没有算法的程序优化

魏梓轩

2019 年 9 月 1 日

加快程序的运行速度，利用高效的算法提高产生结果的效率是一方面，编写出几乎没有缺陷的优秀代码是另一方面。而优秀的代码，往往是算法应用和程序实现过程中最重要的一环。拿最简单的二分查找算法来说，不同实现方法（真实坐标、偏移量）下所使用退出条件的符号也有所不同（$\leq$ or $<$）。（middle）$+1$ 或者 -1 操作，也尤其讲究。稍有不慎，就会陷入死循环中。简单的代码修改（如 $+1$，-1）有可能使得我们所写的程序成功运行，而有时候也会带来**显著的性能提升**。本文聚焦于自己曾经吐槽，尚且可以进一步优化和尝试改过的代码:)

1 Man from Istria

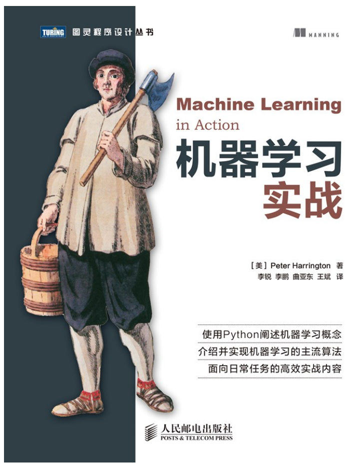

图 1: 《机器学习实战》封面

“Man from Istria”封面插画来自于几乎人手一本的著名神书——《机器学习实战》。封面上“左擎锄，右牵桶”的这位老大爷显然不像是一位程序员。当然，老大爷也有自己的名字，我们可以尊称他 Hacquet （1739−1815）。他有着令人尊敬的职业——内科医生及科学家，曾花费数年研究不同地区的植物、地质和人种。这种精神值得大家去学习，因此这本神书以此封面所带来的寓意也十分显然了（当然，这是我语文保持多年高分的秘诀，妄加正义的猜测 =，=）。作为一本实战类书籍，书中的代码给大家提供了一些很

好的实现例子。结合着算法的数学描述，我们可以很快熟悉由理论到工程的实现过程。然而，书中所提供的并非都是**深入工程实践**的例子，有些只是为了偷懒，而去简单的调用了一些库函数，如 `numpy.sum`，`numpy.mean` 等等。

如果你手头恰好有这本书，可以翻开 164 页（第 9 章，树回归）。在回归树切分函数（代码清单 9-2）中，计算数据子集切分误差的函数如下：

```
def regErr(dataSet):
    return np.var(dataSet[:, -1]) * np.shape(dataSet)[0]
```

在选取最佳切分函数 `chooseBestSplit` 中，计算某切分点当前误差的方法为：

```
# errType = regErr
newS = errType(mat0) + errType(mat1)
```

在连续的实数空间中，我们选择切分点的思路一般是：1）排序；2）依次选取实数值作为切分点把数据分为两个子集。一般来说，位于实数空间的真实数据，很难会出现几个连续的相同值。因此，切分操作对于排序好的数据也算是"雨露均沾"（也有"独得皇上恩宠"的方法，参考 XGboost）。上述代码带来的问题在于**每次切分成两个子集 `mat0`、`mat1` 后，都需要 从头开始 计算切分误差**。什么意思呢？我们选择一个相对好推导的"切分均值"来说明，以下是一段比较懒的代码：

```
def regMean(dataSet):
    return np.mean(dataSet[:, -1])


mat0, mat1 = binSplitDataset(dataSet, featIndex, splitVal)
newM = regMean(mat0) + regMean(mat1)
```

这段代码的问题同样在于每次切分后，都需要**从头开始**计算均值。而这两个子集所发生的数据改变，仅仅是把 `mat1` 中的极值移动到 `mat0` 中。我们用 `array` 表示整个数据集合，上述代码可以描述成以下数学模型：

$$\frac{\sum_{l=1}^{t} array_l}{t} + \frac{\sum_{r=t+1}^{N} array_r}{N-t}$$

当我们把 $array_{t+1}$ 的值从右子集移动到左子集，公式变成

$$\frac{\sum_{l=1}^{t+1} array_l}{t+1} + \frac{\sum_{r=t+2}^{N} array_r}{N-t-1}$$

