EI328 Final Project

- Patent Classification Using Min Max Modular -

Project Report

陈怡峰 5140309458
王君 5140719030

Shanghai Jiao Tong University
Computer Science and Techonolgy

Computer Science And Technology
Shanghai Jiao Tong University

Title:
EI328 Final Project

Theme:
Patent Classification Using Min Max Modular

Project Period:
Spring 2017

Project Group:
Group 8

Participant(s):
陈怡峰
王君

Supervisor(s):
Prof. BaoLiang Lv

Page Numbers: 20

Date of Completion:
September 9, 2017

项目概要:

本项目旨在利用机器学习实现对专利的分类. 我们使用了 LibLinear,LibSVM 等分类器, 以及 Min Max Modular 的学习结构, 对容量为 100k 级别的 5000 维样本数据进行了训练. 训练过程中, 我们分别使用了 MPI,pthread 等并行库来实现加速, 并在性能更加强大的服务器上验证了并行效果. 最终, 我们在 30k 的测试数据集上达到了 96% 的准确率.

报告结构:

本报告将按照大作业问题的顺序进行分析和解答. 受限于 python 的执行效率, 我们分别使用 python 和 c 实现了上述任务. 由于篇幅问题, 主要列出部分 python 核心功能代码,C 语言详见附带代码.

实验平台:

Ubuntu 16.10 + 2 cores + 8G + Python
Ubuntu 16.04 + 2 cores + 12G + C
Ubuntu 16.04 + 8 cores + 16G + C

Contents

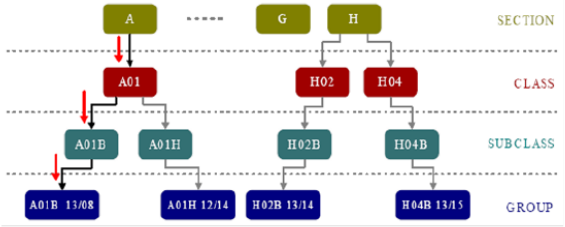
Chapter 0

项目介绍

0.1 项目背景

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。随着专利申请越来越多，对专利进行人工分类耗费的成本越来越高，难度越来越大，对于专利有根据的自动分类方法的需求日益增长。同时，随着对机器学习研究的不断深入，专利分类这种大规模、数据分布系数的线性分类问题，逐渐有了解决办法。通过求解该问题，不仅可以为专利分类问题提供一种解决方案，节省人力物力成本，也可以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。

现有的专利分类问题，具有从上到下依次为 Section，Class，Subclass 和 Group 的分类结构。



本项目要解决的是一个最顶层（Section 层）的二分类问题，利用 LIBLINEAR,LIBSVM 等分类器，使用 Min-Max-Modular 结构对日文专利数据集的一个子集在 Section 层进行了二分类。

0.2 数据集

本项目包含两个数据集, 其中:

Dataset Insight					
Dataset	Total	Class A	Class B	Class C	Class D
train.txt	113128	27876	59597	23072	2583
test.txt	37786	9150	20148	7664	824

数据集中, 数据由两部分组成. 第一部分是若干个形如 $A01C/21/00$ 的标号数据. 第二部分是由 {序号: 值} 稀疏表示的特征向量. 所有特征已经提前做好了归一化.

0.3 评估指标

在本项目中, 我们使用准确率,F1,ROC 曲线来评估模型.

0.3.1 准确率

即预测正确的概率

0.3.2 F1

F1 考虑了测试的精度 p 和回忆率 r 来计算得分：p 是正数结果的数量除以所有阳性结果的数量，r 是正数结果的数量除以阳性数应该返回的结果。F 1 分数可以解释为精度和召回的加权平均数, 往往用作信息检索测量搜索，文档分类和查询分类的性能衡量.

$$p = \frac{TP}{TP + FP}$$

$$r = \frac{TP}{TP + FN}$$

$$F1 = \frac{2rp}{r + p}$$

0.3.3 ROC

使用回归模型进行分类任务时, 我们会对样本产生实值估计，然后将预测值与一个阈值比较来实现分类操作。为了绘制 ROC 曲线，引入阈值 t，改变分类器预测时对正负类的倾向，同时需要计算在不同的阈值之下真正类率 TPR 和假正类率 FPR. 统计不同的阈值下 TPR 与 FPR，分别为横纵坐标绘制得到 ROC 曲线。一般而言,ROC 曲线越平滑, 其面积越大, 则模型越佳.

