机器学习实验报告

实验名称：实现支持向量机

学生姓名： 唐梓烨

学生学号： 58122310

完成日期： 2024/6/8

目录

任务描述

通过两种方式实现 SVM：
1. 手动实现 SMO 算法，并与直接使用传统二次规划方法进行对比，
2. 通过 scikit-learn 库实现软间隔 SVM。
并在 breast cancer 数据集上进行验证与实验。该数据集是一个二分类问题，属性均为连续属性，并已进行标准化。

数据集简介

"Breast Cancer Wisconsin (Diagnostic) Data Set"数据集是一个著名的机器学习数据集，它通常用于分类任务，即根据肿瘤的特征来预测肿瘤是良性还是恶性。
数据类型：结构化数据
任务类型：分类
样本数量：569
特征数量：30 个数值特征
目标变量：诊断结果，包含两个类别（良性和恶性）

实验内容

1. 手动实现 SVM 的 SMO 算法

SVM 的对偶问题实际是一个二次规划问题，除了 SMO 算法外，传统二次规划方法也可以用于求解对偶问题。求得最优拉格朗日乘子后，超平面参数 ***w***, *b* 可由以下式子得到：

$$\mathbf{w} = \sum_{i=1}^{m} \alpha_i y_i x_i$$

$$b = \frac{1}{|S|} \sum_{s \in S} \left(\frac{1}{y_s} - \sum_{i \in S} \alpha_i y_i \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_s \right).$$

请完成以下任务：

(1) [10pts] 不考虑软间隔情况，直接使用传统二次规划（QP）方法求解（实现）训练集上的硬间隔 SVM 对偶问题。观察并回答，这样的求解方法是否会出现问题，为什么？

(2) [40pts] 不限定硬间隔或是软间隔 SVM，也不限定是否为核化 SVM，根据需要选择合适的方法，手动实现 SMO 算法求解 SVM 对偶问题。注意第 3 步，KKT 条件验证步骤不能缺少。

(3) [10pts] 对测试数据进行预测，确保预测结果尽可能准确。

2. 使用 sklearn 库简洁实现软间隔 SVM

(1) [20pts] 使用 sklearn 库简洁实现软间隔 SVM。首先实现以下 4 个示例性的 SVM 模型：

- 线性 SVM：正则化常数 C=1，核函数为线性核，
- 线性 SVM：正则化常数 C=1000，核函数为线性核，
- 非线性 SVM：正则化常数 C=1，核函数为多项式核，d=2，
- 非线性 SVM：正则化常数 C=1000，核函数为多项式核，d=2，

观察并比较它们在测试集上的性能表现。

(2) [20pts] 参数选择与参数分析

参数的选择对 SVM 的性能有很大的影响。确定正则化常数 C 与核函数及其参数的选择范围（可以在以下常用核函数中选择一种或多种），选用合适的实验评估方法（回顾第 2 章的内容，如 K 折交叉验证法等）进行参数选择，并进行参数分析实验。

实验超参数介绍

1. 正则化参数 C：

描述：控制模型在训练数据中的错误分类和决策边界的复杂性之间的权衡。较大的 CCC 倾向于正确分类训练数据中的所有点(低偏差、高方差)，较小的 CCC 则允许更多的误分类（高偏差、低方差）。

选择：常用值范围包括 0.01, 0.1, 1, 10, 100, 1000 等。

2. 核函数类型：

描述：核函数用于将数据映射到更高维的空间，以使得在该空间中更容易找到线性可分的超平面。常用的核函数包括线性核、多项式核、高斯核（RBF 核）和 Sigmoid 核。

选择：选择合适的核函数类型，如 'linear', 'poly', 'rbf', 'sigmoid'

3. 核函数参数：

1) 多项式核函数（poly）：

degree：多项式的次数。常用值为 2, 3, 4。

coef0：核函数中的独立项。常用值为 0 或 1。

2) 高斯核（RBF 核）：

gamma：核函数的带宽参数，控制单个训练样本的影响范围。常用值范围包括 'scale', 'auto', 0.001, 0.01, 0.1, 1, 10。

3) Sigmoid 核函数：

gamma：与 RBF 核相同，控制影响范围。

coef0：核函数中的独立项。常用值为 0 或 1。

4. 支持向量的容忍度（tol）：

描述：停止标准。控制优化过程中的容忍误差。较小的 tol 可能会导致更长的训练时间，但可能会得到更精确的解。

选择：常用值范围为 1e-4, 1e-3, 1e-2。

实验过程与结果

1. 手动实现 SVM 的 SMO 算法

(1) 不考虑软间隔情况，直接使用传统二次规划（QP）方法求解（实现）训练集上的硬间隔 SVM 对偶问题。观察并回答，这样的求解方法是否会出现问题，为什么？

实验思路：使用传统的二次规划（QP）方法求解硬间隔 SVM 的对偶问题时，可以通过标准的优化库（如 CVXOPT）实现。

硬间隔 SVM 的对偶问题可以表示为以下优化问题：

$$\max_{\alpha} \quad \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j \langle x_i, x_j\rangle$$

