**智能信息系统综合实践**

**实验报告**

|  |  |
| --- | --- |
| **题 目：** | **第四次作业 SVM** |
| **年 级：** | **2021** |
| **专 业：** | **软件工程** |
| **学 号：** | **2021117405** |
| **姓 名：** | **孙潇桐** |

目录

[1. 题目 3](#_Toc161827811)

[2. 解题步骤 3](#_Toc161827812)

[2.1. 题目一：使用SVM对样本进行分类 3](#_Toc161827813)

[2.1.1. 对数据的处理 3](#_Toc161827814)

[2.1.2. 对SVM的训练 3](#_Toc161827815)

[2.1.3. 对最佳参数模型的测试 6](#_Toc161827816)

[2.1.4. 模型的可视化 7](#_Toc161827817)

[2.2. 与逻辑回归分类器的比较 10](#_Toc161827818)

[2.2.1. 逻辑分类器的实现和训练 10](#_Toc161827819)

[2.2.2. 对逻辑分类器的测试 11](#_Toc161827820)

[2.2.3. 逻辑分类器的可视化 12](#_Toc161827821)

[2.2.4. SVM和逻辑分类器的比较 14](#_Toc161827822)

[3. 总结 14](#_Toc161827823)

[4. 附件 15](#_Toc161827824)

[4.1. 完整代码 15](#_Toc161827825)

[4.1.1. SVM分类器的实现和测试：svm.py 15](#_Toc161827826)

[4.1.2. 逻辑分类器的实现和测试：logic.py 20](#_Toc161827827)

[4.1.3. 工具类：utils.py 24](#_Toc161827828)

[4.2. 热力图的原始数据 26](#_Toc161827829)

# 题目

1. 用Iris Dataset（选前两类），交叉验证，测试SVM分类器性能，改变SVM超参数，对比不同超参数对结果的影响。
2. 对比SVM和第一章中的逻辑回归分类器，自定衡量指标。

# 解题步骤

## 题目一：使用SVM对样本进行分类

### 对数据的处理

通过观察，Iris数据集里面有三类样本。但是为了方便与后面的逻辑回归分析器相比较，我从原始的样本集中删除了第三类样本，于是数据集中的样本种类就从 [‘Setosa’, ‘Versicolor’, ‘Virginica’] 变为了 [‘Setosa’, ‘Versicolor’]，后面的实验都是基于我修改过的数据集。

接下来需要将得到的数据导入内存，从而完成接下来的计算。因为我在后面选择的**交叉验证**方式是**留出法**，所以根据以往的经验，我将测试集设置为样本总数的30%。设置随机种子为42以保证每次得到的数据集都是一样的，方便后续模型之间的比较，下面是我的实现代码，其中的load\_data函数是我写的一个写在utils.py文件中的工具函数，用于从csv文件中读取数据，完整的代码我会在文档的最后贴出。

# 加载数据集

X, Y, title = load\_data("data\iris.csv")

tags = np.unique(Y)

# 将数据集划分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.3, random\_state=42

)

经过这样的处理，就得到了训练集的样本集和标签以及测试集的样本和标签。

### 对SVM的训练

这次的实验，由于样本集可能不是线性可分的，于是我选择了使用RBF（径向基函数核）作为SVM的核函数。核函数隐含地决定了数据映射到新的特征空间后的分布，所以可以解决一些线性不可分的问题。在SVM中，广泛使用的RBF核函数为高斯核：

同时，为了更好的效果，我也使用了软边界SVM，以允许支持向量机在某些样本上出错而换取更好的泛化能力。为此，优化目标需要引入惩罚系数C。于是，在加入核函数和惩罚系数之后的最优化问题变成了：

很显然，此处的模型有两个十分重要的参数：C和 (gamma)。

* C：从上面的式子可以很容易看出来，惩罚系数C越高，对错误的容忍度越低。相应的，其泛化能力就会相对更差。而C越低，对错误的容忍度越高。但是过低的惩罚系数会带来欠拟合的问题，合理的降低惩罚系数就能在保证模型正确性的基础上提高泛化能力。
* (gamma)：在我们这里使用的高斯核中 ，此处的 负责调整高斯函数的形状。根据我们曾经学过的正态分布， 越大的时候图像会变得又高又瘦，而反之则又矮又胖。在SVM中，我的理解就是给与支持向量不同举例的样本以不同的权重。当 很大的时候，只有非常贴近支持向量的样本才会对模型产生比较大的影响。 很小的时候，距离远的样本也会产生较大的影响。我认为更通俗的理解就是， 负责调节支持向量考虑周围的**样本半径**，也就是RBF中的R。综上所述， 越大，支持向量考虑的样本半径越大，反之半径越小。

为了能够探究这两个参数在样本集上如何影响准确率，我的设计就是对每个参数都给出几个典型值，然后比较使用这些典型值作为参数的SVM模型**交叉验证**的结果。在本次实验中，我使用交叉测试中的留出发，也就是在**测试集**上对模型进行测试，得到模型在测试集上的准确率。当然，模型的参数和具体的样本息息相关，我这里得到的结论也只能代表使用这些参数的SVM在Iris数据集上的表现。下面是我的算法设计：

# 定义不同的超参数

C\_values = [0.001, 0.01, 0.1, 1, 5, 10, 100, 10000]

gamma\_values = [0.001, 0.01, 0.1, 1, 5, 10, 100, 10000]

# 存储结果的字典

results = {}

# 遍历超参数组合

for c in C\_values:

    for gamma in gamma\_values:

        # 初始化SVM分类器

        svm = SVC(kernel=svm\_kernal, C=c, gamma=gamma)