Chapter 1

串行分类

1.1 问题描述:

使用 LibLinear 直接学习上述两类问题, 用常规的单线程程序实现.

1.2 解决思路:

为了对专利的最顶层进行分类，把 A 类作为正类，其他作为负类. 由于原项目中包含多个标号, 且标号信息与 LibLinear 的标准输入不符. 故首先需要对训练数据和测试数据进行预处理，将标号为 A 类的改为 1 类，将其他类别改为-1 类。

1.3 核心代码:

1.3.1 Python Interface

transFile: 将输入文本转换为标准的 LibLinear 输入文件形式

```
def transFile(filename):
  fw=open('trans_'+filename,'w')  #创建转换后文件，准备写入
  fr=open('../'+filename,'r')     #读取源文件
  for line in fr:                 #遍历每行
    index, data=line.split(' ',1) #以空格拆分标签和特征向量
    if index[0]=='A':
      fw.write(str(1))            #如果标签为A，写入1
    else:
      fw.write(str(-1))           #其余写入-1
    fw.write(' ')
    for i in range(len(data)):    #写入特征向量
      fw.write(data[i])
  fw.close()
  fr.close()
```

main.py: 训练 & 预测:

```
label, data = svm_read_problem("trans_train.txt")
prob = problem(label, data)
param = parameter('-s 0 -c 4')
m = train(prob,param)
test_label, test_data = svm_read_problem("trans_test.txt")
p_label, p_acc, p_va = predict(test_label, test_data, m, "-b 0")
```

1.4 结果

1.4.1 速度测试

Time Evaluation		
Interface	Train Time(s)	Predict Time(ms)
Python	18.4	0.14
C	0.63	0.04

1.4.2 准确率测试

Accuracy Evaluation		
Method	Accuracy	F1
LibLinear-all	96.44%	0.925

1.4.3 ROC 曲线

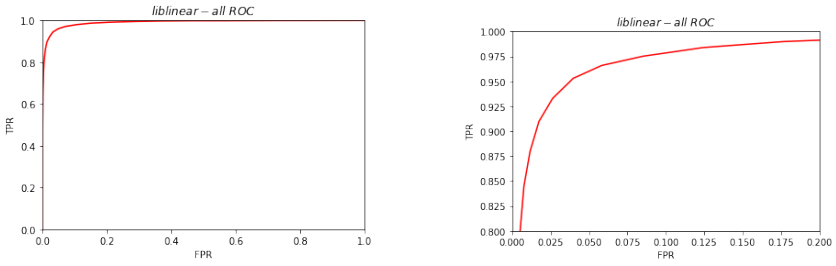


Figure 1.1: liblinear-all ROC cureve

Chapter 2

无先验 min-max-modular 分类

2.1 问题描述:

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。

2.2 解决思路:

在进行 label 的改写后, 一共有 27876 个正样本,85252 个负样本, 可见存在严重的样本不均衡问题. 因此, 我们考虑使用 Min Max Modular 将问题分为较为均衡的子问题进行训练. 在划分子问题的过程中, 我们不考虑子问题已知的先验信息, 而是将所有的 2,3,4 类都视为负类, 随机划分为相同大小的子问题.

在这个问题的解决上，因为每个子问题的训练和预测过程是互不相干，不受约束的, 因此可以利用多进程或者多线程的变成来提高学习效率，缩短程序运行时间。由于多线程可以在线程间共享数据, 我们就免去了多进程间发送训练数据, 测试数据的时间, 因此多线程更加适合用于解决我们的问题. 在 C 实现的代码中, 我们使用 pthread 库实现了多线程. 在 Python 实现的代码中, 由于 Python 对于多线程有 GIL (Global Interpreter Lock) 机制，导致在多核机器上运行多线程代码时，同一时刻只运行一个线程。因此我们分别尝试了 Python 自带的进程池功能以及 MPI 库提供的多进程机制, 实现了多进程的 min-max-modular 分类器.

