proba_svm))
# print(classification_report(y_test_clf, y_pred_svm))
```

5.3 K-最近邻 (K-Nearest Neighbors, KNN)

KNN 是一种简单直观的**非参数、基于实例**的分类算法。

- **核心思想:** 对于一个新的数据点，找到训练集中与它最相似（距离最近）的 K 个邻居，然后根据这 K 个邻居的类别进行投票（少数服从多数）来决定新数据点的类别。
- **特点:**
  - 算法简单，易于理解。
  - 无需显式训练模型（惰性学习 Lazy Learning），训练阶段只是存储数据。
  - 对数据分布没有假设。
  - 对特征尺度非常敏感，**必须进行特征缩放**。
  - 预测阶段计算量大，需要计算新样本与所有训练样本的距离。
  - 对 K 值的选择敏感。
  - 容易受到维度灾难的影响（高维空间中距离度量可能失效）。
- **主要超参数:**
  - `n_neighbors`: K 的值，即邻居的数量。
  - `weights`: 投票权重 ('uniform' - 所有邻居权重相同, 'distance' - 距离越近权重越大)。
  - `metric`: 距离度量方法 ('minkowski' - 默认，p=2 时为欧氏距离, p=1 时为曼哈顿距离)。
  - `p`: Minkowski 距离的参数。

```
from sklearn.neighbors import KNeighborsClassifier

# 1. 初始化模型
knn_clf = KNeighborsClassifier(n_neighbors=5, weights='uniform',
metric='minkowski', p=2)

# 2. 训练模型 (使用缩放后的数据) - KNN 的 fit 只是存储数据
knn_clf.fit(X_train_clf_scaled, y_train_clf)

# 3. 预测
y_pred_knn = knn_clf.predict(X_test_clf_scaled)
y_pred_proba_knn = knn_clf.predict_proba(X_test_clf_scaled)[:, 1]

# 4. 评估
print("\n--- K-Nearest Neighbors (KNN) Evaluation ---")
print("Accuracy:", accuracy_score(y_test_clf, y_pred_knn))
```

```
print("AUC:", roc_auc_score(y_test_clf, y_pred_proba_knn))
# print(classification_report(y_test_clf, y_pred_knn))
```

5.4 决策树 (Decision Tree)

决策树是一种模仿人类决策过程的树状结构模型。

- **核心思想:** 从根节点开始，根据某个特征的阈值将数据划分到不同的子节点，重复这个过程直到达到叶子节点（代表一个类别或一个预测值）。选择划分特征和阈值的标准通常是使得划分后的**信息熵 (Information Entropy)** 或 **基尼不纯度 (Gini Impurity)** 最小化。
- **特点:**
  - 模型直观，易于解释和可视化。
  - 可以处理数值和类别特征。
  - 对特征缩放不敏感。
  - 容易过拟合（生成过于复杂的树），需要进行剪枝 (Pruning) 或设置停止条件。
  - 对于特征中微小的变动可能导致生成完全不同的树，不太稳定。
- **主要超参数:**
  - `criterion`: 划分质量的度量 ('gini' - 基尼不纯度, 'entropy' - 信息增益)。
  - `max_depth`: 树的最大深度。控制过拟合的关键参数。
  - `min_samples_split`: 内部节点分裂所需的最小样本数。
  - `min_samples_leaf`: 叶子节点所需的最小样本数。
  - `max_features`: 寻找最佳分裂时考虑的特征数量。

```
from sklearn.tree import DecisionTreeClassifier
# from sklearn.tree import plot_tree # 用于可视化

# 1. 初始化模型（限制深度防止过拟合）
dt_clf = DecisionTreeClassifier(criterion='gini', max_depth=10,
min_samples_split=5,
                                min_samples_leaf=3, random_state=42)

# 2. 训练模型（可以使用原始数据或缩放数据，树模型不敏感）
dt_clf.fit(X_train_clf, y_train_clf)

# 3. 预测
y_pred_dt = dt_clf.predict(X_test_clf)
y_pred_proba_dt = dt_clf.predict_proba(X_test_clf)[:, 1]

# 4. 评估
print("\n--- Decision Tree Evaluation ---")
print("Accuracy:", accuracy_score(y_test_clf, y_pred_dt))
print("AUC:", roc_auc_score(y_test_clf, y_pred_proba_dt))
# print(classification_report(y_test_clf, y_pred_dt))

