超并行机器学习与海量数据挖掘

科创 4J 报告

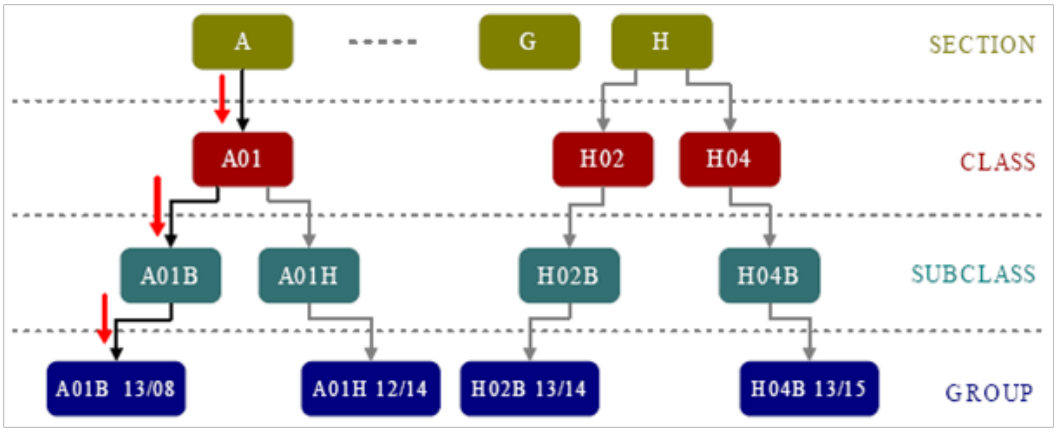
贾冬雨 孙翛然

项目简介

大规模专利分类是一个很具有代表性的模式分类 问题,它具有目前超大规模复杂模式分类问题的 全部特征。本作业要解决的是一个专利分类最顶层(SECTION层)的两类分类问题,它是大规模日文 专利数据集的一个子集。选择该问题的目的是希望通过求解该问题,让同 学们加深对并行机器学习、并行程序设计和大规 模数据挖掘的理解,培养大家的工程实践能力, 了解已有方法和技术的局限性,为未来的理论研 究和技术开发打下基础。

大作业问题描述

对第一层的A与其他B、C、D分类,这里A类为 正类,其它三类为负类



- train.txt 是训练用数据集, 含 113128 个样本。其 中 A 类的数量是 27876, B 类是 59597, C 类是 23072, D 类是 2583。
- test.txt 是测试用数据集, 含 37786 个样本
- 一行代表一个样本。样本格式如下: 标号 1 标号 2 标号 3 ...:数据
- 所有样本的特征维数均为 **5000**。
- 样本的输入已经进行了归一化处理,即取值在 (**0**,**1**)之间。

实验平台：

- 操作系统：Ubuntu 14.04（MAC 虚拟机）
- 编程语言：C/C++
- 电脑型号：Macbook Pro
- 处理器：2.2 GHz Intel Core i7(虚拟机分配 2 核)
- 内存：16 GB 1600 MHz DDR3（虚拟机分配 4G）

任务一：用LIBLINEAR直接学习上述两类问题,用常规的单线 程程序实现。

任务概述：

- 将文件转换为 liblinear 可以直接读取的文件格式
- 为了实现后续的 ROC 功能，修改 liblinear 源代码，直接输出预测值，而不是分类结果
- 制作 ROC 图
- 尝试实验不同分类器效果，不同权重的影响

输入格式转换

每行表示一个样本，给出的格式标号比较复杂，是多层次的多标号的，方便起见我们仅仅统计每行第一个字母，A 转换为 1，否则转换为-1， 空格之后的格式为 <index:value> ,不需要进行更改。