2.3 核心代码:

2.3.1 Python Interface

decompose: 将训练集拆分为 48 个子问题，并存储对应文件

```
def decompose():
  fw=[]
  for i in range(4):
```

```
    fw.append([])
    for j in range(12):
      fw[i].append(open('de_train_'+str(i)+'_'+str(j)+'.txt','w'))

  fr=open('../train.txt','r')
  for line in fr:
    index, data=line.split(' ',1)
    if index[0]=='A':
      i=randint(0,3)
      for j in range(12):
        fw[i][j].write(str(1)+' ')
        for n in range(len(data)):
          fw[i][j].write(data[n])
    else:
      i=randint(0,11)
      for j in range(4):
        fw[j][i].write(str(-1)+' ')
        for n in range(len(data)):
          fw[j][i].write(data[n])
  for i in range(4):
for j in range(12):
fw[i][j].close()
fr.close()
```

main.py -min_vote(i): 对正类第 i 个子模块的子问题进行最小模块处理. 最大化模块处理与之类似, 不再赘述.

```
def min_vote(i):
  fw=open("min_result_"+str(i)+".txt",'w')
  fr=[]
  for j in xrange(12):
    fr.append(open('result_'+str(i)+'_'+str(j)+'.txt','r'))
  for line in fr[0]:
    result=str(line)
    for k in xrange(1,12):
      temp=str(fr[k].readline())
      if temp<result:
        result=temp
  fw.write(str(result))
  for j in xrange(12):
    fr[j].close()
  fw.close()
  result_index=svm_read_problem("")
```

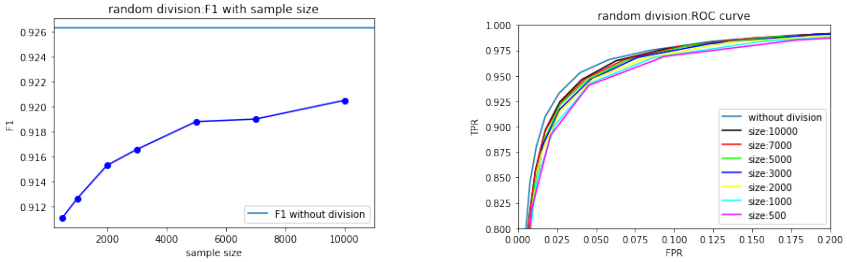
创建进程池:

```
if __name__=="__main__":
pool=Pool(processes=2)
for i in xrange(4):
for j in xrange(12):
pool.apply_async(train_module_roc,(i,j,k))
pool.close()
```

```
pool.join()
pool=Pool(processes=2)
for i in xrange(4):
pool.apply_async(min_vote,(i,))
pool.close()
pool.join()
max_vote(k)
```

2.4 选择子问题规模

为了决定划分子问题的规模, 我们对不同的子问题规模进行了实验:



(a) F1-model size (b) ROC-model size

Figure 2.1: liblinear-random:model selection

我们可以清晰地看到, 随着子问题规模的增大, 模型对整体的表征能力增强了, 即其 F1 值,ROC 曲线面积都在随着子问题规模增大而增大. 但这种增强的关系在后期逐渐变缓, 即选择模型大小为 5000 和模型大小为 10000 相差不大. 为了提高模型的鲁棒性, 增强并行效果, 我们选择了子问题规模为 5000 作为一个折中的选择.

2.5 结果

选择子问题规模为 5000 时, 我们将一共训练 75 个模型, 每个子问题由 5000 个正样本和 5000 个负样本构成, 最终测试结果如下:

2.5.1 速度测试

Time Evaluation		
Method	Train Time(s)	Predict Time(ms)
Python + Serial	122.3	13.04
Python + MPI	190.8	8.67
C + serial	3.89	0.11
C + Pthread	1.98	0.07

