Data Mining Homework2

Large-scale classification-SYSU-2016

13354433 张楠

一、数据描述

共计三个数据文件，分别为 train.txt, test.txt, sample_submission.csv.

train.txt: 训练数据集，共 2177020 个训练数据，第一列为 label，只有两类用 0 和 1 表示。训练数据的 features 只含有 value 为 1 的项，表示为 index:1。

test.txt: 测试数据集，共 220245 个。与数据集格式一致，第一列 label 替换为 id。

sample_submission.csv: 以 id,label 为格式进行提交。

本实验代码运行环境为 mac os，使用 python2.7 、 clang++

二、问题分析

此次作业训练数据集较大，超过 1G。但是特点在于三万多维的训练数据，value 为 1 的数量只有 80 个左右。因此可以用稀疏矩阵来表示训练数据集。

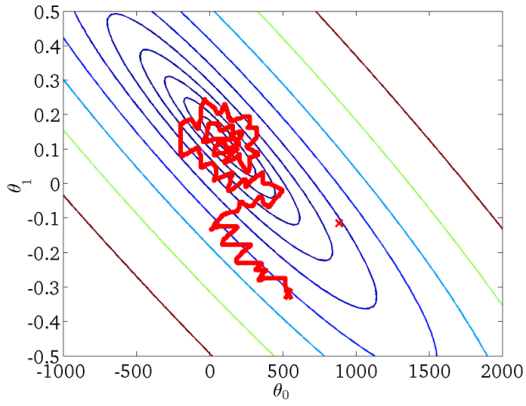
问题是二分类的问题，采取线性回归的方法，由于数据量较大，标准梯度下降的方法收敛速度会很慢，因此才用随机梯度下降的方法。原理是根据某个单独样例的误差增量计算权值更新，得到近似的梯度下降搜索（随机取一个样例）。

随机梯度下降与标准梯度下降的重要差别在于：标准梯度下降是在权值更新前对所有样例汇总误差，而随机梯度下降的权值是通过考查某个训练样例来更新的。

Stochastic gradient descent

1. Randomly shuffle (reorder) training examples

2. Repeat {
   for $i := 1, \ldots, m$ {
   $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
   (for every $j = 0, \ldots, n$)
   }
   }



三、探索过程

经过问题分析，我首先选择 python 作为本次作业使用的编程语言。python 优点在于读写文件方便，有 numpy 等方便的库来对数据进行操作。

数据结构：

在数据存储结构上，最初我使用了三万八千多维的 list，作为 vector 来与 weight 相乘，由于 list 中很多 value 都为 0，做了很多无用的计算，同时浪费内存。这种方法迭代一次需要 4 小时。

经过对 list 中存储数据的改进，效率有所提升，理论上每个数据的计算次数从 38000 减少到 80，但是一次迭代仍需 30 分钟左右。

```
def stochGradAscent(data_matrix, class_labels, num_iter=100):
    m = 2177020
    #m = 20000
    n = 11392
    # m rows = m datas
    # n cols = n dims
    weights = np.zeros(n)
    for j in range(num_iter):
        data_index = range(m)
        for i in range(m):
            if i % 1000 == 0:
                print'num: ', i, ' is dealing'
            alpha = 4 / (1.0 + j + i) + 0.01
            randIndex = int(random.uniform(0, len(data_index)))
            h = sigmoid( weights[ data_matrix[randIndex] ].sum() )
            error = class_labels[randIndex] - h
            weights[ data_matrix[randIndex] ] += alpha * error
            del (data_index[randIndex])
    return weights
```

这里的随机梯度下降函数参考了《机器学习实战》的 chapter5 中的内容。

动态的 learning rate 可以缓解数据波动，同时常数项的存在保证了多次迭代后新数据仍有一定的影响力。同时随机选取样本来更新 weights 可以减少周期性的波动。

由于迭代太过耗时，我提交了一次迭代的结果，正确率在 0.571 左右。通过

python -m profile logisitic_regression.py

运行程序进行分析（使用 20000 个训练数据），发现一些耗时的操作。

```
3429455 function calls (3429367 primitive calls) in 14.147 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        2    0.000    0.000    0.000    0.000 :0(POINTER)
        8    0.000    0.000    0.000    0.000 :0(__contains__)
       21    0.000    0.000    0.000    0.000 :0(_getframe)
        6    0.000    0.000    0.000    0.000 :0(add)
      272    0.001    0.000    0.001    0.000 :0(add_docstring)
        8    0.000    0.000    0.000    0.000 :0(all)
  1650060    2.543    0.000    2.543    0.000 :0(append)
```

```
1608768    2.774    0.000    2.774    0.000 :0(split)
```

