(SE332-1: Machine Learning-Fall 2022)Project 1-SVM

520030910314 肖蔚尔

1. Gradient descent SVM

1. 实现过程

- 预处理（Feature Engineering）

  数据集由特征（X:feature）和标签（Y:label）组成，原始data中每一行都是一个示例。大多数情况下，原始数据要么与模型不兼容，要么妨碍其性能。为解决以上问题而使用预处理技术：通过**从原始数据中提取特征来生成规范数据集。 归一化**是将一系列值转换为标准值范围的过程，通常在区间 [-1, 1] 或 [0, 1] 中。这不是必须的要求，但它提高了学习速度（例如，梯度下降的收敛速度更快）并防止数值溢出。在*load_data()*函数中添加以下代码以规范化：

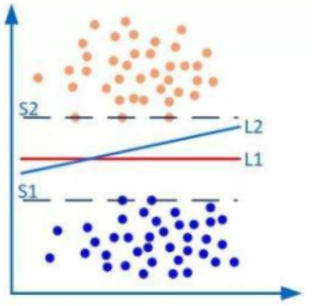
  ```
  # Y输入为01，SVM中为+-1!!!
  Y_train = Y_train * 2 - 1
  Y_test = Y_test * 2 - 1
  # 预处理feature——均值-标准差缩放
  X_train_normal = preprocessing.scale(X_train)
  X_train = pandas.DataFrame(X_train_normal)
  X_test_normal = preprocessing.scale(X_test)
  X_test = pandas.DataFrame(X_test_normal)
  ```

  优化效果：

	时间	准确率
MinMaxScaler优化前	66.07	89.50%
MinMaxScaler优化后	33.94	91.00%

- 计算损失函数

  SVM的目标是找到一个超平面，以最大的边距二分点集示例，同时保持错误分类尽可能低

  要实现这一目标？我们将最小化如下所示的成本/目标函数：

$$J(\mathbf{w}) = \frac{1}{2}\|\mathbf{w}\|^2 + C\left[\frac{1}{N}\sum_i^n \max(0,\ 1 - y_i * (\mathbf{w}\cdot x_i + b))\right]$$

compute_cost()中截距b并未出现，因为截距已在数据处理中被推入权重向量

```
# compute_cost()
N = X.shape[0]
distances = 1 - Y * (np.dot(X, W))
distances[distances < 0] = 0  # = max(0, distance)
hinge_loss = reg_strength * (np.sum(distances) / N)
cost = 1 / 2 * np.dot(W, W) + hinge_loss
return cost
```

```
# 数据处理时的向量操作： 给X_train添加一列值为1，为了将截距b推入权重向量w中
X_train.insert(loc=len(X_train.columns), column='intercept',
value=1)
X_train=X_train.to_numpy()
# X_test格式需与X_train统一！！！
X_test.insert(loc=len(X_test.columns), column='intercept', value=1)
X_test=X_test.to_numpy()
```

- 计算损失函数的梯度

  损失函数形式变化，等价于原形式，如下图一式；用下图二式实现 *calculate_cost_gradient*()

$$J(\mathbf{w}) = \frac{1}{N}\sum_i^n \left[\frac{1}{2}\|\mathbf{w}\|^2 + C\max(0,\ 1 - y_i * (\mathbf{w}\cdot x_i))\right]$$

$$\nabla_w J(\mathbf{w}) = \frac{1}{N}\sum_i^n \begin{cases} \mathbf{w} & \text{if } \max(0,\ 1 - y_i * (\mathbf{w}\cdot x_i)) = 0 \\ \mathbf{w} - Cy_i\, x_i & \text{otherwise} \end{cases}$$

- 随机梯度下降

  在SVM中最小化损失函数的原因是，损失函数本质上是衡量模型训练效果优劣的指标。要找到J(w)的最小值则必须：

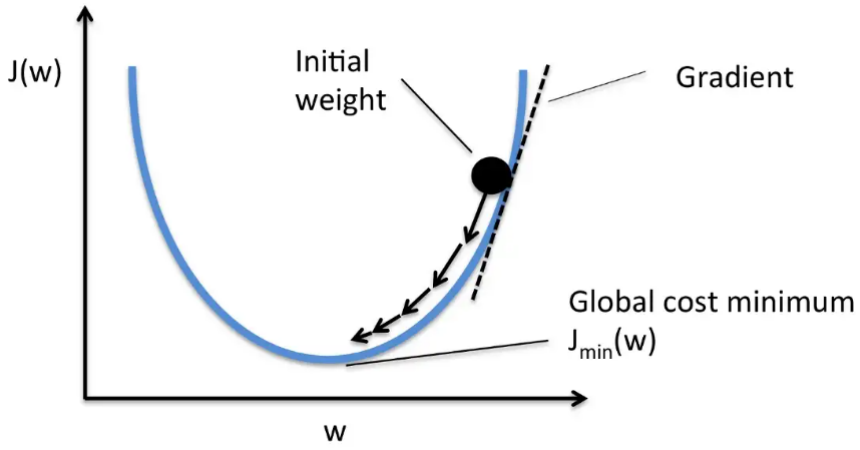
  1. *Minimize* ||w||² which *maximizes margin* (2/||w||)
  2. *Minimize the sum of hinge loss* which *minimizes misclassifications*.