# 可视化（如果需要）
# plt.figure(figsize=(20,10))
# plot_tree(dt_clf, filled=True, feature_names=[f'feature_{i}' for i in
range(X_train_clf.shape[1])], class_names=['0', '1'], max_depth=3, fontsize=10)
# plt.show()
```

5.5 随机森林 (Random Forest)

随机森林是一种强大的**集成学习 (Ensemble Learning)** 方法，它通过构建多个决策树并结合它们的预测结果来提高模型的性能和稳定性。

- **核心思想:**
  - **Bagging (Bootstrap Aggregating):** 从原始训练集中有放回地抽取多个子样本集（Bootstrap Samples）。
  - **随机特征选择:** 在每个子样本集上训练一个决策树。在每个节点分裂时，不是考虑所有特征，而是随机选择一部分特征进行最佳分裂点的寻找。
  - **投票:** 对于分类问题，最终预测结果由所有树的投票决定（多数票）。
- **特点:**
  - 通常比单棵决策树具有更高的准确性和稳定性。
  - 能有效防止过拟合。
  - 能够处理高维数据。
  - 对特征缩放不敏感。
  - 可以评估特征的重要性。
  - 模型复杂度较高，解释性不如单棵决策树。
  - 训练时间和内存消耗相对较大。
- **主要超参数:**
  - `n_estimators`: 森林中树的数量。通常越多越好，但会增加计算成本，超过一定数量后性能提升会饱和。
  - `criterion`, `max_depth`, `min_samples_split`, `min_samples_leaf`, `max_features`: 与决策树类似，但通常 `max_features` 设置为 'sqrt' (特征数的平方根) 或 'log2'。
  - `bootstrap`: 是否使用 Bootstrap 样本 (True - 默认)。
  - `oob_score`: 是否使用袋外样本 (Out-of-Bag Samples) 来估计泛化误差 (True)。OOB 样本是那些在构建某棵树时未被抽到的样本。

```
from sklearn.ensemble import RandomForestClassifier

# 1. 初始化模型
rf_clf = RandomForestClassifier(n_estimators=100, # 树的数量
                                max_depth=15,      # 限制树深度
                                min_samples_split=5,
                                min_samples_leaf=3,
                                max_features='sqrt', # 每次分裂考虑
sqrt(n_features) 个特征
                                bootstrap=True,
                                oob_score=True,    # 使用 OOB 评估
                                random_state=42,
                                n_jobs=-1)         # 使用所有 CPU 核心

# 2. 训练模型 (可以使用原始数据或缩放数据)
rf_clf.fit(X_train_clf, y_train_clf)

# 3. 预测
y_pred_rf = rf_clf.predict(X_test_clf)
y_pred_proba_rf = rf_clf.predict_proba(X_test_clf)[:, 1]
```

```
# 4. 评估
print("\n--- Random Forest Evaluation ---")
if rf_clf.oob_score_:
    print(f"OOB Score: {rf_clf.oob_score_:.4f}") # 袋外样本评估准确率
print("Test Accuracy:", accuracy_score(y_test_clf, y_pred_rf))
print("Test AUC:", roc_auc_score(y_test_clf, y_pred_proba_rf))
# print(classification_report(y_test_clf, y_pred_rf))

