**Lab 2 实验报告**

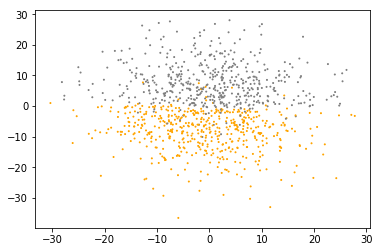
PB20000296 郑滕飞

**框架部分：**

**1、数据生成部分**

这部分已经由助教的代码实现了，此处解释一些性质，用于之后对结果的分析。

生成标错的方法是随机反转一部分标签，且pred的绝对值越低(也即越接近边界的)越容易被反转，因此，利用二维数据绘制散点图的结果类似于此：



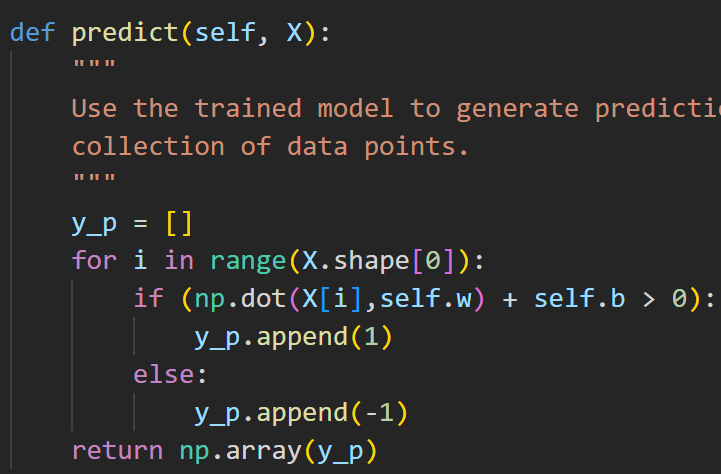
如果想要消除标错，只需要注释掉label[i] \*= -1一句，而提升标错只需要将函数中np.abs(pred\_n[i])的比重提升。

此外，由于点的随机生成是对每个维度取均值为0方差为10的高斯分布随机数，数据的量级是相近的，不需要对初始数据进行额外的归一化等处理。

**2、SVM类**

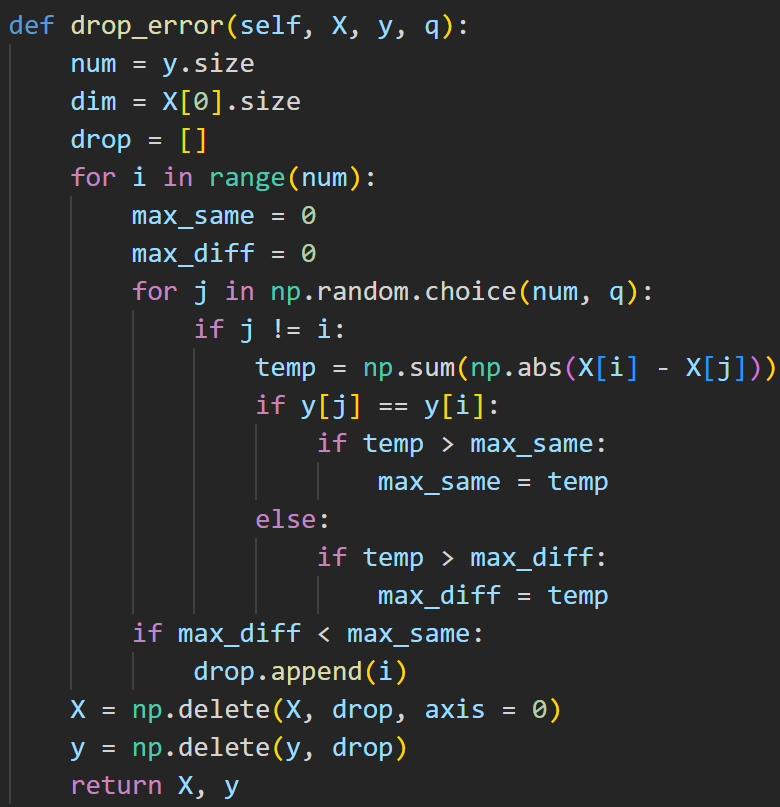
由于两个SVM中有很多共同部分，这里先构造了SVM类，再将SVM1与SVM2作为它的子类构造。

SVM类包含除了匹配之外的所有部分，即创建与预测。此外，为了减轻误差，我还设计了一个函数drop\_error。init直接根据维度建立w，并建立b即可，预测函数如下：



