****

**Python机器学习上机实验报告**

Machine Learning (Python-Based) Hands-on Lab Report

学 院： 商学院

班 级： 信管T2201

姓 名： 方珺逸

学 号： 8202221230

指导教师： 储军飞

2025年1月

**目录**

[一、梯度下降与线性回归 4](#_Toc24172)

[1.1 编写梯度下降算法（Momentum + RMSprop） 4](#_Toc29543)

[1.2 多元线性回归任务 7](#_Toc17172)

[1.3 蒙特卡洛模拟 10](#_Toc26977)

[1.4 调用 Sklearn 库完成多元线性回归任务 13](#_Toc13037)

[二、逻辑回归 16](#_Toc27551)

[2.1 编写针对逻辑回归的梯度下降算法 16](#_Toc24949)

[2.2 Logistic regression 算法实现对 iris 数据的分类 19](#_Toc16484)

[三、K近邻算法（KNN） 24](#_Toc16818)

[3.1 KNN算法 24](#_Toc1315)

[3.2使用KNN完成对 iris 数据集的分类 28](#_Toc16306)

[四、神经网络 30](#_Toc12430)

[4.1 BP神经网络 30](#_Toc11982)

[4.2反向传播算法实现及动画化\* 32](#_Toc27206)

[五、聚类 37](#_Toc23333)

[5.1 聚类算法 37](#_Toc3303)

[5.2 基于kmeans.txt 数据集的聚类对比 40](#_Toc24565)

[5.3 两种聚类算法的对比 50](#_Toc3453)

[六、回归算法模型评估和调参 53](#_Toc12732)

[6.1 手动调参 53](#_Toc7084)

[6.2 自动调参\* 65](#_Toc574)

[七、分类算法模型评估和调参 69](#_Toc10259)

[八、聚类任务综合 75](#_Toc5607)

[九、半监督学习 80](#_Toc10917)

[9.1 半监督学习算法 80](#_Toc7463)

[9.2 结合聚类改进处理的半监督学习 84](#_Toc4381)

[十、集成学习 89](#_Toc31157)

[十一、房价预测 94](#_Toc15795)

[11.1仅考虑线性回归 94](#_Toc22234)

[11.2基于梯度提升决策树的预测 97](#_Toc177)

[11.3 转换为分类问题 101](#_Toc30010)

[十二、医疗诊断 107](#_Toc2985)

[12.1 最佳模型分类性能优化 107](#_Toc19887)

[12.2 使用复杂数据同时完成分类和回归任务 113](#_Toc4947)

[① 数据分析与问题诊断 113](#_Toc3123)

[② 回归 118](#_Toc15708)

[③ 分类 123](#_Toc10993)

[12.3 使用GPU训练的神经网络分类器\*[[1]](#footnote-0) 127](#_Toc8624)

**零、引言**

1. 实验目的

掌握常见的机器学习算法的python实现，同时复习本学期的ML课程。

1. 实验准备

了解常见的机器学习算法，包括线性回归、逻辑回归、K近邻、神经网络等。掌握Python编程及相关库（如Pandas、NumPy、Sklearn等）的使用。

1. 实验环境

**操作系统**：Windows 11

**硬件:**

* CPU：12th Gen Intel(R) Core(TM) i5-12500H 2.50 GHz
* RAM：16.0 GB
* GPU：NVIDIA GPU GEFORCE RTX3060

**软件**：

* Python 3.11
* PyCharm2024.3 Prof.版

1. 实验内容

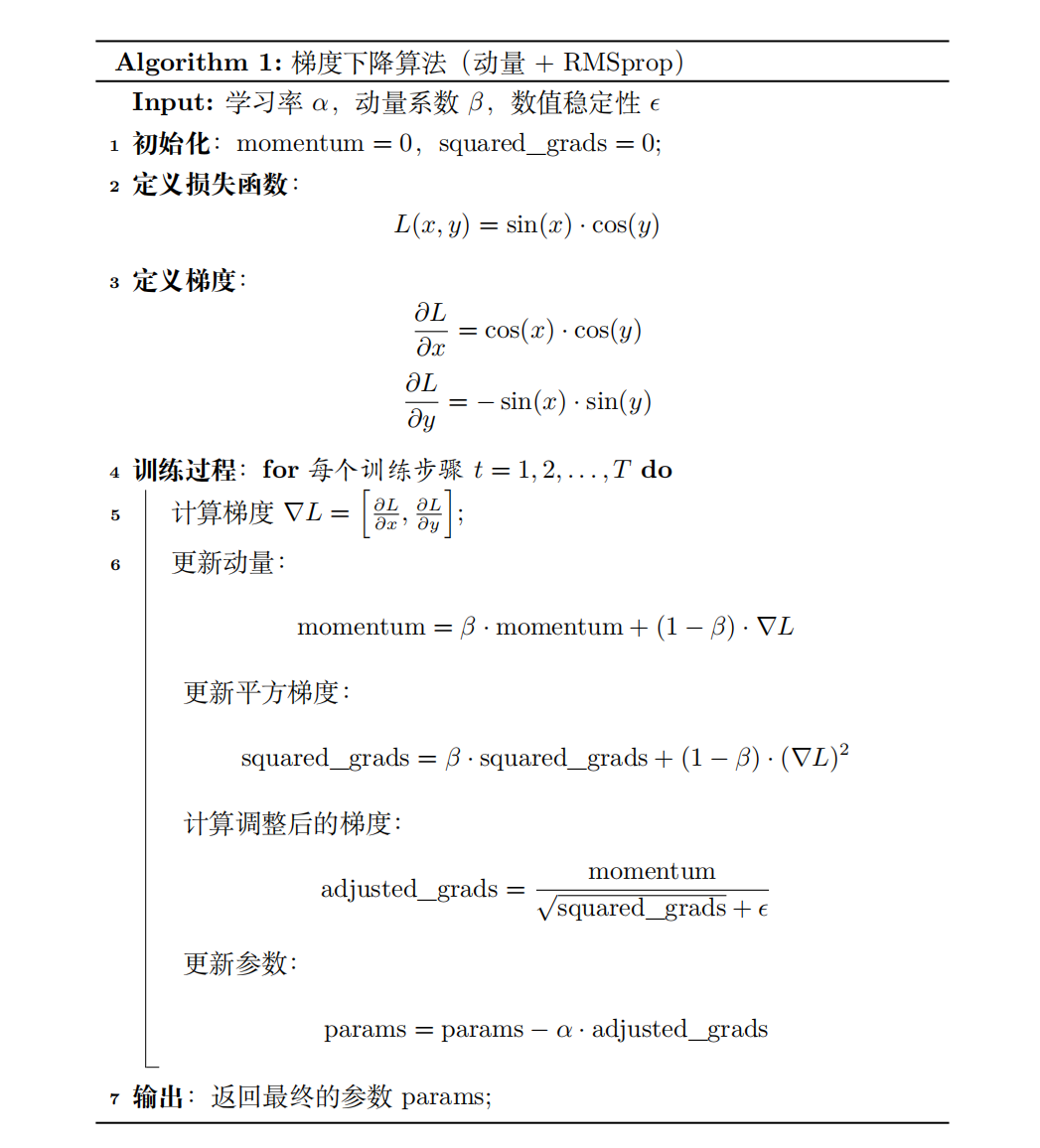
见机器学习上机实验手册-V2024。

**一、梯度下降与线性回归**

**1.1 编写梯度下降算法（Momentum + RMSprop）**

**目标**：实现一个梯度下降优化器类，结合Momentum和RMSprop更新方法来调整参数。

**伪代码：**

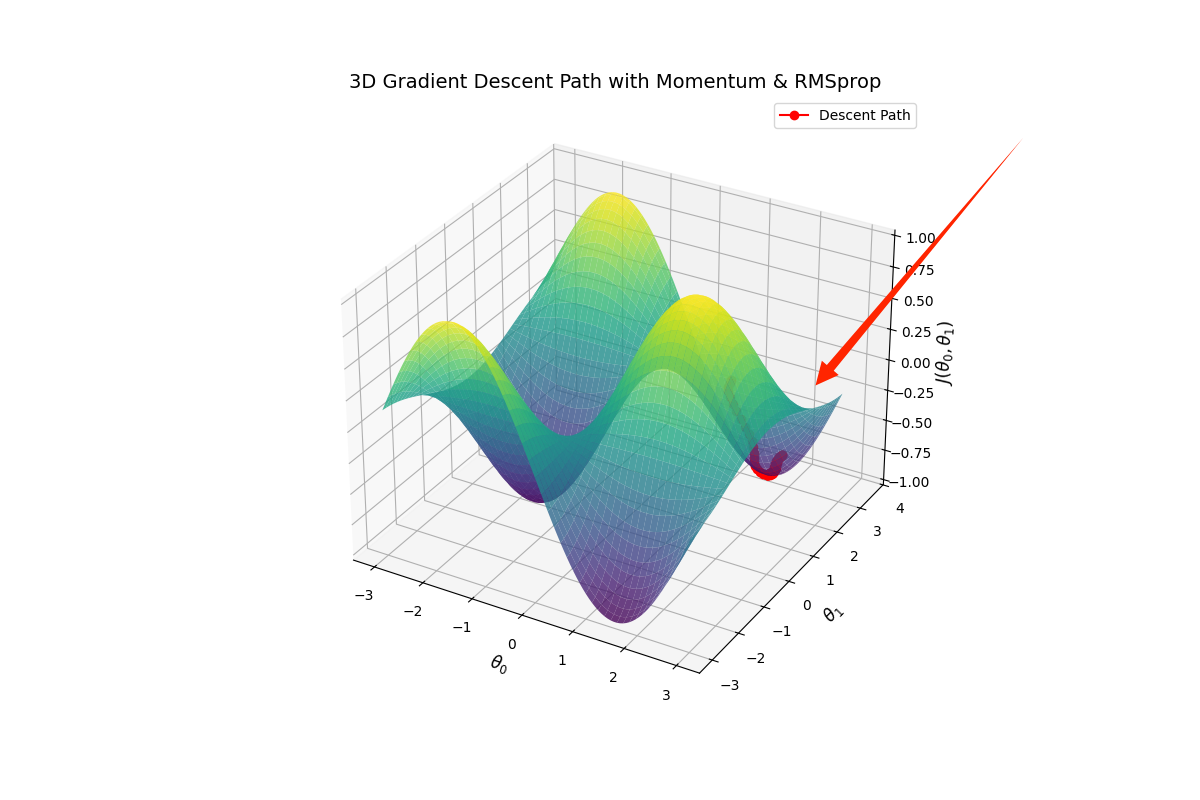


**图1.1 梯度下降算法**

**Python实现：**

import numpy as np  
import matplotlib.pyplot as plt  
class Optimizer:  
 def \_\_init\_\_(self, learning\_rate, beta, epsilon):  
 self.learning\_rate=learning\_rate  
 self.beta = beta  
 self.epsilon = epsilon  
 self.momentum = 0  
 self.squared\_grads = 0  
 def update(self, params, grads):  
 self.momentum = self.beta \* self.momentum + (1 - self.beta) \* grads  
 self.squared\_grads = self.beta \* self.squared\_grads + (1 - self.beta) \* (grads \*\* 2)  
 adjusted\_grads = self.momentum / (np.sqrt(self.squared\_grads) + self.epsilon)  
 params -= self.learning\_rate \* adjusted\_grads  
 return params  
def loss\_function(x, y):  
 return np.sin(x) \* np.cos(y)  
def loss\_gradients(x, y):  
 grad\_x = np.cos(x) \* np.cos(y) *# 对 x 的偏导数* grad\_y = -np.sin(x) \* np.sin(y) *# 对 y 的偏导数* return np.array([grad\_x, grad\_y])  
*# 测试优化器并绘制 3D 路径图*def test\_optimizer\_3d():  
 *# 创建优化器，使用 Momentum 和 RMSprop* optimizer = Optimizer(learning\_rate=0.1, beta=0.9, epsilon=1e-8)  
 *# 初始化参数，选择一个更合适的初始点* params = np.array([1.5, 1.5]) *# 更好的起始点* param\_history = [params.copy()] *# 记录每一步的参数* num\_iterations = 50  
 for \_ in range(num\_iterations):  
 grads = loss\_gradients(params[0], params[1]) *# 计算梯度* params = optimizer.update(params, grads) *# 更新参数* param\_history.append(params.copy()) *# 保存历史路径* param\_history = np.array(param\_history) *# 转换为 NumPy 数组* fig = plt.figure(figsize=(12, 8))  
 ax = fig.add\_subplot(111, projection='3d')x = np.linspace(-3, 3, 100)  
 y = np.linspace(-3, 3, 100)  
 X, Y = np.meshgrid(x, y)  
 Z = loss\_function(X, Y)ax.plot\_surface(X, Y, Z, cmap='viridis', alpha=0.8)  
  
 *# 绘制下降路径* path\_x = param\_history[:, 0]  
 path\_y = param\_history[:, 1]  
 path\_z = loss\_function(path\_x, path\_y)  
 ax.plot(path\_x, path\_y, path\_z, color='red', marker='o', label="Descent Path")ax.set\_xlabel(r'$\theta\_0$', fontsize=12)  
 ax.set\_ylabel(r'$\theta\_1$', fontsize=12)  
 ax.set\_zlabel(r'$J(\theta\_0, \theta\_1)$', fontsize=12)  
 ax.set\_title("3D Gradient Descent Path with Momentum & RMSprop", fontsize=14)  
 ax.legend()  
 plt.show()test\_optimizer\_3d()

**输出与结果展示：**



**图1.2 梯度下降可视化**

**反思与感悟：**结合Momentum和RMSprop的优化器能有效加速收敛，尤其在复杂高维问题中。

**1.2 多元线性回归任务**

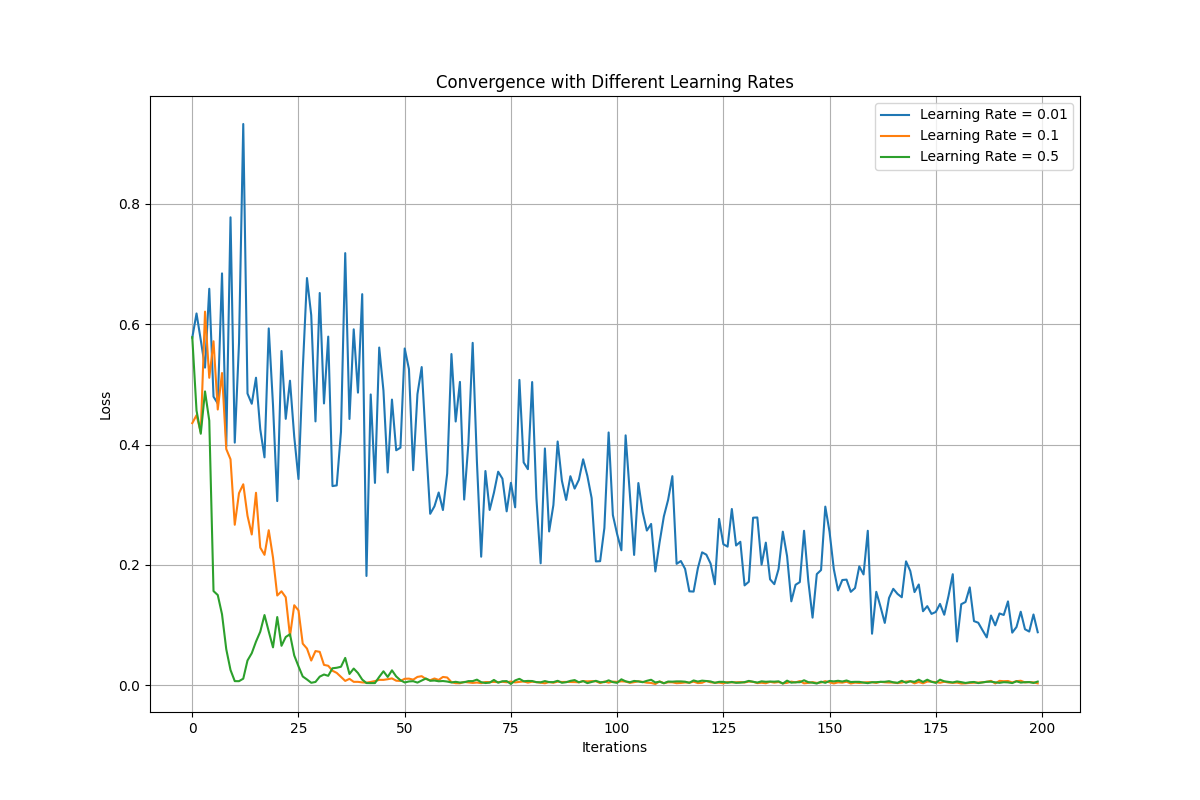
**目标：**实现多元线性回归的损失函数和梯度计算，并结合自定义优化器（Momentum和RMSprop）进行训练，并讨论使用不同学习率对于算法收敛情况的影响。

**思路：**首先定义线性回归的损失函数（均方误差，MSE）和对应的梯度（针对每个参数）。通过梯度下降方法更新权重。每次训练时，选择一个**小批量**数据计算损失和梯度，使用优化器更新权重。

**Python实现：**

*# -\*- coding: utf-8 -\*-*import numpy as np  
import matplotlib.pyplot as plt  
*# 多元线性回归损失函数（MSE）*def linear\_regression\_loss(X, y, w):  
 m = len(y)  
 predictions = X.dot(w)  
 loss = (1 / (2 \* m)) \* np.sum((predictions - y) \*\* 2)  
 return loss  
def linear\_regression\_gradients(X, y, w):  
 m = len(y)  
 predictions = X.dot(w)  
 gradients = (1 / m) \* X.T.dot(predictions - y)  
 return gradients  
*# 优化器类（上一题已经写出）*class Optimizer:  
 def \_\_init\_\_(self, learning\_rate, beta, epsilon):  
 self.learning\_rate = learning\_rate  
 self.beta = beta  
 self.epsilon = epsilon  
 self.momentum = 0  
 self.squared\_grads = 0  
 def update(self, params, grads):  
 self.momentum = self.beta \* self.momentum + (1 - self.beta) \* grads  
 self.squared\_grads = self.beta \* self.squared\_grads + (1 - self.beta) \* (grads \*\* 2)  
 adjusted\_grads = self.momentum / (np.sqrt(self.squared\_grads) + self.epsilon)  
 params -= self.learning\_rate \* adjusted\_grads  
 return params  
def generate\_data(n\_samples=100, n\_features=2, noise=0.1):  
 *# 随机生成样本数量为n\_samples，特征数量为n\_features的输入数据* X = np.random.rand(n\_samples, n\_features)  
 true\_weights = np.array([3.0, -2.0]) *# 真实的参数* y = X.dot(true\_weights) + noise \* np.random.randn(n\_samples) *# 加入噪声* return X, y  
  
def train\_linear\_regression(X, y, optimizer, num\_iterations=200, batch\_size=32):  
 w = np.zeros(X.shape[1])  
 loss\_history = []  
 for i in range(num\_iterations):  
 *# 随机选取batch!* indices = np.random.choice(len(X), batch\_size)  
 X\_batch = X[indices]  
 y\_batch = y[indices]  
 grads = linear\_regression\_gradients(X\_batch, y\_batch, w)  
 *# 更新参数* w = optimizer.update(w, grads)  
 *# 计算当前的损失* loss = linear\_regression\_loss(X\_batch, y\_batch, w)  
 loss\_history.append(loss)  
  
 return w, loss\_history  
*# 画出不同学习率下的收敛情况*def plot\_convergence(X, y):  
 learning\_rates = [0.01, 0.1, 0.5] *# 三种不同的学习率* plt.figure(figsize=(12, 8))  
  
 for lr in learning\_rates:  
 optimizer = Optimizer(learning\_rate=lr, beta=0.9, epsilon=1e-8)  
 w, loss\_history = train\_linear\_regression(X, y, optimizer, num\_iterations=200)  
 *# 绘制损失曲线* plt.plot(loss\_history, label=f'Learning Rate = {lr}')  
 print(f'Final weights for learning rate {lr}: {w}')  
 plt.xlabel('Iterations')  
 plt.ylabel('Loss')  
 plt.title('Convergence with Different Learning Rates')  
 plt.legend()  
 plt.grid(True)  
 plt.show()  
*# 生成数据*X, y = generate\_data(n\_samples=200, n\_features=2, noise=0.1)  
*# 画出不同学习率下的收敛情况*plot\_convergence(X, y)

**输出与结果展示：**



**图1.3 学习率对收敛的影响**

**反思与感悟：**

从图中可以看出，学习率对收敛速度和稳定性有显著影响。

低学习率（0.01）： 损失下降缓慢，可能需要更多的迭代才能收敛。

中等学习率（0.1）： 损失迅速下降并趋于稳定，表明这是一个不错的选择。

高学习率（0.5）： 损失迅速下降，但可能会在最小值附近波动，甚至造成Loss函数反弹的情况。

**1.3 蒙特卡洛模拟**

**目标：使用统计建模库 statsmodels 进行线性回归，并计算t统计量和F统计量。通过蒙特卡洛模拟评估回归结果的显著性。**

**思路：利用 statsmodels 执行最小二乘回归，计算回归系数、标准误差、t统计量、F统计量，并进行假设检验（模拟p值）。通过蒙特卡洛模拟生成多个样本进行统计推断，最后进行可视化分析。**

**t统计量为：**

**F统计量为：**

**Python实现：**

import numpy as np  
import matplotlib.pyplot as plt  
import statsmodels.api as sm  
np.random.seed(10086)  
n=100  
p=1  
X=np.random.randn(n,p)  
y=2 \* X[:,0] + np.random.randn(n)  
X=sm.add\_constant(X)  
model = sm.OLS(y,X)  
results = model.fit()  
  
betas = results.params  
stderr = results.bse  
t\_stats = betas / stderr  
f\_stat = results.fvalue  
f\_p\_value = results.f\_pvalue  
print("Regression Coefficients:", betas)  
print("Standard Errors:", stderr)  
print("t-statistics:", t\_stats)  
print("F-statistic:", f\_stat)  
print("F-statistic p-value:", f\_p\_value)  
  
*# Monte Carlo Simulation*n\_simulations = 10000  
t\_stats\_sim = np.zeros(n\_simulations)  
f\_stats\_sim = np.zeros(n\_simulations)  
for i in range(n\_simulations):  
 y\_sim = 2 \* X[:, 1] + np.random.randn(n) *# Simulate data* model\_sim = sm.OLS(y\_sim, X)  
 results\_sim = model\_sim.fit()  
  
 *# Record t-statistics and F-statistics for each simulation* t\_stats\_sim[i] = results\_sim.params[1] / results\_sim.bse[1]  
 f\_stats\_sim[i] = results\_sim.fvalue  
  
*# Calculate p-value for t-statistics*t\_p\_value\_sim = np.mean(np.abs(t\_stats\_sim) >= np.abs(t\_stats[1]))  
  
*# Calculate p-value for F-statistics*f\_p\_value\_sim = np.mean(f\_stats\_sim >= f\_stat)  
  
plt.figure(figsize=(14, 7))  
plt.subplot(1, 2, 1)  
plt.hist(t\_stats\_sim, bins=50, color='dodgerblue', alpha=0.7, edgecolor='black', label='Simulated t-statistics')  
plt.axvline(t\_stats[1], color='firebrick', linestyle='--', label=f'Observed t-statistic {t\_stats[1]:.3f}')  
plt.title('t-statistics Distribution', fontsize=14)  
plt.xlabel('t-statistic', fontsize=12)  
plt.ylabel('Frequency', fontsize=12)  
plt.legend(loc='upper left')  
plt.grid(True, linestyle='--', alpha=0.7)  
  
plt.subplot(1, 2, 2)  
plt.hist(f\_stats\_sim, bins=50, color='mediumseagreen', alpha=0.7, edgecolor='black', label='Simulated F-statistics')  
plt.axvline(f\_stat, color='firebrick', linestyle='--', label=f'Observed F-statistic {f\_stat:.3f}')  
plt.title('F-statistics Distribution', fontsize=14)  
plt.xlabel('F-statistic', fontsize=12)  
plt.ylabel('Frequency', fontsize=12)  
plt.legend(loc='upper left')  
plt.grid(True, linestyle='--', alpha=0.7)  
plt.tight\_layout()  
plt.show()  
print(f"Monte Carlo simulated p-value for t-statistic: {t\_p\_value\_sim:.4f}")  
print(f"Monte Carlo simulated p-value for F-statistic: {f\_p\_value\_sim:.4f}")

**输出与结果展示：**

Regression Coefficients: [-0.02678519 2.04569005]

Standard Errors: [0.09509515 0.09074614]

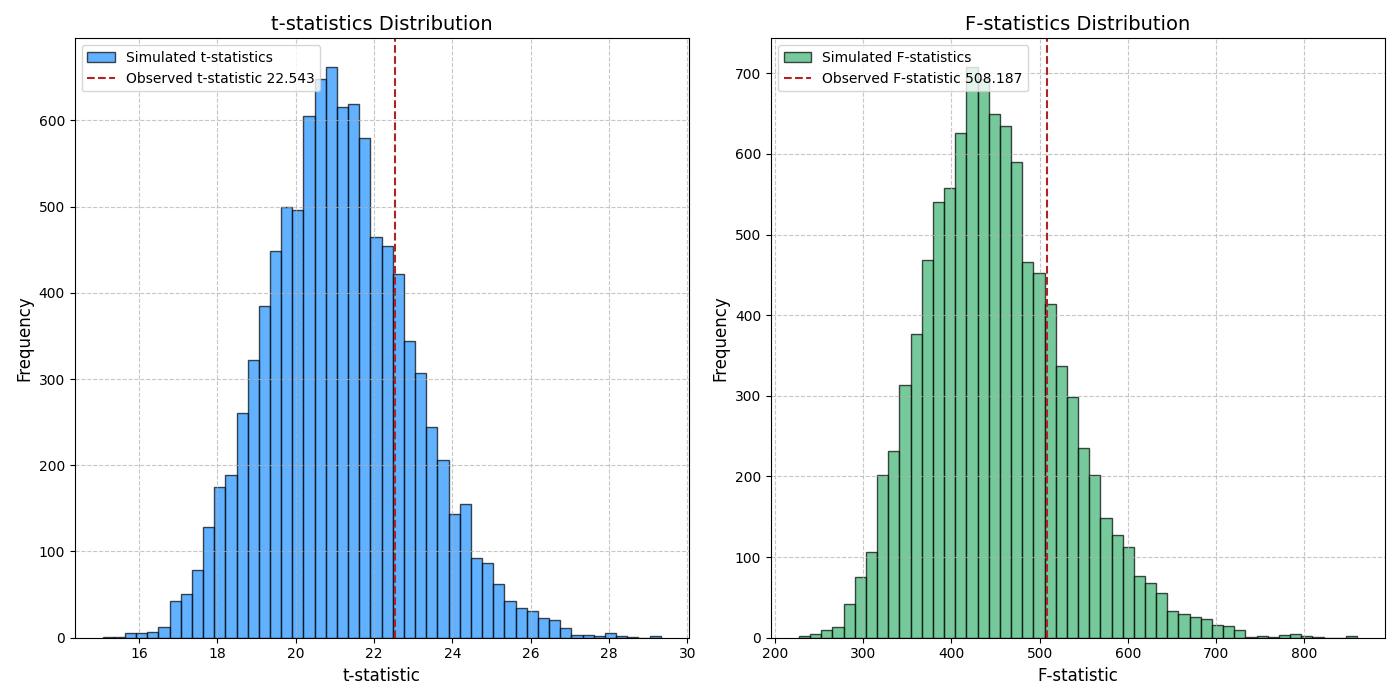
t-statistics: [-0.2816673 22.5429991]

F-statistic: 508.18680822902365

F-statistic p-value: 1.4623025435259942e-40

Monte Carlo simulated p-value for t-statistic: 0.2151

Monte Carlo simulated p-value for F-statistic: 0.2151



**图1.4 蒙特卡洛模拟**

**反思与感悟：**蒙特卡洛模拟为假设检验提供了强有力的工具，表示“碰运气”试出来的概率，这能反映建立的回归模型是否稳健。

**1.4 调用 Sklearn 库完成多元线性回归任务**

****目标：**使用 sklearn 线性回归和自定义优化器对比，评估不同学习率下的模型收敛速度和性能。比较其与梯度下降算法得到的解的不同之处。**

****数据集：**sklearn.datasets 提供的糖尿病数据集 ，一个内置的[回归分析](https://so.csdn.net/so/search?q=%E5%9B%9E%E5%BD%92%E5%88%86%E6%9E%90&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_52447486/article/details/_blank)数据集。**

****思路：**首先加载并标准化数据，使用自定义的梯度下降方法（结合Momentum和RMSprop）训练线性回归模型，并与 sklearn 的线性回归模型做对比。通过计算均方误差（MSE）评估两者的拟合效果，并绘制损失曲线观察不同学习率下的收敛情况。**

****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_diabetes  
from sklearn.linear\_model import LinearRegression  
from sklearn.metrics import mean\_squared\_error  
from sklearn.preprocessing import StandardScaler  
class Optimizer:  
 def \_\_init\_\_(self, learning\_rate, beta, epsilon):  
 self.learning\_rate = learning\_rate  
 self.beta = beta  
 self.epsilon = epsilon  
 self.momentum = 0  
 self.squared\_grads = 0  
 def update(self, params, grads):  
 self.momentum = self.beta \* self.momentum + (1 - self.beta) \* grads  
 self.squared\_grads = self.beta \* self.squared\_grads + (1 - self.beta) \* (grads \*\* 2)  
 adjusted\_grads = self.momentum / (np.sqrt(self.squared\_grads) + self.epsilon)  
 params -= self.learning\_rate \* adjusted\_grads  
 return params  
*# 多元线性回归损失函数（MSE）*def linear\_regression\_loss(X, y, w):  
 m = len(y)  
 predictions = X.dot(w)  
 loss = (1 / (2 \* m)) \* np.sum((predictions - y) \*\* 2)  
 return loss  
*# 多元线性回归梯度*def linear\_regression\_gradients(X, y, w):  
 m = len(y)  
 predictions = X.dot(w)  
 gradients = (1 / m) \* X.T.dot(predictions - y)  
 return gradients  
def train\_linear\_regression(X, y, optimizer, num\_iterations=200, batch\_size=32):  
 *# 初始化权重* w = np.zeros(X.shape[1])  
 loss\_history = []  
 for i in range(num\_iterations):  
 *# 随机选取batch* indices = np.random.choice(len(X), batch\_size)  
 X\_batch = X[indices]  
 y\_batch = y[indices]  
 grads = linear\_regression\_gradients(X\_batch, y\_batch, w)  
 w = optimizer.update(w, grads)  
 loss = linear\_regression\_loss(X\_batch, y\_batch, w)  
 loss\_history.append(loss)  
 return w, loss\_history  
def load\_data():  
 diabetes = load\_diabetes()  
 X = diabetes.data  
 y = diabetes.target  
 scaler = StandardScaler()  
 X\_scaled = scaler.fit\_transform(X)  
 return X\_scaled, y  
def compare\_solutions(X, y):  
 optimizer = Optimizer(learning\_rate=0.01, beta=0.9, epsilon=1e-8)  
 w\_gd, loss\_history = train\_linear\_regression(X, y, optimizer, num\_iterations=500)  
 model = LinearRegression()  
 model.fit(X, y)  
 w\_sklearn = model.coef\_  
 print("Gradient Descent Weights:", w\_gd)  
 print("Sklearn Weights:", w\_sklearn)  
 y\_pred\_gd = X.dot(w\_gd)  
 y\_pred\_sklearn = model.predict(X)  
 mse\_gd = mean\_squared\_error(y, y\_pred\_gd)  
 mse\_sklearn = mean\_squared\_error(y, y\_pred\_sklearn)  
 print(f"Mean Squared Error (Gradient Descent): {mse\_gd}")  
 print(f"Mean Squared Error (Sklearn): {mse\_sklearn}")  
 return w\_gd, w\_sklearn, mse\_gd, mse\_sklearn

X, y = load\_data()  
w\_gd, w\_sklearn, mse\_gd, mse\_sklearn = compare\_solutions(X, y)

****输出与结果展示：****

Gradient Descent Weights: [ 1.59403611 0.19261283 3.81104365 3.38322878 1.69974145 1.36656961 -3.39396097 3.20274263 3.88314213 2.98760101]

Sklearn Weights: [ -0.47612079 -11.40686692 24.72654886 15.42940413 -37.67995261 22.67616277 4.80613814 8.42203936 35.73444577 3.21667372]