  其中hinge loss如下：

$$\max(0,\ 1 - y_i * (w\cdot x_i + b))$$

  使用梯度下降法实现最小化，原理：

  1. 求成本函数的梯度，即∇J(w')
  2. 以一定的速率与梯度相反移动，即 w' = w' — ∝(∇J(w'))
  3. 重复步骤 1-3 直到收敛，即我们发现 w' 其中 J(w) 最小



- 梯度下降终止条件

  如果迭代之间的改进不大于一个小的阈值，或者迭代的次数已经达到了预先指定的最大值，则可以终止学习过程。

```
# 在第 2^nth 次迭代进行收敛检验：迭代之间的改进不大于一个小的阈值
if iter_time == 2 ** nth or iter_time == max_iterations - 1:
    cost = compute_cost(weights, features_x, outputs_y)
    print("Iter_time is:{} and Cost is: {}".format(iter_time,
cost))
    # 终止迭代：cost变化量 < 当前cost*cost_threshold
    if abs(prev_cost - cost) < cost_threshold * prev_cost:
        return weights
    prev_cost = cost
    nth += 1
```

2. 预测X_test、X_train数据集结果精度&时间

- 预测X_test

Iter_time	1	2	3	4	5	6	7	8	9	10	AVE
Time	33.90	34.93	34.28	33.96	33.68	34.52	32.68	34.90	33.67	32.84	33.93

  由于多次求平均的操作中每次训练数据集都使用X_train中全部数据，因此预测结果准确率始终一致为：**91.5%**

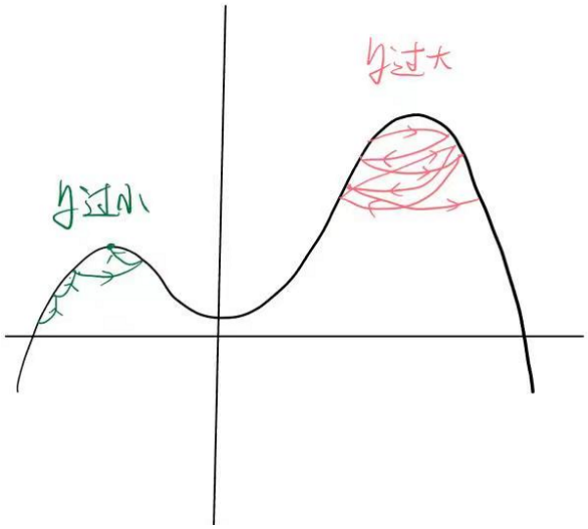
- 预测X_train

Iter_time	1	2	3	4	5	6	7	8	9	10	AVE
Time	33.11	26.04	31.76	31.84	34.53	34.06	34.08	33.30	33.45	34.16	32.63

  由于多次求平均的操作中每次训练数据集都使用X_train中全部数据，因此预测结果准确率始终一致为：**89.50%**

3. 梯度下降过程中cost变化及参数取值

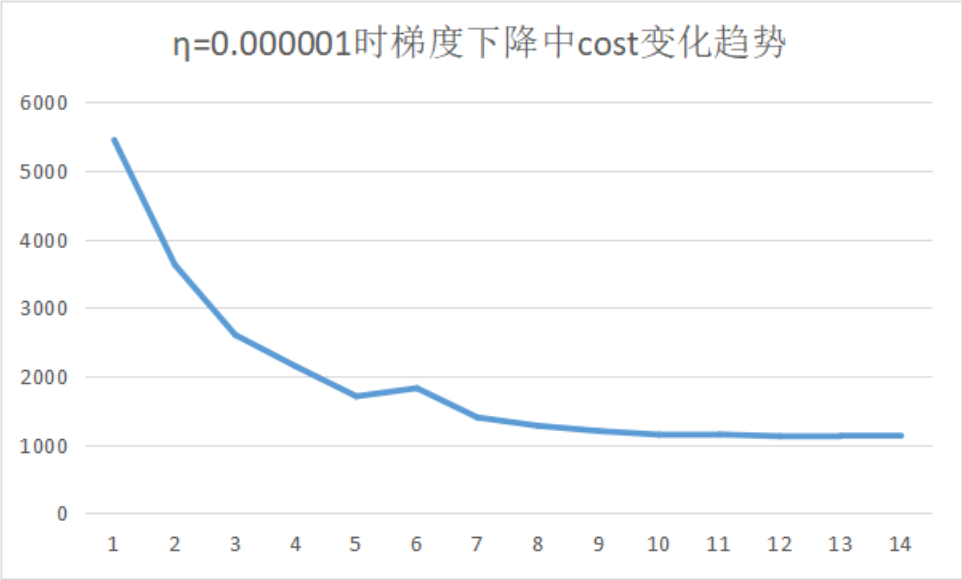
经过观察，若学习率η过大时梯度变化过快，可能会使损失函数直接越过最优点，容易发生梯度爆炸，cost震动幅度较大，模型难以收敛。如果学习率过小，损失函数的变化速度过慢，耗时长时间性能差，且有可能只能取得局部最优。