具体代码参看 preprocess 文件夹。

修改输出内容

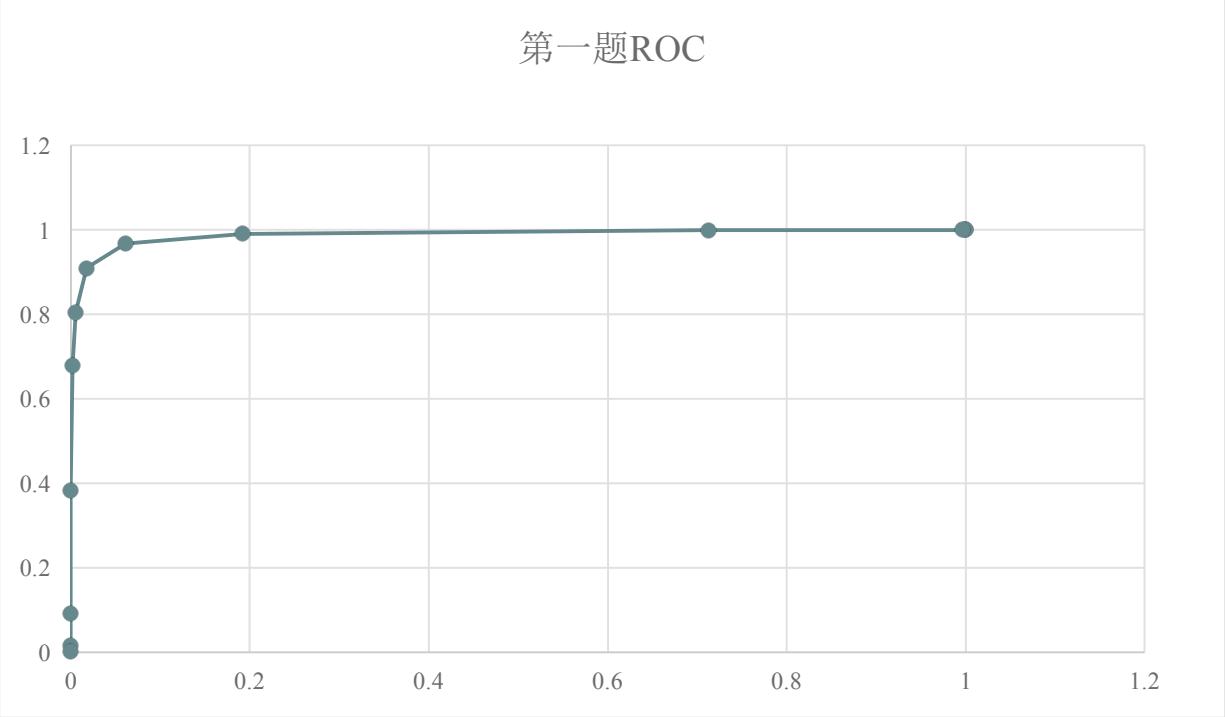
liblinear 源代码直接输出的即为预测结果，但是由于我们需要制作 ROC 曲线，我们直接修改源代码，让其直接返回预测值。并且由于 liblinear 仅仅并没有统计 TN,TP,FN,FP,所以我们在源代码中增加对这些量的计数。

统计 TP,TN,FP,FN 具体代码参考 predict.c 文件.修改 predict 返回值参见 liblinear.cpp 中 predict_value 函数，将其返回值从 label 改成计算值。

制作 ROC 图

通过对预测结果施以不同的阈值，可以得到几组不同的数据，ROC 代码参看 ROC 文件夹

阈值	TP	TN	FP	FN	FPR	TPR
-8	9150	0	28636	0	1	1
-6	9150	0	28636	0	1	1
-4	9150	111	28525	0	0.99612376	1
-2	9145	8221	20415	5	0.712913815	0.999453552
-1	9065	23135	5501	85	0.192100852	0.990710383
-0.5	8848	26886	1750	302	0.061111887	0.966994536
0	8324	28137	499	826	0.017425618	0.909726776
0.5	7363	28481	155	1787	0.005412767	0.804699454
1	6215	28582	54	2935	0.001885738	0.679234973
2	3496	28632	4	5654	0.000139684	0.382076503
4	824	28636	0	8326	0	0.090054645
6	151	28636	0	8999	0	0.016502732
8	3	28636	0	9147	0	0.000327869



尝试 liblinear 不同参数的运行情况

命令	想法	Accuracy
train	L2R_L2LOSS_SVC_DUAL	96.49%
train -w1 1 -w-1 3	不平衡	95.96%
train -s 4	不同的 svm,MCSVM_CS	96.45%
train -B 1	增加 bias	96.46%

具体可参见脚本/Q1/Q1.sh

我们主要基于以下一个想法来尝试参数，首先是不平衡的问题，由于训练样本中 A 类样本数量少于非 A 类，我们思考认为调节对不同类别样本的罚值权重或许可以对平衡问题有所帮助，由于 A 类和非 A 类的比例为 1：3 左右。 即对于 A 类问题，罚值权重为 3，非 A 类罚值权重为 1，希望这种方式可以部分解决非平衡问题。然而实验结果告诉我们并不是这样。

另外，对于 liblinear 自带的几个向量机，基本原理上大同小异，只有 MIRA support Vector 原理上有较大差异。所以我们也使用了-s 4 这个选项，但由于我们样本仅仅有两个标号，并不能展示 MIRA 的优势。

最后我们尝试添加了 bias, 但是效果都没有默认的情况好，这应该也是 liblinear 的设计者大致将最优方案设置为默认的情况吧。

```
"  for multi-class classification\n"
"    0 -- L2-regularized logistic regression (primal)\n"
"    1 -- L2-regularized L2-loss support vector classification (dual)\n"
"    2 -- L2-regularized L2-loss support vector classification (primal)\n"
"    3 -- L2-regularized L1-loss support vector classification (dual)\n"
"    4 -- support vector classification by Crammer and Singer\n"
"    5 -- L1-regularized L2-loss support vector classification\n"
"    6 -- L1-regularized logistic regression\n"
"    7 -- L2-regularized logistic regression (dual)\n"
"  for regression\n"
"   11 -- L2-regularized L2-loss support vector regression (primal)\n"
"   12 -- L2-regularized L2-loss support vector regression (dual)\n"
"   13 -- L2-regularized L1-loss support vector regression (dual)\n"
```