$$subject\ to \quad \sum_{i=1}^{n}\alpha_i y_i = 0$$

$$\alpha_i \geq 0, i = 1, \ldots, n$$

使用 CVXOPT 库简洁实现 QP 方法求解硬间隔 SVM 对偶问题：

i. 设置 QP 问题的参数

P：由标签 y 和核矩阵 K 计算得到的矩阵，用于定义目标函数。

q：常量向量，全为 -1，用于定义目标函数的线性项。

G 和 h：用于定义不等式约束，确保拉格朗日乘子 alpha 非负。

A 和 b：用于定义等式约束，确保拉格朗日乘子的加权和为 0。

ii. 求解 QP 问题

使用 cvxopt 库的 qp 方法求解二次规划问题。solution['x'] 包含优化得到的拉格朗日乘子 alphas。

iii. 用实验内容部分给出的公式计算权重向量 w、支持向量、偏置项 b

实验结果：在训练集上训练的模型在测试集上的分类准确率为 91%。

使用 QP 求解硬间隔 SVM 可能存在的问题：

i. 最主要的问题就是数据集不可分。硬间隔 SVM 要求数据完全线性可分，如果数据集不可分，则二次规划问题可能无法求解或求解结果无意义。这也是影响模型分类准确率的重要因素。

Ii. 其他可能存在的问题包括 QP 算法的计算复杂度过高；在求解 QP 问题时，数值稳定性可能成为问题，特别是在处理大规模或高维数据时，可能会导致解的精度下降；由于 QP 求解需要存储核矩阵 K，内存消耗为 $O(n^2)$。对于大规模数据集，内存消耗可能成为瓶颈。

(2) 不限定硬间隔或是软间隔 SVM，也不限定是否为核化 SVM，根据需要选择合适的方法，手动实现 SMO 算法求解 SVM 对偶问题。对测试数据进行预测，确保预测结果尽可能准确。

实验思路：选择高斯核作为核函数，手动实现 SMO 算法求解软间隔 SVM 用于实验。其核心思想是通过选择一对拉格朗日乘子来优化目标函数，每次选择一对乘子进行优化，而将其他乘子视为常数。

i. 初始化模型参数，实现高斯核函数

ii. 使用 SMO 算法来拟合 SVM 模型，通过迭代更新拉格朗日乘子来优化模型参数。

初始化：初始化拉格朗日乘子α为零向量，设置迭代计数器为 0。计算核矩阵 K，其中 K[i, j]表示样本 i 和样本 j 之间的核函数值。

选择一对乘子：先遍历所有的α，选择违反 KKT 条件的乘子作为第一个乘子α_i。再根据第一个乘子α_i选择第二个乘子α_j。

更新乘子：计算乘子α_i和α_j的边界 L 和 H，保证它们满足约束条件 $0 \leq \alpha \leq C$。计算优化前后的目标函数值。如果乘子α_j的变化小于阈值，跳过本次迭代，否则更新乘子α_i和α_j，更新阈值 b。

迭代进行以上步骤即可找到最优化的拉格朗日乘子α，用于计算支持向量和决策函数。

实验结果：

在 KKT 条件验证时，发现有些支持向量并不完全满足 KKT 条件。

原因：

i. 在 SMO 算法中，通过选择一对乘子来优化目标函数，每次选择的一对违反 KKT 条件的乘子。然后通过计算新的乘子值并对其进行剪切，可以保证在一定程度上满足 KKT 条件。然而，在实际应用中，由于 SMO 算法是一种启发式算法，通过不断迭代更新乘子来逼近目标函数的最优解，而不是通过解析求解直接得到。因此，在每次迭代中，选择的乘子对可能只是局部最优的，不一定是全局最优的，可能会导致一些边界情况下不满足 KKT 条件。

ii. 在软间隔支持向量机中也会影响 KKT 条件的满足性。具体来说，在软间隔支持向量机中，有一些样本可能会落在间隔边界上，或者在超平面错误的一侧。这些样本将对应于非零的拉格朗日乘子α，其值在 0 到 C 之间。对于这些样本，KKT 条件中的不等式约束可能不再严格成立。在实际优化过程中，软间隔支持向量机通常通过放宽 KKT 条件来处理这些情况，允许一定程度的违反。

最终选择高斯核作为核函数的软间隔 SVM 在测试集上的分类准确率达到了 98%，效果优异。

原因：

i. 高斯核函数的优点:

高斯核函数具有非线性映射能力，通过引入非线性映射，能够更好地捕捉数据的非线性关系。这种非线性映射能力使得模型能够更好地适应不同类型的数据分布，提高了模型的泛化能力。