        # 训练模型

        svm.fit(X\_train, y\_train)

        # 计算当前参数的模型在测试集上的准确率

        score = accuracy\_score(y\_test, svm.predict(X\_test))

        # 存储结果

        results[(c, gamma)] = score

results中储存了所有的参数和准确率，最后我们就能在results里面找出最好的参数，然后使用最好的参数训练模型以完成后面的测试。为了更加好的展示结果，我画出了参数和准确率的热力图，下面是我的实现过程和结果：

实现过程：

# 将结果转换为二维数组以方便画图

scores = np.array(list(results.values())).reshape(len(C\_values), len(gamma\_values))

# 使用热力图展示结果

plt.imshow(scores, interpolation="nearest", cmap=plt.cm.tab10)

plt.title(f"SVM with Kernal {svm\_kernal}")

plt.xlabel("gamma")

plt.ylabel("C")

# 展示热力条

plt.colorbar()

# 设置x轴上的数字

plt.xticks(np.arange(len(gamma\_values)), gamma\_values)

# 设置y轴上的数字

plt.yticks(np.arange(len(C\_values)), C\_values)

结果：因为完整的结果列表比较长，我放在[附件](#_热力图的原始数据)中（4.2节）

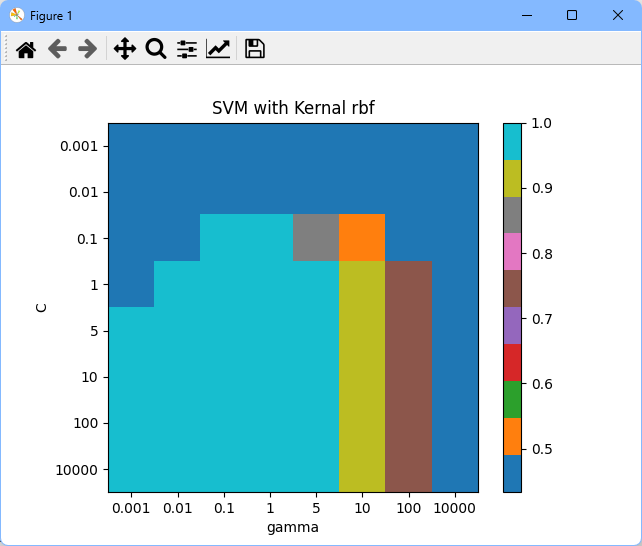


Figure 1 不同参数训练的热力图

从这张图可以很轻易的看出：

1. 对于惩罚系数来说，当惩罚系数过小的时候，模型出现了非常严重的欠拟合问题，准确率一度不足50%，不论gamma如何取值。但是对于这个数据集，当惩罚系数十分大的时候，也能够达到100%的准确率，此时SVM已经成为了硬边界SVM，由此可见这个数据集还是很好划分的，没有出现过拟合问题。
2. 对于gamma值来说，当gamma值越大，也就是支持向量考虑的半径越来越大的时候准确率越低。可能是被其他类别的向量干扰了，导致优化过程出现问题。

### 对最佳参数模型的测试

经过上面的过程，我们已经得到最好的参数，然后使用最好的参数训练模型之后就可以对我们的模型进行进一步测试。我的方案是使用混淆矩阵来计算查全率，查准率和F1值，下面是我的实现方案：

# 混淆矩阵：纵轴为预测值，横轴为实际值

cm = np.zeros((2, 2))

for x, tag in zip(X\_test, y\_test):

    # 计算预测标签

    cur\_tag = best\_svm.predict([x])[0]

    # 将预测标签从字符串转为数字

    cur\_tag = conv(cur\_tag)

    tag = conv(tag)

    cm[cur\_tag][tag] += 1

# 查准率

precision = cm.diagonal() / np.sum(cm, axis=1)

# 查全率

recall = cm.diagonal() / np.sum(cm, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果

display(cm, precision, recall, f1, tags)

其中的display函数是我写的一个用于输出结果的函数，放在utils.py中，完整代码会在4.1节中给出，下面是输出的结果。

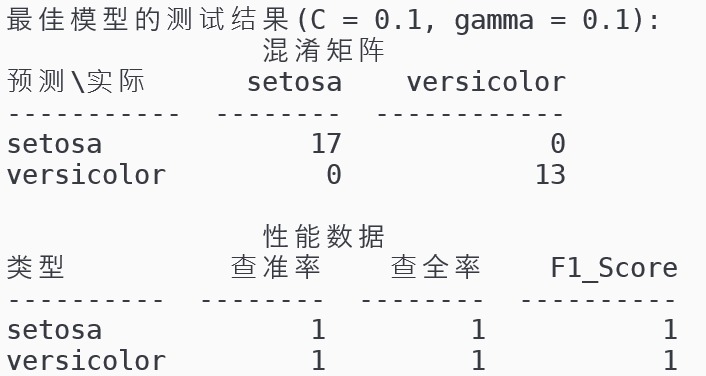


Figure 2 最佳模型的测试结果

### 模型的可视化

由于经过了核函数的数据已经被映射到更高的维度，所以没法直接画出支持向量。而且样本有四个特征值，也不好可视化，所以我可视化的时候使用了前三个特征值。经过我的测试，是否使用第四个特征似乎对前两个类别的分类影响不大：

