# 上 海 交 通 大 学

## 机器学习

lab1

学生姓名：杨景凯

学　　号：520021910550

2022 年 10 月 27 日

# 目录

# 1 基本介绍

## 1.1 目的

本次 Lab 主要是实现两种方法的 SVM 模型并对比其准确度和时间。其中，两种方法分别是：使用 sklearn 包中的线性 SVM 模型和使用梯度下降法的手动实现的 SVM 模型。

## 1.2 环境

### 1.2.1 python 环境

本次使用的 python 版本为 3.7.9 64bit。

### 1.2.2 包环境

通过 pip freeze 将环境生成 requirements.txt 文件，如下所示：

- $numpy == 1.21.4$
- $scikit\_learn == 1.1.3$

### 1.2.3 机器环境

理论上，本代码在所有 64bit 的机器上均能正常运行。本次实验所用的环境为：

- CPU: Intel(R) Core(TM) i5-6300U CPU @ 2.40GHz 2.50 GHz
- RAM: 4GB 1867MHZ

# 2 SVM-sklearn

## 2.1 实现方法

通过初始化 SVM 线性模型，并输入训练数据，其能自动训练。将测试数据输入模型，得到准确度。准确度的计算方式是，模型计算出的预测值与实际值差的绝对值的平均。

## 2.2 参数

经过测试与选取，我选择了如下参数：

- $THRESHOLD = 1^{-3}$
- $MAX\_ITERATION = -1$

其中，选择 $THRESHOLD = 1^{-3}$ 是为了在保证时间较短情况下增加准确度，选择 $MAX\_ITERATION = -1$ 是为了避免其提早结束导致准确度骤降（若选择和梯度下降相同的最大迭代次数，即 8000 次，则其准确度只有 0.71，并抛出警告其提前结束）

## 2.3 结果

根据上述参数配置，得到结果如下：

- $train\_acc$ : 0.975
- $test\_acc$ : 0.91
- $Running\ time$ : $1.3125 Second$

同时可以画出准确度与时间随 $THRESHOLD$ 的变化图如下：

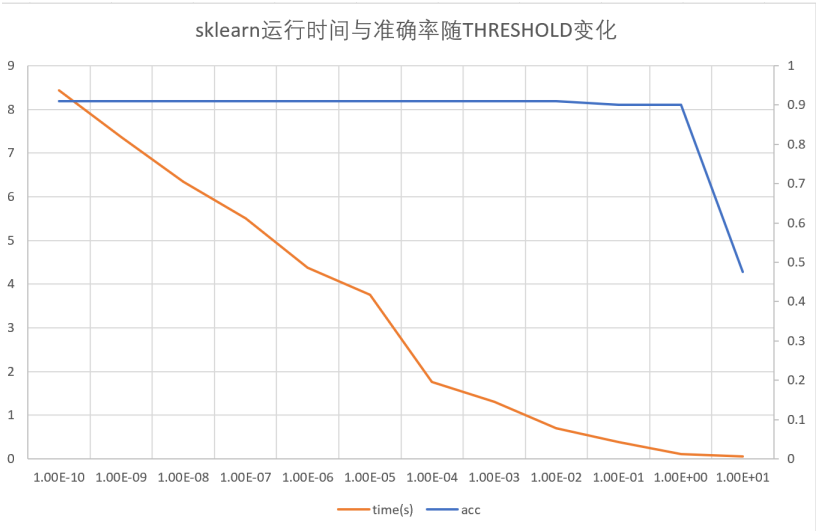


图 1: sklearn 准确度与时间随 THRESHOLD 变化图

可以发现，其在 THRESHOLD 约为 $1 \cdot 10^{-3}$ 时达到稳定，此时运行时间为 1.3125s，准确率为 0.91。最终结果为：

$$
\begin{aligned}
w = [ & -2.802051483528587, 0.06326673, 0.07667397, 0.18258601, 0.60309771, \\
& -0.28708404, -0.20567921, 0.4027636, -0.65930869, -0.84561177, \\
& -0.81467686, 0.52399726, -0.67723609, -0.20361242 - 0.44343857, \\
& -0.22124865, 0.50415385, -0.41556441, -0.12376121, 0.01310567, \\
& -0.51855263, -0.5626143, 0.44617828, 0.50301771, -0.07107305, \\
& -0.2538178, 0.99309579, 0.52277441, 0.00215304, 0.00112656]
\end{aligned}
$$

