超并行机器学习与海量数据挖掘大作业报告

凌康伟
5140219295
宋海越
5140309197

2017 年 4 月 28 日

目录

1 实验目的

1.1 问题背景

专利的自动分类问题是具有重要实践意义的机器学习问题之一。随着专利数量的日益增加，耗费大量人力物力的手工分类时代渐渐被人工智能分类专利的时代取代。我们的目的是处理大量日文专利的分类，但这个问题被简化成只分类成两类的问题，A 类或者非 A 类。虽然问题已经被简化了很多但是依然很具有挑战性，比如问题规模很大，训练数据达到了 100MB 以上，并且问题具有层次结构，除了顶层标号还有很多下层标号，我们也需要将其考虑进去，实现更优的子问题划分。并且数据很不均匀，某类的数据数量可能比其他类多很多，是十分不平衡的问题。SVM（支持向量机）可以较好解决这个问题，它学习能力强大可以保证经验误差较小并且由于其最大化边界使其具有很优秀的泛化性能。此次实验旨在通过对日文专利数据的分类，加深我们对 SVM 的理解并且了解不同的数据分类方法来达到更优的结果。

1.2 问题描述

在此次试验中我们需要将日文专利分成正类和负类，正类包括第一层标号是 A 的所有数据。负类包括第一层是 BCD 的所有数据。训练集中有 A 类 27876 个，B 类 59597 个，C 类 23072 个，D 类 2583 个。共 113128 个样本。负类的数量是正类的三倍多，所以此问题并不平衡。并且负类中 D 类数据很少。

每一个样本包含两部分，第一部分是标号部分，每个标号包含 ABCD 中一个字母代表所属大类，以及之后的两位数字代表第二层小类，更细分的小类以斜杠隔开。

标号和数据之间以冒号隔开，数据有 5000 维，并且所有输入都已经进行了归一化处理。

1.3 实验要求

1. 用 LIBLINEAR 直接学习上述两类问题，用常规的单线程程序实现。

2. 用最小最大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 学习，用多线程或多进程程序实现。

3. 用最小最大模块化网络解决上述两类问题，用基于先验知识（层次化标号结构信息）的问题分解策略分解原问题，每个子问题用 LIBLINEAR 学习，用多线程或多进程程序实现。

4. 给出上述三种分类器的 F1 值，画出这三种分类器的 ROC 曲线，并说明哪种

分类器分类性能最好。

5. 比较 LIBLINEAR 分类器与基于先验知识分解的最小最大模块化 LIBLINEAR 分类器的训练时间和分类时间。最小最大模块化 LIBLINEAR 分类器只考虑完全并行情况下所需的时间，可以在代码中添加计时器对每个进程进行计时来确定完全并行情况下消耗的时间。

6. （任选项）实现一个基分类器，如 MLP 或 SVM 多项式核，完成上述第 2 和第 3 个任务，并与 LIBLINEAR 作比较。

7. （任选项）使用 GPU 或服务器实现上述第 1 至第 3 个任务。

2 实验算法

2.1 SVM 算法

LIBLINEAR 的基础是 SVM 算法。我们可以把训练数据点想象成是高维的一些点，我们需要找到一个超平面把这些点给分割开来。我们对这个超平面的要求是在正确分类的前提下保持两类数据的间距（margin）最大，而这些间距是由一些边缘的数据点确定的，这些数据点就叫支持向量。支持向量机的优点是可以最大化间距来达到很好的泛化效果，并且针对数据非线性可分的情况我们还可以用核方法来把数据映射到高维空间。

