**机器学习应用实践（实验一） ——线性回归（一）**

**工业智能2201班 刘天行 20225354**

1. **实验目的**

1、掌握机器学习算法的开发流程。

2、掌握 Scikit-Learn 官方网站的查看、 学习和使用方法。

3、掌握 Scikit-Learn 中线性回归算法的使用， 解决波士顿房价预测问题。

1. **开发环境：****Visual Studio Code + Jupyter notebook拓展 + python 3.12.2**

文本

描述已自动生成

1. **实验内容**

**题目一：采用 scikit-learn 中的 LinearRegression(最小二乘)线性回归模型对波士顿房价数据集进行预测，分别使用正则方程和随机梯度下降方法建模。**

**具体内容：**

1. 导入数据
2. 查看数据集的描述、特征名、标签名、数据样本量等信息。
3. 获取样本的特征数据和标签数据。
4. 划分数据（分成训练集和测试集）
5. 数据归一化
6. 训练模型
7. 使用 sklearn 中线性回归的正规方程（LinearRegression）优化方法建模。
8. 使用 sklearn 中线性回归的随机梯度下降（SGDRegressor）优化方法建模。
9. 模型评估（2 个模型）

评价指标： MSE 和 R2 值。

**根据这样的实验步骤，进行一系列实验，回答以下四个讨论题：**

**【讨论一】 梯度下降和正规方程两种算法有何不同？**

分析梯度下降和正规方程两种算法的差异（计算时间、 评价指标对比）与优劣点。提示： python 中计时器-timeit.default\_timer()方法。

**【讨论二】数据归一化对算法有什么影响？**

对比使用数据归一化和不使用数据归一化，正规方程和梯度下降算法性能是否有差异？分析原因。

**【讨论三】 梯度下降算法中的学习率如何影响其工作？**

尝试修改随机梯度下降算法(SGDRegressor)的学习率， 观察参数对模型性能的影响（通过数据分析即可）， 试分析学习率与模型性能之间的关系。

**【讨论四】（选做）模型的泛化能力如何？**

（1）分别计算模型在训练样本上性能和在测试样本上的性能，判断模型是过拟合还是欠拟合？

（2）数据集划分不同对模型性能是否有影响？可尝试修改方法(train\_test\_split)中的 test\_size 参数，观察数据集划分对模型性能的影响。

（3）尝试使用其他线性回归模型。线性回归模型中除了 LinearRegression，还有 Ridge(岭回归)、Lasso、 Polynomial regression（多项式回归）等模型，自学官网资料，使用不同模型进行建模，观察不同模型训练后的模型权重差异，试分析模型的使用场合。

**题目二：采用梯度下降法（BGD）优化线性回归模型，对波士顿房价进行预测。**

（ 1）导入数据（从.csv 文件中导入数据代码如下）

（ 2）划分数据（分成训练集和数据集）

（ 3）数据归一化

（ 4）训练模型 model(train\_x,train\_y)

1. 初始化参数 w。可使用 np.concatenate 数组拼接函数，将截距与权重参数合并在一起（也可以不拼接合并）。
2. 求 f(x)。
3. 求 J(w)。
4. 求梯度。
5. 更新参数 w。
6. 的过程经过 epochs 次迭代。

（5）画出损失函数虽迭代次数的变化曲线。（ 通过损失函数变化曲线来观察梯度下降执行情况）

（6）测试集数据进行预测，模型评估。

（7）可视化： 展示数据拟合的效果。

**题目三（选做） ：小批量梯度下降算法（ MBGD）的编程实现。**

1. **实验情况**

**题目一：**

获取数据集：

import pandas as pd

import numpy as np

# 设置数据文件的路径

data\_path = "boston\_house\_prices.csv"

# 使用逗号作为分隔符加载数据

raw\_df = pd.read\_csv(data\_path, sep=",")

# 将第一行设置为列头

raw\_df.columns = raw\_df.iloc[0]

raw\_df = raw\_df[1:]  # 删除重复的列名行

# 转换数据类型为数值型，方便处理

raw\_df = raw\_df.apply(pd.to\_numeric, errors='coerce')

# 提取特征值 X 和目标值 y

X = raw\_df.drop(columns=["MEDV"])  # 移除目标列以分离特征

y = raw\_df["MEDV"].values  # 将目标值列提取为数组

# 打印数据结构和一些样本以确认

print(raw\_df.head())

print(X.head())

print(y[:5])

# 波士顿房价数据集是一个用于回归分析的数据集，其中包含506个观测值，每个观测值都有13个特征和1个目标值。

# 这13个特征值包括：

# CRIM：城镇人均犯罪率。

# ZN：大于25,000平方英尺的住宅土地的比例。

# INDUS：城镇非零售业务地区的比例。

# CHAS：查尔斯河虚拟变量（如果土地在河边，则为1；否则为0）。

# NOX：一氧化氮浓度。

# RM：住宅平均房间数。

# AGE：1940年以前建造的自住房屋的比例。

# DIS：到波士顿五个就业中心的加权距离。

# RAD：径向公路可达性指数。

# TAX：每10,000美元的全额财产税率。

# PTRATIO：城镇学生与教师比例。

# B：1000(Bk - 0.63)^2，其中Bk是城镇黑人的比例。

# LSTAT：人口中低收入者的比例。

# 目标值/标签值是：

# MEDV：自有住房的中位数价值（以1000美元计）。

# 波士顿房价数据集的特征值是这13个与房价相关的特征，而标签值是房价的中位数价值

X.shape, y.shape

划分数据：

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# train\_test\_split 方法,设定数据划分数据集比例，进行数据集划分

# test\_size=0.2表示测试集占总数据集的20%,也就是说，80%的数据用于训练，20%的数据用于测试(比例默认为0.25)

# random\_state=42是随机数种子。设置随机数种子可以保证每次运行代码时，数据的划分结果相同,保证可重复性

数据归一化(z-score标准化)：

from sklearn import preprocessing

import numpy as py

# 将列特征转化为标准正态分布，和整体分布相关，每个样本点都能对标准化产生影响

# 对于每个数值特征，算出它的均值（mean）和标准差（std）

# Z = (X - mean) / std

scaler = preprocessing.StandardScaler().fit(X\_train) # 统计训练集的均值、方差

X\_train\_std = scaler.transform(X\_train)

X\_test\_std  = scaler.transform(X\_test)

# 将每一个特征都转化为标准正态分布

# 标准化的主要目的是消除不同特征之间的尺度差异，让每个特征都处在同样的尺度上

# 用归一化的数据代替原始数据

X\_train = X\_train\_std

X\_test  = X\_test\_std

至此，完成全部数据预处理的部分

1. **梯度下降和正规方程两种算法有何不同？**

此处我们在上面对数据集进行观察的基础上进行如下的操作：

**训练模型：**

正规方程求解线性回归：

from sklearn.linear\_model import LinearRegression

import time

from sklearn.metrics import mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

reg = LinearRegression()# 初始化线性回归模型

start\_time = time.time()# 记录起始时间

reg.fit(X\_train, y\_train)# 拟合训练数据

print("Linear Regression fitting time:", time.time() - start\_time, "seconds")# 打印拟合过程所耗时间

y\_pred = reg.predict(X\_test)# 使用测试集进行预测

# 计算训练集和测试集的均方误差（MSE）

train\_mse = mean\_squared\_error(y\_train, reg.predict(X\_train))

test\_mse = mean\_squared\_error(y\_test, y\_pred)

# 打印训练集和测试集的MSE

print("Training MSE: ", train\_mse)

print("Testing MSE:  ", test\_mse)

# 计算训练集和测试集的R²（决定系数）

train\_r2 = r2\_score(y\_train, reg.predict(X\_train))

test\_r2 = r2\_score(y\_test, y\_pred)

# 打印训练集和测试集的R²

print("Training R^2: ", train\_r2)

print("Testing R^2: ", test\_r2)

# 绘制真实值和预测值的折线图

plt.figure(figsize=(12, 6))

plt.plot(y\_test, label='Actual Values', color='blue')

plt.plot(y\_pred, label='Predicted Values', color='orange')

plt.legend()

plt.title("Actual vs Predicted Values")

plt.xlabel("Sample Index")

plt.ylabel("Value")

plt.show()

# 获取模型系数和截距

coefficients = reg.coef\_

intercept = reg.intercept\_

# 打印模型系数和截距

print("Coefficients:\n", coefficients)

print("Intercept:", intercept)

# 绘制线性回归模型的图像

plt.figure(figsize=(10, 6))

plt.scatter(X\_test[:, 0], y\_test, color='blue', label='Test Data')

plt.plot(X\_test[:, 0], coefficients[0] \* X\_test[:, 0] + intercept, color='red', label='Regression Line')

plt.xlabel("Feature")

plt.ylabel("Target")

plt.title("Linear Regression Model")

plt.legend()

plt.show()

运行结果：

Linear Regression fitting time: 0.0019974708557128906 seconds Training MSE: 21.641412753226312 Testing MSE: 24.291119474973517 Training R^2: 0.7508856358979673 Testing R^2: 0.668759493535632

Coefficients: [-1.00213533 0.69626862 0.27806485 0.7187384 -2.0223194 3.14523956 -0.17604788 -3.0819076 2.25140666 -1.76701378 -2.03775151 1.12956831 -3.61165842] Intercept: 22.796534653465343

图形用户界面, 图表, 折线图

描述已自动生成

图表, 散点图

描述已自动生成

梯度下降法求解线性回归问题：

from sklearn.linear\_model import SGDRegressor

import time

from sklearn.metrics import mean\_squared\_error

import matplotlib.pyplot as plt

# 随机梯度下降算法在每次更新中根据一个样本计算损失函数的梯度

# 其优点是运行速度快，适用于大批量数据训练。

# SGDRegressor 可以支持不同的 loss函数和正则化惩罚项来拟合线性回归模型。

# 设置字体

plt.rcParams['font.family'] = 'SimHei'

plt.rcParams['axes.unicode\_minus'] = False

# 设置多次循环次数和初始学习率

iterations = 10

initial\_eta = 0.0005

# 存储预测结果

y\_predict\_list = []

# 对不同学习率进行模型训练和评估

for i in range(iterations):

    eta = initial\_eta \* (i + 4) # 设置当前学习率

    reg = SGDRegressor(eta0=eta, random\_state=42)# 创建SGDRegressor模型，设置学习率和随机种子

    start\_time = time.time()# 记录起始时间

    reg.fit(X\_train, y\_train)# 拟合模型

    y\_predict = reg.predict(X\_test)# 预测测试集

    y\_predict\_list.append(y\_predict)# 记录预测结果

    elapsed\_time = time.time() - start\_time# 计算训练时间

    # 计算MSE和R²

    test\_mse = mean\_squared\_error(y\_test, y\_predict)

    test\_r2 = r2\_score(y\_test, y\_predict)