高斯核函数还有局部适应性，高斯核函数在计算样本之间的相似性时，对距离较近的样本给予较高的相似度，而对距离较远的样本给予较低的相似度。这种局部适应性使得模型能够更好地捕捉数据的局部特征，提高了模型对噪声和异常值的鲁棒性，从而增强了泛化能力。

ii. 软间隔的优点：

软间隔支持向量机通过引入松弛变量来处理不可分的数据点，减少了模型对噪声和异常值的敏感性。这种机制使得模型更具鲁棒性，能够更好地处理真实世界中复杂的数据，提高了泛化能力。

2. 使用 sklearn 库简洁实现软间隔 SVM

(1) 首先实现以下 4 个示例性的 SVM 模型：

- 线性 SVM：正则化常数 C=1，核函数为线性核，

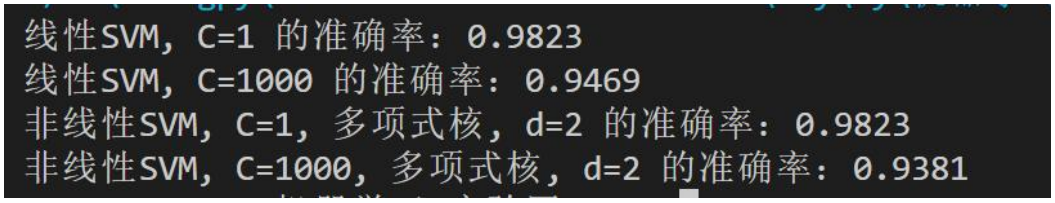
- 线性 SVM：正则化常数 C=1000，核函数为线性核，
- 非线性 SVM：正则化常数 C=1，核函数为多项式核，d=2，
- 非线性 SVM：正则化常数 C=1000，核函数为多项式核，d=2，

观察并比较它们在测试集上的性能表现。

实验思路：

使用 sklearn.svm 的 SVC 类仅需传入相关参数即可十分方便的实例化目标 SVM。再将模型进行训练即可。

实验结果：



结果分析：

C 参数是正则化参数，它控制了模型对误分类样本的惩罚程度。C 较小的情况下，模型对误分类样本的惩罚较轻，容错性较强，可能会导致模型过于简单，出现欠拟合；而 C 较大的情况下，模型对误分类样本的惩罚较重，容错性较低，可能会导致模型过于复杂，出现过拟合。C=1 时的性能比 C=1000 时更好，这可能是因为 C=1 时模型对数据集容错性更强，能够更好地适应测试数据集中的未见数据。

而在非线性 SVM 中，除了正则化参数 C 外，还引入了核函数和多项式核的次数 d 作为超参数。多项式核函数通过引入高维空间的多项式特征来进行非线性映射，可以更好地处理非线性问题。C=1000 时的准确率较低可能是由于模型过于复杂，导致了过拟合的问题。

(2) 参数选择与参数分析

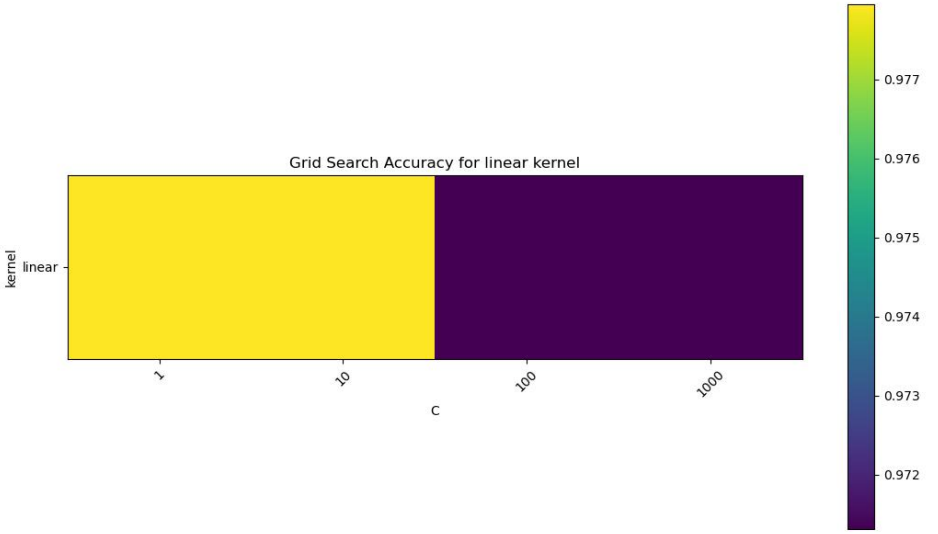
参数的选择对 SVM 的性能有很大的影响。确定正则化常数 C 与核函数及其参数的选择范围（可以在以下常用核函数中选择一种或多种），选用合适的实验评估方法（回顾第 2 章的内容，如 K 折交叉验证法等）进行参数选择，并进行参数分析实验。

