**机器学习应用实践（实验三） —支持向量机**

**工业智能2201班 刘天行 20225354**

1. **实验目的**

1、 熟悉支持向量机算法的原理。

2、 掌握 Scikit-Learn 中支持向量机算法的使用。

3、掌握支持向量机四种核函数对比及使用场合。

4、 掌握 Scikit-Learn 中提供的交叉验证与模型评估方法。

5、 掌握网格搜索的原理及 Scikit-Learn 中 GridSearchCV 的使用。

6、熟悉 Scikit-Learn 自带分类数据集（乳腺癌威斯康星州数据集）的使用。

7、 具备使用 python 实现支持向量机 SMO 算法的编程能力。

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**

**1、题目一： 采用 scikit-learn 中的线性 SVM 对 iris 数据集进行二分类。**

具体内容：

（1） 选取两个特征和两类数据使用 scikit-learn 中的 SVM 进行二分类。

（2）输出：决策边界的参数和截距、 支持向量等。

（3）可视化：通过散点图可视化数据样本（之前选择的两个特征） ，并画出决策边界和 2 个最大间隔边界，标出支持向量。

【讨论一】 选取的两个特征能否线性可分？若线性可分， 可选择 scikit-learn 中何种 SVM 进行建

模？若线性不可分，可选择 scikit-learn 中何种 SVM 进行建模？

【讨论二】 SVM 中的惩罚系数 C 对模型有何影响？

（1）尝试改变惩罚系数 C，分析其变化对应间隔宽度、支持向量数量的变化趋势，并解释原因。

（2） 尝试改变惩罚系数 C，分析其对 iris 分类模型性能的影响，并解释原因。

**2、题目二（选做） ： 采用不同的 SVM 核函数对多种类型数据集进行二分类。**

具体内容：

（1） 使用 scikit-learn 中提供的样本生成器 make\_blobs、 make\_classification、 make\_moons、

make\_circles 生成一系列线性或非线性可分的二类别数据（数据量任取） 。

（2） 建模： 分别将 SVM 中四种核函数（线性核、多项式核、高斯核、 S 形核） 用于上述四种

数据集。

提示： 对于每一种核函数，选择最适合的核参数（如 RBF 核中 gamma、多项式核中 degree 等） 。

可通过超参数曲线帮助选择超参数。

（3）可视化：通过散点图可视化数据样本，并画出 SVM 模型的决策边界。

（4）模型评价： 分类准确率。

【讨论三】 如何选择最优超参数？

为每种模型选择适合的核函数及核参数，参数寻优方式自选。

【讨论四】 不同核函数在不同数据集上表现如何？

通过观察不同核函数在不同数据集上的决策边界和分类准确率，分析不同核函数的适用场合。

**3、题目三： 使用 scikit-learn 中的 SVM 分类器对乳腺癌威斯康星州数据集进行分类。**

（1） 导入数据集： 乳腺癌威斯康星州数据集是 sklearn 中自带的数据集（load\_breast\_cancer）。

通过查看数据量和维度、 特征类型（离散 or 连续） 、特征名、标签名、标签分布情况、 数据集

的描述等信息了解数据集。

（2） 建模：分别使用四种核函数对数据集进行分类。

（3） 模型评价： 每种核函数下的分类准确率、计算时间等。

【讨论五】四种核函数在这个数据集上表现如何？

提示： 不要求可视化，从准确率上判断即可。

【讨论六】 SVM 是否需要进行数据归一化处理？ 数据归一化对核函数有何影响？

**4、题目四： 编写 SMO 算法实现线性 SVM 分类器，对 iris 数据集进行二分类。**

具体内容：

（1） 选取两个特征和两类数据进行二分类。

注意：二分类标签为 1 和-1。

（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（参考程序模板： SVM\_numpy\_template.py）

（5） 输出： SVM 对偶问题目标函数的最优解𝛼， 决策函数的参数和截距，支持向量等。

（6）可视化：通过散点图可视化训练数据样本，并画出决策面和 2 个最大间隔面，标出支持向

量（包括间隔上和间隔内的样本） ，能够帮助检验算法正确性。

（7）测试集数据进行预测， 评估模型性能。

【讨论七】 描述软间隔 SVM 中的 C 参数、拉格朗日乘子α、支持向量与最优决策面和间隔区域之间的关系。

1. **实验情况**

**1、题目一： 采用 scikit-learn 中的线性 SVM 对 iris 数据集进行二分类。**

**准备工作：**

from sklearn.datasets import load\_iris

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

import matplotlib.pyplot as plt

import matplotlib as mpl

import numpy as np

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.inspection import DecisionBoundaryDisplay

from sklearn import svm, datasets

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

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

iris = load\_iris()

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]

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=345, stratify=y)

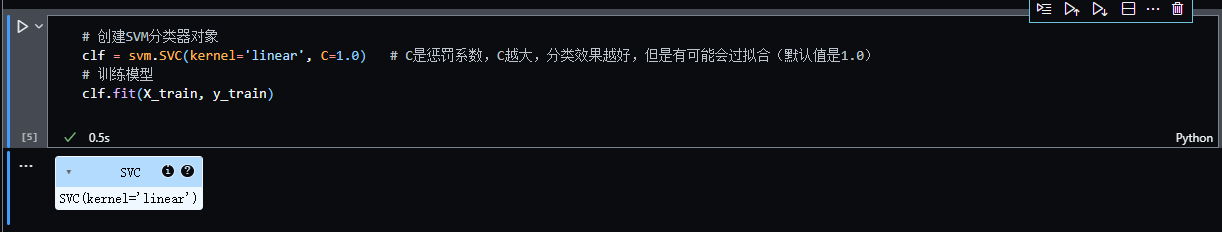
**首先，选取两个特征和两类数据使用 scikit-learn 中的 SVM 进行二分类**

# 创建SVM分类器对象

clf = svm.SVC(kernel='linear', C=1.0)   # C是惩罚系数，C越大，分类效果越好，但是有可能会过拟合（默认值是1.0）

# 训练模型

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

**运行结果：**

**接下来，输出：决策边界的参数和截距、支持向量等**

# 输出决策边界的参数和截距

print("决策边界的参数:", clf.coef\_)

print("截距:", clf.intercept\_)

# 输出支持向量

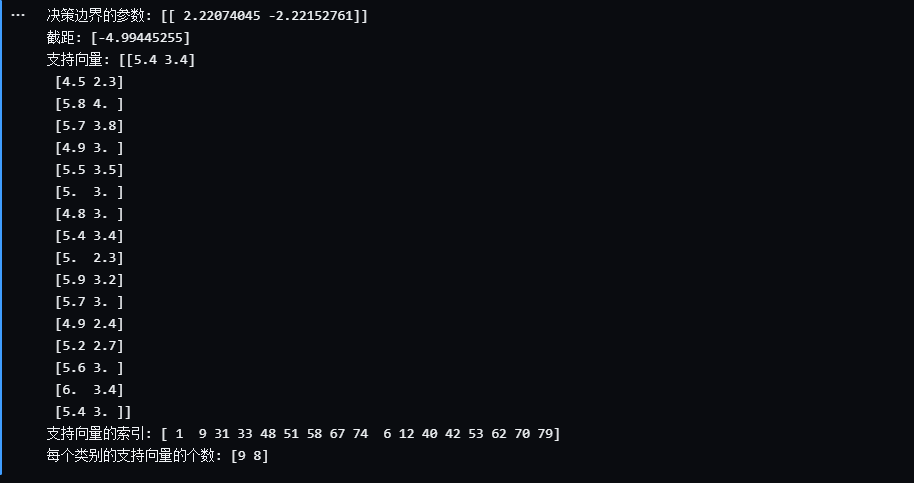
print("支持向量:", clf.support\_vectors\_)