2.2 最大最小模块化网络

为了解决大规模不平衡的问题，我们可以用最大最小模块网络。思路是首先把大规模问题分解成一些独立的子问题，然后并行学习这些子问题，最后把这些子问题的答案结合起来生成原问题的解。分解算法采用部分对部分（partVSpart）的策略。首先把整个训练集分为正类和负类。假设 (x,y) 是一个样例，D 是整个训练集。我们首先把训练集分为 D^+ 和 D^-，其中:

$$D^+ = \{(x, y) \in D \mid y = +1\} \tag{1}$$

$$D^- = \{(x, y) \in D \mid y = -1\} \tag{2}$$

之后我们需要把 D 分解成 $D_i^+(i = 1, ..., N^+)$ 和 $D_i^-(i = 1, ..., N^-)$ 为了保证平衡性，我们需要确保 $|D_i^+| \approx |D_j^-|$ for i=1,...,N^+ and j=1,...,N^-. 之后我们需要把这些小数据组连接起来构成大量子问题。我们连接 D_i^+ 和 D_j^- 构成子问题的训练数据，其中

$D_{ij} = D_i^+ \cup D_j^-$。我们得到了 $N^+ \times N^-$ 个独立的子问题。对于每一个子问题，我们都用一个模型训练，所以我们得到了 $N^+ \times N^-$ 个独立的模型 M_{ij}。

接下来我们需要考虑模型的合并，我们用最小化原则和最大化原则来将这 $N^+ \times N^-$ 个模型产生的输出组合在一起。假设 (x_t, y_t) 是一个测试数据，每一个模型 M_{ij} 会对这个数据进行预测产生答案 p_{ij}。

我们首先用最小化原则，对于有相同正训练数据的模型，我们取其中的最小值作为输出。

$$(x_t, y_t) \xrightarrow{M_{ij}} p_{ij} \tag{3}$$

对于 $j = 1, ..., N^-,$, 结果 $p_i j$ 都在相同的正数据 D_i^+ 上训练过，所以对他们采用最小化原则。

接下来我们对最小化原则后产生的结果采用最大化原则来产生最终结果 p。

$$p = \max_{i=1,...,N^+} p_i \tag{4}$$

整个流程如图 1 所示：

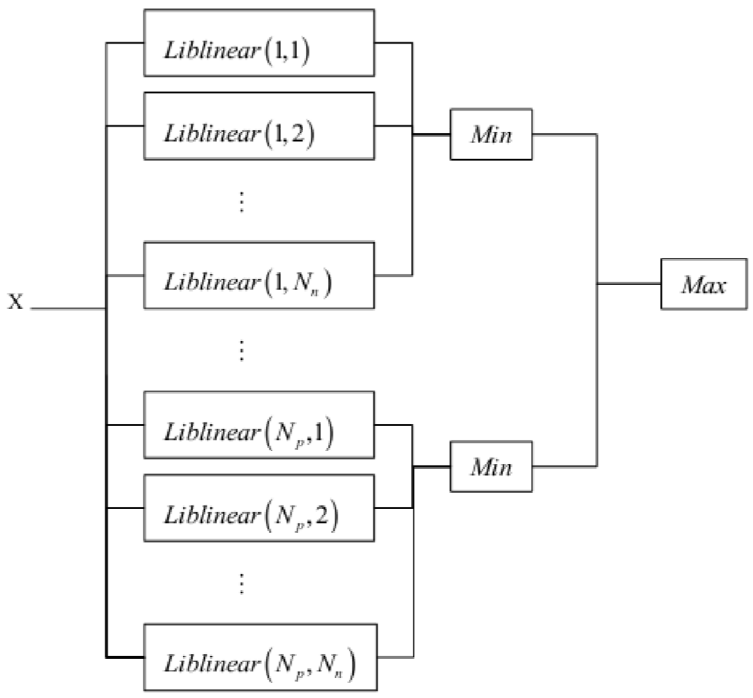


图 1: The structure of M^3-Liblinear

2.3 数据分类算法

在用最大最小化模块网络算法是，我们需要把数据分成 $D_i^+(i = 1, ..., N^+)$ 和 $D_i^-(i = 1, ..., N^-)$ 不同的分类方案会有不同的效果，我们实验了两种不同的数据分类算法，分别是随机分类算法和基于先验信息的按标号分类算法。一下是算法的描述部分，具体算法详见程序说明部分。