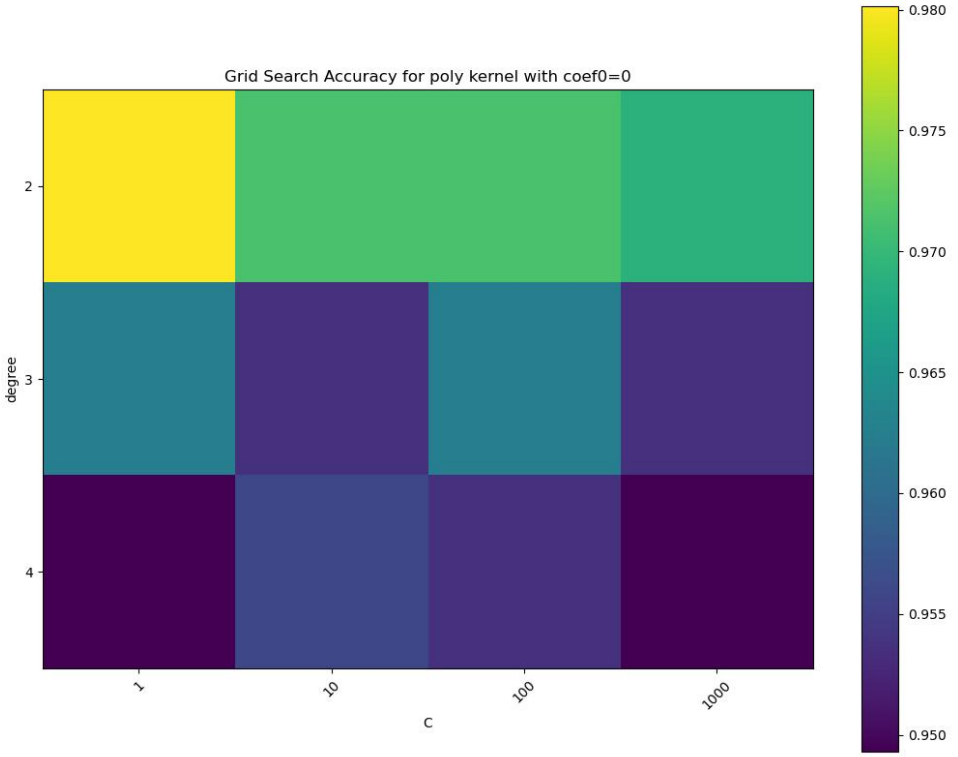
实验思路：

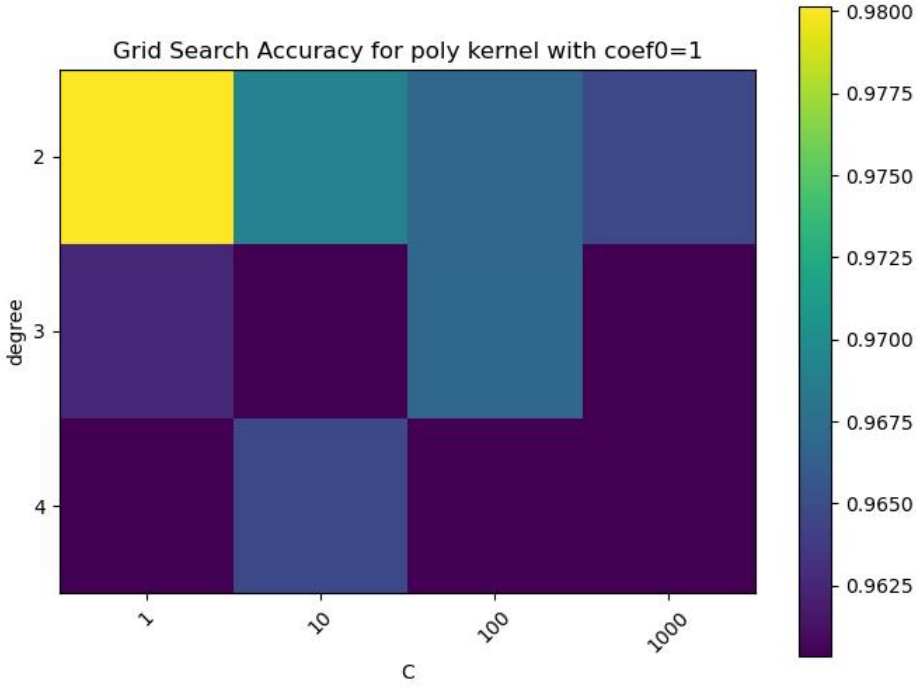
选择采用网格搜索的方法，在给定的参数空间中逐一尝试每种可能的参数组合。并选用 K 折交叉验证的评估方法进行评估比，降低数据集随机性。