# 输出支持向量的索引

print("支持向量的索引:", clf.support\_)

# 输出每个类别的支持向量的个数

print("每个类别的支持向量的个数:", clf.n\_support\_)

**运行结果：**

**之后进行可视化：画出散点图可视化数据样本（之前选择的两个特征），并画出决策边界和 2 个最大间隔边界，标出支持向量**

# 二分 + 染色

custom\_cmap = mpl.colors.ListedColormap(['#a6f2a5', '#90CAF9'])

disp = DecisionBoundaryDisplay.from\_estimator(

    clf, X\_test, grid\_resolution=1000, eps=0.5,

    response\_method="predict", plot\_method='pcolormesh',

    xlabel=iris.feature\_names[0], ylabel=iris.feature\_names[1],

    alpha=0.6, cmap=custom\_cmap

)

disp.ax\_.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, edgecolor="k", cmap=plt.cm.Paired)

disp.ax\_.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, edgecolor="k", cmap=plt.cm.Paired)

plt.title("全集采用SVM分类")

plt.show()

# 绘制决策面、最大间隔和支持向量

plt.figure(figsize=(7, 5))

plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, cmap=plt.cm.Paired, edgecolors='k', s=30)

ax = plt.gca()

xlim = ax.get\_xlim()

ylim = ax.get\_ylim()

xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 1000), np.linspace(ylim[0], ylim[1], 1000))

Z = clf.decision\_function(np.c\_[xx.ravel(), yy.ravel()])

Z = Z.reshape(xx.shape)

plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5, linestyles=['--', '-', '--'])

plt.scatter(clf.support\_vectors\_[:, 0], clf.support\_vectors\_[:, 1], s=200, linewidth=2,

            facecolors='none', edgecolors='k')

feature\_names = iris.feature\_names

plt.xlabel(feature\_names[0])

plt.ylabel(feature\_names[1])

plt.title('决策面、最大间隔和支持向量')

plt.show()

**运行结果：**

**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 选取的两个特征能否线性可分？若线性可分，可选择 scikit-learn 中何种 SVM 进行建模？若线性不可分，可选择 scikit-learn 中何种 SVM 进行建模？**

从图中可以看出，这两个特征是线性可分的，对于线性可分的情况，可以选择使用scikit-learn中的线性SVM（支持向量机）模型进行建模，即使用svm.SVC(kernel='linear')。

对于线性不可分的情况，可以选择使用非线性SVM模型进行建模。scikit-learn中提供了多种非线性SVM模型，其中一种常用的模型是使用高斯核函数的SVM，即使用svm.SVC(kernel='rbf')。

这种模型可以处理线性不可分的情况，并通过引入核函数来将数据映射到高维空间中进行分类。也可以使用多项式、高斯、S型等核函数，利用核方法将数据维度提升后再提取特征；或者利用软间隔方法，实现非线性分类。

**讨论二 SVM 中的惩罚系数 C 对模型有何影响？**

# 设置不同的C值,生成从0.001到10的等比数列，100个数

C\_values = np.linspace(0.011, 1, 50)

margin\_widths = []

support\_vectors\_counts = []

accuracies = []

for C in C\_values:

    # 创建SVM分类器对象

    clf = svm.SVC(kernel='linear', C=C)

    # 训练模型

    clf.fit(X\_train, y\_train)

    # 获取支持向量的数量

    support\_vectors = clf.support\_vectors\_

    support\_vectors\_count = support\_vectors.shape[0]

    support\_vectors\_counts.append(support\_vectors\_count)

    # 获取间隔宽度

    margin = 2 / np.linalg.norm(clf.coef\_)

    margin\_widths.append(margin)

    # 在测试集上评估模型性能

    accuracy = clf.score(X\_test, y\_test)

    accuracies.append(accuracy) # accuracy = clf.score(X\_test, y\_test), 计算模型在测试集上的准确性

# 绘制间隔宽度和支持向量数量的变化

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

plt.subplot(1, 2, 1)

plt.plot(C\_values, margin\_widths, marker='o')

plt.xlabel('惩罚系数C')

plt.ylabel('间隔宽度')          # 间隔宽度实际上是2/||w||，||w||是模型的权重向量的范数，||w||越小，间隔越大，模型越好

plt.title('随着C值改变，间隔宽度变化')

plt.subplot(1, 2, 2)

plt.plot(C\_values, support\_vectors\_counts, marker='o')

plt.xlabel('惩罚系数C')

plt.ylabel('支持向量数量')

plt.title('随着C值改变，支持向量数量变化')

plt.tight\_layout()

plt.show()

# 绘制模型性能的变化

plt.plot(C\_values, accuracies, marker='o')

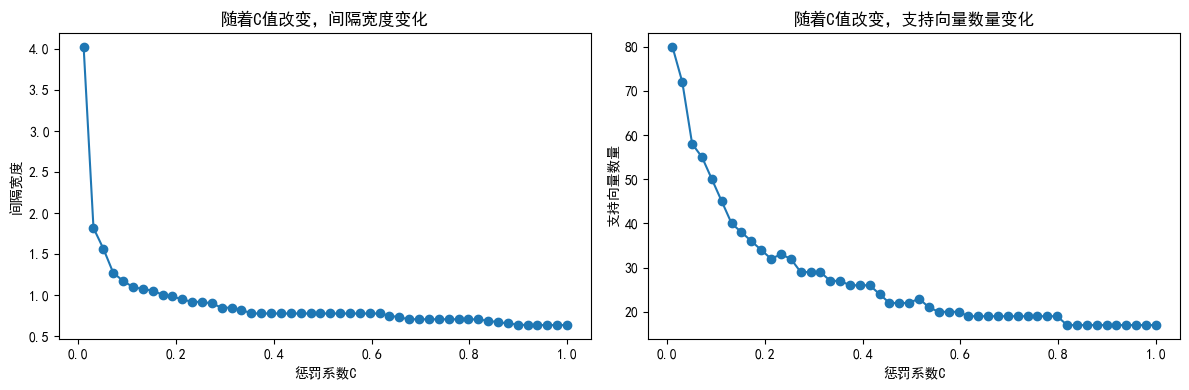
plt.xlabel('惩罚系数C')

plt.ylabel('准确度')

plt.title('改变C值对模型的性能（在测试集上的准确性）的影响')

plt.show()