Mean Squared Error (Gradient Descent): 27715.453983206655

Mean Squared Error (Sklearn): 2859.69634758675

****反思与感悟：****

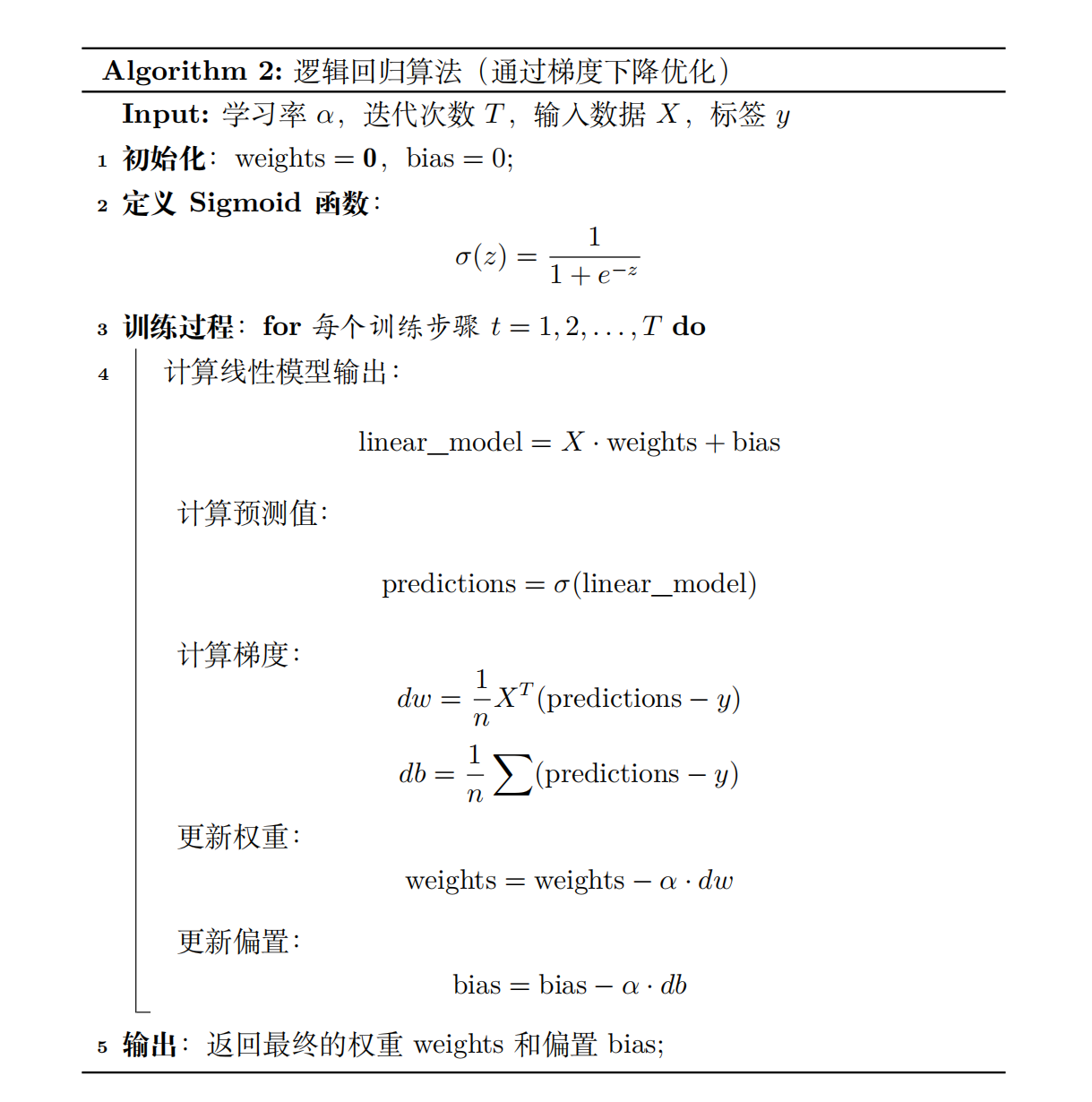
**Sklearn库的实现不仅权重值不同，而且均方误差显著低于自己写的梯度下降算法的实现。这表明Sklearn的模型在测试集上的预测更为准确。因为这个包使用了更高效的优化方法，如正规方程（Normal Equation）、坐标下降（Coordinate Descent），这些方法在处理线性回归问题时通常更为稳定和高效。**

**二、逻辑回归**

**2.1 编写针对逻辑回归的梯度下降算法**

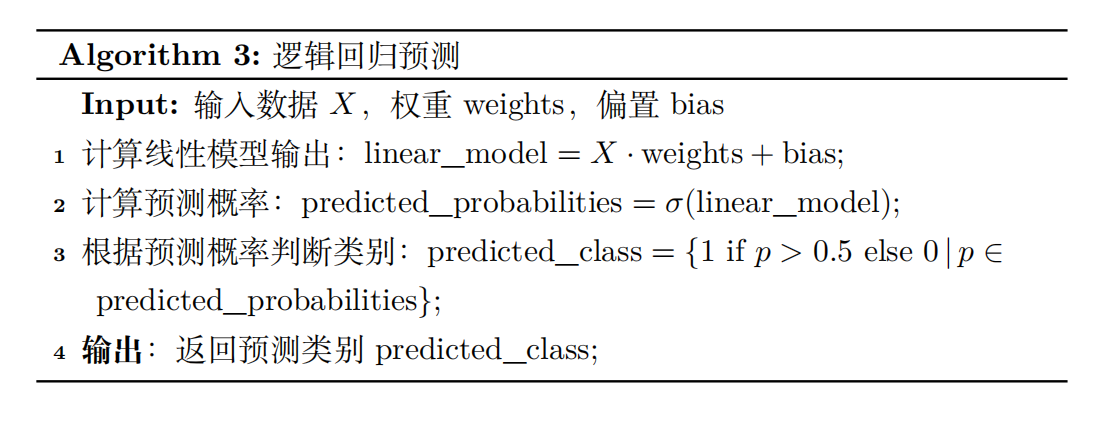
****目标：手搓****一个基于梯度下降的逻辑回归模型，手动推导其参数的更新公式。

****优化的伪代码：****



**图2.1 参数更新**

****预测的伪代码：****



**图2.2 预测**

****Python实现：****

*# -\*- coding: utf-8 -\*-*import numpy as np  
class LogisticRegressionGD:  
 def \_\_init\_\_(self, learning\_rate, num\_iterations):  
 self.learning\_rate = learning\_rate  
 self.num\_iterations = num\_iterations  
 self.weights = None  
 self.bias = 0  
 def sigmoid(self, z):  
 return 1 / (1 + np.exp(-z))  
 def fit(self, X, y):  
 n\_samples, n\_features = X.shape  
 self.weights = np.zeros(n\_features)  
 for \_ in range(self.num\_iterations):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 predictions = self.sigmoid(linear\_model)  
 dw = (1 / n\_samples) \* np.dot(X.T, (predictions - y))  
 db = (1 / n\_samples) \* np.sum(predictions - y)  
 self.weights -= self.learning\_rate \* dw  
 self.bias -= self.learning\_rate \* db  
 def predict(self, X):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 y\_predicted = self.sigmoid(linear\_model)  
 y\_predicted\_cls = [1 if i > 0.5 else 0 for i in y\_predicted]  
 return np.array(y\_predicted\_cls)  
  
def test\_logistic\_regression():  
 from sklearn.datasets import make\_classification  
 from sklearn.model\_selection import train\_test\_split  
 from sklearn.metrics import accuracy\_score  
  
 *# 创建模拟二分类数据集进行测试* X, y = make\_classification(n\_samples=100, n\_features=2, n\_redundant=0, n\_informative=2, random\_state=42, n\_clusters\_per\_class=1)  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
 classifier = LogisticRegressionGD(learning\_rate=0.01, num\_iterations=1000)  
 classifier.fit(X\_train, y\_train)  
 predictions = classifier.predict(X\_test)  
 print(f"Accuracy: {accuracy\_score(y\_test, predictions)}")  
test\_logistic\_regression()

****输出与结果展示：Accuracy: 0.95****

**2.2 Logistic regression 算法实现对 iris 数据的分类**

****目标：**考虑如何使用二分类算法解决多分类问题，**用PCA将四维数据降到二维，绘制决策边界。对比不同的多分类训练策略：**One-vs-All,、One-vs-One,、Error-Correcting Output Codes (ECOC)**的效果

****思路：****需要采取鸢尾花的所有特征。

**数据集：**Iris数据集（安德森鸢尾花卉数据集）。

**One-vs-All**（一对全）的核心思想是将多类问题转化为多个二分类问题。具体做法是对于每一个类别，训练一个分类器来判断该类别与其他类别的区别。每个分类器的目标是区分一个类别与其余类别的样本。

**One-vs-One**（一对一）的核心思想是训练多个分类器，每个分类器只处理两个类别的样本。对于每一对类别，训练一个二分类器来区分这两个类别。

**ECOC**策略为每个类别分配一个二进制向量，类别数减 1 即为编码长度。每个分类器负责预测一个二进制位的值（0 或 1）。最终的类别由所有分类器输出的二进制位拼接而成，使用一定的解码方法将二进制向量转换回实际的类别。

****Python实现：****

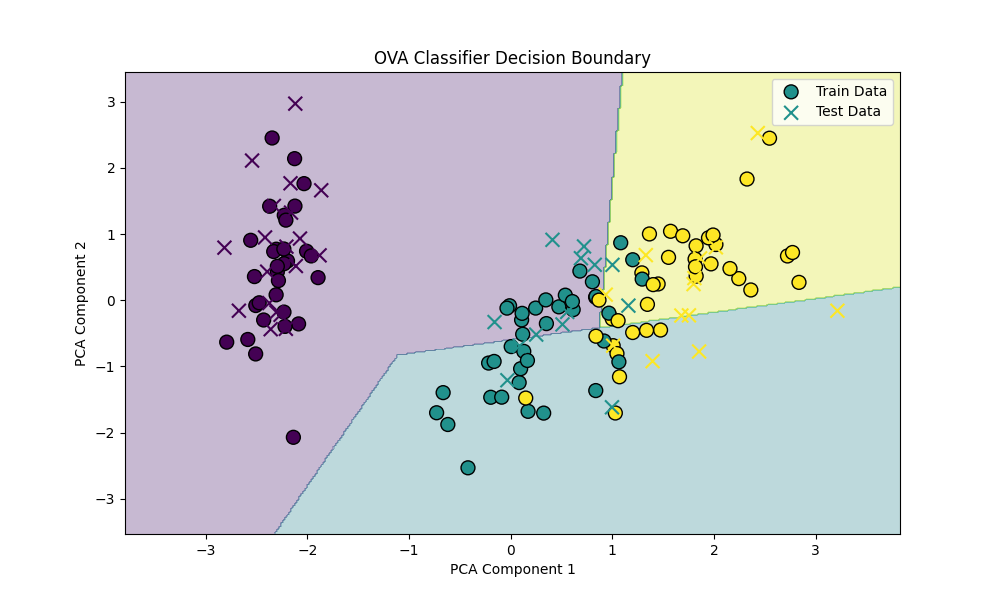
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_iris  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score  
from sklearn.preprocessing import StandardScaler  
from sklearn.decomposition import PCA  
  
class LogisticRegressionGD:  
 def \_\_init\_\_(self, learning\_rate, num\_iterations):  
 self.learning\_rate = learning\_rate  
 self.num\_iterations = num\_iterations  
 self.weights = None  
 self.bias = 0  
  
 def sigmoid(self, z):  
 return 1 / (1 + np.exp(-z))  
  
 def fit(self, X, y):  
 n\_samples, n\_features = X.shape  
 self.weights = np.zeros(n\_features)  
 for \_ in range(self.num\_iterations):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 predictions = self.sigmoid(linear\_model)  
 dw = (1 / n\_samples) \* np.dot(X.T, (predictions - y))  
 db = (1 / n\_samples) \* np.sum(predictions - y)  
 self.weights -= self.learning\_rate \* dw  
 self.bias -= self.learning\_rate \* db  
  
 def predict(self, X):  
 linear\_model = np.dot(X, self.weights) + self.bias  
 y\_predicted = self.sigmoid(linear\_model)  
 return np.array([1 if i > 0.5 else 0 for i in y\_predicted])  
  
def one\_vs\_all(X\_train, y\_train, num\_classes, learning\_rate, num\_iterations):  
 classifiers = []  
 for i in range(num\_classes):  
 y\_binary = np.where(y\_train == i, 1, 0)  
 classifier = LogisticRegressionGD(learning\_rate, num\_iterations)  
 classifier.fit(X\_train, y\_binary)  
 classifiers.append(classifier)  
 return classifiers  
  
def predict\_one\_vs\_all(X, classifiers):  
 predictions = np.zeros((X.shape[0], len(classifiers)))  
 for i, classifier in enumerate(classifiers):  
 predictions[:, i] = classifier.predict(X)  
 return np.argmax(predictions, axis=1)  
  
def one\_vs\_one(X\_train, y\_train, num\_classes, learning\_rate, num\_iterations):  
 classifiers = {}  
 for i in range(num\_classes):  
 for j in range(i + 1, num\_classes):  
 y\_binary = np.where((y\_train == i) | (y\_train == j), y\_train, -1)  
 y\_binary = np.where(y\_binary == i, 1, 0)  
 classifier = LogisticRegressionGD(learning\_rate, num\_iterations)  
 classifier.fit(X\_train, y\_binary)  
 classifiers[(i, j)] = classifier  
 return classifiers  
  
def predict\_one\_vs\_one(X, classifiers):  
 predictions = np.zeros(X.shape[0])  
 for (i, j), classifier in classifiers.items():  
 pred = classifier.predict(X)  
 predictions[pred == 1] = i  
 predictions[pred == 0] = j  
 return predictions  
  
def ecoc(X\_train, y\_train, num\_classes, learning\_rate, num\_iterations):  
 classifiers = []  
 code\_length = num\_classes - 1  
 codebook = np.array([[(i >> j) & 1 for j in range(code\_length)] for i in range(num\_classes)])  
 for i in range(code\_length):  
 y\_binary = np.array([codebook[label, i] for label in y\_train])  
 classifier = LogisticRegressionGD(learning\_rate, num\_iterations)  
 classifier.fit(X\_train, y\_binary)  
 classifiers.append(classifier)  
 return classifiers, codebook  
  
def predict\_ecoc(X, classifiers, codebook):  
 predictions = np.zeros((X.shape[0], len(classifiers)))  
 for i, classifier in enumerate(classifiers):  
 predictions[:, i] = classifier.predict(X)  
 return np.dot(predictions, 2\*\*np.arange(predictions.shape[1])[::-1])  
  
iris = load\_iris()  
X = iris.data  
y = iris.target  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)  
  
classifiers\_ova = one\_vs\_all(X\_train, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
y\_pred\_ova = predict\_one\_vs\_all(X\_test, classifiers\_ova)  
accuracy\_ova = accuracy\_score(y\_test, y\_pred\_ova)  
  
classifiers\_ovo = one\_vs\_one(X\_train, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
y\_pred\_ovo = predict\_one\_vs\_one(X\_test, classifiers\_ovo)  
accuracy\_ovo = accuracy\_score(y\_test, y\_pred\_ovo)  
  
classifiers\_ecoc, codebook = ecoc(X\_train, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
y\_pred\_ecoc = predict\_ecoc(X\_test, classifiers\_ecoc, codebook)  
accuracy\_ecoc = accuracy\_score(y\_test, y\_pred\_ecoc)  
  
print(f'One-vs-All accuracy: {accuracy\_ova \* 100:.2f}%')  
print(f'One-vs-One accuracy: {accuracy\_ovo \* 100:.2f}%')  
print(f'ECOC accuracy: {accuracy\_ecoc \* 100:.2f}%')  
best\_model = 'OVA'  
pca = PCA(n\_components=2)  
X\_train\_2d = pca.fit\_transform(X\_train)  
X\_test\_2d = pca.transform(X\_test)  
if best\_model == 'OVA':  
 classifiers = one\_vs\_all(X\_train\_2d, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
 y\_pred = predict\_one\_vs\_all(X\_test\_2d, classifiers)  
elif best\_model == 'OVO':  
 classifiers = one\_vs\_one(X\_train\_2d, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
 y\_pred = predict\_one\_vs\_one(X\_test\_2d, classifiers)  
else:  
 classifiers, codebook = ecoc(X\_train\_2d, y\_train, num\_classes=3, learning\_rate=0.1, num\_iterations=1000)  
 y\_pred = predict\_ecoc(X\_test\_2d, classifiers, codebook)  
h = .02  
x\_min, x\_max = X\_train\_2d[:, 0].min() - 1, X\_train\_2d[:, 0].max() + 1  
y\_min, y\_max = X\_train\_2d[:, 1].min() - 1, X\_train\_2d[:, 1].max() + 1  
xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))  
  
Z = np.zeros(xx.shape)  
for i in range(xx.shape[0]):  
 for j in range(xx.shape[1]):  
 point = np.array([xx[i, j], yy[i, j]])  
 if best\_model == 'OVA':  
 Z[i, j] = np.argmax([clf.predict(point.reshape(1, -1)) for clf in classifiers])  
 elif best\_model == 'OVO':  
 Z[i, j] = predict\_one\_vs\_one(point.reshape(1, -1), classifiers)  
 else:  
 Z[i, j] = predict\_ecoc(point.reshape(1, -1), classifiers, codebook)  
  
plt.figure(figsize=(10, 6))  
plt.contourf(xx, yy, Z, alpha=0.3)  
plt.scatter(X\_train\_2d[:, 0], X\_train\_2d[:, 1], c=y\_train, marker='o', edgecolor='k', s=100, label="Train Data")  
plt.scatter(X\_test\_2d[:, 0], X\_test\_2d[:, 1], c=y\_test, marker='x', edgecolor='k', s=100, label="Test Data")  
plt.title(f'{best\_model} Classifier Decision Boundary')  
plt.xlabel('PCA Component 1')  
plt.ylabel('PCA Component 2')  
plt.legend(loc='best')  
plt.show()

****输出与结果展示：****

****One-vs-All 分类准确率: 75.56%****

****One-vs-One 分类准确率: 33.33%****

****ECOC 分类准确率: 42.22%****



**图2.3 降维后可视化分类边界**

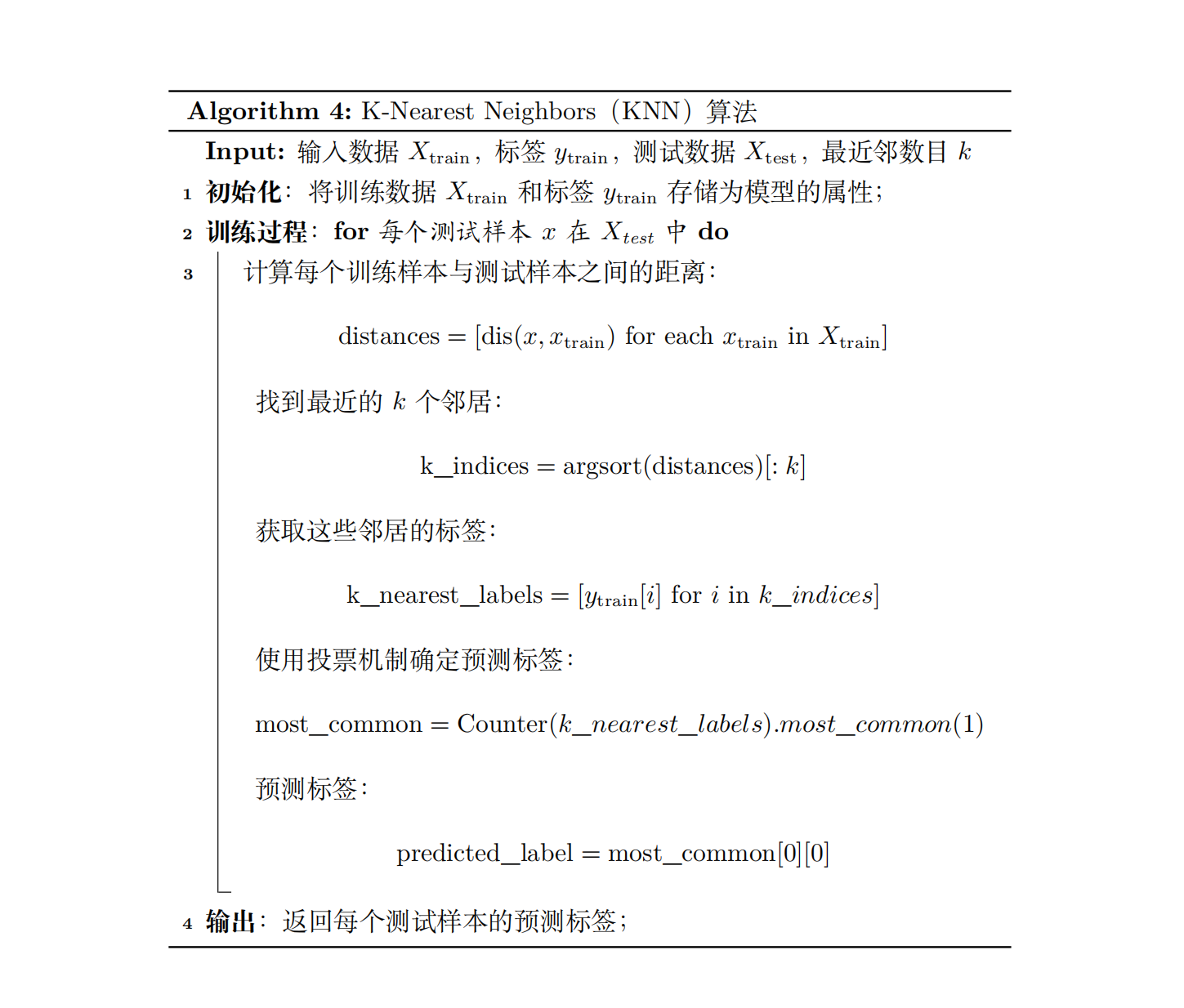
**三、K近邻算法（KNN）**

**3.1 KNN算法**

****目标：**手搓KNN算法，通过可视化展示KNN在二维数据上的分类效果和最近邻的选取过程。**

****思路：这个算法属于懒惰学习，比较简单。**通过计算欧几里得距离并按距离排序来选择最近的k个邻居，通过**投票**确定最终预测结果。**

****伪代码：****

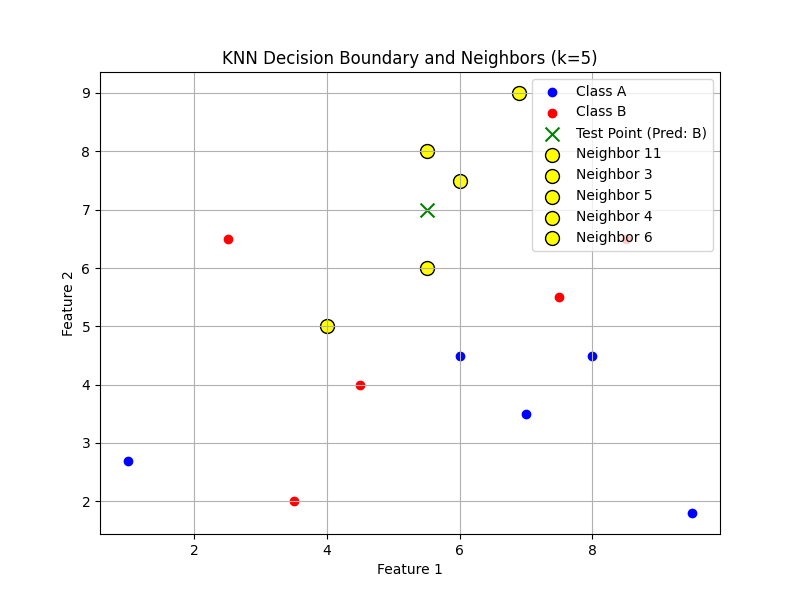


**图3.1 KNN伪代码**

****Python实现：****

*# -\*- coding: utf-8 -\*-*import numpy as np  
import matplotlib.pyplot as plt  
from collections import Counter  
def dis(x1, x2):  
 return np.sqrt(np.sum((x1 - x2) \*\* 2)) *# 计算欧几里得距离*class KNN:  
 def \_\_init\_\_(self, k=10):  
 self.k = k *# 最近邻数目* def fit(self, X\_train, y\_train):  
 self.X\_train = X\_train  
 self.y\_train = y\_train  
 def predict(self, X\_test):  
 return [self.\_predict(x) for x in X\_test]  
 def \_predict(self, x):  
 distances = [dis(x, x\_train) for x\_train in self.X\_train]  
 print(f"遍历所有训练样本与新样本之距离: {distances}")  
 *# 找到最近的k个邻居* k\_indices = np.argsort(distances)[:self.k]  
 print(f"选取最近的{self.k}个样本的索引: {k\_indices}")  
 k\_nearest\_labels = [self.y\_train[i] for i in k\_indices]  
 print(f"它们的标签: {k\_nearest\_labels}")  
 most\_common = Counter(k\_nearest\_labels).most\_common(1)  
 print(f"投票得出的标签: {most\_common[0][0]}")  
 return most\_common[0][0]  
def plot\_decision\_boundary(X\_train, y\_train, X\_test, y\_pred, k=10):  
 *# 画出训练数据* plt.figure(figsize=(8, 6))  
 plt.scatter(X\_train[y\_train == 'A'][:, 0], X\_train[y\_train == 'A'][:, 1], color='blue', label='Class A')  
 plt.scatter(X\_train[y\_train == 'B'][:, 0], X\_train[y\_train == 'B'][:, 1], color='red', label='Class B')  
 plt.scatter(X\_test[:, 0], X\_test[:, 1], color='green', marker='x', s=100, label=f'Test Point (Pred: {y\_pred[0]})')  
 knn = KNN(k=k)  
 knn.fit(X\_train, y\_train)  
 distances = [dis(X\_test[0], x) for x in X\_train]  
 k\_indices = np.argsort(distances)[:k]  
 for idx in k\_indices:  
 plt.scatter(X\_train[idx, 0], X\_train[idx, 1], color='yellow', edgecolors='black', s=100, marker='o',  
 label=f'Neighbor {idx + 1}')  
 plt.title(f"KNN Decision Boundary and Neighbors (k={k})")  
 plt.xlabel("Feature 1")  
 plt.ylabel("Feature 2")  
 plt.legend(loc="best")  
 plt.grid(True)  
 plt.show()  
X\_train = np.array([  
 [1.0, 2.7], [9.5, 1.8], [5.5, 8.0], [6.9, 9.0], [5.5, 6.0],  
 [4.0, 5.0], [7.0, 3.5], [8.0, 4.5], [2.5, 6.5], [3.5, 2.0],  
 [6.0, 7.5], [7.5, 5.5], [8.5, 6.5], [6.0, 4.5], [4.5, 4.0]  
])  
y\_train = np.array(['A', 'A', 'B', 'B', 'B', 'A', 'A', 'A', 'B', 'B', 'A', 'B', 'B', 'A', 'B'])  
X\_test = np.array([[5.5, 7.0]])  
knn = KNN(k=5)  
knn.fit(X\_train, y\_train)  
y\_pred = knn.predict(X\_test)  
print(f'Predicted Class: {y\_pred[0]}')  
plot\_decision\_boundary(X\_train, y\_train, X\_test, y\_pred, k=5)

****输出与结果展示：****



**图3.2 KNN模拟**

遍历所有训练样本与新样本之距离:

[6.224146527838174,6.560487786742691, 1.0, 2.4413111231467406, 1.0, 2.5, 3.8078865529319543, 3.5355339059327378, 3.0413812651491097,5.385164807134504, 0.7071067811865476,2.5, 3.0413812651491097, 2.5495097567963922, 3.1622776601683795]

选取最近的5个样本的索引: [10 2 4 3 5]

它们的标签: ['A', 'B', 'B', 'B', 'A']

投票得出的标签: B

Predicted Class: B

****反思与感悟：****

1. **懒惰学习**：指直到出现新的测试样本，该算法才开始依据[训练样本](https://zhida.zhihu.com/search?content_id=239666030&content_type=Article&match_order=1&q=%E8%AE%AD%E7%BB%83%E6%A0%B7%E6%9C%AC&zhida_source=entity" \t "https://zhuanlan.zhihu.com/p/_blank)进行样本的预测处理工作。也就是说，该算法事先不会对训练样本进行任何处理，只会懒散地等待测试样本的到来，然后才开始工作。
2. 由于**维度诅咒**（curse of dimensionality），KNN非常容易过拟合。维度诅咒指的是对于固定大小的训练数据集，随着维度的增加，特征空间变得越来越稀疏。我们可以认为，即使是最近的邻居，在高维空间中也太远了，无法给出一个很好的估计。

虽然可以用正则化来避免过拟合。然而，在正则化不适用的模型中，例如决策树和 KNN，我们可以使用特征选择和降维技术来帮助我们避免维数的诅咒。

**3.2 使用KNN完成对 iris 数据集的分类**

****目标：****使用手动和sklearn的KNN实现，完成对iris数据集的分类。

**数据集：**Iris数据集（安德森鸢尾花卉数据集）。

****Python实现（仅展示核心代码）****

data = datasets.load\_iris()  
sample = data['data']  
target = data['target']  
print(f"Feature Names: {data['feature\_names']}")  
print(f"Target Names: {data['target\_names']}")  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(sample, target, test\_size=0.3, random\_state=2020)  
def sklearn\_knn(k):  
 knn = KNN(n\_neighbors=k)  
 knn.fit(X\_train, y\_train)  
 accuracy = knn.score(X\_test, y\_test)  
 return knn, accuracy  
best\_k\_sklearn = None  
best\_accuracy\_sklearn = 0  
best\_knn\_sklearn = None  
for k in [1,2,3,5,6,7,8,9,10]:  
 model, accuracy = sklearn\_knn(k)  
 if accuracy > best\_accuracy\_sklearn:  
 best\_accuracy\_sklearn = accuracy  
 best\_knn\_sklearn = model  
 best\_k\_sklearn = k  
print(f"Best k for sklearn KNN: {best\_k\_sklearn} with accuracy: {best\_accuracy\_sklearn \* 100:.2f}%")  
best\_k\_manual = None  
best\_accuracy\_manual = 0  
best\_knn\_manual = None  
for k in [1,2,3,5,6,7,8,9,10]:  
 model, accuracy = manual\_knn(k)  
 if accuracy > best\_accuracy\_manual:  
 best\_accuracy\_manual = accuracy  
 best\_knn\_manual = model  
 best\_k\_manual = k  
print(f"Best k for Manual KNN: {best\_k\_manual} with accuracy: {best\_accuracy\_manual \* 100:.2f}%")  
*# 1. 对 sklearn 的 KNN 使用最佳 k 重新训练*best\_knn\_sklearn.fit(X\_train, y\_train)  
sklearn\_predictions = best\_knn\_sklearn.predict(X\_test)  
print(f"Re-trained sklearn KNN Predictions (k={best\_k\_sklearn}): {sklearn\_predictions}")  
*# 2. 对手动实现的 KNN 使用最佳 k 重新训练*best\_knn\_manual.fit(X\_train, y\_train)  
manual\_predictions = best\_knn\_manual.predict(X\_test)  
print(f"Re-trained Manual KNN Predictions (k={best\_k\_manual}): {manual\_predictions}")

****输出与结果展示：****

Feature Names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

Target Names: ['setosa' 'versicolor' 'virginica']

Best k for sklearn KNN: 6 with accuracy: 95.56%

Best k for Manual KNN: 7 with accuracy: 95.56%

Re-trained sklearn KNN Predictions (k=6): [2 0 1 1 1 1 2 1 0 0 2 1 0 2 2 0 1 1 2 0 0 2 1 0 2 1 1 1 0 0 2 0 0 0 2 0 0 1 0 2 0 2 1 0 1]

Re-trained Manual KNN Predictions (k=7): [2, 0, 1, 1, 1, 2, 2, 1, 0, 0, 2, 1, 0, 2, 2, 0, 1, 1, 2, 0, 0, 2, 2, 0, 2, 1, 1, 1, 0, 0, 2, 0, 0, 0, 2, 0, 0, 1, 0, 2, 0, 2, 1, 0, 1]

****反思与感悟：****

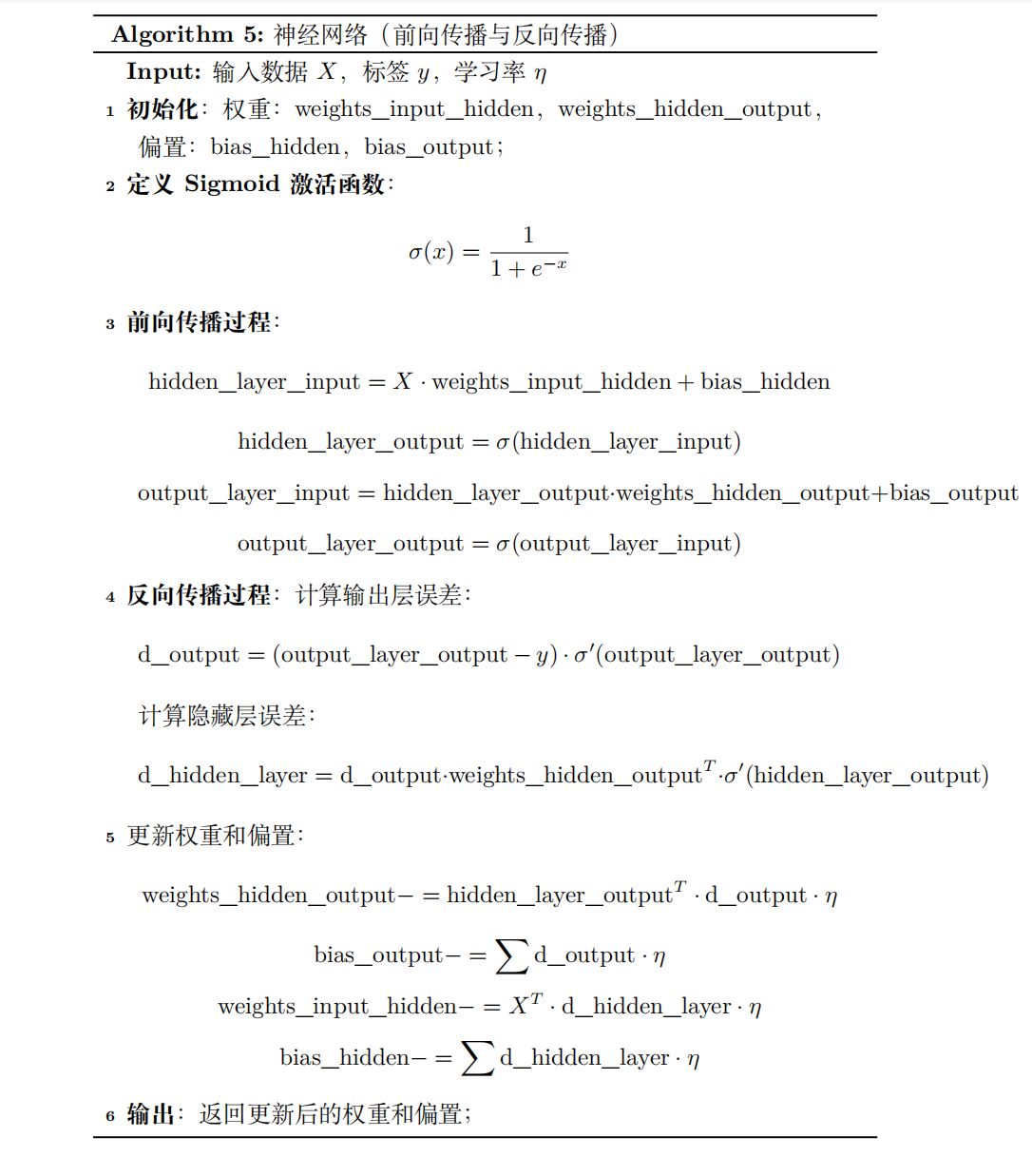
选择合适的**K**值对于KNN的表现至关重要，过小或过大的**K**值都会影响准确性，**调参**是必要的步骤。