    # 绘制真实值和预测值的折线图

    plt.figure()

    plt.plot(y\_test, label='真实值', color='purple')

    plt.plot(y\_predict, label='预测值', color='orange')

    plt.legend()

    plt.title(f"学习率={eta:.6f}, MSE={test\_mse:.6f}, R^2={test\_r2:.6f}, 训练耗时={elapsed\_time:.6f}秒")

# 绘制最后一次测试集和预测值的折线图

plt.figure()

plt.plot(y\_test, label='真实值', color='purple')

plt.plot(y\_predict, label='预测值', color='orange')

plt.legend()

plt.title(f"学习率={eta:.6f}, MSE={test\_mse:.6f}, R^2={test\_r2:.6f}, 训练耗时={elapsed\_time:.6f}秒")

plt.show()

# 查看训练完成后的SGDRegressor模型的系数和截距

print("R2:",reg.score(X\_train, y\_train))# R^2

print("coef:\n",reg.coef\_)# 系数

print("intercept:",reg.intercept\_)# 偏置

print("MSE:",mean\_squared\_error(y\_test, y\_predict))# 均方误差

print("params:",reg.get\_params())# 参数

# 可视化

plt.figure(figsize=(10,5))

plt.plot(y\_test)

plt.plot(y\_predict)

plt.xticks(())

plt.show()

运行结果：

图表, 折线图

描述已自动生成

图表, 折线图

描述已自动生成

图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成图表, 折线图

描述已自动生成

R2: 0.749384748659065 coef: [-0.90158104 0.49285972 -0.00631569 0.76490925 -1.75383491 3.26184982 -0.19531696 -2.82857441 1.37910373 -0.87348503 -1.96813398 1.13438027 -3.5867964 ] intercept: [22.81140031] MSE: 25.054530831361923 params: {'alpha': 0.0001, 'average': False, 'early\_stopping': False, 'epsilon': 0.1, 'eta0': 0.006500000000000001, 'fit\_intercept': True, 'l1\_ratio': 0.15, 'learning\_rate': 'invscaling', 'loss': 'squared\_error', 'max\_iter': 1000, 'n\_iter\_no\_change': 5, 'penalty': 'l2', 'power\_t': 0.25, 'random\_state': 42, 'shuffle': True, 'tol': 0.001, 'validation\_fraction': 0.1, 'verbose': 0, 'warm\_start': False}

正规化方程优化是通过最小二乘法直接计算出的结果，训练集不变，结果、计算量也不会改变。随机梯度下降是对每个批次的数据中随机抽取一个计算损失、修正系数，结果具有随机性，计算量也不确定。

重复拟合并通过metric. mean\_squared\_error计算每一次预测结果与标签的误差平方和的平均值，得到MSE，再计算1000次的平均值mean\_MSE。可以发现最小二乘法的MSE是固定的；随机梯度下降法的结果不稳定，但整体的值小于最小二乘法，说明随机梯度下降法的拟合效果较好。

1. **数据归一化对算法有什么影响？**

将这两行代码注释：

文本

描述已自动生成

直接利用split之后的数据集进行训练，不进行归一化，最终预测结果相近。系数的分布有所不同。

分别运行修改后和修改前的代码，得到下面的结果

正规方程，未归一化: MSE = 24.2911, R2 = 0.6687, 时间 = 0.0019 秒

正规方程，归一化: MSE = 24.2911, R2 = 0.6687, 时间 = 0.0004 秒

梯度下降，未归一化: MSE = 196.8981, R2 = -1.2862, 时间 = 0.001999 秒

梯度下降，归一化: MSE = 25.0545, R2 = 0.6583, 时间 = 0.00205 秒

学习率设置不合理会导致无法收敛；

从拟合方法上看，最小二乘法几乎不受影响，随机梯度下降收到了较大的影响。

将所有特征的数据归一化，缩放至同一尺度内，可以防止不同特征的属性值差异过大，导致系数更新速率不同，影响最终训练结果。

1. **梯度下降算法中的学习率如何影响其工作？**

为了探究学习率对梯度下降算法的影响，我们设定了一组从0.002到0.0065不同的学习率，分别用随机梯度下降法训练回归模型，依然通过上述三个指标来对判断模型的好坏并判断学习率对随机梯度下降法算法性能的影响。

运行代码得到的结果在上面可见，其中

学习率 = 0.002, MSE = 25.224, R2 = 0.656, 运行时间 = 0.00555 秒

学习率 = 0.0025, MSE =25.00405, R2 = 0.659, 运行时间 = 0.0045 秒

学习率 = 0.003, MSE =25.097, R2 = 0.657, 运行时间 = 0.0029 秒

学习率 = 0.0035, MSE =24.945, R2 = 0.659, 运行时间 = 0.0029 秒

学习率 = 0.004, MSE =24.828, R2 = 0.661, 运行时间 = 0.0030 秒

学习率 = 0.0045, MSE =24.846, R2 = 0.661, 运行时间 = 0.0029 秒

学习率 = 0.005, MSE =24.755, R2 = 0.662, 运行时间 = 0.0033 秒

学习率 = 0.0055, MSE =25.228, R2 = 0.655, 运行时间 = 0.0020 秒

学习率 = 0.006, MSE =25.135, R2 = 0.657, 运行时间 = 0.0020 秒

学习率 = 0.0065, MSE =25.054, R2 = 0.658, 运行时间 = 0.0026 秒

不同的学习率的模型MSE和并无太大差距别，运行时间变化趋势也并非单调的。学习率在梯度下降算法中控制着参数更新的步长。过高的学习率可能导致模型在最小值附近震荡甚至发散，而过低的学习率会导致收敛过慢。

1. **数据集的划分对模型性能的影响**

为了探究训练集和测试机比例对模型的影响，我使用np.linspace生成一个包含19个值的列表，表示测试集比例的变化范围（从0.05到0.95），分别训练模型并比较MSE的变化

1. import numpy as np
2. from sklearn.model\_selection import train\_test\_split
3. from sklearn import preprocessing
4. from sklearn.linear\_model import LinearRegression
5. from sklearn.metrics import mean\_squared\_error
6. import matplotlib.pyplot as plt
7. data\_file = r"boston\_house\_prices.csv"
8. with open(data\_file, encoding='utf-8') as f:
9. data = np.loadtxt(data\_file, delimiter=',', skiprows=2)
10. X = data[:, :-1]    # X：数据集的特征值
11. y = data[:, -1]     # y：数据集的标签值
12. test\_MSE\_list = []
13. rates = np.linspace(0.05,0.95,19).tolist()  # np.linspace生成一个包含19个值的列表，表示测试集比例的变化范围（从0.05到0.95）
14. for rate in rates:
15. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=rate, random\_state=54)
16. std\_scaler = preprocessing.StandardScaler().fit(X\_train) # 统计训练集的均值、方差
17. X\_train\_std = std\_scaler.transform(X\_train)
18. X\_test\_std  = std\_scaler.transform(X\_test)
20. # 用归一化的数据代替原始数据
21. X\_train = X\_train\_std
22. X\_test  = X\_test\_std
23. reg = LinearRegression()            # 创建对象,采用的还是正规方程法
24. reg.fit(X\_train, y\_train)           # 拟合
25. y\_predict = reg.predict(X\_test)     # 测试集预测值
26. test\_MSE\_list.append(mean\_squared\_error(y\_test, y\_predict))
27. plt.plot(rates,test\_MSE\_list)
28. plt.xlabel("test\_size")
29. plt.ylabel("MSE")
30. plt.title("MSE随测试集比例变化的曲线")
31. plt.show()
32. # 由结果图可以看出，当测试集比例在0.05到0.15之间时，MSE值较小，说明模型的预测能力较好
33. # 当测试集比例大于0.15时，MSE值逐渐增大，说明模型的预测能力变差
34. # 因为训练集的数据越多，模型的训练效果才越好，所以测试集比例越大，模型的预测能力越差

运行结果如图：

图表, 折线图, 直方图

描述已自动生成

由结果图可以看出，当测试集比例在0.05到0.15之间时，MSE值较小，说明模型的预测能力较好

当测试集比例大于0.15时，MSE值逐渐增大，说明模型的预测能力变差，因为训练集的数据越多，模型的训练效果才越好，所以测试集比例越大，模型的预测能力越差，足够的训练数据可以帮助线性回归模型更准确地估计模型参数，提升模型的预测性能。如果训练数据量不足，模型可能无法捕捉数据的真实模式，从而影响模型的预测精度。如果训练集太小，模型可能会过拟合训练数据，即在训练数据上表现良好但在测试数据上表现不佳。反之，如果训练集太大，而测试集太小，可能会低估模型在未见数据上的泛化能力。

5. **使用其他的线性回归模型**

我分别使用实验教学文档中提到的Lasso，Ridge和多项式回归来进行模型的训练

代码如下：

Lasso

from sklearn.linear\_model import Lasso

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import make\_pipeline

from sklearn.metrics import mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

# 创建一个Pipeline对象，包含两个步骤：首先，使用StandardScaler对特征进行标准化；

# 然后，使用Lasso进行回归。这个Pipeline对象就相当于一个带有Lasso正则化的线性回归模型

pipeline = make\_pipeline(StandardScaler(), Lasso(alpha=0.1))

# 用训练数据来训练这个Pipeline

pipeline.fit(X\_train, y\_train)

# 在测试集上进行预测

y\_predict = pipeline.predict(X\_test)

# 计算相关系数（R²得分）和均方误差（MSE）

train\_score = pipeline.score(X\_train, y\_train)

test\_mse = mean\_squared\_error(y\_test, y\_predict)

test\_r2 = r2\_score(y\_test, y\_predict)

# 打印相关系数和均方误差

print('训练集相关系数: ', train\_score)

print("测试集MSE: ", test\_mse)

print("测试集R²: ", test\_r2)

# 绘制真实值和预测值的折线图

plt.figure(figsize=(12, 6))

plt.plot(y\_test, label='真实值', color='purple')

plt.plot(y\_predict, label='预测值', color='orange')

plt.legend()

plt.title("Lasso回归预测结果")

plt.xlabel("样本索引")

plt.ylabel("房价")

plt.show()

# 通过调节Lasso模型中的alpha参数，可以控制正则化的强度，避免过拟合或欠拟合

# 较高的alpha值会增加正则化力度，减少过拟合的风险，但可能导致欠拟合

# 较低的alpha值会减少正则化力度，模型可能更好地拟合训练数据，但可能导致过拟合

运行结果：

训练集相关系数: 0.9288132515243274

测试集MSE: 69.7909090945629

测试集R²: 0.18962577021764915

图表

描述已自动生成

Ridge

from sklearn.linear\_model import Ridge

# 这段代码使用了Ridge回归模型，这是一种线性回归模型，

# 在多元线性回归基础上加入了L2正则项，针对模型中存在的共线性关系的为变量增加一个小的平方偏差因子，防止不同特征之间相互关联影响模型训练，来约束模型的权重

# 这里的alpha参数就是正则化项的强度，alpha的值越大，正则化项的影响就越大。在这个例子中，alpha的值被设置为0.5。

from sklearn.model\_selection import GridSearchCV

# 定义要搜索的alpha值

parameters = {'alpha': [1e-10, 1e-5, 1e-4, 1e-3,1e-2, 1, 5, 10,11, 20]}

ridge = Ridge()

