机器学习作业一

马凌霄(1501111302)　　李奕(1501214394)

Adult Data Set

摘　　要

本文以UCI Machine Learning Repository的Adult Data Set为数据集，该数据集目前取得的最高分类准确率为86.2%。我们训练k-Nearest Neighbors(KNN）, Logistics Regression(LR)，Support Vector Machine(SVM)三个分类器，分别取得了81.98%，81.23%，83.25%的最好分类准确率。

首先，我们对数据进行了预处理，删除了不完整记录，名词性属性转换成数值型数值，将某些属性值离散化，删除某些属性项等；为了解决正负样本不均匀的情况，三个分类器都采用了样本权重修正，同时为了加快分类速度，KNN采用CUDA的GPU加速，取得了速度比CPU快了4倍的效果，对K值进行了参数择优；LR和SVM分类器调用了python中scikit-learn package, 并对LR的正则方法，正则系数和SVM的核函数，核函数系数，惩罚系数进行了参数择优。

实验结果表明，SVM在adult数据集取得了最好的效果，LR耗时最短，而KNN采用新颖的GPU加速，值得持续探索。但是由于时间有限，还有很多值得思考的地方，比如数据预处理，分类器训练速度优化等。

目　　录

§1 背景介绍

1.1 小组成员和分工

- 马凌霄
  - 学号：1501111302
  - 院系：信息科学技术学院
  - 邮箱：xysmlx@pku.edu.cn
  - 分工：k-Nearest Neighbors，GPU加速kNN，数据预处理，数据可视化，实验报告LaTeX排版
- 李奕
  - 学号：1501214394
  - 院系：信息科学技术学院
  - 邮箱：liyi193328@163.com
  - 分工：Logistics Regression，Support Vector Machine，数据预处理

1.2 数据介绍：Adult Data Set

本文选取UCI Machine Learning Repository中的Adult数据集[1]。
Adult数据集是根据某人的各种信息预测他的收入是否超过50,000/年。
Adult数据集有48842条记录。Adult数据集的每条记录有以下信息：

- age: continuous.
- workclass: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
- fnlwgt: continuous.
- education: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool. education-num: continuous.
- marital-status: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.

[1] https://archive.ics.uci.edu/ml/datasets/Adult

- occupation: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
- relationship: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
- race: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
- sex: Female, Male.
- capital-gain: continuous.
- capital-loss: continuous.
- hours-per-week: continuous.
- native-country: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad&Tobago, Peru, Hong, Holand-Netherlands.

1.3 测试环境

测试使用了两台计算机及其GPU进行测试，硬件配置和编译器版本如表1-5所示。

表 1 计算机1

项目	详细信息
CPU	Core i7-2630QM (2.0GHz, 4 Cores, 6MB L3 Cache)
内存	10GB DDR3 1333MHz
测试所用磁盘	480GB Sandisk Extreme Pro SSD (Read: 550MB/s)
操作系统	Windows 10 Professional x64

表 2 计算机2

项目	详细信息
CPU	Core i5-4460 (3.2GHz, 4 Cores, 6MB L3 Cache)
内存	16GB DDR3 1600MHz
测试所用磁盘	1TB Seagate 7200RPM HDD (Read: 121MB/s)
操作系统	Windows 10 Professional x64

表 3 GPU1

项目	详细信息
型号	nVIDIA GT550M
流处理器	1480 MHz× 96 Cores
显存	2GB DDR3 900MHz
显存位宽	128bit

表 4 GPU2

项目	详细信息
型号	nVIDIA GTX745
流处理器	1033 MHz× 384 Cores
显存	4GB DDR3
显存位宽	128bit

表 5 编译器版本

项目	版本
C/C++	Microsoft Visual Studio 2013
C/C++	GNU C++ 4.8
Python	Python 3.4
GPU	CUDA 7.5

1.4 评价指标

本文选取精确度（Precision）、准确率（Accuracy）、召回率（Recall）、转移性（Specificity）、F-measure这五个指标作为评价指标。