**通过上述代码，设置不同的C值，分别创建SVM分类器，记录模型性能随C值的变化，得到如下的结果。**

**图表

描述已自动生成**

从中可以分析出以下几点：

随着C值的增加，间隔宽度逐渐减小。这是因为较大的C值会导致模型更加关注于准确分类每个样本，迫使决策边界更紧密地围绕训练样本，从而减小间隔宽度。

较小的C值会导致模型更加关注于最大化间隔，容忍一些分类错误，因此间隔宽度会增加。

支持向量数量与惩罚系数C之间的关系：

随着C值的增加，支持向量的数量逐渐减少。这是因为较大的C值会强制模型更严格地分类样本，减少对误分类样本的容忍度，因此支持向量的数量会减少。

较小的C值会容忍更多的分类错误，鼓励间隔更宽，因此支持向量的数量会增加。

模型性能（准确性）与惩罚系数C之间的关系：

随着C值的增加，模型的性能（在测试集上的准确性）可能会有所改善。这是因为较大的C值会强制模型更严格地分类样本，减少分类错误的可能性。

然而，当C值过大时，模型可能会过拟合训练数据，导致在测试数据上的性能下降。这是由于过大的C值使模型过于依赖于训练数据的细节和噪声，无法良好地泛化到新的数据。

综上所述，惩罚系数C在SVM模型中起到了平衡分类间隔、支持向量数量和模型性能之间的作用。选择合适的C值是调整SVM模型的重要参数，需要根据具体问题和数据集的特征来进行选择。

**2、题目二（选做） ： 采用不同的 SVM 核函数对多种类型数据集进行二分类。**

from sklearn.datasets import load\_iris

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

import matplotlib.pyplot as plt

import matplotlib as mpl

import numpy as np

import sklearn

from sklearn.svm import SVC

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.inspection import DecisionBoundaryDisplay

from sklearn.model\_selection import validation\_curve

from sklearn import datasets

from sklearn.pipeline import Pipeline

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

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

**使用 scikit-learn 中提供的样本生成器 make\_blobs、make\_classification、make\_moons、make\_circles 生成一系列线性或非线性可分的二类别数据**

X1,y1 = datasets.make\_blobs(n\_samples = 500,centers = 2,cluster\_std=0.4, random\_state = 0)  #聚类数据集

X2,y2 = datasets.make\_classification(n\_samples=500,n\_features=2,n\_informative=2,n\_redundant=0,random\_state=20)  #分类数据集

X3,y3 = datasets.make\_circles(n\_samples = 500, noise = 0.2, factor = 0.2, random\_state=0)   #环形数据集

X4,y4 = datasets.make\_moons(n\_samples = 500, noise = 0.2, random\_state=0)                   #月牙形数据集

X\_train1, X\_test1, y\_train1, y\_test1 = train\_test\_split(X1, y1, test\_size=0.2, random\_state=1)

X\_train2, X\_test2, y\_train2, y\_test2 = train\_test\_split(X2, y2, test\_size=0.2, random\_state=1)

X\_train3, X\_test3, y\_train3, y\_test3 = train\_test\_split(X3, y3, test\_size=0.2, random\_state=1)

X\_train4, X\_test4, y\_train4, y\_test4 = train\_test\_split(X4, y4, test\_size=0.2, random\_state=1)

plt.figure(figsize=(16,4))

plt.subplot(141)

for i in range(len(y1)):

    if y1[i]:

        plt.scatter(X1[i,0],X1[i,1],c='r')

    else:

        plt.scatter(X1[i,0],X1[i,1],c='b')

plt.title('blobs')

plt.subplot(142)

for i in range(len(y2)):

    if y2[i]:

        plt.scatter(X2[i,0],X2[i,1],c='r')

    else:

        plt.scatter(X2[i,0],X2[i,1],c='b')

plt.title('classification')

plt.subplot(143)

for i in range(len(y3)):

    if y3[i]:

        plt.scatter(X3[i,0],X3[i,1],c='r')

    else:

        plt.scatter(X3[i,0],X3[i,1],c='b')

plt.title('circles')

plt.subplot(144)

for i in range(len(y4)):

    if y4[i]:

        plt.scatter(X4[i,0],X4[i,1],c='r')

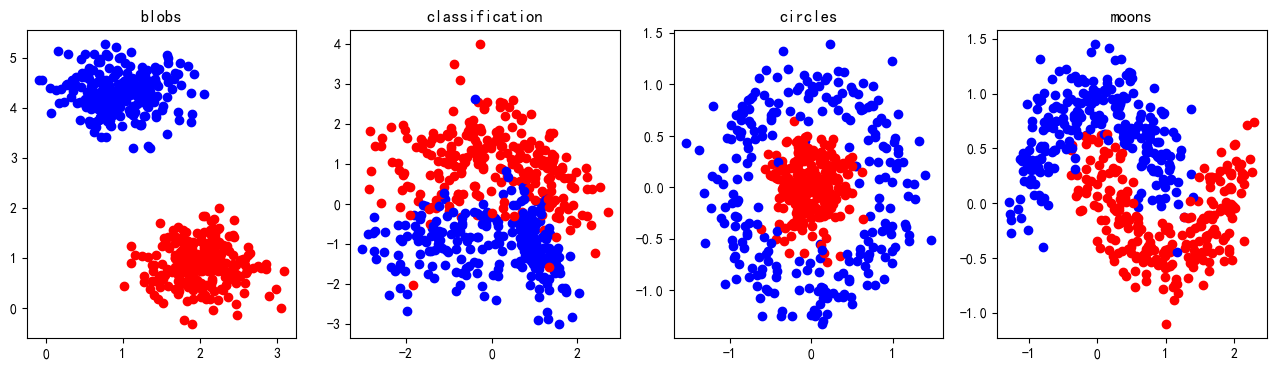
    else:

        plt.scatter(X4[i,0],X4[i,1],c='b')

plt.title('moons')

plt.show()

**得到样本如下：**

****

**建模：分别将 SVM 中四种核函数（线性核、多项式核、高斯核、S 形核）用于上述四种数据集**

# 定义四个数据集的训练集和测试集

datasets = [

    (X\_train1, y\_train1, X\_test1, y\_test1),

    (X\_train2, y\_train2, X\_test2, y\_test2),

    (X\_train3, y\_train3, X\_test3, y\_test3),

    (X\_train4, y\_train4, X\_test4, y\_test4)

]

# 定义核函数和对应的参数范围

param\_grids = {

    'linear': {'C': [0.1, 1, 10]},

    'poly': {'degree': [1,2, 3, 4,5], 'C': [0.1, 1, 3,5,7,10], 'coef0': [0, 1,2,3,4,5,6,7,8]},

    'rbf': {'gamma': [0.1, 1, 10], 'C': [0.1, 1, 3,5,7,10]},

    'sigmoid': {'gamma': [0.1, 1, 10], 'C': [0.1, 1, 3,5,7,10]}

}

# 遍历数据集和核函数参数范围

for i, dataset in enumerate(datasets):

    X\_train, y\_train, X\_test, y\_test = dataset

    # 初始化最佳参数和对应的性能评分

    best\_params = {}

    best\_score = {}

    # 遍历核函数和参数范围

    for kernel, param\_grid in param\_grids.items():