任务二：用最大最小模块化网络解决上述两类问题，用随机方式分解原问题，每个子问题用 liblinear 来学习，用多进程或多线程来实现。

任务概述

- 根据样本数量划分样本
- 合并样本
- 使用 MPI 多进程方案分别训练每个子问题
- 设计新的 minmax 预测函数

这些任务有以下几个问题，首先是文件 IO 问题，由于子问题数量大，如果直接合并样本将出现大量冗余数据，训练的文件 IO 耗时多，不利于我们的工作。所以我们决定不将合并结果输出到文件，而是修改 liblinear 源代码，使之能够读入两个文件，并在内部随机的对输入样本进行重排序，这样将大大减少中间文件加快运行速度。

其次是多进程的问题，刚开始采用了 fork 和 execul 方法，但是考虑到对多进程程序的封装以及进程间通信的方便，改用 MPI 方法，这样对多进程的封装更为规范。另外，本组还尝试了在 windows 下使用 windows API 进程多进程编程。

最后是 minmax 的预测函数，还是考虑 IO 的问题，刚开始的解决方案是直接调用写好的 predict.c ，然后生成 27 个子结果，然后再进行 minmax. 这样的解决方案一方面是生成了很多冗余的中间结果，另一方面是没有深入理解 predict.c 文件内部的工作原理，所以我们采用了将中间结果保持在内存中的解决方案。大大加速了预测速度。

样本划分

我们对样本进行估计，A 类大约有近 2.7 万个，非 A 类 8.4 万个，比例大致是 1：3，所以我们把 A 类分为 3 部分，非 A 类分为 9 部分，代码详见 Q2 文件夹。我们这一部分仅仅只是做划分，并不进行合并，合并部分将直接修改 liblinear 而在内存中合并，这样便减少了 IO.

并行训练

在 liblinear-1.96/trainparallel.cpp 文件是我们采用 mpi 编程方法参照原有的 train.c 函数编写的。 该程序提供了灵活的参数选择，并根据参数使用适当的进程数量，并且修改了 read_problem 函数，使之可以读入两个文件，并在内部按随机顺序组合。

MPI 部分非常简单，我们并没有用到通信，但我们还是让各个进程像 0 号进程传输了无用的数据，这里这个数据是用来告知本进程已经训练完毕，方便 0 号进程统计全部进程的训练时间。在各个进程内部我们设置了计时器.

```
timeval starttime,endtime;

gettimeofday(&starttime,0);

//do something****************//

gettimeofday(&endtime,0);

double timeuse = 1000000*(endtime.tv_sec - starttime.tv_sec) +
endtime.tv_usec - starttime.tv_usec;

timeuse /=1000;

printf("use time:%f\n",timeuse);
```

多进程结构：

```
MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &mpid);

MPI_Comm_size(MPI_COMM_WORLD, &all);

//do something*********************//

MPI_Finalize();
```

最大最小模块化网络

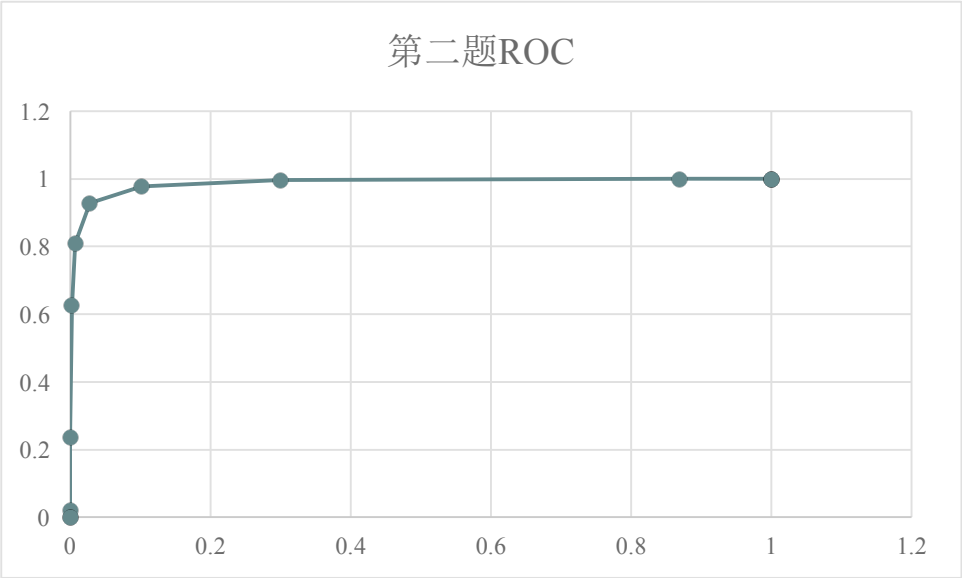
代码详见 liblinear-1.96/minmaxPredict.cpp, 文件读入测试数据和 27 个子模型，每个子模型都对测试数据进行预测，并在内存中进行 minmax 操作，输出最终结果。为了支持后续 ROC 操作，这里并不是进行 and, or 操作， 而是对于 and 取 min, 对于 or 改为取 max, 这样直接输出决策值，方便制作 ROC 曲线。

结果

Accuracy = 96.1573% (36334/37786)
TP = 8488
FP = 790
TN = 27846
FN = 662
TPR = 0.92765
FPR = 0.0275877
F1 = 0.921207