**四、神经网络**

**4.1 BP神经网络**

****目标：**手搓BP神经网络、反向传播求导更新权重的核心代码。**

****伪代码：****



**图4.1 神经网络的重要算法伪代码**

****Python实现：****

*# 神经网络类*class NeuralNetwork:  
 def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):  
 self.input\_size = input\_size  
 self.hidden\_size = hidden\_size  
 self.output\_size = output\_size  
 *# 初始化权重和偏置* self.weights\_input\_hidden = np.random.randn(input\_size, hidden\_size)  
 self.bias\_hidden = np.zeros((1, hidden\_size))  
 self.weights\_hidden\_output = np.random.randn(hidden\_size, output\_size)  
 self.bias\_output = np.zeros((1, output\_size))  
 *# 前向传播* def forward(self, X):  
 self.input\_layer = X  
 self.hidden\_layer\_input = np.dot(X, self.weights\_input\_hidden) + self.bias\_hidden  
 self.hidden\_layer\_output = self.sigmoid(self.hidden\_layer\_input) *# 使用sigmoid激活函数* self.output\_layer\_input = np.dot(self.hidden\_layer\_output, self.weights\_hidden\_output) + self.bias\_output  
 self.output\_layer\_output = self.sigmoid(self.output\_layer\_input) *# 使用sigmoid激活函数* return self.output\_layer\_output  
 *# 激活函数（Sigmoid）* def sigmoid(self, x):  
 return 1 / (1 + np.exp(-x))  
 *# Sigmoid的导数* def sigmoid\_derivative(self, x):  
 return x \* (1 - x)  
 *# 均方误差* def mse(self, y\_pred, y\_true):  
 return np.mean((y\_pred - y\_true) \*\* 2)  
 *# 反向传播* def backward(self, X, y, learning\_rate):  
 *# 计算输出层的误差* self.d\_output = (self.output\_layer\_output - y) \* self.sigmoid\_derivative(self.output\_layer\_output)  
 self.d\_hidden\_layer = self.d\_output.dot(self.weights\_hidden\_output.T) \* self.sigmoid\_derivative(self.hidden\_layer\_output)  
 *# 更新权重和偏置* self.weights\_hidden\_output -= self.hidden\_layer\_output.T.dot(self.d\_output) \* learning\_rate  
 self.bias\_output -= np.sum(self.d\_output, axis=0, keepdims=True) \* learning\_rate  
  
 *# 修正 input -> hidden 层的权重更新* self.weights\_input\_hidden -= np.dot(self.input\_layer.T, self.d\_hidden\_layer) \* learning\_rate  
 self.bias\_hidden -= np.sum(self.d\_hidden\_layer, axis=0, keepdims=True) \* learning\_rate

**4.2 反向传播算法实现及动画化\***

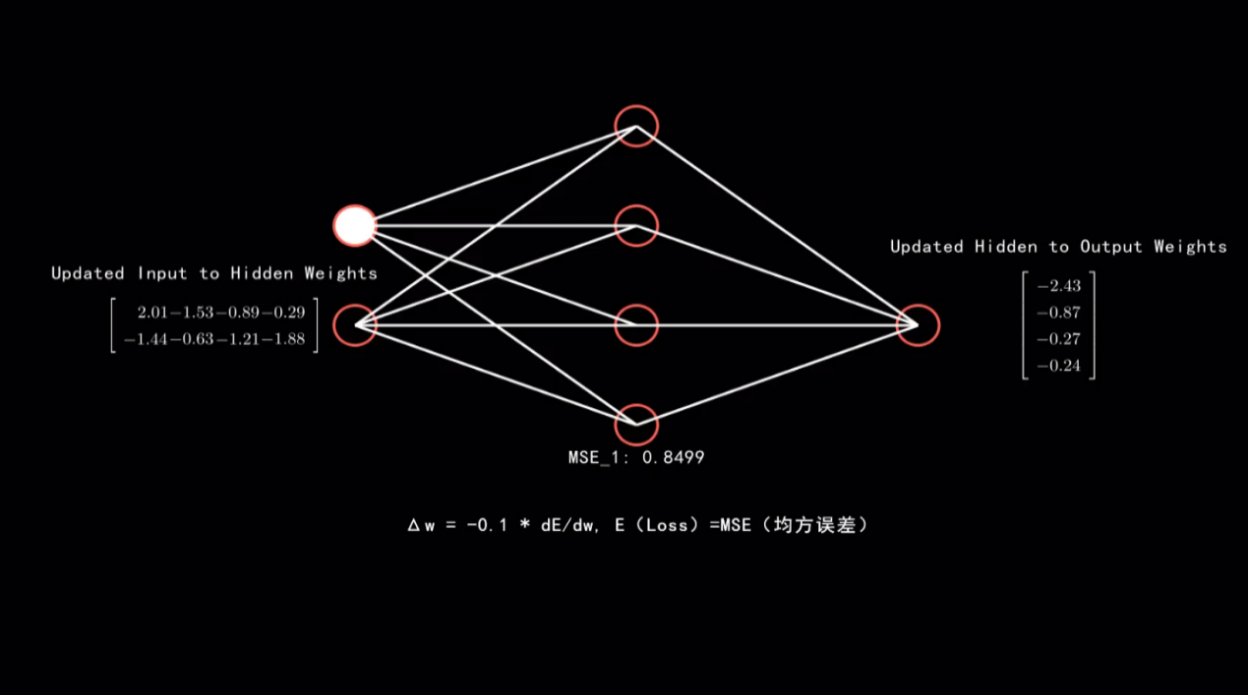
****目标：**设计一个简单的神经网络，编写反向传播算法，使用梯度下降算法实现参数更新。**通过 manim 库[[2]](#footnote-1)可视化展示神经网络的前向传播和反向传播过程，帮助学习者直观理解神经网络的工作原理及梯度下降算法。

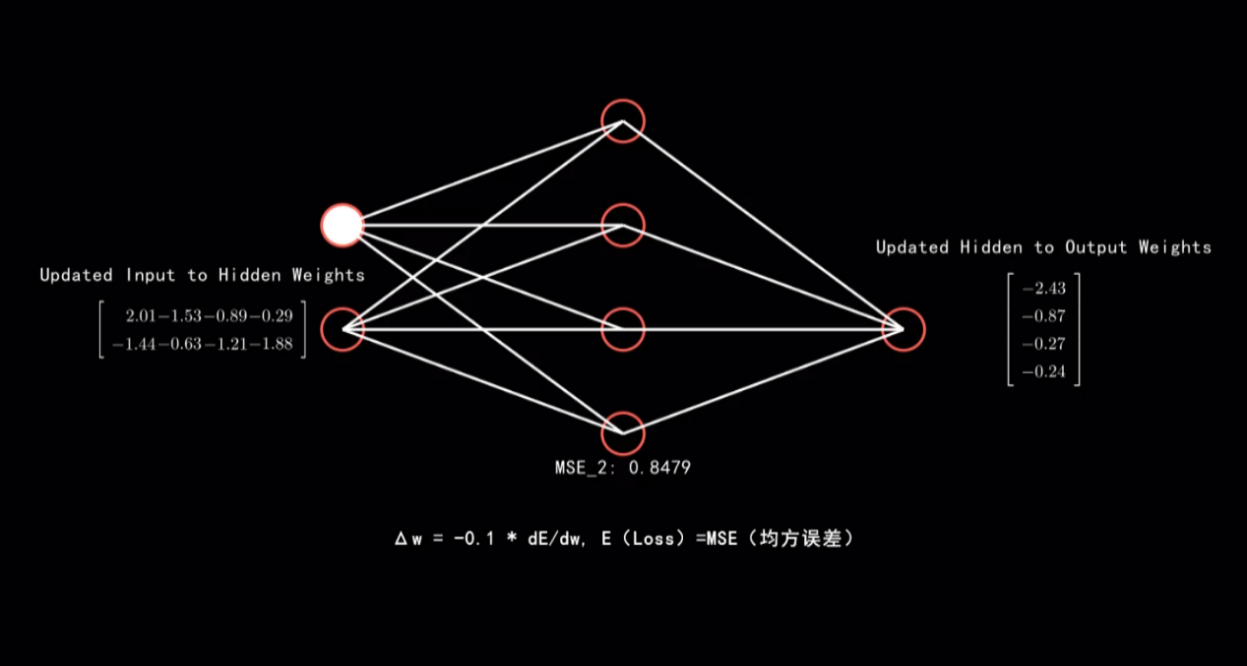
****思路：****首先手动实现一个简单的神经网络，包括前向传播、反向传播、Sigmoid 激活函数及损失计算。然后利用 manim 构建神经网络的图形，展示数据流动、误差反向传播和权重更新的动画。通过创建一个简单的 XOR 问题数据集，展示神经网络在训练过程中的每一步，包括输入层到输出层的数据流动、误差的反向传播及权重更新。需要安装的依赖：Miktex。

****Python实现：（这里仅展示部分核心代码）****  
**# Manim场景：XOR异或问题**class NeuralNetworkScene(Scene):  
 def construct(self):  
 *# 调整相机的比例和画布大小* self.camera.frame\_width = 20  
 self.camera.frame\_height = 15  
 self.camera.frame\_center = ORIGIN  
 *# 创建神经网络模型* input\_size = 2  
 hidden\_size = 4  
 output\_size = 1  
 nn = NeuralNetwork(input\_size, hidden\_size, output\_size)  
 *# 输入数据和标签* X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) *# 输入数据（4个样本，2个特征）* y = np.array([[0], [1], [1], [0]]) *# 输出标签（XOR 问题）  
 # 显示初始权重矩阵* weight\_input\_hidden\_matrix = self.create\_weight\_matrix(nn.weights\_input\_hidden, "Input to Hidden Weights", LEFT \* 6)  
 weight\_hidden\_output\_matrix = self.create\_weight\_matrix(nn.weights\_hidden\_output, "Hidden to Output Weights", RIGHT \* 6)  
 self.play(Write(weight\_input\_hidden\_matrix))  
 self.play(Write(weight\_hidden\_output\_matrix))  
 self.wait(1)  
 *# 创建神经网络图形* input\_circles, hidden\_circles, output\_circles, lines\_input\_hidden, lines\_hidden\_output = self.create\_neural\_network\_graph(input\_size, hidden\_size, output\_size)  
 *# 显示神经网络结构* self.play(LaggedStart(\*[Create(circle) for circle in input\_circles + hidden\_circles + output\_circles], lag\_ratio=0.1))  
 self.play(LaggedStart(\*[Create(line) for line in lines\_input\_hidden + lines\_hidden\_output], lag\_ratio=0.1))  
 *# 训练神经网络并制作动画* for epoch in range(1): *# 便于展示* for i in range(len(X)):  
 *# 第一次前向传播过程* output = nn.forward(X[i:i+1])  
 mse\_1 = nn.mse(output, y[i:i+1]) *# 第一次正向传播后的MSE* self.show\_forward\_propagation(X[i:i+1], nn, input\_circles, hidden\_circles, output\_circles, lines\_input\_hidden, lines\_hidden\_output, mse\_1, "MSE\_1")  
 *# 反向传播过程* nn.backward(X[i:i+1], y[i:i+1], learning\_rate=0.1)  
 self.show\_backward\_propagation(nn, X[i:i+1], y[i:i+1], 0.1, input\_circles, hidden\_circles, output\_circles)  
 *# 第二次前向传播计算新的输出与MSE* output\_2 = nn.forward(X[i:i+1])  
 mse\_2 = nn.mse(output\_2, y[i:i+1]) *# 第二次正向传播后的MSE* self.show\_forward\_propagation(X[i:i+1], nn, input\_circles, hidden\_circles, output\_circles, lines\_input\_hidden, lines\_hidden\_output, mse\_2, "MSE\_2")  
 *# 更新权重矩阵并展示* updated\_weight\_input\_hidden\_matrix = self.create\_weight\_matrix(nn.weights\_input\_hidden, "Updated Input to Hidden Weights", LEFT \* 6)  
 updated\_weight\_hidden\_output\_matrix = self.create\_weight\_matrix(nn.weights\_hidden\_output, "Updated Hidden to Output Weights", RIGHT \* 6)  
 self.play(Transform(weight\_input\_hidden\_matrix, updated\_weight\_input\_hidden\_matrix))  
 self.play(Transform(weight\_hidden\_output\_matrix, updated\_weight\_hidden\_output\_matrix))  
  
 *# 等待0.5秒，便于查看动画效果* self.wait(0.5)

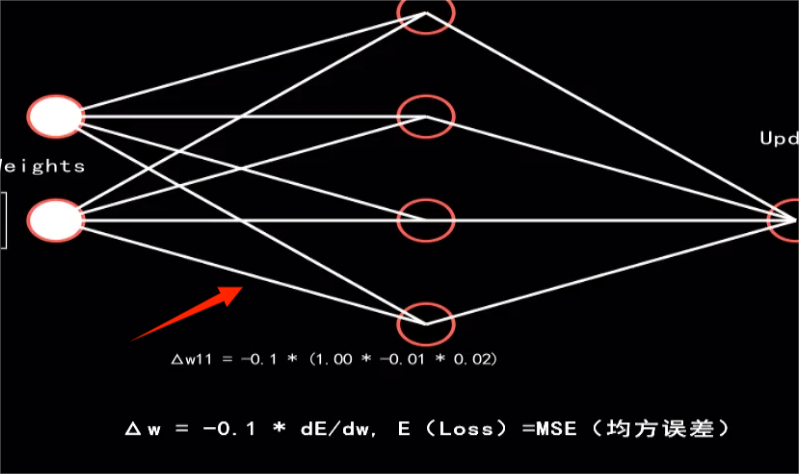
*运行指令：  
# manim -pqh C:\Users\小方\AppData\Local\Programs\Python\Python311\练习\机器学习ML\_上机实验报告\4.神经网络\神经网络\_反向传播算法实现及动画化.py NeuralNetworkScene*

****输出与结果展示：（视频位于本节代码压缩包中，这里提供截图）****

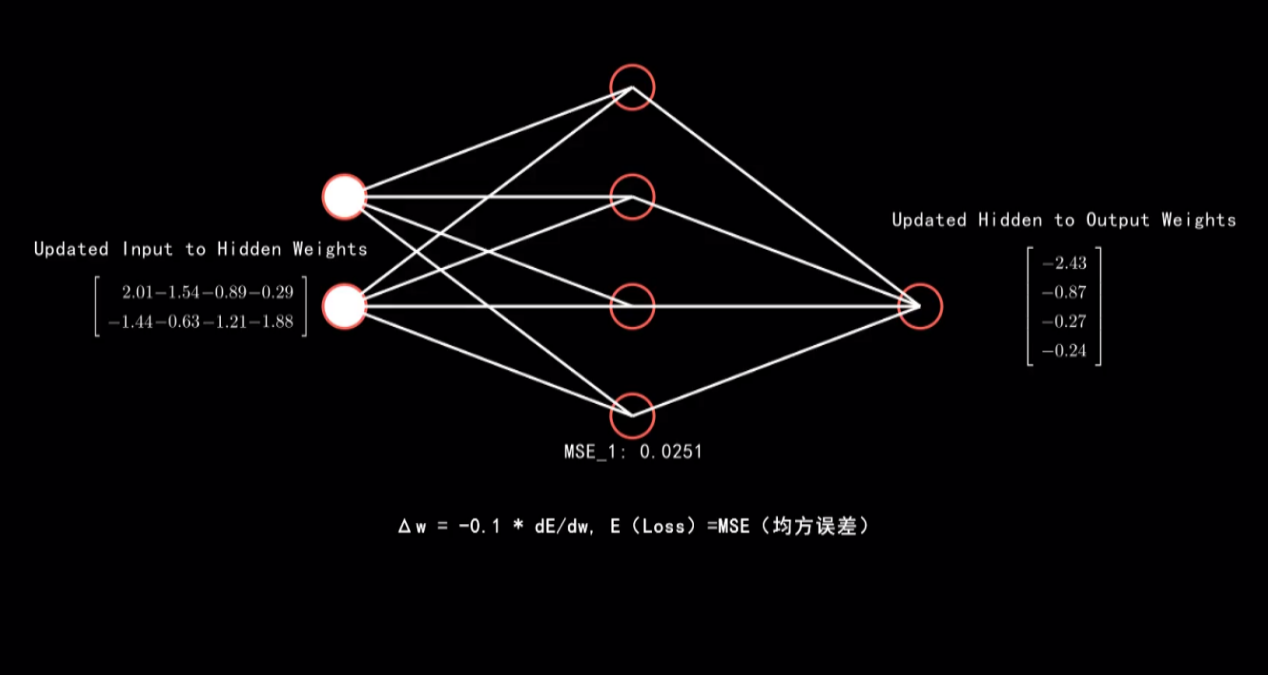


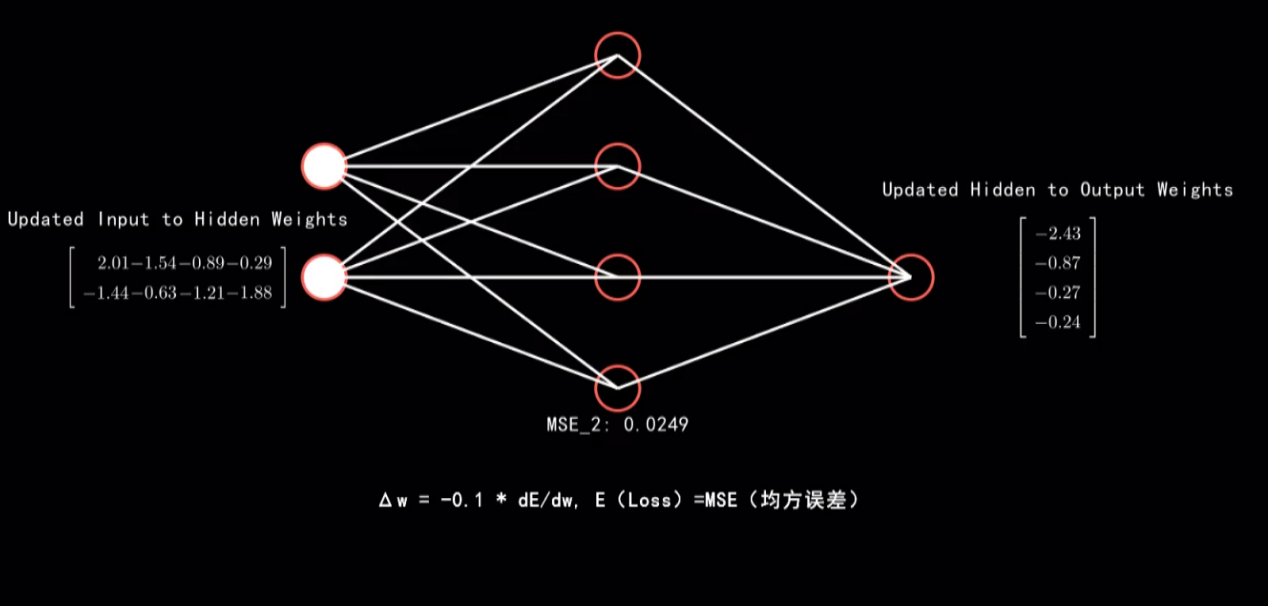


****按顺序更新权重：****



****反向传播更新后：****





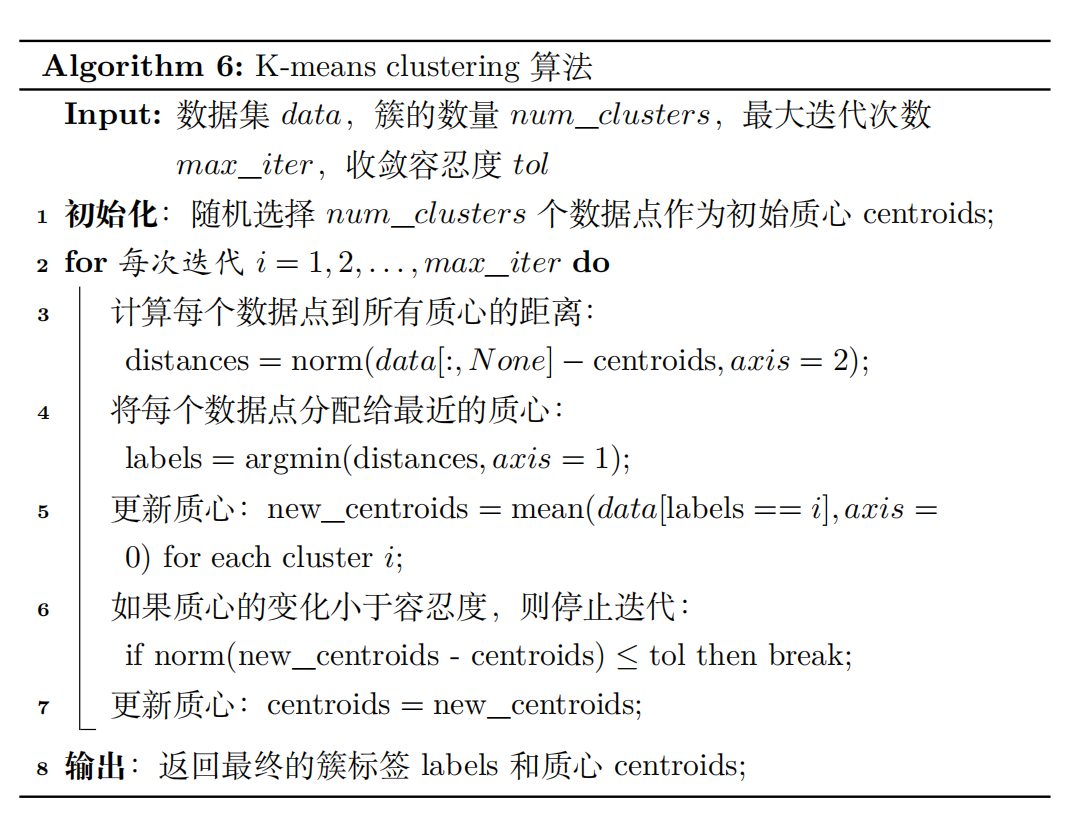
**图4.2 神经网络可视化**

**五、聚类**

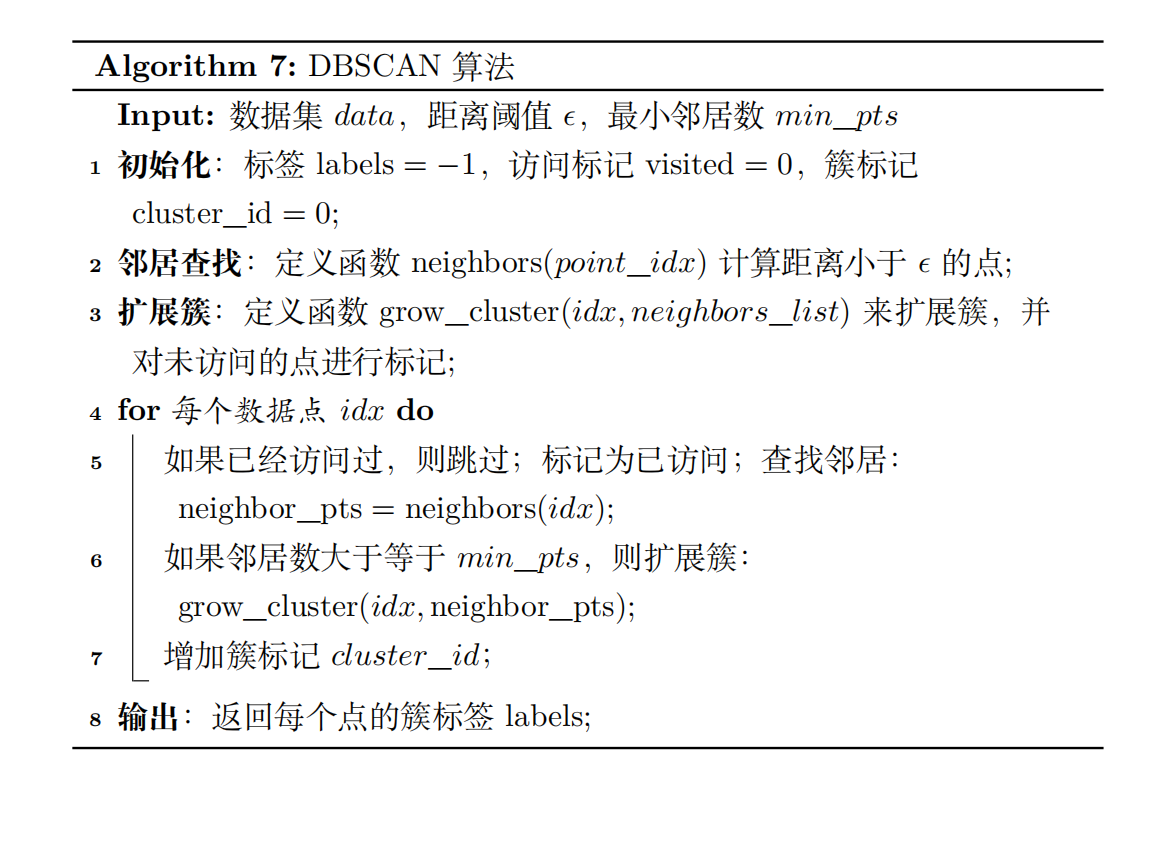
**5.1 聚类算法**

****目标：**手搓**两种经典的聚类算法：K-means 和 DBSCAN

****伪代码：****



**图5.1 K-Means算法伪代码**



**图5.2 DBSCAN算法伪代码**

****Python实现：****

*# -\*- coding: utf-8 -\*-*import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import make\_blobs, make\_circles  
*# K-means clustering*def kmeans(data, num\_clusters, max\_iter=100, tol=1e-4):  
 np.random.seed(66666)  
 num\_points, num\_features = data.shape  
 centroids = data[np.random.choice(num\_points, num\_clusters, replace=False)]  
 for \_ in range(max\_iter):  
 distances = np.linalg.norm(data[:, None] - centroids, axis=2)  
 labels = np.argmin(distances, axis=1)  
 new\_centroids = np.array([data[labels == i].mean(axis=0) for i in range(num\_clusters)])  
 if np.all(np.linalg.norm(new\_centroids - centroids, axis=1) <= tol):  
 break  
 centroids = new\_centroids  
 return labels, centroids  
*# DBSCAN clustering*def dbscan(data, eps, min\_pts):  
 num\_points = data.shape[0]  
 labels = -np.ones(num\_points)  
 visited = np.zeros(num\_points, dtype=bool)  
 cluster\_id = 0  
 def neighbors(point\_idx):  
 dists = np.linalg.norm(data - data[point\_idx], axis=1)  
 return np.where(dists <= eps)[0]  
 def grow\_cluster(idx, neighbors\_list):  
 labels[idx] = cluster\_id  
 i = 0  
 while i < len(neighbors\_list):  
 pt = neighbors\_list[i]  
 if not visited[pt]:  
 visited[pt] = True  
 new\_neighbors = neighbors(pt)  
 if len(new\_neighbors) >= min\_pts:  
 neighbors\_list = np.unique(np.concatenate((neighbors\_list, new\_neighbors)))  
 if labels[pt] == -1:  
 labels[pt] = cluster\_id  
 i += 1  
 for idx in range(num\_points):  
 if visited[idx]:  
 continue  
 visited[idx] = True  
 neighbor\_pts = neighbors(idx)  
 if len(neighbor\_pts) >= min\_pts:  
 grow\_cluster(idx, neighbor\_pts)  
 cluster\_id += 1  
 return labels

**5.2 基于kmeans.txt 数据集的聚类对比**

****目标：**将手写的 K-means 和 DBSCAN 聚类算法应用到实际的数据集（如 kmeans.txt 数据集）上，**实现与 sklearn 版本的对比。**通过分析不同的超参数和聚类结果、**度量指标**，展示如何选择合适的聚类方法和参数。**

****数据集：**kmeans.txt 数据集**

****知识点回顾：****

**（1）轮廓系数用于衡量样本在其所属簇内的紧密度与它与最邻近簇的分离度。其公式为：**

**其中：**

**是样本与同簇内其他点的平均距离。**

**是样本与最邻近簇的所有点的平均距离。**

**的值范围是[-1,1],其中 1 表示最佳分离，-1 表示错误的聚类。**

**最终的轮廓系数为所有样本轮廓系数的平均值：**

**（2）Davies-Bouldin 指数(Davies-Bouldin Index,DBI) Davies-Bouldin 指数衡量聚类质量，值越低表示聚类效果越好。其公式为：**

**其中：**

**是聚类数。**

**是簇内样本的平均距离(簇的紧密度)。**

**是簇和簇之间的距离(簇的分离度)。**

**（3）Calinski-Harabasz 指数(Calinski-Harabasz Index, CHI) Calinski-Harabasz 指数衡量簇内紧密度和簇间分离度的比值，值越大表示聚类效果越好。其公**

**式为：**

**其中：**

**是样本数量。**

**是簇的数量。**

**是簇间离散度矩阵。**

**是簇内离散度矩阵。**

**Tr 表示矩阵的迹，即对角线元素的和。**

1. **指标含义**

**轮廓系数（Silhouette Score）： 该值介于-1到1之间，表示聚类的紧密度和分离度。较高的值表示聚类效果较好。**

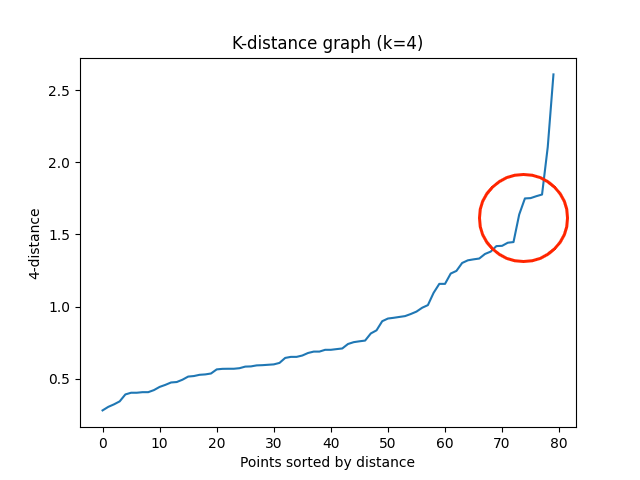
**Davies-Bouldin指数（Davies-Bouldin Index）： 该值非负，较低的值表示聚类效果较好。它衡量聚类之间的相似度，值越低表示聚类效果越好。**

**Calinski-Harabasz指数（Calinski-Harabasz Index）： 该值越高越好，表示聚类的方差比，较高的值表示聚类效果较好。**

****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.cluster import KMeans, DBSCAN  
from sklearn.neighbors import NearestNeighbors  
from sklearn.metrics import silhouette\_score, davies\_bouldin\_score, calinski\_harabasz\_score  
  
def kmeans(data, num\_clusters, max\_iter=100, tol=1e-4):  
 np.random.seed(66666)  
 num\_points, num\_features = data.shape  
 centroids = data[np.random.choice(num\_points, num\_clusters, replace=False)]  
 for \_ in range(max\_iter):  
 distances = np.linalg.norm(data[:, None] - centroids, axis=2)  
 labels = np.argmin(distances, axis=1)  
 new\_centroids = np.array([data[labels == i].mean(axis=0) for i in range(num\_clusters)])  
 if np.all(np.linalg.norm(new\_centroids - centroids, axis=1) <= tol):  
 break  
 centroids = new\_centroids  
 return labels, centroids  
  
def dbscan(data, eps, min\_pts):  
 num\_points = data.shape[0]  
 labels = -np.ones(num\_points)  
 visited = np.zeros(num\_points, dtype=bool)  
 cluster\_id = 0  
 def neighbors(point\_idx):  
 dists = np.linalg.norm(data - data[point\_idx], axis=1)  
 return np.where(dists <= eps)[0]  
 def grow\_cluster(idx, neighbors\_list):  
 labels[idx] = cluster\_id  
 i = 0  
 while i < len(neighbors\_list):  
 pt = neighbors\_list[i]  
 if not visited[pt]:  
 visited[pt] = True  
 new\_neighbors = neighbors(pt)  
 if len(new\_neighbors) >= min\_pts:  
 neighbors\_list = np.unique(np.concatenate((neighbors\_list, new\_neighbors)))  
 if labels[pt] == -1:  
 labels[pt] = cluster\_id  
 i += 1  
 for idx in range(num\_points):  
 if visited[idx]:  
 continue  
 visited[idx] = True  
 neighbor\_pts = neighbors(idx)  
 if len(neighbor\_pts) >= min\_pts:  
 grow\_cluster(idx, neighbor\_pts)  
 cluster\_id += 1  
 return labels  
  
def plot\_k\_distance(data, k=4):  
 neighbors = NearestNeighbors(n\_neighbors=k)  
 neighbors.fit(data)  
 distances, indices = neighbors.kneighbors(data)  
 k\_distances = np.sort(distances[:, k - 1], axis=0)  
 plt.plot(k\_distances)  
 plt.title(f"K-distance graph (k={k})")  
 plt.xlabel("Points sorted by distance")  
 plt.ylabel(f"{k}-distance")  
 plt.show()  
  
def visualize\_results(data, labels, centroids=None, algorithm\_name="Clustering"):  
 plt.figure(figsize=(8, 6))  
 unique\_labels = np.unique(labels)  
 for lbl in unique\_labels:  
 if lbl == -1:  
 color = "gray"  
 label = "Noise"  
 else:  
 color = None  
 label = f"Cluster {lbl}"  
 plt.scatter(data[labels == lbl, 0], data[labels == lbl, 1], label=label, c=color)  
 if centroids is not None:  
 plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', marker='X', label='Centroids')  
 plt.title(f"{algorithm\_name} Results")  
 plt.legend()  
 plt.show()  
  
def sklearn\_kmeans(data, num\_clusters):  
 kmeans = KMeans(n\_clusters=num\_clusters, random\_state=42)  
 kmeans.fit(data)  
 return kmeans.labels\_, kmeans.cluster\_centers\_  
  
def sklearn\_dbscan(data, eps, min\_samples):  
 dbscan = DBSCAN(eps=eps, min\_samples=min\_samples)  
 dbscan.fit(data)  
 return dbscan.labels\_  
  
def compute\_metrics(data, predicted\_labels):  
 silhouette = silhouette\_score(data, predicted\_labels)  
 dbi = davies\_bouldin\_score(data, predicted\_labels)  
 ch\_score = calinski\_harabasz\_score(data, predicted\_labels)  
 return silhouette, dbi, ch\_score  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 data = np.loadtxt('kmeans.txt')  
 plot\_k\_distance(data, k=4)  
 eps = 1.7  
 min\_pts = 5  
 k = 7  
  
 kmeans\_labels, kmeans\_centroids = kmeans(data, k)  
 visualize\_results(data, kmeans\_labels, kmeans\_centroids, "Custom K-means")  
  
 sklearn\_kmeans\_labels, sklearn\_kmeans\_centroids = sklearn\_kmeans(data, k)  
 visualize\_results(data, sklearn\_kmeans\_labels, sklearn\_kmeans\_centroids, "Sklearn K-means")  
  
 dbscan\_labels = dbscan(data, eps, min\_pts)  
 visualize\_results(data, dbscan\_labels, algorithm\_name="Custom DBSCAN")  
  
 sklearn\_dbscan\_labels = sklearn\_dbscan(data, eps, min\_pts)  
 visualize\_results(data, sklearn\_dbscan\_labels, algorithm\_name="Sklearn DBSCAN")  
  
 print("Metrics for Custom K-means:")  
 silhouette\_kmeans, dbi\_kmeans, ch\_kmeans = compute\_metrics(data, kmeans\_labels)  
 print(f"Silhouette Score: {silhouette\_kmeans}, Davies-Bouldin Index: {dbi\_kmeans}, Calinski-Harabasz Index: {ch\_kmeans}")  
  
 print("\nMetrics for Sklearn K-means:")  
 silhouette\_sklearn\_kmeans, dbi\_sklearn\_kmeans, ch\_sklearn\_kmeans = compute\_metrics(data, sklearn\_kmeans\_labels)  
 print(f"Silhouette Score: {silhouette\_sklearn\_kmeans}, Davies-Bouldin Index: {dbi\_sklearn\_kmeans}, Calinski-Harabasz Index: {ch\_sklearn\_kmeans}")  
  
 print("\nMetrics for Custom DBSCAN:")  
 silhouette\_dbscan, dbi\_dbscan, ch\_dbscan = compute\_metrics(data, dbscan\_labels)  
 print(f"Silhouette Score: {silhouette\_dbscan}, Davies-Bouldin Index: {dbi\_dbscan}, Calinski-Harabasz Index: {ch\_dbscan}")  
  
 print("\nMetrics for Sklearn DBSCAN:")  
 silhouette\_sklearn\_dbscan, dbi\_sklearn\_dbscan, ch\_sklearn\_dbscan = compute\_metrics(data, sklearn\_dbscan\_labels)  
 print(f"Silhouette Score: {silhouette\_sklearn\_dbscan}, Davies-Bouldin Index: {dbi\_sklearn\_dbscan}, Calinski-Harabasz Index: {ch\_sklearn\_dbscan}")