# 查看特征重要性
# importances = rf_clf.feature_importances_
# feature_importance_df = pd.DataFrame({'Feature': [f'feature_{i}' for i in
range(X_train_clf.shape[1])], 'Importance': importances})
# print("\nFeature Importances:\n",
feature_importance_df.sort_values(by='Importance', ascending=False).head(10))
```

5.6 其他常用分类器

- **朴素贝叶斯 (Naive Bayes):** 基于贝叶斯定理和特征条件独立假设的分类器。简单快速，特别适用于文本分类。有多种变体（高斯、多项式、伯努利）。
- **梯度提升决策树 (Gradient Boosting Decision Tree, GBDT):** 另一种强大的集成方法，通过迭代地训练新的树来修正前面树的残差。Scikit-learn 提供 `GradientBoostingClassifier`。
- **XGBoost, LightGBM, CatBoost:** GBDT 的高效实现和改进版本，通常性能更好，速度更快，支持更多功能（如正则化、缺失值处理）。这些库需要单独安装，但提供了 Scikit-learn 风格的接口。**示例代码中使用的就是 XGBoost (`xgb.XGBClassifier`)。**我们将在后续章节详细讨论 XGBoost。

选择哪个模型取决于具体问题、数据特性、计算资源以及对模型解释性的要求。通常需要尝试多种模型并进行比较。

第六章：模型评估与选择

训练好模型后，我们需要客观地评估其性能，并根据评估结果选择最佳模型或调整模型。简单地看模型在训练集上的表现是不可靠的，因为它可能只是记住了训练数据（过拟合），而在未见过的数据上表现很差。我们真正关心的是模型的**泛化能力 (Generalization Ability)**，即在**测试集**或未来新数据上的表现。

6.1 分类评估指标

对于分类问题，有多种评估指标，选择哪个指标取决于具体的业务场景和目标。

6.1.1 混淆矩阵 (Confusion Matrix)

混淆矩阵是评估分类模型性能的基础，它展示了模型预测结果与真实标签之间的对应关系。对于二分类问题（例如，预测是否下雨），混淆矩阵通常如下：

	预测为负类 (No Rain)	**预测为正类 (Rain)**
实际为负类 (No Rain)	True Negative (TN)	False Positive (FP)
实际为正类 (Rain)	False Negative (FN)	True Positive (TP)

- **True Positive (TP):** 实际为正类，预测也为正类（正确预测下雨）。
- **True Negative (TN):** 实际为负类，预测也为负类（正确预测不下雨）。
- **False Positive (FP):** 实际为负类，预测为正类（错误预测下雨，"误报"，Type I Error）。
- **False Negative (FN):** 实际为正类，预测为负类（错误预测不下雨，"漏报"，Type II Error）。

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# 使用前面章节的随机森林预测结果作为例子
# y_test_clf: 真实标签
# y_pred_rf: 随机森林预测的标签 (通常基于 0.5 阈值)

cm = confusion_matrix(y_test_clf, y_pred_rf)
print("Confusion Matrix (Random Forest):\n", cm)

tn, fp, fn, tp = cm.ravel() # 解包矩阵元素
print(f"TN: {tn}, FP: {fp}, FN: {fn}, TP: {tp}")

# 可视化混淆矩阵
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['No Rain',
'Rain'])
disp.plot(cmap=plt.cm.Blues)
plt.title('Confusion Matrix - Random Forest')
plt.show()
```

6.1.2 准确率 (Accuracy)

- **定义:** 预测正确的样本数占总样本数的比例。
- **公式:** `Accuracy = (TP + TN) / (TP + TN + FP + FN)`
- **优点:** 直观易懂。
- **缺点:** 在**类别不平衡**的数据集上具有误导性。例如，如果 99% 的样本是负类，一个将所有样本都预测为负类的模型也能达到 99% 的准确率，但这显然不是一个好模型。

```
from sklearn.metrics import accuracy_score

acc = accuracy_score(y_test_clf, y_pred_rf)
print(f"Accuracy: {acc:.4f}")
# 或者直接从混淆矩阵计算: (tp + tn) / (tp + tn + fp + fn)
```

6.1.3 精确率 (Precision)

- **定义:** 在所有**预测为正类**的样本中，实际也为正类的比例。衡量模型预测的正类有多"准"。
- **公式:** `Precision = TP / (TP + FP)`
- **关注点:** 减少误报 (FP)。适用于那些"宁可放过，不可杀错"的场景，例如垃圾邮件检测（将正常邮件误判为垃圾邮件的代价很高）。

```

```

```
from sklearn.metrics import precision_score

precision = precision_score(y_test_clf, y_pred_rf) # 默认计算正类的精确率
print(f"Precision (for class 1 'Rain'): {precision:.4f}")
```

6.1.4 召回率 (Recall) / 敏感度 (Sensitivity) / 命中率 (Hit Rate) / 检出概率 (POD)

- **定义:** 在所有**实际为正类**的样本中，被模型成功预测为正类的比例。衡量模型有多能"找全"正类样本。
- **公式:** `Recall = TP / (TP + FN)`
- **关注点:** 减少漏报 (FN)。适用于那些"宁可杀错，不可放过"的场景，例如疾病诊断（漏诊的代价很高）、灾害预警（如降雨预测，漏报强降雨可能导致严重后果）。
- **在气象领域，这个指标常被称为 POD (Probability of Detection)。**

```
from sklearn.metrics import recall_score

recall = recall_score(y_test_clf, y_pred_rf) # 默认计算正类的召回率
print(f"Recall / POD (for class 1 'Rain'): {recall:.4f}")
```

6.1.5 F1 分数 (F1-Score)

- **定义:** 精确率和召回率的调和平均数。综合考虑了精确率和召回率。
- **公式:** `F1 = 2 * (Precision * Recall) / (Precision + Recall)`
- **特点:** 当精确率和召回率都较高时，F1 分数也会较高。它比准确率更能反映模型在不平衡数据上的性能。

```
from sklearn.metrics import f1_score

f1 = f1_score(y_test_clf, y_pred_rf) # 默认计算正类的 F1 分数
print(f"F1 Score (for class 1 'Rain'): {f1:.4f}")
```

