SVM 的 SMO & DCD 学习与实现

汪加林

2016 年 12 月 8 日

1 Introduction

SVM 在深度学习流行以前，应该是相当受人关注的一种方法。这两年，用过它去解决问题，也看过相关的推导。但一直碍于坊间谣言，推导都这么难，更别说实现了。Talk is Cheap, Show me the Code。这次潜心学习，实现，深深体会到这句话是多么的正确。这次的整个学习过程也让自己对 SVM 的理解更上一层楼了，特此记录下来希望给其他人一些帮助。

关于理论等，我不想提及太多，网上流传甚广的 July 的博客和李航老师的 < 统计学习方法 > 这本书讲的都比较清楚。

下面是整个文档的组织内容。

- 代码说明
- 优化加速思路
- 实现细节
- SMO 一些理论上的讨论
- LinearSVM 的大规模坐标下降方法实现
- 实验
- 类的设计

2 代码说明

- SimplifiedSMO: 最简化版本的 SMO 算法实现

- SimplifiedSMOWithKM：在 SimplifiedSMO 版本的实现上加入了缓存核矩阵
- StandardSMO : 标准的 SMO 算法实现
- SVMCoorDescent : 针对大规模问题的 LinearSVM 的 Dual Coordinate Descent 算法实现。

3 优化加速思路

原始版本 v__1 : 本次实现主要是参照**支持向量机通俗导论 (理解 SVM 的三层境界)，统计学习方法中的支持向量机章节**，实现的第一个非常简化的 SMO 版本。

- 选取第一个优化变量 ($\alpha 2$) 时采用的循环遍历整个数据样本点的方法，选取第二个样本点时简单的采用使 $|E_2 - E_1|$ 最大的标准。
- 未将内积结果缓存至核矩阵
- 整个优化的迭代次数由参数 $maxEpoch$ 控制的。

经过我后面的学习发现存在以下问题，但依然是一个不错的 Demo。

- 选取第一个优化变量时的策略太简单
- 理论上的整个优化的控制应该由实际优化过程中是否有优化过程决定
- 每一次优化时的各种特殊情况未考虑 (例如 $L = H, \eta = 0$ 等)

缓存核矩阵 v__2 ：第二个版本，在原始版本的基础上加入了缓存核矩阵

- 将内积结果缓存至核矩阵

SMO 论文版实现 : 本次实现是参照 [2] 中的论文伪码实现的，这篇论文被引几千次，大家一定要看看。

- 未采用缓存核矩阵
- 选取两个优化变量时的策略更全面、复杂
- 在进行参数更新时，考虑的情况更全面, 例如:$\eta = 0$ 等;

未实现其他思路 : Shrinking 等, 可以参考 [3]。

4 一些实现细节

4.1 缓存核矩阵

缓存核矩阵实现的方式是采用 $HashMap < String, Double >$ 的方式，$Key = id1 + "S" + id2$，采取将两点的索引唯一映射成一个键值。

5 SMO 一些理论上的讨论

5.1 为什么采用拉格朗日对偶

原始问题：

$$\begin{cases} min\ \frac{1}{2}||w||^2 \\ s.t\ \ y_1(wx_i + b) - 1 \geq 0, i = 1, 2, ..., N \end{cases} \tag{1}$$

对偶问题：

$$\begin{cases} min\ \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j - \sum_{i}^{N}\alpha_i \\ s.t\ \ 0 \leq \alpha_i \leq C \\ \quad \alpha_i \geq 0, i = 1, 2, ..., N \end{cases} \tag{2}$$

那么为什么要将原始问题转变为对偶问题呢？首先两个问题是等价的这一点是肯定的，原因其他地方有讲。对于原始问题，要求解 w，w 的维度为数据维度。而对偶问题要求解 α 的维度为数据样本点个数 N。这样有两个优点：

- 对偶问题更容易求解, 分解为一个个独立的拉格朗日乘子求解
- 易于引入核函数，推广到非线性分类问题

5.2 为什么使用核函数

使用核函数的最大的好处就是可以非常方便的计算高维特征空间的点的内积。大家在学习核函数这一块的时候，可能会经常看到这么一句不那么令人理解的话。“使用高斯核函数可以原始数据的维度映射到无穷多维”。

$$k(x_1, x_2) = exp(-\frac{||x_1 - x_2||^2}{2\sigma^2})$$

首先来一个将维度映射到有限高维度的例子。两个向量 $x_1 = (\eta_1, \eta_2)$ 和 $x_2 = (\sigma_1, \sigma_2)$。多项式核函数 $k(x, z) = (x * z + 1)^p$, 考虑 $p = 2$。那么得到

$$k(x_1, x_2) = 2\eta_1\sigma_1 + \eta_1^2\sigma_1^2 + \eta_1\sigma_1 + 2\eta_2\sigma_2 + \eta_2^2\sigma_2^2 + 2\eta_1\sigma_1\eta_2\sigma_2 + 1$$

如果采取函数先映射再内积的话可以使用以下映射函数

$$\psi(x_1) = (\sqrt(2)\eta_1, \eta_1^2, \sqrt(2)\eta_2, \eta_2^2, \sqrt(2)\eta_1\eta_2, 1)$$