# 创建GridSearchCV对象，使用折交叉验证（cv=5），训练数据被分成了5份，

# 每次都使用其中的4份来训练模型，然后在剩下的1份上进行验证，这样一共可以进行5次训练和验证，更准确地评估模型的性能

ridge\_regressor = GridSearchCV(ridge, parameters, scoring='neg\_mean\_squared\_error', cv=5)

# 使用GridSearchCV对象拟合数据

ridge\_regressor.fit(X\_train, y\_train)

# 打印最优的alpha值

print(ridge\_regressor.best\_params\_)

# 打印最优模型的MSE

print(ridge\_regressor.best\_score\_)

# 使用最优模型在测试集上进行预测

y\_predict = ridge\_regressor.predict(X\_test)

# 计算并打印测试集上的MSE

reg = Ridge(alpha=.5)            # 创建对象

reg.fit(X\_train, y\_train)           # 拟合

y\_predict = reg.predict(X\_test)     # 测试集预测值

print("test\_MSE: ",mean\_squared\_error(y\_test, y\_predict))

plt.title("Ridge回归预测结果")

plt.plot(y\_test)

plt.plot(y\_predict)

plt.plot(y\_test,label='Test')

plt.plot(y\_predict,label='Predict')

plt.legend() # 添加图例

plt.show()

# 最终在测试集上的均方误差比直接多元线性回归小一些，说明正则项损失函数是比较有效的

运行结果：

{'alpha': 5}

-30.986120973531854

test\_MSE: 126.55507028531561

图表, 折线图

描述已自动生成

多项式回归

from sklearn.preprocessing import PolynomialFeatures

from sklearn.pipeline import make\_pipeline

# Pipeline是一个包装器，用于封装多个处理步骤，以便它们可以被当作一个整体来使用

# 首先利用PolynomialFeatures构建数据集的各阶幂，再利用之前的多元线性回归LinearRegression()进行拟合

# 创建一个Pipeline对象，包含两个步骤：首先，使用PolynomialFeatures生成2阶的多项式特征；

# 然后，使用LinearRegression进行回归。这个Pipeline对象就相当于一个2阶的多项式回归模型

pipeline = make\_pipeline(PolynomialFeatures(5), LinearRegression())

# 当模型多项式阶数大于2阶，相关系数为 1.0，表示在训练集上模型的拟合效果非常好，然而测试集上的均方误差却相当高

# 这是过拟合（overfitting）的表现，这里选用1阶

# 用训练数据来训练这个Pipeline。

# 这个过程包括两个步骤：首先，使用PolynomialFeatures对训练数据进行转换，生成 2 阶的多项式特征；然后，使用LinearRegression拟合这些特征和目标值

pipeline.fit(X\_train,y\_train)                                      #训练模型

y\_predict = pipeline.predict(X\_test)#在测试集上预测结果并保存在pred变量中

score = pipeline.score(X\_train,y\_train)#计算相关系数

print('相关系数: ', score)

print("test\_MSE: ",mean\_squared\_error(y\_test, y\_predict))

plt.plot(y\_test,label='Test')

plt.plot(y\_predict,label='Predict')

plt.legend() # 添加图例

plt.title("多项式回归预测结果,采用训练阶数：{}".format(5))

plt.show()

# 经过测试发现，PolynomialFeatures(n)中阶数n=2时效果尚可，比直接线性回归稍差一些。当n>=3时会发散，即过拟合

运行结果：

相关系数: 1.0 test\_MSE: 121108342215.88416

图表, 条形图, 直方图

描述已自动生成

通过查阅更多资料以及对上面结果的分析，我们可以总结出：

1. 线性回归：适用于特征数量较少且多重共线性不严重的数据集。
2. 岭回归：适用于特征数量较少且多重共线性不严重的数据集。
3. Lasso 回归：适用于特征数量较少且多重共线性不严重的数据集。
4. 多项式回归：适用于特征数量较少且多重共线性不严重的数据集。

**题目二：**

本题目需要自己编写梯度下降法BGD函数来优化线性回归模型

import numpy as np

import matplotlib.pyplot as plt

import time

plt.rcParams['axes.unicode\_minus'] = False

def load\_data(test\_rate=0.2):

    # 1.从文件导入数据

    data\_file = r"boston\_house\_prices.csv"

    with open(data\_file, encoding='utf-8') as f:

        data = np.loadtxt(data\_file, delimiter=',', skiprows=2)

    X = data[:, :-1]

    y = data[:, -1]

    # 随机打乱数据

    np.random.seed(42)  # 使用相同的随机种子，保证每次运行得到相同的结果

    shuffle\_indices = np.random.permutation(len(X)) # 生成随机序列

    X, y = X[shuffle\_indices], y[shuffle\_indices]   # 用随机序列对数据进行重排

    # 2.将原数据集拆分成训练集和测试集，按照test\_rate划分

    num\_test = int(len(X) \* test\_rate)

    X\_test, y\_test = X[:num\_test], y[:num\_test]

    X\_train, y\_train = X[num\_test:], y[num\_test:]

    # 3、返回训练和测试数据集的特征和标签

    return X\_train, y\_train, X\_test, y\_test

def data\_processing(train\_data, test\_data, kind=1):

    # 1、传入train\_data,test\_data以及选择那一种归一化的kind 参数

    train\_std = np.zeros\_like(train\_data, dtype=float)  # 生成一个和train\_data维度相同的全0数组

    test\_std = np.zeros\_like(test\_data, dtype=float)

    # 当kind=1时，进行z-score标准化处理；当kind=2时，进行最大最小归一化处理

    if kind==1:

        avg = np.mean(train\_data, axis=0)   # 均值

        std = np.std(train\_data, axis=0)    # 标准差

        train\_std = (train\_data - avg) / std

        test\_std = (test\_data - avg) / std

    # 3.最大最小归一化处理

    else:

        max = np.max(train\_data, axis=0)

        min = np.min(train\_data, axis=0)

        train\_std = (train\_data - min)/(max - min)

        test\_std = (test\_data - min)/(max - min)

    # 返回处理后的数据集

    return train\_std, test\_std

def get\_batches(dataset, labels, batch\_size=1, drop\_last=False):    # 将数据集划分为多个批次，每个批次的样本数量为 batch\_size

    # drop\_last：是否丢弃最后一个不足batch\_size的批次

    batches = []

    index = 0

    data\_len = dataset.shape[0]

    if drop\_last:

        data\_len -= data\_len % batch\_size

    while index < data\_len:

        # 如果数据集的长度大于等于index+batch\_size，则取出index到index+batch\_size的数据

        if index + batch\_size < data\_len:

            x = dataset[index:index + batch\_size]

            y = labels[index:index + batch\_size]

        elif not drop\_last:

            x = dataset[index:]

            y = labels[index:]

        index += batch\_size

        batches.append([x, y])

    # 返回划分好的批次,其中包含了批次数据集的的特征和标签

    return batches

class LinearRegression\_numpy(object):

    def \_\_init\_\_(self, num\_of\_weights):

        # 初始化系数w的值，利用随机数生成函数生成服从标准正态分布的随机数

        # 这里是将截距 b并入了系数w中，所以系数w的长度比特征数多1

        self.w = np.random.randn(num\_of\_weights + 1)

    # 将预测输出的过程以“类和对象”描述

    def forward(self, X):

        # y\_pred = np.dot(X, self.w)  # 矩阵乘法

        # 这个函数的作用是将X和self.w进行矩阵乘法运算，得到y\_pred

        return np.dot(X, self.w[1:]) + self.w[0]

    def loss(self, X, y):

        # 这个函数的作用是计算模型在数据集X上的均方误差

        y\_pred = self.forward(X)            # 调用forward函数，得到预测值

        return np.mean((y - y\_pred)\*\*2)

    def gradient(self, X, y):

        # 这个函数的作用是计算模型在数据集X上的梯度

        y\_pred = self.forward(X)            # 调用forward函数，得到预测值

        grad\_w = -2/len(X) \* np.dot(X.T, (y - y\_pred))  # 根据公式计算梯度，这里是对w求导

        grad\_b = -2/len(X) \* np.sum(y - y\_pred)     # 根据公式计算梯度，这里是对b求导

        return grad\_w, grad\_b

    def update(self, grad\_w, grad\_b, eta):

        # 这个函数的作用是根据梯度和学习率更新系数w（包含截距）

        self.w[1:] -= eta \* grad\_w  # 根据梯度和学习率更新系数w

        self.w[0] -= eta \* grad\_b   # 根据梯度和学习率更新截距b

    def train\_BGD(self, X, y, num\_epoches, eta):

        # 这个函数的作用是使用梯度下降法训练模型

        loss\_history = [] # 用于保存每次迭代后的损失函数值

        for epoch\_id in range(num\_epoches): # 开始迭代，迭代周期是num\_epoches，每次迭代都要计算当前的梯度，并更新参数

            grad\_w, grad\_b = self.gradient(X, y)    # 计算梯度，计算当前参数下损失函数的梯度

            self.update(grad\_w, grad\_b, eta)        # 更新参数

            loss = self.loss(X, y)

            loss\_history.append(loss)               # 计算损失函数的值，保存到loss\_history中

        return loss\_history

    def train\_MBGD(self, X, y, num\_epoches, batch\_size, eta, drop\_last=False):

        loss\_history = []           # 用于保存每次迭代后的损失函数值

        np.random.seed(54)          # 设置随机种子，保证每次运行结果一致

        for epoch\_id in range(num\_epoches):

            # for X, y in get\_batches(X, y, batch\_size, drop\_last=drop\_last):

                # 经典的四步训练流程：前向计算->计算损失->计算梯度->更新参数（分别调用类的方法）

                indices = np.random.choice(len(X), batch\_size)  # 随机采样batch\_size个样本，返回采样样本的下标

                X\_batch, y\_batch = X[indices], y[indices]       # 根据下标取出对应的样本

                loss = self.loss(X\_batch, y\_batch)              # 计算损失函数值，保存到loss\_history中

                loss\_history.append(loss)

                grad\_w, grad\_b = self.gradient(X\_batch, y\_batch)# 计算当前参数下损失函数的梯度

                self.update(grad\_w, grad\_b, eta)                # 更新参数

        return loss\_history

plt.rcParams['axes.unicode\_minus'] = False

plt.rcParams['font.family'] = 'SimHei'

plt.rcParams['font.size'] = 11

num\_epoches = 2000 # 训练轮数

lr = 0.001234        # 学习率，BGD 和 MBGD 的学习率都采用这个值

# 1、获取数据并划分数据集

X\_train, y\_train, X\_test, y\_test = load\_data(test\_rate=0.2)