# 3 SVM-gradient decent

## 3.1 实现方法

几乎相同于课堂上展示的伪代码，但是使用 np 与矩阵来进行计算，加速计算速度也减少代码量。最终的损失函数为：

$$
L(w, w_0|D) = \begin{cases} \dfrac{\lambda}{2}||w||^2 & y^{(l)}(w^T x^{(l)} + w_0) \geq 1 \\ \dfrac{1}{N}\Sigma_{l=1}^{N} 1 - y^{(l)}(w^T x^{(l)} + w_0) + \dfrac{\lambda}{2}||w||^2 & otherwise \end{cases}
$$

其中，$\lambda$ 表示为下述 PENALTY。详细地，方法为：

1. 随机产生初始的 w 值。
2. 对于 $w_1 - w_N$，首先计算所有满足 $y^{(l)}(w^T x^{(l)} + w_0) \geq 1$ 对应的下标集合 $I$，对于每个下标，计算 $\Delta w_j = -\frac{1}{N} \cdot \Sigma_{i \in I} x_j^{(i)} \cdot y^{(i)} + PENALTY \cdot w_j$。对于 $w_0$，$\Delta w_0 = -\Sigma_{i \in I} y^{(i)}$。
3. 更新 $w_0 = w_0 - STEP\_SIZE \cdot \Delta w_0$，$w = w - STEP\_SIZE \cdot \Delta w$。

将测试数据输入模型，得到准确度。准确度的计算方式是，模型计算出的预测值与实际值差的绝对值的平均。同时由于初始随机性的选取，我将代码运行 5 次，准确度与时间计算取平均值。

### 3.2 参数

经过测试与选取，我选择了如下参数：

- $PENALTY = 0.001$
- $THRESHOLD = 1^{-9}$
- $STEP\_SIZE = 0.0003$
- $MAX\_ITERATION = 8000$

其中，选择 $PENALTY = 0.001$ 是为了降低 w 的平方和的占比，选择 $THRESHOLD = 1^{-9}$ 和 $MAX\_ITERATION = 8000$ 是为了在保证时间较短情况下增加准确度，选择 $STEP\_SIZE = 0.0003$ 是为了增加准确度，使得其缓慢迭代。

### 3.3 结果

根据上述参数配置，得到结果如下：

- $train\_acc$ : 0.9325
- $test\_acc$ : 0.92
- $Running\ time$ : $2.78125 Second$