随机分类算法

训练数据集有 ABCD 四类数据，其中 A 是正类，BCD 是负类。采用随机分类算法时，首先确定分块的大小 K，这样可以保证不会有不平衡的问题。然后把 A 类中的所有数据打乱，然后分成 $\frac{|D^+|}{K}$ 类，把 BCD 中的数据一起打乱，然后分成 $\frac{|D^-|}{K}$ 类，每个模型 M_{ij} 取一个正小类 D_i^+ 和一个负小类 D_j^- 作为训练集。每一个模型的训练数据中都有 K 个正数据和 K 个负数据，十分均衡。

随机分类忽略的先验信息，在特征空间每个模型的训练数据之间的距离可能比较远，没有规律，所以很多时候随机分类算法的表现并不算太好。

按标号分类算法

基于先验信息的按标号分类算法以二级信息（字母后面的两位数字）为基准，把含有相同二级信息的数据尽量分在一类里面，这样可以让这些数据在特征空间中距离更短。比如在正类 A 中，我们把 A01 按照 K 的尺度分成若干小类，其中不包含除 A01 以外的正类数据。

Algorithm 1 Algorithm for CLASS task decomposition

Input: $modulesize$, D(data set), S(a set of the subset to be split)
Output: T(a set of the split subsets)

```
S ← {D}
T ← ∅
while S is not empty do
  for each element E ∈ S do
    remove E from S
    if |E| is much larger than modulesize then
      if E has subclasses then
        split E by taxonomy
      else
        split E randomly
      end if
      add the subclasses of E to S
    else
      add E to T
    end if
  end for
end while
merge the sets whose size are much smaller than modulesize
```

图 2: Algorithm for CLASS task decomposition

2.4 多进程

由于 python 编程的简洁方便，以及很多现有的机器学习以及数据分析相关工具，我们选择 python 来开发和实验。

由于 GIL(Global Interpreter Lock, 全局锁) 的存在，python 中的多线程其实并不是真正的多线程，多线程无法利用多核，对于 CPU 计算密集的任务，要选择多进程来利用计算机的多个处理器进行并行计算。python 提供了非常好用的多进程包 multiprocessing。而 min-max network 的计算具有非常高的对称性，因此我们用了更高一层的 python 封装库 joblib 来进行并行训练。

Joblib 提供了一个简单的辅助的类来多进程并行计算循环。其核心就是通过将代码写成 python 生成器的形式，然后转化为并行的计算。

```
1 >>> from math import sqrt
2 >>> [sqrt(i ** 2) for i in range(10)]
```

```
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Listing 1: Normal example

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Listing 2: Joblib example

在底层，`Parallel` 会创建一个进程池，然后通过 `fork` 出的很多个 python 解释器的进程来执行循环的每一个 item。`delayed` 函数是 joblib 中的一种函数式风格的调用。

3 程序说明

3.1 架构

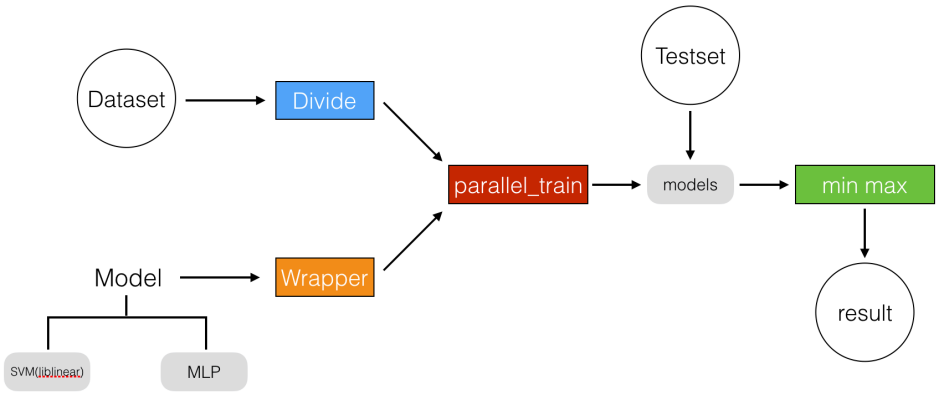


图 3: 系统架构

我们的系统的结构如图三，训练数据集经过 `divide` 被划分成特定大小的数据块。同时 `Model` 经过 `Wrapper` 后，与数据块一起作为 `parallel_train` 的参数。`parallel_train` 会根据数据块的结构训练一定数量的模型，同时在并行的过程中，当模型训练结束，会对测试数据集进行预测。所有模型的训练、对测试集的 predict 的结束后，并行的部分结束。之后所有的模型的输出会 feed 到 `min max` network 中，最终会输出分类的结果以及分类结果的 value(绘制 ROC 曲线)。

3.2 数据预处理

由于提供的训练文件和测试文件的格式不完全符合 liblinear 的要求，因此需要对数据进行预处理。我们将专利标号为 A 的 sample 作为正类，label 是 1，其余的 label 都是-1。通过运行如下命令来对数据预处理。