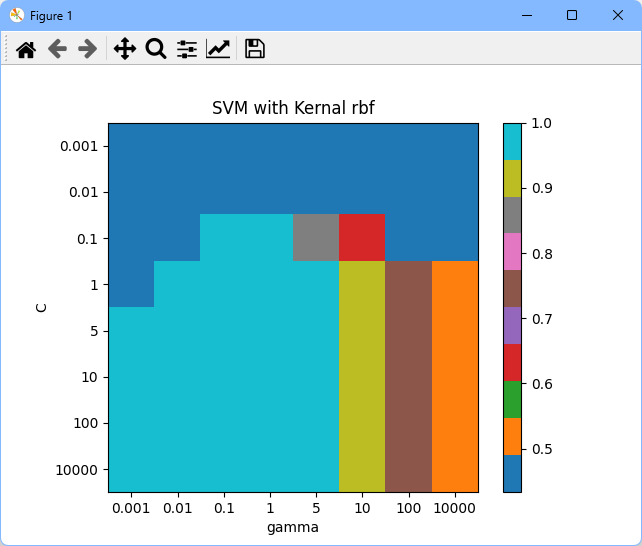
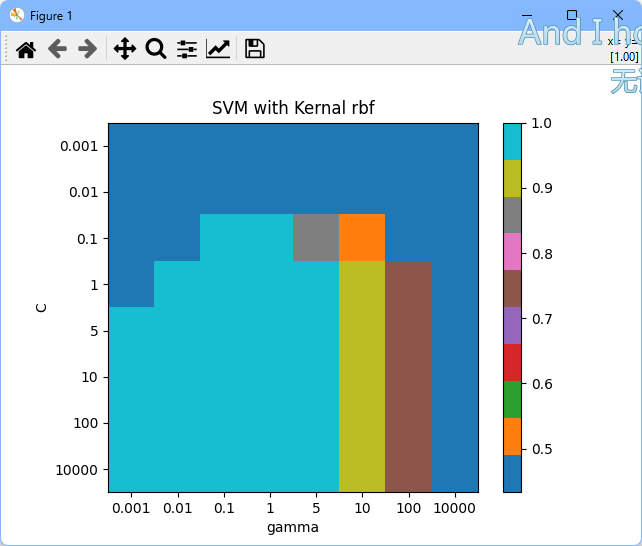
 

Figure 3使用四个特征 Figure 4使用前三个特征

所以，接下来的可视化我都使用前三个特征值进行。我的思路是在样本的数值范围内对一些点进行取样，然后对这些点进行判断，将这些点分为两个类别，然后对这两个类别之间举例很近的且属于不同类别的点加入列表。这样在图上就可以画出超平面在三维的投影和支持向量，下面是我都具体实现：

def visualize\_none\_linear\_svm(svm, X, y):

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection="3d")

    # 分离样本

    sample0 = y == tags[0]

    X\_0, X\_1 = X[sample0], X[~sample0]

    ax.scatter(X\_0[:, 0], X\_0[:, 1], X\_0[:, 2], label=tags[0])

    ax.scatter(X\_1[:, 0], X\_1[:, 1], X\_1[:, 2], label=tags[1])

    # 确定数据范围

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

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

    z\_min, z\_max = X[:, 2].min() - 1, X[:, 2].max() + 1

    # 在数据范围内均匀的生成点

    xx, yy, zz = np.meshgrid(

        np.linspace(x\_min, x\_max, 30),

        np.linspace(y\_min, y\_max, 30),

        np.linspace(z\_min, z\_max, 30),

    )

    points = np.c\_[xx.ravel(), yy.ravel(), zz.ravel()]

    # 对这些点的类别进行预测

    Z = svm.predict(points)

    # 找出两类样本的下标

    C0 = Z == tags[0]

    # 分离两类样本

    dot0 = points[C0]

    dot1 = points[~C1]

    # 用于记录边节点的数组

    edge\_dots0 = []

    edge\_dots1 = []

    edge\_dots = []

    for a in tqdm(dot0):

        tmp = np.sum(a - dot1, axis=1)

        idx = tmp == 0

        # 记录边界点

        if np.sum(tmp == 0) > 0:

            edge\_dots0.append(a)

            for b in dot1[idx]:

                edge\_dots1.append(b)

                edge\_dots.append((a + b) / 2)

    def plot\_surface(ax, dots, idx=0):

        dots = np.array(dots)

        print(f"ploting {dots.shape[0]} points")

        XX = dots[:, 0]

        YY = dots[:, 1]

        ZZ = dots[:, 2]

        ax.plot\_trisurf(XX, YY, ZZ, alpha=0.1, label=f"divider\_{idx}")

    # 分别画出样本边界和各自的支持向量

    plot\_surface(ax, edge\_dots0, 0)

    plot\_surface(ax, edge\_dots1, 1)

    plot\_surface(ax, edge\_dots, 2)

    ax.legend()

    # 设置坐标轴标签

    ax.set\_xlabel(title[0])

    ax.set\_ylabel(title[1])

    ax.set\_zlabel(title[2])

    plt.show()