第一问的正确率在 96.5%左右，略高于随机化结果，但是我们在实验中每次随机的结果可能会略有浮动，在一次随机化结果中，准确率竟然高达 98%左右，但是由于随机化方法无法复现，我们只能认为随机化是一种可能有效的方案。

thd	TP	TN	FP	FN	FPR	TPR
-8	9150	0	28636	0	1	1
-6	9150	0	28636	0	1	1
-4	9150	2	28634	0	0.999930158	1
-2	9147	3744	24892	3	0.869255483	0.999672131
-1	9110	20069	8567	40	0.299168878	0.995628415
-0.5	8948	25726	2910	202	0.101620338	0.977923497
0	8489	27845	791	661	0.027622573	0.927759563
0.5	7407	28436	200	1743	0.006984216	0.809508197
1	5710	28579	57	3440	0.001990501	0.624043716
2	2167	28635	1	6983	3.49211E-05	0.236830601
4	196	28636	0	8954	0	0.021420765
6	1	28636	0	9149	0	0.00010929
8	0	28636	0	9150	0	0



运行时间

为了分析加速比，我们又写了 liblinear-1.96/trainSerial.cpp 和脚本文件 liblinear-1.96/Serailtime.sh 这个脚本运行串行的最大最小模块化网络并给出运行时间最终比较时间为：

并行（mpi）	**7.75s**	**27 个子任务**
串行	17.845s	27 个子任务
串行	6.37s	1 个任务

从理论上来讲，我们的并行程序应该为 17/27 秒的运行时间。这意味着我们程序的并行话程度并不理想，原因可能是因为虚拟机配置并不高，内核对程序的并行化程度不高。但是仅从 IO 的角度考虑的话并行程序已经很接近串行程序的速度，在数据量明显多于单个任务的情况下。

对于一个任务，训练规模大约在 11 万。对于 27 个子问题的情况，训练规模大约在 49 万，数据规模是单个任务的 5 倍，但是运行时间是单个任务的 3 倍。

任务三：基于先验知识用最大最小模块化网络解决上述分类问题。

在第二问中，我们已经成功的只做了多进程并行训练的程序，已经 minmax 的预测程序。对于任务三而言，唯一的不同就是分类方案不同，然后直接调用第二问中的 trainparallel 和 minmaxPredict 函数就可以了。所以我们将这一问的分析重点放在如何分解样本上。

分析样本

我们用 C++ stl 中的 map 来统计不同标号的出现数目，并且累加结果。得到结果：

A1:3721 # 3721

A21:142 # 3863

A22:52 # 3915

A23:1850 # 5765

A24:61 # 5826

…….

前者表示标号和数量，后者表示累加数量。

为了和第二问保持一致，我们依然将 A 类分为 3 个子问题，非 A 类分为 9 个子问题。根据样本数量，我们将 C 类分为两个问题，B 和 D 类一共分为 9 个子问题。首先，我们把单独标号数目较大者拿出来，如：

B41:9814

B60:9305

B65:9480

单独算作一组，然后把连续的标号分为一组，分组方案为：

A is split to 3 groups. [0,46] [47,61] [62,63]	B&D is split to 7 groups: B[0,21]；B[22,27]； B[28,32][42,44]； B[41]；B[60]；B[65]； B[61-64][66-82]D；	C is split to 2 groups: C[1,8] C[9,30]

样本划分

由于在第二问的随机方法中每次结果不确定，而且偶然出现了 98%的高准确率，我们决定在先验的部分中也加入随机部分的影响。而且考虑到上课中讲到适当引入重叠可以提升边际预测准确率，我们最终尝试了以下四种方案：

- 完全先验（按照上述讨论进行先验划分）
- A 类先验，其他随机。
- A 类随机，其他先验
- 引入重叠

引入重叠部分，即将边界标号同时给到两个子问题中，如 A 类的 46，47 为子问题边界标号，则将 46 这个标号同时分配到子问题一合子问题二中，以此类推。若边界标号规模较大，则随机选取部分边界标号分配到两个问题中。

结果为：

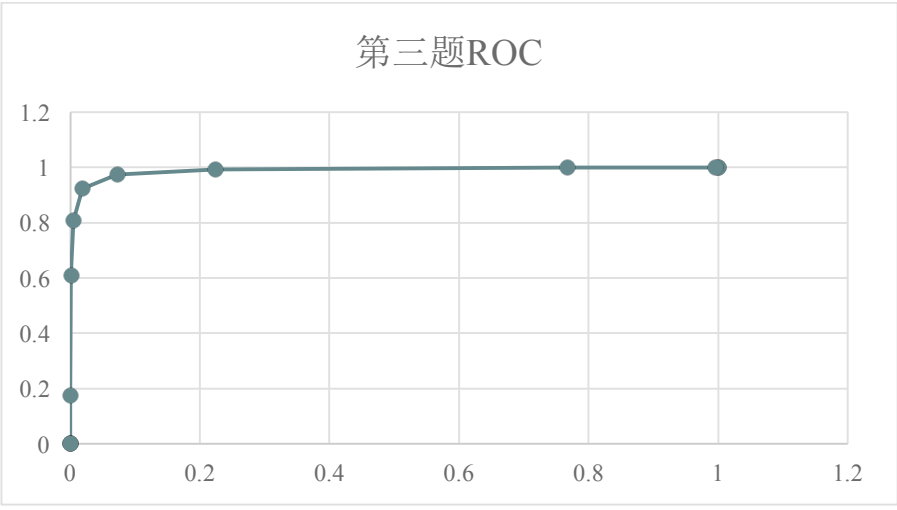
类别	Accuracy	时间	F1
A 类先验，其他随机	96.4881%	7.56s	0.926068
A 类随机，其他先验	96.1282%	7.24s	0.921407
完全先验	96.7025%	7.45s	0.931236
完全随机	96.1573%	7.36s	0.921207