假设原始样本中有两类，其中：

- 总共有P个类别为1的样本，假设类别1为正例。
- 总共有N个类别为0的样本，假设类别0为负例。

经过分类后：

- 有TP个类别为1 的样本被系统正确判定为类别1，FN 个类别为1 的样本被系统误判定为类别0，显然有$P = TP + FN$；
- 有FP 个类别为0 的样本被系统误判断定为类别1，TN 个类别为0 的样本被系统正确判为类别0，显然有$N = FP + TN$；

定义1. 精确度（Precision）：

$$P = \frac{TP}{(TP + FP)}$$

反映了被分类器判定的正例中真正的正例样本的比重。

定义2. 准确率（Accuracy）：

$$A = \frac{(TP + TN)}{(P + N)} = \frac{(TP + TN)}{(TP + FN + FP + TN)}$$

反映了分类器统对整个样本的判定能力——能将正的判定为正，负的判定为负。

定义3. 召回率(Recall)，也称为True Positive Rate:

$$R = \frac{TP}{(TP + FN)} = 1 - \frac{FN}{T}$$

反映了被正确判定的正例占总的正例的比重。

定义4. 转移性（Specificity），也称为True Negative Rate：

$$S = \frac{TN}{(TN + FP)} = 1 - \frac{FP}{N}$$

明显的这个和召回率是对应的指标，只是用它在衡量类别0的判定能力。

定义5. F-measure：

$$F = \frac{2 * \text{召回率} \times \text{准确率}}{(\text{召回率} + \text{准确率})}$$

§ 2 数据处理

2.1 数据预处理

由于仅使用原数据在各个算法中测试的结果均不太好，所以需要对数据进行预处理，根据多次调整数据预处理并且进行实验，得出最佳的数据预处理方法如下所示：

- 将字符串标签转化为$0, 1, 2, 3$这样的标签。
- 删去不完整的记录。
- 删去fnlwgt、education-num、marital-status、capital-gain、capital-loss、native-country。
- 对于hours-per-week
  - 数值≤ 39，标记为0；
  - 数值> 39，标记为1；
- 将age离散化：分为11组分别标记为$0-10$：≤ 20，$21-25$，$26-31$，$32-36$，$37-40$，$41-46$，$47-51$，$52-56$，$57-60$，$61-66$，> 66

经过预处理后，数据集变为：45222条记录，每条记录有8个信息和1个最终标记。数据样例如图1所示。

	A	B	C	D	E	F	G	H	I
1	age	workclass	education	occupation	relationship	race	sex	hours-per-week	target
2	4	1	1	1	1	1	1	1	0
3	6	2	1	2	2	1	1	0	0
4	4	3	2	3	1	1	1	1	0
5	7	3	3	3	2	2	1	1	0
6	2	3	1	4	3	2	2	1	0
7	4	3	4	2	3	1	2	1	0
8	6	3	5	5	1	2	2	0	0
9	7	2	2	2	2	1	1	1	1
10	2	3	4	4	1	1	2	1	1
11	5	3	1	2	2	1	1	1	1
12	4	3	6	2	2	2	1	1	1

图 1 预处理后的Adult数据集

2.2 训练集与测试集划分：10-fold cross-validation

本文使用10折交叉验证（10-fold cross-validation）的方法进行训练集和测试集的划分：将数据集按照$9:1$划分为训练集和测试集，重复10次，每次选取不同的10%作为测试集，其余90%作为测试集，取10次的平均值作为最终的测试结果。

§ 3 k-Nearest Neighbors

3.1 算法简介

k-Nearest Neighbors算法的思想是：如果一个样本在特征空间中的k个最相似（即特征空间中最邻近）的样本中的大多数属于某一个类别，则该样本也属于这个类别。

k-Nearest Neighbors算法的流程如算法1所示。

算法 1: k-Nearest Neighbors

Input: 训练数据集D, 测试数据集T, 阈值K

Output: 预测的标签

```
Function kNN(D,T,K):
    初始化距离为最大值;
    for t ∈ T do
        for d ∈ D do
            计算t与d ∈ D的距离dis;
            sort(dis);
            得到目前K个最临近样本中的最大距离maxdis;
            if dis < maxdis then
                将该训练样本作为K-最近邻样本;
            end
        end
        统计K-最近邻样本中每个类标号出现的次数;
        选择出现频率最大的类标号作为未知样本的类标号;
    end
end
```