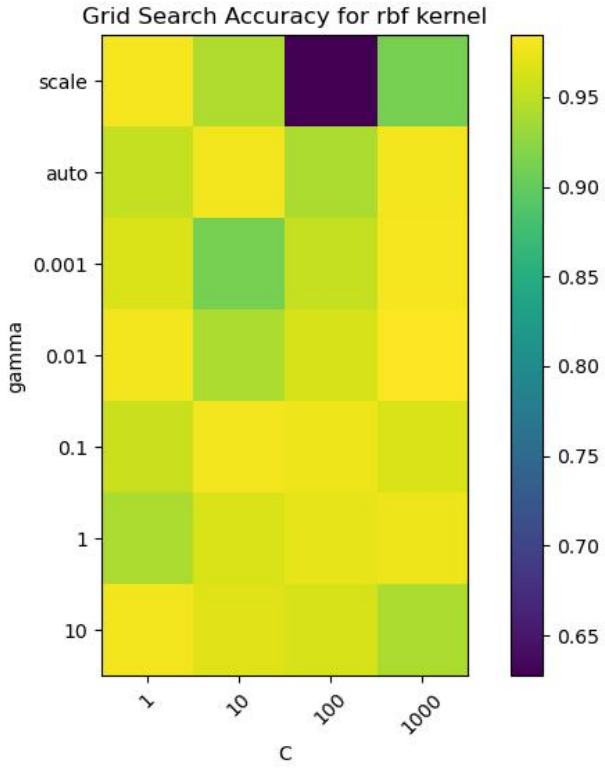
网格搜索可以使用 sklearn.model_selection 的 GridSearchCV 类，可以很方便的实现不同参数类型的组合。主要实现了 C: [1, 10, 100, 1000]，kernel: ['linear'，'poly'，'rbf'，'sigmoid']，gamma: ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10]之间的组合，对于非线性核函数还尝试了它的不同的维度 d。

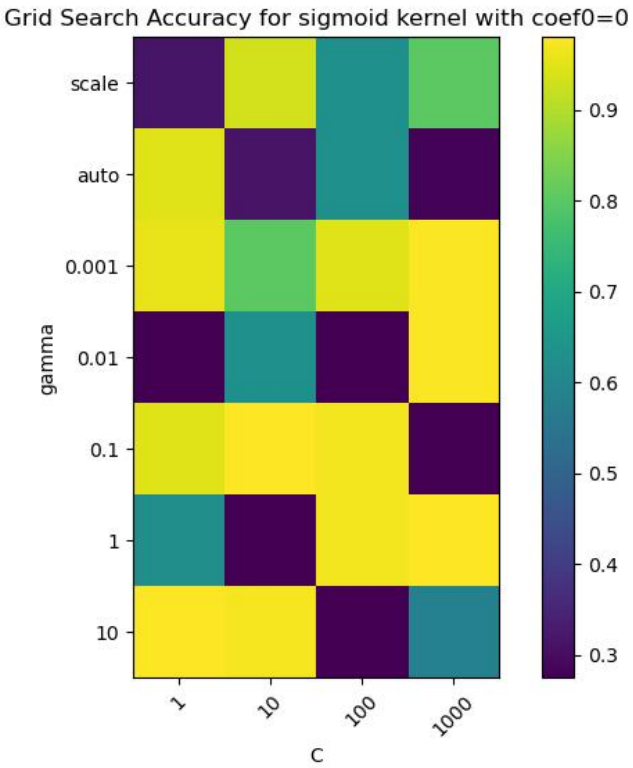
不同组合的准确度可视化：

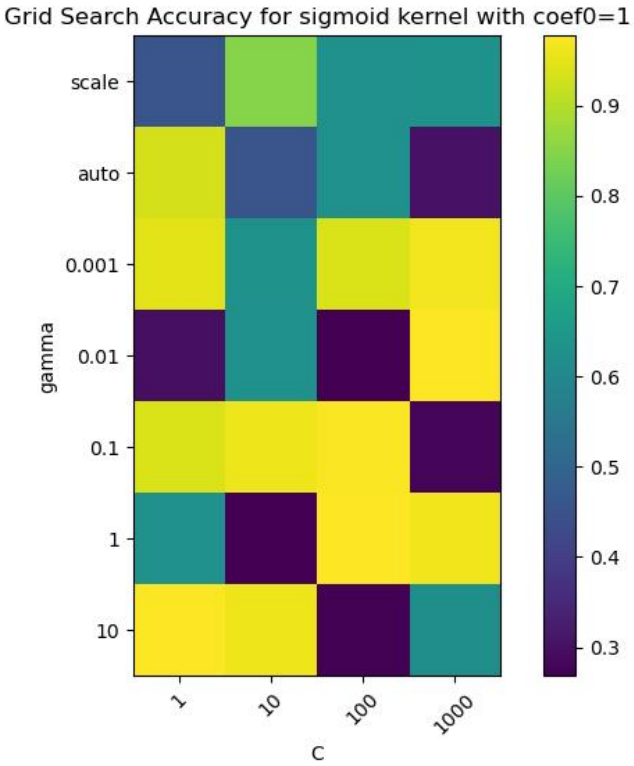

Grid Search Accuracy for linear kernel
kernel
linear
1
10
100
1000
C
0.977
0.976
0.975
0.974
0.973
0.972