在这个过程中我们是否可以利用上一步的状态进行计算呢？那么，作进一步推导，

$$\begin{aligned}
& \frac{\sum_{l=1}^{t+1} array_l}{t+1} + \frac{\sum_{r=t+2}^{N} array_r}{N-t-1} \\
= & \frac{\sum_{l=1}^{t} array_l + array_{t+1}}{t+1} + \frac{\sum_{r=t+1}^{N} array_r - array_{t+1}}{N-t-1} \\
= & \left(\frac{\sum_{l=1}^{t+1} array_l}{t} * \frac{t}{t+1} + \frac{array_{t+1}}{t+1}\right) + \left(\frac{\sum_{r=t+1}^{N} array_r}{N-t} * \frac{N-t}{N-t-1} - \frac{array_{t+1}}{N-t-1}\right)
\end{aligned}$$

利用上一步算出的均值状态替换上式第三行中的 $\sum$ 表达式，再对比上式第一行，可以发现经过进一步推导式子中的加法运算 明显少了很多。这种优化在面对大型向量的时候尤为有用，而程序中需要添加的特性仅仅是对上一步状态进行记录。学过《系统辨识》这门课的同学，可以再回顾一下"递归最小二乘法"的精髓，我觉得就是记录状态。一个 Batch 进来，重新计算，太难了！

回过头来，对于"神书"所述 `np.var`（方差）的计算方式，有没有递归或者记录状态的优化方式呢？显然是有的，但是这个推导有点烦，就不放了。代码可能如下：

```
# 先算一个初始状态，minEventsLength为最小的区间长度
mat0, mat1 = binSplitDataSet(dataSet, minEventsLength)
lNum = minEventsLength
lAvg = np.mean(mat0[:, -1])
lSE = np.sum((mat0[:, -1] - lAvg)**2)
rNum = m - minEventsLength
rAvg = np.mean(mat1[:, -1])
rSE = np.sum((mat1[:, -1] - rAvg)**2)
# 迭代过程，灵魂都在这里
for idx in range(minEventsLength + 1, m - minEventsLength):
  # left re-calculation
  lNum += 1
  delta = dataSet[idx, -1] - lAvg
  lAvg += delta/lNum
  newDelta = dataSet[idx, -1] - lAvg
  lSE += delta*newDelta

  # right re-calculation
  rNum -= 1
  delta = dataSet[idx, -1] - rAvg
  rAvg -= delta/rNum
  newDelta = dataSet[idx, -1] - rAvg
  rSE -= delta*newDelta

  newS = (lSE + rSE)/m
```

当然，你会觉得上面的代码阅读性差、Code Style 略丑。一方面是我的原因，另一方面是高性能的代码很难有好看的，真的:)

2 可能不靠谱的谱聚类

谱聚类是靠谱的，只不过我们很有可能写出不那么靠谱的代码。注意以下公式，来自于全连接法计算邻接矩阵 $\mathbf{W}$：

$$\omega_{ij} = \exp(-\frac{||x_i - x_j||_2^2}{2\sigma^2})$$

我们主要关注 $||x_i - x_j||_2^2$ 的计算。有一部分同学看到这个公式就着手去做了，写出了下面的代码：

```
for row_i in feature.shape[0]:
    for row_j in feature.shape[1]:
        diff = feature[row_i, :] - feature[row_j, :]
       W[i, j] = diff.dot(diff.T)   # 内积可能报错，但很好debug，
                                    # 至少保证ndim==2
```

我认为这段代码还可以这么写：

```
square_sum = (feature * feature).sum(axis=1).reshape((-1, 1))
W = square_sum + square_sum.T - 2*feature.dot(feature.T)
```

用到了 ndarray 的 broadcast 机制。

3 总结

假装有总结:-(