6.1.6 特异度 (Specificity) / 真负类率 (True Negative Rate)

- **定义:** 在所有**实际为负类**的样本中，被模型成功预测为负类的比例。衡量模型正确识别负类的能力。
- **公式:** `Specificity = TN / (TN + FP)`
- **与召回率的关系:** 召回率关注正类，特异度关注负类。

```
# Scikit-learn 没有直接的 specificity_score，但可以从混淆矩阵计算
specificity = tn / (tn + fp) if (tn + fp) > 0 else 0
print(f"Specificity (for class 0 'No Rain'): {specificity:.4f}")
# 或者计算负类的召回率: recall_score(y_test_clf, y_pred_rf, pos_label=0)
```

6.1.7 误报率 (False Alarm Ratio, FAR)

- **定义:** 在所有**预测为正类**的样本中，实际为负类的比例。衡量预测的正类中有多少是错误的。

- **公式:** `FAR = FP / (TP + FP)`
- **与精确率的关系:** `FAR = 1 - Precision`
- **关注点:** 衡量误报的程度。在气象预报中常用，希望 FAR 尽可能低。

```
# Scikit-learn 没有直接的 far_score，但可以从混淆矩阵计算
far = fp / (tp + fp) if (tp + fp) > 0 else 0
print(f"False Alarm Ratio (FAR): {far:.4f}")
```

6.1.8 临界成功指数 (Critical Success Index, CSI) / 威胁评分 (Threat Score, TS)

- **定义:** 正确预测的正类占所有预测为正类或实际为正类的样本（除去正确预测的负类）的比例。
- **公式:** `CSI = TP / (TP + FN + FP)`
- **特点:** 综合考虑了命中 (TP)、漏报 (FN) 和误报 (FP)，对 TN 不敏感。在气象等领域广泛用于评估事件（如降雨）预报的整体技巧。CSI 越高越好。

```
# Scikit-learn 没有直接的 csi_score，但可以从混淆矩阵计算
csi = tp / (tp + fn + fp) if (tp + fn + fp) > 0 else 0
print(f"Critical Success Index (CSI) / Threat Score (TS): {csi:.4f}")
```

6.1.9 ROC 曲线 (Receiver Operating Characteristic Curve) 与 AUC (Area Under the Curve)

- **背景:** 大多数分类器输出的是概率，我们需要选择一个阈值（如 0.5）来将概率转换为类别预测。不同的阈值会产生不同的 TP、FP、TN、FN，从而影响上述指标。
- **ROC 曲线:** 以**假正类率 (False Positive Rate, FPR = FP / (FP + TN) = 1 - Specificity)** 为横坐标，**真正类率 (True Positive Rate, TPR = Recall = TP / (TP + FN))** 为纵坐标，绘制不同阈值下的点连接成的曲线。
- **AUC (Area Under the ROC Curve):** ROC 曲线下的面积。
  - AUC 值介于 0 和 1 之间。
  - AUC = 1: 完美分类器。
  - AUC = 0.5: 随机猜测（无区分能力）。
  - AUC < 0.5: 比随机猜测还差。
  - AUC 衡量的是模型**整体区分正负样本的能力**，与具体阈值无关。它对于评估不平衡数据集上的模型性能尤其有用。AUC 越高，模型性能越好。

```
from sklearn.metrics import roc_curve, auc, roc_auc_score

# 需要使用预测概率 y_pred_proba_rf (正类的概率)
fpr, tpr, thresholds = roc_curve(y_test_clf, y_pred_proba_rf)
roc_auc = auc(fpr, tpr)
# 或者直接计算 AUC: roc_auc = roc_auc_score(y_test_clf, y_pred_proba_rf)

print(f"AUC: {roc_auc:.4f}")

# 绘制 ROC 曲线
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (area =
```

```
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Random Guess')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.grid(alpha=0.3)
plt.show()
```

6.1.10 精确率-召回率曲线 (Precision-Recall Curve)