```
$ python3 preprocess.py data/train.txt
```

处理好的符合 liblinear 格式的数据会写入后缀名为 `.svm` 的文件中，原有的专利标号会提取到 `.tag` 文件中，用于数据分块。

3.3 数据划分

为了把 *train.txt* 中的数据的正类和负类分别分为等大小的若干个正子类和父子类，需要对原始数据进行划分，划分的方法有两种，第一种是随机划分，第二种是按标签划分。划分函数在 *divide.py* 中实现。

```
def divide(tag,feature,size_part,sort_tag=2)
```

1) 这个函数是接口函数和主要处理函数，tag 是每个数据的标号组成的数组，feature 是这些标号对应的数据组成的数组，tag_i 和 $feature_i$ 对应，完整地说明了一个数据。*size_part* 代表子类的大小，比如 $size\_part = 100$ 说明我们把正类和负类分别按 100 的尺度分成很多子类，也就是每个 M_{ij} 含有 100 个正类数据和 100 个负类数据。*sort_tag* 是数据划分方法，分别有随机划分（打乱原有顺序）和按照 *train.txt* 里的顺序不变，还有按照标号排序后划分。

首先我们先按照 *sort_tag* 对数据进行排序或打乱或保持原样。然后对正类和负类分别建立两个数组，其中一个数组中全部数据标签都一样，如果达到了 *sort_part* 就把这个数组加到答案数组中去，另外一个保存某个标签数据加到第一个数组中后剩下的数据。例如，$sort\_part = 100$，我们标号 A01 有 230 个数据，我们先把前 100 个数据加到第一个数组，然后把这个数组加入到答案里面，清空这个数组，继续做剩下的 130 个，加到第一个数组后还剩下 30 个，这时加到第二个数组（之前第二个数组也可能有其他标号剩下的数据），等第二个数组满之后也加到答案数组里面。

```
def add(i,tag,feature,a1,a1x,a2,a2x,ans,ansx,k)
```

2) 此函数为添加一个数据的接口，由这个函数分配加到第一个数组还是第二个数组，判断的依据是是否和上一个数据一样，如果一样就代表当前标号还有数据，加到第一数组否则加到第二数组。

```
def throw_remain(a1,a1x,a2,a2x,ans,ansx,k)
```

3) 这个函数的功能是将第二数组中数据存入答案中，并且清空第二数组。每次当某一标号的数据处理完毕后，首先将第一数组的数据加到第二数组中，然后清空第一数组，然后判断第二数组中的数据时候大于 *sort_part*，如果大于说明可以放入答案中，此时把前 *sort_part* 个数据放入答案中。

3.4 模型及包装 Wrapper

`Model` 类是所有模型类的父类，定义了实际的模型需要定义的训练方法和预测方法。以 SVMModel（liblinear）为例子，其实现了以下方法：

- `train(self, X, y)`

  用分配到的训练数据 X, y 进行训练。

- `predict(self, X, y=[])`

  对提供的测试 X 预测。

对于模型的训练调用模型类的 `train` 方法以及之后的 `predict` 方法对测试集分类。除此之外，需要对训练的时间以及分类的时间进行统计，这些工作是在 `wrapper` 中完成的，具体实现可以参考 `wrapper.py` 文件。

3.5 多进程并行

`joblib` 的存在让多进程的代码变得非常简单。我们代码中是以下这个函数负责 spawn 多个进程进行训练。