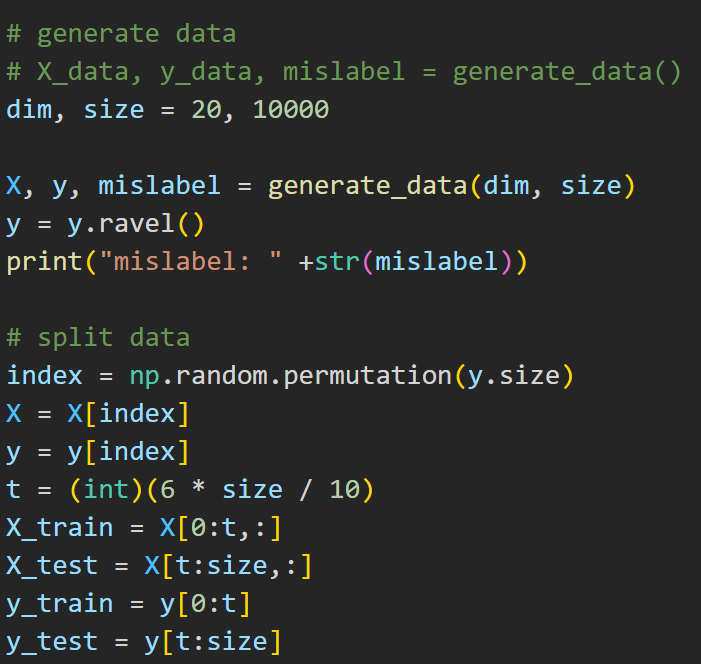
drop\_error函数的设计来源于一个构想：由于数据中一开始存在一些噪点，而SVM对误差的敏感程度较高，如果通过预处理能减轻噪音，就可以增加精度。

于是，我试图通过判断每个点到自己类与另一类的点的最小距离来确定噪点。如果一个点到自己类的最小距离小于到另一类的，即很有可能为噪点。由于样本规模问题，对每个样本随机选取若干个(实际操作时一般为100)进行这样的比较，然后确定是否舍去：

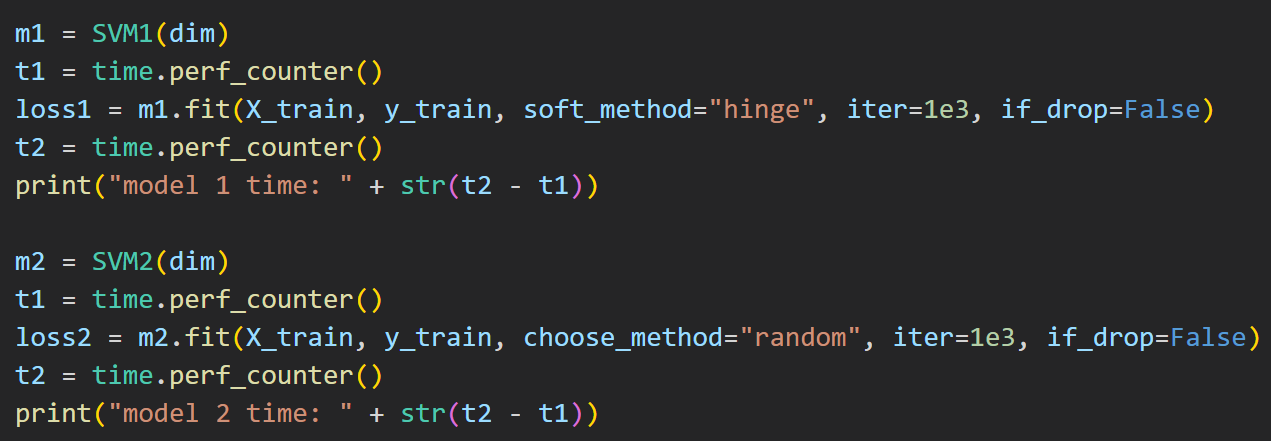


(此预处理方式的实际效果会在后方结果展示时说明)

**3、生成数据与训练**



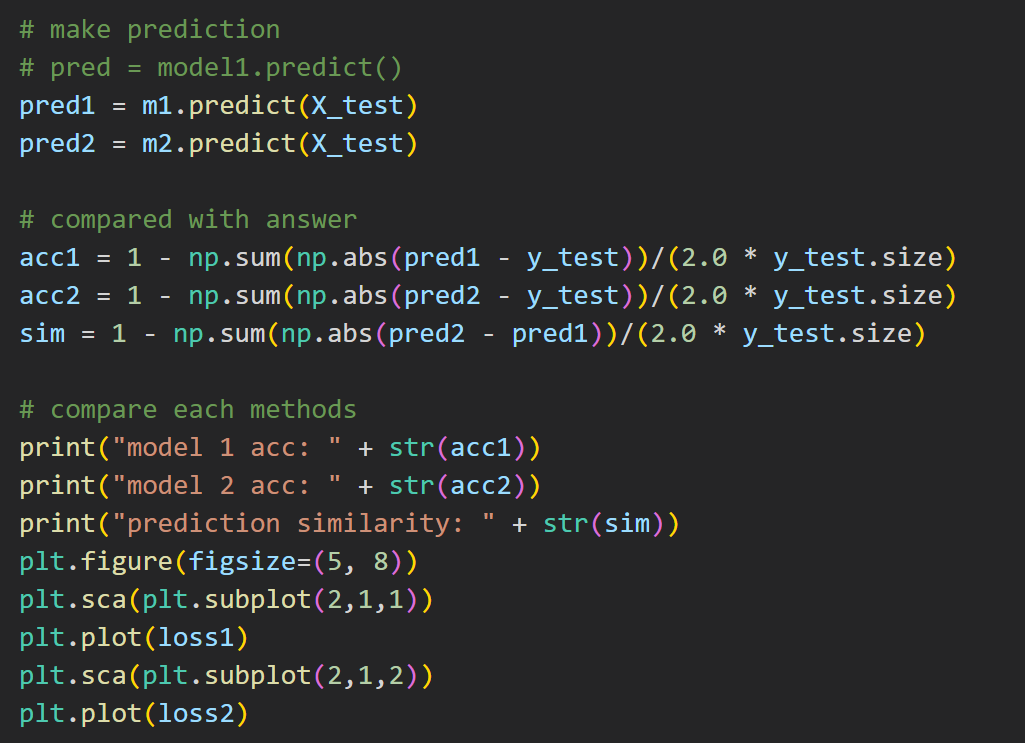
生成数据与分割数据与上一个实验中的类似，为随机打乱后按比例分割。由于实际对比时训练样本至少为1000，确定分类边界不需要过多样本，这里对数据与测试按照6:4进行留出。然后是训练部分：



