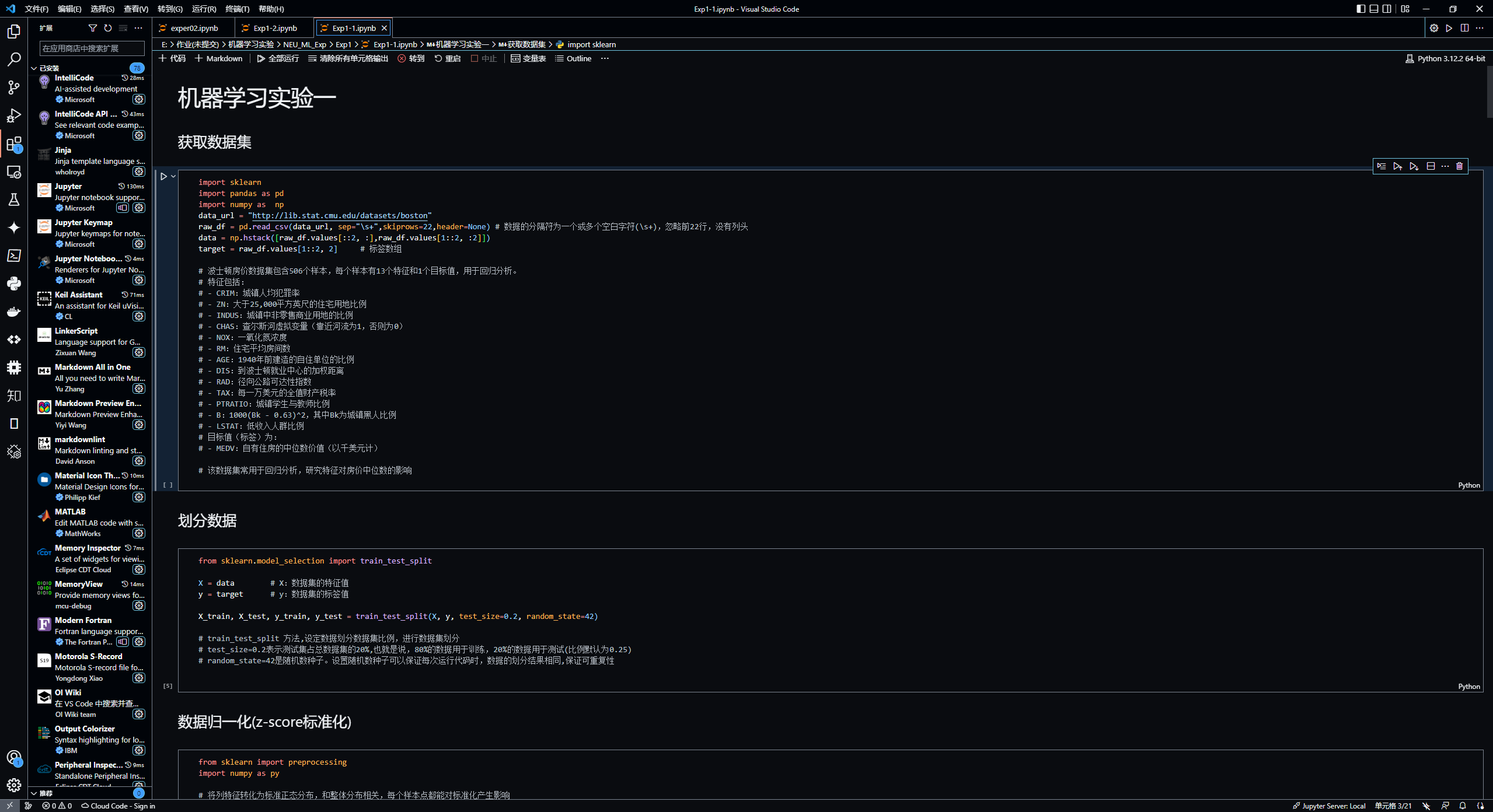
**机器学习应用实践（实验二） —逻辑回归**

**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**



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

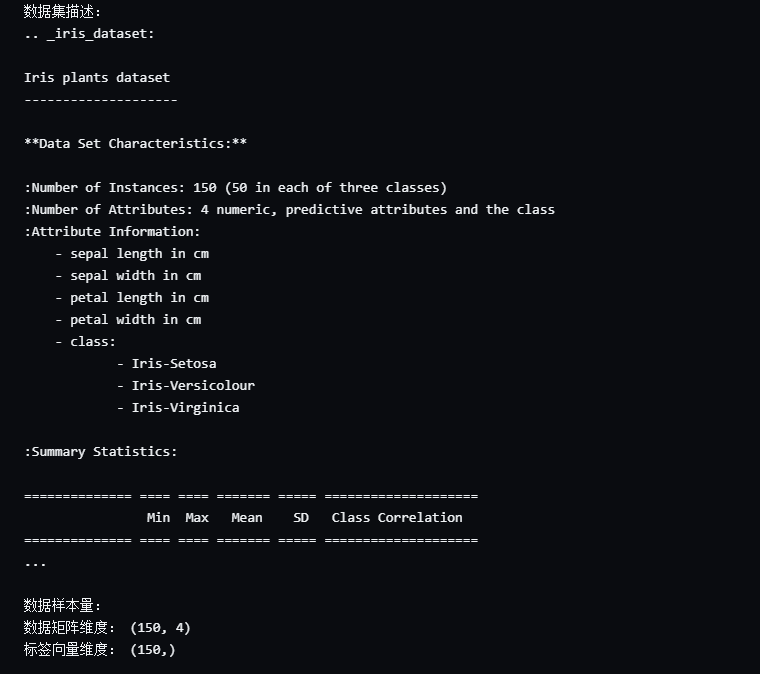
# 然而，如果数据集的类别分布不均衡，如本例那样数据是按类别排序的，那么随机划分就会破坏数据的类别分布

# 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：

 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")

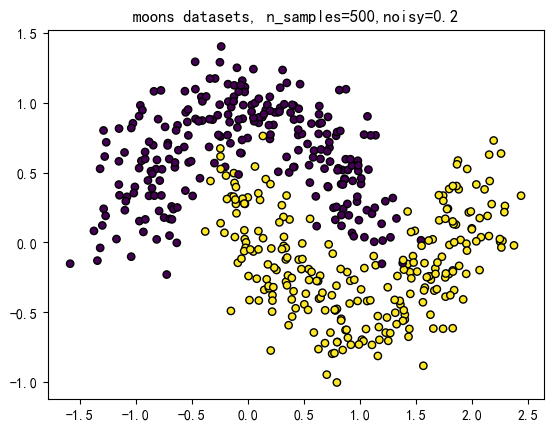
X, y = datasets.make\_moons(n\_samples=500, noise=0.2,random\_state=520)

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