2.5.2 准确率测试

Accuracy Evaluation		
Method	Accuracy	F1
LibLinear-random	95.33%	0.918

2.5.3 ROC 曲线

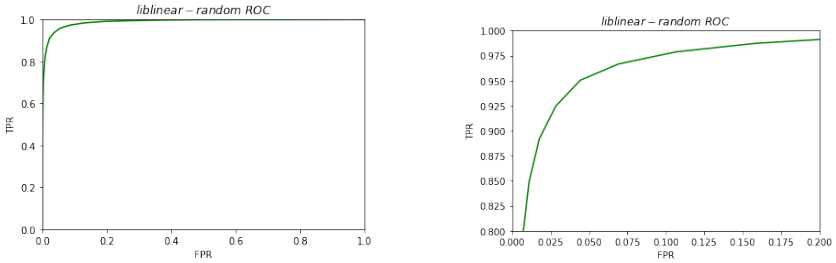


Figure 2.2: liblinear-random ROC cureve

Chapter 3

有先验 min-max-modular 分类

3.1 问题描述:

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, 用基于 min-max-modular 的无先验并行分类多线程或多进程程序实现。

3.2 解决思路:

在上一章节中, 我们在切分子问题时将 2,3,4 类一并视作负类, 随机切分. 即切分的过程破坏了问题原先存在的结构. 因此, 在本章节, 我们将使用基于先验的分类方法, 即按照标号信息划分子问题. 我们使用了两种不同的先验切分策略:1. 按照较细颗粒度标签直接划分问题 2. 按照粗颗粒度标签信息划分子问题后, 在子问题内随机切分.

策略 1: 如何选取先验信息的颗粒度成为首先需要考虑的问题。为此，我们遍历训练集得到使用前 i-1 位作为先验标号信息时，各个类别的标号和样本数目信息。考虑到拆分子问题时，一方面，子问题不能太多，以免子问题样例过少造成误差太高，另一方面，子问题不能太少，以免体现不出子问题规模缩小的优势。经过多次实验，我们发现取标号取标号信息前两位时，进行子问题拆分最佳。(对于有多个标号的样本，统一采取第一个标号进行分类。) 分类信息如下:

A		B		C		D	
A0	3721	B0	6752	C0	15488	D0	2312
A2	2105	B2	14106	C1	2117	D2	271
A4	6599	B3	1601	C2	5033		
A6	15550	B4	12068	C3	434		
		B6	24828				
		B8	242				

策略 2: 直接按照 A,B,C,D 将问题划分为 {A,B},{A,C},{A,D} 三个子问题. 再在这 3 个子问题中随机切分, 形成更多孙子问题。对于孙子问题的规模, 我们选择了从 500-2000 的若干规模进行了测试.

3.3 核心代码:

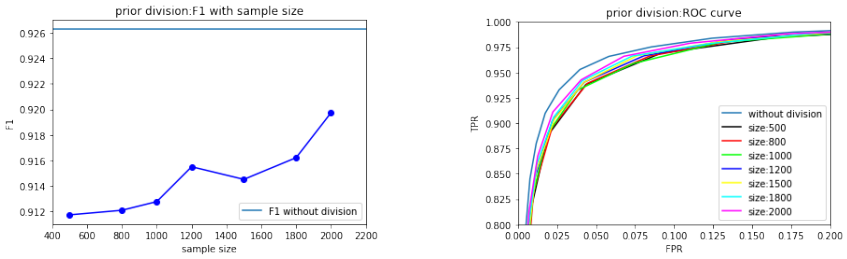
3.3.1 Python Interface

decompose: 按照策略 1 划分问题

```
def decompose():
  fw=[]
  count=0
  for i in range(4):
    fw.append([])
    for j in range(12):
      fw[i].append(open('de_train_'+str(i)+'_'+str(j)+'.txt','w'))
  fr=open('../train.txt','r')
  classPLUS={"A0":0, "A2":1, "A4":2, "A6":3}
  classMINUS={"B8":0, "B3":1, "B4":2, "B6":3, "B0":4, "B2":5, "C3":6, "C2"
    :7, "C1":8, "C0":9, "D2":10, "D0":11}
  for line in fr:
    temp=line[0:2]
    index, data=line.split(' ',1)
    if classPLUS.has_key(temp):
      count=classPLUS[temp]
      for j in range(12):
        fw[count][j].write(str(1)+' ')
        for n in range(len(data)):
          fw[count][j].write(data[n])
    elif classMINUS.has_key(temp):
      count=classMINUS[temp]
      for j in range(4):
        fw[j][count].write(str(-1)+' ')
        for n in range(len(data)):
          fw[j][count].write(data[n])
    for i in range(4):
      for j in range(12):
        fw[i][j].close()
  fr.close()
```