下面是可视化结果：

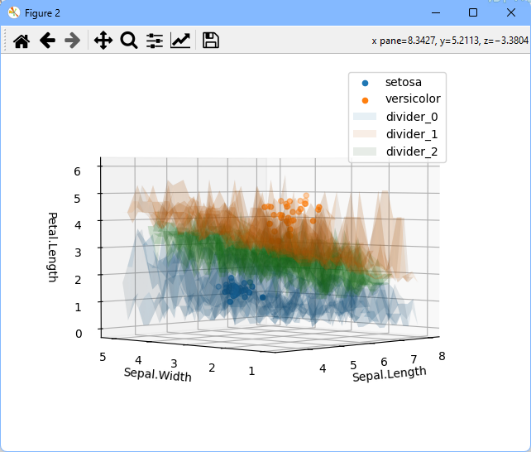
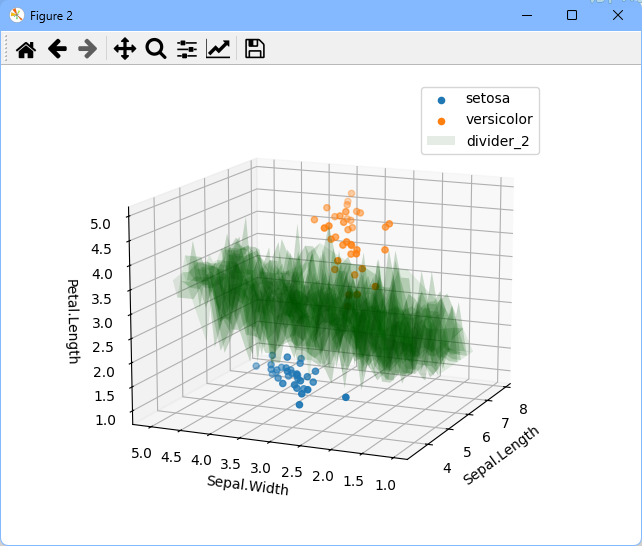
 

Figure 5 可视化支持向量和超平面 Figure 6 可视化超平面

## 与逻辑回归分类器的比较

### 逻辑分类器的实现和训练

由于第二次实验的时候只取了前两个特征值，我对上次的程序进行了一些改进，使其能够支持任意个特征值。但是为了方便后面的可视化，我还是只使用了前三个特征值。逻辑分类器的原理是对特征值进行线性变换，得到结果送入sigmoid激活函数来得到预测的结果：

逻辑分类器特别需要注意的一点就是对数据的标准化，如果不进行标准化的话不论如何调整参数都难以收敛（经验之谈）。下面是我对数据的处理过程：

# 存储每个特征的标准差，平均值和参数

std, mean, a = [], [], [random()]

# 对数据集进行标准化

for i in range(num\_para):

    s, m, X[:, i] = standardize(X[:, i])

    std.append(s)

    mean.append(m)

    a.append(random())

其中的standardize方法是我写的一个标准化函数，保存在utils.py中：

def standardize(list):

    mean = np.mean(list)

    std = np.std(list)

    return std, mean, (list - mean) / std

为了方便在模型之间比较，我在逻辑分类器中也使用了相同的样本划分参数（30%的测试集和42的随机种子），只要使用相同的参数得到的结果就是一样的：

# 将数据集划分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.3, random\_state=42

)

接下来就是训练过程了，直接使用梯度下降算法。我的训练超参数是2000次训练以及0.01的学习率，下面是具体的实现过程：

for \_ in tqdm(range(epoch)):

    sum = [0.0] \* len(a)

    for x, tag in zip(X\_train, y\_train):

        # 计算预测标签

        inner = a[0]

        for i in range(1, len(a)):

            inner += a[i] \* x[i - 1]

        cur\_tag = sigmoid(inner)

        # 计算梯度

        diff = cur\_tag - tag

        sum[0] += diff \* dsigmoid(inner, 1)

        for i in range(1, len(a)):

            sum[i] += diff \* dsigmoid(inner, x[i - 1])

    for i, cur\_sum in enumerate(sum):

        a[i] -= alpha \* (cur\_sum / num)

### 对逻辑分类器的测试

和上面的SVM模型一样我也使用留出法进行交叉验证，并使用混淆矩阵计算模型的查全率，查准率和F1值。下面是我的实现过程：

# 混淆矩阵：纵轴为预测值，横轴为实际值

cm = np.zeros((2, 2))

for x, tag in zip(X\_test, y\_test):

    # 计算预测标签

    inner = a[0]

    for i in range(1, len(a)):

        inner += a[i] \* x[i - 1]

    cur\_tag = sigmoid(inner)

    cur\_tag = round(cur\_tag)

    tag = round(tag)

    cm[cur\_tag][tag] += 1

# 查准率

precision = cm.diagonal() / np.sum(cm, axis=1)

# 查全率

recall = cm.diagonal() / np.sum(cm, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果

display(cm, precision, recall, f1, tags)

得到的结果如下：

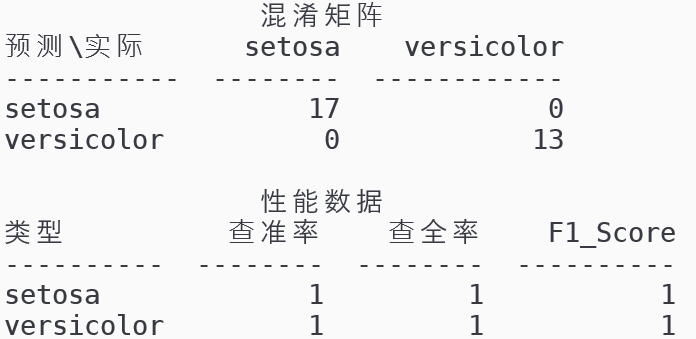


Figure 7 逻辑分类器的测试结果

很显然这个数据集对sigmoid逻辑分类器毫无挑战，测试集上的数据都对了。

### 逻辑分类器的可视化

上文提到过，为了方便可视化，我并没有使用全部四个特征值，只用了前三个。但这并不妨碍这个逻辑分类器拿下测试集全对。同时，为了与SVM更好的比较，我也对逻辑分类器进行了可视化的操作。

由于没有核函数，所以可以很轻易的在三维空间画出分界线。由于sigmoid函数的判断的依据是变量是否大于0，所以只需要画出内部函数等于0的平面就可以，也就是画出平面：

下面是我的代码实现：

def visualize\_model(X, y):