Grid Search Accuracy for poly kernel with coef0=0
degree
2
3
4
1
10
100
1000
C
0.980
0.975
0.970
0.965
0.960
0.955
0.950


Grid Search Accuracy for poly kernel with coef0=1
degree
2
3
4
C
1
10
100
1000
0.9800
0.9775
0.9750
0.9725
0.9700
0.9675
0.9650
0.9625


Grid Search Accuracy for rbf kernel
gamma
scale
auto
0.001
0.01
0.1
1
10
C
1
10
100
1000
0.95
0.90
0.85
0.80
0.75
0.70
0.65





最终得到准确度最高的参数组合：

```
Best Parameters: {'C': 100, 'gamma': 'auto', 'kernel': 'rbf'}
Best Accuracy: 0.9845909645909646
```

代码附录

SVM.qp

```
import numpy as np
import pandas as pd
from cvxopt import matrix, solvers

# 定义硬间隔支持向量机类
class HardMarginSVM:
    def __init__(self):
        self.w = None # 权重向量
        self.b = None # 偏置项
    def fit(self, X, y):
        n_samples, n_features = X.shape

        y = y.flatten()
        # 定义二次规划问题
        K = np.dot(X, X.T)
        P = matrix(np.outer(y, y) * K)
        q = matrix(-np.ones(n_samples))
        G = matrix(-np.eye(n_samples))
        h = matrix(np.zeros(n_samples))
        A = matrix(y, (1, n_samples), 'd')
        b = matrix(0.0)
        # 解决二次规划问题
        solvers.options['show_progress'] = False
        solution = solvers.qp(P, q, G, h, A, b)
        alphas = np.array(solution['x']).flatten()
        # 计算权重向量和偏置项
        self.w = np.sum((alphas * y)[:, None] * X, axis=0)
        support_vector_indices = np.where(alphas > 1e-5)[0]
        self.b = np.mean(y[support_vector_indices] - np.dot(X[support_vector_indices], self.w))
    def predict(self, X):
        return np.sign(np.dot(X, self.w) + self.b)
def Accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
# 读取数据
X_train = pd.read_csv('breast_cancer_Xtrain.csv', header=0).values
X_test = pd.read_csv('breast_cancer_Xtest.csv', header=0).values
y_train = pd.read_csv('breast_cancer_Ytrain.csv', header=0).values.flatten()  # Ensure y is 1D
y_test = pd.read_csv('breast_cancer_Ytest.csv', header=0).values.flatten()  # Ensure y is 1D
```

```
# 创建并训练 SVM 模型
svm = HardMarginSVM()
svm.fit(X_train, y_train)
# 预测并计算准确率
y_pred = svm.predict(X_test)
accuracy = Accuracy(y_test, y_pred)
print("准确率: ", accuracy)
```