- **背景:** 当数据**极度不平衡**时（例如，正类样本非常少），ROC 曲线可能显得过于乐观，因为 FPR 的分母 (FP + TN) 主要由大量的 TN 主导。
- **PR 曲线:** 以**召回率 (Recall)** 为横坐标，**精确率 (Precision)** 为纵坐标，绘制不同阈值下的点连接成的曲线。
- **关注点:** 更关注模型在正类上的表现。曲线越靠近右上角 (Precision=1, Recall=1)，模型性能越好。
- **AP (Average Precision):** PR 曲线下的面积（计算方式略有不同，是 Precision 的加权平均）。

```
from sklearn.metrics import precision_recall_curve, average_precision_score

precision, recall, thresholds_pr = precision_recall_curve(y_test_clf,
y_pred_proba_rf)
ap = average_precision_score(y_test_clf, y_pred_proba_rf)

print(f"Average Precision (AP): {ap:.4f}")

# 绘制 PR 曲线
plt.figure(figsize=(8, 6))
plt.plot(recall, precision, color='blue', lw=2, label=f'PR curve (AP = {ap:.2f})')
# 添加随机猜测线（对于 PR 曲线，基线是正类样本的比例）
baseline = np.sum(y_test_clf == 1) / len(y_test_clf)
plt.plot([0, 1], [baseline, baseline], color='red', lw=2, linestyle='--',
label=f'Random Guess ({baseline:.2f})')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc="lower left")
plt.grid(alpha=0.3)
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
plt.show()
```

6.1.11 如何选择指标?

- **类别平衡:** 如果类别大致平衡，准确率是一个不错的起点，但结合 F1、AUC 通常更好。
- **类别不平衡:**
  - **关注正类（少数类）:** 优先看 Precision, Recall, F1-Score, PR 曲线, AUC。

    - **关注负类（多数类）：**优先看 Specificity。
- **不同错误的代价：**
    - **FP 代价高 (误报代价高):** 关注 Precision, FAR。
    - **FN 代价高 (漏报代价高):** 关注 Recall (POD)。
- **综合评估:** CSI, F1-Score, AUC 提供了更全面的视角。
- **气象领域:** POD, FAR, CSI 是非常关键的指标。

示例代码中的选择: `xgboost1.py` 和 `ensemble_learning` 系列脚本中广泛使用了混淆矩阵、Accuracy、POD、FAR、CSI，这符合气象预报问题的评估需求。AUC 也在一些地方被提及或计算，用于评估模型整体的区分能力。

6.2 交叉验证 (Cross-Validation)

简单的训练/测试集划分可能受到数据划分随机性的影响。某次划分得到的测试集可能特别“简单”或特别“难”，导致评估结果不稳定。**交叉验证**是一种更可靠的模型评估方法。

6.2.1 K 折交叉验证 (K-Fold Cross-Validation)

1. 将训练数据集随机划分为 K 个大小相似的互斥子集（称为“折”，Fold）。
2. 进行 K 轮迭代：
    - 在第 i 轮，将第 i 折作为**验证集 (Validation Set)**，其余 K-1 折作为**训练集 (Training Set)**。
    - 在训练集上训练模型。
    - 在验证集上评估模型，得到一个性能指标（如 Accuracy, AUC）。
3. 将 K 轮得到的性能指标进行平均（有时也看标准差），得到最终的交叉验证评估结果。

优点:

- 所有数据都被用于训练和验证。
- 评估结果更稳定、更可靠，减少了单次划分的随机性。

6.2.2 分层 K 折交叉验证 (Stratified K-Fold)

对于分类问题，特别是类别不平衡时，简单的 K 折划分可能导致某些折中某个类别的样本比例与整体差异很大。**分层 K 折**在划分时会保持每个折中类别比例与原始数据集大致相同。**对于分类问题，通常推荐使用分层 K 折。**

6.2.3 在 Scikit-learn 中使用交叉验证

Scikit-learn 提供了便捷的交叉验证工具。

```
from sklearn.model_selection import cross_val_score, StratifiedKFold

# 使用之前的随机森林分类器 rf_clf 和完整训练数据 X_train_clf, y_train_clf
# 注意：交叉验证是在训练集内部进行的，测试集仍然保留用于最终评估

# 定义分层 K 折交叉验证器
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42) # 通常 5 或 10 折

# 使用 cross_val_score 计算交叉验证得分（例如，AUC）
# scoring 参数指定评估指标，'roc_auc' 是常用的选择
# n_jobs=-1 使用所有 CPU 核心并行计算
```

```
print("\nPerforming 5-Fold Stratified Cross-Validation...")
auc_scores = cross_val_score(rf_clf, X_train_clf, y_train_clf, cv=cv,
scoring='roc_auc', n_jobs=-1)

print(f"AUC scores for each fold: {auc_scores}")
print(f"Mean AUC: {np.mean(auc_scores):.4f}")
print(f"Standard Deviation of AUC: {np.std(auc_scores):.4f}")

# 也可以获取其他指标，例如 accuracy
# accuracy_scores = cross_val_score(rf_clf, X_train_clf, y_train_clf, cv=cv,
scoring='accuracy', n_jobs=-1)
# print(f"\nMean Accuracy: {np.mean(accuracy_scores):.4f}")
```