```
def parallel_train(train_func, Xs, Ys, n_jobs=8, testX=None):
    """
    train models in parallel (M3-network)

    :param train_func: the function to train a model, should accept three
    argument (X, y, identifier=(i,j))
    :param Xs: a list of list of X
    :param Ys: a list of list of Y
    :param n_jobs: train in n separate process (-1 means use all cpu cores)
    :return: a list of models' prediction on testX or model identity if testX is
None, models can be saved when done training.
    """
    minmodules = len(Xs[0])
```

```
    maxmodules = len(Xs)

    return Parallel(n_jobs=n_jobs)(delayed(train_func)(Xs[i][j],
                                                       Ys[i][j],
                                                       (i, j),
                                                       testX=testX)
                                   for i in range(maxmodules) for j in range(
    minmodules))
```

代码中 `Xs[i][j]` 代表是正类中第 i 组数据与负类中第 j 组对应的模型的训练数据，在数据划分之后通过 `utils.py` 中的 `getData` 函数组装。

3.6 Min Max Modular Network

`min max` 的部分如1, 实现在 `wrapper.py` 里。

- `minmax(pred_val, pred_label, minmax_shape)`

  输入的参数是预测的值，预测的分类（label），以及 min max 模块化网络的形状。并行训练完毕后得到的结果是平铺的列表，需要按照模块化网络的形状进行 `reshape` 然后 min max 得到最后的结果。

4 实验结果

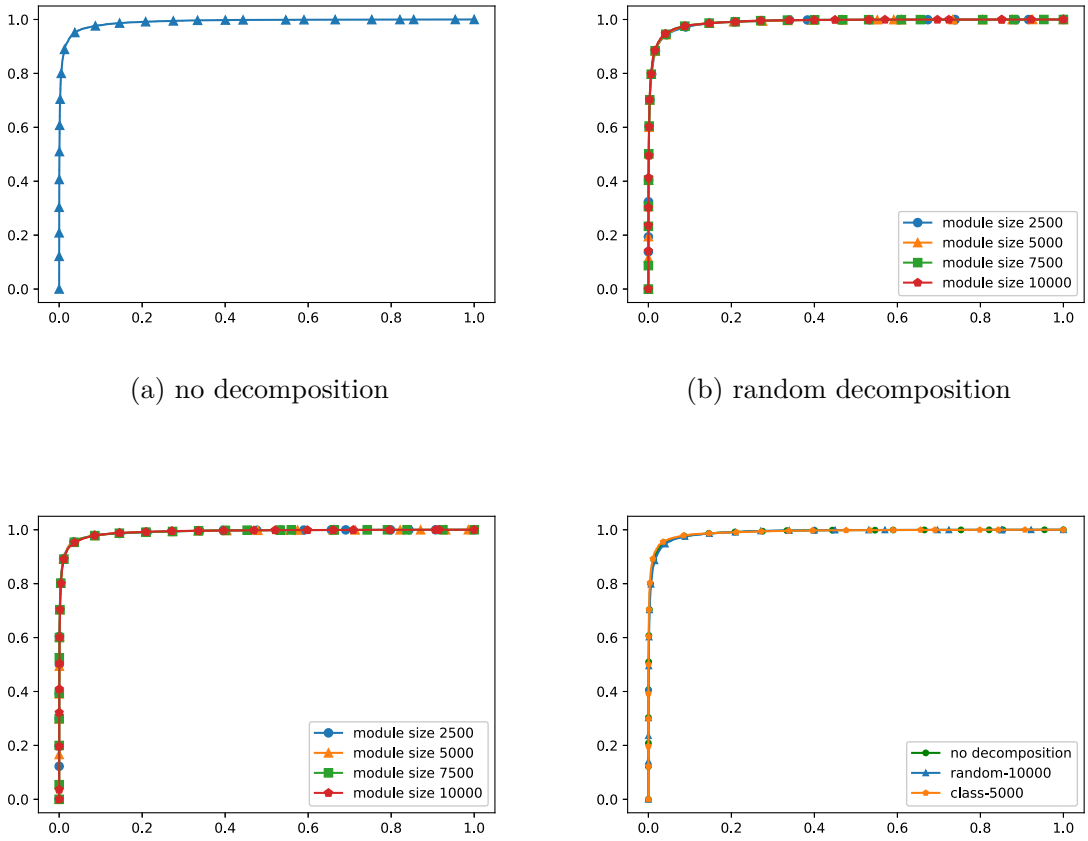
4.1 训练结果

表 1: 不同分类器分类效果

分类器	模块大小	accuracy	precision	recall	F1