        # 使用GridSearchCV进行参数搜索

        grid\_search = GridSearchCV(SVC(kernel=kernel), param\_grid, cv=5)    # 5次交叉验证

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

        # 保存最佳参数和对应的性能评分

        best\_params[kernel] = grid\_search.best\_params\_

        best\_score[kernel] = grid\_search.best\_score\_

    # 输出最佳参数和对应的性能评分

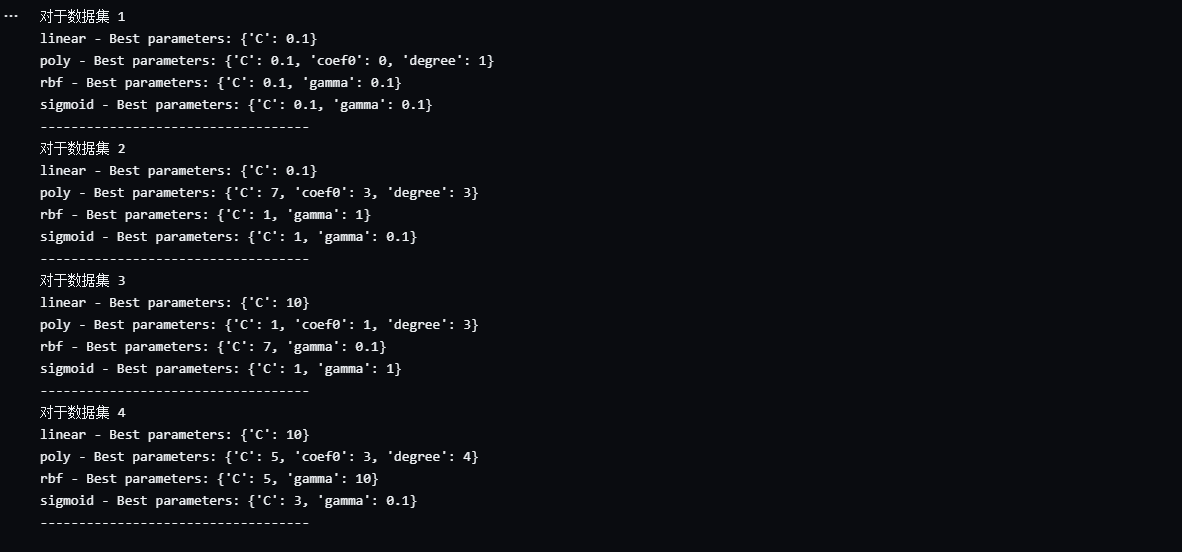
    print("对于数据集", i+1)

    for kernel in param\_grids.keys():

        print(f"{kernel} - Best parameters:", best\_params[kernel])

    print("-----------------------------------")

**运行结果：**



def plot\_decision\_boundary(clf, X, y, ax):

    # 自定义的颜色映射。三个颜色对应于三个类别的数据点

    custom\_cmap = mpl.colors.ListedColormap(['#FF0000', '#FFF60D', '#90CAF9'])

    ax.scatter(X[:, 0], X[:, 1], c=y, s=20, cmap=custom\_cmap)

    xlim = ax.get\_xlim()

    ylim = ax.get\_ylim()

    xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),

                         np.linspace(ylim[0], ylim[1], 100))

    Z = clf.decision\_function(np.c\_[xx.flatten(), yy.flatten()])

    # Z = clf.predict\_proba(np.c\_[xx.flatten(), yy.flatten()])[:, 1]

    Z = Z.reshape(xx.shape)

    ax.contourf(xx, yy, np.where(Z > 0, 1, -1), cmap=plt.cm.Spectral, alpha=0.1)

    # 添加绘制支持向量

    # 首先获取支持向量（support\_vectors = clf.named\_steps['svm\_clf'].support\_vectors\_），然后将它们从标准化的空间转换回原始的空间，

    # 最后在图中绘制这些支持向量，支持向量被绘制为黑色的空心圆。

    support\_vectors = clf.named\_steps['svm\_clf'].support\_vectors\_

    # Then, inverse transform these support vectors

    support\_vectors = clf.named\_steps['scaler'].inverse\_transform(support\_vectors)

    # Finally, plot these support vectors

    ax.scatter(support\_vectors[:, 0], support\_vectors[:, 1],

               s=100, facecolors='none', edgecolors='k', marker='o')

    ax.set\_title("train\_acc: {}/{}\nval\_acc:     {}/{}"

                 .format(np.sum(clf.predict(X) == y), len(y), np.sum(clf.predict(X\_test) == y\_test), len(y\_test)))

    # ax.axis('off')

**数据集一：**

plt.figure(figsize=[13, 3])

# 线性核函数

linear\_kernel\_svm\_clf = Pipeline([

    ("scaler", StandardScaler()), # 使用standardScaler进行数据归一化

    ("svm\_clf", SVC(kernel="linear", C=0.1))

])

linear\_kernel\_svm\_clf.fit(X\_train1, y\_train1)

ax1 = plt.subplot(141)

plot\_decision\_boundary(linear\_kernel\_svm\_clf, X1, y1, ax1)

plt.title("Linear Kernel train\_acc: {}/{}\nval\_acc:{}/{}".format(np.sum(linear\_kernel\_svm\_clf.predict(X1) == y1), len(y1),

                                                                       np.sum(linear\_kernel\_svm\_clf.predict(X\_test1) == y\_test1), len(y\_test1)))

# 多项式核函数

poly\_kernel\_svm\_clf = Pipeline([

    ("scaler", StandardScaler()),

    ("svm\_clf", SVC(kernel="poly", C=0.1, coef0=0, degree=1))

])

poly\_kernel\_svm\_clf.fit(X\_train1, y\_train1)

ax2 = plt.subplot(142)

plot\_decision\_boundary(poly\_kernel\_svm\_clf, X1, y1, ax2)

plt.title("Polynomial Kernel train\_acc: {}/{}\nval\_acc:{}/{}".format(np.sum(poly\_kernel\_svm\_clf.predict(X1) == y1), len(y1),

                                                                       np.sum(poly\_kernel\_svm\_clf.predict(X\_test1) == y\_test1), len(y\_test1)))

# 高斯核函数

rbf\_kernel\_svm\_clf = Pipeline([

    ("scaler", StandardScaler()),

    ("svm\_clf", SVC(kernel="rbf", C=0.1, gamma=0.1))

])

rbf\_kernel\_svm\_clf.fit(X\_train1, y\_train1)

ax3 = plt.subplot(143)

plot\_decision\_boundary(rbf\_kernel\_svm\_clf, X1, y1, ax3)

plt.title("RBF Kernel train\_acc: {}/{}\nval\_acc:{}/{}".format(np.sum(rbf\_kernel\_svm\_clf.predict(X1) == y1), len(y1),

                                                                       np.sum(rbf\_kernel\_svm\_clf.predict(X\_test1) == y\_test1), len(y\_test1)))

# S形核函数

sigmoid\_kernel\_svm\_clf = Pipeline([

    ("scaler", StandardScaler()),

    ("svm\_clf", SVC(kernel="sigmoid", C=0.1, gamma=0.1))

])

sigmoid\_kernel\_svm\_clf.fit(X\_train1, y\_train1)

ax4 = plt.subplot(144)

plot\_decision\_boundary(sigmoid\_kernel\_svm\_clf, X1, y1, ax4)

plt.title("Sigmoid Kernel train\_acc: {}/{}\nval\_acc:{}/{}".format(np.sum(sigmoid\_kernel\_svm\_clf.predict(X1) == y1), len(y1),