注意: 如果使用了 `Pipeline`（包含预处理步骤），可以直接对整个 Pipeline 进行交叉验证，确保预处理步骤在每一折内部正确执行。

6.3 模型选择

交叉验证不仅可以用于评估单个模型的性能，还可以用于比较不同模型或同一模型的不同超参数设置。

- **比较不同算法:** 对逻辑回归、SVM、随机森林等不同算法进行交叉验证，选择平均性能指标（如 AUC 或 F1）最高的算法。
- **超参数调优:** 对同一算法的不同超参数组合（例如，随机森林的 `n_estimators`, `max_depth`）进行交叉验证，选择性能最佳的超参数组合。这通常通过**网格搜索 (Grid Search)** 或**随机搜索 (Randomized Search)** 实现，我们将在下一章讨论。

在选择最终模型时，除了交叉验证得分，还需要考虑：

- **模型复杂度:** 更简单的模型通常更容易解释和部署。
- **训练/预测时间:** 对于实时性要求高的应用，模型速度很重要。
- **可解释性:** 某些场景下（如金融风控、医疗），模型的可解释性至关重要。决策树、逻辑回归通常比 SVM（带核）、深度学习模型更易解释。

第七章：超参数调优

在选择了模型算法后，我们还需要确定模型的**超参数 (Hyperparameters)**。超参数是在模型训练开始**之前**设置的参数，它们控制着学习过程本身，而不是通过训练数据学习到的参数（如逻辑回归的系数 `w` 或 SVM 的支持向量）。

例如，对于随机森林：

- `n_estimators` (树的数量)
- `max_depth` (树的最大深度)
- `min_samples_split` (节点分裂的最小样本数)
- `max_features` (每次分裂考虑的特征数)

这些都是超参数。选择合适的超参数对模型的性能至关重要。手动尝试不同的组合既耗时又低效。**超参数调优**（或超参数优化）就是自动寻找最佳超参数组合的过程。

7.1 常见的超参数调优方法

7.1.1 网格搜索 (Grid Search)

- **思想:** 定义一个包含多个超参数候选值的“网格”。网格搜索会尝试网格中所有可能的超参数组合。
- **过程:**
    1. 为每个要调优的超参数指定一组候选值。
    2. 对所有超参数组合，使用交叉验证（通常是 K 折）评估模型性能。
    3. 选择在交叉验证中平均性能最好的那组超参数。
- **优点:** 简单，能保证找到指定网格内的最佳组合。
- **缺点:** 计算成本高，尤其是当超参数数量多或候选值范围大时，组合数量会呈指数级增长。

7.1.2 随机搜索 (Randomized Search)

- **思想:** 不尝试所有组合，而是在指定的超参数分布（或列表）中随机采样固定数量的组合。
- **过程:**
    1. 为每个要调优的超参数指定一个分布（例如，均匀分布、对数均匀分布）或一个候选值列表。
    2. 指定要尝试的组合数量 (`n_iter`)。
    3. 随机采样 `n_iter` 组超参数组合。
    4. 对每组采样到的组合，使用交叉验证评估模型性能。
    5. 选择在交叉验证中平均性能最好的那组超参数。
- **优点:**
    - 计算效率通常比网格搜索高得多，可以在有限的时间内探索更广阔的超参数空间。
    - 研究表明，随机搜索往往能以更少的迭代次数找到与网格搜索相当甚至更好的结果，因为并非所有超参数都同等重要。
- **缺点:** 不保证找到全局最优解，结果具有一定的随机性。

7.1.3 贝叶斯优化 (Bayesian Optimization)

- **思想:** 将超参数调优视为一个优化问题：找到使模型性能指标（如 AUC）最大化的超参数组合。贝叶斯优化使用概率模型（通常是高斯过程）来建模性能指标与超参数之间的关系，并利用过去的评估结果来智能地选择下一个要尝试的超参数组合。
- **过程:**
    1. 定义超参数的搜索空间。
    2. 选择一个概率代理模型（如高斯过程）来估计目标函数（模型性能）。
    3. 选择一个采集函数（Acquisition Function，如 Expected Improvement）来决定下一个最有希望提升性能的采样点。
    4. 迭代进行：
        - 根据采集函数选择下一个超参数组合。
        - 使用该组合训练并评估模型。
        - 使用新的评估结果更新概率代理模型。
- **优点:**
    - 通常比网格搜索和随机搜索更高效（需要评估的组合次数更少），尤其是在评估单个组合成本很高时（例如，训练大型深度学习模型）。
    - 能够更好地处理连续型超参数。
- **缺点:**
    - 实现相对复杂。
    - 对初始采样点和概率模型的选择可能比较敏感。
- **常用库:** Optuna, Hyperopt, Scikit-optimize (skopt)。**本教程的示例代码 `xgboost_optuna.py` 使用了 Optuna 进行 XGBoost 的超参数优化。**