再内积可以得到

$$< \psi(x_1), \psi(x_2) >= k(x_1, x_2)$$

可以发现，采用先映射在内积的方法是非常麻烦、费时的，在下面无穷维空间甚至会出现无法计算的情况。

再来看看 e^x 的泰勒展开式：

$$e^x = 1 + x + \frac{x^2}{2!} + ... + \frac{x^n}{n!} + ...$$

$$k(x_1, x_2) = 1 + (-\frac{||x_1 - x_2||^2}{2\sigma^2}) + \frac{(-\frac{||x_1-x_2||^2}{2\sigma^2})^2}{2!} + ... + \frac{(-\frac{||x_1-x_2||^2}{2\sigma^2})^n}{n!} + ...$$

展开上式，可以得到 $x^0, x^1, ..., x^n, ...$，这些就代表在无穷维空间的特征。然而使用一个映射函数 $\psi(x)$ 将 x 映射得到无穷维的上述特征是非常不明智的做法。

因此，核函数有下列好处：

- 能够将原始数据的维度映射到高维空间甚至无穷维空间
- 可以直接计算，而不用先映射。

6 LinearSVM 的加速实现

6.1 Coordinate Descent Method For SVM [1]

相比较于 SMO 每次只优化一个变量来看，Coordinate Descent 每次只优化一个变量。

6.2 Coordinate descent algorithm 推导

$$\begin{cases} min\ \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j - \sum_i^N \alpha_i \\ s.t\ 0 \le \alpha_i \le C \end{cases} \tag{3}$$

可能大家会疑惑和标准的 SVM 相比少了一个约束 $\sum_{i=1}^{N}\alpha_i y_i = 0$。将 $w^{new} = [w^{old}, b], x^{new} = [x^{old}, 1]$，那么

$$w^{old}x^{old} + b = w^{new}x^{new}$$

Algorithm 1 Coordinate descent algorithm with randomly selecting one instance at a time

Given α and the corresponding $w = \sum_i y_i \alpha_i x_i$

while α is not optimal **do**

Randomly choose $i \in \{1, ..., l\}$

$\hat{\alpha_i} = \alpha_i$

calculate the Gradient

$$G = y_i w^T x_i - 1$$

calculate Projection Gradient PG.

$$PG = \begin{cases} min(G, 0)\ if\ \alpha_i = 0, \\ max(G, 0)\ if\ \alpha_i = C, \\ G\ if\ 0 < \alpha_i < C \end{cases}$$

if $|PG| \neq 0$ **then**

$\alpha_i = min(max(\alpha_i - G/x_i^2), U)$

$w = w + (\alpha_i - \hat{\alpha_i}) y_i x_i$

end if

end while

，代入后，就可以消掉这个约束了。

$$f(x) = \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j y_i y_j x_i x_j$$

假设选定 α_1，优化 $f(x)$

$$f(\alpha_1) = \frac{1}{2}x_1^2\alpha_1^2 + (y_1x_1\sum_{i=2}^{N}\alpha_i y_i x_i) - \alpha_i + constant$$

$$\nabla_{\alpha_1} f(x) = x_1^2\alpha_1 + (y_1x_1\sum_{i=2}^{N}\alpha_i y_i x_i - 1)$$

令

$$\nabla_{\alpha_1} f(x) = 0$$

得到：

$$\begin{aligned}\alpha_1^{new} &= \frac{1 - (y_1x_1\sum_{i=2}^{N}\alpha_i y_i x_i)}{x_1^2} \\ &= \frac{(1 - y_1x_1\sum_{i=1}^{N}\alpha_i y_i x_i) + \alpha_1^{old}x_1^2}{x_1^2} \\ &= a_1^{old} - \frac{y_1w^Tx_1 - 1}{x_1^2}\end{aligned}$$

同时要考虑到 $0 \le \alpha \le C$ 便得到如上算法所示的 α_i。

w 的更新:

$$\begin{aligned}w^{new} &= \sum_{i=2}^{N}\alpha_i y_i x_i + \alpha_1^{new}y_1x_1 \\ &= \sum_{i=2}^{N}\alpha_i y_i x_i + \alpha_1^{old}y_1x_1 + (\alpha_1^{new} - \alpha_1^{old})y_1x_1 \\ &= w + (\alpha_1^{new} - \alpha_1^{old})y_1x_1\end{aligned}$$

7 实验

下面是我用 heart_scale 数据跑的一些实验结果。下面是共同的设置：

$$C = 1\ gaussSigma = 0.01, \epsilon = 0.001, maxEpoch = 5000$$

代码	核函数	最大迭代次数	准确率	运行时间
SimplifiedSMO	Linear	5000	0.746	7746
SimplifiedSMO	Linear	5000	0.746	1418979
SimplifiedSMOWithKM	Linear	5000	0.746	75588
SimplifiedSMOWithKM	Gauss	5000	1.0	170085
StandardSMO	Linear	5000	0.546	4548
DCD	非核函数	5000	0.804	41