    # 取出模型权重

    w, b = a[1:], a[0]

    # 创建网格点以绘制决策边界

    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.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

    # 计算决策平面

    decision\_plane = -(w[0] \* xx + w[1] \* yy + b) / w[2]

    # 创建3D图形对象

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection="3d")

    # 绘制决策平面

    ax.plot\_surface(xx, yy, decision\_plane, alpha=0.5)

    # 绘制两种类型的散点图

    # 分离样本

    sample0 = y == 0

    X\_0, X\_1 = X[sample0], X[~sample0]

    ax.scatter(X\_0[:, 0], X\_0[:, 1], X\_0[:, 2], label=tags[0])

    ax.scatter(X\_1[:, 0], X\_1[:, 1], X\_1[:, 2], label=tags[1])

    ax.legend()

    # 设置坐标轴标签

    ax.set\_xlabel(title[0])

    ax.set\_ylabel(title[1])

    ax.set\_zlabel(title[2])

    # 显示图形

    plt.show()

结果：

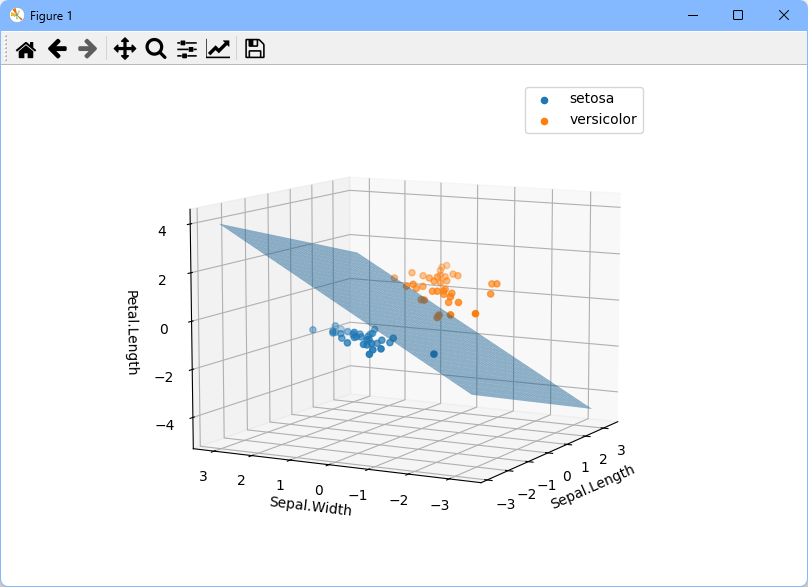


Figure 8 可视化逻辑分类器

很显然，逻辑分类器已经完美的将样本分离开。而且可以看得出来，这个数据集是线性可分的。

### SVM和逻辑分类器的比较

我将从三个方面比较这两种模型：

* 准确率：根据上面[图2](#图2)（SVM的性能）和[图7](#图7)（逻辑分类器的性能）中的数据显示，这两个模型都在相同的训练集和测试集下取得了全对的成绩，所以查准率，查全率和F1的值均为1。所以至少在这个数据集上，两个模型在准确度方面并没有拉开明显的差距。出现这种情况的原因，可以从[图8](#图8)中看出。这个数据集是线性可分的，这样的话SVM中使用的核函数将数据映射到高维的操作就没有优势了，反而会降低SVM模型的训练和预测的效率。
* 可解释性：逻辑分类器因为没有核函数，可解释性非常强。每个权重都对应一个特征值，从权重就能看出某个特征值的重要程度。而核函数SVM解释性就稍微欠缺一些，因为决策边界在更高的维度。而且逻辑分类器最后的输出是一个概率，而SVM直接得到的就是标签。
* 推理速度：由于需要计算核函数的存在，理论上来说SVM的推理速度会慢一些。但是由于这个数据集太小了，我没测出来速度差别。为了进一步探究推理速度的差别，我给两个模型写了benchmark函数，随机生成了同样的10000个数据，代码比较简单这里就不贴了，完整代码在[附件](#_完整代码)（第4节）可以看到。下面是我得到的结果：

Figure 9 SVM推理测试结果 Figure 10 逻辑分类器推理测试结果

很显然逻辑分类器的推理速度更快。

# 总结

这次SVM的实验花费了我非常多的时间和精力，从复习上个学习学过的模型的原理到可视化模型的方法，到思考模型的测试策略。每一个环节都十分重要，我也都遇到了一些困难。

这次我花费最多时间解决的问题就是模型的可视化问题，首先这次的数据有四个特征值，显然四个特征值没办法直接可视。经过测试，我发现最后一个特征值对前两个类别的分类不太重要，于是我选择了使用前三个特征值重新训练模型然后进行可视化。同时由于核函数的存在，我没法直接画出超平面在三维的投影，于是只能另辟蹊径，在样本的数据范围内采样一部分点，然后对这些点中的边节点进行处理后作为超平面的三维投影展示。从上面的图看的出来，由于采样的精度不够高，超平面也不够平滑，但是提高采样精度计算的耗时就非常长，这算是我的一个取舍。

在和逻辑分类器对比的时候也出了问题，由于第二次实验刚刚复习过逻辑分类器，所以在将逻辑分类器从上次实验的两个特征值改为任意特征值的过程中没有遇到大问题。但是当我看到两个模型的测试结果都是完全正确的时候，我明白了这个数据集不能让这两个模型拉开差距，于是我只能将重心转移到比较这两个模型的其他方面比如推理速度，并设计实验验证我的想法。