3.4 选择子问题规模

对于策略 1, 我们无需决定子问题规模. 对于策略 2, 我们同样地对不同的模型大小进行了测试:



(a) F1-model size (b) ROC-model size

Figure 3.1: liblinear prior:model selection

同样地, 我们也发现了和第二问相同的规律, 即子问题的规模越大, 其 F1 与 ROC 曲线越好. 在这里, 收到 D 类型大小的限制, 我们选择了 2000 作为策略 2 的子问题规模.

3.5 结果

在选择策略 2 子问题规模为 2000 的情况下, 我们进行了测试:

3.5.1 速度测试

Time Evaluation		
Method	Train Time(s)	Predict Time(ms)
Python + Serial	247.3	45.44
Python + MPI	154.6	31.04
C + serial	6.33	0.56
C + pthread	3.68	0.28

3.5.2 准确率测试

Accuracy Evaluation		
Method	Accuracy	F1
$llblinear - prior_1$	96.575%	0.927
$llblinear - prior_2$	96.1%	0.920

3.5.3 ROC 曲线

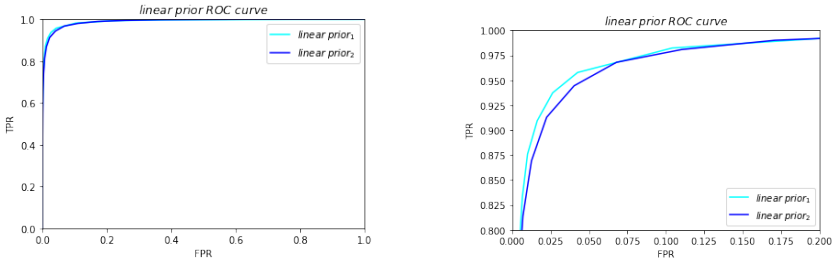


Figure 3.2: liblinear-prior ROC cureve

Chapter 4

模型比较

4.1 问题描述:

本章节对应第 4, 第 5 问. 将从准确度, 运行时间等对模型进行分析和评估.

4.2 模型比较

4.2.1 速度测试

为了方便评估, 我们统一采用并行条件下, 基于 C 实现的分类器速度:

Time Evaluation		
Method	Train Time(s)	Predict Time(ms)
liblinear-all	0.63	0.04
liblinear-random	1.98	0.07
liblinear-prior	3.68	0.27

同时, 我们还测量了训练一个子模型的速度: 对于一个 10000 样本容量 (5000+5000) 的模型, 训练时间在 0.11s 左右. 对于 4000 样本容量 (2000+2000) 需要 0.056s. 可见, 由于最大最小模块分类器中, 每个子问题的规模均比原问题的规模小, 故训练时间均比直接训练的时间短很多。即若考虑完全并行, 则对于上述表格中的 liblinear-random 只需要 0.11s 就能完成训练..

在实际运行中, 考虑到并不存在完全并行的情况, 并且线程的创建删除等带来了额外开销, 使得最大最小模块化分类器的运行时间不如 liblinear-all。值得一提的是, 这一差距将在动态语言上被显著放大, 对于 Python 而言,liblinear-random 的预测时间会比 liblinear-one 的 Python 实现慢 60+ 倍 (这是因为我们申请了巨大的数组来存放 min,max 值, 这类数组在动态语言中被创建和删除的效率很低).