同时可以画出当 $MAX\_ITERATION = 8000$ 时准确度与时间随 THRESHOLD 的变化图如下：

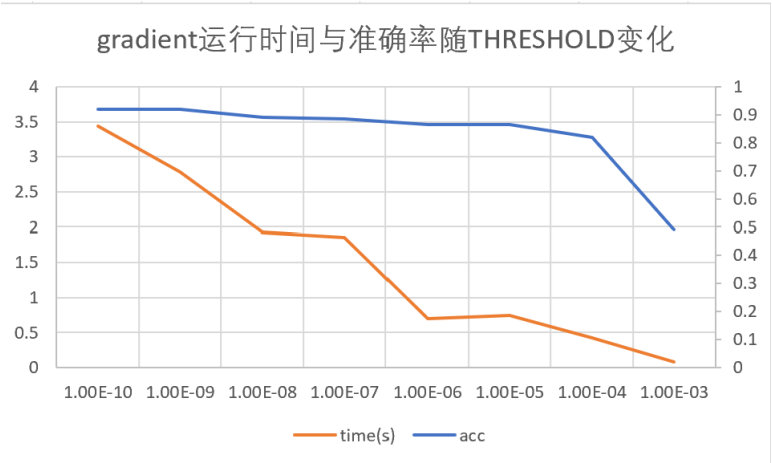


图 2: gradient 准确度与时间随 THRESHOLD 变化图

可以发现，其在 $THRESHOLD$ 约为 $1 \cdot 10^{-9}$ 时达到稳定。同时，设置为此值时，迭代次数差不多为 8000（当设置为 $1 \cdot 10^{-8}$ 时，迭代次数约为 4000）。同时，也可以画出准确度与时间随 $STEP\_SIZE$ 变化图如下：

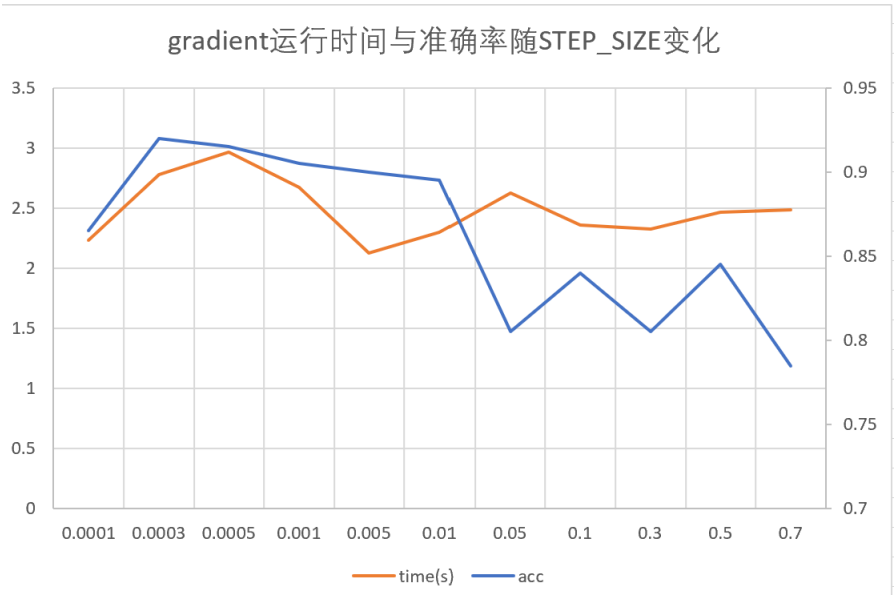


图 3: gradient 准确度与时间随 $STEP\_SIZE$ 变化图

可以发现，其基本保持不变，因此我选择最大值，即 $STEP\_SIZE =$

0.0003。画出准确度与时间随 $PENALTY$ 变化图如下：

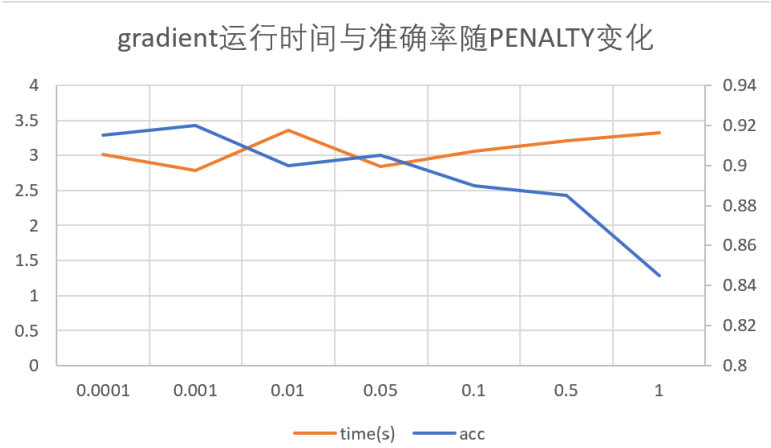


图 4: gradient 准确度与时间随 PENALTY 变化图

对于 PENALTY 来说，由于并未规定取多少，因此我取了可以使得运行时间最短同时准确率最高的 0.001。此时，运行时间为 2.78125s，准确率为 0.92。最终结果为：