引入重叠（少）	96.7634%	7.96s	0.932554
引入重叠（多）	96.6866%	9.59s	0.931216

重叠部分详见文件夹 overlap,对于 我们对边界部分的重叠了 200 个左右的样本，可以看到提升还是了准确率和 F1 值。当引入重叠较多时，我们重叠了 1000 个左右的样本，准确率轻微下降了。原因可能是对于引入重叠较多的情况下，子问题规模变大，复杂度上升，线性方法刻画难度上升导致的。而对于少量的重叠，可以较好地处理边界情况而导致准确度上升。

但是由于这几种分类方案准确率都在 96%以上，我们可以认为其分类效果几乎在同一水平，ROC 曲线几乎完全重合。仅给出完全先验的 ROC 曲线。

thd	TP	TN	FP	FN	FPR	TPR
-8	9150	0	28636	0	1	1
-6	9150	0	28636	0	1	1
-4	9150	116	28520	0	0.995949155	1
-2	9147	6634	22002	3	0.768333566	0.999672131
-1	9085	22219	6417	65	0.22408856	0.992896175
-0.5	8918	26564	2072	232	0.072356474	0.974644809
0	8437	28103	533	713	0.018612935	0.922076503
0.5	7387	28505	131	1763	0.004574661	0.807322404
1	5582	28608	28	3568	0.00097779	0.610054645
2	1620	28636	0	7530	0	0.17704918
4	1	28636	0	9149	0	0.00010929
6	0	28636	0	9150	0	0
8	0	28636	0	9150	0	0