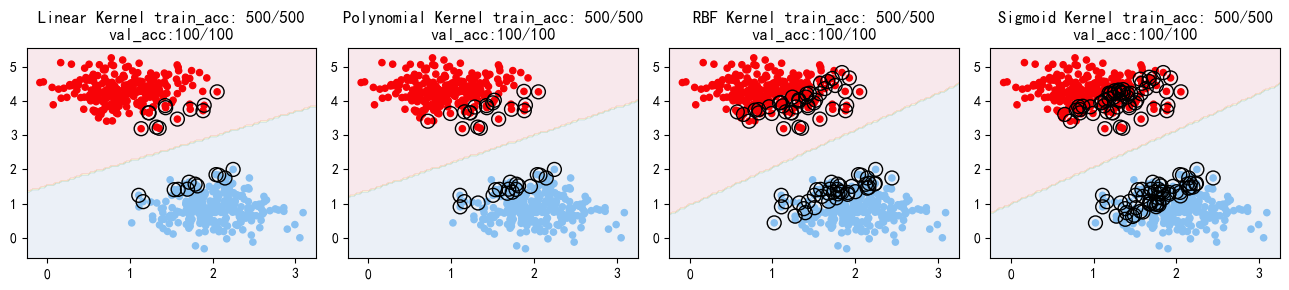
                                                                       np.sum(sigmoid\_kernel\_svm\_clf.predict(X\_test1) == y\_test1), len(y\_test1)))

plt.tight\_layout()

plt.show()

**数据集二，三，四同上，此处省略代码。**

**可视化的结果：**

**图表, 散点图

描述已自动生成图表, 散点图

描述已自动生成图表, 散点图

描述已自动生成**

**讨论三：对于参数的寻优，我采取定义核函数和对应的参数范围，之后遍历数据集和核函数参数范围之后使用GridSearchCV进行参数搜索并进行多次交叉验证的方法。**

**对应代码片段如下：**

# 定义核函数和对应的参数范围

param\_grids = {

    'linear': {'C': [0.1, 1, 10]},

    'poly': {'degree': [1,2, 3, 4,5], 'C': [0.1, 1, 3,5,7,10], 'coef0': [0, 1,2,3,4,5,6,7,8]},

    'rbf': {'gamma': [0.1, 1, 10], 'C': [0.1, 1, 3,5,7,10]},

    'sigmoid': {'gamma': [0.1, 1, 10], 'C': [0.1, 1, 3,5,7,10]}

}

# 遍历数据集和核函数参数范围

for i, dataset in enumerate(datasets):

    X\_train, y\_train, X\_test, y\_test = dataset

    # 初始化最佳参数和对应的性能评分

    best\_params = {}

    best\_score = {}

    # 遍历核函数和参数范围

    for kernel, param\_grid in param\_grids.items():

        # 使用GridSearchCV进行参数搜索

        grid\_search = GridSearchCV(SVC(kernel=kernel), param\_grid, cv=5)    # 5次交叉验证

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

        # 保存最佳参数和对应的性能评分

        best\_params[kernel] = grid\_search.best\_params\_

        best\_score[kernel] = grid\_search.best\_score\_

**讨论四：由结果进行分析，并结合查询到的资料，我们可以对这四种核函数进行简要的概括和总结：**

**线性核函数：**

**决策边界：线性核函数在处理线性可分的数据集时表现良好，决策边界为直线或超平面。**

**适用场合：线性核函数适用于线性可分的数据集，当数据集可以通过一条直线或超平面进行有效分割时，线性核函数是一个简单而有效的选择。**

**优势：线性核函数计算效率高，参数选择较为简单。**

**多项式核函数：**

**决策边界：多项式核函数在处理非线性可分的数据集时可以引入多项式特征，从而将数据映射到更高维度的空间，在该空间中构建决策边界。**

**适用场合：多项式核函数适用于具有一定非线性特征的数据集，当数据集不能通过线性边界进行有效分割时，可以尝试使用多项式核函数。**

**优势：多项式核函数能够处理一定程度的非线性关系，可以适应更复杂的数据集。**

**高斯核函数：**

**决策边界：高斯核函数（也称为径向基函数核）在处理非线性可分的数据集时通过引入高斯函数，构建以支持向量为中心的圆形或椭圆形决策边界。**

**适用场合：高斯核函数适用于具有复杂非线性关系的数据集，当数据集无法通过线性或多项式边界进行有效分割时，可以尝试使用高斯核函数。**

**优势：高斯核函数能够处理更为复杂的非线性关系，适用于各种类型的数据集。**

**S形核函数：**

**决策边界：S形核函数在处理非线性可分的数据集时，通过引入Sigmoid函数，构建以支持向量为中心的S形决策边界。**

**适用场合：S形核函数适用于某些特定的非线性关系，当数据集具有特定的非线性模式，并且其他核函数效果不佳时，可以尝试使用S形核函数。**

**优势：S形核函数对于特定类型的非线性关系具有一定的适应性。**

**将不同模型在不同类别样本上的效果可视化**

**3、题目三： 使用 scikit-learn 中的 SVM 分类器对乳腺癌威斯康星州数据集进行分类。**

from sklearn.datasets import load\_iris

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

from sklearn.svm import LinearSVC

import matplotlib.pyplot as plt

import matplotlib as mpl

import numpy as np

import sklearn

from sklearn.svm import SVC

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import StandardScaler

from sklearn.inspection import DecisionBoundaryDisplay

from sklearn.model\_selection import validation\_curve

from sklearn import datasets

import time

from sklearn.datasets import load\_breast\_cancer

data = load\_breast\_cancer()

frame = data.frame # 传入参数 as\_frame=True 时才会返回

X = data.data

y = data.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=1)

feature\_names = data.feature\_names

target\_names = data.target\_names

print("数据量及维度：",X.shape)

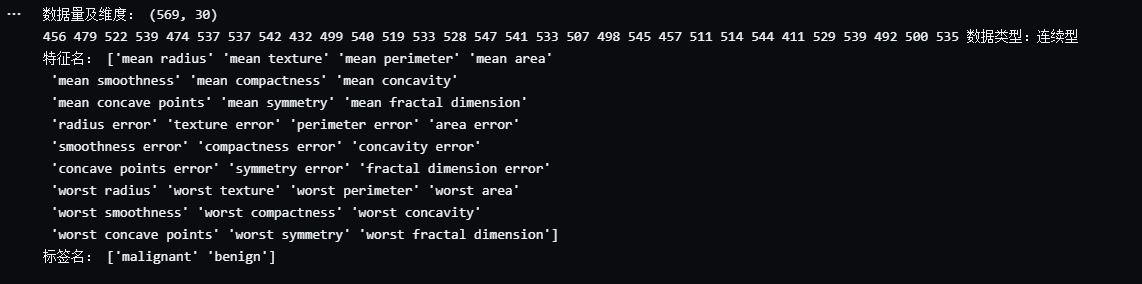
for i in range(30):

    print(len(set(X[:,i])), end=' ')

print("数据类型：连续型")

print("特征名：",feature\_names)

print("标签名：",target\_names)



**线性核:**

start = time.time()

# 归一化处理：

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

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

clf = LinearSVC(random\_state=555, tol=1e-3, C=1, dual=False)

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

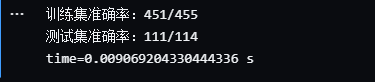
pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

print("训练集准确率：{}/{}".format(np.sum(pre\_train==y\_train),len(y\_train)))

print("测试集准确率：{}/{}".format(np.sum(pre\_test==y\_test),len(y\_test)))

print("time={} s".format(time.time()-start))



**多项式核：**

start = time.time()

clf = make\_pipeline(StandardScaler(), \

    SVC(kernel='poly',  # kernel{‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’} or callable, default=’rbf’

        degree=1,      # 仅在poly有效

        gamma="scale", # gamma{‘scale’, ‘auto’} or float, default=’scale’

        coef0=0,       # 只有对’poly’ 和,’sigmod’核函数有用

        ) )