****输出与结果展示：****

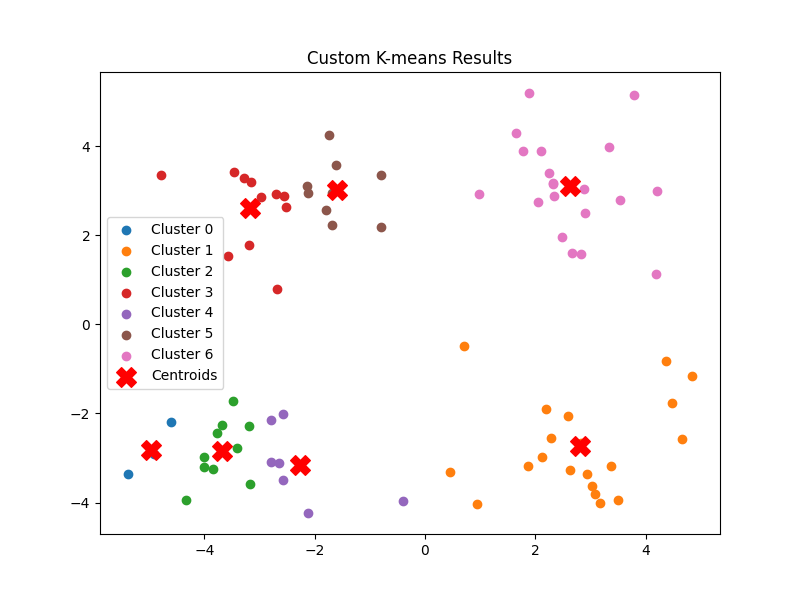


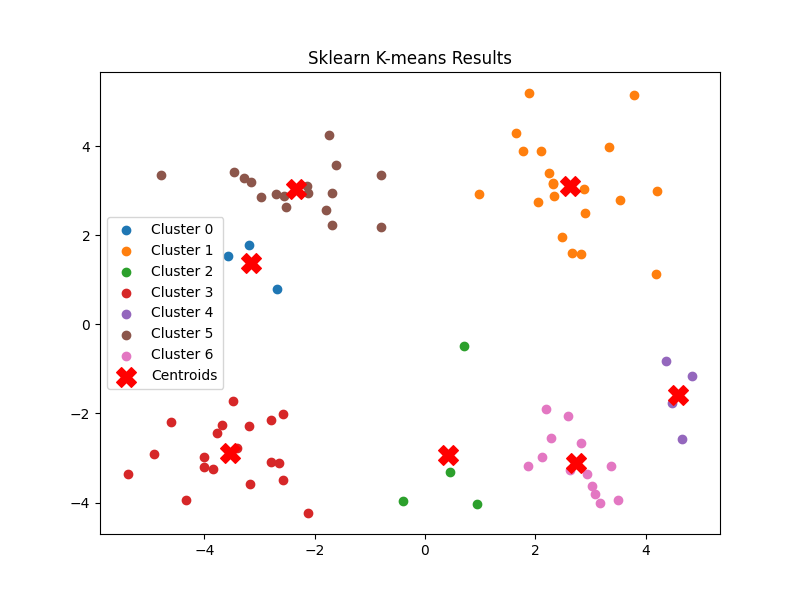
**图5.3 K-distance 图**

**K-distance 图展示了每个数据点与其 第 K 个最近邻（通常为 min\_samples）之间的距离。具体步骤如下：对于数据集中的每个点，计算其与其他所有点的距离，并找出第 K 个最近邻。对这些第 K 个最近邻的距离进行排序，并绘制一个图，X 轴表示数据点的索引或点的排序，Y 轴表示每个点的第 K 个最近邻距离。**

**图中通常会出现一个明显的拐点，代表从密集区域（簇）到稀疏区域的过渡。**

**因为拐点代表了从数据点的密集区域过渡到稀疏区域的边界。在密集区域，数据点彼此接近，eps 值较小；而在稀疏区域，数据点之间的距离增大，说明这些点无法组成一个簇。拐点处的 eps 值能够有效地分隔簇和噪声，避免过度聚合不同簇或错误识别噪声，从而使 DBSCAN 算法能更准确地识别数据结构。**





**图5.4 算法对比1**

**比较两个图中聚类中心的位置，有些中心位置非常接近，而有些则有明显的偏差。例如，在左上角，两个实现的中心位置相似，但在右下角，它们的位置有显著差异。**

**两种实现以相同的方式处理噪声或异常值？可能库里面封装的代码实现更严格地将点分配到聚类中，这会导致不同的聚类结果。**

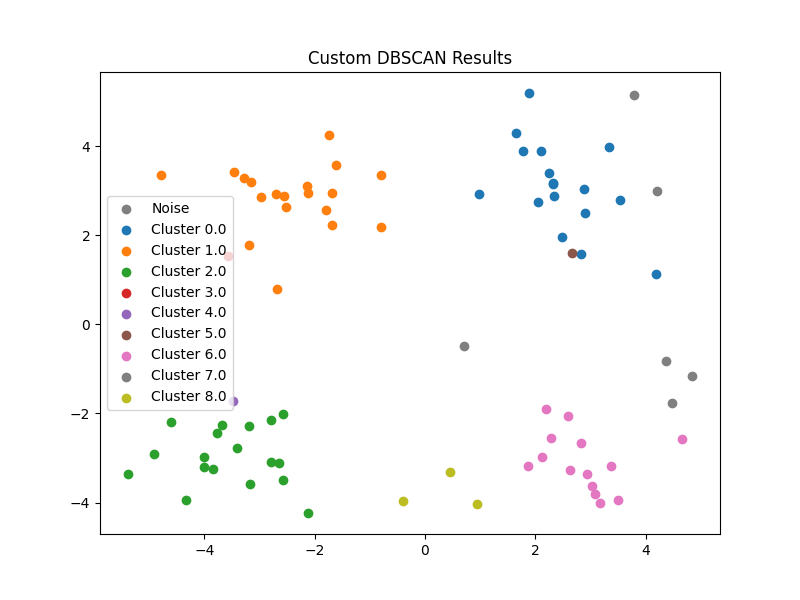
Metrics for Custom K-means:

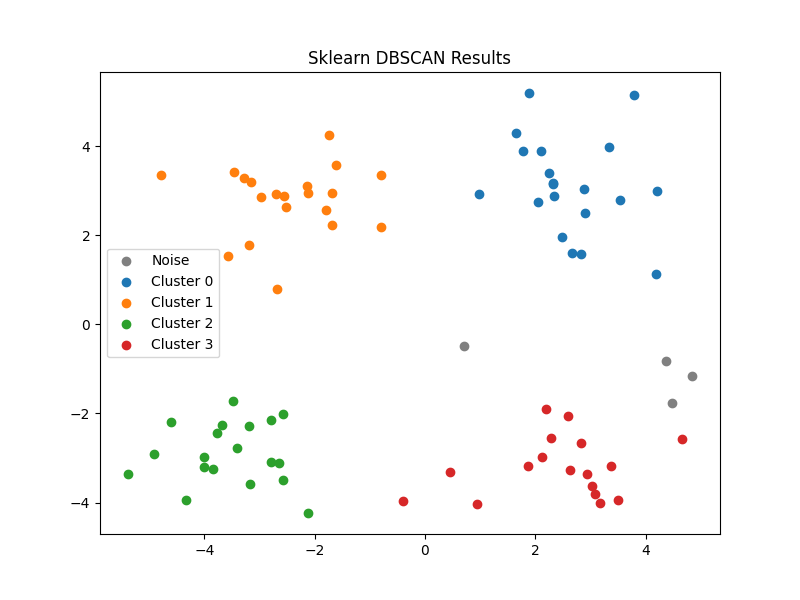
Silhouette Score: 0.4336109700745233, Davies-Bouldin Index: 0.863705426726655, Calinski-Harabasz Index: 137.24990191522303

Metrics for Sklearn K-means:

Silhouette Score: 0.523879116584782, Davies-Bouldin Index: 0.7054019190420534, Calinski-Harabasz Index: 167.02741700237306

**自定义 K-means 的轮廓系数为 0.434，相较于 Sklearn 的 K-means（0.524）明显较低，这表明自定义 K-means 聚类的样本间紧密度和分离度较差，聚类质量相对较低。与此同时，自定义 K-means 的 Davies-Bouldin 指数为 0.864，略高于 Sklearn K-means（0.705），这进一步反映了自定义 K-means 聚类的簇间分离度较差，可能存在一些重叠区域。此外，Calinski-Harabasz 指数（137.25 对比 167.03）同样显示 Sklearn K-means 在聚类的整体结构上表现更优。**





**图5.4 算法对比2**

**从聚类数量上看，自定义实现的聚类数量更多，是由于eps值较小或min\_samples值较高，导致算法识别出更多的独立聚类。两种实现都识别出噪声点，但数量不同，这取决于超参数的设置。**

**自定义实现对数据的形状和密度更为敏感，而sklearn实现在处理密度变化时更为稳健。**

Metrics for Custom DBSCAN:

Silhouette Score: 0.14775757617736446, Davies-Bouldin Index: 0.952215458487467, Calinski-Harabasz Index: 80.11266167079567

Metrics for Sklearn DBSCAN:

Silhouette Score: 0.5928893681004913, Davies-Bouldin Index: 0.7216710521865004, Calinski-Harabasz Index: 190.53645385565227

**对于 DBSCAN 算法，自定义实现和 Sklearn 版本之间的差距则更加显著。自定义 DBSCAN 的轮廓系数为 0.148，远低于 Sklearn 的 0.593，表明自定义 DBSCAN 聚类的紧密度和分离度都较差，聚类效果不好。自定义 DBSCAN 的 Davies-Bouldin 指数为 0.952，高于 Sklearn 的 0.722，说明自定义 DBSCAN 可能存在更多的簇间重叠，导致了更低的聚类质量。Calinski-Harabasz 指数（80.11 对比 190.54）也显示 Sklearn DBSCAN 在聚类的结构和质量上显著优于自定义实现。**

****反思与感悟：****

**（1）综合来看，Sklearn 提供的 K-means 和 DBSCAN 在所有评价指标上均优于自定义的实现，表明其聚类效果更好，算法实现更为高效且稳定。自定义实现可能存在一些优化空间，特别是在聚类结果的紧密度和分离度方面。**

**（2）DBSCAN 的效果极其依赖于 eps 和 min\_samples 的选择。对于 K-means，算法的初始质心选择会影响最终的聚类结果，可能出现局部最优解。可以通过多次随机初始化来解决这个问题。K-means 对噪声点较为敏感，可能会将噪声误分为簇的一部分，导致聚类结果不准确。DBSCAN 在这方面表现更好。**

**5.3 两种聚类算法的对比**

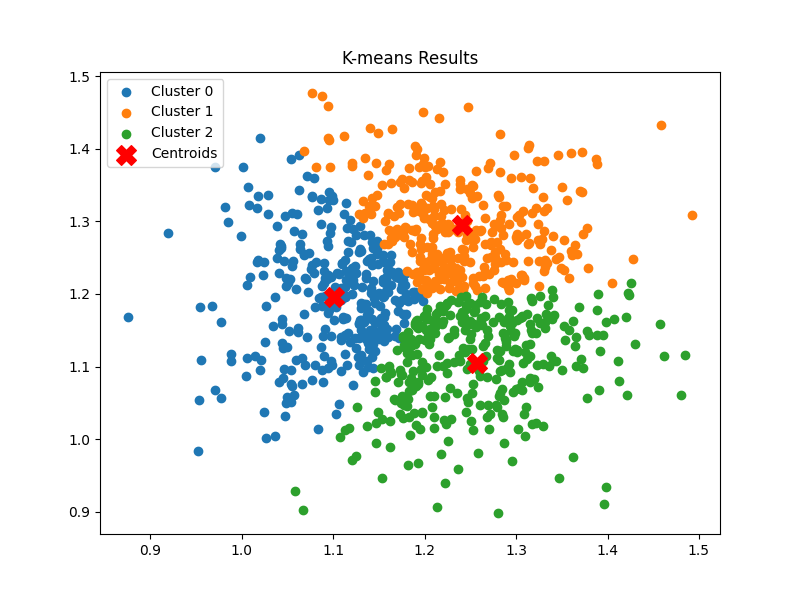
****目标：****展示并对比两种常见的聚类算法：**K-means** 和 **DBSCAN**，通过可视化的方式帮助理解它们在不同数据集上的表现。

****思路：****簇的边界使用凸包（Convex Hull）来绘制，噪声点用灰色标记。

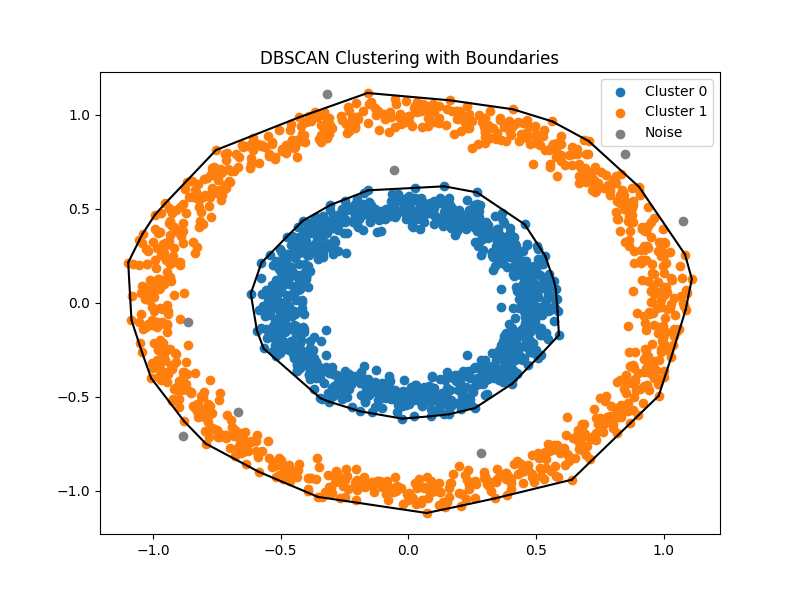
****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn import datasets  
from sklearn.cluster import KMeans, DBSCAN  
from scipy.spatial import ConvexHull  
def visualize\_results(data, labels, centroids=None, algorithm\_name="Clustering"):  
 plt.figure(figsize=(8, 6))  
 unique\_labels = np.unique(labels)  
 for lbl in unique\_labels:  
 if lbl == -1:  
 color = "gray"  
 label = "Noise"  
 else:  
 color = None  
 label = f"Cluster {lbl}"  
 plt.scatter(data[labels == lbl, 0], data[labels == lbl, 1], label=label, c=color)  
  
 if centroids is not None:  
 plt.scatter(centroids[:, 0], centroids[:, 1], s=200, c='red', marker='X', label='Centroids')  
  
 plt.title(f"{algorithm\_name} Results")  
 plt.legend()  
 plt.show()  
  
def plot\_dbscan\_boundaries(data, labels, eps):  
 unique\_labels = set(labels)  
  
 plt.figure(figsize=(8, 6))  
 for label in unique\_labels:  
 if label == -1:  
 plt.scatter(data[labels == label, 0], data[labels == label, 1], c='gray', label='Noise')  
 continue  
 cluster\_points = data[labels == label]  
 plt.scatter(cluster\_points[:, 0], cluster\_points[:, 1], label=f'Cluster {label}')  
 if len(cluster\_points) >= 3:  
 hull = ConvexHull(cluster\_points)  
 for simplex in hull.simplices:  
 plt.plot(cluster\_points[simplex, 0], cluster\_points[simplex, 1], 'k-')  
 plt.title("DBSCAN Clustering with Boundaries")  
 plt.legend()  
 plt.show()  
*# Generate datasets*X\_dbscan, y\_dbscan = datasets.make\_circles(n\_samples=2000, factor=0.5, noise=0.05)  
X\_kmeans, y\_kmeans = datasets.make\_blobs(n\_samples=1000, centers=[[1.2, 1.2]], cluster\_std=0.1)  
kmeans = KMeans(n\_clusters=3, random\_state=66666)  
kmeans\_labels = kmeans.fit\_predict(X\_kmeans)  
kmeans\_centroids = kmeans.cluster\_centers\_  
dbscan = DBSCAN(eps=0.08, min\_samples=10)  
dbscan\_labels = dbscan.fit\_predict(X\_dbscan)  
visualize\_results(X\_kmeans, kmeans\_labels, kmeans\_centroids, "K-means")  
plot\_dbscan\_boundaries(X\_dbscan, dbscan\_labels, eps=0.1)

****输出与结果展示：****



**图5.5 可视化聚类1**



**图5.5 可视化聚类2**

****反思与感悟：****

**不同的聚类算法有不同的适用场景，K-means 更适合处理均匀大小的球形簇，而 DBSCAN 可以处理形状不规则的簇。选择合适的算法是非常重要的。**

**六、回归算法模型评估和调参**

**6.1 手动调参**

****目标：**本任务旨在通过使用sklearn中的多种回归算法，对load\_diabetes数据集进行模型评估与调参，以找到最优的回归模型及其参数配置，进而深入理解不同回归方法的特性、参数调节方式以及模型性能评估方法。**

****数据集：**sklearn.datasets 提供的糖尿病数据集 ，一个内置的[回归分析](https://so.csdn.net/so/search?q=%E5%9B%9E%E5%BD%92%E5%88%86%E6%9E%90&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_52447486/article/details/_blank)数据集。**

**目标值（y）：一年后的疾病进展量化指标，范围从 25 到 346。**

****思路：**手动更改参数列表，代码展示的是第三次调参的结果，已经得到了比较好的效果（和下面的自动调参库对比）。**

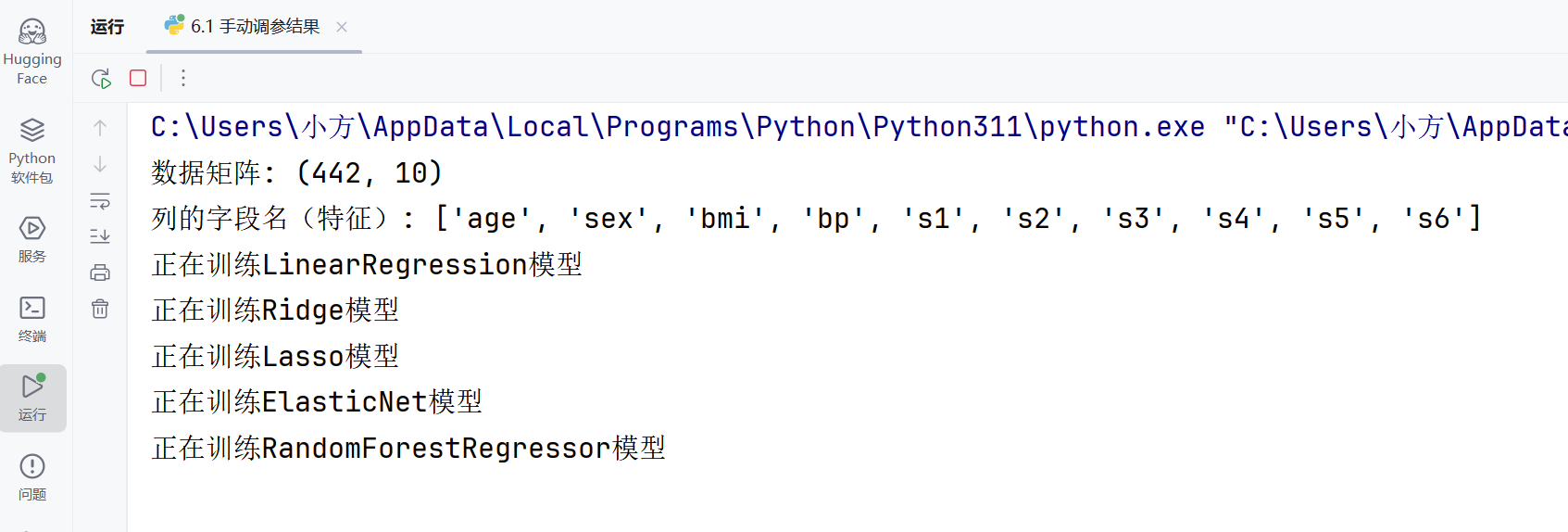
grid\_search = GridSearchCV(model, param\_grid, cv=5, scoring='r2', n\_jobs=-1)

**交叉验证是一种统计方法，用来评估机器学习模型在未知数据上的泛化能力。基本思想是将原始数据分成若干个子集（通常称为“折”），然后多次训练模型，每次用不同的子集作为验证集，其余的数据作为训练集。通过这种方式，可以更可靠地估计模型的表现，并且有助于避免过拟合。GridSearchCV 是 scikit-learn 库中提供的一个工具，用于执行网格搜索（grid search）来寻找给定模型的最佳参数组合。**

****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_diabetes  
from sklearn.model\_selection import train\_test\_split, GridSearchCV  
from sklearn.metrics import mean\_squared\_error, r2\_score  
from sklearn.linear\_model import LinearRegression, Ridge, Lasso, ElasticNet  
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, AdaBoostRegressor  
from sklearn.svm import SVR  
from sklearn.tree import DecisionTreeRegressor  
from sklearn.neighbors import KNeighborsRegressor  
import pandas as pd  
import os  
os.makedirs("C:/Temp", exist\_ok=True)  
os.environ['JOBLIB\_TEMP\_FOLDER'] = "C:/Temp"  
data=load\_diabetes()  
X, y=data.data, data.target  
print("数据矩阵:",X.shape)  
print("列的字段名（特征）:",data.feature\_names)  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=666666)  
models = {  
 "LinearRegression": LinearRegression(),  
 "Ridge": Ridge(),  
 "Lasso": Lasso(),  
 "ElasticNet": ElasticNet(),  
 "RandomForestRegressor": RandomForestRegressor(random\_state=666),  
 "GradientBoostingRegressor": GradientBoostingRegressor(random\_state=666),  
 "AdaBoostRegressor": AdaBoostRegressor(random\_state=666),  
 "SVR": SVR(),  
 "DecisionTreeRegressor": DecisionTreeRegressor(random\_state=666),  
 "KNeighborsRegressor": KNeighborsRegressor()  
}  
params = {  
 "Ridge": {"alpha": [0.05, 0.1, 0.15, 0.2, 0.5]},  
 "Lasso": {"alpha": [0.01, 0.05, 0.1, 0.2, 0.5, 1, 1.5]},  
 "ElasticNet": {"alpha": [0.05, 0.1, 0.2, 0.5], "l1\_ratio": [0.7, 0.8, 0.9]},  
 "RandomForestRegressor": {  
 "n\_estimators": [200, 300, 400, 500],  
 "max\_depth": [10, 15, 20, 25],  
 "min\_samples\_split": [2, 5, 10],  
 "min\_samples\_leaf": [1, 2, 3, 4]  
 },  
 "GradientBoostingRegressor": {  
 "n\_estimators": [100, 150, 200, 300],  
 "learning\_rate": [0.01, 0.05, 0.1],  
 "max\_depth": [3, 4, 5],  
 "min\_samples\_split": [2, 5, 10]  
 },  
 "AdaBoostRegressor": {  
 "n\_estimators": [50, 100, 150, 200],  
 "learning\_rate": [0.01, 0.05, 0.1, 0.15]  
 },  
 "SVR": {  
 "C": [1, 10, 100, 1000, 5000],  
 "gamma": [0.01, 0.05, 0.1, 0.5, 1],  
 "kernel": ["rbf"]  
 },  
 "DecisionTreeRegressor": {  
 "max\_depth": [5, 10, 15, 20, 25, 30],  
 "min\_samples\_split": [2, 5, 10],  
 "min\_samples\_leaf": [1, 2, 3, 4]  
 },  
 "KNeighborsRegressor": {  
 "n\_neighbors": [3, 5, 7, 9, 11],  
 "weights": ["uniform", "distance"],  
 "algorithm": ["auto", "ball\_tree", "kd\_tree", "brute"]  
 }  
}  
results = []  
for model\_name, model in models.items():  
 print(f"正在训练{model\_name}模型")  
 param\_grid = params.get(model\_name, {})  
 grid\_search = GridSearchCV(model, param\_grid, cv=5, scoring='r2', n\_jobs=-1)  
 grid\_search.fit(X\_train, y\_train)  
 best\_model = grid\_search.best\_estimator\_  
 best\_params = grid\_search.best\_params\_  
 cv\_score = grid\_search.best\_score\_  
 y\_pred = best\_model.predict(X\_test)  
 test\_r2 = r2\_score(y\_test, y\_pred)  
 test\_mse = mean\_squared\_error(y\_test, y\_pred)  
 results.append({  
 "Model": model\_name,  
 "最佳参数组合": best\_params,  
 "CV R^2": cv\_score,  
 "Test R^2": test\_r2,  
 "Test 均方误差": test\_mse  
 })  
results\_df = pd.DataFrame(results).sort\_values(by="Test R^2", ascending=False)  
print("\n模型表现总结:")  
print(results\_df)  
plt.figure(figsize=(10, 6))  
plt.barh(results\_df["Model"], results\_df["Test R^2"], color='skyblue')  
plt.xlabel("Test R^2 Score")  
plt.title("Comparison of Models")  
plt.gca().invert\_yaxis()  
plt.show()  
print("\nModel Performance with Best Parameters:")  
for index, row in results\_df.iterrows():  
 print(f"\nModel: {row['Model']}")  
 print(f"Best Params: {row['最佳参数组合']}")  
 print(f"CV R^2: {row['CV R^2']:.4f}")  
 print(f"Test R^2: {row['Test R^2']:.4f}")  
 print(f"Test MSE: {row['Test 均方误差']:.4f}")  
best\_model\_name = results\_df.iloc[0]["Model"]  
best\_model\_params = params.get(best\_model\_name, {})  
print(f"\nFurther tuning for best model: {best\_model\_name}")  
print("Initial Best Params:", results\_df.iloc[0]["最佳参数组合"])

****输出与结果展示：****



模型表现总结:

Model ... Test 均方误差

0 LinearRegression ... 3127.567187

2 Lasso ... 3167.970502

1 Ridge ... 3178.898761

7 SVR ... 3217.787663

9 KNeighborsRegressor ... 3327.203617

6 AdaBoostRegressor ... 3502.884128

5 GradientBoostingRegressor ... 3508.279918

4 RandomForestRegressor ... 3621.887788

3 ElasticNet ... 4079.225613

8 DecisionTreeRegressor ... 4875.728106

Model Performance with Best Parameters:

Model: LinearRegression

Best Params: {}

CV R^2: 0.4670

Test R^2: 0.5146

Test MSE: 3127.5672

Model: Lasso

Best Params: {'alpha': 0.05}

CV R^2: 0.4705

Test R^2: 0.5084

Test MSE: 3167.9705

Model: Ridge

Best Params: {'alpha': 0.05}

CV R^2: 0.4711

Test R^2: 0.5067

Test MSE: 3178.8988

Model: SVR

Best Params: {'C': 1000, 'gamma': 1, 'kernel': 'rbf'}

CV R^2: 0.4726

Test R^2: 0.5006

Test MSE: 3217.7877

Model: KNeighborsRegressor

Best Params: {'algorithm': 'auto', 'n\_neighbors': 11, 'weights': 'distance'}

CV R^2: 0.4219

Test R^2: 0.4836

Test MSE: 3327.2036

Model: AdaBoostRegressor

Best Params: {'learning\_rate': 0.1, 'n\_estimators': 200}

CV R^2: 0.4483

Test R^2: 0.4564

Test MSE: 3502.8841

Model: GradientBoostingRegressor

Best Params: {'learning\_rate': 0.05, 'max\_depth': 3, 'min\_samples\_split': 2, 'n\_estimators': 100}

CV R^2: 0.4437

Test R^2: 0.4555

Test MSE: 3508.2799

Model: RandomForestRegressor

Best Params: {'max\_depth': 15, 'min\_samples\_leaf': 4, 'min\_samples\_split': 2, 'n\_estimators': 300}

CV R^2: 0.4567

Test R^2: 0.4379

Test MSE: 3621.8878

Model: ElasticNet

Best Params: {'alpha': 0.05, 'l1\_ratio': 0.9}

CV R^2: 0.3426

Test R^2: 0.3669

Test MSE: 4079.2256

Model: DecisionTreeRegressor

Best Params: {'max\_depth': 5, 'min\_samples\_leaf': 4, 'min\_samples\_split': 10}

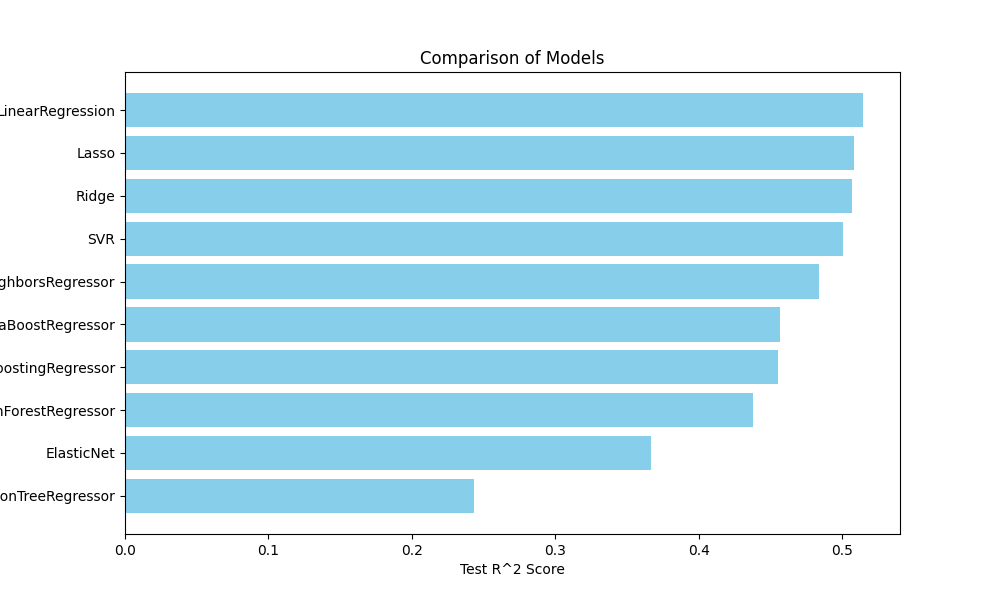
CV R^2: 0.3232

Test R^2: 0.2433

Test MSE: 4875.7281

Further tuning for best model: LinearRegression

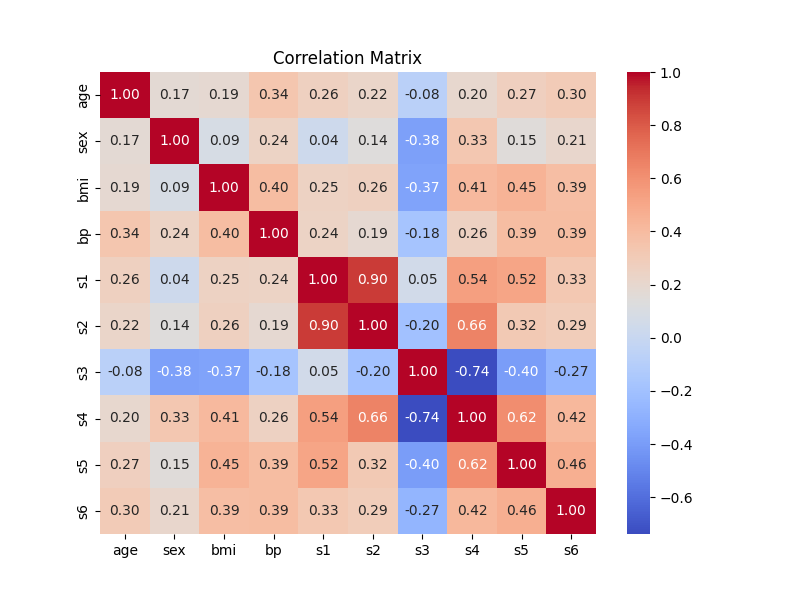
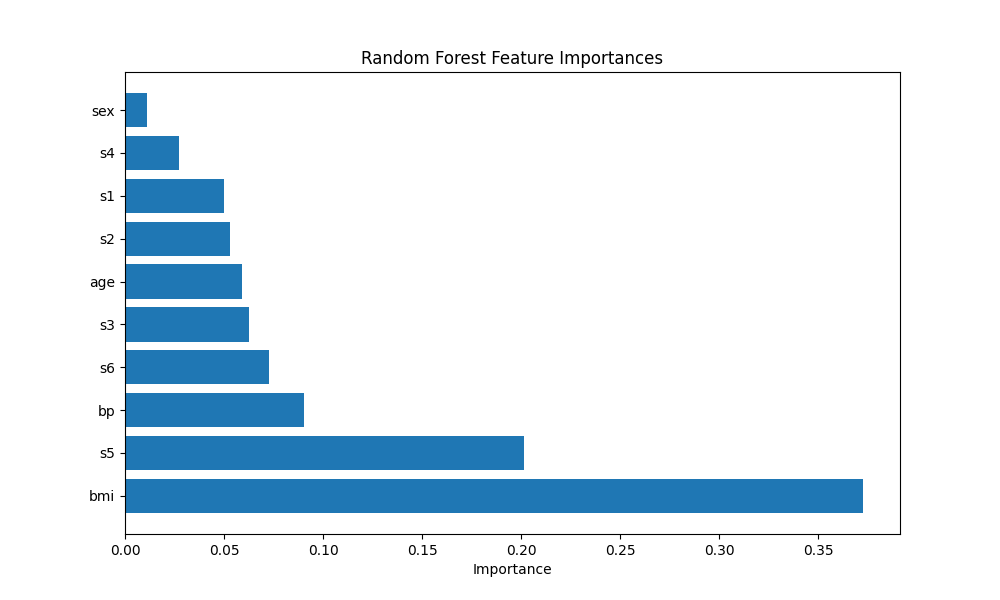
Initial Best Params: {}