Table 1: 实验比较

从上面的实验结果不难发现，非线性核确实有更好的分类效果。而增加核缓存核矩阵后确实能大幅加速，而之所以 $SimplifiedSMO$ 比 $SimplifiedWithKM$ 更快，是因为数据维度太低，普通的内积计算很快，比去查询 $HashSet$ 更快。另外值得一提的是，DCD(i.e. Dual Coordinate Descent) 简直太赞了！速度相差百倍之多啊！

A SimplifiedSMO 类的设计

SVMModel 表示 SVM 模型决策函数的一些参数 α, b，数据集 (x_i, y_i)，核函数 K，

$$f(x) = sign(\sum_{i=1}^{N} \alpha_i y_i K(x, x_i) + b)$$

- double alpha[ ] : α 参数数组
- double b : 超平面截距
- double x[ ][ ] : 数据集 x
- double C ： 惩罚系数
- int y[ ] : 数据集类别 y
- SVMModel(double[ ] xi, double[ ] yi, double C, KernelFunction K): 构造函数
- initParameter() ： 初始化变量 α 为 0
- double calculateGx(double[ ] x): 计算 $g(x)$

$$g(x) = \sum_{i=1}^{N} \alpha_i y_i K(x, x_i)$$

- int predictClass(double[ ] x): 根据 $f(x)$ 预测变量取值为 x 时的类别，返回 1 代表正类，-1 代表负类

SVMSMO 用 SMO 序列最小最优化算法训练模型的类，即计算 α, b 等参数。

- public double[ ] E : 存储 E，$E = g(x) - y$
- public int[ ] indexOpt = new int[2] : 优化的参数的索引
- KernelFunction K : 核函数类
- SVMModel model ： SVM 模型的参数
- int maxEpoch ： 最大循环次数
- public boolean[ ] selectMask ： 变量是否已被更新过
- double epsilon : 优化计算过程中的精度误差
- SVMSMO(double[ ][ ] x, int[ ] y, double C, KernelFunction.kernel k, double epsilon, int noChangeEcho): 核函数为线性核的构造函数
- public SVMSMO(double[ ][ ] x, int[ ] y, double C, KernelFunction.kernel k, double parameter, double epsilon, int noChangeEcho) ： 核函数为非线性核的构造函数，parameter 为控制核函数的参数
- public void smoTrain(): 训练函数
- public boolean sumToZeroCondi(): 参数 α 是否下面等式

$$\sum_{i=1}^{N} \alpha_i y_i = 0$$

- public boolean boundCondi(): 参数 α 是否满足 $0 <= \alpha_i <= C, i = 1, 2, ..., N$
- chooseOptAlpha(): 选择需要优化的两个变量
- chooseFirstAlpha(): 选择第一个需要的变量
- chooseSecondAlpha(int j) : 选择第二个需要优化的变量, j 是第一个优化变量的索引, 选取的标准是使 $E_2 - E_1$ 最大, 即要使 α_1 变化最大。

$$\alpha_2^{new,unc} = \alpha_2^{old} + \frac{y_2(E_1 - E_2)}{2}$$

- int getMaxIndex(double x[ ]): 返回数组 $x[\ ]$ 中最大元素的索引
- void updateE() : 计算 E, 根据 $g(x) = \sum_{i=1}^{N} \alpha_i y_i K(x_i, x) + b$,

$$E_i = g(x_i) - y_i, i = 1, 2$$

- void updateParameter() ： 更新变量 α 和 b
- double getL(): 计算 L
- double getH(): 计算 H
- public void calculateB(double alpha1, double alpha1Old, double alpha2, double alpha2Old) ： 计算 b
- int violateKKT(int i): i 是样本点索引, 判断训练样本点是否违反 KKT 条件

KernelFunction ： 核函数的实现

- private kernel flag ： 指示计算内积的核函数类别
- private double gaussSigma ： 高斯核函数的参数
- private double polyDim ： 多项式核函数的参数
- KernelFunction(kernel k) ： 线性核构造函数
- KernelFunction(kernel k, double parameter) ： 多项式核、高斯核函数构造函数
- void setKernel(kernel k): 选择核函数的类别
- double calculateK(double[ ] x1, double[ ] x2): 计算 $x1, x2$ 的内积
- double kernelLinear(double[ ] x1, double[ ] x2) : 线性核函数的实现

$$K(x_1, x_2) = x_1 x_2$$

- double kernelPoly(double[ ] x1, double[ ] x2) : 多项式核函数的实现

$$K(x_1, x_2) = (x_1 x_2 + 1)^d$$

- double kernelGauss(double[ ] x1, double[ ] x2) : 高斯核函数的实现

$$K(x_1, x_2) = e^{(-\frac{||x_1 - x_2||^2}{2\sigma^2})}$$

SVMFileReader : heart_scale 文件的读取

References

[1] Cho Jui Hsieh, Kai Wei Chang, Chih Jen Lin, S. Sathiya Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear svm. In *ICML*, pages 1369–1398, 2008.

[2] John C. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. 1998.

[3] By B Sch?lkopf, C. Burges, and A. Smola. T.joachims, making large-scale svm learning practical. In *Advances in Kernel Methods - Support Vector Learning, MIT-Press*, 2010.