使用的距离度量为欧式距离，如式1所示。

$$dis(A,B) = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2} \tag{1}$$

3.2 算法实现

- 语言：C++
- 使用的库：无

k-Nearest Neighbors的各个函数定义如下所示：

```
1  class KNN
2  {
3  public:
4      static const int maxCol = 10;
5      static const int maxRow = 46010;
6
7      static const int testNum = 4000; // Data number for test
8
9  public:
10     struct Node
11     {
12         double data[maxCol];
13         string label;
14     };
15
16     struct CMP
17     {
18         bool operator ()(const prid &A, const prid &B) const
19         {
20             return A.second < B.second;
21         }
22     };
23
24 public:
25     void init(int _k, int _row, int _col, string path); // init
26     void input(); // Input data
27     void ZScoreNorm(); // Z-Score Norm
28     void MaxMinNorm(); // Max min Norm
29     double dis(const Node &A, const Node &B); // Calculate distance
           between A and B
30     void CalDis(); // Calculate distance between dataTest and dataTrain
31     string MaxFreqLabel(); // Calculate freq label
32     void knn(); // Run knn
33     void debug(); // For debug
34
35 public:
36     ifstream fin;
37     string filepath;
38
39     int therK;
40     int row;
41     int col;
42
43     Node dataSet[maxRow];
44     Node dataTest;
```

```
45
46      vector<prid> vec;
47      map<string, int> mp;
48  };
```

变量简介

- struct Node：存储一条记录的结构体。
  - $double\ data[maxCol]$：存储一条记录的每个信息。
  - $string\ label$：存储该记录的标签。
- $ifstream\ fin$：文件输入，用于读取文件。
- $string\ filepath$：输入文件的路径。
- $int\ therK$：kNN的阈值K。
- $int\ row$：记录的条数（行数）。
- $int\ col$：记录的列数。
- $Node\ dataSet[maxRow]$：存储整个记录。
- $vector < prid > \ vec$：存储$int, double$的$pair$，int为该记录在$dataSet$的下标，$double$为该记录与测试数据的距离。
- $map < string, int > \ mp$：用于统计某标签出现的次数。

函数简介

- $void\ init(int\ \_k, int\ \_row, int\ \_col, string\ path)$：初始化所有变量
  - 输入：kNN的阈值$_k$，数据集的行数$_row$，数据集的列数$_col$，数据文件路径$path$
  - 输出：无，初始化好的变量
- $void\ input()$：使用ifstream从数据文件读取数据
- $void\ ZScoreNorm()$：Z-Score标准化
- $void\ MaxMinNorm()$：Max-Min标准化
- $double\ dis(const\ Node\ \&A, const\ Node\ \&B)$：按照式1计算数据记录A和B的距离

  - 输入：数据记录A，数据记录B
  - 输出：A和B的距离

- *void CalDis()*：计算测试数据记录与所有训练数据记录的距离
  - 输入：测试数据记录，训练数据记录集合
  - 输出：测试数据记录与所有训练数据记录的距离，用vector输出
  - 流程：遍历一遍训练数据记录集合，调用dis函数计算所有的距离

- *string MaxFreqLabel()*：找出出现次数最高的label，
  - 输入：测试数据记录，训练数据记录集合
  - 输出：预测得到的标签
  - 流程：
    1. 调用$CalDis()$计算未知样本和每个训练样本的距离dis
    2. 得到目前K个最临近样本中的最大距离$maxdis$
    3. 如果dis小于$maxdis$，则将该训练样本作为$K-$最近邻样本
    4. 对于每个训练样本，执行1-3步
    5. 统计$K-$最近邻样本中每个类标号出现的次数
    6. 选择出现次数最大的作为预测标签，return标签

- *void knn()*：kNN主控
  - 流程：
    1. 对于一个测试样本，调用$MaxFreqLabel()$
    2. 判断正确与否，统计TP、TN、FP、FN
    3. 对于每个测试样本，执行1-2步
    4. 计算精确度、准确率、召回率、转移性、F-measure