4.2.2 准确度比较

Accuracy Evaluation		
Method	Accuracy	F1
$liblinear-all$	96.4%	0.925
$liblinear-random$	95.3%	0.918
$liblinear-prior_1$	96.57%	0.927
$liblinear-prior_2$	96.1%	0.920

可以看到,liblinear-prior 的准确率和 F1 值比起 liblinear-random 有了比较明显的提升, 这证明基于先验地划分子问题能保持问题的结构, 使得分类更加准确.$liblinear-prior_1$ 即细粒度的先验划分在准确率和 F1 值上已经打败了 liblinear-one, 但是 $liblinear-prior_2$ 在准确率和 F1 上依然比不上 liblinear -all. 即有 $F1_{prior_1} > F1_{all} > F1_{prior_2} > F1_{random}$.

4.2.3 ROC 曲线比较

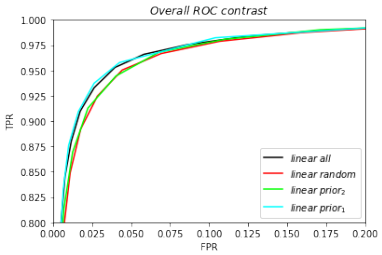


Figure 4.1: ROC Comparison

同样地,$liblinear-prior_1$ 比所有其他方法 ROC 曲线面积更大, 性能更佳, 即 $liblinear-prior_1$>liblinear-all>$liblinear-prior_2$ > liblinear-random.

4.3 结论

$liblinear-prior_1$ 在各项指标中都胜出了, 其性能更佳. 对于拥有计算资源不足的用户,liblinear-all 比较适合. 对于拥有充足计算资源的用户,liblinear-prior 可以在保证预测准确性和泛化能力的前提下更好地完成分类.

Chapter 5

使用 LibSVM

5.1 问题描述:

使用 SVM 多项式核, 完成上述第 2 和第 3 个任务, 并与 LIBLINEAR 作比较。

5.2 解决思路:

SVM 相较于 LibLinear 线性分类器, 其训练时间将大大增长. 考虑到 Python 语言对象创建和释放导致的低效, 我们对于 LibSVM 的实现是基于 C 语言实现的.LibSVM 和 LibSVM 都是由台湾林智仁的教授团队开发的, 因此两者的代码接口都较为相似, 只需要对代码作少许改动即可. 然而,LibSVM 相比起 LibLinear 有更多的超参数, 如何调整参数成为了最首要问题. 由于笔记本计算能力的限制, 我们从原训练数据中抽取了 400 条均匀包含各个类的数据, 选择参数使得 SVM 在现在这一个小的训练集上打到较高准确率. 这种训练毫无疑问是过拟合的, 但是其对我们的调参具有启示性的作用, 我们可以按照其找到的参数进行进一步细调. 最后, 我们确定的参数为"-c 2014 -g 0.000048828125".

5.3 核心代码:

thread_train: 用于多线程训练子模型的函数. 本项目中采用的多线程任务分配方式是: 将正样本均分给每个子线程, 子线程对其分到的正样本和所有负样本进行训练, 训练完成后, 将模型存在 model_list 中. 这样做有效地保证了子线程的任务分配均衡, 最大限度地发挥并行性能.

```
void *thread_train(void *thread_param){
//some trifles are omitted
  for (int it = step*start;it < upper;it++){
    for(int it1 = 0;it1<split_negative;it1++){
      mini_prob.l = size[0] + size[1];
      mini_prob.y = Malloc(double,mini_prob.l);
```

```
      mini_prob.x = Malloc(struct svm_node*,mini_prob.l);
      //generate mini prob's x
      mini_maker(mini_prob,x_dist[0],it*size[0],size[0],x_dist[1],it1*size
    [1],size[1]);
      mini_model = svm_train(&mini_prob, &param);
      printf("%d %d model is trained!\n",it,it1);
      model_list[it][it1] = mini_model;
      free(mini_prob.y);
      free(mini_prob.x);
    }
  }
  pthread_exit(NULL);
}
```

thread_test: 用于多线程测试子模型的函数. 同样地, 为了保证子线程的任务分配均衡, 我们将测试数据均分给子线程, 子线程对其分配的数据调用所有模型, 使用 min-max-modular 求解.

```
  int start = *(int *)thread_param;
  int step = test_num / PRED_PTHREAD;
  int upper = step * (start + 1);
  if(start == PRED_PTHREAD-1){
    upper = test_num;
  }

  for(int i=start*step;i<upper;i++){
    double temp;
    double max = -999999;
    for(int it = 0;it<split_positive;it++){
      double min = 999999;
      for(int it1 = 0;it1<split_negative;it1++){
        svm_predict_values(model_list[it][it1],test_x[i],&temp);
        if(temp < min){
          min = temp;
        }
      }
      if(min > max){
        max = min;
      }
    }
  pred_y[i] = max;
  }
  pthread_exit(NULL);
}
```