SVM.smo

```
import numpy as np
import pandas as pd
import random


class SVM:
    def __init__(self, C=1.0, kernel='linear', gamma=0.1, tol=0.001, max_iter=40):
        self.C = C  # 正则化参数
        self.kernel = kernel  # 核函数类型
        self.gamma = gamma  # 高斯核函数的带宽参数
        self.tol = tol  # 迭代停止的阈值
        self.max_iter = max_iter  # 最大迭代次数
        self.b = 0  # 偏置项
        self.alpha = None  # 拉格朗日乘子
        self.support_vectors_ = None  # 支持向量
        self.dual_coef_ = None  # 决策函数的系数
        self.support_ = None  # 支持向量的索引

    # 随机选择 j
    def _select_j(self, i, m):
        j = i
        while j == i:
            j = random.randint(0, m - 1)
        return j
    # 限制 alpha 的范围
    def _clip_alpha(self, alpha, H, L):
        return max(L, min(alpha, H))
    # 核函数计算
    def _kernel(self, X1, X2):
        if self.kernel == 'rbf':
            if X2 is None:
                X2 = X1
            K = np.zeros((X1.shape[0], X2.shape[0]))
            for i in range(X1.shape[0]):
                for j in range(X2.shape[0]):
                    K[i, j] = np.exp(-self.gamma * np.linalg.norm(X1[i] - X2[j]) ** 2)
```

```
            return K
    # 使用 SMO 算法来拟合 SVM 模型，通过迭代更新拉格朗日乘子来优化模型参数。
    def fit(self, X, y):
        self.n_samples, self.n_features = X.shape
        self.alpha = np.zeros(self.n_samples) # 初始化拉格朗日乘子为零向量
        iter_num = 0 # 初始化迭代次数
        K = self._kernel(X, None) # 计算核矩阵 K
        # 开始迭代训练模型
        while iter_num < self.max_iter:
            alpha_changed = 0 # 用于记录在当前迭代中是否有拉格朗日乘子更新
            for i in range(self.n_samples):
                # 计算第 i 个样本的预测误差
                E_i = self._E(i, K, y)
                # 检查是否满足 KKT 条件
                if (y[i] * E_i < -self.tol and self.alpha[i] < self.C) or (y[i] * E_i > self.tol
and self.alpha[i] > 0):
                    # 选择第二个乘子 j
                    j = self._select_j(i, self.n_samples)
                    # 计算第 j 个样本的预测误差
                    E_j = self._E(j, K, y)
                    alpha_i_old = self.alpha[i]
                    alpha_j_old = self.alpha[j]
                    L, H = self._compute_L_H(alpha_i_old, alpha_j_old, y[i], y[j])
                    # 如果 L 和 H 相等，则跳过本次迭代
                    if L == H:
                        continue
                    # 计算 eta，如果 eta 大于等于零，则跳过本次迭代
                    eta = 2 * K[i, j] - K[i, i] - K[j, j]
                    if eta >= 0:
                        continue
                    # 更新第二个乘子 alpha_j
                    self.alpha[j] -= y[j] * (E_i - E_j) / eta
                    self.alpha[j] = self._clip_alpha(self.alpha[j], H, L)
                    # 如果 alpha_j 的变化量小于给定阈值，则跳过本次迭代
                    if abs(self.alpha[j] - alpha_j_old) < 1e-5:
                        continue
                    # 更新参数
                    self.alpha[i] += y[j] * y[i] * (alpha_j_old - self.alpha[j])
                    b1, b2 = self._compute_b(E_i, E_j, K, i, j, alpha_i_old, alpha_j_old, y)
                    self.b = self._update_b(self.alpha[i], self.alpha[j], b1, b2)
                    # 记录有乘子更新
                    alpha_changed += 1
            iter_num = iter_num + 1 if alpha_changed == 0 else 0
        # 计算支持向量和决策函数系数
```

```
        self.support_ = np.where(self.alpha > 0)[0]
        self.support_vectors_ = X[self.support_]
        self.dual_coef_ = self.alpha[self.support_] * y[self.support_]
    # 计算样本点的误差
    def _E(self, i, K, y):
        return np.dot(self.alpha * y, K[:, i]) + self.b - y[i]
    # 计算 L 和 H
    def _compute_L_H(self, alpha_i_old, alpha_j_old, y_i, y_j):
        if y_i != y_j:
            L = max(0, alpha_j_old - alpha_i_old)
            H = min(self.C, self.C + alpha_j_old - alpha_i_old)
        else:
            L = max(0, alpha_j_old + alpha_i_old - self.C)
            H = min(self.C, alpha_j_old + alpha_i_old)
        return L, H
    # 计算 b1 和 b2
    def _compute_b(self, E_i, E_j, K, i, j, alpha_i_old, alpha_j_old, y):
        b1 = self.b - E_i - y[i] * (self.alpha[i] - alpha_i_old) * K[i, i] - y[j] * (self.alpha[j]
- alpha_j_old) * K[i, j]
        b2 = self.b - E_j - y[i] * (self.alpha[i] - alpha_i_old) * K[i, j] - y[j] * (self.alpha[j]
- alpha_j_old) * K[j, j]
        return b1, b2
    # 更新 b
    def _update_b(self, alpha_i, alpha_j, b1, b2):
        if 0 < alpha_i < self.C:
            return b1
        elif 0 < alpha_j < self.C:
            return b2
        else:
            return (b1 + b2) / 2
    # 进行分类
    def predict(self, X):
        K = self._kernel(X, self.support_vectors_)
        return np.sign(np.dot(K, self.dual_coef_) + self.b)
def Accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
# 读取数据
X_train = pd.read_csv('breast_cancer_Xtrain.csv', header=0).values
X_test = pd.read_csv('breast_cancer_Xtest.csv', header=0).values
y_train = pd.read_csv('breast_cancer_Ytrain.csv', header=0).values.flatten()  # Ensure y is 1D
y_test = pd.read_csv('breast_cancer_Ytest.csv', header=0).values.flatten()  # Ensure y is 1D
# 创建并训练 SVM 模型
classifier = SVM(C=2, kernel='rbf', gamma=0.1)
classifier.fit(X_train, y_train)
```

```
# 预测并计算准确率
y_pred = classifier.predict(X_test)
accuracy = Accuracy(y_test, y_pred)
print("准确率: ", accuracy)
```

SVM.skl

```
import numpy as np
import pandas as pd
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# 读取数据
X_train = pd.read_csv('breast_cancer_Xtrain.csv', header=0).values
X_test = pd.read_csv('breast_cancer_Xtest.csv', header=0).values
y_train = pd.read_csv('breast_cancer_Ytrain.csv', header=0).values.flatten()  # 确保 y 是一维的
y_test = pd.read_csv('breast_cancer_Ytest.csv', header=0).values.flatten()  # 确保 y 是一维的
# 创建和训练 SVM 模型
models = [
    ("线性 SVM, C=1", SVC(C=1, kernel='linear')),
    ("线性 SVM, C=1000", SVC(C=1000, kernel='linear')),
    ("非线性 SVM, C=1, 多项式核, d=2", SVC(C=1, kernel='poly', degree=2)),
    ("非线性 SVM, C=1000, 多项式核, d=2", SVC(C=1000, kernel='poly', degree=2))
]
# 评估模型
for name, model in models:
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"{name} 的准确率: {accuracy:.4f}")
```