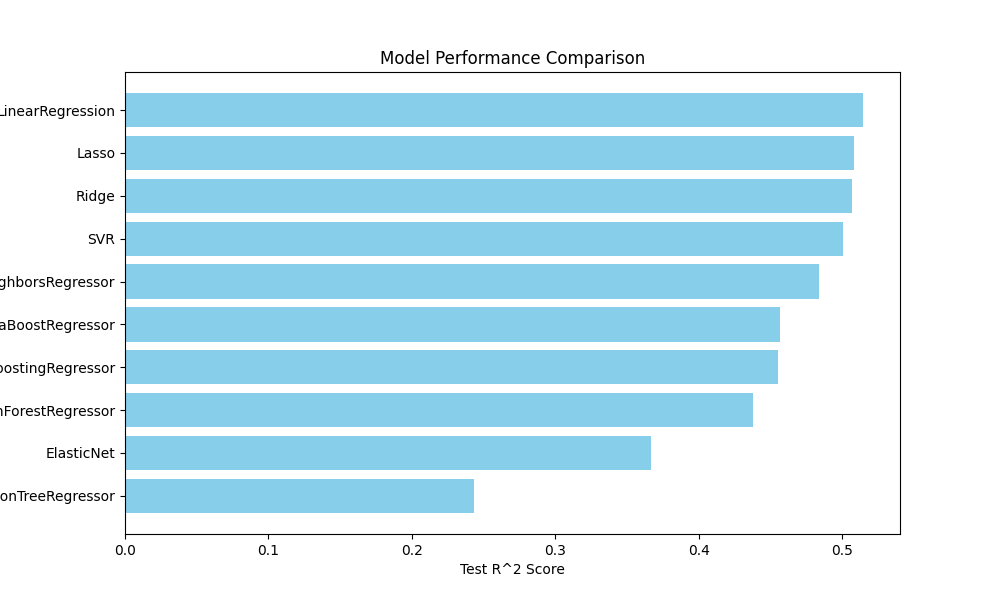
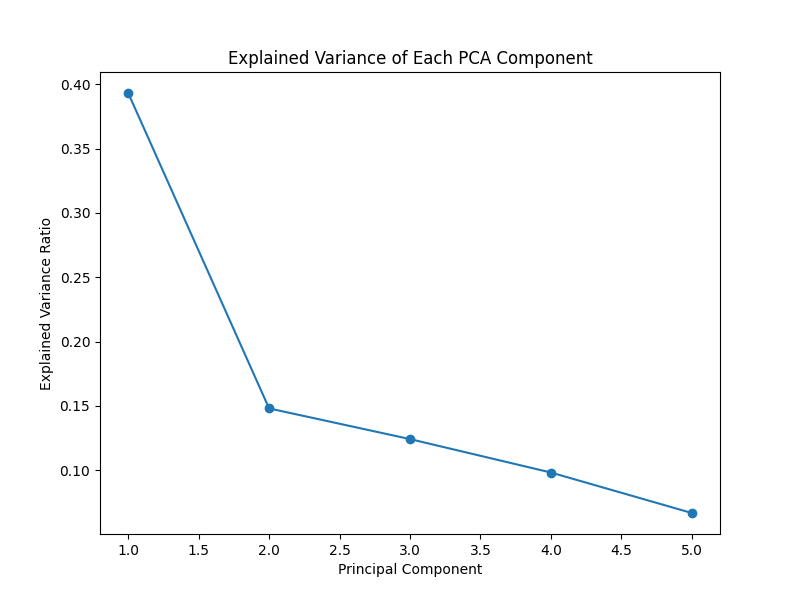
**图6.1 回归算法比较（未改进）**

**可以看到R方都在0.5以下，试着找找原因，发现是因为特征选择的问题。**改进后的代码，加入了特征选取：****

*# Feature Correlation Analysis*feature\_df = pd.DataFrame(X, columns=data.feature\_names)  
correlation\_matrix = feature\_df.corr()  
plt.figure(figsize=(8, 6))  
sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', fmt='.2f')  
plt.title("Correlation Matrix")  
plt.show()  
  
*# Rank Features Based on Random Forest Feature Importance*rf\_model = RandomForestRegressor(random\_state=666)  
rf\_model.fit(X\_train, y\_train)  
importances = rf\_model.feature\_importances\_  
indices = np.argsort(importances)[::-1]  
  
*# Plot Feature Importance*plt.figure(figsize=(10, 6))  
plt.title("Random Forest Feature Importances")  
plt.barh(range(X\_train.shape[1]), importances[indices], align='center')  
plt.yticks(range(X\_train.shape[1]), [data.feature\_names[i] for i in indices])  
plt.xlabel('Importance')  
plt.show()  
  
*# Recursive Feature Elimination (RFE)*print("\nPerforming Recursive Feature Elimination (RFE)...")  
selector = RFE(estimator=LinearRegression(), n\_features\_to\_select=5)  
selector = selector.fit(X\_train, y\_train)  
  
*# RFE Ranking and Selected Features*selected\_features = [data.feature\_names[i] for i in range(len(data.feature\_names)) if selector.support\_[i]]  
print(f"Selected Features after RFE: {selected\_features}")  
  
*# PCA for Dimensionality Reduction (Optional)*pca = PCA(n\_components=5) *# Reduce to 5 components*X\_train\_pca = pca.fit\_transform(X\_train)  
X\_test\_pca = pca.transform(X\_test)  
  
*# Visualize Explained Variance of PCA Components*plt.figure(figsize=(8, 6))  
plt.plot(range(1, len(pca.explained\_variance\_ratio\_) + 1), pca.explained\_variance\_ratio\_, marker='o')  
plt.title('Explained Variance of Each PCA Component')  
plt.xlabel('Principal Component')  
plt.ylabel('Explained Variance Ratio')  
plt.show()

** **

**图6.2 分析特征**

****

**图6.3 重新回归得到的评分**

****可以发现特征选择了之后，R方就超过0.5了，解释效果变得更好了。****

****反思与感悟：****

**表1 常用回归模型参数含义汇总表**

|  |  |  |
| --- | --- | --- |
| 模型 | 可调参数 | 参数含义 |
| Ridge | alpha | 正则化参数，控制模型的复杂度。值越大，正则化越强，模型越简单，防止过拟合。 |
| Lasso | alpha | 正则化参数，控制模型的复杂度。与Ridge类似，**但Lasso会导致一些特征系数变为零**，从而实现特征选择。 |
| ElasticNet | alpha | 正则化参数，结合了Lasso和Ridge的特点，控制模型的复杂度。 |
| l1\_ratio | Lasso与Ridge的权重比例，l1\_ratio = 1表示Lasso，l1\_ratio = 0表示Ridge，其他值表示两者的组合。 |
| RandomForestRegressor | n\_estimators | 森林中树的数量，树木越多，模型越复杂。 |
| max\_depth | 每棵树的最大深度，控制树的复杂度，防止过拟合。 |
| min\_samples\_split | 划分内部节点所需的最小样本数，数值越大，树越简单。 |
| min\_samples\_leaf | 每棵树叶子节点的最小样本数，数值越大，树越简单，减少过拟合的风险。 |
| GradientBoostingRegressor  （梯度提升算法） | n\_estimators | 基学习器的数量（树的数量）。更多的树通常意味着更好的拟合，但也会增加计算量。 |
| learning\_rate | 学习率，控制每棵树的贡献。较小的学习率通常需要更多的树。 |
| max\_depth | 每棵树的最大深度，控制树的复杂度。 |
| min\_samples\_split | 划分内部节点所需的最小样本数，控制树的复杂度。 |
| AdaBoostRegressor | n\_estimators | 基学习器的数量，更多的基学习器可以提高模型的准确性。 |
| learning\_rate | 学习率，控制每棵树的权重，较低的学习率通常能得到更好的效果，但需要更多的树。 |
| SVR | C | 惩罚参数，控制模型对误差的容忍度。C值越大，模型对误差的容忍度越小，更容易过拟合。 |
| gamma | 核函数的参数，控制支持向量的影响范围。较大的gamma值意味着训练样本对结果影响较小，过小的gamma值则会导致欠拟合。 |
| kernel | 核函数类型，用于映射数据到更高维度空间。常见值有linear、rbf（径向基函数）、poly（多项式）等。 |
| DecisionTreeRegressor | max\_depth | 树的最大深度，控制模型复杂度，避免过拟合。 |
| min\_samples\_split | 划分内部节点所需的最小样本数，数值越大，模型越简单。 |
| min\_samples\_leaf | 每个叶子节点的最小样本数，数值越大，模型越简单，减少过拟合的风险。 |
| KNeighborsRegressor | n\_neighbors | 近邻数目，决定了预测时考虑的邻居数量。 |
| weights | （改进）权重类型，uniform表示所有邻居权重相等，distance表示邻居的距离越近，权重越大。 |
| algorithm | 计算最近邻的算法。auto表示自动选择，ball\_tree、kd\_tree、brute等是不同的计算方法。 |

**线性回归模型在本数据集上表现相对较好，而一些复杂的集成模型如随机森林回归等并未取得预期效果，这说明并非模型越复杂性能就一定越好，需根据数据特点选择合适的模型。此外，参数调节是一个细致且关键的环节，合适的参数范围和组合能显著影响模型性能，但寻找最优参数组合往往需要多次尝试与验证，耗费一定时间和计算资源。在后续工作中，可尝试更高效的调参方法，如**自动调参库等**，以提高调参效率。**

**6.2 自动调参\***

****目标：任务导向**，为实现高效率的自动调参，节省计算资源，自学Python的**optuna**调参库。**

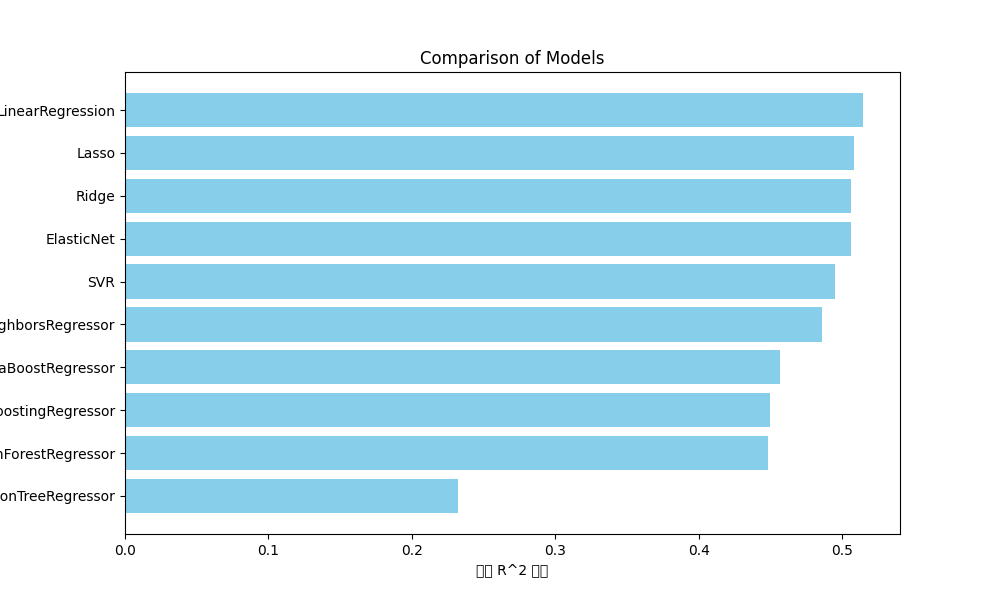
****思路：**学习一下自动调参函数的构造方式。基本格式如下：**

if model\_name == "RandomForestRegressor":  
 n\_estimators = trial.suggest\_int('n\_estimators', 50, 500)  
 max\_depth = trial.suggest\_int('max\_depth', 5, 20)  
 min\_samples\_split = trial.suggest\_int('min\_samples\_split', 2, 10)  
 min\_samples\_leaf = trial.suggest\_int('min\_samples\_leaf', 1, 4)  
 model = RandomForestRegressor(n\_estimators=n\_estimators, max\_depth=max\_depth,  
 min\_samples\_split=min\_samples\_split, min\_samples\_leaf=min\_samples\_leaf)  
 return np.mean(cross\_val\_score(model, X\_train, y\_train, cv=5, scoring='r2'))

****Python实现：**（与6.1基本相同，这里只展示核心代码）**

models = [  
 "LinearRegression", "Ridge", "Lasso", "ElasticNet",  
 "RandomForestRegressor", "GradientBoostingRegressor",  
 "AdaBoostRegressor", "SVR", "DecisionTreeRegressor", "KNeighborsRegressor"  
]  
results = []  
best\_params\_dict = {} *# 用来存储每个模型的最佳参数和表现*for model\_name in models:  
 print(f"正在优化 {model\_name}...")  
 study = optuna.create\_study(direction="maximize")  
 study.optimize(lambda trial: objective(trial, model\_name), n\_trials=50)  
 best\_params = study.best\_params  
 best\_score = study.best\_value  
 best\_params\_dict[model\_name] = {"Best Params": best\_params, "Best R^2": best\_score}  
 if model\_name == "LinearRegression":  
 model = LinearRegression()  
 elif model\_name == "Ridge":  
 model = Ridge(alpha=best\_params['alpha'])  
 elif model\_name == "Lasso":  
 model = Lasso(alpha=best\_params['alpha'])  
 elif model\_name == "ElasticNet":  
 model = ElasticNet(alpha=best\_params['alpha'], l1\_ratio=best\_params['l1\_ratio'])  
 elif model\_name == "RandomForestRegressor":  
 model = RandomForestRegressor(n\_estimators=best\_params['n\_estimators'], max\_depth=best\_params['max\_depth'],  
 min\_samples\_split=best\_params['min\_samples\_split'], min\_samples\_leaf=best\_params['min\_samples\_leaf'])  
 elif model\_name == "GradientBoostingRegressor":  
 model = GradientBoostingRegressor(n\_estimators=best\_params['n\_estimators'], learning\_rate=best\_params['learning\_rate'],  
 max\_depth=best\_params['max\_depth'], min\_samples\_split=best\_params['min\_samples\_split'])  
 elif model\_name == "AdaBoostRegressor":  
 model = AdaBoostRegressor(n\_estimators=best\_params['n\_estimators'], learning\_rate=best\_params['learning\_rate'])  
 elif model\_name == "SVR":  
 model = SVR(C=best\_params['C'], gamma=best\_params['gamma'], kernel=best\_params['kernel'])  
 elif model\_name == "DecisionTreeRegressor":  
 model = DecisionTreeRegressor(max\_depth=best\_params['max\_depth'], min\_samples\_split=best\_params['min\_samples\_split'],  
 min\_samples\_leaf=best\_params['min\_samples\_leaf'])  
 elif model\_name == "KNeighborsRegressor":  
 model = KNeighborsRegressor(n\_neighbors=best\_params['n\_neighbors'], weights=best\_params['weights'], algorithm=best\_params['algorithm'])  
 model.fit(X\_train, y\_train)  
 y\_pred = model.predict(X\_test)  
 test\_r2 = r2\_score(y\_test, y\_pred)  
 test\_mse = mean\_squared\_error(y\_test, y\_pred)  
 results.append({  
 "Model": model\_name,  
 "Best Params": best\_params,  
 "Test R^2": test\_r2,  
 "Test MSE": test\_mse  
 })  
  
results\_df = pd.DataFrame(results).sort\_values(by="Test R^2", ascending=False)  
print("\n模型表现总结:")  
print(results\_df)  
plt.figure(figsize=(10, 6))  
plt.barh(results\_df["Model"], results\_df["Test R^2"], color='skyblue')  
plt.xlabel("测试 R^2 分数")  
plt.title("Comparison of Models")  
plt.gca().invert\_yaxis()  
plt.show()  
print("\n每个模型的最佳参数和表现:")  
for model, params in best\_params\_dict.items():  
 print(f"{model}:\n {params}")

****输出与结果展示：****



**图6.4 回归算法比较（自动调参后）**

每个模型的最佳参数和表现:

LinearRegression:

{'Best Params': {}, 'Best R^2': 0.466958136669638}

Ridge:

{'Best Params': {'alpha': 0.052421975802026545}, 'Best R^2': 0.4710516831432711}

Lasso:

{'Best Params': {'alpha': 0.05156972861431409}, 'Best R^2': 0.4705199285757226}

**ElasticNet:**

**{'Best Params': {'alpha': 0.002468942380259733, 'l1\_ratio': 0.9327409530916666}, 'Best R^2': 0.4710343613596507}**

RandomForestRegressor:

{'Best Params': {'n\_estimators': 223, 'max\_depth': 15, 'min\_samples\_split': 2, 'min\_samples\_leaf': 4}, 'Best R^2': 0.4609279927110788}

GradientBoostingRegressor:

{'Best Params': {'n\_estimators': 329, 'learning\_rate': 0.011602516821846385, 'max\_depth': 3, 'min\_samples\_split': 6}, 'Best R^2': 0.4415784901567169}

AdaBoostRegressor:

{'Best Params': {'n\_estimators': 189, 'learning\_rate': 0.1804548245858442}, 'Best R^2': 0.45973617929434896}

SVR:

{'Best Params': {'C': 467.9915899791573, 'gamma': 0.33638762913205106, 'kernel': 'linear'}, 'Best R^2': 0.4686826811000115}

DecisionTreeRegressor:

{'Best Params': {'max\_depth': 5, 'min\_samples\_split': 2, 'min\_samples\_leaf': 4}, 'Best R^2': 0.31612256467692573}

KNeighborsRegressor:

{'Best Params': {'n\_neighbors': 13, 'weights': 'distance', 'algorithm': 'auto'}, 'Best R^2': 0.4327849400807534}

****反思与感悟：****

**比起网格搜索和随机搜索，**Optuna**[[3]](#footnote-2)最明显的优势就是快。虽然最后的提升效果和人工手动调参差不多，（也有变好的比较显著的，比如弹性网，这是因为它的在数据集上的最优参数太极端了，0.002的阿尔法值，这比较难试出来）但是在整体效率上来看，**Optuna**能够大大减少手动去试的调参时间。**

**七、分类算法模型评估和调参**

****目标**：使用 sklearn.datasets 中的 load\_iris 数据集，调用一些库里面的分类算法并了解其应用场景。**

**数据集：**Iris数据集（安德森鸢尾花卉数据集）。

**目标值（y）**：三种鸢尾花的类别标签，分别为 0（Setosa）、1（Versicolour）、2（Virginica）。

****思路：**与上一题类似，不过要注意，**对于没有 predict\_proba 方法的模型，移除 roc\_auc\_ovr 评分。每个模型的超参数通过 Optuna 调优，并根据每个模型的不同支持进行不同的评分。使用 matplotlib 将各模型的评分进行可视化比较。

****Python实现：****

import optuna  
from sklearn.datasets import load\_iris  
from sklearn.model\_selection import cross\_validate  
from sklearn.preprocessing import StandardScaler  
from sklearn.pipeline import Pipeline  
from sklearn.linear\_model import LogisticRegression  
from sklearn.svm import SVC  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.neural\_network import MLPClassifier  
from sklearn.naive\_bayes import GaussianNB  
from xgboost import XGBClassifier  
import pandas as pd  
import matplotlib.pyplot as plt  
X, y = load\_iris(return\_X\_y=True)  
*# 输出数据结构和基本统计信息*print("数据集X的形状:", X.shape)  
print("数据集y的形状:", y.shape)  
print("特征数据类型:\n", pd.DataFrame(X).dtypes)  
print("数据集基本统计信息:\n", pd.DataFrame(X).describe())  
models = {  
 'LogisticRegression': (LogisticRegression, {  
 'C': optuna.distributions.FloatDistribution(1e-5, 1e5),  
 'solver': optuna.distributions.CategoricalDistribution(['liblinear', 'saga'])  
 }),  
 'SVC': (SVC, {  
 'C': optuna.distributions.FloatDistribution(1e-5, 1e5),  
 'kernel': optuna.distributions.CategoricalDistribution(['linear', 'rbf']),  
 'gamma': optuna.distributions.FloatDistribution(1e-5, 1e5)  
 }),  
 'DecisionTreeClassifier': (DecisionTreeClassifier, {  
 'max\_depth': optuna.distributions.IntDistribution(1, 20),  
 'min\_samples\_split': optuna.distributions.IntDistribution(2, 20),  
 'min\_samples\_leaf': optuna.distributions.IntDistribution(1, 20)  
 }),  
 'RandomForestClassifier': (RandomForestClassifier, {  
 'n\_estimators': optuna.distributions.IntDistribution(50, 300),  
 'max\_depth': optuna.distributions.IntDistribution(3, 20),  
 'min\_samples\_split': optuna.distributions.IntDistribution(2, 20)  
 }),  
 'XGBClassifier': (XGBClassifier, {  
 'n\_estimators': optuna.distributions.IntDistribution(50, 300),  
 'max\_depth': optuna.distributions.IntDistribution(3, 20),  
 'learning\_rate': optuna.distributions.FloatDistribution(0.01, 0.1)  
 }),  
 'KNeighborsClassifier': (KNeighborsClassifier, {  
 'n\_neighbors': optuna.distributions.IntDistribution(3, 30),  
 'weights': optuna.distributions.CategoricalDistribution(['uniform', 'distance']),  
 'p': optuna.distributions.IntDistribution(1, 2)  
 }),  
 'MLPClassifier': (MLPClassifier, {  
 'hidden\_layer\_sizes': optuna.distributions.IntDistribution(50, 500),  
 'activation': optuna.distributions.CategoricalDistribution(['relu', 'tanh']),  
 'alpha': optuna.distributions.FloatDistribution(1e-5, 1e1),  
 'learning\_rate\_init': optuna.distributions.FloatDistribution(1e-5, 1e-1)  
 }),  
 'GaussianNB': (GaussianNB, {}), *#这个没有参数可调* 'AdaBoostClassifier': (AdaBoostClassifier, {  
 'n\_estimators': optuna.distributions.IntDistribution(50, 300),  
 'learning\_rate': optuna.distributions.FloatDistribution(0.01, 1.0)  
 }),  
}  
def objective(trial, model\_name, model\_class, param\_space):  
 *# 使用 trial.suggest\_\* 方法从参数空间中抽取超参数* pipeline = Pipeline([(  
 'scaler', StandardScaler()),  
 ('classifier', model\_class(\*\*{key: trial.suggest\_float(key, value.low, value.high) if isinstance(value,  
 optuna.distributions.FloatDistribution) else trial.suggest\_int(  
 key, value.low, value.high) if isinstance(value,  
 optuna.distributions.IntDistribution) else trial.suggest\_categorical(  
 key, value.choices) for key, value in param\_space.items()})) *# 适配所有类型的分布* ])  
 *# 定义评分方式* scoring = ['accuracy', 'precision\_macro', 'recall\_macro', 'f1\_macro']  
 if hasattr(pipeline.named\_steps['classifier'], 'predict\_proba'):  
 scoring.append('roc\_auc\_ovr')  
 scores = cross\_validate(pipeline, X, y, cv=5, scoring=scoring)  
 return scores['test\_accuracy'].mean()  
study\_results = {}  
for name, (model\_class, param\_space) in models.items():  
 study = optuna.create\_study(direction='maximize')  
 study.optimize(lambda trial: objective(trial, name, model\_class, param\_space), n\_trials=100)  
 study\_results[name] = study.best\_trial  
results\_df = pd.DataFrame({  
 name: {  
 'best\_params': trial.params,  
 'accuracy': trial.value,  
 'precision': cross\_validate(Pipeline([('scaler', StandardScaler()), ('classifier', model\_class(\*\*trial.params))]), X, y, cv=5, scoring='precision\_macro')['test\_score'].mean(),  
 'recall': cross\_validate(Pipeline([('scaler', StandardScaler()), ('classifier', model\_class(\*\*trial.params))]), X, y, cv=5, scoring='recall\_macro')['test\_score'].mean(),  
 'f1': cross\_validate(Pipeline([('scaler', StandardScaler()), ('classifier', model\_class(\*\*trial.params))]), X, y, cv=5, scoring='f1\_macro')['test\_score'].mean(),  
 'roc\_auc': cross\_validate(Pipeline([('scaler', StandardScaler()), ('classifier', model\_class(\*\*trial.params))]), X, y, cv=5, scoring='roc\_auc\_ovr')['test\_score'].mean() if hasattr(model\_class, 'predict\_proba') else None,  
 }  
 for name, (model\_class, \_) in models.items()  
 for trial in [study\_results[name]]  
}).T  
plt.figure(figsize=(12, 8))  
results\_df[['accuracy', 'precision', 'recall', 'f1', 'roc\_auc']].plot(kind='bar', title='Model Performance Comparison')  
plt.xticks(rotation=45, ha="right")  
plt.tight\_layout()  
plt.show()  
print("模型评估结果：")  
print(results\_df)

****输出与结果展示：****

数据集X的形状: (150, 4)

数据集y的形状: (150,)

特征数据类型:

0 float64

1 float64

2 float64

3 float64

数据集基本统计信息:

0 1 2 3

count 150.000000 150.000000 150.000000 150.000000

mean 5.843333 3.057333 3.758000 1.199333

std 0.828066 0.435866 1.765298 0.762238

min 4.300000 2.000000 1.000000 0.100000

25% 5.100000 2.800000 1.600000 0.300000

50% 5.800000 3.000000 4.350000 1.300000

75% 6.400000 3.300000 5.100000 1.800000

max 7.900000 4.400000 6.900000 2.500000

best\_params ... roc\_auc

LogisticRegression {'C': 53192.54213111228, 'solver': 'saga'} ... 0.998667

SVC {'C': 63217.67986691621, 'kernel': 'linear', '... ... NaN

DecisionTreeClassifier {'max\_depth': 3, 'min\_samples\_split': 15, 'min... ... 0.980667

RandomForestClassifier {'n\_estimators': 264, 'max\_depth': 15, 'min\_sa... ... 0.994667

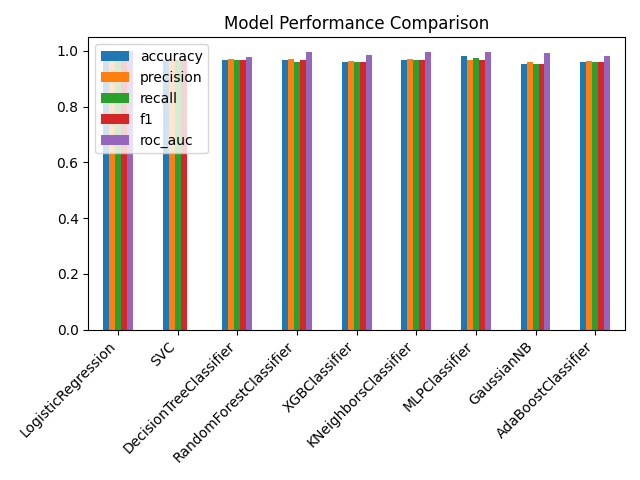
XGBClassifier {'n\_estimators': 68, 'max\_depth': 13, 'learnin... ... 0.984667

KNeighborsClassifier {'n\_neighbors': 8, 'weights': 'uniform', 'p': 2} ... 0.994667

MLPClassifier {'hidden\_layer\_sizes': 87, 'activation': 'tanh... ... 0.994667

GaussianNB {} ... 0.993333

AdaBoostClassifier {'n\_estimators': 246, 'learning\_rate': 0.98586... ... 0.981

**图7.1 分类算法比较（自动调参后）**

根据模型评估结果，**逻辑回归**和**提升**分类器在该数据集上表现最佳，具有接近1的ROC-AUC值，表明它们能够几乎完美地区分不同类别。

****反思与感悟：****

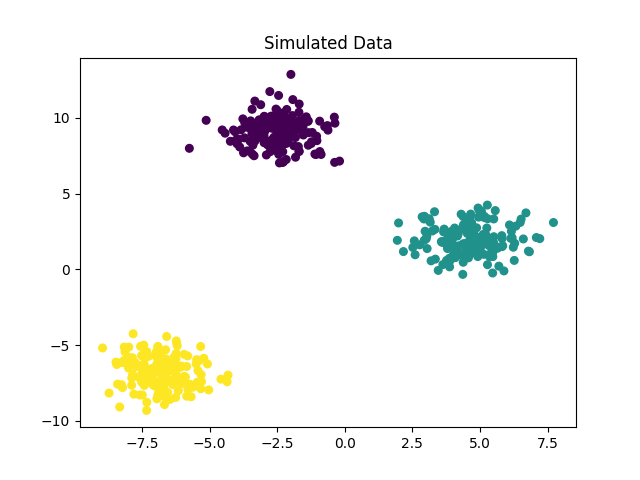
**表2 常用分类模型参数含义汇总表**

|  |  |  |
| --- | --- | --- |
| 模型 | 超参数 | 说明 |
| 支持向量机分类 (SVC) | C | 正则化参数。 |
| kernel | 使用的核函数类型（如 'linear', 'rbf'）。 |
| gamma | 核函数的系数（如 'rbf', 'poly', 'sigmoid'）。 |
| 决策树 (DecisionTreeClassifier) | max\_depth | 树的最大深度。 |
| min\_samples\_split | 分裂一个内部节点所需的最小样本数。 |
| min\_samples\_leaf | 叶节点所需的最小样本数。 |
| 随机森林 (RandomForestClassifier) | n\_estimators | 森林中树的数量。 |
| max\_depth | 树的最大深度。 |
| min\_samples\_split | 分裂一个内部节点所需的最小样本数。 |
| XGBoost (XGBClassifier) | n\_estimators | 提升回合（树）的数量。 |
| max\_depth | 树的最大深度。 |
| learning\_rate | 梯度下降的步长。 |
| K近邻 (KNeighborsClassifier) | n\_neighbors | 用于分类的邻居数量。 |
| weights | 预测时使用的加权函数（'uniform' 或 'distance'）。 |
| p | 曼哈顿距离或欧氏距离的幂参数。 |
| 多层感知器 (MLPClassifier) | hidden\_layer\_sizes | 每层隐藏神经元的数量。 |
| activation | 激活函数（'relu', 'tanh'）。 |
| alpha | L2 正则化项。 |
| learning\_rate\_init | 初始学习率。 |
| 朴素贝叶斯 (GaussianNB) | N/A | 此模型没有超参数可调。 |
| AdaBoost (AdaBoostClassifier) | n\_estimators | 提升回合（基分类器）的数量。 |
| learning\_rate | 每个分类器在集成中的权重 |

**八、聚类任务综合**

****目标：**通过使用不同的聚类算法对仿真数据集进行聚类分析，探索各算法在该数据集上的表现，找出最佳参数配置，并评估聚类效果，以加深对各聚类算法原理、特点及适用场景的理解。**

****思路：**利用make\_blobs生成包含三个簇的仿真数据集，并可视化数据分布，以便直观了解数据特征。分别调用KMeans、AgglomerativeClustering、DBSCAN、GaussianMixture这四种聚类算法，为每个算法设置基本参数，并探索其调优范围，以找到适合该数据集的参数配置。使用adjusted\_rand\_score和silhouette\_score这两个指标来评估各算法的聚类效果，前者衡量聚类结果与真实标签的一致性，后者衡量簇内紧密度和簇间分离度。**