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

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

print("训练集准确率：{}/{}".format(np.sum(pre\_train==y\_train),len(y\_train)))

print("测试集准确率：{}/{}".format(np.sum(pre\_test==y\_test),len(y\_test)))

print("time={}".format(time.time()-start))

文本

描述已自动生成

**高斯核：**

start = time.time()

# 归一化处理：

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

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

# 高斯核 SVC

clf = SVC(kernel='rbf', random\_state=555, tol=1e-3, C=1)

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

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

print("训练集准确率：{}/{}".format(np.sum(pre\_train==y\_train), len(y\_train)))

print("测试集准确率：{}/{}".format(np.sum(pre\_test==y\_test), len(y\_test)))

print("time={} s".format(time.time()-start))

文本

描述已自动生成

**S形核：**

start = time.time()

# 归一化处理：

scaler = StandardScaler()

X\_train = scaler.fit\_transform(X\_train)

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

# S形核 SVC

clf = SVC(kernel='sigmoid', random\_state=555, tol=1e-3, C=1)

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

pre\_train = clf.predict(X\_train)

pre\_test = clf.predict(X\_test)

print("训练集准确率：{}/{}".format(np.sum(pre\_train==y\_train), len(y\_train)))

print("测试集准确率：{}/{}".format(np.sum(pre\_test==y\_test), len(y\_test)))

print("time={} s".format(time.time()-start))

文本

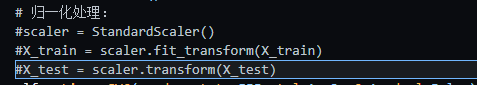
描述已自动生成

**讨论五：**

**从训练集的准确度来看：线性核>高斯核>多项式核>S形核**

**从测试集的准确度来看：线性核=高斯核>多项式核>S形核**

**讨论六：**

**为了探究归一化对核函数的影响，我们将这段代码注释**

**再次运行程序，得到四种核函数在无归一化情况下运行的效果**

文本

描述已自动生成

文本

中度可信度描述已自动生成

文本

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

**与经过归一化的运行结果相比，未归一化的情况下四种核函数的准确率并无明显差异，而在运行时间上，未归一化的情况下，使用高斯核和多项式核的运行时间远高于归一化的时候，而线性核和S形核的运行时间相比进行归一化的情况则有所减少。**

**通过查阅资料，我们可以得知，在使用支持向量机（SVM）时，数据归一化处理通常是必要的。SVM 对输入特征的尺度非常敏感，特征值范围较大的特征可能会主导决策边界，从而影响模型的性能。数据归一化可以确保特征在相同的尺度上进行比较，避免某些特征由于较大的尺度而主导模型的决策，提高模型的收敛速度和稳定性。**

**4、题目四： 编写 SMO 算法实现线性 SVM 分类器，对 iris 数据集进行二分类。**

import random

import matplotlib.pyplot as plt

import numpy as np

from sklearn.datasets import load\_iris

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

import matplotlib as mpl

from sklearn.metrics import roc\_curve

class Supper\_Vector\_Machine:

    def \_\_init\_\_(self, dataset, target, C, tol, iter\_max):

        """

           参数:

           dataset : 训练样本数据

           target : 训练样本标签

           C ：惩罚系数

           toler ： 终止条件（迭代阈值）

           Iter\_max ： 终止条件（最大迭代次数）

           属性：

           alpha ： SVM对偶问题的拉格朗日乘子（乘子个数=样本数据量）

           w(coef\_) ：线性模型（决策面）的系数

           b ：线性模型（决策面）的截距

        """

        self.X = dataset

        self.y = target

        self.C = C

        self.tol = tol

        self.iter\_max = iter\_max

        self.alpha = np.zeros\_like(target).astype(float)

        self.w = np.random.randn(dataset.shape[1])

        self.b = np.random.randn(1)

        print(self.w)

        print(self.alpha)

        print(self.w\*self.w.T)

    def Fx(self, i):

        """

           f(x):决策函数

           参数:i (一个待优化的拉格朗日乘子a[i]的下标)

        """

        # 将第i行样本带入决策函数中计算。

        # 返回：计算结果f(xi)

        # fx = 0

        # for k in range(len(self.y)):

        #     fx += self.alpha[k] \* self.y[k] \* np.sum(self.X[k]\* self.X[i])

        # return fx + self.b

        return np.sum((self.alpha\*self.y).reshape(-1,1)\*self.X\*self.X[i]) + self.b

    def Kernel(self, i, j):

        """

           参数:i，j (两个待优化的拉格朗日乘子a[i]和a[j]的下标)

        """

        # Kernel(xi, xj): 核函数计算（线性核就是x\_i ^ T \* x\_j）

        # 返回：计算结果

        return (self.X[i] \* self.X[i]).sum() + (self.X[j] \* self.X[j]).sum() - 2\*(self.X[i] \* self.X[j]).sum()

    def random\_j(self, i):

        """

           参数:i(一个待优化的拉格朗日乘子a[i]的下标)

        """

        # 随机选择另一个待优化的拉格朗日乘子a[j]的下标j(j与i不相同）

        # 返回：j

        j = random.randint(0,len(self.y)-1)

        while j==i:

            j = random.randint(0,len(self.y)-1)

        return j

    def get\_L\_H(self, i, j):

        """

           参数:i，j (两个待优化的拉格朗日乘子a[i]和a[j]的下标)

        """

        # 计算上下界

        # 返回：上界和下界

        if self.y[i]!=self.y[j]:

            L = max(0., self.alpha[j] - self.alpha[i])

            H = min(self.C, self.C + self.alpha[j] - self.alpha[i])

        else:

            L = max(0, self.alpha[j] + self.alpha[i] - self.C)

            H = min(self.C, self.alpha[j] + self.alpha[i])

        return L, H

    def filter(self, L, H, alpha\_j):

        """

           参数:

            i，j：两个待优化的拉格朗日乘子a[i]和a[j]的下标

            L, H：上下界

        """

        # 按边界对a[j]值进行修剪，使其在（L, H）范围内。

        # 返回：修剪后的a[j]

        if alpha\_j>H:

            return H

        elif L<=alpha\_j<=H:

            return alpha\_j

        elif alpha\_j<L:

            return L

        else:

            raise("function filter error!!")

    def decision\_function(self,dataset):

        """

            参数：dataset：样本

            输出：1\*n的数组wx+b

        """

        return (self.w \* dataset).sum(axis=1) + self.b

    def predict(self,dataset, target=None):

        """

           参数:

           dataset : 样本数据

           target : 样本标签

        """

        pre = self.decision\_function(dataset)

        pre = np.where(pre>0, 1, -1)

        return pre

    def SMO(self):

        # 外循环：迭代次数

        # change\_num用于记录拉格朗日乘子更新的次数，iter用于记录遍历拉格朗日乘子的次数

        iter = 0

        iterr = 0

        while iter < self.iter\_max:

            print('\n',"\*"\*10,"iterr",iterr,"iter",iter,"\*"\*10)

            print("w={},b={}".format(self.w,self.b))

            change\_num = 0

            # 内循环：遍历拉格朗日乘子，作为a[i]

            for i in range(len(self.y)):

                # 1、计算Fx(i)，Ei

                Fx\_i = self.Fx(i)

                Ei = Fx\_i - self.y[i]

                # 2、随机选择另一个要优化的拉格朗日乘子a[j]：random\_j方法

                j = self.random\_j(i)