SVM.args

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV, StratifiedKFold
from sklearn.metrics import accuracy_score
# 读取数据
X_train = pd.read_csv('breast_cancer_Xtrain.csv', header=0).values
X_test = pd.read_csv('breast_cancer_Xtest.csv', header=0).values
y_train = pd.read_csv('breast_cancer_Ytrain.csv', header=0).values.flatten()  # 确保 y 是一维的
y_test = pd.read_csv('breast_cancer_Ytest.csv', header=0).values.flatten()  # 确保 y 是一维的
# 定义参数网格
param_grid = [
```

```
    {
        'C': [1, 10, 100, 1000],
        'kernel': ['linear']
    },
    {
        'C': [1, 10, 100, 1000],
        'kernel': ['poly'],
        'degree': [2, 3, 4],
        'coef0': [0, 1]  # 偏移量 c
    },
    {
        'C': [1, 10, 100, 1000],
        'kernel': ['rbf'],
        'gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10]
    },
    {
        'C': [1, 10, 100, 1000],
        'kernel': ['sigmoid'],
        'gamma': ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10],
        'coef0': [0, 1]  # 偏移量 c
    }
]
# 进行网格搜索
grid_search = GridSearchCV(SVC(), param_grid, cv=StratifiedKFold(n_splits=5), scoring='accuracy',
n_jobs=-1, return_train_score=True)
grid_search.fit(X_train, y_train)
# 得到每个参数组合的交叉验证准确率
results = pd.DataFrame(grid_search.cv_results_)
results = results[['params', 'mean_test_score', 'rank_test_score']]
# 输出最佳参数组合和最佳模型的分类准确率
best_params = grid_search.best_params_
best_score = grid_search.best_score_
print("Best Parameters:", best_params)
print("Best Accuracy:", best_score)
# 绘制图表函数
def plot_grid_search(cv_results, param1, param2, param1_name, param2_name, title):
    scores_mean = cv_results['mean_test_score']
    scores_mean = np.array(scores_mean).reshape(len(param2), len(param1))
    # 绘制热力图
    plt.figure(figsize=(8, 6))
    plt.imshow(scores_mean, interpolation='nearest', cmap='viridis')
    plt.xlabel(param1_name)
    plt.ylabel(param2_name)
    plt.colorbar()
```

```
    plt.title(title)
    plt.xticks(np.arange(len(param1)), param1, rotation=45)
    plt.yticks(np.arange(len(param2)), param2)
    plt.show()
# 提取参数组合并绘制图表
for kernel in ['linear', 'poly', 'rbf', 'sigmoid']:
    if kernel == 'linear':
        param1_name = 'C'
        param2_name = 'kernel'
        param1 = [1, 10, 100, 1000]
        param2 = [kernel]
        plot_grid_search(results[results['params'].apply(lambda x: x['kernel'] == kernel)],
param1, param2, param1_name, param2_name, f'Grid Search Accuracy for {kernel} kernel')
    elif kernel == 'poly':
        for coef0 in [0, 1]:
            param1_name = 'C'
            param2_name = 'degree'
            param1 = [1, 10, 100, 1000]
            param2 = [2, 3, 4]
            filtered_results = results[results['params'].apply(lambda x: x['kernel'] == kernel and
x['coef0'] == coef0)]
            if len(filtered_results) == len(param1) * len(param2):  # Ensure complete data
                plot_grid_search(filtered_results, param1, param2, param1_name, param2_name,
f'Grid Search Accuracy for {kernel} kernel with coef0={coef0}')
    elif kernel == 'rbf':
        param1_name = 'C'
        param2_name = 'gamma'
        param1 = [1, 10, 100, 1000]
        param2 = ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10]
        filtered_results = results[results['params'].apply(lambda x: x['kernel'] == kernel)]
        if len(filtered_results) == len(param1) * len(param2):  # Ensure complete data
            plot_grid_search(filtered_results, param1, param2, param1_name, param2_name, f'Grid
Search Accuracy for {kernel} kernel')
    elif kernel == 'sigmoid':
        for coef0 in [0, 1]:
            param1_name = 'C'
            param2_name = 'gamma'
            param1 = [1, 10, 100, 1000]
            param2 = ['scale', 'auto', 0.001, 0.01, 0.1, 1, 10]
            filtered_results = results[results['params'].apply(lambda x: x['kernel'] == kernel and
x['coef0'] == coef0)]
            if len(filtered_results) == len(param1) * len(param2):  # Ensure complete data
                plot_grid_search(filtered_results, param1, param2, param1_name, param2_name,
f'Grid Search Accuracy for {kernel} kernel with coef0={coef0}')
```