这次实验让我受益匪浅，使我对numpy库和sklearn库的理解更进一步的同时也锻炼了我解决问题的能力。上面提到的两个库都是在机器学习领域非常重要的工具，只有用好他们才能更好的完成实验，并理解这些模型的底层原理。我将继续保持学习，多学，多写，争取有朝一日能成为机器学习大师。

# 附件

## 完整代码

### SVM分类器的实现和测试：svm.py

from utils import load\_data, display, tabulate, gen\_data, np

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score

import matplotlib.pyplot as plt

from tqdm import tqdm

from time import time

# svm使用的核函数

svm\_kernal = "rbf"  # "sigmoid","rbf"

# 加载数据集

X, Y, title = load\_data("data\iris.csv")

tags = np.unique(Y)

# 将数据集划分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.3, random\_state=42

)

# 定义不同的超参数

C\_values = [0.001, 0.01, 0.1, 1, 5, 10, 100, 10000]

gamma\_values = [0.001, 0.01, 0.1, 1, 5, 10, 100, 10000]

# 存储结果的字典

results = {}

times = []

# 遍历超参数组合

for c in C\_values:

    for gamma in gamma\_values:

        begin = time()

        # 初始化SVM分类器

        svm = SVC(kernel=svm\_kernal, C=c, gamma=gamma)

        # 训练模型

        svm.fit(X\_train, y\_train)

        # 计算当前参数的模型在测试集上的准确率

        score = accuracy\_score(y\_test, svm.predict(X\_test))

        # 存储结果

        results[(c, gamma)] = score

        times.append(time() - begin)

times = np.array(times)

print(f"平均训练用时{times.mean()}s")

# 打印结果

print("SVM分类器不同超参数的性能:")

headers = ["C", "gamma", "准确率"]

data = []

for params, score in results.items():

    data.append((params[0], params[1], score))

print(tabulate(data, headers=headers, floatfmt=".2f"))

# 取出最优参数

best\_params = max(results, key=results.get)

best\_svm = SVC(kernel=svm\_kernal, C=best\_params[0], gamma=best\_params[1])

# 训练模型

best\_svm.fit(X\_train, y\_train)

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

print("\n最佳模型在测试集上的准确率: {:.2f}\n".format(test\_score))

# 将结果转换为二维数组以方便画图

scores = np.array(list(results.values())).reshape(len(C\_values), len(gamma\_values))

# 使用热力图展示结果

plt.imshow(scores, interpolation="nearest", cmap=plt.cm.tab10)

plt.title(f"SVM with Kernal {svm\_kernal}")

plt.xlabel("gamma")

plt.ylabel("C")

plt.colorbar()

# 设置x轴上的数字

plt.xticks(np.arange(len(gamma\_values)), gamma\_values)

# 设置y轴上的数字

plt.yticks(np.arange(len(C\_values)), C\_values)

def conv(tag):

    return 0 if tag == tags[0] else 1

print(f"最佳模型的测试结果(C = {best\_params[0]}, gamma = {best\_params[1]}):")

time\_begin = time()

# 对测试集进行预测

predicted\_y = best\_svm.predict(X\_test)

# 混淆矩阵：纵轴为预测值，横轴为实际值

cm = np.zeros((2, 2))

for cur\_tag, tag in zip(predicted\_y, y\_test):

    # 将预测标签从字符串转为数字

    cur\_tag = conv(cur\_tag)

    tag = conv(tag)

    cm[cur\_tag][tag] += 1

print(f"测试用时{time()-time\_begin}s")

# 查准率

precision = cm.diagonal() / np.sum(cm, axis=1)

# 查全率

recall = cm.diagonal() / np.sum(cm, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果

display(cm, precision, recall, f1, tags)

# 取出svm的参数

def visualize\_linear\_model(svm, X, y):

    w, b = svm.coef\_[0], svm.intercept\_[0]

    # 创建网格点以绘制决策边界

    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.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

    # 计算决策平面

    decision\_plane = -(w[0] \* xx + w[1] \* yy + b) / w[2]

    # 创建3D图形对象

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection="3d")

    # 绘制决策平面

    ax.plot\_surface(xx, yy, decision\_plane, alpha=0.5)

    # 绘制两种类型的散点图

    # 分离样本

    sample0 = y == tags[0]

    X\_0, X\_1 = X[sample0], X[~sample0]

    ax.scatter(X\_0[:, 0], X\_0[:, 1], X\_0[:, 2], label=tags[0])

    ax.scatter(X\_1[:, 0], X\_1[:, 1], X\_1[:, 2], label=tags[1])

    ax.legend()

    # 设置坐标轴标签

    ax.set\_xlabel(title[0])

    ax.set\_ylabel(title[1])

    ax.set\_zlabel(title[2])

    # 显示图形

    plt.show()

def visualize\_svm(svm, X, y):

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection="3d")

    # 分离样本

    sample0 = y == tags[0]

    X\_0, X\_1 = X[sample0], X[~sample0]

    ax.scatter(X\_0[:, 0], X\_0[:, 1], X\_0[:, 2], label=tags[0])

    ax.scatter(X\_1[:, 0], X\_1[:, 1], X\_1[:, 2], label=tags[1])

    # 确定数据范围

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

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

    z\_min, z\_max = X[:, 2].min() - 1, X[:, 2].max() + 1

    # 在数据范围内均匀的生成点

    xx, yy, zz = np.meshgrid(

        np.linspace(x\_min, x\_max, 30),

        np.linspace(y\_min, y\_max, 30),

        np.linspace(z\_min, z\_max, 30),

    )

    points = np.c\_[xx.ravel(), yy.ravel(), zz.ravel()]

    # 对这些点的类别进行预测

    Z = svm.predict(points)