liblinear		96.50	94.35	90.97	92.63
liblinear + 随机划分					
	2500	95.92	91.33	91.87	91.60
	5000	95.91	91.11	92.11	91.61
	7500	95.97	91.08	92.42	91.75
	10000	96.09	91.35	**92.62**	91.98
liblinear + 先验划分					
	2500	96.78	**94.37**	92.21	93.28
	5000	**96.79**	94.36	92.25	**93.29**
	7500	96.73	94.06	92.31	93.18
	10000	96.68	94.06	92.10	93.07

表 2: 不同分类器训练时间和分类时间

分类器	模块大小	分类器数量	单个分类器训练时间/s	分类时间/s
liblinear		1	11.1	4.02
liblinear + 随机划分				
	2500	420 (12×35)	1.02	8.76
	5000	108 (6×18)	2.15	8.03
	7500	48 (4×12)	3.32	7.88
	10000	27 (3×9)	3.30	7.56
liblinear + 先验划分				
	2500	420	0.92	9.29
	5000	108	2.58	8.30
	7500	48	3.23	8.43
	10000	27	3.48	8.54



(a) no decomposition

(b) random decomposition

(c) class decomposition

(d) 比较

图 4: ROC 曲线

4.2 结果分析

首先从精度上进行分析，总体来讲具有先验知识的最大最小网络表现相对最为出色，F1 很高，这说明有效利用先验知识对分类器的分类是有很大帮助的。直接训练和随机划分的最大最小网络各有优劣，但总的来说随机划分表现不佳。所以没有先验知识的话，随机划分数据的意义并不大。

再看训练的速度，使用最大最小网络可以有效增加训练的并行度，并且模块大小越小速度越快，用时远远少于只用单个分类器的训练时间。但是最大最小网络的分类速度较慢，因为需要得到多个网络的答案（虽然每个网络比较小），从时间上看，整体趋势是模块大小越大，分类器数量越少，分类时间就越短。

从 ROC 来看，几种方法的性能都很好，说明本问题线性可分。

5 任选：MLP

5.1 算法

多层感知机 (Multilayer Percetron, MLP) 是一种前向结构的人工神经网络，映射一组输入向量到一组输出向量。MLP 是感知机的推广，克服了感知机不能对线性不可分数据进行识别的弱点。

神经网络参数如下：

隐含层数量 1

隐含层大小 100

激活函数 ReLU

梯度下降算法 adam

5.2 程序说明

在原来系统框架的基础上实现了新的 Model - MLPModel, 在 `model/MLP_model.py` 中。利用 python 的机器学习库 **scikit-learn** 提供的 `MLPClassifier`，我们只需要做一层简单的封装，以满足框架的调用。