3.3 GPU加速机器学习& CUDA介绍

随着数据集的增大，目前已有的机器学习算法训练大规模数据所需时间越来越多。而且对于较为复杂的机器学习算法，训练的时间也非常高。由于机器学习算法有着大量迭代等适合并行化的过程而且一般GPU的核心数能达到3072(nVIDIA TITAN X)而主流CPU核心数为4，所以可以使用GPU的大量计算核心加速机器学习算法。

CUDA（Compute Unified Device Architecture，统一计算架构）是由nVIDIA所推出的一种集成技术，是该公司对于GPGPU的正式名称。通过CUDA，用户可

利用nVIDIA的显卡进行GPU计算。CUDA可以兼容OpenCL或者自家的C-编译器。无论是CUDA C-语言或是OpenCL，指令最终都会被驱动程序转换成PTX代码，交由显示核心计算。

使用CUDA进行GPU加速的数据交换主要通过计算机内存和GPU显存之间互相传输数据。所以GPU加速的过程即为：

1. 把数据从内存传输到显存
2. 让GPU对显存中的数据进行计算，把结果写入显存
3. 将计算结果从显存中传输到内存

3.4 GPU加速的算法实现

在这里，GPU加速主要运用于测试样本与训练样本集合的距离计算。设计思想是：把测试样本与训练样本集合的距离计算交给GPU计算，让GPU每个计算核心计算一个测试样本与一个训练样本的距离，再输出到显存，传输给内存。

GPU计算距离的函数设计如下所示：

```
1  __global__ void DisKernel(double *traindata, double *testdata, double *dis
           , int pitch, int N, int D)
2  {
3      int tid = blockIdx.x;
4      if (tid < N)
5      {
6          double temp = 0;
7          double sum = 0;
8          for (int i = 0; i < D; i++)
9          {
10             temp = *((double*)((char*)traindata + tid * pitch) + i) -
            testdata[i];
11             sum += temp * temp;
12         }
13         dis[tid] = sum;
14     }
15 }
```

其余部分和标准kNN的设计一样

3.5 实验

3.5.1 参数K选取实验

K选取3-15，统计精确度、准确率、召回率、转移性、F-measure，如表6和图2所示。

- $K = 6$：此时召回率为0.691445最高，F-measure为0.748997最高。
- $K = 12$：此时准确率为0.81975最高。召回率和F-measure也比较高。设置为最优参数。

表 6 参数K选取实验

k	精确度	准确率	召回率	转移性	F-measure
3	0.585075	0.80225	0.611227	0.862739	0.69383
4	0.473632	0.80975	0.672316	0.839307	0.734661
5	0.597015	0.8065	0.619195	0.866381	0.700544
6	0.490547	0.817	0.691445	0.844235	0.748997
7	0.581095	0.81825	0.65618	0.86463	0.728307
8	0.514428	0.81875	0.685676	0.849661	0.746328
9	0.591045	0.81825	0.652747	0.86699	0.726188
10	0.508458	0.81425	0.672368	0.847531	0.736539
11	0.587065	0.81725	0.651214	0.865869	0.724845
12	0.528358	0.81975	0.682519	0.852886	0.744867
13	0.573134	0.8155	0.650847	0.862279	0.72393
14	0.528358	0.81375	0.662095	0.851782	0.73013
15	0.569154	0.81375	0.647059	0.86104	0.720894

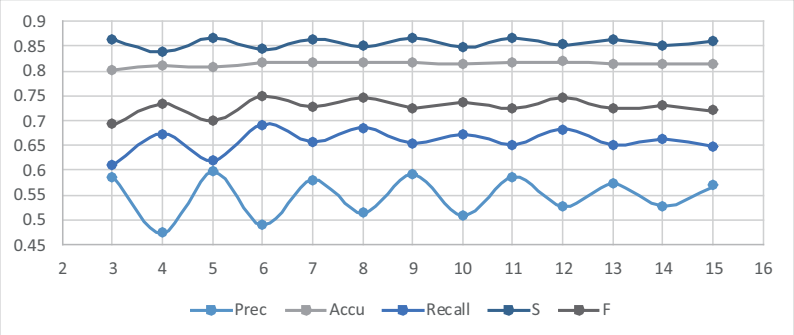

图 2 参数K选取实验

3.5.2 最优参数下的实验

由上一小节可知，选取最优的K值为12，当$K = 12$时，准确率、召回率、F-measure、精确率如表7所示。

表 7 kNN最优参数下的实验

数据集	准确率	召回率	F-measure	精确率	最优参数
原始数据	0.77025	0.589212	0.667676	0.282587	k=7
处理数据	0.81975	0.682519	0.744867	0.528358	k=12