在读取数据时，由于没有将数据一次性处理好，每次迭代都需要重新解析数据，占一次迭代时间的 5/14 左右。因此我将训练数据一次性整理成只存 label 和 index 的格式。但是虽然优化了这一部分，耗时的一大部分还是在随机梯度下降的部分。

为了追求高效率，我使用 c++将代码重写，逻辑部分不变。文件读写部分使用了 stringstream，非常方便。

训练数据都存在双重 vector 中，同样的也是仅存储 label 和 index 来提升效率。

```
int main(){

    ifstream readfile;
    readfile.open("/Users/All4win/Documents/Three/DataMining/HW2/train_data.txt",ios::in);
    string line;
    int num;
    char s;
    int count = M;
    while(getline(readfile,line) &&count--)
    {
        istringstream ss(line);

        vector<int> temp_v;
        ss>>num;
        temp_v.push_back(num);
        while(ss>>s>>num){
            temp_v.push_back(num);
        }
        data_matrix.push_back(temp_v);
    }
    readfile.close();
    cout<<"read is over"<<endl;
```

c++中随机数比较复杂，直接使用标准库中的函数讲数据进行洗牌，然后顺序读取便可模拟随机读取：

std::random_shuffle(data_matrix.begin(), data_matrix.end());

C++版本的代码迭代一次仅需要几秒钟，让我不禁感慨 C++效率之高。迭代 50 次后，准确率提升到 0.581，有了一定的进步。

并行化：

由于随机梯度下降，每次随机选择数据时，weight 都会改变，因此 cost 只能在一轮迭代结束后再对数据遍历一次进行计算。由于数据量很大，可以选择多线程来计算。

需要告诉每个线程要计算的 start 和 end，使用 2 个线程的话就传递 0-m/2 和 m/2-m。

向线程传递参数使用结构体来传递：

```
struct thread_data
{
    int start;
    int end;
    int thread_id;
    double *result;
};
```

线程函数开始时需要类型转换，因为参数类型只能为 void＊

```
void *calcCostThread(void *threadarg)
{
   struct thread_data *my_data;
   my_data = (struct thread_data *) threadarg;
```

多个线程将 cost 累加后传递给由结构题 result 指针指向的数组，就可以得到 cost 之和了。当然还是要确保多个线程都执行结束才能进行下一步的动作，因此需要使用 pthread_join() 函数来等待线程的完成。

```
int rc;
pthread_t threads[NUM_THREADS];
pthread_attr_t attr;

thread_data td[NUM_THREADS];
void *status;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for (int t = 0; t < NUM_THREADS; ++t)
{

    td[t].thread_id = t;
    td[t].start = s[t];
    td[t].end = e[t];
    td[t].result = &res_mid[t];

    rc = pthread_create(&threads[t], NULL, calcCostThread, (void *)&td[t]);

    if (rc){
        cout << "Error:unable to create thread," << rc << endl;
        exit(-1);
    }
}

pthread_attr_destroy(&attr);
```

```
for (int t = 0; t < NUM_THREADS; ++t)
{

    rc = pthread_join(threads[t], &status);
    if (rc){
        cout << "Error:unable to join," << rc << endl;
        exit(-1);
    }

}
```

经过多线程的改进后，计算 j(theta)的时间由 2s 减少至 1s。

四、总结

这次作业是我第一次接触到数据量较大的机器学习问题，使用的方法是较为简单的逻辑回归，优点在于实现简单，迭代速度快。但是并行化计算方面，随机梯度下降很难分成多个任务进行，只能在 cost function 的部分进行并行化，效率提升比较有限。但是话说回来，通过这次作业，也确确实实地学到了很多知识。在 python 部分，也踩了许多坑，包括写 list 带括号［］，转为 string 带引号等。收获也不少，例如掌握用数组作为索引来获取别的数组的值的方法。C++中也是初次使用多线程的方法，巩固了文件流读写的部

分，用到了一些不常用的 STL 函数，如 random_shuffle。其实应该使用一些别的方法来尝试，随机森林等方法可以更好的应用并行化的思想，但是由于时间关系未能去尝试了。机器学习是计算机科学的热潮，希望在今后的学习中能更多接触它，提升处理问题的能力。