    # 找出两类样本的下标

    C0 = Z == tags[0]

    C1 = ~C0

    # 分离两类样本

    dot0 = points[C0]

    dot1 = points[C1]

    # print(dot0.shape, dot1.shape)

    # 用于记录边节点的数组

    edge\_dots0 = []

    edge\_dots1 = []

    edge\_dots = []

    for a in tqdm(dot0):

        tmp = np.sum(a - dot1, axis=1)

        idx = tmp == 0

        # 记录边界点

        if np.sum(tmp == 0) > 0:

            edge\_dots0.append(a)

            for b in dot1[idx]:

                edge\_dots1.append(b)

                edge\_dots.append((a + b) / 2)

    def plot\_surface(ax, dots, idx=0):

        dots = np.array(dots)

        print(f"ploting {dots.shape[0]} points")

        XX = dots[:, 0]

        YY = dots[:, 1]

        ZZ = dots[:, 2]

        ax.plot\_trisurf(XX, YY, ZZ, alpha=0.1, label=f"divider\_{idx}")

    # 分别画出样本边界和各自的支持向量

    plot\_surface(ax, edge\_dots0, 0)

    plot\_surface(ax, edge\_dots1, 1)

    plot\_surface(ax, edge\_dots, 2)

    ax.legend()

    # 设置坐标轴标签

    ax.set\_xlabel(title[0])

    ax.set\_ylabel(title[1])

    ax.set\_zlabel(title[2])

    plt.show()

def benchmark(num):

    x = gen\_data(X, num)

    time\_begin = time()

    res = best\_svm.predict(x)

    print(f"预测{num}个样本耗时{(time()-time\_begin)\*1000:.5}ms")

    print(res)

visualize\_svm(best\_svm, X\_train, y\_train)

benchmark(10000)

### 逻辑分类器的实现和测试：logic.py

from utils import (

    load\_data,

    standardize,

    sigmoid,

    dsigmoid,

    display,

    gen\_data,

    np,

    standardize\_list,

)

from random import random

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split  # , cross\_val\_score

from tqdm import tqdm

from time import time

X, Y, title = load\_data("data\iris.csv")

print(title)

tags = np.unique(Y)

# 学习率

alpha = 0.01

# 训练次数

epoch = 2000

# 样本个数和参数个数

num\_all, num\_para = X.shape

# print(X.shape)

# y = sigmoid(a0 + a1 \* x1 + a2 \* x2 + x3 \* a3 + x4 \* a4...)

# 存储每个特征的标准差，平均值和参数

std, mean, a = [], [], [random()]

# 对数据集进行标准化

for i in range(num\_para):

    s, m, X[:, i] = standardize(X[:, i])

    std.append(s)

    mean.append(m)

    a.append(random())

# 将标签转化为 0 和 1 进行二分类

Y\_conv = []

for tag in Y:

    if tag == tags[0]:

        Y\_conv.append(0)

    else:

        Y\_conv.append(1)

Y = np.array(Y\_conv, dtype=float)

# 将数据集划分为训练集和测试集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, Y, test\_size=0.3, random\_state=42

)

num = X\_train.shape[0]

print("training...")

time\_begin = time()

for \_ in tqdm(range(epoch)):

    # for \_ in range(epoch):

    sum = [0.0] \* len(a)

    for x, tag in zip(X\_train, y\_train):

        # 计算预测标签

        inner = np.sum(a[1:] \* x) + a[0]

        cur\_tag = sigmoid(inner)

        # 计算梯度

        diff = cur\_tag - tag

        sum[0] += diff \* dsigmoid(inner, 1)

        for i in range(1, len(a)):

            sum[i] += diff \* dsigmoid(inner, x[i - 1])

    for i, cur\_sum in enumerate(sum):

        a[i] -= alpha \* (cur\_sum / num)

print(f"训练用时{time()-time\_begin}s")

# 混淆矩阵：纵轴为预测值，横轴为实际值

cm = np.zeros((2, 2))

for x, tag in zip(X\_test, y\_test):

    # 计算预测标签

    inner = np.sum(a[1:] \* x) + a[0]

    cur\_tag = sigmoid(inner)

    cur\_tag = round(cur\_tag)

    tag = round(tag)

    cm[cur\_tag][tag] += 1

# 查准率

precision = cm.diagonal() / np.sum(cm, axis=1)

# 查全率

recall = cm.diagonal() / np.sum(cm, axis=0)

# F1

f1 = 2 \* np.multiply(precision, recall) / (precision + recall)

# 展示结果

display(cm, precision, recall, f1, tags)

def visualize\_model(X, y):

    # 取出模型权重

    w, b = a[1:], a[0]

    # 创建网格点以绘制决策边界

    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.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

    # 计算决策平面

    decision\_plane = -(w[0] \* xx + w[1] \* yy + b) / w[2]

    # 创建3D图形对象

    fig = plt.figure()

    ax = fig.add\_subplot(111, projection="3d")

    # 绘制决策平面

    ax.plot\_surface(xx, yy, decision\_plane, alpha=0.5)

    # 绘制两种类型的散点图

    # 分离样本

    sample0 = y == 0

    X\_0, X\_1 = X[sample0], X[~sample0]

    ax.scatter(X\_0[:, 0], X\_0[:, 1], X\_0[:, 2], label=tags[0])

    ax.scatter(X\_1[:, 0], X\_1[:, 1], X\_1[:, 2], label=tags[1])

    ax.legend()

    # 设置坐标轴标签

    ax.set\_xlabel(title[0])

    ax.set\_ylabel(title[1])

    ax.set\_zlabel(title[2])