这里利用time module进行计时，而fit函数的返回值为每次迭代后的损失，用来绘制曲线。

此部分最后会打印数据集的误标率与两种训练方式的时间。

**4、预测与对比**



这里将打印三个数据：两个模型的效果与它们预测的相似度。此外，利用matplotlib绘制两个模型训练时的损失曲线。

**两种SVM匹配算法：**

**1、方法选择**

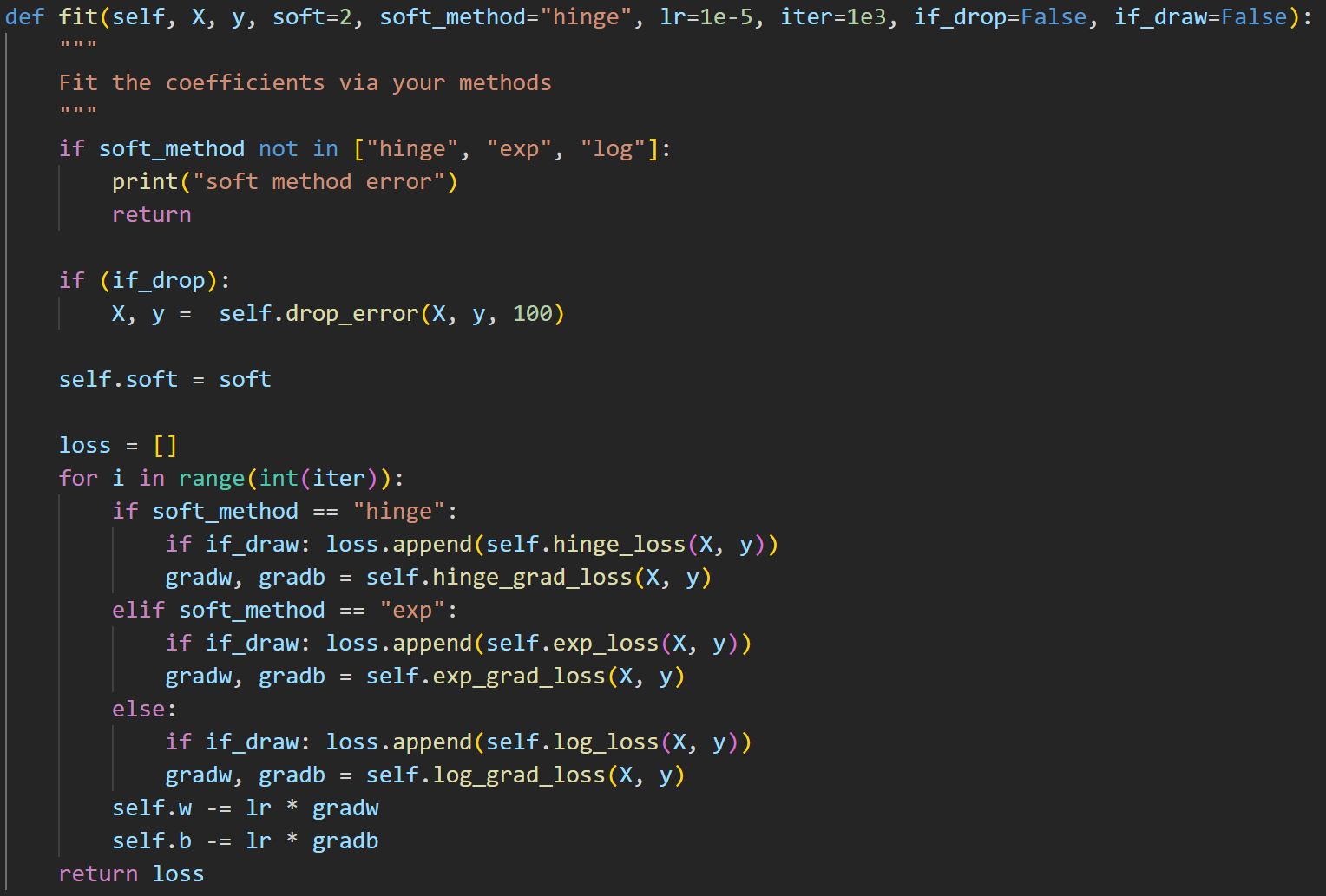
由于数据集中存在误标，为线性不可分数据集，想获得较好结果必须采用软间隔的SVM，且根据实验提示，无需使用核方法。