$$
\begin{aligned}
w = [ & -0.61092617, -2.29493064e-01, -1.32799101e-02, -2.37979400e-01, 2.50455259e-01, \\
& -1.45104633e-01, 9.34174521e-02, 3.64139625e-02, -1.06806357e-01, -1.27525606e-01, \\
& -1.87934020e-01, 1.07382368e-01, -2.22253625e-01, -8.32127053e-02, -2.72055261e-01, \\
& 7.61677805e-02, -2.29022182e-02, -1.11535143e-01, 3.70389924e-02, -6.91222326e-03, \\
& 7.80859202e-03, 3.87350089e-02, -2.39034455e-04, -4.33990394e-03, 4.08749327e-02, \\
& -3.02514495e-02, 1.03504090e-02, 3.15397126e-02, -4.06747076e-03, -7.86600723e-04]
\end{aligned}
$$

## 3.4 变化

最终变化情况如图所示：

```
epoch: 200 th: [0.00020503] acc: 0.605 loss: [0.79000089]
epoch: 400 th: [0.00014925] acc: 0.685 loss: [0.63000172]
epoch: 600 th: [0.00013768] acc: 0.76 loss: [0.48000255]
epoch: 800 th: [0.00011675] acc: 0.84 loss: [0.32000343]
epoch: 1000 th: [0.00010389] acc: 0.86 loss: [0.28000432]
epoch: 1200 th: [8.35178086e-05] acc: 0.84 loss: [0.32000521]
epoch: 1400 th: [0.00010285] acc: 0.865 loss: [0.27000617]
epoch: 1600 th: [8.44416168e-05] acc: 0.84 loss: [0.32000719]
epoch: 1800 th: [5.26135391e-05] acc: 0.84 loss: [0.32000824]
epoch: 2000 th: [5.3547975e-05] acc: 0.84 loss: [0.32000932]
epoch: 2200 th: [2.83029279e-05] acc: 0.85 loss: [0.30001044]
epoch: 2400 th: [3.19883897e-05] acc: 0.855 loss: [0.29001158]
epoch: 2600 th: [3.2335216e-05] acc: 0.86 loss: [0.28001271]
epoch: 2800 th: [2.23243991e-05] acc: 0.86 loss: [0.28001388]
epoch: 3000 th: [5.10966629e-05] acc: 0.86 loss: [0.28001507]
epoch: 3200 th: [0.00010566] acc: 0.885 loss: [0.23001626]
epoch: 3400 th: [0.00010566] acc: 0.88 loss: [0.24001743]
epoch: 3600 th: [9.13975632e-05] acc: 0.87 loss: [0.26001866]
epoch: 3800 th: [9.15708336e-05] acc: 0.87 loss: [0.26001997]
epoch: 4000 th: [0.00010321] acc: 0.88 loss: [0.24002117]
epoch: 4200 th: [1.71150175e-05] acc: 0.865 loss: [0.27002237]
epoch: 4400 th: [6.00962289e-05] acc: 0.875 loss: [0.25002356]
epoch: 4600 th: [1.16460148e-07] acc: 0.885 loss: [0.23002458]
epoch: 4800 th: [3.37434671e-07] acc: 0.895 loss: [0.21002554]
epoch: 5000 th: [8.00578423e-05] acc: 0.88 loss: [0.24002653]
epoch: 5200 th: [8.28814577e-05] acc: 0.885 loss: [0.23002771]
epoch: 5400 th: [1.69477388e-05] acc: 0.89 loss: [0.22002884]
epoch: 5600 th: [8.4914497e-05] acc: 0.91 loss: [0.18002996]
epoch: 5800 th: [0.00015852] acc: 0.88 loss: [0.24003106]
epoch: 6000 th: [1.16953583e-07] acc: 0.905 loss: [0.19003208]
epoch: 6200 th: [2.2178255e-06] acc: 0.91 loss: [0.18003295]
epoch: 6400 th: [2.55073685e-06] acc: 0.91 loss: [0.18003377]
epoch: 6600 th: [2.55073683e-06] acc: 0.91 loss: [0.18003459]
epoch: 6800 th: [2.55804757e-06] acc: 0.915 loss: [0.17003541]
epoch: 7000 th: [2.55804754e-06] acc: 0.915 loss: [0.1700362]
epoch: 7200 th: [2.58249582e-06] acc: 0.92 loss: [0.16003701]
epoch: 7400 th: [2.24974696e-06] acc: 0.92 loss: [0.1600378]
epoch: 7600 th: [2.66252349e-06] acc: 0.925 loss: [0.15003857]
epoch: 7800 th: [6.7472654e-05] acc: 0.91 loss: [0.18003937]
epoch: 8000 th: [2.4340729e-06] acc: 0.92 loss: [0.1600404]
```