name	value	explanation
max_iterations	5000	最大迭代次数
reg_strength	10000	正则化强度
learning_rate	0.000001	学习率
cost_threshold	0.0001	阈值设置（将Δcost与当前cost的1%比较）

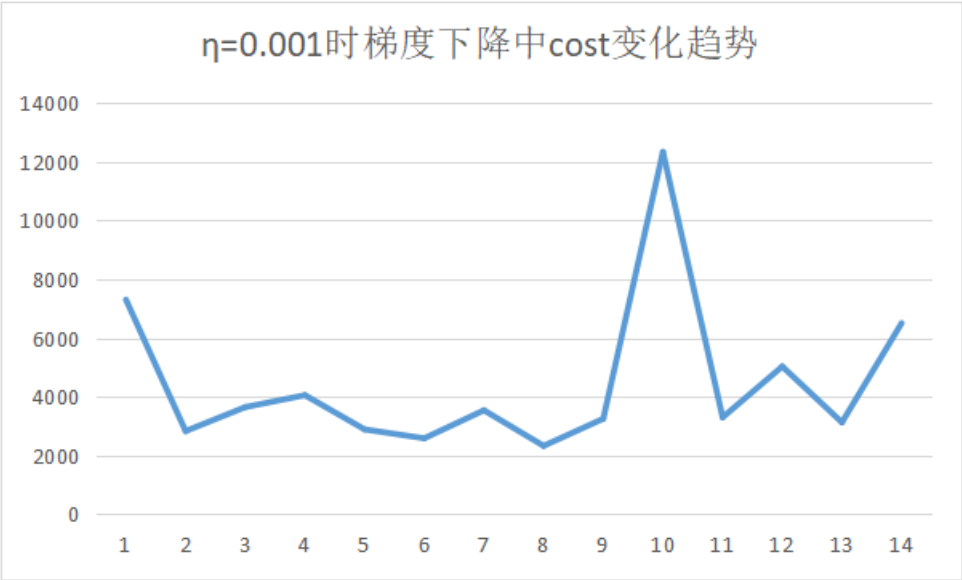
参数取上表中值时cost输出原始数据及变化折线图如下，可直观地看出cost梯度下降且收敛。

```
training started...
Iter_time is:1 and Cost is: 5450.674931077067
Iter_time is:2 and Cost is: 3628.26715212214
Iter_time is:4 and Cost is: 2602.0551559007095
Iter_time is:8 and Cost is: 2143.2324882322196
Iter_time is:16 and Cost is: 1705.7698076819026
Iter_time is:32 and Cost is: 1824.6795790607202
Iter_time is:64 and Cost is: 1396.7252628071915
Iter_time is:128 and Cost is: 1275.5766749371558
Iter_time is:256 and Cost is: 1199.2147024900041
Iter_time is:512 and Cost is: 1145.5189776489278
Iter_time is:1024 and Cost is: 1148.9629066955536
Iter_time is:2048 and Cost is: 1122.5907447352215
Iter_time is:4096 and Cost is: 1130.7913333048155
Iter_time is:4999 and Cost is: 1121.7676748047293
training finished...
```