由于软间隔的条件，硬间隔所对应的不等式约束最优化问题被转化为了无约束最优化(且为二次)，于是牛顿迭代、梯度下降等应对无约束最优化的方法都可以采用。这里采取较好实现的梯度下降，并且对比Hinge、Exp、Log三种误差函数下的特性。

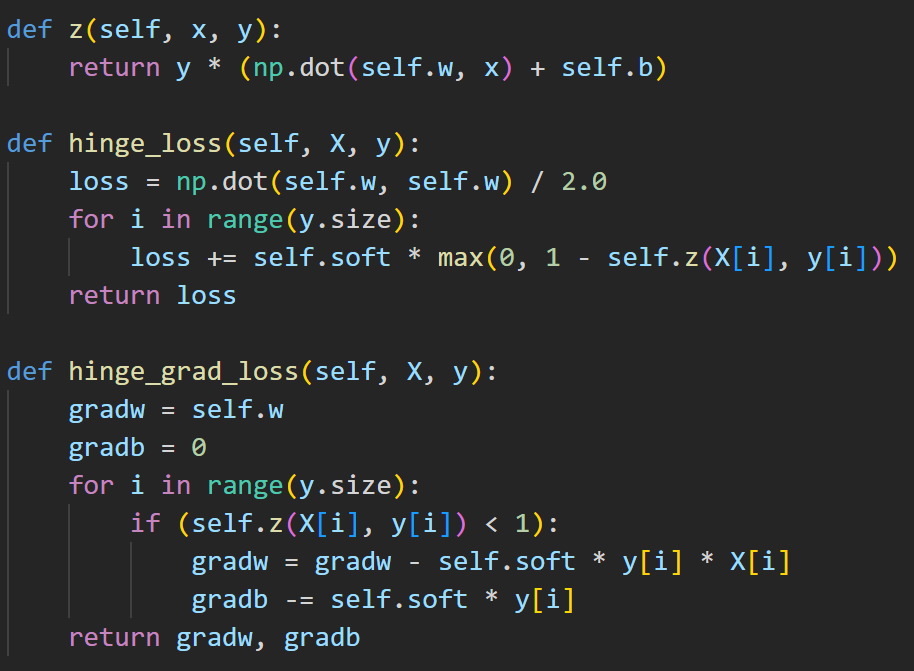
第二种算法则选取为SVM量身定制的SMO算法，并且对比不同的选择策略的结果。

**2、梯度下降**

此方法的主要匹配函数实现较为简单：



其中，soft代表软间隔的强度，soft\_method则是选取的损失函数，按照PPT，选择了Hinge损失、Exp损失和Log损失三种，分别构造了对应的计算损失与损失梯度的函数，如Hinge损失的计算：