```
    def train(self, X, y):
        # need to transform the input data after dividing (violates the
    csr_matrix)
        X = vstack(X, "csr")
        self.model.fit(X, y)

    def predict(self, X, y=[], options=""):
        label = self.model.predict(X)
        val = self.model.predict_proba(X)
        select = 0 if self.model.classes_[0] == 1 else 1
        val = [v[select] for v in val]
        return label, val
```

如上图，为了能使用 svm 格式的稀疏矩阵，在使用 `svm_read_problem` 时要将其参数 `return_scipy` 置为 True，从而得到 `csr_matrix` 的稀疏输入数据集。还有一个问题是，在对数据集划分后，破坏了原来的稀疏矩阵的格式，在训练时，需要重新用 `vstack` 来重组。

5.3 结果及分析

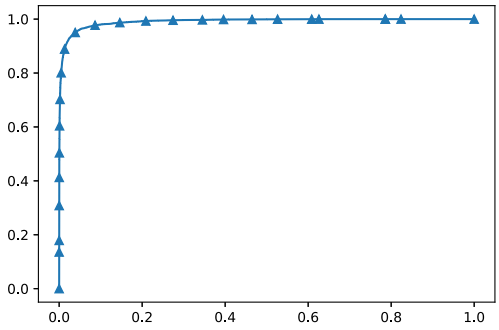
表 3: 不同 MLP 分类器分类效果

分类器	模块大小	accuracy	precision	recall	F1
MLP		96.47	93.67	91.63	92.64
MLP + 随机划分					
	2500	95.66	89.64	**92.81**	91.20
	5000	96.01	91.04	92.67	91.84
	10000	96.23	92.03	92.44	92.24
MLP + 先验划分					
	2500	96.52	93.61	91.90	92.75
	5000	**96.71**	93.38	92.68	**93.18**
	10000	96.62	**93.70**	92.22	92.96

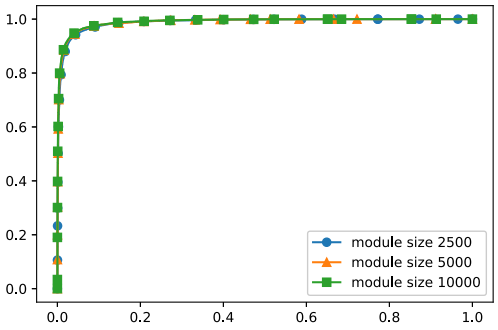
表 4: 不同 MLP 分类器训练时间和分类时间

分类器	模块大小	分类器数量	单个分类器训练时间/s	分类时间/s
MLP		1	198.12	4.48
MLP + 随机划分			10 iterations	
	2500	420 (12×35)	25.4	8.53
	5000	108 (6×18)	56.42	8.24
	10000	27 (3×9)	110.01	9.01
MLP + 先验划分			10 iterations	
	2500	420	16	7.99
	5000	108	38	8.14
	10000	27	104.23	6.40

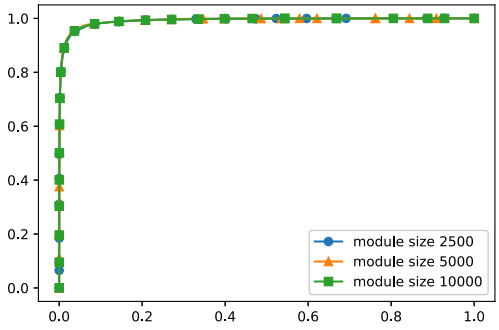
因为本来该问题线性可分度就比较好,所以加上 MLP 之后 accuracy,precision,recall 和 F1 变化并不大。但是训练时间增加的比较多，说明用这种方法训练代价还是比较大的。但是分类时间和原来方法相比变化并不大。



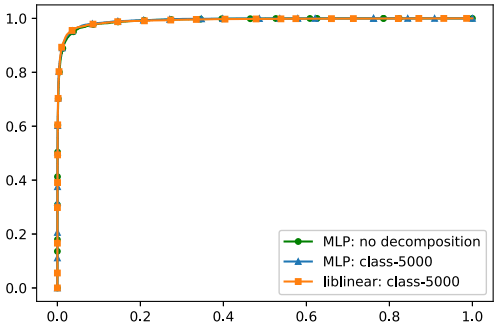
(a) MLP: no decomposition



(b) MLP: random decomposition



(c) MLP: class decomposition



(d) 与 liblinear 比较

图 5: ROC 曲线