# 2、数据标准化/归一化处理

# data\_processing(X\_train, X\_test)

X\_train, X\_test = data\_processing(X\_train, X\_test, 1)

# 3、LinearRegression\_numpy 类实例化

reg\_bgd = LinearRegression\_numpy(X\_train.shape[1])# 调用类中自定义的 train\_BGD()，训练模型

start = time.time()

losses\_bgd = reg\_bgd.train\_BGD(X\_train, y\_train, num\_epoches=num\_epoches, eta=lr)

print("BGD 训练耗时",time.time()-start,"s")

# 4、使用测试集进行测试，模型评价

# 将模型训练得到的系数w与测试数据test\_data计算得到预测值。

# 计算测试集的MSE

MSE\_bgd = reg\_bgd.loss(X\_test, y\_test)

# 5、画出损失函数的变化趋势，画出预测值与真实值曲线

plt.figure()

plt.plot(np.arange(int(num\_epoches / 10),num\_epoches), losses\_bgd[int(num\_epoches / 10):])  # 只绘制后 90% 的训练轮数，都绘制的话趋势变化可视化效果不好

plt.xlabel("训练轮数")

plt.ylabel("损失函数值")

plt.title("BGD 损失函数变化趋势")

# plt.show()

# 测试集

plt.figure(figsize=(7,9))

plt.subplot(212)

plt.plot(y\_test,label='test\_true')

plt.plot(reg\_bgd.forward(X\_test),label='test\_predict')

plt.title("BGD Test Data MSE={:.2f}".format(MSE\_bgd))

# plt.show()

plt.legend() # 添加图例

# 训练集

MSE = reg\_bgd.loss(X\_train, y\_train)

plt.subplot(211)

plt.plot(y\_train,label='train\_true')

plt.plot(reg\_bgd.forward(X\_train),label='train\_predict')

plt.title("BGD Train Data MSE={:.2f}".format(MSE))

plt.legend() # 添加图例

plt.show()

运行代码得到结果如下图表

描述已自动生成 图表

描述已自动生成



**题目三：**

题目三中要求编程实现小批量梯度下降算法MBGD。

# 1、获取数据并划分数据集

X\_train, y\_train, X\_test, y\_test = load\_data(test\_rate=0.2)

# 2、数据标准化/归一化处理

# data\_processing(X\_train, X\_test)

X\_train, X\_test = data\_processing(X\_train, X\_test, 1)

reg\_mbgd = LinearRegression\_numpy(X\_train.shape[1])

MSE\_list = []

lr = 0.001        # 学习率，BGD 和 MBGD 的学习率都采用这个值

num\_epoches = 2000 # 训练轮数

fig, ax1 = plt.subplots()

# fig, ax2 = plt.subplots()

ax1.grid()

ax1.set\_xlabel('迭代次数')

ax1.set\_ylabel('损失函数值')

for batch\_size in [40,80,100,200]:

    # 当 batch\_size=len(X\_train)时，等价于BGD，当batch\_size=1时，等价于SGD

    num\_epoches\_mbgd = int(4\*batch\_size) # 确保对于不同的 batch\_size，训练的轮数相同，因为不同的 batch\_size,

                                            # 每轮的迭代次数等于样本总数/batch\_size，所以通过调整 num\_epoches 来保证每轮迭代次数相同

    MSE = 0

    start = time.time()

    losses\_mbgd = reg\_mbgd.train\_MBGD(X\_train, y\_train, num\_epoches=num\_epoches, batch\_size=batch\_size, eta=lr)

    y\_predict = reg\_mbgd.forward(X\_test)

    # plt.show()

    if batch\_size == 20:

        ax1.plot(losses\_mbgd,  linestyle='-', color='b', label='batch\_size=20 的损失曲线')

    elif batch\_size == 40:

        ax1.plot(losses\_mbgd,  linestyle='-', color='r', label='batch\_size={}的损失曲线'.format(40))

    elif batch\_size == 80:

        ax1.plot(losses\_mbgd,  linestyle='-', color='g', label='batch\_size={}的损失曲线'.format(80))

    elif batch\_size == 100:

        ax1.plot(losses\_mbgd,  linestyle='-', color='y', label='batch\_size={}的损失曲线'.format(100))

    elif batch\_size == 200:

        ax1.plot(losses\_mbgd,  linestyle='-', color='c', label='batch\_size={}的损失曲线'.format(200))

    ax1.legend(loc='upper left')

    losses\_mbgd=[]

    plt.figure()

    plt.figure(figsize=(7,9))

    plt.subplot(212)

    plt.plot(y\_test)

    plt.plot(reg\_bgd.forward(X\_test))

    MSE\_mbgd=reg\_mbgd.loss(X\_test, y\_test)

    plt.title("MBGD Test Data MSE={:.2f},when batch\_size is {:.4f}".format(MSE\_mbgd,batch\_size))

    # plt.show()

    # 训练集

    MSE = reg\_bgd.loss(X\_train, y\_train)

    plt.subplot(211)

    plt.plot(y\_train)

    plt.plot(reg\_bgd.forward(X\_train))

    plt.title("MBGD Train Data MSE={:.2f}".format(MSE))

    print("MBGD 训练耗时",(time.time()-start)/10.0,"s")

    print("num\_epoches={}, batch\_size={},lr={:.8f}, MSE={:.2f}".format(num\_epoches,batch\_size,lr,reg\_mbgd.loss(X\_test, y\_test)))

# 由图，较大的批量可以使模型对每一步的参数更新有更准确的估计，可能会使得模型在训练集上的表现更好

# 训练稳定性：较小的批量会导致训练过程中损失函数的震荡更加剧烈，而较大的批量可以使训练过程更加稳定

这样我们得到了如下的结果：文本

描述已自动生成

图表, 直方图

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成 图表

描述已自动生成

**机器学习应用实践（实验二） —逻辑回归**

**工业智能2201班 刘天行 20225354**

1. **实验目的**
2. 掌握逻辑回归算法的原理。
3. 掌握 Scikit-Learn 中逻辑回归算法的使用，以及特征可视化和绘制（线性和非线性）决策边界的方法。
4. 掌握 OvR、 OvO、 MvM 等多分类策略，及 softmax 回归的原理及编程实现。
5. 掌握 Scikit-Learn 中多种分类评估指标的函数使用方法。
6. 熟悉手动调整模型超参数提高泛化能力的方法。
7. 熟悉 Scikit-Learn 自带分类数据集（iris 数据集）的使用。
8. 具备使用 python 实现二分类和多分类逻辑回归算法的编程能力。
9. **开发环境：Visual Studio Code + Jupyter notebook拓展 + python 3.12.2**

文本

描述已自动生成

1. **实验内容**
2. 题目一：采用 scikit-learn 中的 LogisticRegression 逻辑回归模型对 iris 数据集进行二分类。

具体内容：

（1）特征可视化： 任选两个特征和两种类别进行散点图可视化，观察是否线性可分。

（2）模型建立： 使用选取的特征和两种类别建立二分类模型。

（3）输出：决策函数的参数、预测值、分类准确率等。

（4）决策边界可视化：将二分类问题的边界可视化。

1. 题目二：采用 scikit-learn 中的 LogisticRegression 逻辑回归模型对 iris 数据集进行多分类。

具体内容：

（1）模型建立：任选两个特征和全部类别进行散点图可视化，并建立多分类模型。

（2）输出：决策函数的参数、预测值、分类准确率等。

（3）决策边界可视化：将多分类问题的边界可视化。

【讨论一】（选做）不同多分类策略的效果如何？有何差异？

（1）尝试对比 LogisticRegression 中的 multi\_class =’ovr’或’multinomial’两种多分类的差异。

（2）尝试使用 Multiclass classification 中提供的 3 种多分类策略，并对比效果。

提示：进行对比时，要保证数据集划分一致且分析的特征一致。

可从训练集、测试集准确率，和边界可视化角度进行对比。

1. 题目三：采用 scikit-learn 中的 LogisticRegression 逻辑回归模型对非线性数据集进行分类。

具体内容：

（1）数据集：使用 sklearn 自带数据生成器 make\_moons 产生两类数据样本。

（2）特征衍生（数据增强）：使用 sklearn 自带 sklearn.preprocessing.PolynomialFeatures 生成指定阶次的多项式特征，从而得到所有多项式组合成的新特征矩阵，degree 参数任选。

（3）模型建立：在新特征基础上建立逻辑回归二分类模型。

（4）决策边界可视化：绘制决策边界，观察非线性边界的变化。

【讨论二】在不加正则项的情况下，改变特征衍生的特征数量（即 degree 参数），观察决策边界的变化情况，以及训练集和测试集分数，体会模型从欠拟合 ->拟合 ->过拟合的过程。

提示：可使用 for 循环对不同 degree 进行遍历，观察模型的建模结果。

【讨论三】（选做）在讨论二的基础上选择一种模型过拟合的 degree，在模型中分别加入’l1’

和’l2’正则项，观察决策边界的变化情况，以及训练集和测试集分数，体会两种正则项对模型的作用。

【讨论四】可尝试手动调整 degree、正则项系数 C 和正则项种类，寻找使模型泛化性能最好的一组参数。

提示：手动调参采用“单一变量”原则。 可先设定正则项种类（如‘l1’） 和正则项系数 C（如默认） ，

再人为设定特征最高阶次 degree 的范围进行 degree 寻优， 在选定的 degree 和‘l1’正则化后， 设定正则项系数 C 的范围进行寻优。

1. 题目四： 使用 numpy 编写逻辑回归算法，对 iris 数据进行二分类。

具体内容：

（ 1）任选两个特征和两个类别进行二分类。

（ 2）输出：决策函数的参数、预测值、分类准确率等。

（ 3）可视化：选取两个特征进行散点图可视化，并可视化决策边界。

1. 题目五（ 选做）： 使用 numpy 编写逻辑回归算法，对 iris 数据进行多分类。

具体内容：输出决策函数的参数、预测值、分类准确率等。

提示：

（ 1）可采用 OVR、 OVO、 ECOC 策略。

（ 2）可采用 CrossEntropy Loss + softmax 策略。

a）需将三个类别（如 0,1,2）进行 one-hot 编码。

b）每个线性分类器对应一组模型参数， 3 个线性分类器对应 3 组模型参数。

c）可通过 softmax 回归计算多种类别的概率（ K 种类别概率和为 1）。

d）通过最小化 CrossEntropy Loss 的梯度下降算法进行分类器参数寻优。

1. **实验情况**
2. 题目一
3. # 数据集与使用库的导入
4. from sklearn.datasets import load\_iris
5. from sklearn.linear\_model import LogisticRegression
6. import matplotlib.pyplot as plt
7. from sklearn.model\_selection  import train\_test\_split
8. import numpy as np
9. import sklearn
10. plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示
11. plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