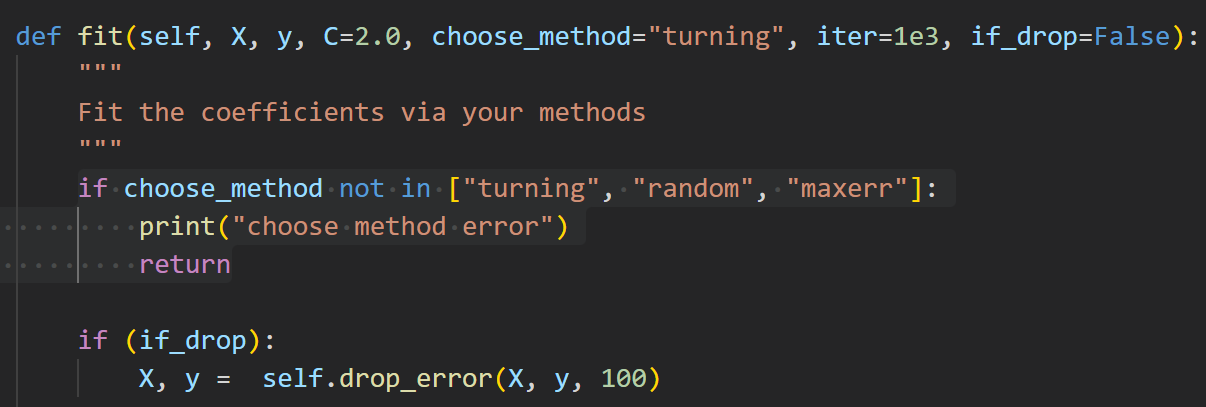


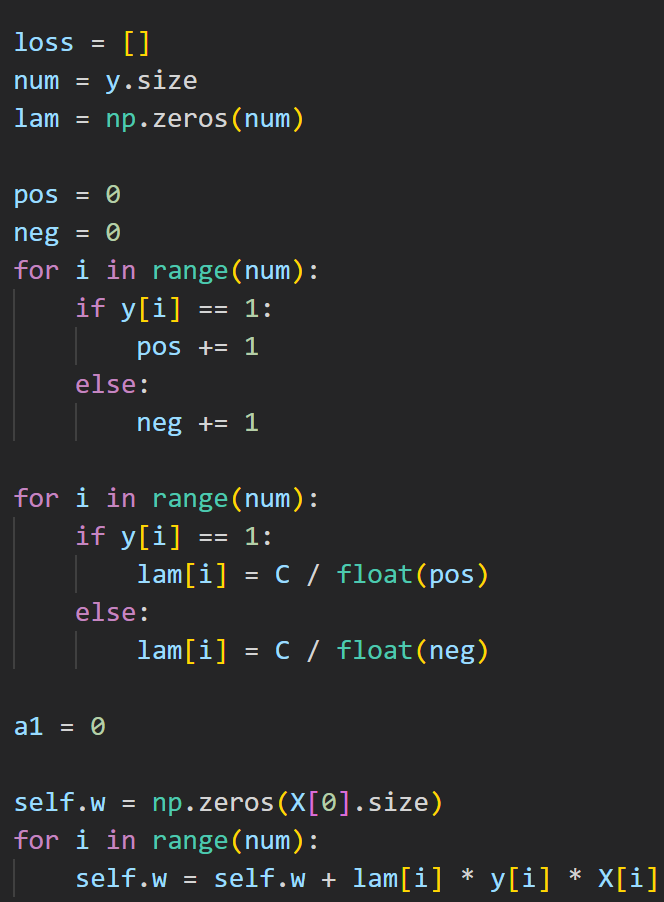
lr和iter分别是学习率与迭代次数，这里的损失梯度并没有对样本数量归一化，因此学习率的值需要设定至较低[Lab 1实验报告中已解释归一化效果与学习率设定相同]。

**3、SMO**

SMO算法的迭代分为三个部分：按照当前的向量更新解与预测、选取向量中两个要优化的分量、优化。根据理论推导，软间隔的SMO算法中，软间隔系数C (与梯度下降方法中的soft类似)会成为的上界，而下界为0。

初始化部分如下：

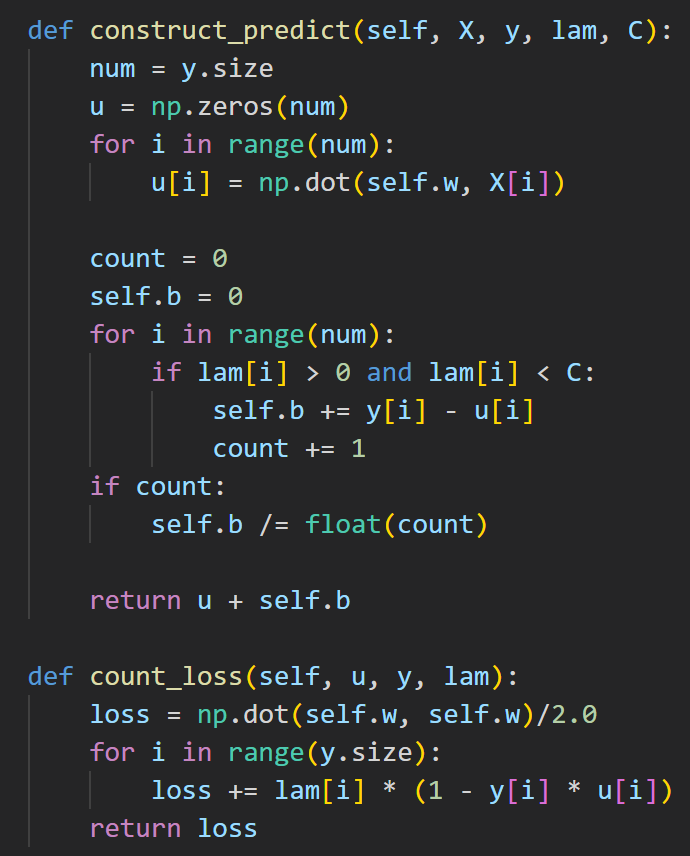




这里选取的初始化策略是全部赋值为非零值，这是由于lambda全为0的初始化策略会导致优化效率降低，之后会进行对比。参数choose method代表选取第一个分量的方法，分为最大偏差(老师PPT上)、轮流与随机三种方式。