**图8.1 仿真数据，提供了真实标签**

****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import make\_blobs  
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN  
from sklearn.mixture import GaussianMixture  
from sklearn.metrics import adjusted\_rand\_score, silhouette\_score  
from sklearn.preprocessing import StandardScaler  
from scipy.cluster.hierarchy import dendrogram, linkage  
X, y\_true = make\_blobs(n\_samples=500, centers=3, cluster\_std=1.0, random\_state=42)  
plt.scatter(X[:, 0], X[:, 1], c=y\_true, cmap='viridis', s=30)  
plt.title("Simulated Data")  
plt.show()  
scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
kmeans = KMeans(n\_clusters=3, random\_state=42)  
kmeans.fit(X\_scaled)  
y\_kmeans = kmeans.predict(X\_scaled)  
kmeans\_ari = adjusted\_rand\_score(y\_true, y\_kmeans)  
kmeans\_silhouette = silhouette\_score(X\_scaled, y\_kmeans)  
plt.scatter(X\_scaled[:, 0], X\_scaled[:, 1], c=y\_kmeans, cmap='viridis', s=30)  
plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1], c='red', marker='X', s=100, label="Centroids")  
plt.title("KMeans Clustering")  
plt.legend()  
plt.show()  
agg = AgglomerativeClustering(n\_clusters=3)  
y\_agg = agg.fit\_predict(X\_scaled)  
agg\_ari = adjusted\_rand\_score(y\_true, y\_agg)  
agg\_silhouette = silhouette\_score(X\_scaled, y\_agg)  
plt.scatter(X\_scaled[:, 0], X\_scaled[:, 1], c=y\_agg, cmap='viridis', s=30)  
plt.title("Agglomerative Clustering")  
plt.show()  
linked = linkage(X\_scaled, 'ward')  
plt.figure(figsize=(10, 7))  
dendrogram(linked)  
plt.title("Hierarchical Clustering Dendrogram")  
plt.show()  
dbscan = DBSCAN(eps=0.5, min\_samples=5)  
y\_dbscan = dbscan.fit\_predict(X\_scaled)  
dbscan\_ari = adjusted\_rand\_score(y\_true, y\_dbscan) if len(set(y\_dbscan)) > 1 else None  
dbscan\_silhouette = silhouette\_score(X\_scaled, y\_dbscan) if len(set(y\_dbscan)) > 1 else None  
plt.scatter(X\_scaled[:, 0], X\_scaled[:, 1], c=y\_dbscan, cmap='viridis', s=30)  
plt.title("DBSCAN Clustering")  
plt.show()  
gmm = GaussianMixture(n\_components=3, random\_state=42)  
y\_gmm = gmm.fit\_predict(X\_scaled)  
gmm\_ari = adjusted\_rand\_score(y\_true, y\_gmm)  
gmm\_silhouette = silhouette\_score(X\_scaled, y\_gmm)  
plt.scatter(X\_scaled[:, 0], X\_scaled[:, 1], c=y\_gmm, cmap='viridis', s=30)  
x\_min, x\_max = X\_scaled[:, 0].min() - 1, X\_scaled[:, 0].max() + 1  
y\_min, y\_max = X\_scaled[:, 1].min() - 1, X\_scaled[:, 1].max() + 1  
xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 500), np.linspace(y\_min, y\_max, 500))  
Z = gmm.score\_samples(np.c\_[xx.ravel(), yy.ravel()])  
Z = Z.reshape(xx.shape)  
plt.contour(xx, yy, Z, levels=np.linspace(Z.min(), Z.max(), 10), cmap='Blues', alpha=0.6)  
plt.title("Gaussian Mixture Model Clustering")  
plt.show()

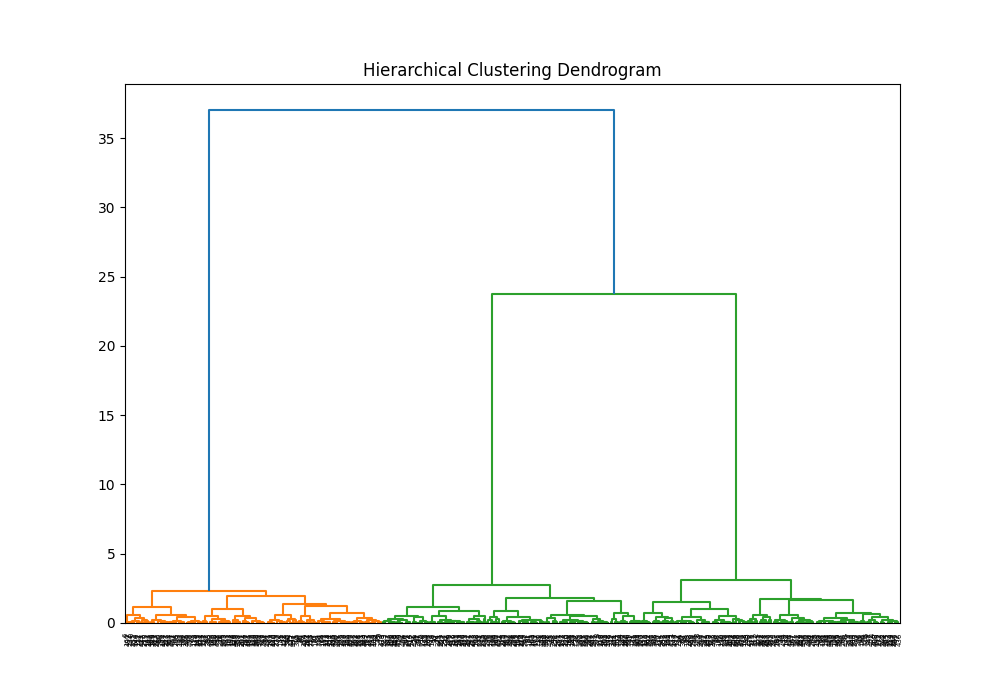
****输出与结果展示：****

KMeans - ARI: 1.0 Silhouette Score: 0.8438960565792794 Best Params: {'n\_clusters': 3}

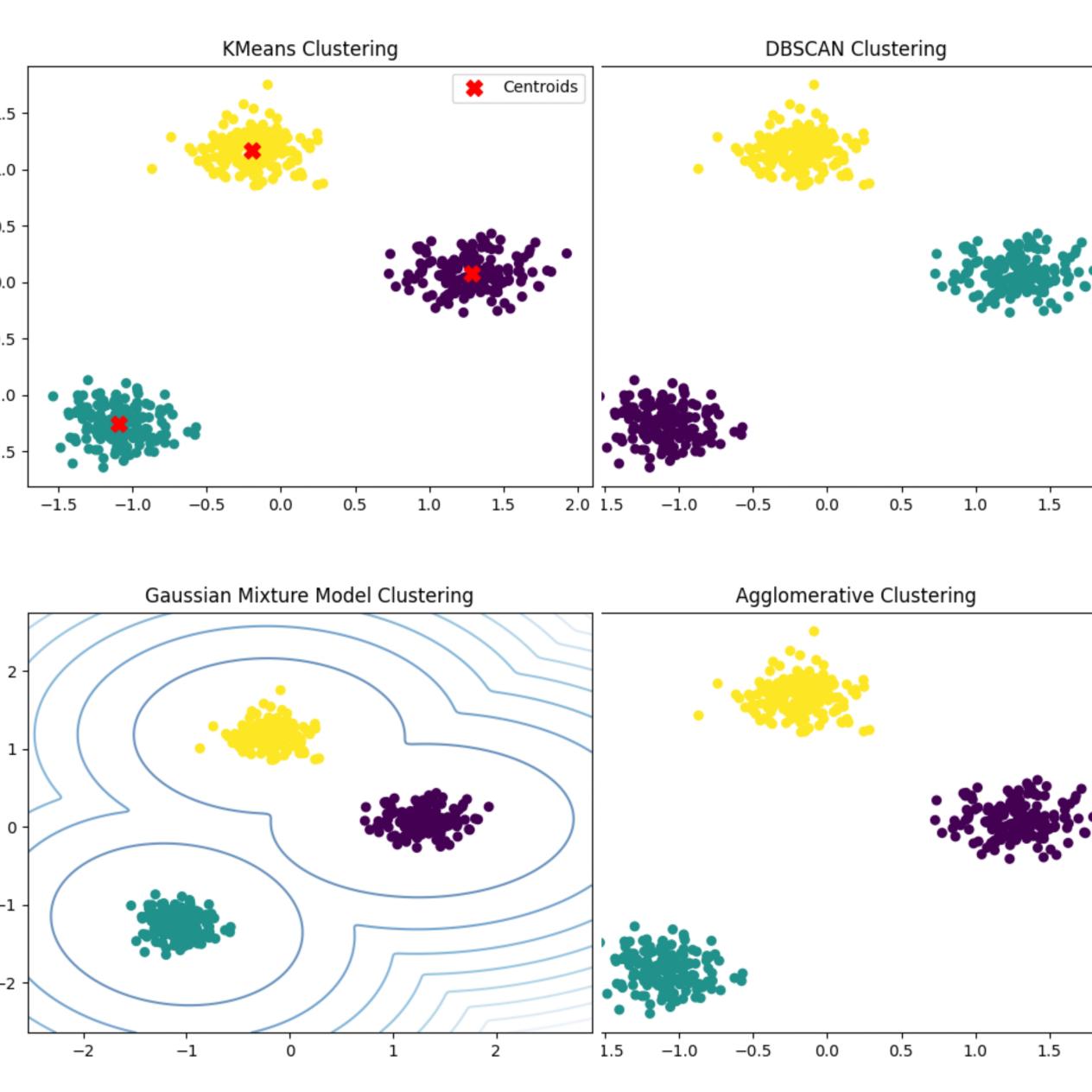
Agglomerative Clustering - ARI: 1.0 Silhouette Score: 0.8438960565792794 Best Params: {'n\_clusters': 3}

DBSCAN - ARI: 1.0 Silhouette Score: 0.8438960565792794 Best Params: {'eps': 0.5, 'min\_samples': 5}

Gaussian Mixture Model - ARI: 1.0 Silhouette Score: 0.8438960565792794 Best Params: {'n\_components': 3}



**图8.2 层次聚类谱系图**

********

**图8.3 聚类可视化**

****反思与感悟：****

**在本次实验中，所有聚类算法（KMeans、Agglomerative Clustering、DBSCAN、Gaussian Mixture Model）都取得了ARI为1.0的理想结果，这表明数据本身的分布非常适合这些聚类方法。因此，得出的聚类效果非常好，但这也意味着数据的复杂性较低，几乎不存在噪声或者簇间重叠的情况。**

**尽管所有模型的表现都非常好，这也提醒我们，不同的聚类算法可能在面对不同类型的数据时会有不同的表现。KMeans和Agglomerative Clustering在数据较为平衡和均匀的情况下表现优秀，但如果数据中有明显的噪声或形状不规则的簇，DBSCAN和Gaussian Mixture Model会更有优势。**

**九、半监督学习**

**9.1 半监督学习算法**

****目标：**通过半监督学习方法（LabelPropagation 和 LabelSpreading）对部分标记的数据集进行学习和预测，评估不同参数设置下的模型性能，并比较两种方法的优劣。**

****数据集： semi\_supervised\_data.csv****

****思路：****

**使用 LabelPropagation 和 LabelSpreading 两种半监督学习算法。**

**调整超参数 kernel 和 max\_iter，观察不同参数组合对模型性能的影响。**

**评估模型的 Accuracy 和 Adjusted Rand Index（ARI）。**

****LabelPropagation 和 LabelSpreading** 是 scikit-learn 中提供的两种半监督学习算法，它们通过构建数据点之间的**亲和图**来传播标签信息。LabelPropagation 采用简单的迭代方式更新标签，适用于低维稠密数据集，但对噪声较为敏感且不保证收敛。相比之下，LabelSpreading 引入了拉普拉斯平滑机制，不仅提高了对噪声的鲁棒性，还确保了算法的收敛性，更适合高维稀疏数据。**

**在使用上，两者都允许用户指定核函数（如 RBF 或 KNN）以定义数据点间的相似度，并提供了参数控制最大迭代次数 (max\_iter) 和收敛阈值 (tol)。LabelSpreading 额外包含一个正则化参数 (alpha) 来平衡新信息与平滑项。**

****Python实现：****

import numpy as np  
import pandas as pd  
import seaborn as sns  
import matplotlib.pyplot as plt  
from sklearn.semi\_supervised import LabelPropagation, LabelSpreading  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, adjusted\_rand\_score  
from sklearn.preprocessing import StandardScaler  
  
data = pd.read\_csv('semi\_supervised\_data.csv')  
  
X = data.iloc[:, :11].values  
y = data['partial\_labels'].values  
  
if np.any(np.isnan(X)) or np.any(np.isnan(y)):  
 print("Data contains NaN values.")  
  
if np.any(np.isinf(X)) or np.any(np.isinf(y)):  
 print("Data contains infinite values.")  
  
scaler = StandardScaler()  
X = scaler.fit\_transform(X)  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  
  
kernels = ['rbf', 'knn']  
max\_iters = [500, 1000]  
  
results = []  
  
for kernel in kernels:  
 for max\_iter in max\_iters:  
 lp\_model = LabelPropagation(kernel=kernel, max\_iter=max\_iter)  
 lp\_model.fit(X\_train, y\_train)  
 lp\_pred = lp\_model.predict(X\_train)  
 lp\_accuracy = accuracy\_score(y\_train[y\_train != -1], lp\_pred[y\_train != -1])  
 lp\_ari = adjusted\_rand\_score(y\_train[y\_train != -1], lp\_pred[y\_train != -1])  
  
 ls\_model = LabelSpreading(kernel=kernel, max\_iter=max\_iter)  
 ls\_model.fit(X\_train, y\_train)  
 ls\_pred = ls\_model.predict(X\_train)  
 ls\_accuracy = accuracy\_score(y\_train[y\_train != -1], ls\_pred[y\_train != -1])  
 ls\_ari = adjusted\_rand\_score(y\_train[y\_train != -1], ls\_pred[y\_train != -1])  
  
 results.append((kernel, max\_iter, lp\_accuracy, ls\_accuracy, lp\_ari, ls\_ari))  
  
results\_df = pd.DataFrame(results, columns=['Kernel', 'Max Iter', 'LP Accuracy', 'LS Accuracy', 'LP ARI', 'LS ARI'])  
  
plt.figure(figsize=(12, 6))  
pivoted\_df = results\_df.pivot(index='Max Iter', columns='Kernel', values='LP Accuracy')  
pivoted\_df.plot(kind='bar', figsize=(10, 6))  
plt.title("Label Propagation Accuracy for Different Kernels and Max Iterations")  
plt.ylabel('Accuracy')  
plt.show()  
  
plt.figure(figsize=(12, 6))  
pivoted\_df = results\_df.pivot(index='Max Iter', columns='Kernel', values='LS Accuracy')  
pivoted\_df.plot(kind='bar', figsize=(10, 6))  
plt.title("Label Spreading Accuracy for Different Kernels and Max Iterations")  
plt.ylabel('Accuracy')  
plt.show()  
  
plt.figure(figsize=(12, 6))  
pivoted\_df = results\_df.pivot(index='Max Iter', columns='Kernel', values='LP ARI')  
pivoted\_df.plot(kind='bar', figsize=(10, 6))  
plt.title("Label Propagation ARI for Different Kernels and Max Iterations")  
plt.ylabel('Adjusted Rand Index (ARI)')  
plt.show()  
  
plt.figure(figsize=(12, 6))  
pivoted\_df = results\_df.pivot(index='Max Iter', columns='Kernel', values='LS ARI')  
pivoted\_df.plot(kind='bar', figsize=(10, 6))  
plt.title("Label Spreading ARI for Different Kernels and Max Iterations")  
plt.ylabel('Adjusted Rand Index (ARI)')  
plt.show()  
  
print("Label Propagation Accuracy:", results\_df['LP Accuracy'].mean())  
print("Label Spreading Accuracy:", results\_df['LS Accuracy'].mean())  
print("Label Propagation ARI:", results\_df['LP ARI'].mean())  
print("Label Spreading ARI:", results\_df['LS ARI'].mean())

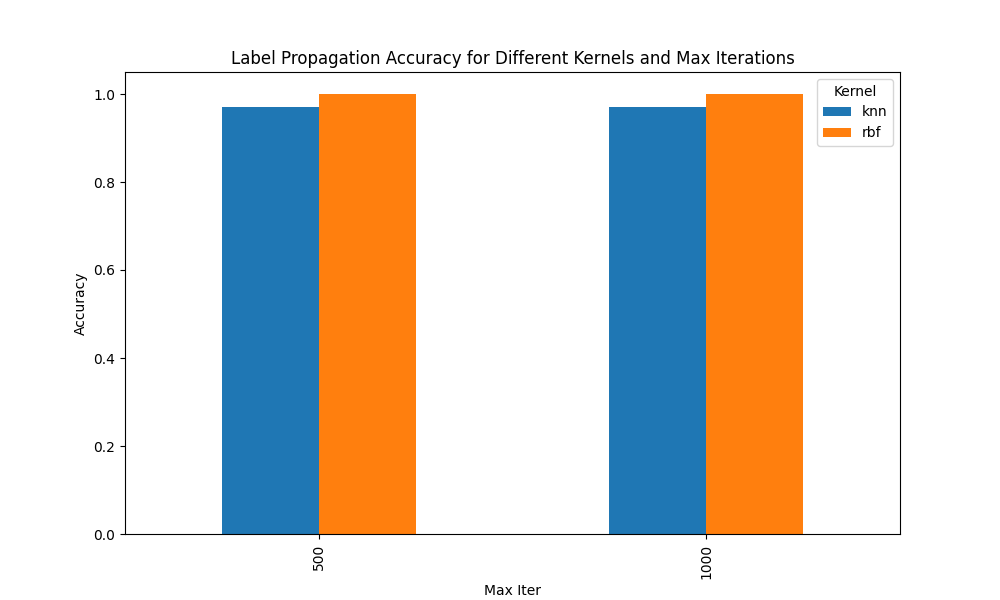
****输出与结果展示：****

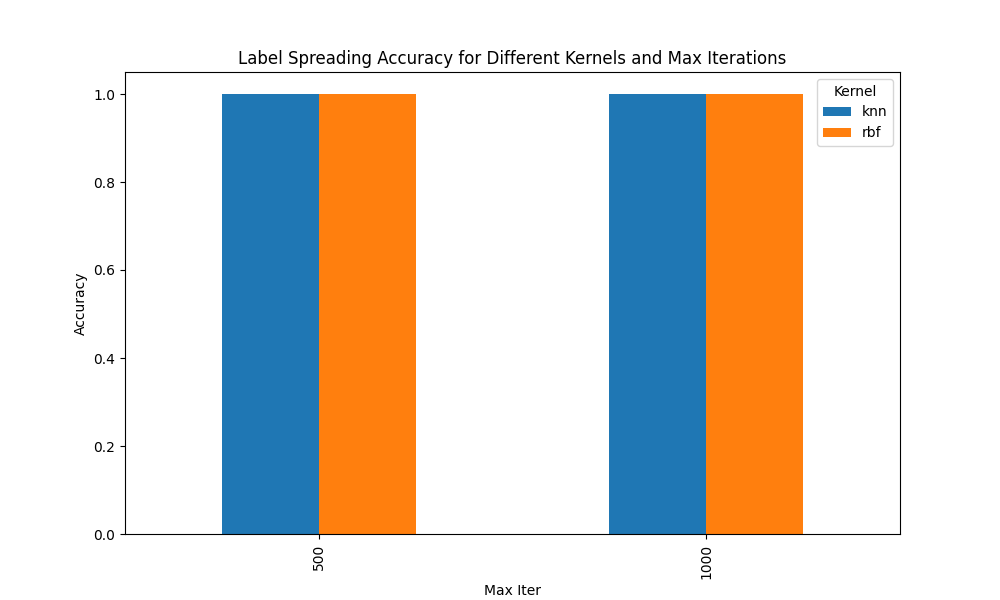
**Label Propagation Accuracy: 0.9848484848484849**

**Label Spreading Accuracy: 1.0**

**Label Propagation ARI: 0.9507877844777282**

**Label Spreading ARI: 1.0**





**图9.1算法对比**

****反思与感悟：****

**本任务使用的数据集具有较为清晰的类别边界和较为均匀的类别分布，这是半监督学习算法在此数据集上表现优异的原因之一。在实际应用中（比如最后一题），数据集的特性更加复杂，如**类别不平衡、噪声数据较多**等，这会对学习算法的性能产生很大的影响。**

**9.2 结合聚类改进处理的半监督学习**

****目标：****通过结合聚类方法（KMeans、DBSCAN、Spectral Clustering）预处理无标签数据，改进半监督学习算法（LabelPropagation 和 LabelSpreading）的性能，并比较改进效果。

****思路：****我们先对无标签数据进行聚类，从聚类结果中生成伪标签（pseudo-labels），然后将这些伪标签与已有的部分标签数据一起传递给半监督学习模型。

****Python实现：****

import numpy as np  
import pandas as pd  
from sklearn.semi\_supervised import LabelPropagation, LabelSpreading  
from sklearn.cluster import KMeans, DBSCAN, SpectralClustering  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import accuracy\_score, adjusted\_rand\_score  
from sklearn.preprocessing import StandardScaler  
  
data = pd.read\_csv('semi\_supervised\_data.csv')  
X = data.iloc[:, :11].values  
y = data['partial\_labels'].values  
  
if np.any(np.isnan(X)) or np.any(np.isnan(y)):  
 print("Data contains NaN values.")  
if np.any(np.isinf(X)) or np.any(np.isinf(y)):  
 print("Data contains infinite values.")  
  
scaler = StandardScaler()  
X = scaler.fit\_transform(X)  
  
def apply\_clustering\_and\_get\_pseudo\_labels(X, y, method='kmeans'):  
 unlabeled\_indices = np.where(y == -1)[0]  
 X\_unlabeled = X[unlabeled\_indices]  
  
 if method == 'kmeans':  
 cluster\_model = KMeans(n\_clusters=len(np.unique(y[y != -1])), random\_state=42)  
 elif method == 'dbscan':  
 cluster\_model = DBSCAN(eps=0.5, min\_samples=5)  
 elif method == 'spectral':  
 cluster\_model = SpectralClustering(n\_clusters=len(np.unique(y[y != -1])), affinity='nearest\_neighbors', random\_state=42)  
 else:  
 raise ValueError("Unsupported clustering method: {}".format(method))  
  
 cluster\_model.fit(X\_unlabeled)  
 pseudo\_labels = cluster\_model.labels\_  
  
 pseudo\_labels[pseudo\_labels == -1] = np.max(pseudo\_labels) + 1  
  
 y[unlabeled\_indices] = pseudo\_labels  
 return y  
  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)  
  
kernels = ['rbf', 'knn']  
max\_iters = [500, 1000]  
  
results = []  
  
clustering\_methods = ['kmeans', 'dbscan', 'spectral']  
for method in clustering\_methods:  
 y\_with\_pseudo\_labels = apply\_clustering\_and\_get\_pseudo\_labels(X\_train.copy(), y\_train.copy(), method)  
  
 for kernel in kernels:  
 for max\_iter in max\_iters:  
 lp\_model = LabelPropagation(kernel=kernel, max\_iter=max\_iter)  
 lp\_model.fit(X\_train, y\_with\_pseudo\_labels)  
 lp\_pred = lp\_model.predict(X\_train)  
 lp\_accuracy = accuracy\_score(y\_train[y\_train != -1], lp\_pred[y\_train != -1])  
 lp\_ari = adjusted\_rand\_score(y\_train[y\_train != -1], lp\_pred[y\_train != -1])  
  
 ls\_model = LabelSpreading(kernel=kernel, max\_iter=max\_iter)  
 ls\_model.fit(X\_train, y\_with\_pseudo\_labels)  
 ls\_pred = ls\_model.predict(X\_train)  
 ls\_accuracy = accuracy\_score(y\_train[y\_train != -1], ls\_pred[y\_train != -1])  
 ls\_ari = adjusted\_rand\_score(y\_train[y\_train != -1], ls\_pred[y\_train != -1])  
  
 results.append((method, kernel, max\_iter, lp\_accuracy, ls\_accuracy, lp\_ari, ls\_ari))  
  
results\_df = pd.DataFrame(results, columns=['Clustering Method', 'Kernel', 'Max Iter', 'LP Accuracy', 'LS Accuracy', 'LP ARI', 'LS ARI'])  
  
print("Results Summary:")  
print(results\_df)  
  
print("\nAverage Results:")  
for method in clustering\_methods:  
 method\_results = results\_df[results\_df['Clustering Method'] == method]  
 avg\_lp\_accuracy = method\_results['LP Accuracy'].mean()  
 avg\_ls\_accuracy = method\_results['LS Accuracy'].mean()  
 avg\_lp\_ari = method\_results['LP ARI'].mean()  
 avg\_ls\_ari = method\_results['LS ARI'].mean()  
  
 print(f"Clustering Method: {method}")  
 print(f" Avg LP Accuracy: {avg\_lp\_accuracy:.4f}")  
 print(f" Avg LS Accuracy: {avg\_ls\_accuracy:.4f}")  
 print(f" Avg LP ARI: {avg\_lp\_ari:.4f}")  
 print(f" Avg LS ARI: {avg\_ls\_ari:.4f}")

****输出与结果展示：****

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Clustering Method | Kernel | Max Iter | LP Accuracy | LS Accuracy | LP ARI | LS ARI |
| kmeans | rbf | 500 | 1 | 1 | 1 | 1 |
| kmeans | rbf | 1000 | 1 | 1 | 1 | 1 |
| kmeans | knn | 500 | 0.121212 | 0.121212 | 0.647519 | 0.647519 |
| kmeans | knn | 1000 | 0.121212 | 0.121212 | 0.647519 | 0.647519 |
| dbscan | rbf | 500 | 1 | 1 | 1 | 1 |
| dbscan | rbf | 1000 | 1 | 1 | 1 | 1 |
| dbscan | knn | 500 | 0.424242 | 0.424242 | 0 | 0 |
| dbscan | knn | 1000 | 0.424242 | 0.424242 | 0 | 0 |
| spectral | rbf | 500 | 1 | 1 | 1 | 1 |
| spectral | rbf | 1000 | 1 | 1 | 1 | 1 |
| spectral | knn | 500 | 0.393939 | 0.393939 | 0.726531 | 0.810295 |
| spectral | knn | 1000 | 0.393939 | 0.393939 | 0.726531 | 0.810295 |

**平均结果**

**Clustering Method: kmeans**

**Avg LP Accuracy: 0.5606**

**Avg LS Accuracy: 0.5606**

**Avg LP ARI: 0.8238**

**Avg LS ARI: 0.8238**

**Clustering Method: dbscan**

**Avg LP Accuracy: 0.7121**

**Avg LS Accuracy: 0.7121**

**Avg LP ARI: 0.5000**

**Avg LS ARI: 0.5000**

**Clustering Method: spectral**

**Avg LP Accuracy: 0.6970**

**Avg LS Accuracy: 0.6970**

**Avg LP ARI: 0.8633**

**Avg LS ARI: 0.9051**

KMeans、DBSCAN 和 Spectral Clustering 三种聚类方法在结合半监督学习算法时表现各异。KMeans 在使用 rbf 核时，LabelPropagation 和 LabelSpreading 均取得了完美的 Accuracy 和 ARI，这表明 KMeans 能够为半监督学习提供高质量的初始标签信息，尤其适用于该数据集。

DBSCAN 同样在 rbf 核下表现出色，但在 knn 核下 Accuracy 较低且 ARI 为 0，说明 DBSCAN 在处理某些特定数据分布时可能不够稳定，特别是在 knn 核下。

Spectral Clustering 在 rbf 核下也取得了完美的 Accuracy 和 ARI，但在 knn 核下 Accuracy 较低，尽管 ARI 仍然较高，这表明 Spectral Clustering 在某些情况下能够提供较好的初始标签信息，但 knn 核可能不适用于该数据集

****反思与感悟：****

通过与9.1对比，发现采用聚类改进不如不用。聚类算法在给未标记数据分配伪标签时，依赖于数据点之间的相似度和聚类结构。尤其是对于密度差异大的数据，像DBSCAN这样基于密度的聚类算法，会将一些真正的标签不相似的数据划分在一起，从而产生不准确的伪标签。这种噪声标签会影响半监督学习算法的表现。半监督学习中的标签传播（Label Propagation）和标签扩展（Label Spreading）算法，假设数据的相似性较高时，标签可以在数据点之间有效传播。如果聚类结果质量较差（例如，伪标签错误或分配不合理），标签传播的效果也会受到影响，从而导致分类性能下降。

同时呢，核函数的选择对半监督学习算法的性能有显著影响。在所有聚类方法下，使用 rbf 核时，LabelPropagation 和 LabelSpreading 均取得了完美的 Accuracy 和 ARI，这表明 rbf 核在该数据集上非常有效，能够捕捉到数据的非线性关系。相比之下，使用 knn 核时，KMeans 和 Spectral Clustering 的 Accuracy 较低，尽管 ARI 仍然较高，这表明 knn 核在某些情况下可能不够稳定，特别是在数据分布较为复杂时。

**十、集成学习**

****目标：****利用集成学习中的 RandomForestClassifier、AdaBoostClassifier 和 GradientBoostingClassifier 模型进行训练和预测，并通过参数调整优化模型性能，最终对模型进行评估和结果展示.

****数据集：****load\_digits（内置的手写数字识别数据集）1797个样本，每个样本包括8\*8像素的图像和一个[0, 9]整数的标签。

****Python实现：****

import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.datasets import load\_digits  
from sklearn.model\_selection import train\_test\_split, GridSearchCV  
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier  
from sklearn.metrics import accuracy\_score  
digits = load\_digits() *# 加载手写数字数据集*X = digits.data *# 特征数据*y = digits.target *# 标签数据*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
models = {  
 "Random Forest": RandomForestClassifier(random\_state=42),  
 "AdaBoost": AdaBoostClassifier(algorithm='SAMME', random\_state=42), *# 使用SAMME算法* "Gradient Boosting": GradientBoostingClassifier(random\_state=42)  
}  
param\_grids = {  
 "Random Forest": {  
 "n\_estimators": [50, 100, 200],  
 "max\_depth": [None, 10, 20, 30],  
 "min\_samples\_split": [2, 5, 10],  
 "min\_samples\_leaf": [1, 2, 4]  
 },  
 "AdaBoost": {  
 "n\_estimators": [50, 100, 200],  
 "learning\_rate": [0.01, 0.1, 1.0]  
 },  
 "Gradient Boosting": {  
 "n\_estimators": [50, 100, 200],  
 "learning\_rate": [0.01, 0.1, 0.5],  
 "max\_depth": [3, 5, 7]  
 }  
}  
  
results = {}  
  
for model\_name, model in models.items():  
 print(f"正在调优 {model\_name} ...")  
 grid\_search = GridSearchCV(model, param\_grids[model\_name], cv=3, scoring='accuracy', n\_jobs=1)  
 grid\_search.fit(X\_train, y\_train)  
 best\_params = grid\_search.best\_params\_  
 best\_model = grid\_search.best\_estimator\_  
  
 *# 训练集和测试集准确率* train\_accuracy = accuracy\_score(y\_train, best\_model.predict(X\_train))  
 test\_accuracy = accuracy\_score(y\_test, best\_model.predict(X\_test))  
  
 results[model\_name] = {  
 "Best Params": best\_params,  
 "Train Accuracy": train\_accuracy,  
 "Test Accuracy": test\_accuracy  
 }  
 print(f"{model\_name} 调优完成.")  
 print(f"最佳参数: {best\_params}")  
 print(f"训练集准确率: {train\_accuracy:.4f}")  
 print(f"测试集准确率: {test\_accuracy:.4f}")  
 print("-" \* 50)  
sorted\_results = sorted(results.items(), key=lambda x: x[1]["Test Accuracy"], reverse=True)  
  
print("\n模型按测试集准确率排序:")  
for model\_name, result in sorted\_results:  
 print(f"{model\_name}: 测试集准确率 = {result['Test Accuracy']:.4f}")  
  
fig, axes = plt.subplots(2, 5, figsize=(10, 5))  
axes = axes.ravel()  
for i in np.arange(10):  
 axes[i].imshow(X\_test[i].reshape(8, 8), cmap="gray")  
 axes[i].set\_title(f"Label: {y\_test[i]}")  
 axes[i].axis("off")  
plt.show()

**输出与结果展示：**

**正在调优 Random Forest ...**

**Random Forest 调优完成.**

**最佳参数: {'max\_depth': 10, 'min\_samples\_leaf': 1, 'min\_samples\_split': 2, 'n\_estimators': 200}**

**训练集准确率: 1.0000**

**测试集准确率: 0.9722**

**--------------------------------------------------**

**正在调优 AdaBoost ...**

**AdaBoost 调优完成.**

**最佳参数: {'learning\_rate': 1.0, 'n\_estimators': 200}**

**训练集准确率: 0.8991**

**测试集准确率: 0.8639**

**--------------------------------------------------**

**正在调优 Gradient Boosting ...**

**Gradient Boosting 调优完成.**

**最佳参数: {'learning\_rate': 0.5, 'max\_depth': 3, 'n\_estimators': 100}**

**训练集准确率: 1.0000**

**测试集准确率: 0.9778**

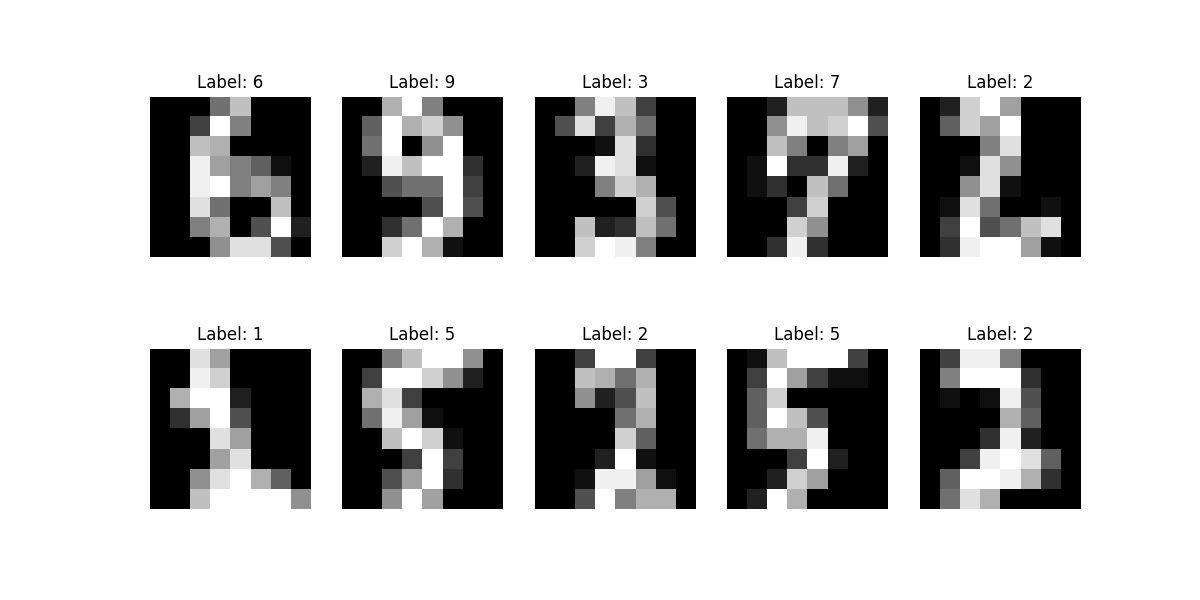
**--------------------------------------------------**