特征可视化：任选两个特征和两种类别进行散点图可视化，观察是否线性可分

# 加载 iris 数据集

iris = load\_iris()

# 查看数据集的描述

print("数据集描述：")

print(iris.DESCR)

# 查看特征名

print("\n特征名：")

print(iris.feature\_names)

# 查看标签名

print("\n标签名：")

print(iris.target\_names)

# 查看数据样本量

print("\n数据样本量：")

print("数据矩阵维度：", iris.data.shape)

print("标签向量维度：", iris.target.shape)

# 鸢尾花数据集的详细内容：

# 特征（输入）：花萼长度（sepal length）：以厘米为单位

# 特征（输入）：花萼宽度（sepal width）：以厘米为单位

# 特征（输入）：花瓣长度（petal length）：以厘米为单位

# 特征（输入）：花瓣宽度（petal width）：以厘米为单位

# 标签（输出）：类别（species）：包含三个类别，

# 分别是山鸢尾（setosa）、变色鸢尾（versicolor）和维吉尼亚鸢尾（virginica）

# 数据集中共有 150 个样本，每个样本有 4 个特征，分别是花萼长度、花萼宽度、花瓣长度、花瓣宽度，标签为鸢尾花的种类。

# 数据集中的数据矩阵的维度为 150×4，标签向量的维度为 150×1。

# 其中，标签向量中的 0、1、2 分别代表山鸢尾、变色鸢尾和维吉尼亚鸢尾。

# 山鸢尾、变色鸢尾和维吉尼亚鸢尾的样本量分别为 50、50 和 50

# 导入数据集并划分训练集、测试集

X = iris.data[:, :2]  # 使用前两个特征

y = iris.target

# 使用前两个类别的数据

X = X[y != 2]

y = y[y != 2]

class\_names = iris.target\_names[:2] # 这里选择前两类

# 绘制所选择的两个类别的散点图

plt.figure(figsize=(10, 6))

plt.scatter(X[y == 0, 0], X[y == 0, 1], label=class\_names[0], c='r')    # y==0 为第一类

plt.scatter(X[y == 1, 0], X[y == 1, 1], label=class\_names[1], c='b')    # y==1 为第二类

plt.xlabel('花萼长度')

plt.ylabel('花萼宽度')

plt.title('鸢尾花数据集')

plt.legend()

plt.show()

# # 划分训练集和测试集，这里是按照 8:2 的比例划分

# # train\_test\_split的默认行为是随机抽样。如果你的数据已经是有序的，那么随机抽样可能会在训练集和测试集中创建类别的不均衡分布

# # 分别对两个种类的数据集进行划分，而不能直接对整个数据集进行划分

# X1\_train, X1\_test, y1\_train, y1\_test = train\_test\_split(X[:50,:2], y[:50], test\_size=0.2, random\_state=555)

# X2\_train, X2\_test, y2\_train, y2\_test = train\_test\_split(X[50:100,:2], y[50:100], test\_size=0.2, random\_state=555)

# # 下面将两个数据集合并

# X\_train = np.concatenate((X1\_train,X2\_train), axis=0)

# y\_train = np.concatenate((y1\_train,y2\_train), axis=0)

# X\_test = np.concatenate((X1\_test,X2\_test), axis=0)

# y\_test = np.concatenate((y1\_test,y2\_test), axis=0)

# 简洁的话，采用stratify参数，可以直接按照原始数据集的类别比例来划分训练集和测试集

# train\_test\_split函数在没有指定stratify参数时，是进行随机划分的，也就是说从全部数据中随机抽取一定比例的数据作为测试集

# 然而，如果数据集的类别分布不均衡，如本例那样数据是按类别排序的，那么随机划分就会破坏数据的类别分布

# 在train\_test\_split中，可以通过设置stratify参数为目标标签y来实现分层抽样

# 在进行分层抽样时，函数会首先根据目标标签y的类别分布计算出每个类别应该在训练集和测试集中各占的比例，然后按这个比例从每个类别中随机抽取样本

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=555, stratify=y)

输出：

文本

描述已自动生成 图表, 散点图

描述已自动生成

模型建立：使用选取的特征和两种类别建立二分类模型

clf = LogisticRegression(random\_state=0).fit(X\_train, y\_train)

result = clf.predict(X\_test)

plt.figure(figsize=(12, 3))

plt.subplot(121)

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap=plt.cm.Paired)# 绘制测试集散点图

plt.title('测试集真实标签')

plt.subplot(122)

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=result, cmap=plt.cm.Paired)# 绘制测试集散点图

plt.title('测试集预测标签')

plt.show()

图表, 散点图

描述已自动生成

输出：决策函数的参数、预测值、分类准确率等

test\_predict = clf.predict(X\_test)

train\_predict = clf.predict(X\_train)

coef = clf.coef\_

intercept = clf.intercept\_

# 计算训练集准确率，调用的是 sklearn 库中的 accuracy\_score 函数

# 预测标签与真实标签进行比较，相同的数量除以总数即为准确率

print("训练集准确率：", sklearn.metrics.accuracy\_score(y\_train, train\_predict))

print("测试集准确率：", sklearn.metrics.accuracy\_score(y\_test, test\_predict))

print("权重：", coef)

print("截距：", intercept)

print("函数为: y = {:.2f}x1 + {:.2f}x2 + {:.2f},其中x1为花萼长度,x2为花萼宽度".format(coef[0][0], coef[0][1], intercept[0]))

# 所求解的线性方程没有随机性，因为随机数种子random\_state=0，所以每次运行的结果都是一样的

得到结果：

文本

描述已自动生成

决策边界可视化：将二分类问题的边界可视化

X\_max, X\_min = X[:, 0].max(), X[:, 0].min()

xx = np.array([X\_min, X\_max])               # xx 为生成的直线上的点

yy = -coef[0][0] / coef[0][1] \* xx - intercept[0] / coef[0][1]      # 根据直线方程计算 yy

plt.figure(figsize=(12, 4))

plt.subplot(121)

plt.plot(xx, yy, c='g',linewidth=2,linestyle='--')

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap=plt.cm.Paired)   # 绘制测试集散点图, c=y\_test 为真实标签

plt.title('测试集决策边界')

plt.subplot(122)

plt.title('训练集决策边界')

plt.plot(xx, yy, c='g',linewidth=2,linestyle='--')  # xx 为生成的直线上的点, yy 为直线上点的纵坐标

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=plt.cm.Paired)   # 绘制训练集散点图, c=y\_train 为真实标签

plt.show()

图表, 散点图

描述已自动生成

题目二

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import accuracy\_score

plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示

plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

# 题目一中只选择了两个类别，这里我们选择全部三个类别，但是仍然只选择两个特征，即花萼长度和花萼宽度

iris = load\_iris()

X = iris.data[:, :2]  # 使用前两个特征

y = iris.target

class\_names = iris.target\_names # 这里选择全部三类

# 划分训练集和测试集,stratify参数，可以直接按照原始数据集的类别比例来划分训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=555,stratify=y)

任选两个特征和全部类别进行散点图可视化，并建立多分类模型

from sklearn.linear\_model import LogisticRegression

# 指定 penalty='none'，即不使用正则化项，

# multi\_class='multinomial'，即使用 softmax（交叉熵损失函数）

clf = LogisticRegression(random\_state=5,penalty=None,multi\_class='multinomial').fit(X\_train, y\_train)

result = clf.predict(X\_test)

plt.figure(figsize=(12, 3.8))

plt.subplot(121)

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, cmap=plt.cm.Paired)# 绘制测试集散点图

plt.title('测试集真实标签')

plt.subplot(122)

plt.scatter(X\_test[:, 0], X\_test[:, 1], c=result, cmap=plt.cm.Paired)# 绘制测试集散点图

plt.title('测试集预测标签')

plt.show()

图表, 散点图

描述已自动生成

输出：决策函数的参数、预测值、分类准确率等

import sklearn.metrics

# 输出：决策函数的参数、预测值、分类准确率等

test\_predict = clf.predict(X\_test)

train\_predict = clf.predict(X\_train)

coef = clf.coef\_

intercept = clf.intercept\_

print("训练集准确率：", sklearn.metrics.accuracy\_score(y\_train, train\_predict))

print("测试集准确率：", sklearn.metrics.accuracy\_score(y\_test, test\_predict))

print("权重：", coef)

print("截距：", intercept)

print("函数为：y = {:.2f}x1 + {:.2f}x2 + {:.2f}，其中x1为花萼长度，x2为花萼宽度".format(coef[0][0], coef[0][1], intercept[0]))

得到输出：

文本

描述已自动生成

决策边界可视化：将多分类问题的边界可视化

# 定义函数来绘制决策边界

def plot\_decision\_boundary(X, y, model, ax, steps=1000, cmap='Paired',title='决策边界'):

    """

    绘制决策边界, X 为数据集，y 为标签，model 为训练好的模型，ax 为绘制的坐标轴，steps 为网格的步长，cmap 为颜色

    """

    cmap = plt.get\_cmap(cmap)

    # 定义坐标轴范围,并创建网格

    x\_min, x\_max = X[:, 0].min() - 1, X[:, 0].max() + 1

    y\_min, y\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, steps),

                         np.linspace(y\_min, y\_max, steps))

    # 计算 Z，即每个网格点的预测值

    Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)

    # 绘制等高线和训练集散点图

    ax.contourf(xx, yy, Z, cmap=cmap, alpha=0.8)

    scatter = ax.scatter(X[:, 0], X[:, 1], c=y, cmap=cmap, edgecolor='k')

    # 图例

    legend1 = ax.legend(\*scatter.legend\_elements(),

                        title="Classes")

    ax.add\_artist(legend1)

    ax.set\_xlim(X[:,0].min()-0.5, X[:,0].max()+0.5)

    ax.set\_ylim(X[:,1].min()-0.5, X[:,1].max()+0.5)

    ax.set\_xticks(())

    ax.set\_yticks(())

    ax.set\_title(title)

# 绘制训练集决策边界

fig, ax = plt.subplots(figsize=(8,6))

plot\_decision\_boundary(X\_train, y\_train, clf, ax,title='训练集决策边界')

# 绘制测试集决策边界

fig, ax = plt.subplots(figsize=(8,6))

plot\_decision\_boundary(X\_test, y\_test, clf, ax,title='测试集决策边界')

plt.show()