****

**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))

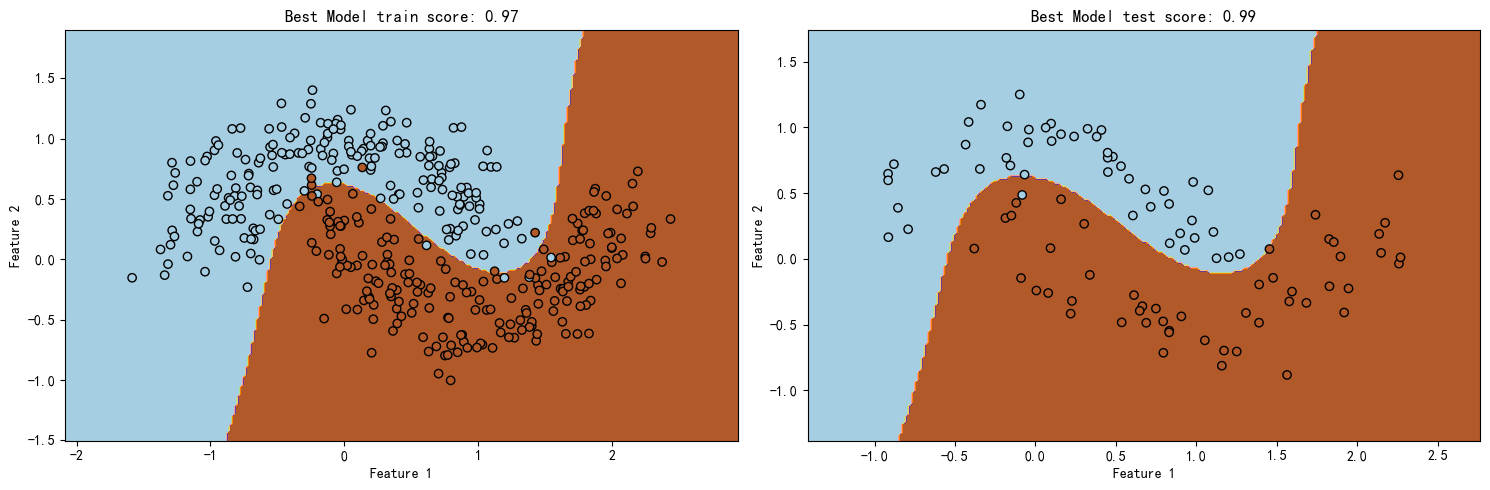
plot\_decision\_boundary(X\_train, y\_train, best\_model, ax[0], title='Best Model train score: %.2f' % best\_model.score(X\_train, y\_train))

# 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()

****

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()]

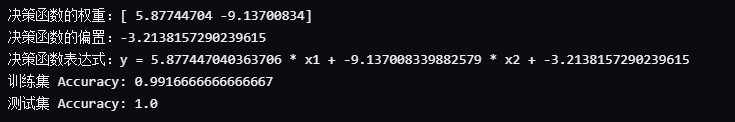
probs = model.predict\_prob(grid).reshape(xx1.shape)

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：



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