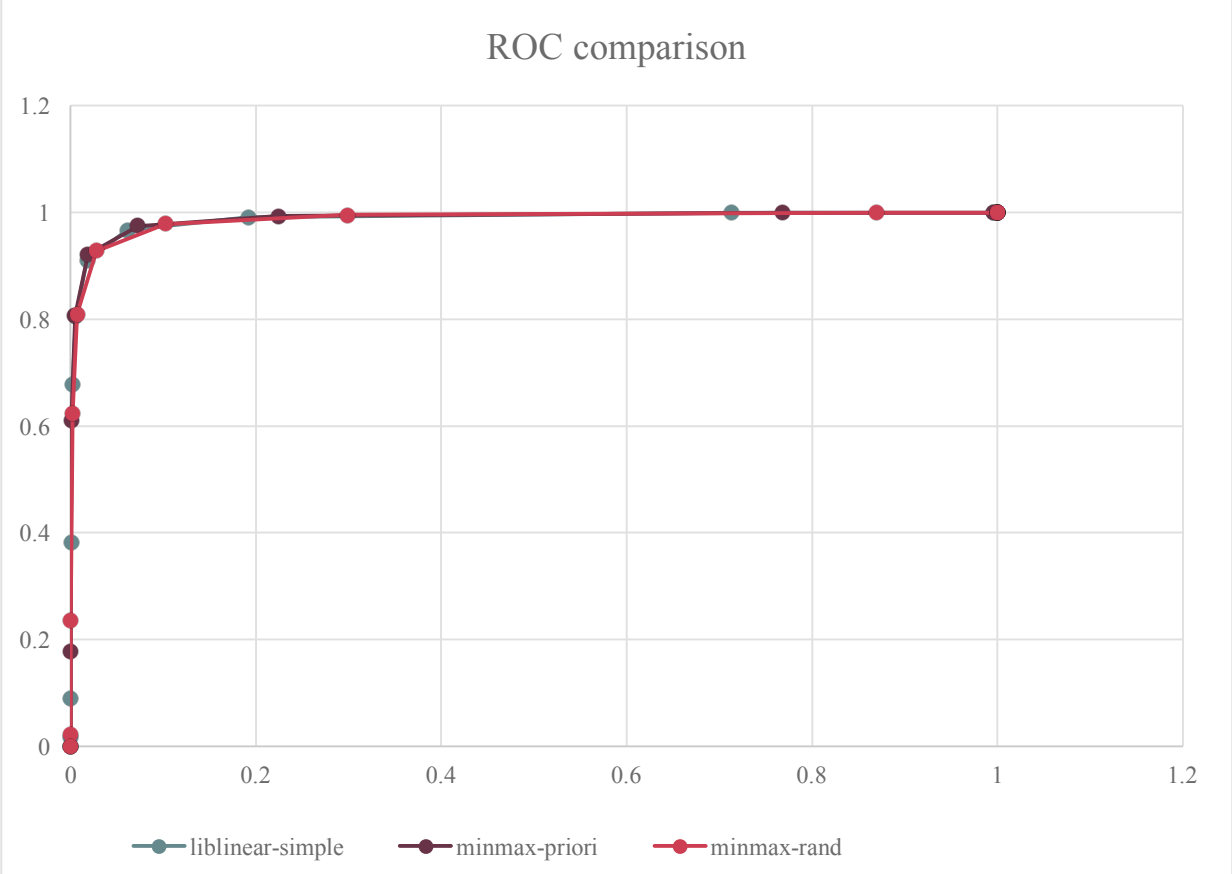


任务四与五，ROC 曲线与时间分析

ROC 曲线已经包含在前三个任务中，时间也包含在之前每一个任务中。

这里宏观的来总结一下，由于准确率很高，所有的方案准确率都在 95%以上，ROC 曲线几乎完全重合，没有显著差别，比较起来意义不大。

	liblinear-simple		minmax-priori		minmax-rand	
thd	FPR	TPR	FPR	TPR	FPR	TPR
-8	1	1	1	1	1	1
-6	1	1	1	1	1	1
-4	0.99612376	1	0.995949155	1	0.999930158	1
-2	0.712913815	0.999453552	0.768333566	0.999672131	0.869255483	0.999672131
-1	0.192100852	0.990710383	0.22408856	0.992896175	0.299168878	0.995628415
-0.5	0.061111887	0.966994536	0.072356474	0.974644809	0.101620338	0.977923497
0	0.017425618	0.909726776	0.018612935	0.922076503	0.027622573	0.927759563
0.5	0.005412767	0.804699454	0.004574661	0.807322404	0.006984216	0.809508197
1	0.001885738	0.679234973	0.00097779	0.610054645	0.001990501	0.624043716
2	0.000139684	0.382076503	0	0.17704918	3.49211E-05	0.236830601
4	0	0.090054645	0	0.00010929	0	0.021420765
6	0	0.016502732	0	0	0	0.00010929



对于时间测量而言，很有意思的是，由于我们目前的计算机体系结构，反复命中可以大大提升程序执行效率，因为数据都在内存或者 cache 里面。所以当我们做实验时候，以 27 个子问题的并行程序执行为例：

执行三次的时间分别为：

13.6s

8.7s

6.8s

而与此同时首次执行仅有一个任务的第一问代码时，花费时间为 7s 左右，这就说明反复运行一个程序确实会提高其运行效率。这无疑对我们对时间进行分析提高了难度。所以我们统一将所有程序运行三遍，并以三遍的平均时间来分析。

类别	第一次执行时间（s）	第二次执行时间(s)	第三次执行时间(s)	二三次平均时间(s)
串行（1 个子问题）	8.98	6.01	6.12	6.065
串行（27 个子问题）	20.76	16.63	17.16	16.895
并行（随机 27 个子问题）	13.2	7.336	7.23	7.283

可以看到第一次时间相对比较长，这也是正常可以理解的。

并行化程度来看，在 27 个子问题的程序中，并行程序的执行速度大约是串行的两倍，这也许是因为虚拟机仅有两个核的缘故。

拓展部分

结合第三第四次作业的情况，我们把第三第四次作业中的 LMS 和 perceptron 方法拿过来制作自己的分类器。由于这个试验中我们使用了线性的划分方法，所以事实上对于 MLP 是没有必要的。

LMS & Perceptron 简要介绍

LMS Learning Rule

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha \nabla F(\mathbf{x})\Big|_{\mathbf{x}=\mathbf{x}_k}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + 2\alpha e(k)\mathbf{z}(k)$$

$${}_1\mathbf{w}(k+1) = {}_1\mathbf{w}(k) + 2\alpha e(k)\mathbf{p}(k)$$

$$b(k+1) = b(k) + 2\alpha e(k)$$

Perceptron Learning Rule

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t-a)\mathbf{p}$$

$$b^{new} = b^{old} + e$$

我们修改了 liblinear 源代码，linear.cpp 和 linear.h . 这样只用调用参数 –s 14 和 –s 15 就可以使用 Perceptron 和 LMS 分类器。

代码描述

首先在 linear.h 中加入 Perceptron 和 LMS 两种分类器的名称。然后我们尽量维持 model 不被修改，保持原有代码的兼容性。

首先我们先弄清楚原有数据结构的形式，注意到 feature_node 是以数组的形式存贮 index, 并以-1 代表结束的。若添加了 bias, 则 train.c 函数会给数据最后默认加上一维 1.

首先我们在 linear.c 中加入 hardlim 和 purelin 两个函数 用于进行向量乘法。不必单独考虑 b, 仅需将 w 向乘上 feature 向量即可。

然后我们实现 perceptron 和 LMS 两种方法，这两种方法内部结构十分相似，仅是 hardlim 和 purelin 以及是否有学习系数的差别。所以我们将其放在一个函数中，最大迭代次数设置为 1000。

然后我们在 train_one 中加如对 Perceptron 和 LMS 的支持。

另外，在 train.c 函数中，对于选取 LMS 和 Perception 的用户而言，自动设置 bias 值为 1，自动设置默认误差为 0.002.