图表, 散点图

描述已自动生成

图表, 散点图

描述已自动生成

讨论一：不同多分类策略的效果如何？有何差异？、

from sklearn.metrics import confusion\_matrix

from sklearn.preprocessing import PolynomialFeatures

# 对比 LogisticRegression 中的 multi\_class =’ovr’或’multinomial’两种多分类的差异

# 创建两个pipeline

ovr\_pipeline = Pipeline([

    ('scaler', StandardScaler()),   # 数据标准化

    ('clf', LogisticRegression(random\_state=5, multi\_class='ovr'))   # 使用ovr策略

])

multinomial\_pipeline = Pipeline([

    ('poly\_features', PolynomialFeatures(degree=5)),  # 设置degree参数为所需的多项式阶数

    ('scaler', StandardScaler()),   # 数据标准化

    ('clf', LogisticRegression(random\_state=5, multi\_class='multinomial'))   # 使用multinomial策略

])

fig, ax = plt.subplots(2, 2)

fig.set\_size\_inches(10, 6)

# 训练模型

ovr\_pipeline.fit(X\_train, y\_train)

# 预测

y\_pred = ovr\_pipeline.predict(X\_train)

# 计算准确率,使用accuracy\_score函数

ovr\_accuracy = accuracy\_score(y\_train, y\_pred)

# 绘制训练集决策边界

plot\_decision\_boundary(X\_train, y\_train, ovr\_pipeline, ax[0][0],title='OvR训练集决策边界,正确率为：'+str(ovr\_accuracy))

# 预测

y\_pred = ovr\_pipeline.predict(X\_test)

# 计算准确率,使用accuracy\_score函数

ovr\_accuracy = accuracy\_score(y\_test, y\_pred)

# print("使用ovr策略的准确率为：", ovr\_accuracy)

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, ovr\_pipeline, ax[0][1],title='OvR测试集决策边界,正确率为：'+str(ovr\_accuracy))

# 训练模型

multinomial\_pipeline.fit(X\_train, y\_train)

# 预测

y\_pred = multinomial\_pipeline.predict(X\_train)

# 计算准确率,使用accuracy\_score函数

multinomial\_accuracy = accuracy\_score(y\_train, y\_pred)

# 绘制训练集决策边界

plot\_decision\_boundary(X\_train, y\_train, multinomial\_pipeline, ax[1][0],title='multinomial训练集决策边界,正确率为：'+str(multinomial\_accuracy))

# 预测

y\_pred = multinomial\_pipeline.predict(X\_test)

# 计算准确率,使用accuracy\_score函数

multinomial\_accuracy = accuracy\_score(y\_test, y\_pred)

# print("使用multinomial策略的准确率为：", multinomial\_accuracy)

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, multinomial\_pipeline, ax[1][1],title='multinomial测试集决策边界,正确率为：'+str(multinomial\_accuracy))

# 观察ovr一对多策略和multinomial多项式逻辑回归策略的决策边界

# 对于本次数据集，ovr的效果较好，每次只需要区分出某一类，训练三个分类器即可。由于数据集较小且结果较接近，暂时不能得出准确且普适的结论

图表, 散点图

描述已自动生成

# 使用 Multiclass classification 中提供的 3 种多分类策略，并对比效果

from sklearn.multiclass import OneVsOneClassifier, OneVsRestClassifier, OutputCodeClassifier

# 创建三个pipeline

# 使用OvO策略

ovo\_pipeline = Pipeline([

    ('scaler', StandardScaler()),   # 数据标准化

    ('clf', OneVsOneClassifier(LogisticRegression(random\_state=5)))   # 使用OvO策略

])

# 使用OvR策略

ovr\_pipeline = Pipeline([

    ('scaler', StandardScaler()),   # 数据标准化

    ('clf', OneVsRestClassifier(LogisticRegression(random\_state=5)))   # 使用OvR策略

])

# 使用 error-correcting output codes 策略

outputcode\_pipeline = Pipeline([

    ('scaler', StandardScaler()),   # 数据标准化

    # 使用OutputCode策略,即使设置了random\_state，每次运行结果也不一样，这是因为OutputCodeClassifier中使用了随机数

    ('clf', OutputCodeClassifier(LogisticRegression(random\_state=15)))

])

# 创建画布

fig, ax = plt.subplots(3, 2)

fig.set\_size\_inches(12, 15)

# 训练模型

ovo\_pipeline.fit(X\_train, y\_train)

# 预测

y\_pred = ovo\_pipeline.predict(X\_train)

ovo\_accuracy = accuracy\_score(y\_train, y\_pred)

# print("使用OvO策略的训练集准确率为：", ovo\_accuracy)

# 绘制训练集决策边界

plot\_decision\_boundary(X\_train, y\_train, ovo\_pipeline, ax[0][0],title='OvO训练集决策边界,正确率为：'+str(ovo\_accuracy))

# 预测

y\_pred = ovo\_pipeline.predict(X\_test)

# 计算准确率,使用accuracy\_score函数

ovo\_accuracy = accuracy\_score(y\_test, y\_pred)

# print("使用OvO策略的测试集准确率为：", ovo\_accuracy)

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, ovo\_pipeline, ax[0][1],title='OvO测试集决策边界,正确率为：'+str(ovo\_accuracy))

# 训练模型

ovr\_pipeline.fit(X\_train, y\_train)

# 预测

y\_pred = ovr\_pipeline.predict(X\_train)

ovr\_accuracy = accuracy\_score(y\_train, y\_pred)

# print("使用OvR策略的训练集准确率为：", ovr\_accuracy)

# 绘制训练集决策边界

plot\_decision\_boundary(X\_train, y\_train, ovr\_pipeline, ax[1][0],title='OvR训练集决策边界,正确率为：'+str(ovr\_accuracy))

# 预测

y\_pred = ovr\_pipeline.predict(X\_test)

# 计算准确率,使用accuracy\_score函数

ovr\_accuracy = accuracy\_score(y\_test, y\_pred)

# print("使用OvR策略的测试集准确率为：", ovr\_accuracy)

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, ovr\_pipeline, ax[1][1],title='OvR测试集决策边界,正确率为：'+str(ovr\_accuracy))

# 训练模型

outputcode\_pipeline.fit(X\_train, y\_train)

# 预测

y\_pred = outputcode\_pipeline.predict(X\_train)

outputcode\_accuracy = accuracy\_score(y\_train, y\_pred)

# print("使用OutputCode策略的训练集准确率为：", outputcode\_accuracy)

# 绘制训练集决策边界

plot\_decision\_boundary(X\_train, y\_train, outputcode\_pipeline, ax[2][0],title='OutputCode训练集决策边界,正确率为：'+str(outputcode\_accuracy))

# 预测

y\_pred = outputcode\_pipeline.predict(X\_test)

# 计算准确率,使用accuracy\_score函数

outputcode\_accuracy = accuracy\_score(y\_test, y\_pred)

# print("使用OutputCode策略的测试集准确率为：", outputcode\_accuracy)

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, outputcode\_pipeline, ax[2][1],title='OutputCode测试集决策边界,正确率为：'+str(outputcode\_accuracy))

plt.show()

# fig, ax = plt.subplots(1, 3)

# fig.set\_size\_inches(10, 4)

# 绘制OvO策略的混淆矩阵

# y\_pred = ovo\_pipeline.predict(X\_test)

# cm\_ovo = confusion\_matrix(y\_test, y\_pred)

# im\_ovo = ax[0].imshow(cm\_ovo, interpolation='nearest', cmap=plt.cm.Blues)

# ax[0].set\_title('OvO混淆矩阵')

# fig.colorbar(im\_ovo, ax=ax[0], shrink=0.6)

# # 绘制OvR策略的混淆矩阵

# y\_pred = ovr\_pipeline.predict(X\_test)

# cm\_ovr = confusion\_matrix(y\_test, y\_pred)

# im\_ovr = ax[1].imshow(cm\_ovr, interpolation='nearest', cmap=plt.cm.Blues)

# ax[1].set\_title('OvR混淆矩阵')

# fig.colorbar(im\_ovr, ax=ax[1], shrink=0.6)

# 绘制OutputCode策略的混淆矩阵

# y\_pred = outputcode\_pipeline.predict(X\_test)

# cm\_outputcode = confusion\_matrix(y\_test, y\_pred)

# im\_outputcode = ax[2].imshow(cm\_outputcode, interpolation='nearest', cmap=plt.cm.Blues)

# ax[2].set\_title('OutputCode混淆矩阵')

# fig.colorbar(im\_outputcode, ax=ax[2], shrink=0.6)

plt.tight\_layout()

plt.show()

# 对比发现OVO的精度最高，因为其原理是训练n(n-1)/2个分类器，分类结果更好。

# 在多次尝试后发现，ECOC的结果并不稳定

图表

中度可信度描述已自动生成

题目三

1. import matplotlib.pyplot as plt
2. import numpy as np
3. from sklearn import datasets, linear\_model
4. from sklearn.pipeline import Pipeline
5. from sklearn.model\_selection import train\_test\_split

数据集：使用 sklearn 自带数据生成器 make\_moons 产生两类数据样本

plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示

plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

plt.title("moons datasets, n\_samples=500,noisy=0.2")

X, y = datasets.make\_moons(n\_samples=500, noise=0.2,random\_state=520)

plt.scatter(X[:,0],X[:,1],c=y,marker='o',s=28,edgecolors='k')

plt.show()

# 划分训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=555, stratify=y)# 这里也需要设置分层抽样

**图表, 散点图

描述已自动生成**

**特征衍生（数据增强）：使用sklearn 自带的**

**sklearn.preprocessing.PolynomialFeatures 生成指定阶次的多项式特征，从而得到所有多项式组合成的新特征矩阵，degree 参数任选。**

**模型建立：在新特征基础上建立逻辑回归二分类模型**

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import PolynomialFeatures

#  特征衍生（数据增强）：使用 sklearn 自带 sklearn.preprocessing.PolynomialFeatures 生成指定阶次的多项式特征，从而得到所有多项式组合成的新特征矩阵

# 使用了 Pipeline，可以将多个算法串联起来,# 创建一个包含特征衍生和逻辑回归的 pipeline

pipeline = Pipeline([

    ("poly\_features", PolynomialFeatures(degree=3)),    # 特征衍生

    ("log\_reg", LogisticRegression()),

])

pipeline.fit(X\_train, y\_train)

print('Test score: ', pipeline.score(X\_test, y\_test))   # 这里的 score 是 accuracy，即正确率

**决策边界可视化：绘制决策边界，观察非线性边界的变化**

def plot\_decision\_boundary(X, y, model, ax, title=''):

    # 生成网格采样点

    h = 0.02  # 网格步长

    x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

    y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

    xx, yy = np.meshgrid(np.arange(x\_min, x\_max, h), np.arange(y\_min, y\_max, h))

    # 对网格中的点进行预测

    Z = model.predict(np.c\_[xx.ravel(), yy.ravel()])