**模型按测试集准确率排序:**

**Gradient Boosting: 测试集准确率 = 0.9778**

**Random Forest: 测试集准确率 = 0.9722**

**AdaBoost: 测试集准确率 = 0.8639**



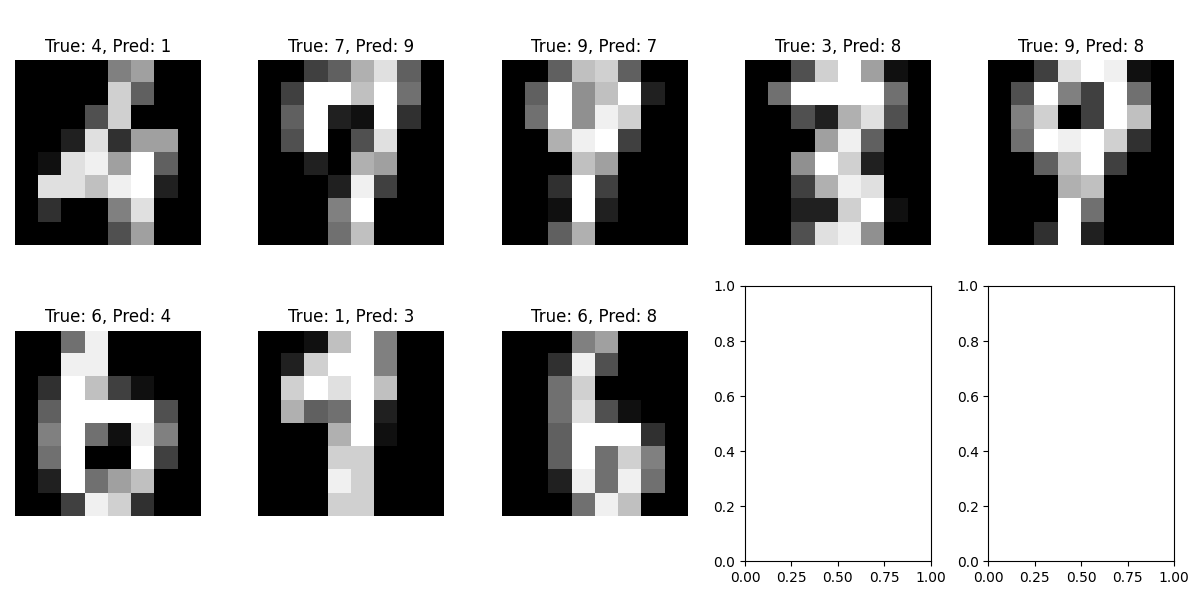
**图10.1 手写数据展示**

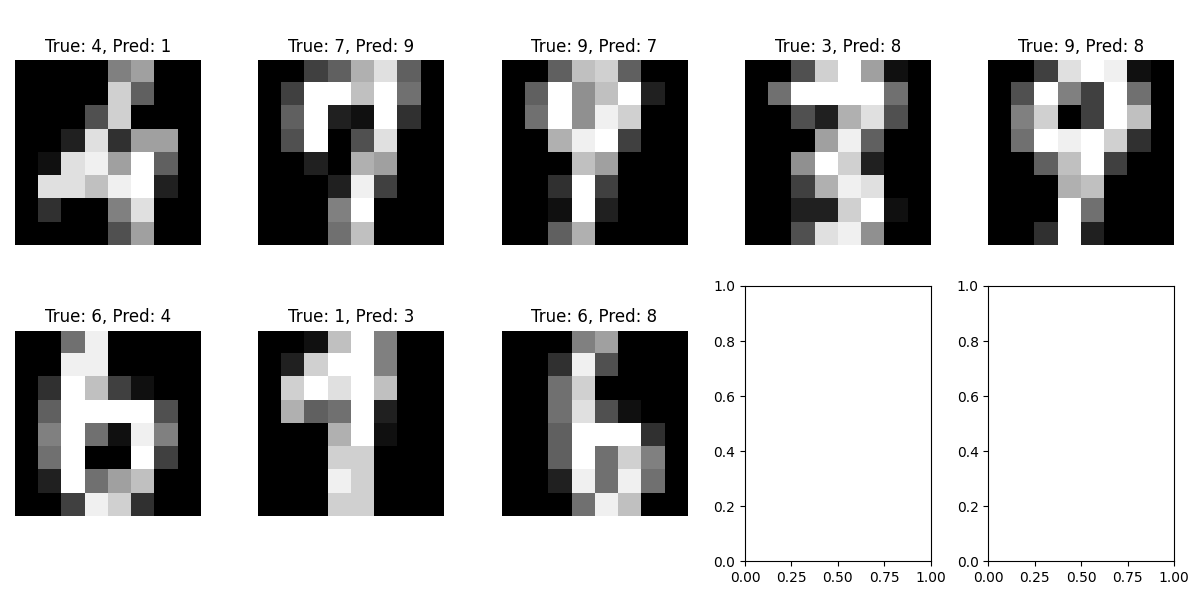
****反思与感悟：****

1. 在实验过程中，可能会遇到一些问题，如模型**训练时间过长**、参数调优效果不理想等。在实验过程中，要保持耐心和细心，不断尝试和调整，以克服遇到的问题，顺利完成实验任务。

（2）后面新增一个改进，输出被错误预测的手写图像，看看长得有多奇怪才会被错分：

*# Error analysis: Misclassified digits*misclassified = []  
for i in range(len(y\_test)):  
 if y\_test[i] != grid\_search.best\_estimator\_.predict(X\_test)[i]:  
 misclassified.append(i)  
  
if misclassified:  
 print(f"\nNumber of misclassified digits: {len(misclassified)}")  
 fig, axes = plt.subplots(2, 5, figsize=(12, 6))  
 axes = axes.ravel()  
 for i, idx in enumerate(misclassified[:10]): *# Display the first 10 misclassified digits* axes[i].imshow(X\_test[idx].reshape(8, 8), cmap="gray")  
 axes[i].set\_title(f"True: {y\_test[idx]}, Pred: {grid\_search.best\_estimator\_.predict(X\_test)[idx]}")  
 axes[i].axis("off")  
 plt.tight\_layout()  
 plt.show()  
else:  
 print("\nNo misclassified digits.")





**图10.2 被错分的8张图片（Random Forest）**

****第一行左二、左三都是很容易被误认的，甚至人也无法分辨，从这里可以看出集成学习的威力，随机森林在训练集上准确率到达100%，在测试集上也没有过拟合，准确率为97.22%。****

**十一、房价预测**

**11.1仅考虑线性回归**

****目标：****仅考虑线性回归模型，分析不同自变量组合对模型拟合效果的影响，并绘制相关图表展示结果，最终输出整体结论。

****思路：****遍历自变量的不同组合，从仅使用一个特征开始，逐步增加特征数量，每次使用线性回归模型进行训练，然后在测试集上进行预测，并计算 R² 分数和均方误差（MSE）作为评估指标，记录不同特征组合下的模型性能。

****数据集：real\_estate\_dataset.csv****

****Python实现：****

*# -\*- coding: utf-8 -\*-*import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.linear\_model import LinearRegression  
from sklearn.model\_selection import train\_test\_split  
from sklearn.metrics import r2\_score, mean\_squared\_error  
data = pd.read\_csv('real\_estate\_dataset.csv')  
X = data[['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors', 'LocationCode', 'YearBuilt']]  
y = data['Price']  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  
r2\_scores = []  
mse\_scores = []  
for i in range(1, len(X.columns) + 1):  
 X\_train\_subset = X\_train.iloc[:, :i]  
 X\_test\_subset = X\_test.iloc[:, :i]  
 model = LinearRegression()  
 model.fit(X\_train\_subset, y\_train)  
 y\_pred = model.predict(X\_test\_subset)  
 r2 = r2\_score(y\_test, y\_pred)  
 mse = mean\_squared\_error(y\_test, y\_pred)  
 r2\_scores.append(r2)  
 mse\_scores.append(mse)  
 print(f"Features used: {X\_train\_subset.columns.tolist()}")  
 print(f"R^2 Score: {r2:.4f}")  
 print(f"Mean Squared Error: {mse:.2f}")  
 print("-" \* 50)

best\_r2\_index = np.argmax(r2\_scores)  
print(f"Best R^2 Score achieved with {best\_r2\_index + 1} features: {r2\_scores[best\_r2\_index]:.4f}")

****输出与结果展示：****

**Features used: ['SquareFeet']**

**R^2 Score: 0.3747**

**Mean Squared Error: 314453331381.57**

**--------------------------------------------------**

**Features used: ['SquareFeet', 'NumBedrooms']**

**R^2 Score: 0.3765**

**Mean Squared Error: 313582904539.33**

**--------------------------------------------------**

**Features used: ['SquareFeet', 'NumBedrooms', 'NumBathrooms']**

**R^2 Score: 0.3739**

**Mean Squared Error: 314873333843.93**

**--------------------------------------------------**

**Features used: ['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors']**

**R^2 Score: 0.3734**

**Mean Squared Error: 315121526690.36**

**--------------------------------------------------**

**Features used: ['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors', 'LocationCode']**

**R^2 Score: 0.8945**

**Mean Squared Error: 53042702431.04**

**--------------------------------------------------**

**Features used: ['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors', 'LocationCode', 'YearBuilt']**

**R^2 Score: 0.8954**

**Mean Squared Error: 52585465767.36**

**--------------------------------------------------**

**Best R^2 Score achieved with 6 features: 0.8954**

****反思与感悟：****

随着特征数量的增加，R² 分数呈现出先上升后趋于平稳的趋势。在仅使用房屋面积（SquareFeet）时，R² 分数为 0.3747，说明仅用面积一个特征对房价的解释能力较弱。当增加到 5 个特征（包括位置代码）时，R² 分数显著提升至 0.8945，表明位置代码是一个非常重要的特征。进一步增加建造年份（YearBuilt）后，R² 分数微幅提升至 0.8954，说明建造年份对模型性能也有一定的贡献，但提升幅度较小。

在当前数据集和线性回归模型下，使用所有 6 个特征能够获得较好的预测效果。

**11.2基于梯度提升决策树的预测**

****目标：**考虑构建合适的模型进行房价预测。**

****思路：**根据11.1，需要读取包含房价的所有相关特征，定义特征变量（**房屋面积、卧室数量、浴室数量、楼层数、位置代码、建造年份等）**和目标变量（房价）。采用一些算法和包，都训练一遍看看什么更好。**

****数据集：real\_estate\_dataset.csv****

****Python实现：****

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error  
import pandas as pd  
from sklearn.model\_selection import train\_test\_split  
from sklearn.linear\_model import LinearRegression, Lasso, Ridge  
from sklearn.tree import DecisionTreeRegressor  
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, StackingRegressor, AdaBoostRegressor  
from sklearn.svm import SVR  
from sklearn.neighbors import KNeighborsRegressor  
from xgboost import XGBRegressor  
from lightgbm import LGBMRegressor  
from catboost import CatBoostRegressor  
import matplotlib.pyplot as plt  
import numpy as np  
  
data = pd.read\_csv('real\_estate\_dataset.csv')  
X = data[['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors', 'LocationCode', 'YearBuilt']]  
y = data['Price']  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.25, random\_state=6666)  
  
models = {  
 'LinearRegression': LinearRegression(),  
 'Lasso': Lasso(),  
 'Ridge': Ridge(),  
 'DecisionTree': DecisionTreeRegressor(),  
 'RandomForest': RandomForestRegressor(),  
 'SVR': SVR(),  
 'KNN': KNeighborsRegressor(),  
 'XGBoost': XGBRegressor(),  
 'LightGBM': LGBMRegressor(),  
 'CatBoost': CatBoostRegressor(verbose=0),  
 'AdaBoost': AdaBoostRegressor()  
}  
model\_rmse = {}  
for model\_name, model in models.items():  
 model.fit(X\_train, y\_train)  
 y\_pred = model.predict(X\_test)  
 rmse = mean\_squared\_error(y\_test, y\_pred, squared=False)  
 model\_rmse[model\_name] = rmse  
for model\_name, rmse in model\_rmse.items():  
 print(f"{model\_name} RMSE: {rmse:.2f}")  
best\_model\_name, best\_rmse = min(model\_rmse.items(), key=lambda x: x[1])  
print(f"\n最佳模型是 {best\_model\_name}，RMSE: {best\_rmse:.2f}")  
best\_model = models[best\_model\_name]  
y\_pred\_best = best\_model.predict(X\_test)  
best\_rmse\_normalized = best\_rmse / 10000 *# 归一化为万元*print(f"\n归一化后的RMSE (单位：万元): {best\_rmse\_normalized:.2f}")  
plt.figure(figsize=(10, 6))  
plt.scatter(y\_test, y\_pred\_best, color='blue', label='Predicted')  
plt.plot([y\_test.min(), y\_test.max()], [y\_test.min(), y\_test.max()], color='red', linestyle='--', label='True')  
plt.xlabel('True Price')  
plt.ylabel('Predict Price')  
plt.title(f'{best\_model\_name} ')  
plt.legend()  
plt.show()  
relative\_error = np.abs((y\_test - y\_pred\_best) / y\_test) \* 100  
print(f"\n相对误差 (以百分比表示): \n{relative\_error}")  
print(f"\n根据评估结果，最佳模型是{best\_model\_name}，其RMSE为{best\_rmse:.2f}，归一化后的RMSE为{best\_rmse\_normalized:.2f}万元。")

****输出与结果展示：****

**LinearRegression RMSE: 211180.93**

**Lasso RMSE: 211180.84**

**Ridge RMSE: 211177.58**

**DecisionTree RMSE: 101740.90**

**RandomForest RMSE: 66610.20**

**SVR RMSE: 722126.26**

**KNN RMSE: 560428.76**

**XGBoost RMSE: 72640.14**

**LightGBM RMSE: 66520.15**

**CatBoost RMSE: 62848.09**

**AdaBoost RMSE: 126214.27**

**最佳模型是 CatBoost，RMSE: 62848.09**

**归一化后的RMSE (单位：万元): 6.28**

**相对误差 (以百分比表示):**

**63 0.310650**

**834 1.590906**

**517 1.382793**

**514 18.698432**

**784 20.916612**

**...**

**366 3.792178**

**985 28.190815**

**13 33.380720**

**325 6.553007**

**191 2.384214**

****反思与感悟：****

**最小的RMSE 为 6.28 万元，表明 CatBoost 模型具有较高的预测精度。**

**相对误差的分布显示，大多数预测结果的误差在 10% 以内，但也有少数样本的误差较大，如样本 514 和 784 的相对误差分别达到 18.698432% 和 20.916612%。这表明模型在某些样本上存在较大的预测偏差，需要进一步分析这些样本的特点，以优化模型性能。**

**11.3 转换为分类问题**

****目标：**通过对房价进行**三分类（高价、中价、低价）**，构建新的标签，使得问题变成分类问 题，针对该问题，构建合理的分类器。**

****思路：**刚拿到题目肯定会想，首先，需要怎么构建新标签，**划分依据**是什么，因为不同的划分会造成后续分类器的性能不同，各个类别需要划分均匀。其次，**如何选择分类算法**，最后是怎么来**评估**在不同房价分布标准下的分类好坏。**

****数据集：real\_estate\_dataset.csv****

****Python实现：****

****划分方式分两种，一种是朴素的先排序再平均分，另外一种是基于平均值和一个西格玛（标准差）。训练分两段代码，第一段是广泛的试模型，最终找到两种划分方式的最优超参数，第二段输出二者性能，这里只展示第二段代码。****

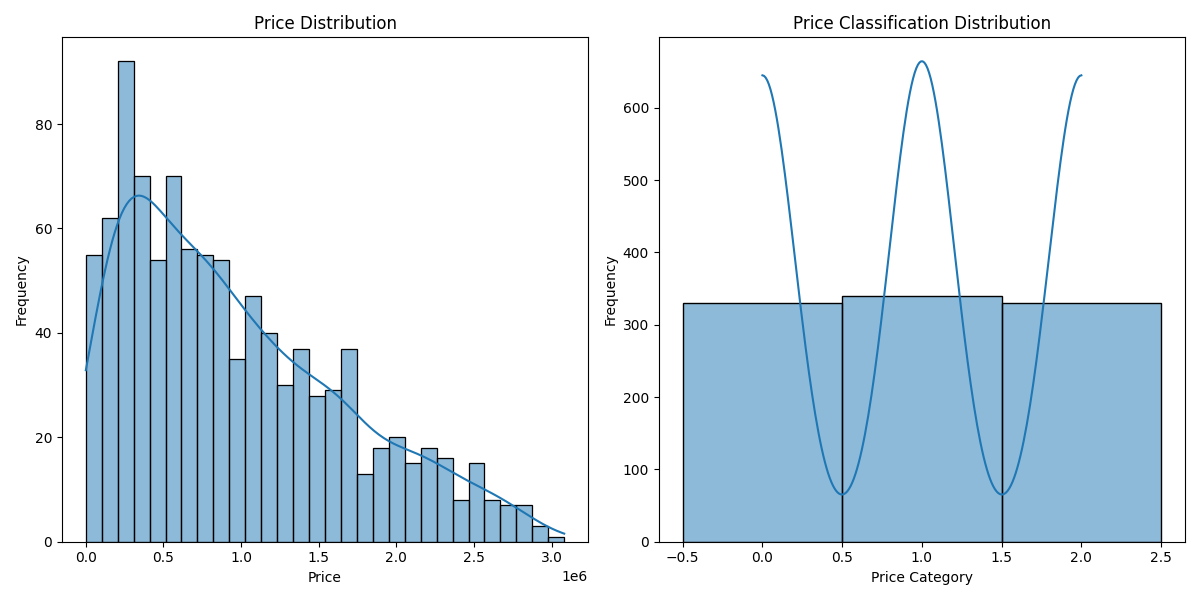
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.model\_selection import train\_test\_split  
from sklearn.preprocessing import StandardScaler  
from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score  
from sklearn.neural\_network import MLPClassifier  
from sklearn.ensemble import GradientBoostingClassifier  
  
data = pd.read\_csv('real\_estate\_dataset.csv')  
X = data[['SquareFeet', 'NumBedrooms', 'NumBathrooms', 'NumFloors', 'LocationCode', 'YearBuilt']]  
y = data['Price']  
  
def classify\_price\_quantile(price, low\_quantile=0.33, high\_quantile=0.67):  
 low\_threshold = y.quantile(low\_quantile)  
 high\_threshold = y.quantile(high\_quantile)  
 if price <= low\_threshold: return 0  
 elif price <= high\_threshold: return 1  
 else: return 2  
  
def classify\_price\_std(price):  
 mean\_price = np.mean(y)  
 std\_price = np.std(y)  
 if price <= mean\_price - std\_price: return 0  
 elif price <= mean\_price + std\_price: return 1  
 else: return 2  
  
def evaluate\_models(y\_class, X\_train, X\_test, y\_train, y\_test, model\_type):  
 model = MLPClassifier(hidden\_layer\_sizes=(150, 75), activation='relu', alpha=0.001, learning\_rate='constant',  
 max\_iter=700, solver='adam', random\_state=42) if model\_type == "NN" else GradientBoostingClassifier(learning\_rate=0.1, max\_depth=5, min\_samples\_split=10, n\_estimators=50, subsample=0.8, random\_state=42)  
 model.fit(X\_train, y\_train)  
 y\_pred = model.predict(X\_test)  
 accuracy = accuracy\_score(y\_test, y\_pred)  
 print(f"\n准确率: {accuracy:.4f}")  
 print("混淆矩阵:")  
 print(confusion\_matrix(y\_test, y\_pred))  
 print("\n分类报告:")  
 print(classification\_report(y\_test, y\_pred, target\_names=['低', '中', '高']))  
 return accuracy  
  
classification\_methods = {  
 "基于分位数(33%和67%)": classify\_price\_quantile,  
 "基于标准差的分类": classify\_price\_std  
}  
  
for method\_name, method\_func in classification\_methods.items():  
 print(f"\n=== 使用 {method\_name} 进行分类 ===")  
 y\_class = y.apply(method\_func)  
  
 plt.figure(figsize=(12, 6))  
 plt.subplot(1, 2, 1)  
 sns.histplot(y, kde=True, bins=30)  
 plt.title('Price Distribution')  
 plt.xlabel('Price')  
 plt.ylabel('Frequency')  
  
 plt.subplot(1, 2, 2)  
 sns.histplot(y\_class, kde=True, bins=3, discrete=True)  
 plt.title('Price Classification Distribution')  
 plt.xlabel('Price Category')  
 plt.ylabel('Frequency')  
  
 plt.tight\_layout()  
 plt.show()  
  
 X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_class, test\_size=0.2, random\_state=42)  
 scaler = StandardScaler()  
 X\_train = scaler.fit\_transform(X\_train)  
 X\_test = scaler.transform(X\_test)  
  
 if method\_name == "基于分位数(33%和67%)":  
 print("使用神经网络进行基于分位数的分类训练")  
 evaluate\_models(y\_class, X\_train, X\_test, y\_train, y\_test, model\_type="NN")  
  
 elif method\_name == "基于标准差的分类":  
 print("使用梯度提升进行基于标准差的分类训练")  
 evaluate\_models(y\_class, X\_train, X\_test, y\_train, y\_test, model\_type="GB")

****输出与结果展示：****

**多模型训练后：**

**Best parameters for Neural Network: {'activation': 'relu', 'alpha': 0.001, 'hidden\_layer\_sizes': (150, 75), 'learning\_rate': 'constant', 'max\_iter': 700, 'solver': 'adam'}**

**Best parameters for Gradient Boosting: {'learning\_rate': 0.1, 'max\_depth': 5, 'min\_samples\_split': 10, 'n\_estimators': 50, 'subsample': 0.8}**



**图11.1 第一种划分方式**

**使用神经网络进行基于分位数的分类训练**

**准确率: 0.9450**

**混淆矩阵:**

**[[53 2 0]**

**[ 4 71 0]**

**[ 0 5 65]]**

**分类报告:**

**precision recall f1-score support**

**低 0.93 0.96 0.95 55**

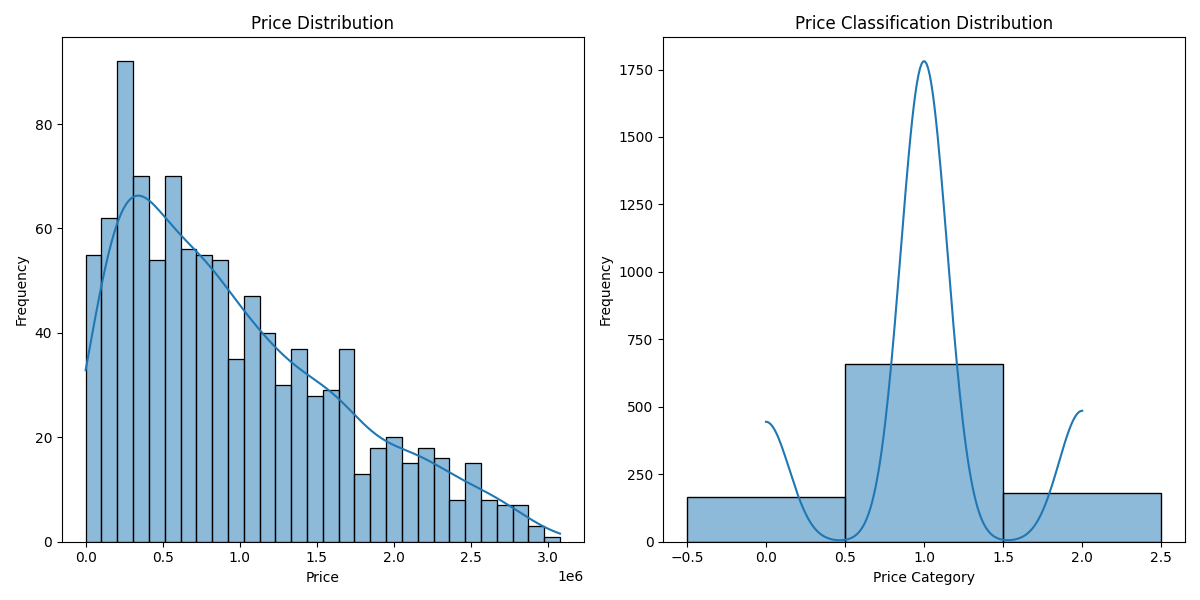
**中 0.91 0.95 0.93 75**

**高 1.00 0.93 0.96 70**

**accuracy 0.94 200**

**macro avg 0.95 0.95 0.95 200**

**weighted avg 0.95 0.94 0.95 200**



**图11.2 第二种划分方式**

**用梯度提升进行基于标准差的分类训练**

**准确率: 0.9450**

**混淆矩阵:**

**[[ 23 4 0]**

**[ 4 132 0]**

**[ 0 3 34]]**

**分类报告:**

**precision recall f1-score support**

**低 0.85 0.85 0.85 27**

**中 0.95 0.97 0.96 136**

**高 1.00 0.92 0.96 37**

**accuracy 0.94 200**

**macro avg 0.93 0.91 0.92 200**

**weighted avg 0.95 0.94 0.94 200**

****反思与感悟：****

**在本次实验中，我们使用了两种不同的标签构建方法（基于分位数和基于标准差）将房价预测问题转化为分类问题，并分别使用了神经网络（MLPClassifier）和梯度提升（GradientBoostingClassifier）进行分类。结果显示，两种方法在不同模型下均取得了较高的准确率，具体如下：**

**神经网络：准确率达到 94.50%，混淆矩阵显示模型在中价和高价类别的分类上表现较为准确，但在低价类别的分类上存在少量误分类。分类报告中的 precision、recall 和 f1-score 均较高，表明模型在各个类别上的分类性能较为均衡。**

**梯度提升：准确率也达到 94.50%，混淆矩阵显示模型在中价和高价类别的分类上表现较为准确，但在低价类别的分类上存在少量误分类。分类报告中的 precision、recall 和 f1-score 均较高，表明模型在各个类别上的分类性能较为均衡。**

**改进方面，可以进行标签构建方法的优化，动态分位数调整，目前使用的分位数（33% 和 67%）是固定的，可以尝试动态调整分位数，使其能够根据数据分布的实际情况自动选择最优的分位数，以更好地处理不同数据集的类别不平衡问题。**

**十二、医疗诊断**

**12.1 最佳模型分类性能优化**

****目标：**通过使用多种机器学习算法解决复杂场景问题，优化分类和回归任务的性能。**

**在分类任务中，重点探索如何**平衡 Precision 和 Recall 的冲突**，确保**“得病”（标签为1）**样例尽可能被召回，同时保证较高的查准率。**

****思路：**先用python进行数据分析，探究一下数据结构，因为提示了数据的缺点，所以需要进行分析。然后呢，检查数据的缺失值并进行必要的预处理；接着，**处理类别不平衡问题，采用SMOTE技术**进行过采样，使得少数类别（得病者）样本数量得到平衡。通过**自动调整阈值**，解决**Precision 和 Recall 的冲突问题。****

****数据集：medical\_diagnosis\_dataset.csv**，10000 条记录，8 个特征（Age, BMI, BloodPressure, Cholesterol, Smoking, FamilyHistory, PhysicalActivity, Disease）**

****Python实现：（部分核心代码）****

smote = SMOTE(sampling\_strategy='auto', random\_state=42)  
X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

*# 使用精度-召回曲线来优化阈值*for model\_name, model in models.items():  
 print(f"调整 {model\_name} 阈值...")if hasattr(model, 'predict\_proba'):  
 y\_prob = model.predict\_proba(X\_test\_scaled)[:, 1]  
 else:  
 *# 对于 Perceptron 使用 decision\_function* y\_prob = model.decision\_function(X\_test\_scaled)  
 y\_prob = (y\_prob - y\_prob.min()) / (y\_prob.max() - y\_prob.min()) *# 将决策函数值归一化到0-1范围* precision, recall, thresholds = precision\_recall\_curve(y\_test, y\_prob)  
  
 *# 选择最优阈值，平衡 Precision 和 Recall* f1\_scores = 2 \* (precision \* recall) / (precision + recall)  
 best\_threshold = thresholds[np.argmax(f1\_scores)] *# 使用最大 F1-score对应的阈值* print(f"{model\_name} 最优阈值: {best\_threshold}")  
  
 *# 使用最佳阈值进行预测* y\_pred\_optimal = (y\_prob >= best\_threshold).astype(int)  
  
 *# 打印优化后的评估指标* print(f"{model\_name} 性能评估（优化阈值）:")  
 print(classification\_report(y\_test, y\_pred\_optimal))

****输出与结果展示：（数据探索和阈值调节部分）****

**RangeIndex: 10000 entries, 0 to 9999**

**Data columns (total 8 columns):**

**# Column Non-Null Count Dtype**

**0 Age 10000 non-null int64**

**1 BMI 10000 non-null float64**

**2 BloodPressure 10000 non-null float64**

**3 Cholesterol 10000 non-null float64**

**4 Smoking 10000 non-null int64**

**5 FamilyHistory 10000 non-null int64**

**6 PhysicalActivity 10000 non-null float64**

**7 Disease 10000 non-null int64**

**疾病标签分布：**

**0 9500**

**1 500**

**Logistic Regression 最优阈值: **0.8734566197663963****

**Logistic Regression 性能评估（优化阈值）:**

**precision recall f1-score support**

**0 0.99 0.99 0.99 1895**

****1 0.89 0.90 0.90 105****

**accuracy 0.99 2000**

**macro avg 0.94 0.95 0.95 2000**

**weighted avg 0.99 0.99 0.99 2000**

**AdaBoost 最优阈值: 0.5135422041576688**

**AdaBoost 性能评估（优化阈值）:**

**precision recall f1-score support**

**0 0.99 0.99 0.99 1895**

**1 0.90 0.84 0.87 105**

**accuracy 0.99 2000**

**macro avg 0.94 0.92 0.93 2000**

**weighted avg 0.99 0.99 0.99 2000**

**Perceptron 最优阈值: 0.7873935046902775**

**Perceptron 性能评估（优化阈值）:**

**precision recall f1-score support**

**0 0.99 0.99 0.99 1895**

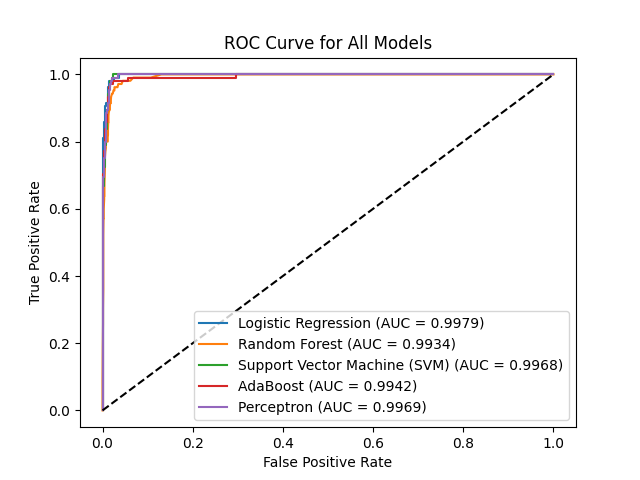
**1 0.85 0.89 0.87 105**

**accuracy 0.99 2000**

**macro avg 0.92 0.94 0.93 2000**

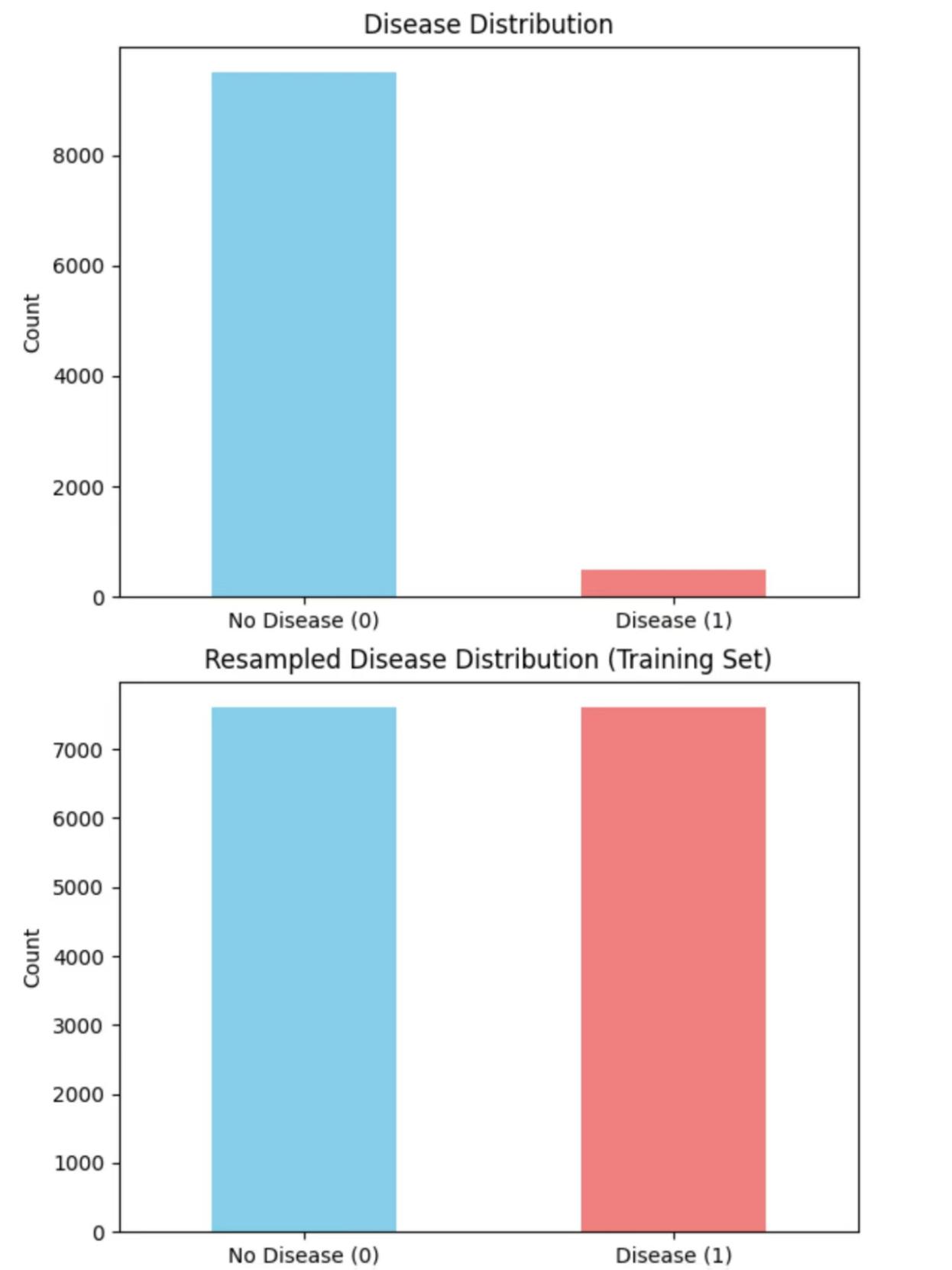
**weighted avg 0.99 0.99 0.99 2000**

****反思与感悟：****

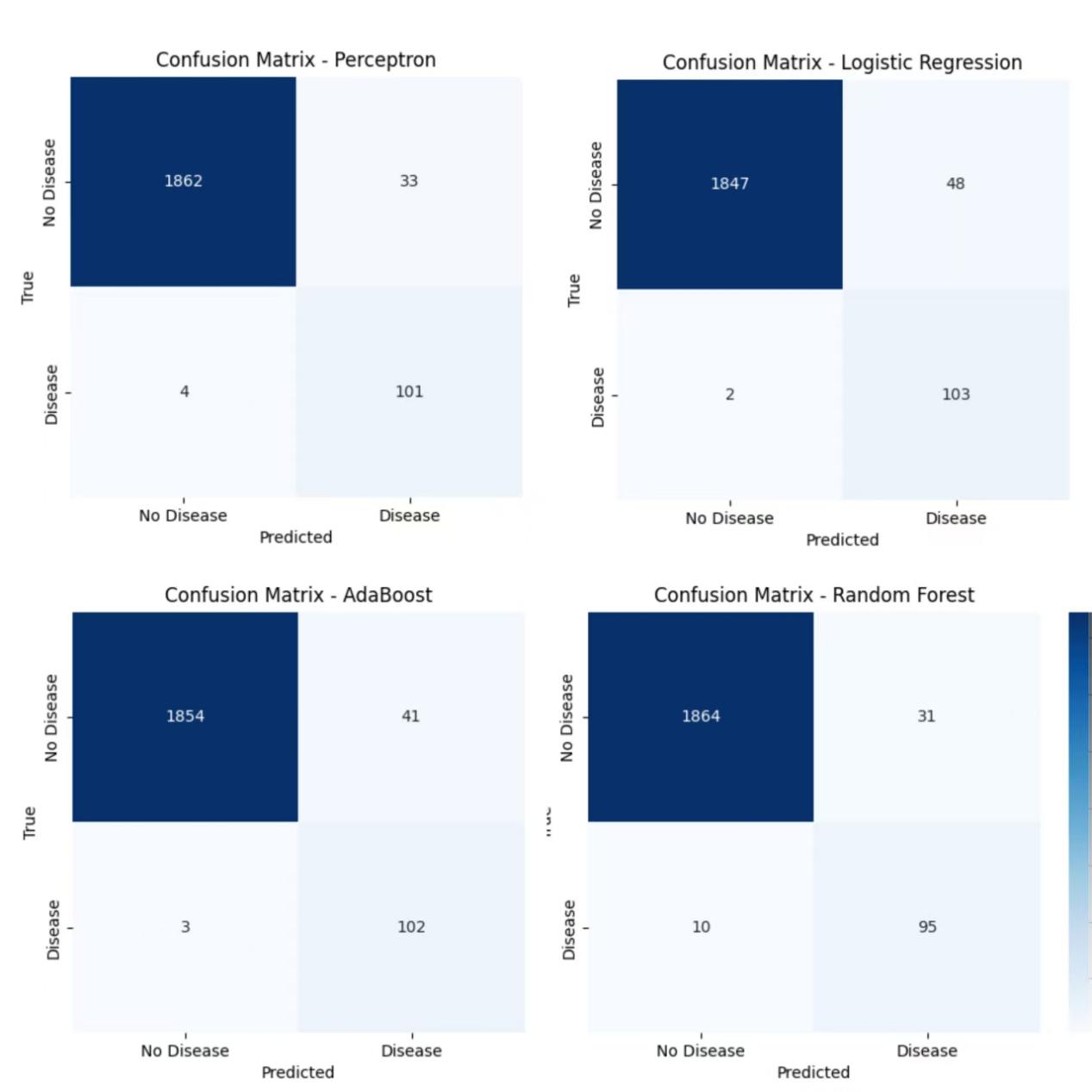


**图12.1可视化结果如混淆矩阵和 ROC 曲线有助于直观分析模型效果**

**在本次实验中，Logistic Regression 模型在调优后的性能表现最为出色，特别是在准确率、精确率、召回率和 F1-score 等指标上均表现出色。因此，推荐使用**阈值改变后的 Logistic Regression 作为最优模型**，用于医疗诊断任务。**