3.5.3 GPU加速的实验

使用表1-5所示的环境，进行2台计算机和2个GPU的对比实验。测得计算所用时间如图3所示，测试发现GPU对kNN的加速程度非常大，即使是入门级的GPU nVIDIA GT550M，所需的时间仅为Core-i7计算时间的1/4。所以GPU能够非常明显地对kNN进行加速。

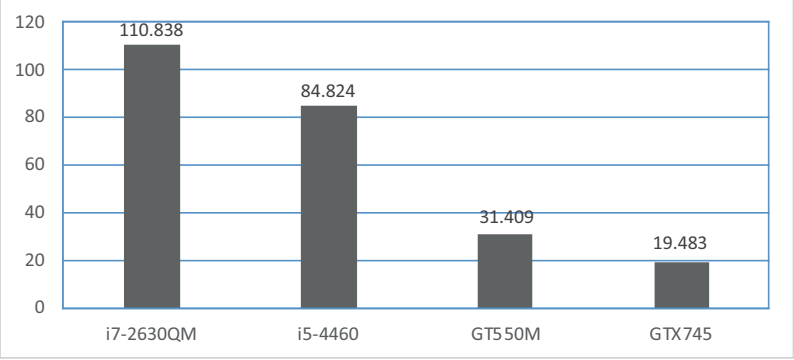

图 3 GPU加速的实验（单位：秒(s)）

§ 4 Logistics Regression

4.1 算法实现

- 语言：Python
- 使用的库：scikit-learn, pylearn2

scikit-learn简述 Logistics回归在scikit-learn中的函数介绍

- 函数默认参数：

```
class sklearn.linear_model.LogisticRegression(penalty='l2', dual=
        False, tol=0.0001, C=1.0, fit_intercept=True,
        intercept_scaling=1, class_weight=None, random_state=
        None, solver='liblinear', max_iter=100, multi_class='ovr
        ', verbose=0, warm_start=False, n_jobs=1)
```

- 重要参数说明：
  - *penalty*：表示正则方式，可用的方式是"l1"或者"l2"分别对应了l1和l2正则方式
  - *C*：逆正则系数，C值越小，代表正则强度越大；反之，越小
  - *class_weight*: 类权重，如果设置为" balanced" ,则类的权重按照正负样本比例，也可以用户自己设定
  - *n_jobs*: 并行线程数目

Logistics Regression核心代码如下所示：

```
log_tuned_parameters = [{'penalty': ['l1'],'C': [1, 5, 10, 100]},
                        {'penalty': ['l2'], 'C': [1, 5 ,10, 100]}]
    print("logistic regression:")
    logS = time.clock()
    logisticRegression = LogisticRegression()
    score = "f1_micro"
    cv =ShuffleSplit(len(trainVec),3, test_size =0.1,random_state=10)
    clf = GridSearchCV(logisticRegression, log_tuned_parameters, cv=cv,
        n_jobs =1,scoring='%s' % score)
    clf.fit(trainVec,trainLabel)
    print(clf.best_params_)
    predict = clf.predict(testVec)
    accuracy = metrics.accuracy_score(testLabel,predict)
    recall =metrics.recall_score(testLabel, predict,pos_label=None,
        average='micro')
    print("accuracy,recall: ",accuracy,recall)
    print(metrics.classification_report(testLabel,predict))
    print("confusion_matrix:")
    cm = metrics.confusion_matrix(testLabel,predict)
    print(cm)
    logE = time.clock()
    print("Logistic Regression time: {0}s".format(logE-logS) )
```