7.2 使用 Scikit-learn 进行网格搜索和随机搜索

Scikit-learn 的 `sklearn.model_selection` 模块提供了 `GridSearchCV` 和 `RandomizedSearchCV` 类。

```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import make_scorer, roc_auc_score
from scipy.stats import randint, uniform # 用于随机搜索的分布

# 使用第五章的示例数据和模型
# X_train_clf, y_train_clf
# X_test_clf, y_test_clf

# 定义要调优的模型
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

# 定义超参数网格 (Grid Search)
param_grid = {
    'n_estimators': [50, 100, 150], # 树的数量
    'max_depth': [10, 15, 20, None], # 最大深度 (None 表示不限制)
    'min_samples_split': [2, 5, 10], # 最小分裂样本数
    'min_samples_leaf': [1, 3, 5],   # 最小叶子节点样本数
    'max_features': ['sqrt', 'log2'] # 考虑的特征数
}
# 总组合数 = 3 * 4 * 3 * 3 * 2 = 216

# 定义超参数分布 (Randomized Search)
param_dist = {
    'n_estimators': randint(50, 201), # 50 到 200 之间的随机整数
    'max_depth': [10, 15, 20, 25, 30, None], # 列表或分布
    'min_samples_split': randint(2, 11), # 2 到 10 之间的随机整数
    'min_samples_leaf': randint(1, 6),   # 1 到 5 之间的随机整数
    'max_features': ['sqrt', 'log2'],
    'bootstrap': [True, False] # 是否使用 bootstrap
}

# 定义交叉验证策略 (与第六章相同)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# 定义评估指标 (例如 AUC)
scoring = 'roc_auc' # 可以是 'accuracy', 'f1', 'precision', 'recall' 等

# --- 7.2.1 网格搜索 (GridSearchCV) ---
print("--- Starting GridSearchCV ---")
# refit=True (默认) 表示在找到最佳参数后，用最佳参数在整个训练集上重新训练模型
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid,
                           scoring=scoring, cv=cv, n_jobs=-1, verbose=1)
# verbose 控制输出信息的详细程度

# 执行搜索 (这可能需要较长时间)
# grid_search.fit(X_train_clf, y_train_clf)
```

```
# # 输出最佳参数和最佳得分
# print("\nBest parameters found by Grid Search:")
# print(grid_search.best_params_)
# print(f"Best {scoring} score found by Grid Search:
{grid_search.best_score_:.4f}")

# # 获取最佳模型
# best_rf_grid = grid_search.best_estimator_

# # 在测试集上评估最佳模型
# y_pred_best_grid = best_rf_grid.predict(X_test_clf)
# y_pred_proba_best_grid = best_rf_grid.predict_proba(X_test_clf)[:, 1]
# print("\nPerformance of Best Model from Grid Search on Test Set:")
# print(f"Accuracy: {accuracy_score(y_test_clf, y_pred_best_grid):.4f}")
# print(f"AUC: {roc_auc_score(y_test_clf, y_pred_proba_best_grid):.4f}")
# print(classification_report(y_test_clf, y_pred_best_grid))

# --- 7.2.2 随机搜索 (RandomizedSearchCV) ---
print("\n--- Starting RandomizedSearchCV ---")
# n_iter 控制随机采样的组合数量
n_iterations = 50 # 尝试 50 种随机组合
random_search = RandomizedSearchCV(estimator=rf, param_distributions=param_dist,
                                   n_iter=n_iterations, scoring=scoring, cv=cv,
                                   n_jobs=-1, random_state=42, verbose=1)

# 执行搜索
random_search.fit(X_train_clf, y_train_clf)

# 输出最佳参数和最佳得分
print("\nBest parameters found by Randomized Search:")
print(random_search.best_params_)
print(f"Best {scoring} score found by Randomized Search:
{random_search.best_score_:.4f}")

# 获取最佳模型
best_rf_random = random_search.best_estimator_

# 在测试集上评估最佳模型
y_pred_best_random = best_rf_random.predict(X_test_clf)
y_pred_proba_best_random = best_rf_random.predict_proba(X_test_clf)[:, 1]
print("\nPerformance of Best Model from Randomized Search on Test Set:")
print(f"Accuracy: {accuracy_score(y_test_clf, y_pred_best_random):.4f}")
print(f"AUC: {roc_auc_score(y_test_clf, y_pred_proba_best_random):.4f}")
# print(classification_report(y_test_clf, y_pred_best_random))

# 可以查看所有尝试过的组合及其结果
# results_df = pd.DataFrame(random_search.cv_results_)
# print("\nRandom Search Results Summary:\n",
results_df.sort_values(by='rank_test_score').head())
```