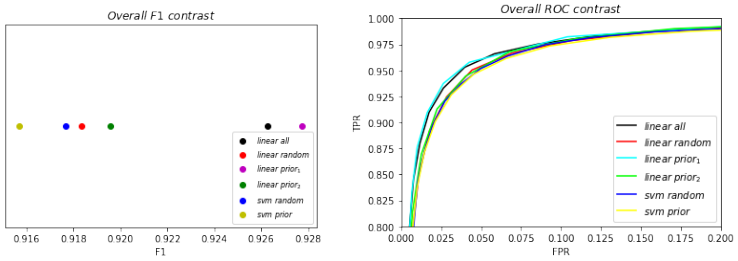
5.4 结果

5.4.1 准确率

Accuracy Evaluation		
Method	Accuracy	F1
LibSVM-random	96.01%	0.915
LibSVM-prior	96.14%	0.917

两者的准确率大致相同, 其中 LibSVM-prior 相对高一些, 符合我们的期望. 值得注意的是 libsvm-random 的准确率明显比 liblinear-random 高, 说明 SVM 比起线性模型的确复杂度更大, 表征能力更强.

5.4.2 F1 & ROC



(a) F1-method (b) ROC-method

Figure 5.1: SVM Comparision

使用 LIBSVM 实现的 Min-Max-Modular 的 F1 值不如所有使用 LibLinear 时的得分. 值得注意的是,libsvm-prior 的 F1 得分甚至不如 libsvm-random, 这可能是参数尚未设置最优导致的. 而就 ROC 曲线而言,libsvm-random,prior 的曲线和 liblinear-random,$liblinear - prior_2$ 大致相当, 都不如 $liblinear - prior_1$.

Chapter 6

使用服务器

6.1 问题描述:

使用服务器实现上述第 1 至第 3 个任务。

6.2 解题思路:

在使用 LibLinear 等快速分类器时, 笔记本的性能足以处理. 但是, 在使用 LibSVM 等复杂分类器处理 100k*5000 的海量数据时, 我们的笔记本就捉襟见肘了. 因此, 我们租用了网易云提供的服务器. 服务器的信息如下:Ubuntu 16.04 + 8 core + 16G + 20G SSD. 为了充分发挥其计算能力, 我们使用了 5 个线程进行训练,8 个线程进行预测.

6.3 运行时间:

LibLinear

Time Evaluation				
Method	ModelNum	ModelSize	TrainTime(s)	PredictTime(ms)
Laptop	310	2000	3.68	0.27
Server	310	2000	1.69	0.04
Laptop	9520	500	17.5	9.16
Server	9520	500	6.87	5.99

可见, 使用了高性能服务器后, 得益于主频和核心的增加, 训练时间减少了 200%, 而预测时间减少了 50%-700% 不等. liblinear-prior(ModelNum=310) 对应的时间已经和 liblinear-all 相当.

LibSVM

Time Evaluation				
Method	ModelNum	ModelSize	TrainTime(s)	PredictTime(ms)
Server	310	2000	692.4	33.3
Server	75	5000	692.0	73.2

值得一提的是, 这里两个不同子问题个数和规模的训练时间却几乎相同, 即更多的子问题却不一定会花更多时间 (本身规模小, 训练时间短).

Chapter 7

总结

7.1 项目总结

通过本次项目, 增加了我们对文档分类常用机器学习方法的了解, 深化了我们对并行机器学习、并行程序设计和大规模数据挖掘的理解, 并让我们体验了服务器的部署和使用, 让我们收益良多.

7.2 致谢

感谢吕宝粮教授对本课程的倾力付出, 感谢助教对我们的热心帮助. 希望本门课程越办越好!