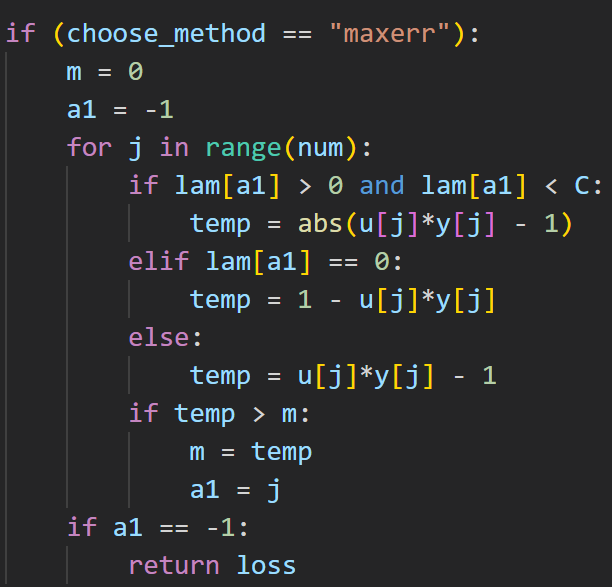
此外，这里先计算出了w的初值，这样每次更新时w的改变只需要计算更新的分量即可。

更新预测与计算损失采用了单独的函数：

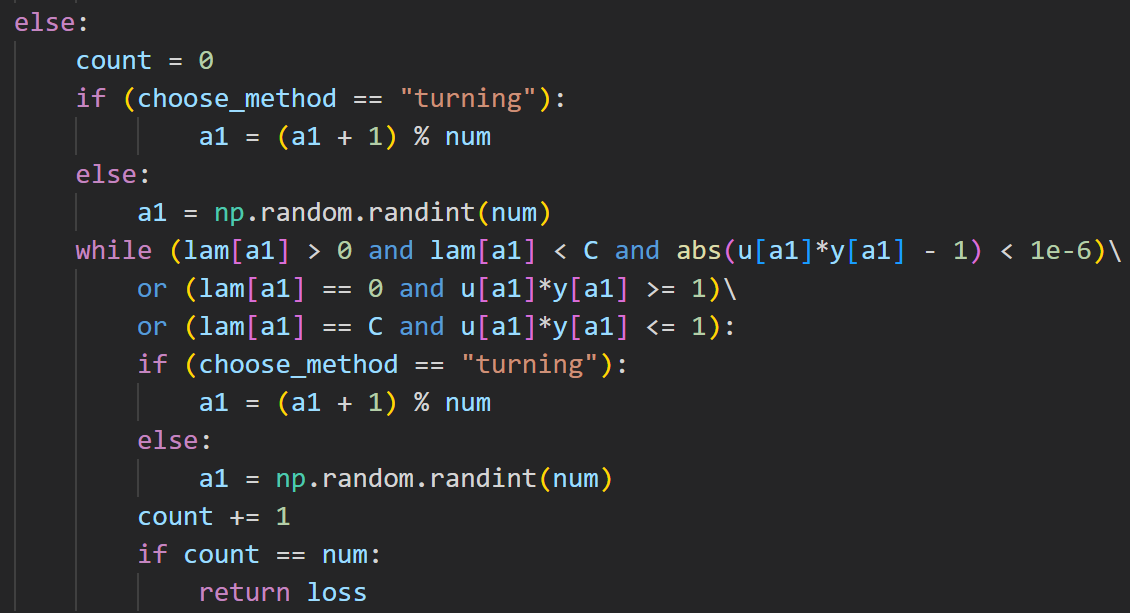


这里，先更新u[i]为不加上b的版本，再以此计算对应的b，避免了多次进行点乘运算。最后，将u[i]加上预测的b作为真实的u[i]返回。而对于损失函数，直接利用公式计算即可。值得注意的是，此处的损失函数由于SMO解决的是对偶问题，所以损失函数需要进行最大化，损失函数作图也是上升的。