代码解释

- log_tuned_parameters：表示的参数的备选项，在分类时会尝试每一个可能参数的每一个取值情况，并选择给定条件（这里实际上是f1_score或者F-measure值）的最优参数
- trainVec, trainLabel表示的是训练数据的样本向量和标签；testVec, testLabel表示的是测试数据向量和标签
- GridSearchCV函数的目的是在训练数据时自动搜索出给定目标下多个参数的最优参数，它的第一个参数是定义好的分类器，第二个参数是需要调整的参数，它是以字典(dict)的形式传入的，cv表示对训练数据采用交叉验证的方式，可以是整数，也可以是定义好的交叉验证类，此时我们传入的是一个交叉验证方式（首先对数据进行shuffle，按照9:1数据分割，迭代10次）；n_jobs 是并行运算的线程数目；scoring是在选择参数的标准，设定的scoring是f1_score，表示在候选参数中选择使得f1_score最大的参数组合
- clf.fit传入的是训练样本和标签（它会自动训练出最优的参数和分类器），然后用clf.predicty作用在predict数据上，得到predict的label
- metrics.accuracy_score输入是测试数据中准确的标签（label）和预测的标签，得到的是相应的准确率；metrics.recall_score得到是召回率; metrics.classification_report 得到的是precision, recall, f1_score三个指标；cm 表达的是它们的混淆矩阵

4.2 实验

准确率，召回率，F-measure，精确率都是根据正负样本量平均的结果，如表8所示。

表 8 Logistics Regression最优参数下的实验

数据集	准确率	召回率	F-measure	精确率	最优参数
原始数据	0.772937	0.789676	0.7566677	0.768876	C=0.5 penalty=l1
处理数据	0.814563	0.812379	0.824573	0.815431	C=1 penalty=l1

§ 5 Support Vector Machine

5.1 算法实现

- 语言：Python
- 使用的库：scikit-learn, pylearn2

SVM在scikit-learn中的函数介绍：

- 函数默认参数：

```
class sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='auto', coef0=0.0, shrinking=True, probability=False, tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, decision_function_shape=None, random_state=None)
```

- 重要参数说明：
  - C：错误项的惩罚系数
  - $Kernel$: 核函数，'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' 或者自己设置的可调用的核函数
  - $class\_weight$: 类权重，如果设置为"balanced",则类的权重按照正负样本比例，也可以用户自己设定
  - $gamma$: 核函数系数
  - n_jobs: 并行线程数目

SVM核心代码如下所示：

```
svm_tuned_parameters = [{'kernel': ['rbf'], 'gamma': [0.5],'C': [1, 5, 10, 100]},{'kernel': ['linear'], 'gamma': [0.5], 'C': [1, 5, 10, 100]}]
    print("\nsvm:")
    svmTS = time.clock()
    svmKernelway = 'rbf'
    svrrbf = SVC(kernel=svmKernelway,C=10,gamma=0.5)
    score = "f1_micro"
    cv =ShuffleSplit(len(trainVec),2, test_size =0.1,random_state=10)
    clf = GridSearchCV(SVC(C=1), svm_tuned_parameters, cv=cv, n_jobs = 2, scoring='%s' % score)
    clf.fit(trainVec,trainLabel)
    print(clf.best_params_)
```

```
11      testLabel = clf.predict(testVec)
12      accuracy = metrics.accuracy_score(testLabel,predict)
13      recall = metrics.recall_score(testLabel,predict,pos_label=None,
            average='micro')
14      print("accuracy,recall: ",accuracy,recall)
15      print(metrics.classification_report(testLabel,predict))
16      print("confusion_matrix:")
17      cm = metrics.confusion_matrix(testLabel,predict)
18      print(cm)
19      svmTE = time.clock()
20      print("svm time: {0}s".format(svmTE-svmTS) )
```

代码解释 在scikit-learn中的svm分类器的调用和logistics回归调用结构几乎一致，只是在参数类别含义不一致，因此代码和logistics回归代码相比较，主要区别除了分类器名字的差别外，其余就是参数设置，svm_tuned_parameters：此处我们对gamma取定为0.5或者1，C值设定为1,5,10,100，在这些参数组合中选择使得f1_score最大的参数组合。