********

**图12.2 解决类别不平衡**

********

**图12.3 各个算法混淆矩阵**

****最终选择的最优模型为阈值改变后的 Logistic Regression。其F1值为0.9，对患病类（“1”）的召回率有90%，精确率有89%。****

**12.2 使用复杂数据同时完成分类和回归任务**

****目标：解决类别不平衡、数据缺失等一系列的问题，完成回归&分类。****

****数据集：complex\_dataset.csv****

1. ****数据分析与问题诊断****

****Python实现：****

print("Data Info:")  
print(data.info()) *# Check data types and missing values*print("\nMissing Values:")  
print(data.isnull().sum()) *# Check how many missing values are present in each column*print("\nDescriptive Statistics:")  
print(data.describe()) *# Get a summary of numerical features*

****输出与结果展示：****

****Missing Values:****

**Feature\_0 500**

**Feature\_1 500**

**Feature\_2 500**

**Feature\_3 500**

**Feature\_4 500**

**Feature\_5 500**

**Feature\_6 500**

**Feature\_7 500**

**Feature\_8 500**

**Feature\_9 500**

**Feature\_10 0**

**Feature\_11 0**

**Feature\_12 0**

**Feature\_13 0**

**Feature\_14 0**

**Feature\_15 0**

**Feature\_16 0**

**Feature\_17 0**

**Feature\_18 0**

**Feature\_19 0**

**Category\_A 0**

**Category\_B 0**

**Category\_C 0**

**Time\_Feature 0**

**Target\_Classification 0**

**Target\_Regression 0**

****Descriptive Statistics:****

**Feature\_0 Feature\_1 ... Target\_Classification Target\_Regression**

**count 4500.000000 4500.000000 ... 5000.000000 5000.000000**

**mean -0.009249 -0.004138 ... 0.054600 5004.813892**

**std 2.879698 0.992705 ... 0.227221 1333.026703**

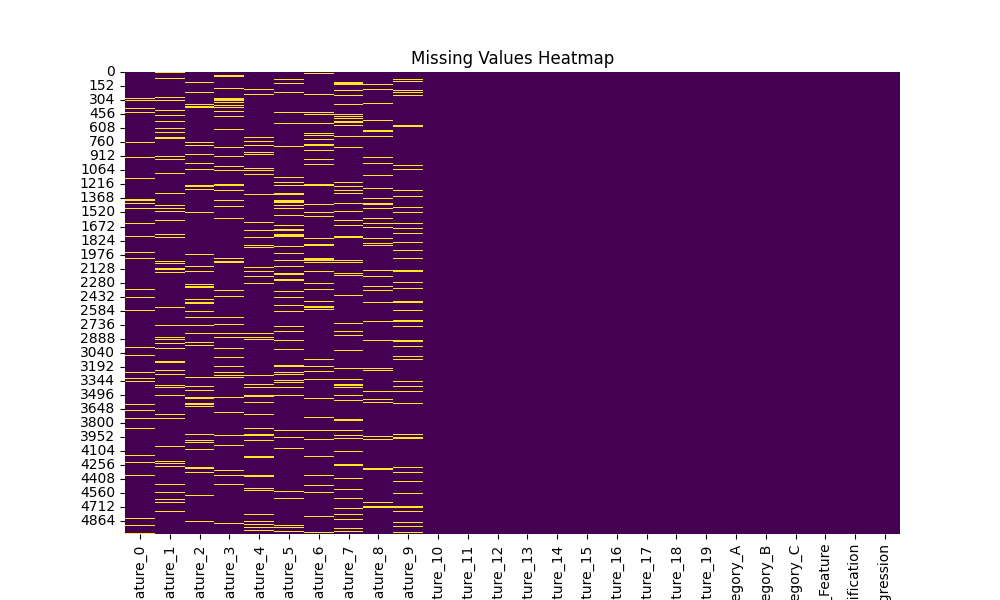
**min -10.438615 -3.940008 ... 0.000000 733.891464**

**25% -1.928304 -0.677341 ... 0.000000 4126.051746**

**50% 0.103706 -0.007290 ... 0.000000 4996.301309**

**75% 1.950082 0.646945 ... 0.000000 5896.191483**

**max 10.083057 3.536048 ... 1.000000 9806.691946**



**图12.4 缺失值分布**

****1.数据集概述****

**数据集大小：5000 条记录，特征数：26 个特征**

**16 个特征（Feature\_10 到 Feature\_19, Category\_A, Category\_B, Category\_C, Time\_Feature, Target\_Classification, Target\_Regression）没有缺失值**

**Target\_Classification：分类目标变量**

**Target\_Regression：回归目标变量**

****10 个特征（Feature\_0 到 Feature\_9）每列缺失 500 个值**，占总记录数的 10%。这些缺失值需要合理填充，以避免影响模型的训练和预测性能。**

**从描述性统计结果来看，部分特征（如 Feature\_0 到 Feature\_9）的分布可能较为广泛，存在较大的标准差（例如 Feature\_0 的标准差为 2.879698），这可能表明数据存在一定的**偏态或异常值**。**

**Target\_Classification：分类目标变量的**分布较为不平衡**，可能需要进行类别平衡处理。**

**Target\_Regression：回归目标变量的**分布范围较广**（从 733.891464 到 9806.691946），需要进行数据变换（如对数变换）以提高模型的拟合效果。**

**部分特征的最小值和最大值可能表明存在**异常值**。例如，Feature\_0 的最小值为 -10.438615，最大值为 10.083057，这表明数据中存在一些**极端值**，需要进一步检查和处理。需要检查特征之间的**相关性**，以避免多重共线性问题。高度相关的特征可能会导致模型的不稳定性和过拟合。对于回归任务，特征的尺度差异较大，需要进行**标准化处理**，以提高模型的收敛速度和预测性能。**

****2.解决方案****

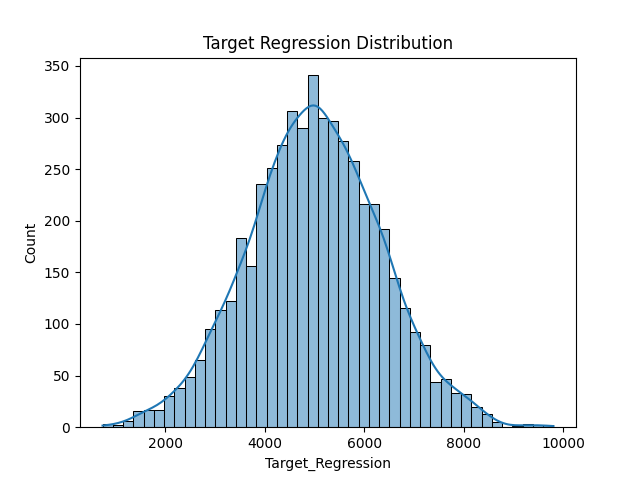
**MICE填充：使用其他变量的信息来预测缺失值，通过多次迭代来改善估计。这种方法可以生成多个完整数据集，适用于复杂的数据集。**

**对数变换：对目标变量 Target\_Regression 进行对数变换，以减少数据的偏态和异常值的影响。**

**Z-score方法：计算每个特征的Z-score，将Z-score绝对值大于3的值视为异常值，并进行处理（如填充为均值或中位数）。**

**特征选择：去除高度相关的特征，选择对目标变量有显著影响的特征。**

**StandardScaler：对特征进行标准化处理，使特征的均值为0，标准差为1。**



**图12.5 回归标签分布**

**Target\_Classification 分布情况：**

**0 4727**

**1 273**

1. ****回归****

*# -\*- coding: utf-8 -\*-*from sklearn.impute import KNNImputer  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.model\_selection import train\_test\_split, GridSearchCV  
from sklearn.preprocessing import StandardScaler, PolynomialFeatures  
from sklearn.linear\_model import LinearRegression, Ridge, Lasso, ElasticNet  
from sklearn.ensemble import RandomForestRegressor, AdaBoostRegressor, GradientBoostingRegressor, BaggingRegressor, \  
 VotingRegressor  
from sklearn.svm import SVR  
from sklearn.metrics import mean\_squared\_error, r2\_score  
from xgboost import XGBRegressor  
from sklearn.neighbors import KNeighborsRegressor  
from sklearn.tree import DecisionTreeRegressor  
import lightgbm as lgb  
import catboost as cb  
from scipy.stats import zscore  
*# 步骤 1: 读取数据集*data = pd.read\_csv("complex\_dataset.csv")  
*# 步骤 2: 数据探索*print("数据概况:")  
print(data.info()) *# 查看数据类型和缺失值*print("\n缺失值情况:")  
print(data.isnull().sum()) *# 查看每一列的缺失值数量*print("\n描述性统计:")  
print(data.describe()) *# 获取数值特征的摘要  
# 使用热图可视化缺失值*plt.figure(figsize=(10, 6))  
sns.heatmap(data.isnull(), cbar=False, cmap="viridis")  
plt.title("缺失值热图")  
plt.show()  
*# 步骤 3: 使用KNN插补法填充缺失值*def predict\_missing\_values\_knn(data):  
 *# 创建KNN插补器* imputer\_knn = KNNImputer(n\_neighbors=5) *# 设置邻居数为5* data\_imputed = pd.DataFrame(imputer\_knn.fit\_transform(data), columns=data.columns)  
 print("\n使用KNN插补法填充缺失值。")  
 return data\_imputed  
*# 使用KNN填充缺失值*data\_filled = predict\_missing\_values\_knn(data)  
*# 检查是否所有缺失值已处理*print("\nKNN插补后缺失值情况:")  
print(data\_filled.isnull().sum())  
*# 步骤 4: 特征工程*data\_encoded = pd.get\_dummies(data\_filled, columns=['Category\_A', 'Category\_B', 'Category\_C'], drop\_first=True)  
*# 步骤 5: 处理异常值 - 使用Z-score方法  
# 计算每个数值型特征的Z-score*z\_scores = np.abs(zscore(data\_encoded.select\_dtypes(include=[np.number])))  
outliers = (z\_scores > 3).all(axis=1)  
*# 移除异常值*data\_no\_outliers = data\_encoded[~outliers]  
*# 步骤 6: 对目标变量（Target\_Regression）进行对数变换*data\_no\_outliers['Target\_Regression'] = np.log1p(data\_no\_outliers['Target\_Regression'])  
*# 步骤 7: 特征标准化 - 对数值特征进行标准化*scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(data\_no\_outliers.drop(columns=['Target\_Regression', 'Target\_Classification']))  
*# 目标变量*y = data\_no\_outliers['Target\_Regression']  
*# 步骤 8: 多项式特征*poly = PolynomialFeatures(degree=2, interaction\_only=False, include\_bias=False)  
X\_poly = poly.fit\_transform(X\_scaled)  
*# 步骤 9: 数据集划分*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_poly, y, test\_size=0.2, random\_state=42)  
*# 步骤 10: 初始化并训练模型*models = {  
 "线性回归": LinearRegression(),  
 "岭回归": Ridge(),  
 "套索回归": Lasso(),  
 "弹性网": ElasticNet(),  
 "随机森林": RandomForestRegressor(),  
 "支持向量机": SVR(),  
 "XGBoost": XGBRegressor(),  
 "KNN": KNeighborsRegressor(),  
 "决策树": DecisionTreeRegressor(),  
 "LightGBM": lgb.LGBMRegressor(),  
 "CatBoost": cb.CatBoostRegressor(learning\_rate=0.1, iterations=500, depth=10, verbose=0),  
 "AdaBoost": AdaBoostRegressor(),  
 "梯度提升": GradientBoostingRegressor(),  
 "集成法": BaggingRegressor(),  
}  
*# 步骤 11: 训练模型并评估*results = {}  
for name, model in models.items():  
 *# 训练模型* model.fit(X\_train, y\_train)  
  
 *# 在测试集上预测* y\_pred = model.predict(X\_test)  
  
 *# 评估模型* mse = mean\_squared\_error(y\_test, y\_pred)  
 r2 = r2\_score(y\_test, y\_pred)  
  
 *# 保存结果* results[name] = {'MSE': mse, 'R^2': r2}  
 print(f"{name} - MSE: {mse:.4f}, R^2: {r2:.4f}")  
  
*# 步骤 12: 模型比较*results\_df = pd.DataFrame(results).T  
print("\n模型比较 (MSE & R^2):")  
print(results\_df.sort\_values(by='MSE'))  
  
*# 步骤 13: 集成方法 - 投票回归器*voting\_regressor = VotingRegressor(estimators=[(name, model) for name, model in models.items()])  
voting\_regressor.fit(X\_train, y\_train)  
*# 评估集成模型*y\_pred\_voting = voting\_regressor.predict(X\_test)  
mse\_voting = mean\_squared\_error(y\_test, y\_pred\_voting)  
r2\_voting = r2\_score(y\_test, y\_pred\_voting)  
print(f"\n投票回归器 - MSE: {mse\_voting:.4f}, R^2: {r2\_voting:.4f}")

****结果输出（部分）：****

****最终选择的最优模型为梯度提升回归GradientBoostingRegressor。****

**模型比较 (MSE & R^2):**

**MSE R^2**

****梯度提升 0.044113 0.479147****

**随机森林 0.044859 0.470347**

**岭回归 0.046103 0.455661**

**线性回归 0.046204 0.454464**

**LightGBM 0.047041 0.444576**

**CatBoost 0.048395 0.428596**

**集成法 0.050446 0.404381**

****反思与感悟：****

**在处理复杂数据集时，本代码采用了一系列数据预处理和特征工程技术，以提升回归模型的性能。首先，通过 KNN 插补法填充缺失值，这种方法利用数据的内在结构，通过查找最相似的 K 个样本，使用这些样本的均值或中位数来推测缺失值，从而保留数据的完整性。接着，对分类变量进行独热编码，将分类变量转换为数值型特征，以便模型能够有效处理。此外，使用 Z-score 方法检测和处理异常值，将 Z-score 绝对值大于 3 的值视为异常值，并进行处理（如填充为均值或中位数），以减少数据噪声对模型的影响。对目标变量进行对数变换，可以减少数据的偏态和异常值的影响，提高模型的拟合效果。对数值特征进行标准化处理，使特征的均值为 0，标准差为 1，这可以提高模型的收敛速度和预测性能。最后，生成多项式特征，增加特征的非线性关系，提高模型的表达能力。这些步骤共同作用，为后续的模型训练和评估提供了高质量的数据基础，显著提升了模型的性能和泛化能力。**

**在模型训练与评估阶段，本代码选择了多种回归模型，包括线性回归、岭回归、LASSO 回归、弹性网、随机森林、支持向量机、XGBoost、KNN、决策树、LightGBM、CatBoost、AdaBoost、梯度提升和集成法。对每个模型进行训练，并在测试集上进行预测，使用均方误差（MSE）和 R 方（R^2）评估每个模型的性能。MSE 衡量预测值与真实值之间的误差，R^2 衡量模型对数据的解释能力。将所有模型的性能结果汇总到一个 DataFrame 中，按 MSE 排序，选择性能最好的模型。使用投票回归器（VotingRegressor）将多个模型的预测结果进行加权平均，以提高模型的鲁棒性和预测性能。通过这些方法，本代码不仅提高了模型的拟合效果，还增强了模型的鲁棒性和泛化能力，为处理复杂数据和提升模型性能提供了有效的解决方案。**

**然而，从模型比较结果来看，梯度提升（Gradient Boosting）和随机森林（Random Forest）表现较好，但 R^2 仍然较低，表明模型对数据的解释能力较弱。这可能是因为数据中存在复杂的非线性关系，或者特征选择不够充分。**

**为此，尝试了许多方法，包括MICE插值，自学习（一边学习一边填补空缺值），调参（调了一天，代码在压缩包中，最终R方不到45%），stacking（最后R方都不到40%），都没有梯度提升回归的MSE ：0.044113 ，可决系数R方： 0.479147高，于是就只能暂时这样了。**

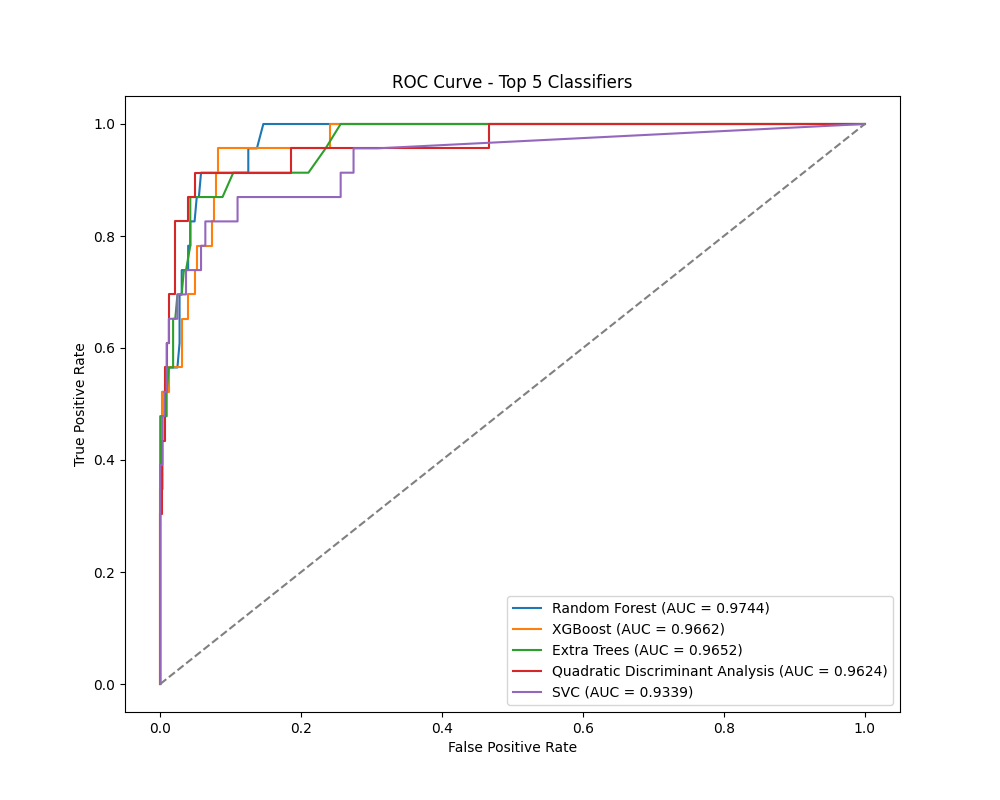
1. ****分类****

**根据（1）中提到的问题，Python解决和实现的核心代码如下：**

*# 使用MICE插值法填补缺失值*imputer = IterativeImputer(max\_iter=10, random\_state=0)  
data\_imputed = pd.DataFrame(imputer.fit\_transform(data), columns=data.columns)  
  
*# 特征和目标分离*X = data\_imputed.drop(columns=['Target\_Classification', 'Target\_Regression'])  
y = data\_imputed['Target\_Classification']  
  
*# 数据标准化*scaler = StandardScaler()  
X\_scaled = scaler.fit\_transform(X)  
  
*# 划分比例*X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.05, random\_state=42)  
  
*# 使用 SMOTE 仅对训练集进行类别不平衡处理*smote = SMOTE(sampling\_strategy='auto', random\_state=42)  
X\_train\_resampled, y\_train\_resampled = smote.fit\_resample(X\_train, y\_train)

*# 使用精度-召回曲线来优化阈值*for model\_name, model in classifiers:  
 print(f"调整 {model\_name} 阈值...")  
  
 *# 预测概率* if hasattr(model, 'predict\_proba'):  
 y\_prob = model.predict\_proba(X\_test)[:, 1]  
 else:  
 *# 对于 Perceptron 使用 decision\_function* y\_prob = model.decision\_function(X\_test)  
 y\_prob = (y\_prob - y\_prob.min()) / (y\_prob.max() - y\_prob.min()) *# 将决策函数值归一化到0-1范围  
  
 # 计算 Precision-Recall 曲线* precision, recall, thresholds = precision\_recall\_curve(y\_test, y\_prob)  
  
 *# 选择最优阈值，平衡 Precision 和 Recall* f1\_scores = 2 \* (precision \* recall) / (precision + recall)  
 best\_threshold = thresholds[np.argmax(f1\_scores)] *# 使用最大 F1-score对应的阈值* print(f"{model\_name} 最优阈值: {best\_threshold}")  
  
 *# 使用最佳阈值进行预测* y\_pred\_optimal = (y\_prob >= best\_threshold).astype(int)  
  
 *# 打印优化后的评估指标* print(f"{model\_name} 性能评估（优化阈值）:")  
 print(classification\_report(y\_test, y\_pred\_optimal))

****结果输出（部分）：****



**图12.5分类结果**

****最终选择的最优模型为阈值改变后的Quadratic Discriminant Analysis，其F1值为0.78，对患病类（“1”）的召回率有83%，精确率有73%。****

****Best Parameters for QDA: {'reg\_param': 0.1}****

**Top 5 Classifiers by ROC-AUC:**

**1. Random Forest: 0.9744**

**2. XGBoost: 0.9662**

**3. Extra Trees: 0.9652**

****4. Quadratic Discriminant Analysis: 0.9624****

**5. SVC: 0.9339**

**Quadratic Discriminant Analysis 最优阈值: 0.8546280811248002**

**Quadratic Discriminant Analysis 性能评估（优化阈值）:**

**precision recall f1-score support**

**0.0 0.99 0.98 0.98 328**

****1.0 0.73 0.83 0.78 23****

**accuracy 0.97 351**

**macro avg 0.86 0.90 0.88 351**

**weighted avg 0.97 0.97 0.97 351**

****反思与感悟：****

**实验结果显示，经过阈值优化后的 QDA 模型在精确率和召回率上达到了较好的平衡，F1-score 为 0.78，对患病类（“1”）的召回率达到 83%，精确率达到 73%。这表明该模型在处理不平衡数据集时具有较好的性能。同时，通过比较不同模型的 ROC-AUC 指标，我们可以看到随机森林（Random Forest）和 XGBoost 在整体性能上表现较好，但 QDA 在经过阈值优化后，对少数类的预测性能显著提升。这说明在处理分类问题时，需要综合考虑模型的选择和参数调优，以达到最佳的性能。**

**12.3 使用GPU训练的神经网络分类器\***

****数据集：complex\_dataset.csv****

****目标：在用一些经典的分类算法得出最优结果之后，我想试试用大批量训练（batch）来训练神经网络看能不能通过增加深度来提高Recall，同时通过调参再次平衡“1”类的Precision和Recall。****

****一开始使用CPU速度极慢，后面自主学习使用了pytorch的深度学习框架，效果很好。****

****Python实现核心代码：****

*# 参数设置*input\_dim = X\_train\_resampled.shape[1] *# 输入特征的维度*hidden\_dim = 64 *# 隐藏层维度*output\_dim = 2 *# 输出分类的个数（二分类）*batch\_size = 64  
epochs = 68  
learning\_rate = 0.001  
  
*# 将数据转换为PyTorch的Tensor格式*X\_train\_tensor = torch.tensor(X\_train\_resampled, dtype=torch.float32)  
y\_train\_tensor = torch.tensor(y\_train\_resampled.values, dtype=torch.long)  
  
X\_test\_tensor = torch.tensor(X\_test, dtype=torch.float32)  
y\_test\_tensor = torch.tensor(y\_test.values, dtype=torch.long)  
  
*# 数据加载器*train\_dataset = TensorDataset(X\_train\_tensor, y\_train\_tensor)  
train\_loader = DataLoader(dataset=train\_dataset, batch\_size=batch\_size, shuffle=True)  
  
*# 初始化模型，优化器和损失函数*device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")  
model = DeepNN(input\_dim, hidden\_dim, output\_dim).to(device)  
optimizer = optim.Adam(model.parameters(), lr=learning\_rate)  
  
*# 加权交叉熵损失*weights = torch.tensor([1.0, 3.0]).to(device) *# 给少数类（患者）更高的权重*criterion = nn.CrossEntropyLoss(weight=weights)  
  
*# 训练过程*for epoch in range(epochs):  
 model.train()  
 running\_loss = 0.0  
 correct = 0  
 total = 0  
  
 for inputs, labels in train\_loader:  
 inputs, labels = inputs.to(device), labels.to(device)  
  
 optimizer.zero\_grad()  
 outputs = model(inputs)  
 loss = criterion(outputs, labels)  
 loss.backward()  
 optimizer.step()  
  
 running\_loss += loss.item()  
  
 \_, predicted = torch.max(outputs.data, 1)  
 total += labels.size(0)  
 correct += (predicted == labels).sum().item()  
  
 print(f'Epoch {epoch + 1}/{epochs}, Loss: {running\_loss / len(train\_loader):.4f}, Accuracy: {correct / total:.4f}')

****输出与结果展示（部分）：****

**CUDA is available. Using GPU.**

**Epoch 60/60, Loss: 0.3198, Accuracy: 0.9899**

**Neural Network:**

**Precision (class 0 - Healthy): 0.9933**

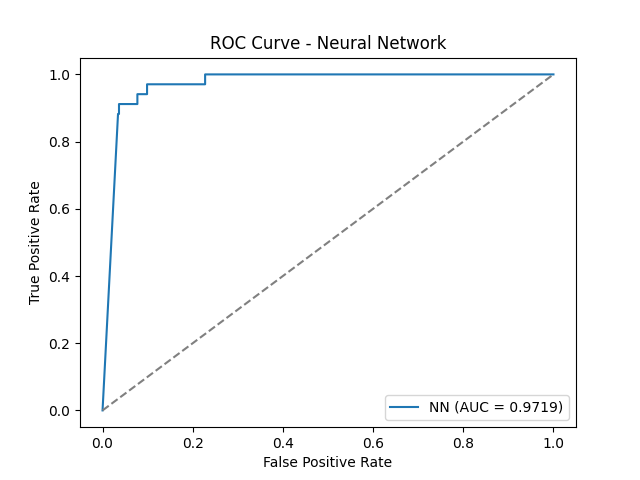
**Recall (class 0 - Healthy): 0.9485**

**Precision (class 1 - Patient): 0.5636**

****Recall (class 1 - Patient): 0.9118****

****F1 Score: 0.6966****

****ROC-AUC: 0.9719****



**图12.6神经网络ROC曲线**

****反思与感悟：****

****可以看到虽然F1值不尽如人意，不如上面的二次判别分析分类（Quadratic Discriminant Analysis）的，但神经网络实现了很高的AUC和 Recall (class 1 - Patient):，达到91.18%!****

****那么下面手动调节阈值，重点探索如何平衡 Precision 和 Recall 的冲突。****

****隐藏层设为128层，threshold = 0.95，epo设为58批次时：****

**CUDA is available. Using GPU.**

**Epoch 58/58, Loss: 0.3187, Accuracy: 0.9911**

**Neural Network:**

**Precision (class 0 - Healthy): 0.9785**

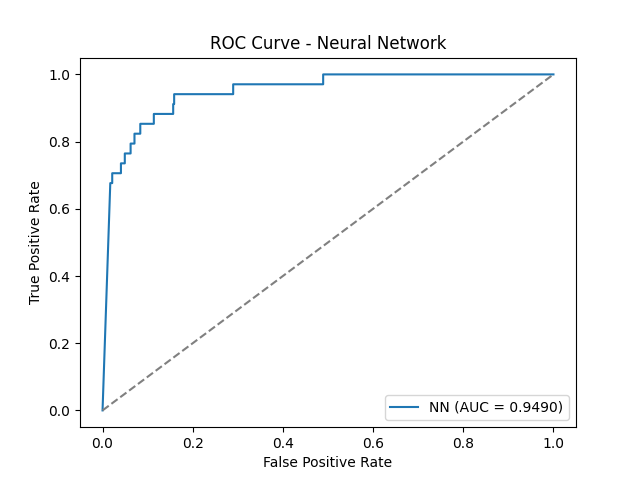
**Recall (class 0 - Healthy): 0.9785**

****Precision (class 1 - Patient): 0.7059****

****Recall (class 1 - Patient): 0.7059****

**F1 Score: 0.7059**

****ROC-AUC: 0.9490****



**图12.7平衡后的神经网络ROC曲线**

****通过调节二分类阈值，虽然“1”类的指标最终效果不如12.2.3的二次判别分析分类（Quadratic Discriminant Analysis），但也实现了平衡。****

**总 结 与 回 顾**

**在本次Python机器学习上机实验中，我遇到了许多挑战，但通过查阅CSDN上的资料、请教同学以及反复的尝试，我逐步克服了这些困难，并找到了有效的解决方法。**

**在实验初期，我主要聚焦于线性回归、逻辑回归、K近邻等经典算法的实现与优化。通过自定义优化器（如Momentum和RMSprop）进行训练。**

**随着实验的深入，我开始尝试使用sklearn库中的线性回归模型，并与自定义优化器进行对比。通过计算均方误差（MSE）评估模型的拟合效果，我深刻体会到了sklearn库在机器学习中的便捷性和高效性。**

**此外，我还手搓了KNN算法，并通过可视化展示了KNN在二维数据上的分类效果和最近邻的选取过程。并在Overleaf上面利用tex记录了一些算法的伪代码。**

**在神经网络方面，为了更直观地理解神经网络的工作原理及梯度下降算法，我还利用manim库可视化了神经网络的前向传播和反向传播过程。这一过程虽然复杂，但对我来说却是一次宝贵的学习经历。**

**对于半监督学习和集成学习等主题，通过LabelPropagation和LabelSpreading算法对部分标记的数据集进行学习和预测，我评估了不同参数设置下的模型性能，并比较了两种方法的优劣。**

**在实验的最后阶段，我使用了一个复杂的数据集，并尝试用多种机器学习算法解决分类和回归任务。为了平衡Precision和Recall的冲突，我采用了SMOTE技术进行过采样，使得少数类别样本数量得到平衡。同时，我也通过调整阈值等方法，进一步优化了模型的性能。**

**在自主探索方面，我了解了Pytorch框架与张量（tensor）的概念，以及深度学习的基本原理。并且学会了如何用GPU加速学习。**

**感谢储老师本学期的教学，老师讲的很好，您辛苦了！**

1. 标\*号为自主学习的方法或内容，不在老师发的上机手册中 [↑](#footnote-ref-0)
2. 博主*3Blue1Brown*制作动画所使用的库，https://zhuanlan.zhihu.com/p/670195488 [↑](#footnote-ref-1)
3. [Optuna](https://optuna.org/)是一个基于贝叶斯优化的超参数优化框架，https://blog.csdn.net/WHYbeHERE/article/details/135483560 [↑](#footnote-ref-2)