选取分量采用了三种方式：最多违反KKT、轮流、随机。第一种方式根据lam向量的情况，按照u\*y-1与正确情况的差距进行选取：

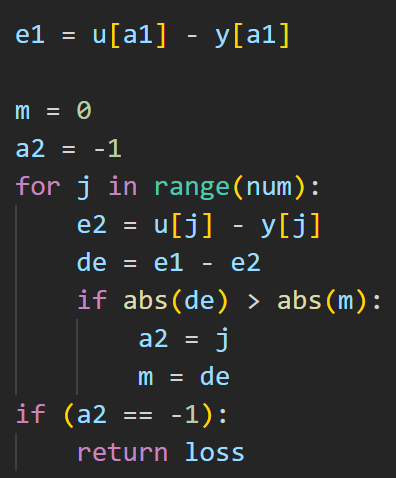


第二、三种都是每次改变选取对象，找到第一个违反的：



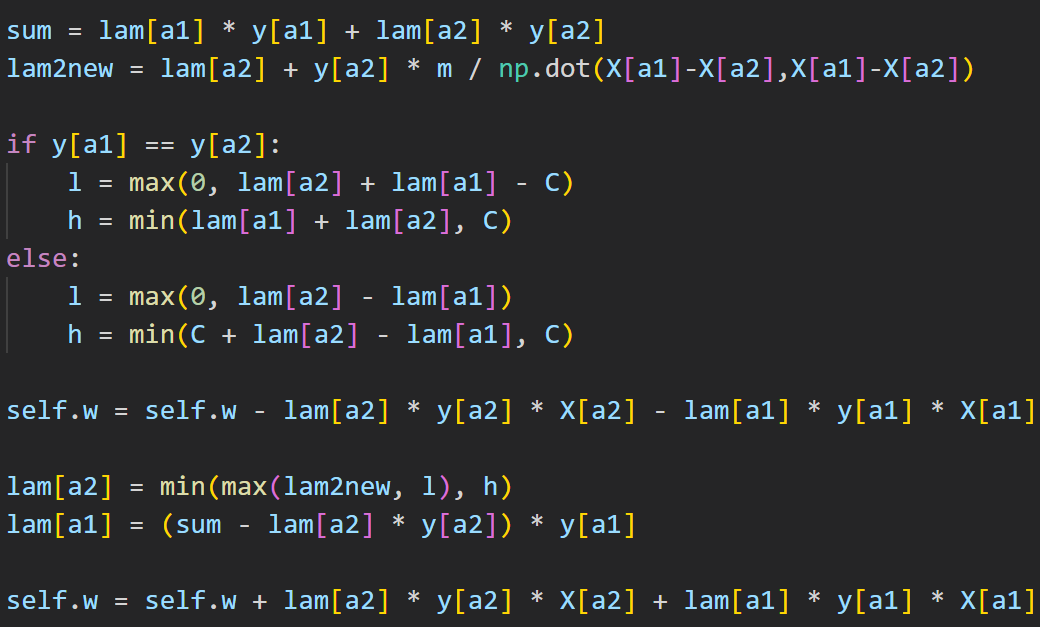
其中turning直接寻找了下一个，而random则进行随机选择。

选取完第一个后，第二个按照改变最多进行寻找：



无论何种选取策略，只要所有分量都满足要求，迭代即自动停下，也就是收敛。

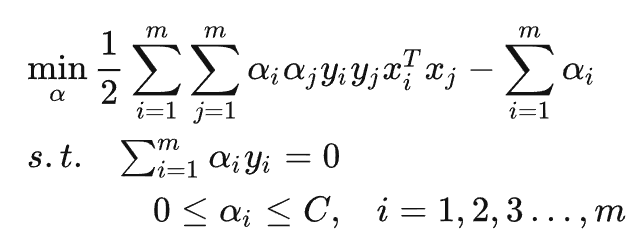
最后，按照查找的算法对分量进行更新：



此处，在更新lam向量的同时，也将w对应更新，减去之前的分量，加上更新后的分量。这样比起每次重新计算可以节省大量的时间。

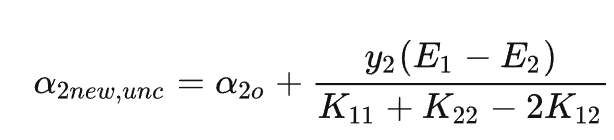
**\*SMO算法原理**[主要参考https://zhuanlan.zhihu.com/p/257866920]**：**

先将软间隔问题转化为其对偶问题：



而由于原问题的特殊性，最优解时应满足KKT条件，即为0到C时时对应的预测u[i]\*y[i]为1，当其为0时u[i]\*y[i]大于等于1，其为C时对应的u[i]\*y[i]小于等于1。

由此，SMO的思路为每次选取两个，看作单变量(由于两个之间存在约束)最优化问题进行更新。为使效果最好，原则上第一个采用违反最多的，第二个采用能导致更新量最大化的(更新量可以由二次函数极值得到具体公式，可发现与两个u[i]-y[i]的差距密切相关)。选取完后，利用计算出的结果：

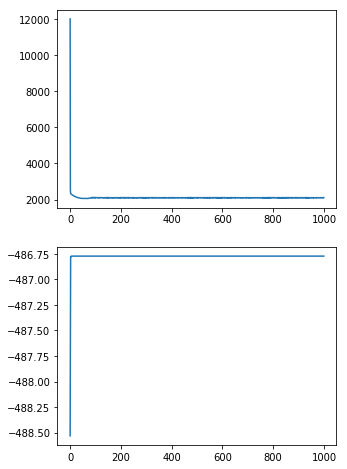


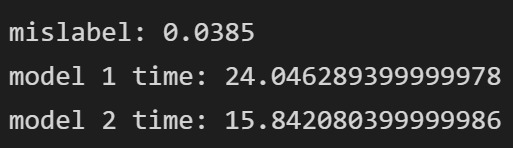
进行更新，但若越过边界需要用不等式条件加以截断，再进一步计算得到第一个分量的更新即可。

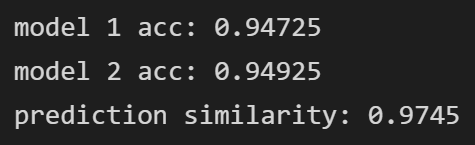
**展示与对比：**

**1、标准结果**

当维度为20，数据集为10000，按照6:4进行分割，默认参数的情况下结果如下：







这里的时间性能并不准确，因为计算出损失函数绘制曲线也会十分耗时。在之后对比时会给出更明确的结果。

[为此，在函数中添加了if\_draw参数，若为False则不计算损失]