5.2 实验

问题 在参数训练过程中，由于数据切割不均匀导致训练样本或者测试样本中正负样本比例不均匀的情况，这样会产生某一类的召回率特别低（有时还会出现0的情况）,如果仅仅只看正负样本召回率的平均值，就会错误地认为分类效果还不错，但是这种情况是畸形的。

解决办法 设定class_weight参数为"balanced"或者自己设定一个正负样本的权重值，让样本多的一方权重稍微低一些，样本少的一方权重高一些，而balanced参数是根据正负样本的数量来设定权重的，我们的程序里面class_weight设定就是balanced。

准确率，召回率，F-measure，精确率都是根据正负样本量平均的结果，如表9所示。

表 9 SVM最优参数下的实验

数据集	准确率	召回率	F-measure	精确率	最优参数
原始数据	0.802537	0.799656	0.806777	0.818876	C=1 gamma=0.5 kernel=rbf
处理数据	0.824663	0.832479	0.835473	0.834431	C=10 gamma=0.5 kernel=rbf

§6 综合测试对比

选取准确率，召回率，F-measure，精确率作为评价指标，kNN、Logistics Regression和SVM的测试结果如表10和图4所示。

k-Nearest Neighbors(KNN）, Logistics Regression(LR)，Support Vector Machine(SVM)三个分类器，分别取得了81.98%，81.23%，83.25%的最好分类准确率。

查阅文献得知，该数据集目前最高的分类准确率为86.2%。

表 10 综合测试对比

算法	准确率	召回率	F-measure	精确率	最优参数
kNN	0.81975	0.682519	0.744867	0.528358	k=12
Logistics	0.814563	0.812379	0.824573	0.815431	C=1 penalty=l1
SVM	0.824663	0.832479	0.835473	0.834431	C=10 gamma=0.5 kernel=rbf

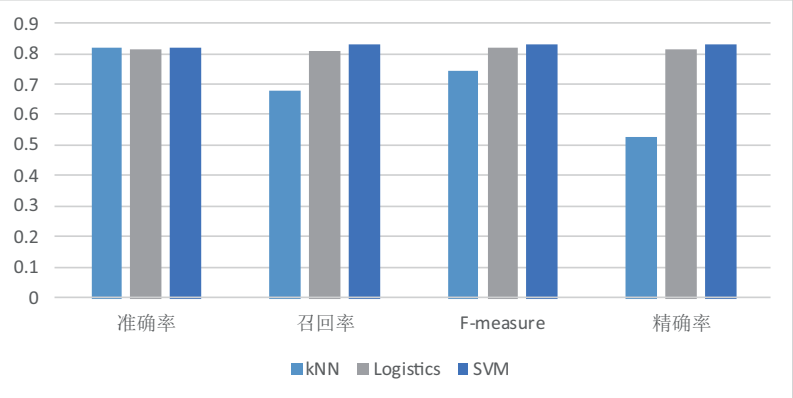

图 4 综合测试对比

§ 7 总结

1. 第一次用实际的数据做分类器训练，对数据预处理或者特征选择有更好的认识：对各种特征的选择或者处理方法只要合理，能够解释，都可以尝试。

2. 通过KNN的GPU实验，学了CUDA编程的基本知识，对并行程序有更好的认识。

3. 了解熟悉了scikit-learn包，学会了调整参数的方法。

4. 由于时间有限，有好多临时结果没有保留下来，重新做一遍又非常耗时，因此只有一个总体的结果，非常遗憾：以后在编写程序过程中注意中间结果的保留!

5. 训练过程是一个复杂的过程，即使一个简单的分类器也有很多值得思考的地方。

参考文献

[1] UCI Machine Learning Repository. Adult Data Set. https://archive.ics.uci.edu/ml/datasets/Adult.

[2] Kohavi R. Scaling Up the Accuracy of Naive-Bayes Classifiers: A Decision-Tree Hybrid[C]//KDD. 1996: 202-207.

[3] 李航. 统计学习方法[J]. 2012.

[4] nVIDIA. CUDA. https://developer.nvidia.com/cuda-downloads.