图 5: 变化情况

画图如下所示:

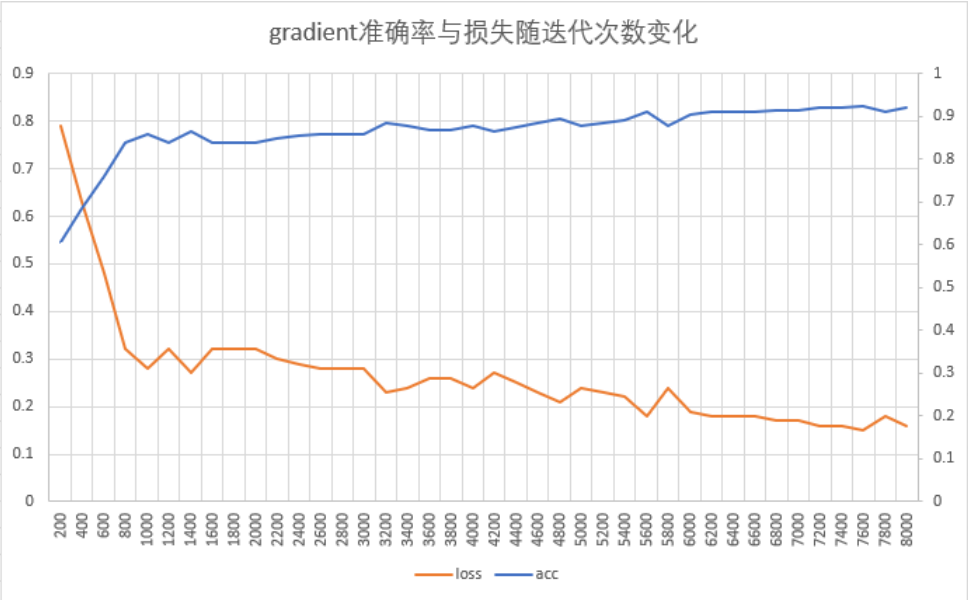


图 6: 变化情况

# 4 对比与分析

## 4.1 时间对比

从时间上来看，达到相同的准确率需要的时间是 sklearn 小于使用 gradient decent 的 SVM。我认为有两个原因：

1. 加速算法的使用。在计算中可以发现，sklearn 的迭代次数远远多于 8000 次，而仍然能够达到快速，是由于用到了很多加速算法。例如多线程并行计算等。

2. 底层的不同。在 sklearn 中，很多函数的编写是使用 C 来编写的，相比 python，C 能提供极大的速度加成。

## 4.2 准确度对比

从准确度上看，两者大体相似。在足够迭代后，均能达到 90% 以上的准确度，且再增加迭代次数不会发生明显变化。这说明参数已经收敛，几乎达到了最佳的准确度。

### 4.3 数据对比

观察发现，两者最终得出的 w 不太一致。这是由于不同方法导致的。支持向量选取的不同也会导致这样的结果。

### 4.4 结论

总之，手动编写的使用 gradient decent 方法的 SVM 也能达到与 sklearn 的 SVM 相似的效果，因此表现了代码的有效性。