                # 3、计算Fx(j)，Ej

                Fx\_j = self.Fx(j)

                Ej = Fx\_j - self.y[j]

                # 4、计算上下界：get\_L\_H方法

                L, H = self.get\_L\_H(i,j)

                if L == H:

                    # 判断如果L == H，则a[i]和a[j]都在边界，不用再进行优化，寻找下一对

                    # print("L============H={}".format(L))

                    continue

                # 5、计算eta：kernel方法

                eta = self.Kernel(i,j)

                if eta <= 0:

                    # 判断如果eta <= 0，则不再进行优化，寻找下一对

                    # print("etaaaaaaaaaaaaa")

                    continue

                # 6、更新a[j]

                alpha\_j\_old = self.alpha[j].copy()

                j\_upgrade = self.y[j] \* (Ei-Ej) / eta

                self.alpha[j] += j\_upgrade

                # print("i={},j={}".format(i,j),end=' ')

                # print("Fx\_={:.2f},{:.2f}".format(Fx\_i[0],Fx\_j[0]),end=' ')

                # print("L={:2f}, H={:.2f}".format(L,H),end=' ')

                # print("eta={:.3f}".format(eta),end=' ')

                # print("j\_upgrade={:.2f}".format(j\_upgrade[0]))

                # 7、修剪a[j]

                self.alpha[j] = self.filter(L,H,self.alpha[j])

                if abs(self.alpha[j]-alpha\_j\_old)<tol:

                    # 判断如果a[j]\_new-a[j]\_old < toler，则不更新a[j]，寻找下一对

                    self.alpha[j] = alpha\_j\_old

                    # print("abs(j\_upgrade)<tol")

                    continue

                # 8、更新a[i]

                i\_upgrade = (self.y[i]\*self.y[j])\*(alpha\_j\_old-self.alpha[j])

                self.alpha[i] += i\_upgrade

                # 9、更新bi和bj

                b\_i = self.b - Ei - self.y[i]\*i\_upgrade\*np.sum(self.X[i]\*self.X[i]) - self.y[j]\*(self.alpha[j]-alpha\_j\_old)\*np.sum(self.X[j]\*self.X[i])

                b\_j = self.b - Ej - self.y[i]\*i\_upgrade\*np.sum(self.X[i]\*self.X[j]) - self.y[j]\*(self.alpha[j]-alpha\_j\_old)\*np.sum(self.X[j]\*self.X[j])

                # 10、更新b

                if 0<self.alpha[i]<self.C:

                    self.b = b\_i

                elif 0<self.alpha[j]<self.C:

                    self.b = b\_j

                else:

                    self.b = (b\_i+b\_j)/2

                change\_num += 1

            # change\_num为0，则表示遍历过一遍所有的拉格朗日乘子，都没有进行更新

            if change\_num == 0:

                iter += 1

            else:

                iter = 0 #一旦拉格朗日乘子有一次更新，就重新遍历，直到完成iter\_max次遍历，都没有进行更新，就认为找到最优系数

            iterr += 1

            self.w = np.dot((self.alpha\*self.y), self.X)

            print("acc=", self.score(self.y,self.predict(self.X)))

        # 11、迭代完成，计算决策面w

        return self.w, self.b

    def score(self, y\_true, y\_pred):

        """

           参数:

           y\_true : 实际标签

           y\_pred : 预测标签

        """

        # 评价指标任选

        return 1-float(np.count\_nonzero(y\_pred-y\_true))/len(y\_true)

def plot\_roc\_curve(fpr, tpr, label=None):

    plt.plot([0, 1], [0, 1], 'k--')

    plt.plot(fpr, tpr, linewidth=2, label=label)

    plt.axis([0, 1, 0, 1])

    plt.xlabel('False Positive Rate')

    plt.ylabel('True Positive Rate')

    plt.show()

C = 10

tol = 1e-3

iter\_max = 100

if \_\_name\_\_ == '\_\_main\_\_':

    # 1、导入数据，划分数据集

    # 注意：选择两个特征进行二分类，二分类标签为1和-1

    iris = load\_iris()

    X = iris.data[:100,(0,1)]

    X = (X-X.mean(axis=0))/X.std(axis=0)

    y = iris.target[:100]

    y = 2\*y-1

    X1\_train, X1\_test, y1\_train, y1\_test = train\_test\_split(X[:50,:2], y[:50], test\_size=0.1, random\_state=1)

    X2\_train, X2\_test, y2\_train, y2\_test = train\_test\_split(X[50:100,:2], y[50:100], test\_size=0.1, random\_state=1)

    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)

    # 2、创建SVM对象，训练模型

    # svm = Supper\_Vector\_Machine(X\_test, y\_test, C, tol, iter\_max)

    svm = Supper\_Vector\_Machine(X\_train, y\_train, C, tol, iter\_max)

    w,b = svm.SMO()

    # print(svm.predict(X\_test,y\_test))

    # 输出决策面系数w和截距b

    # print("w={}, b={}".format(w,b))

    # 3、画出特征散点图和决策面，标出支持向量（通过可视化效果，判断程序编写是否正确）

    plt.figure(figsize=[5,5])

    custom\_cmap = mpl.colors.ListedColormap(['#EF9A9A','#FFF59D','#90CAF9'])

    # plt.scatter(X[:, 0], X[:, 1], c=y, s=25, cmap=custom\_cmap)                   # all

    plt.scatter(X\_train[:, 0], X\_train[:, 1], c=y\_train, s=25, cmap=custom\_cmap, alpha=0.3) # train

    plt.scatter(X\_test[:, 0], X\_test[:, 1], c=y\_test, s=25, cmap=custom\_cmap, alpha=1)    # test

    plt.axis('off')

    ax = plt.gca()

    xlim = ax.get\_xlim()

    ylim = ax.get\_ylim()

    xx, yy = np.meshgrid(np.linspace(xlim[0], xlim[1], 100),

                        np.linspace(ylim[0], ylim[1], 100))

    Z = svm.decision\_function(np.c\_[xx.flatten(), yy.flatten()])

    Z = Z.reshape(xx.shape)

    plt.contourf(xx,yy,np.where(Z>0,1,-1), cmap=plt.cm.Spectral, alpha=0.1)

    plt.contour(xx, yy, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,

                linestyles=['--', '-', '--'])

    decision\_function = svm.decision\_function(X\_train)

    support\_vector\_indices = np.where(y\_train \* decision\_function <= 1+tol\*10)[0]

    support\_vectors = X\_train[support\_vector\_indices]

    plt.scatter(support\_vectors[:, 0], support\_vectors[:, 1], s=100,

                linewidth=1, facecolors='none', edgecolors='k')

    # 4、使用测试集带入模型预测

    pre\_train = svm.predict(X\_train)

    pre\_test = svm.predict(X\_test)

    plt.title("train\_acc: {}/{}\ntest\_acc: {}/{}"\

        .format(np.sum(pre\_train==y\_train),len(y\_train),np.sum(pre\_test==y\_test),len(y\_test)))

    plt.show()

    # 5、模型评价

    score = svm.score(pre\_test,y\_test)