    # 显示图形

    plt.show()

def benchmark(num):

    np.random.seed(42)

    tag = np.array(tags)

    x = gen\_data(X, num)

    x = standardize\_list(x, mean, std)

    time\_begin = time()

    # y=np.random.choice([0,1])

    res = tag[np.round(sigmoid(np.sum(a[1:] \* x, axis=1) + a[0])).astype(int)]

    print(f"预测{num}个样本耗时{(time()-time\_begin)\*1000:.5}ms")

    print(res)

visualize\_model(X\_train, y\_train)

benchmark(10000)

### 工具类：utils.py

import csv

import numpy as np

from tabulate import tabulate

def load\_data(path):

    # 使用的特征数

    ATTR\_NUM = 3

    # 标签的列

    TAG\_COL = 4

    x, y = [], []

    with open(path, "r") as file:

        reader = csv.reader(file)

        title = next(reader)

        for row in reader:

            x.append(np.array(row[:ATTR\_NUM], dtype=float))

            y.append(row[TAG\_COL])

    x = np.array(x)

    y = np.array(y)

    return x, y, title[:ATTR\_NUM]

def standardize(list):

    mean = np.mean(list)

    std = np.std(list)

    return std, mean, (list - mean) / std

def standardize\_list(list, mean, std):

    for i in range(list.shape[1]):

        list[:, i] = standardize\_with\_paras(list[:, i], mean[i], std[i])

    return list

def standardize\_with\_paras(list, mean, std):

    return (list - mean) / std

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

def dsigmoid(z, x):

    return x \* sigmoid(z) \* (1 - x \* sigmoid(z))

# 展示结果（混淆矩阵，查准率，查全率和F1\_Score)

def display(cm, pc, rc, f1, tags):

    print("\t\t混淆矩阵")

    headers = ["预测\实际"]

    data = []

    for x in tags:

        headers.append(x)

    for idx, tag in enumerate(tags):

        row = [tag] + cm[idx].tolist()

        data.append(row)

    print(tabulate(data, headers=headers))

    print("\n\t\t性能数据")

    headers = ["类型", "查准率", "查全率", "F1\_Score"]

    data = []

    for idx, tag in enumerate(tags):

        row = [tag, pc[idx], rc[idx], f1[idx]]

        data.append(row)

    print(tabulate(data, headers=headers))

# 随机生成测试数据

def gen\_data(X, num):

    np.random.seed(42)

    # X, \_, \_ = load\_data("data/iris.csv")

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

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

    z\_min, z\_max = X[:, 2].min() - 1, X[:, 2].max() + 1

    X\_new = np.zeros((num, 3))

    X\_new[:, 0] = np.random.uniform(x\_min, x\_max + 0.1, (num,))

    X\_new[:, 1] = np.random.uniform(y\_min, y\_max + 0.1, (num,))

    X\_new[:, 2] = np.random.uniform(z\_min, z\_max + 0.1, (num,))

    return X\_new

## 热力图的原始数据

|  |  |  |
| --- | --- | --- |
| C | gamma | 准确率 |
| 0 | 0 | 0.43 |
| 0 | 0.01 | 0.43 |
| 0 | 0.1 | 0.43 |
| 0 | 1 | 0.43 |
| 0 | 5 | 0.43 |
| 0 | 10 | 0.43 |
| 0 | 100 | 0.43 |
| 0 | 10000 | 0.43 |
| 0.01 | 0 | 0.43 |
| 0.01 | 0.01 | 0.43 |
| 0.01 | 0.1 | 0.43 |
| 0.01 | 1 | 0.43 |
| 0.01 | 5 | 0.43 |
| 0.01 | 10 | 0.43 |
| 0.01 | 100 | 0.43 |
| 0.01 | 10000 | 0.43 |
| 0.1 | 0 | 0.43 |
| 0.1 | 0.01 | 0.43 |
| 0.1 | 0.1 | 1 |
| 0.1 | 1 | 1 |
| 0.1 | 5 | 0.87 |
| 0.1 | 10 | 0.63 |
| 0.1 | 100 | 0.43 |
| 0.1 | 10000 | 0.43 |
| 1 | 0 | 0.43 |
| 1 | 0.01 | 1 |
| 1 | 0.1 | 1 |
| 1 | 1 | 1 |
| 1 | 5 | 1 |
| 1 | 10 | 0.93 |
| 1 | 100 | 0.77 |
| 1 | 10000 | 0.5 |
| 5 | 0 | 1 |
| 5 | 0.01 | 1 |
| 5 | 0.1 | 1 |
| 5 | 1 | 1 |
| 5 | 5 | 1 |
| 5 | 10 | 0.93 |
| 5 | 100 | 0.77 |
| 5 | 10000 | 0.5 |
| 10 | 0 | 1 |
| 10 | 0.01 | 1 |
| 10 | 0.1 | 1 |
| 10 | 1 | 1 |
| 10 | 5 | 1 |
| 10 | 10 | 0.93 |
| 10 | 100 | 0.77 |
| 10 | 10000 | 0.5 |
| 100 | 0 | 1 |
| 100 | 0.01 | 1 |
| 100 | 0.1 | 1 |
| 100 | 1 | 1 |
| 100 | 5 | 1 |
| 100 | 10 | 0.93 |
| 100 | 100 | 0.77 |
| 100 | 10000 | 0.5 |
| 10000 | 0 | 1 |
| 10000 | 0.01 | 1 |
| 10000 | 0.1 | 1 |
| 10000 | 1 | 1 |
| 10000 | 5 | 1 |
| 10000 | 10 | 0.93 |
| 10000 | 100 | 0.77 |
| 10000 | 10000 | 0.5 |