    Z = Z.reshape(xx.shape)

    # 绘制等高线图

    ax.contourf(xx, yy, Z, cmap=plt.cm.Paired)

    ax.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=plt.cm.Paired)

    ax.set\_xlabel('Feature 1')

    ax.set\_ylabel('Feature 2')

    ax.set\_title(title)

# 绘制训练集决策边界

fig, ax = plt.subplots(1,2)

fig.set\_size\_inches(15, 5)

plot\_decision\_boundary(X\_train, y\_train, pipeline, ax[0], title='训练集决策边界,正确率：%.2f' % pipeline.score(X\_train, y\_train))

# 绘制测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, pipeline, ax[1], title='测试集决策边界,正确率：%.2f' % pipeline.score(X\_test, y\_test))

plt.show()

**图表

描述已自动生成**

**讨论二：在不加正则项的情况下，改变特征衍生的特征数量（即 degree 参数），观察决策边**

**界的变化情况，以及训练集和测试集分数，体会模型从欠拟合 ->拟合 ->过拟合的过程。**

**提示：可使用 for 循环对不同 degree 进行遍历，观察模型的建模结果。可通过绘制训练集和测试集分数曲线帮助观察**

degrees = range(1, 31)  # 不同的 degree 值

train\_scores = []

test\_scores = []

num\_cols = 3  # 每行的子图数目

num\_rows = int(np.ceil(len(degrees) / num\_cols))  # 总行数

fig, axs = plt.subplots(num\_rows, num\_cols, figsize=(15, 4 \* num\_rows))

axs = axs.flatten()

for i, degree in enumerate(degrees):

    # 特征衍生

    poly = PolynomialFeatures(degree=degree)

    # 建立逻辑回归模型

    log\_reg = LogisticRegression()

    # 创建一个包含特征衍生和逻辑回归的 pipeline

    pipeline = Pipeline([

        ("poly\_features", poly),

        ("log\_reg", log\_reg),

    ])

    # 训练模型

    pipeline.fit(X\_train, y\_train)

    # 计算训练集和测试集的分数

    train\_score = pipeline.score(X\_train, y\_train)

    test\_score = pipeline.score(X\_test, y\_test)

    # 保存训练集和测试集的分数

    train\_scores.append(train\_score)

    test\_scores.append(test\_score)

    # 绘制决策边界

    plot\_decision\_boundary(X\_test, y\_test, pipeline, axs[i], title='Degree {}'.format(degree))

    axs[i].set\_title('Degree {}，test\_accu = {:.4f}，train\_accu = {:.4f}'.format(degree, test\_score, train\_score))

    axs[i].set\_xticks([])

    axs[i].set\_yticks([])

# 移除多余的子图

for j in range(len(degrees), num\_rows \* num\_cols):

    fig.delaxes(axs[j])

plt.tight\_layout()

plt.show()

# 绘制训练集和测试集分数曲线

plt.plot(degrees, train\_scores, 'bo-', label='Train Score')

plt.plot(degrees, test\_scores, 'ro-', label='Test Score')

plt.xlabel('Degree')

plt.ylabel('Accuracy')

plt.legend()

plt.title('随着Degree增加，模型准确率的变化')

plt.show()

**图片包含 背景图案

描述已自动生成**

**图表, 折线图, 直方图

描述已自动生成**

决策边界的变化：随着 degree 值的增加，决策边界变得更加复杂。较小的 degree 值（如1、2、3）对数据集的拟合能力较差，决策边界过于简单，可能导致欠拟合。随着 degree 值的增加，决策边界更加弯曲，可以更好地适应数据集，但在一定程度上也会导致过拟合。

模型性能：从训练集和测试集的分数曲线可以看出，随着 degree 值的增加，训练集的分数逐渐增加，而测试集的分数则先增加后减小。在低 degree 值时，模型的复杂度较低，导致欠拟合，训练集和测试集的分数都较低。随着 degree 值的增加，模型的复杂度增加，使得模型能够更好地拟合训练集，因此训练集的分数逐渐增加。然而，当 degree 值过高时，模型过于复杂，出现过拟合现象，导致测试集的分数开始下降。

**讨论三：在讨论二的基础上选择一种模型过拟合的 degree，在模型中分别加入’l1’和’l2’正则项，观察决策边界的变化情况，以及训练集和测试集分数，体会两种正则项对模型的作用**

degree = 21  # 过拟合的 degree 值

# （2）特征衍生

poly = PolynomialFeatures(degree=degree)

# 建立逻辑回归模型

log\_reg\_l1 = LogisticRegression(penalty='l1', solver='liblinear')

log\_reg\_l2 = LogisticRegression(penalty='l2')

# 创建包含特征衍生和逻辑回归的 pipeline

pipeline\_l1 = Pipeline([

    ("poly\_features", poly),

    ("log\_reg\_l1", log\_reg\_l1),

])

pipeline\_l2 = Pipeline([

    ("poly\_features", poly),

    ("log\_reg\_l2", log\_reg\_l2),

])

# 训练模型

pipeline\_l1.fit(X\_train, y\_train)

pipeline\_l2.fit(X\_train, y\_train)

# 在测试集上进行评估

score\_l1 = pipeline\_l1.score(X\_test, y\_test)

score\_l2 = pipeline\_l2.score(X\_test, y\_test)

# print('Test score (L1 regularization):', score\_l1)

# print('Test score (L2 regularization):', score\_l2)

# 绘制决策边界

fig, axs = plt.subplots(1, 2, figsize=(12, 5))

plot\_decision\_boundary(X\_test, y\_test, pipeline\_l1, axs[0], title='L1 Regularization,test score: %.4f,train score: %.4f' % (score\_l1, pipeline\_l1.score(X\_train, y\_train)))

plot\_decision\_boundary(X\_test, y\_test, pipeline\_l2, axs[1], title='L2 Regularization,test score: %.4f,train score: %.4f' % (score\_l2, pipeline\_l2.score(X\_train, y\_train)))

plt.tight\_layout()

plt.show()

# 通过观察决策边界的变化和模型的性能表现，我们可以发现：

# L1 正则化可以使得模型的决策边界更加稀疏，即更多的特征系数为 0，从而达到特征选择的目的

# L2 正则化可以使得模型的决策边界更加平滑，即特征系数更加均匀，从而达到防止过拟合的目的

# 对比加入 L1 正则化和 L2 正则化前后的模型性能，我们可以发现：

# L1 正则化可以使得模型的准确率上升，但是训练集和测试集的准确率差距变大，即模型略微过拟合

# L2 正则化没有改变模型的准确率

**图表, 散点图

描述已自动生成**

**讨论四：可尝试手动调整 degree、正则项系数 C 和正则项种类，寻找使模型泛化性能最好的一组参数。**

from sklearn.model\_selection import GridSearchCV

# 设置待调优的参数范围

param\_grid = {

    'poly\_features\_\_degree': range(1, 21),  # 不同的 degree 值

    'log\_reg\_\_penalty': ['l1', 'l2'],  # 正则项种类

    'log\_reg\_\_C': [0.01, 0.1, 1, 10, 100],  # 正则项系数 C

}

# 创建一个包含特征衍生和逻辑回归的 pipeline

pipeline = Pipeline([

    ("poly\_features", PolynomialFeatures()),

    ("log\_reg", LogisticRegression()),

])

# 使用网格搜索进行参数调优

grid\_search = GridSearchCV(pipeline, param\_grid=param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

# 获取最佳参数组合和最佳模型

best\_params = grid\_search.best\_params\_

best\_model = grid\_search.best\_estimator\_

print("Best Parameters: ", best\_params)

print("Best Score: ", grid\_search.best\_score\_)

# 在测试集上进行评估

test\_score = best\_model.score(X\_test, y\_test)

print("Test Score: ", test\_score)

# 绘制最好模型训练集决策边界

fig, ax = plt.subplots(1,2,figsize=(15, 5))

plot\_decision\_boundary(X\_train, y\_train, best\_model, ax[0], title='Best Model train score: %.2f' % best\_model.score(X\_train, y\_train))

# 绘制最好模型测试集决策边界

plot\_decision\_boundary(X\_test, y\_test, best\_model, ax[1], title='Best Model test score: %.2f' % test\_score)

plt.tight\_layout()

plt.show()

**图表, 散点图

描述已自动生成**

题目四

import numpy as np

import matplotlib.pyplot as plt

from scipy.linalg import expm

from sklearn.datasets import load\_iris

from sklearn.model\_selection  import train\_test\_split

from sklearn.metrics import accuracy\_score

from sklearn.metrics import confusion\_matrix

plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示

plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

# 加载 iris 数据集

iris = load\_iris()

X = iris.data[:, :2]  # 使用前两个特征

y = iris.target

X = X[y != 2]   # 使用前两个类别的数据

y = y[y != 2]

# X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=555, stratify=y)

# 设置随机种子以保证结果可复现

np.random.seed(555)

# 设置测试集的比例

test\_ratio = 0.2

# 确定数据集中有哪些类别，手写实现了stratify参数的按类别划分功能

classes = np.unique(y)

# 初始化训练集和测试集

X\_train = []

X\_test = []

y\_train = []

y\_test = []

# 对每一个类别进行操作

for c in classes:

    # 找到所有属于这个类别的样本

    idx = np.where(y == c)[0]

    # 随机打乱这些样本

    np.random.shuffle(idx)

    # 计算测试集的大小

    test\_size = int(len(idx) \* test\_ratio)

    # 根据打乱的索引数组划分训练集和测试集

    X\_test.extend(X[idx[:test\_size]])

    y\_test.extend(y[idx[:test\_size]])

    X\_train.extend(X[idx[test\_size:]])

    y\_train.extend(y[idx[test\_size:]])

# 将训练集和测试集从列表转换为 numpy 数组

X\_train = np.array(X\_train)

X\_test = np.array(X\_test)

y\_train = np.array(y\_train)

y\_test = np.array(y\_test)

X\_train.shape, X\_test.shape, y\_train.shape, y\_test.shape

# 定义逻辑回归模型

class LogisticRegression:

    def \_\_init\_\_(self, lr=0.01, num\_iter=100000, fit\_intercept=True):

        self.lr = lr

        self.num\_iter = num\_iter

        self.fit\_intercept = fit\_intercept

    def \_\_add\_intercept(self, X):# 为 X 增加一列全 1 的特征，构造类时 fit\_intercept 设为 True 时会用到

        intercept = np.ones((X.shape[0], 1))

        return np.concatenate((intercept, X), axis=1)

    def \_\_sigmoid(self, z):# sigmoid 函数，构造类时会用到

        return 1 / (1 + np.exp(-z))

    def \_\_loss(self, h, y):

        return (-y \* np.log(h) - (1 - y) \* np.log(1 - h)).mean()

    def fit(self, X, y):