**2、梯度下降相关对比**

通过损失曲线可以发现，迭代收敛的速度事实上比默认参数要快。改为100次后，发现仍能有类似的收敛结果。以下为迭代次数100、学习率1e-5时各个方法的对比[由于数据有随机性，每个为三次取平均][此处计算时间时均未绘制损失曲线]：

|  |  |  |  |
| --- | --- | --- | --- |
| **软间隔参数** | Hinge损失 | Exp损失 | Log损失 |
| 0.05 | 95.2% | 94.7% | 95.1% |
| 0.5 | 95.4% | 95.0% | 95.2% |
| 1 | 95.6% | 溢出 | 95.8% |
| 2 | 94.9% | 溢出 | 95.2% |
| 5 | 92.0% | 溢出 | 95.0% |
| **时间** | 1.06s | 4.39s | 4.47s |
| **损失曲线**  **[0.05，1000次]** |  |  |  |

观察损失曲线可以看出， Exp损失的收敛速度快于Hinge，Hinge又快于Log。而Log的准确性一般高于剩下两个。不过，考量速度与参数容忍因素，总体来看，当数据量较大时，Hinge损失的表现是更优的。

**3、SMO相关对比**

SMO自身的对比主要有两点：向量lam初值的影响与选取方式的影响。

同样以迭代次数100构造表格[这里三种方法都是在开始变化很大，之后变化很小，以100为次数构造表格仍然是合理的] [此处计算时间时仍然均未绘制损失曲线]：

|  |  |  |  |
| --- | --- | --- | --- |
| **软间隔参数** | 最大选取 | 轮流选取 | 随机选取 |
| 0.05 | 94.5% | 94.3% | 93.7% |
| 1 | 94.3% | 94.7% | 95.0% |
| 5 | 94.9% | 94.2% | 93.8% |
| **时间** | 2.05s | 1.44s | 1.41s |
| **损失曲线**  **[1，100次]** |  |  |  |

随机选取造成的结果比起其他两种略差，另两种则未看出明显差别。由于最大选取需要比较所有，总体时间是劣于剩下两种的。

不过，以上的比较都是针对全部非零的初值构造。若初值全部为0，情况会很不一样[下方参数为C=1，迭代次数300，此时轮流选取与随机选取基本收敛]：

|  |  |  |  |
| --- | --- | --- | --- |
| **选取方式** | 最大选取 | 轮流选取 | 随机选取 |
| **准确率** | 73.9% | 84.2% | 82.2% |
| **时间** | 6.03s | 4.13s | 4.10s |
| **损失曲线** |  |  |  |

增加迭代次数会发现，这样的初值会导致最大选取的第一个参数每次为同一个，无法找到好的收敛结果。。

由此看来，第一个选取违反KKT条件最大的参数这一原则，在合适的初值选取下，可以被轮流优化替代，且效果并无明显差别。

**4、两种算法对比**

从刚才的结果来看，在调节到合适的参数时，梯度下降的时间与SMO基本相近，且结果的准确率更高。不过，这并不是全部的情况。

在之前，标错率基本在0.03到0.04之间。当修改函数让此值升高时，结果会有变化[下方表格参数C与soft均为1，方法分别为Hinge与Turning，迭代100次]：

|  |  |  |
| --- | --- | --- |
| **标错率** | **梯度下降准确度** | **SMO准确度** |
| 0 | 99.2% | 97.7% |
| 0.038 | 95.1% | 94.4% |
| 0.076 | 91.7% | 91.2% |
| 0.107 | 88.4% | 88.2% |
| 0.151 | 84.2% | 84.3% |

此外，减小训练集会导致两者的准确度同步下降，但梯度下降的结果仍然普遍优于SMO。

另一个有趣的事实是超低迭代次数的情况：

|  |  |  |
| --- | --- | --- |
| **迭代次数、数据集大小** | **梯度下降准确度** | **SMO准确度** |
| 1、10000 | 94.5% | 94.6% |
| 10、10000 | 95.5% | 95.0% |
| 1、100000 | 95.7% | 95.7% |
| 10、100000 | 96.4% | 95.8% |

无论是梯度下降还是SMO，似乎在低迭代次数时都表现出了不错的结果。

最后，回到一开始试图优化结果的drop error方法。当在数据集中去掉了部分样本后，无论是梯度下降还是SMO的准确率都并未受到什么影响。不过，这样选取后的时间能提升约20%。

就目前的对比而言，SMO算法的时间性能好于梯度下降，而梯度下降方法在选取合适的参数后可以做到比SMO更优。不过，SMO对参数变化的稳定性要强不少。尤其是其不需要控制学习率进行调整。

**总结：**

在开始写这个实验之前，结果似乎是清楚的。然而写着写着，问题变得越来越多了——事情是从发现一次迭代也能有很好的结果开始不对劲起来的。总体来说，大概是因为生成的数据过于标准了，两种方法的收敛速度都非常快，也都能在合适情况下达到95%左右的正确率(考虑到错标率，实际正确率可能有98%)。此外，在和人交流的过程中，深刻感受到了每个人都有自己的SMO，大家的方法细微处差别挺大，结果也有较大的差别。