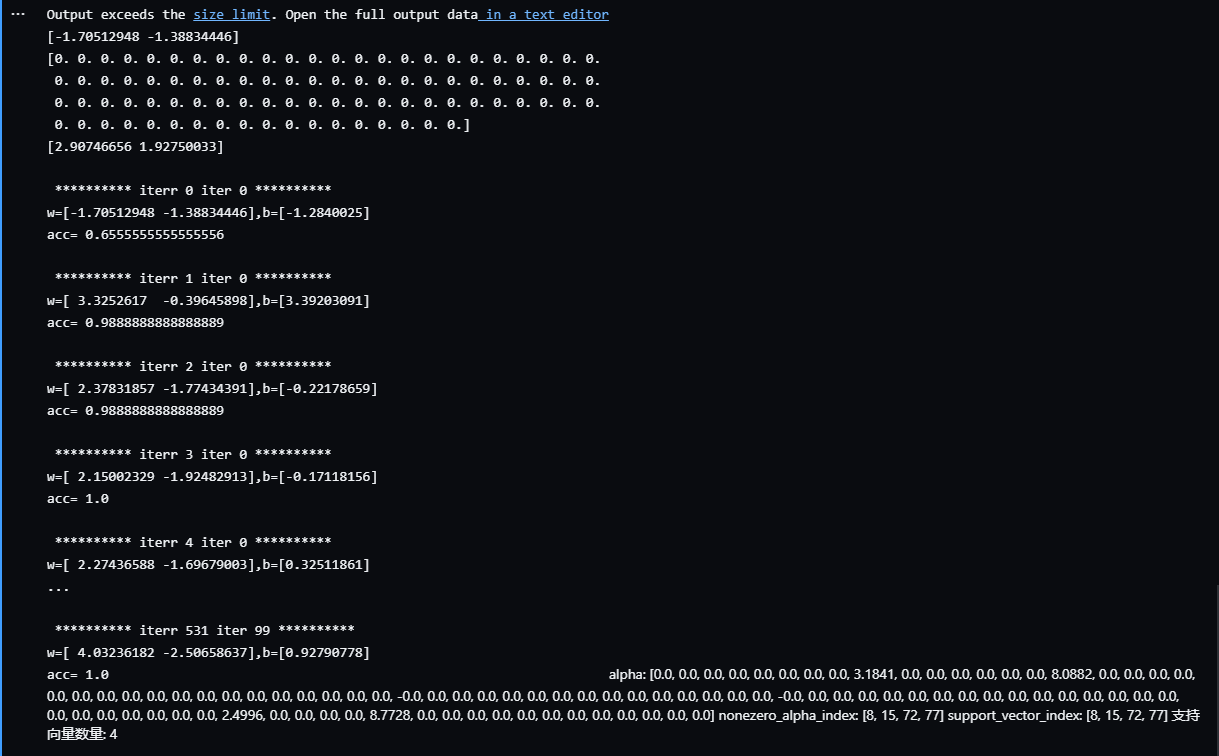
    print("alpha:",np.around(svm.alpha,4).tolist())

    print("nonezero\_alpha\_index:",np.where(np.abs(svm.alpha)>tol\*10)[0].tolist())

    print("support\_vector\_index:",support\_vector\_indices.tolist())

    print("支持向量数量:",support\_vectors.shape[0])

**运行结果：**



**图表, 散点图

描述已自动生成**

**关键代码片段：**

**决策函数：**

def decision\_function(self, dataset):

return (self.w \* dataset).sum(axis=1) + self.b

**计算决策值：**

def Fx(self, i):

return np.sum((self.alpha \* self.y).reshape(-1, 1) \* self.X \* self.X[i]) + self.b

**核函数：**

def Kernel(self, i, j):

return (self.X[i] \* self.X[i]).sum() + (self.X[j] \* self.X[j]).sum() - 2 \* (self.X[i] \* self.X[j]).sum()

**随机选择拉格朗日乘子：**

def random\_j(self, i):

j = random.randint(0, len(self.y) - 1)

while j == i:

j = random.randint(0, len(self.y) - 1)

return j

**计算拉格朗日乘子的上下界：**

def get\_L\_H(self, i, j):

if self.y[i] != self.y[j]:

L = max(0., self.alpha[j] - self.alpha[i])

H = min(self.C, self.C + self.alpha[j] - self.alpha[i])

else:

L = max(0, self.alpha[j] + self.alpha[i] - self.C)

H = min(self.C, self.alpha[j] + self.alpha[i])

return L, H

**更新拉格朗日乘子：**

alpha\_j\_old = self.alpha[j].copy()

j\_upgrade = self.y[j] \* (Ei - Ej) / eta

self.alpha[j] += j\_upgrade

i\_upgrade = (self.y[i] \* self.y[j]) \* (alpha\_j\_old - self.alpha[j])

self.alpha[i] += i\_upgrade

**确保拉格朗日乘子在合法范围内：**

def filter(self, L, H, alpha\_j):

if alpha\_j > H:

return H

elif L <= alpha\_j <= H:

return alpha\_j

elif alpha\_j < L:

return L

else:

raise("function filter error!!")

**更新偏置项：**

b\_i = self.b - Ei - self.y[i] \* i\_upgrade \* np.sum(self.X[i] \* self.X[i]) - self.y[j] \* (self.alpha[j] - alpha\_j\_old) \* np.sum(self.X[j] \* self.X[i])

b\_j = self.b - Ej - self.y[i] \* i\_upgrade \* np.sum(self.X[i] \* self.X[j]) - self.y[j] \* (self.alpha[j] - alpha\_j\_old) \* np.sum(self.X[j] \* self.X[j])

if 0 < self.alpha[i] < self.C:

self.b = b\_i

elif 0 < self.alpha[j] < self.C:

self.b = b\_j

else:

self.b = (b\_i + b\_j) / 2

更新权重：

self.w = np.dot((self.alpha \* self.y), self.X)

迭代停止条件：

if change\_num == 0:

iter += 1

else:

iter = 0

**讨论七：**

**参数𝐶：**

**参数𝐶是一个正则化参数，用于控制模型对误分类的容忍度。C值较小意味着模型更容忍误分类，会导致较大的间隔和较少的支持向量，但可能增加训练数据的误差。相反，较大的C值会使模型对误分类的容忍度降低，产生较小的间隔和更多的支持向量，从而可能减少训练误差但增加过拟合的风险。**

**拉格朗日乘子𝛼：**

**拉格朗日乘子𝛼对应于每个训练数据点，并在优化过程中求解对偶问题时被确定。对于非支持向量，𝛼等于零；对于支持向量，𝛼是正值。具体来说，对于那些恰好位于间隔边界上的支持向量，𝛼的值介于0和𝐶之间。对于那些误分类的数据点，即那些位于决策边界错误一侧的点，其𝛼等于𝐶。**

**支持向量：**

**支持向量是构成SVM决策函数的关键数据点。这些点位于最优决策边界的间隔线上或在错误的一侧。支持向量的特性（即它们的位置和数量）直接影响最优决策面的形状和位置。**

**最优决策面和间隔区域：**

**最优决策面是由支持向量完全确定的。这个决策面是分类问题中的一个超平面，其目的是最大化两个类别之间的间隔。间隔区域被定义为最近的支持向量到决策面的距离，这个区域在两类数据之间提供了一定的空间，使得分类器具有更好的泛化能力。**

**间隔的宽度与𝛼值密切相关，实际上间隔可以通过 1/||w|| 计算得出，其中**

**w是由𝛼和训练数据的线性组合得到的。**

**理论知识参考：https://www.bilibili.com/video/BV13r4y1z7AG/?share\_source=copy\_web&vd\_source=6c2c09437f8e8ba051dd3e5dcaaefe19**