        # 拟合函数，X 为训练集的特征，y 为训练集的标签

        if self.fit\_intercept:

            X = self.\_\_add\_intercept(X)

        self.theta = np.zeros(X.shape[1])

        for i in range(self.num\_iter):

            # 计算预测值

            z = np.dot(X, self.theta)

            h = self.\_\_sigmoid(z)

            gradient = np.dot(X.T, (h - y)) / y.size    # 计算梯度，y.size 为样本数量

            self.theta -= self.lr \* gradient

    def predict\_prob(self, X):

        # 预测函数，X 为测试集的特征

        if self.fit\_intercept:

            X = self.\_\_add\_intercept(X)

        return self.\_\_sigmoid(np.dot(X, self.theta))

    def predict(self, X, threshold=0.5):

        return self.predict\_prob(X) >= threshold    # 大于等于阈值的返回 True

# Stratified train/test split

def stratified\_train\_test\_split(X, y, test\_ratio, random\_seed):

    # 这个函数的作用是按类别划分训练集和测试集，test\_ratio 是测试集的比例，random\_seed 是随机种子

    np.random.seed(random\_seed)

    classes = np.unique(y)

    X\_train, X\_test, y\_train, y\_test = [], [], [], []

    for c in classes:

        idx = np.where(y == c)[0]

        np.random.shuffle(idx)

        test\_size = int(len(idx) \* test\_ratio)

        X\_test.extend(X[idx[:test\_size]])

        y\_test.extend(y[idx[:test\_size]])

        X\_train.extend(X[idx[test\_size:]])

        y\_train.extend(y[idx[test\_size:]])

    return np.array(X\_train), np.array(X\_test), np.array(y\_train), np.array(y\_test)

iris = load\_iris()

X = iris.data[:, :2]

y = (iris.target != 0) \* 1

# 划分数据集

X\_train, X\_test, y\_train, y\_test = stratified\_train\_test\_split(X, y, test\_ratio=0.2, random\_seed=555)

# 训练模型

model = LogisticRegression(lr=0.1, num\_iter=10000)

model.fit(X\_train, y\_train)

# 输出：决策函数的权重、偏置和函数表达式

coef\_ = model.theta[1:]

intercept\_ = model.theta[0]

print(f'决策函数的权重：{coef\_}')

print(f'决策函数的偏置：{intercept\_}')

print(f'决策函数表达式：y = {coef\_[0]} \* x1 + {coef\_[1]} \* x2 + {intercept\_}')

# 预测训练集

preds = model.predict(X\_train)

accuracy = (preds == y\_train).mean()

print(f'训练集 Accuracy: {accuracy}')

# 预测测试集

preds = model.predict(X\_test)

accuracy = (preds == y\_test).mean()

print(f'测试集 Accuracy: {accuracy}')

# 可视化

plt.figure(figsize=(6, 4))

plt.scatter(X[y == 0][:, 0], X[y == 0][:, 1], color='b', label='0')

plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], color='r', label='1')

plt.legend()

x1\_min, x1\_max = X[:,0].min(), X[:,0].max(),

x2\_min, x2\_max = X[:,1].min(), X[:,1].max(),

xx1, xx2 = np.meshgrid(np.linspace(x1\_min, x1\_max), np.linspace(x2\_min, x2\_max))

grid = np.c\_[xx1.ravel(), xx2.ravel()]

probs = model.predict\_prob(grid).reshape(xx1.shape)

plt.contour(xx1, xx2, probs, [0.5], linewidths=1, colors='black');

plt.title('划分两个类别的数据集')

plt.show()

运行结果：

文本

描述已自动生成

图表, 散点图

描述已自动生成

题目五（扩展）

import sklearn

from sklearn.datasets import load\_iris

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

import numpy as np

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import accuracy\_score

plt.rcParams['font.family'] = 'SimHei'      # 中文正常显示

plt.rcParams['axes.unicode\_minus'] = False  # 符号正常显示

# 在下面这个实现中，主要使用了 CrossEntropy Loss（交叉熵损失） + softmax（激活函数） 策略来处理多分类任务

class MultiClassLogisticRegression:

    def \_\_init\_\_(self, coef\_shape, intercept\_shape, learning\_rate=0.01, epochs=1000):

        self.coef = np.random.randn(\*coef\_shape)

        self.intercept = np.random.randn(\*intercept\_shape)

        self.learning\_rate = learning\_rate

        self.epochs = epochs

    @staticmethod   # staticmethod 修饰的函数，可以直接通过类名调用，不需要实例化对象

    def one\_hot(y):# 将标签进行 one-hot 编码

        return np.eye(3)[y.flatten()]

    def predict(self, X):

        # X.shape[1] 为输入矩阵的第二个维度，即特征的数量

        # 在预测阶段，模型会将输入特征和系数矩阵相乘并加上截距，得到一个预测值

        # 然后对这个预测值应用 softmax ，得到每个类别的预测概率。最后选概率最大的类别作为预测结果。

        if X.shape[1] != self.coef.shape[0]:

            raise ValueError("输入矩阵的第二个维度必须与coef匹配！")

        value = np.dot(X, self.coef) + self.intercept

        proba = np.exp(value)

        proba = proba / np.sum(proba, axis=1, keepdims=True)    # softmax 应用，得到每个类别的预测概率

        result = np.argmax(proba, axis=1).reshape(-1, 1)

        return value, proba, result

    @staticmethod

    def cross\_entropy\_loss(label, proba):

        # 交叉熵损失函数: 这是逻辑回归中常用的损失函数，可衡量模型预测结果与真实结果之间的一致性

        # 通过最小化交叉熵损失函数，可以使模型的预测结果更接近真实结果

        # label 为 one-hot 编码后的标签，proba 为预测的概率

        loss = -np.sum(label \* np.log(proba), axis=0)   # 交叉熵损失越小，说明预测的结果越准确

        return loss.sum()

    def update(self, X, label, lr=0.001):

        # 更新参数，lr 为学习率，X 为输入矩阵，label 为 one-hot 编码后的标签

        value, proba, \_ = self.predict(X)

        self.coef -= lr \* np.dot(X.T, proba - label)

        self.intercept -= lr \* np.sum(proba - label)

        return self.cross\_entropy\_loss(label, proba)

    def train(self, X\_train, y\_train):

        lost\_list = []

        for epoch in range(self.epochs):

            # 每次迭代，都会调用一次 update 函数，更新模型参数

            \_lr = self.learning\_rate \* np.power((1 - float(epoch) / self.epochs), 4)# 学习率衰减

            loss = self.update(X\_train, self.one\_hot(y\_train), lr=\_lr)  # 更新参数，返回交叉熵损失

            if not epoch % int(self.epochs / 100):

                lost\_list.append(loss)

            if not epoch % int(self.epochs / 10):

                print("epoch:", epoch, "交叉熵损失值:", loss)

        # 每次调用训练函数，都绘制一次损失函数的变化曲线

        plt.plot(lost\_list)

        plt.title("交叉熵损失值随着训练轮次的变化曲线")

        plt.xlabel("epoch")

        plt.ylabel("loss")

        plt.show()

    def plot\_decision\_boundary(self, X, y):

        # 绘制决策边界

        color = ["r", "g", "b"]

        marker = ["o", "v", "x"]

        class\_label = np.unique(y)

        # 为每个类别分配一种颜色

        cmap = ListedColormap(color[: len(class\_label)])

        x1\_min, x2\_min = np.min(X, axis=0)

        x1\_max, x2\_max = np.max(X, axis=0)

        x1 = np.arange(x1\_min - 1, x1\_max + 1, 0.02)

        x2 = np.arange(x2\_min - 1, x2\_max + 1, 0.02)

        # 生成网格点坐标矩阵

        X1, X2 = np.meshgrid(x1, x2)

        # 将网格点坐标矩阵作为输入，调用 predict 函数，得到预测结果

        Z = np.array(self.predict(np.array([X1.ravel(), X2.ravel()]).T)[2]).reshape(X1.shape)

        plt.contourf(X1, X2, Z, cmap=cmap, alpha=0.5)

        for i, class\_ in enumerate(class\_label):

            plt.scatter(x=X[y == class\_, 0], y=X[y == class\_, 1], c=cmap.colors[i], label=class\_, marker=marker[i])

        plt.legend()

        plt.show()

    def stratified\_train\_test\_split(self, X, y, test\_ratio, random\_seed):

        # 分层划分训练集和测试集，用于替代 train\_test\_split 函数

        np.random.seed(random\_seed)

        classes = np.unique(y)

        X\_train, X\_test, y\_train, y\_test = [], [], [], []

        for c in classes:

            # 对每个类别，都按照相同的比例划分训练集和测试集

            idx = np.where(y == c)[0]

            np.random.shuffle(idx)  # 打乱索引

            test\_size = int(len(idx) \* test\_ratio)

            X\_test.extend(X[idx[:test\_size]])

            y\_test.extend(y[idx[:test\_size]])

            X\_train.extend(X[idx[test\_size:]])

            y\_train.extend(y[idx[test\_size:]])

        return np.array(X\_train), np.array(X\_test), np.array(y\_train), np.array(y\_test)

    @staticmethod

    def draw\_confusion\_matrix(y\_test, test\_pre):

        confusion = confusion\_matrix(y\_test, test\_pre)

        print(confusion)

        plt.imshow(confusion, cmap=plt.cm.Blues)

        indices = range(len(confusion))

        classes = list(set(y\_test.flatten()))

        classes.sort()

        plt.xticks(indices, classes)

        plt.yticks(indices, classes)

        for i in range(len(confusion)):

            for j in range(len(confusion[i])):

                plt.text(j, i, confusion[i][j], c='y', fontsize=13)

        plt.colorbar()

        plt.xlabel('predict')

        plt.ylabel('true')

        plt.show()

# 加载数据集

X, y = load\_iris(return\_X\_y=True)

features = [1,3]

X = X[:, features]

# 初始化模型

model = MultiClassLogisticRegression(coef\_shape=(2, 3), intercept\_shape=(1, 3))

# 分割数据集

X\_train, X\_test, y\_train, y\_test = model.stratified\_train\_test\_split(X, y, test\_ratio=0.2, random\_seed=1)

# 训练模型

model.train(X\_train, y\_train)

# 预测

preds = model.predict(X\_test)[2]

# 绘制混淆矩阵

model.draw\_confusion\_matrix(y\_test, preds)

# 计算准确率

print("准确率:", accuracy\_score(y\_test, preds))

# 绘制决策边界

model.plot\_decision\_boundary(X\_test, y\_test)

# 从混淆矩阵

# [[10  0  0]

#  [ 0 10  0]

#  [ 0  2  8]]

文本

描述已自动生成

图表

描述已自动生成 图片包含 图表

描述已自动生成

图表, 散点图

描述已自动生成