7.3 使用 Optuna 进行贝叶斯优化

Optuna 是一个专门为机器学习超参数优化设计的现代 Python 库。它支持多种优化算法（包括基于树的 Parzen 估计器 TPE，类似于 Hyperopt；以及基于高斯过程的算法），并提供了简洁易用的 API。

核心概念:

- **Study:** 一个优化任务，包含一系列的试验 (Trial)。目标是找到使目标函数最优的试验。
- **Trial:** 对目标函数的一次评估，对应一组特定的超参数。
- **Objective Function:** 一个接受 `trial` 对象作为参数的函数。在此函数内部：
  - 使用 `trial.suggest_float()`, `trial.suggest_int()`, `trial.suggest_categorical()` 等方法定义并获取当前试验要尝试的超参数值。
  - 使用这些超参数训练模型。
  - 评估模型性能（例如，通过交叉验证得到 AUC）。
  - 返回要**最小化**或**最大化**的性能指标值。

Optuna 的优势:

- **高效的采样算法:** 内置多种先进的优化算法。
- **剪枝 (Pruning):** 可以提前停止没有希望的试验，节省计算资源。
- **易于并行化:** 可以轻松地在多个进程或机器上并行运行试验。
- **良好的可视化:** 可以方便地绘制优化历史、参数重要性等图表。
- **与框架无关:** 可以用于优化任何机器学习库（Scikit-learn, XGBoost, LightGBM, PyTorch, TensorFlow 等）的超参数。

示例:

请参考项目中的 `xgboost_optuna.py` 文件，它展示了如何使用 Optuna 来优化 XGBoost 模型的超参数。其基本结构如下：

```
# xgboost_optuna.py (简化示例)
import optuna
import xgboost as xgb
from sklearn.model_selection import cross_val_score, StratifiedKFold
# ... 加载数据 X_train, y_train ...

def objective(trial):
    # 1. 定义超参数搜索空间
    param = {
        'objective': 'binary:logistic',
        'eval_metric': 'auc',
        'tree_method': 'hist',
        'lambda': trial.suggest_float('lambda', 1e-8, 1.0, log=True), # L2 正则化
        'alpha': trial.suggest_float('alpha', 1e-8, 1.0, log=True),   # L1 正则化
        'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
        'subsample': trial.suggest_float('subsample', 0.5, 1.0),
        'learning_rate': trial.suggest_float('learning_rate', 1e-4, 0.1,
log=True),
        'n_estimators': trial.suggest_int('n_estimators', 100, 1000),
        'max_depth': trial.suggest_int('max_depth', 3, 9),
        'min_child_weight': trial.suggest_int('min_child_weight', 1, 10),
        'scale_pos_weight': scale_pos_weight # 假设已计算好
        # ... 其他参数 ...
```

```
    }

    # 2. 训练和评估模型（使用交叉验证）
    model = xgb.XGBClassifier(**param, random_state=42, use_label_encoder=False)
    cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
    scores = cross_val_score(model, X_train, y_train, cv=cv, scoring='roc_auc',
n_jobs=-1)
    auc_mean = np.mean(scores)

    # 3. 返回要优化的指标（Optuna 默认最小化，如果最大化 AUC，可以返回 -auc_mean 或在
study 中指定 direction）
    return auc_mean # Optuna 会自动处理最大化

# 创建 Study 对象
# storage='sqlite:///db.sqlite3' 可以将结果保存到数据库，方便中断和恢复
# pruner=optuna.pruners.MedianPruner() 可以启用剪枝
study = optuna.create_study(direction='maximize',
study_name='xgboost_optimization')

# 运行优化
n_trials = 100 # 尝试 100 次试验
study.optimize(objective, n_trials=n_trials)

# 输出最佳结果
print("Best trial:")
trial = study.best_trial
print(f"  Value (AUC): {trial.value:.4f}")
print("  Params: ")
for key, value in trial.params.items():
    print(f"    {key}: {value}")

# Optuna 还提供了可视化功能
# optuna.visualization.plot_optimization_history(study).show()
# optuna.visualization.plot_param_importances(study).show()
```

对于复杂的模型和较大的数据集，使用 Optuna 等贝叶斯优化工具通常是比网格搜索和随机搜索更有效的选择。

（教程的第七部分结束。请告诉我"继续写"，我会接着编写第八章：集成学习与 XGBoost 详解。）