并且在我们的实验中，统一使用 batch mode 来进行 LMS 和 Perceptron 的更新

Sequential Mode VS Batch Mode

我们刚开始使用的是 Sequential mode ,后来发现准确率不高，改用 Batchmode 后大大提升了准确率。

种类	Accuracy
Perceptron Sequential	89.3%
Perceptron Batch	95.7947%
LMS Sequential	88.1%
LMS Batch	93.831%

LMS Sequential 之所以这么低并不全是 Sequential mode 决定的，而是由于参数选取不合理，学习系数选取不合理导致了收敛过慢。

在以后的问题中，我们同一使用了 Batch Mode.

运行结果

种类	子问题数	Accuracy	时间(s)
LMS	1	93.831%	32.36
Perceptron	1	95.7947%	19.5
LMS(随机)	27	94.4609%	77.8
Perceptron（随机）	27	95.8741%	30.07
LMS（先验）	27	95.6677%	101.57

Perceptron（先验）	27	96.3558%	32.97

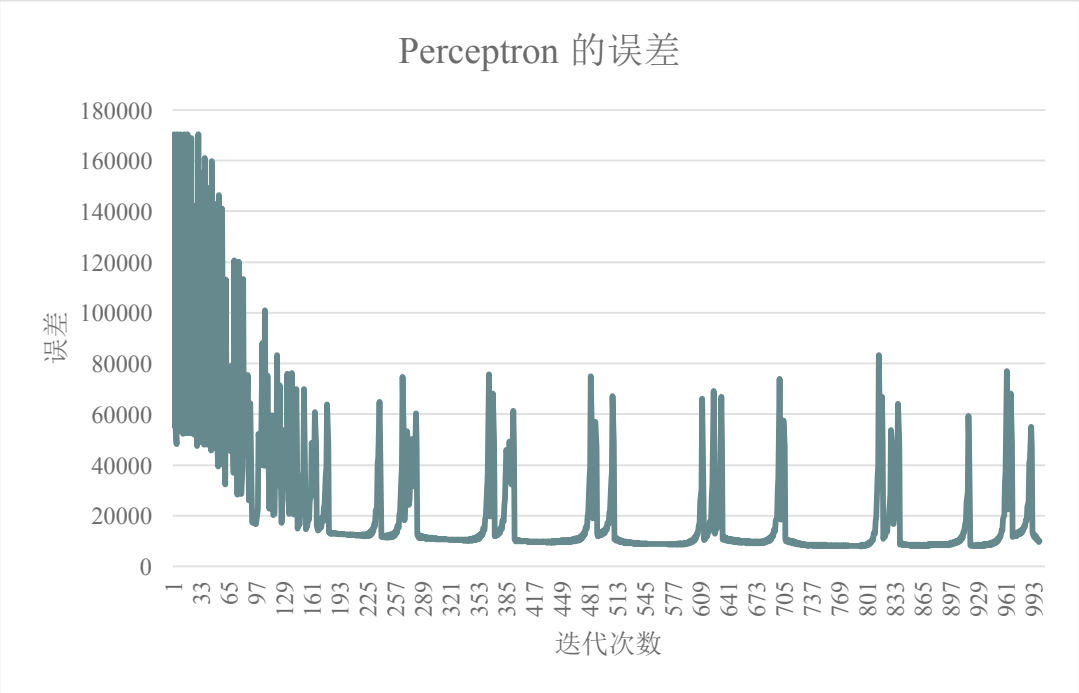
可以清晰的看到，先验>随机>单一问题， Perceptron 好于 LMS, 这和我们经验中的结果是不一样的。分析肯能有以下几种可能。一是我们的步长选取不合适，导致收敛速度慢或者总是过冲震荡。二是我们选取得迭代次数不够多，三是我们选取的误差上限过大。