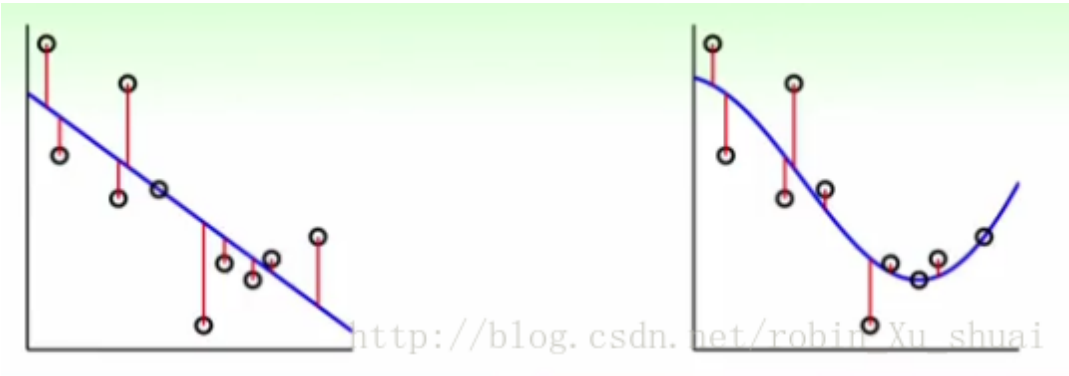
η=0.001（过大）时cost变化如下：

```
training started...
Iter_time is:1 and Cost is: 7307.494775991675
Iter_time is:2 and Cost is: 2829.0103300442693
Iter_time is:4 and Cost is: 3647.148425862651
Iter_time is:8 and Cost is: 4053.206980335325
Iter_time is:16 and Cost is: 2887.577237519697
Iter_time is:32 and Cost is: 2584.0807050320686
Iter_time is:64 and Cost is: 3539.892026962331
Iter_time is:128 and Cost is: 2326.903814782299
Iter_time is:256 and Cost is: 3251.1490245118716
Iter_time is:512 and Cost is: 12344.37186423632
Iter_time is:1024 and Cost is: 3296.4582040651476
Iter_time is:2048 and Cost is: 5032.831467296087
Iter_time is:4096 and Cost is: 3127.784245036868
Iter_time is:4999 and Cost is: 6508.577412398779
training finished...
```



2. Sklearn SVM--梯度下降实现支持向量机

1. 参数设置-修改linear为kernal函数



左边是linear ridge regressionlinear ridge regression的解：

- $w = (\lambda I + X^T X)^{-1} X^T y$

右边是kernel ridge regressionkernel ridge regression的解

- $\beta = (\lambda I + K)^{-1} y$

> 在linear ridge regressionlinear ridge regression中(λI+XTX)(λI+XTX)是d×dd×d的，该算法时间复杂度可以记为O(d3+d2N)O(d3+d2N)；在kernel ridge regressionkernel ridge regression中(λI+K)(λI+K)是N×NN×N的。时间复杂度为O(N3)O(N3)。当N>dNd的时候，使用linear ridge regressionlinear ridge regression比较有效；当资料量NN很大的时候，kernel ridge regressionkernel ridge regression是比较难做的，但是它有很大的自由度，可以做出复杂的曲线。所以使用线性的模型，通常关注的是模型的复杂度和效率；但是当问题很复杂的时候，通常会使用kernelkernel来对数据进行建模，只是缺点就是要付出计算上的复杂度。(参考：https://blog.csdn.net/robin_xu_shuai/article/details/77584906)

```
# 选择不同kernal函数
for fig_num, kernel in enumerate(('linear', 'poly', 'rbf')):
    correct_rate_tot = 0.0
    time_tot = 0.0
    # 多次训练+预测求平均
    for i in range(N):
        cr, tt = svm_sklearn(kernel)
        correct_rate_tot += cr
        time_tot += tt

    avg_cr_tot = correct_rate_tot / N
    avg_tt = time_tot / N
#......
# 训练
    svm = SVC(kernel=kernel)
```

2. 预测精度&时间

kernal func	linear	poly	rbf
ave correct rate	91.00%	46.50%	79.00%
time	1.143796	0.017401	0.020695

3. Performance comparison--性能比较&分析

	时间	准确率
SGD：MinMaxScaler优化前	66.07	89.50%
SGD：MinMaxScaler优化后	33.94	91.00%
sklearn：linear	1.143796	91.00%
sklearn：poly	0.017401	46.50%
sklearn：rbf	0.020695	79.00%

分析可知：对于用SGD实现的SVM，原始数据预处理能有效缩短运行时间，提高准确率，具体原因上文已介绍，不再赘述；

对于sklearn实现的SVM，使用linear函数和不同的kernal函数所用的时间也不同，具体原因上文已介绍，不再赘述；

对于SGD和sklearn对比而言，使用包装库中的svm显然准确率优良，且十分快捷，而自行实现的梯度下降版本准确率做到与调库齐平，较为优良，但时间远长于调库，分析原因可能为：自行实现的代码中包含print输出内容，显然为可避免耗时；其中对矩阵的调用和操作可以进一步简化从而节省时间；代码逻辑可进一步优化；重要的梯度下降迭代部分学习率很难达到最优，且很难与阈值相匹配实现准确率最优且性能最优的效果，可进行进一步的调参。