Minmax 对于问题的准确性有较为明显的提升，先验性对于随机性有较为明显的提升，不过其预测准确率还是比不上 liblinear 自带的分类器。

迭代次数分析

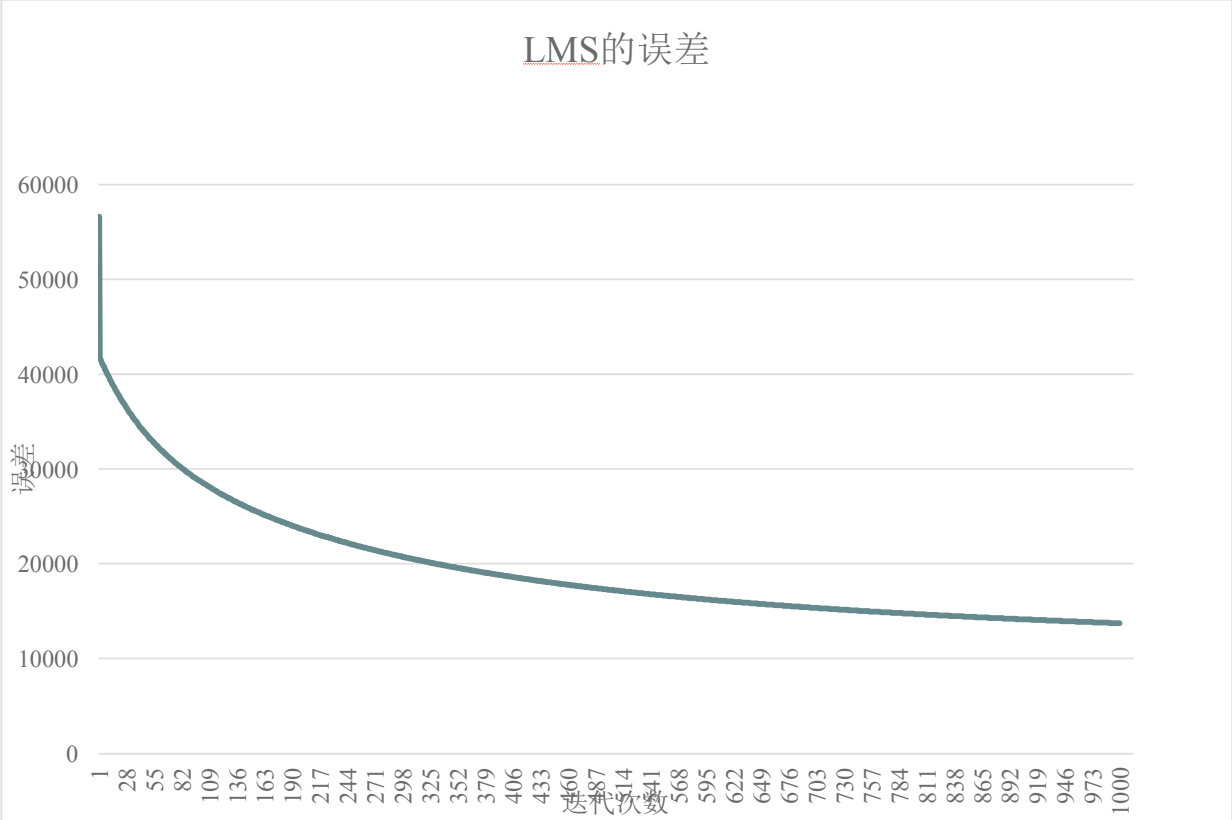
在 LMS 中，迭代次数总是达到最大后（1000）退出，而 perceptron 的收敛速度很快，由于数据量巨大，实时上只要一次迭代就可以达到较高的准确率，但是这样做实际上只是后面的数据在发挥效果，前面的数据只是在大致调整参数。所以我设置了最小迭代次数是 300 次。因而可以看到，我们的 perceptron 的运行速度大约只有 LMS 的一半。

收敛情况分析



可以看到，Perceptron 的缺点在于由于没有学习系数，一次调整可能会引起巨大的跳跃，而在我们的图中可以看到，最小误差大约在 10000 左右，并且在 200 多次后就已经收敛到了比较理想的程度。

对于我们的 LMS 而言，其收敛速度非常缓慢，然而当我们想要调整学习系数时（初始设为 1/problem_line, 发现稍微调大学习系数后问题就发散了。不过也可以看出 LMS 在 1000 次迭代后基本误差趋于稳定了。



MLP 分类器

在尝试过 LMS 和 Perceptron 之后，我们决定进一步使用 MLP 分类器，详见 mlp 文件夹，由于每个节点都需要保存 5000 维的权值，和 liblinear 原有的保存 model 的方式很不一样，所以，我们自己重新写了 mlp 的训练和预测函数，并没有选择在原有的 liblinear 代码中修改。简单起见，我们将 27 个子问题的数据在 mlp 文件夹下生成，然后用 fork execl 分别预测，最后用 minmax 程序合并。

$$\begin{pmatrix} Weight \\ correction \\ \Delta w_{ij}(n) \end{pmatrix} = \begin{pmatrix} Learning \\ parameter \\ \eta \end{pmatrix} \begin{pmatrix} Local \\ gradient \\ \delta_j(n) \end{pmatrix} \begin{pmatrix} Input\ signal \\ of\ neuron\ j \\ y_i(n) \end{pmatrix}$$

- The local gradient is given by

$$\delta_j(n) = e_j(n)\varphi_j'(v_j(n)) \tag{4.14}$$

when the neuron j is in the output layer.

- In the hidden layer, the local gradient is

$$\delta_j(n) = \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{4.24}$$

computed recursively from the local gradients of the following layer, back-propagating error

关于非线性函数，我们选择了 sigmoid 函数，并将其纵向拉伸为两倍后再纵向向下平移一。

然而由于问题是分线性的，一次迭代就要花费数十秒时间，给我们的训练带来很大难度，我们选择了 5 个隐藏层节点，一个输出节点作为基本结构，仅仅迭代 10 次。

类别	任务数	时间	准确率
MLP	1	108.2s	82.52%
MLP 并行随机	27	245.4s	89.21%
MLP 并行先验	27	236.23s	90.35%

由于训练时间比较长，我们没有进行次数更多的迭代，另外学习参数的选取也仅仅是根据结果的人为判断，没有科学的依据，所以准确率不是很尽如人意。但是更多的迭代次数或许能够带来更高的准确率。

组员贡献

贾冬雨 70%：linux 下所有代码任务的编写和整合。数据绘图，表格。报告。

孙翛然 30%：多进程问题在 windows 平台的改写。LMS Perceptron 原理，代码支持。