基于 MapReduce 的软件 Bug 分类

张镓伟 19214895、杨金瑞 19214876、任嘉昕 19214836、陈俊霖 19214945

(中山大学数据科学与计算机学院 广州 510006)

摘 要 在 Github 代码仓库中，存在大量已分类（即加上标签）的软件 bug。但是现在分类标签大都是基于人工添加的，效率比较低。为了提高分类效率，节省人力资源，使得在海量数据的情况下，依然能够快速给软件 bug 执行自动分类，我们利用 MapReduce 分布式编程模型，实现了朴素贝叶斯分类算法，自动地给 Bug 加上标签。实验结果表明，我们的算法能够有效地执行分类功能。我们的代码会在 Github 上公开。

关键词 朴素贝叶斯分类算法、MapReduce、分布式、软件 bug

Software Bug Classification Based on MapReduce

ZHANG Jia-Wei 19214895、YANG Jin-Rui 19214876、REN Jia-Xin 19214836、CHEN Jun-Lin 19214945

（School of Data and Computer Science，Sun Yat-Sen University， Guangzhou 510006）

Abstract There are a large number of classified (i.e., tagged) software bugs in the Github code repository. But now most of the classification labels are manually added, and the efficiency is relatively low. In order to improve the classification efficiency and save human resources, so that software bugs can still be automatically classified automatically in the case of massive data, we use the MapReduce distributed programming model to implement the Naive Bayes classification algorithm and automatically add bugs to the bugs. label. Experimental results show that our algorithm can effectively perform classification functions. Our code will be released on Github.

Key words Naive Bayes classification algorithm；MapReduce；distributed；software bug

1 问题描述

在 Github 代码仓库中，有一个 issues 栏目，这个栏目的目的是收集用户对该开源代码的反馈，即用户会提出软件 bug。目前的 Github issues 中存在大量已分类（即加上标签）的软件 bug（如图 1 中红框所示）。

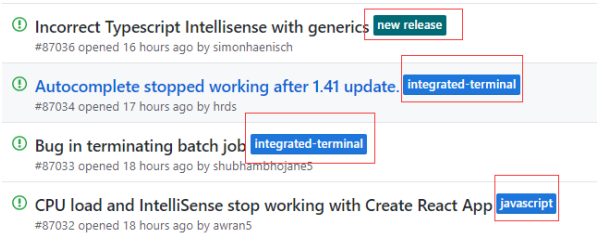

图 1. Github 上的软件 bug 示例

然而，这些分类标签都是人为手工添加的，当 issues 的数量很大的时候，人工添加标签的效率就显得极为低下。

为了解决这个问题，我们设计了基于 MapReduce 并行计算模型的朴素贝叶斯分类方法，使得计算机可以通过分布式计算根据 issue 的标题快速地给软件 bug 进行分类。

2 实验原理

2.1 朴素贝叶斯分类算法

本次实验的问题，其实是一个文本分类的问

题，我们将问题简化成对每个 issue 的标题进行分类（加上标签）。解决这样的问题可以使用朴素贝叶斯算法[1]。

用朴素贝叶斯去解决分类和回归问题首先要知道贝叶斯定理[2]， 贝叶斯定理主要是提供了一个算法解决这样一个问题:已知某条件概率，如何求事件交换后的条件概率，即已知 P(A|B), 求 P(B|A)。 定理如下:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} \quad (1)$$

其中：

$$P(A|B) = \frac{P(AB)}{P(B)} \quad (2)$$

上式中 P(A|B)表示事件 B 已经发生的前提下事件 A 发生的概率， P(AB)表示 A 和 B 同时发生的概率。而当 A 和 B 两个事件是互斥的，完全独立的没有交集的两件事，此时 P(AB)＝P(A)P(B)，这时就是朴素的。

朴素贝叶斯求解分类问题就是对于给出的待分类项，算出它属于每个标签的概率，哪个概率最大，就是哪个标签。首先我们要求出每个样本的特征向量 x，对给定的样本特征 x，样本属于类别 ei 的概率是

$$P\ (ei|x) = \frac{P(x|ei)P(ei)}{x} \quad (3)$$

假设 x 的维数是 K（该文本有 K 个单词），由于是朴素的，特征条件互斥，根据全概率公式有：

$$P(ei|x) = \frac{P(ei)\prod_{j=1}^{K} P(xj|ei)}{P(x)} \quad (4)$$

由于对于同一个文本来讲， P(x)是一样的， 所以预测它属于哪个分类， 只要算上式的分子部分即可。 对于一个给定的测试样本， 分子部分常用的有两种模型：

1.伯努利模型(Bernoulli Model)：

$$P(xj|ei) = \frac{N_{ei}(xj)}{N_{ei}} \quad (5)$$

$$P(ei) = \frac{N_{ei}}{N} \quad (6)$$

其中N_{ei}表示有多少个训练集文本的标签是 ei，$N_{ei}(xj)$表示特征 xj 出现在了多少个标签为 ei 的样本中。 N 表示训练集总样本数。在这个模型中特征向量 x 是一个 K 维 01 向量，每一维表示某个单词是否出现在当前文本里。

2.多项式模型(Multinomial Model)：

$$P(xj|ei) = \frac{NW_{ei}(xj)}{N_{ei}} \quad (7)$$

$$P(ei) = \frac{N_{ei}}{N} \quad (8)$$

其中$NW_{ei}(xj)$表示在训练集中，标签为 ei 的文本中，单词 xj 出现了多少次（一篇文章中出现多次按多次计算），NW_{ei}表示训练集中标签为 ei 的文本中包含多少个单词(不去重)。

从二者的公式我们可以看出，伯努利模型是以文本为粒度，多项式模型是以单词为粒度，伯努利模型相比多项式模型会丢失词频信息，这会造成分类能力下降，所以我们这次实验采用的是多项式模型。

当测试文本中存在一个单词 x，x 是训练集中出现过的单词，但是 x 没有在标签为 ei 的训练集文本中出现过，那么此时$P(xj|ei)$=0，从而导致$P(ei)\prod_{j=1}^{K} P(xj|ei) = 0$。此时，我们可以通过平滑解决这个问题。增加了平滑后的多项式模型公式如下：

$$P(xj|ei) = \frac{NW_{ei}(xj)+alpha}{NW_{ei}+alpha*totalwords} \quad (9)$$

在上式中，0 < alpha <=1，当 alpha=1 时，称为拉普拉斯平滑；当 0 < alpha < 1 时，称为 Lid stone 平滑。totalwords 表示训练集中有多少种不同的单词数。在本次实验中，我们采用拉普拉斯平滑，通过该平滑公式,我们保证了每一项$P(xj|ei)$不为 0。

另外，当 K 很大的时候，$\prod_{j=1}^{K} P(xj|ei)$趋近于 0，在计算机里可能会造成浮点数溢出导致结果为 0，这个时候我们可以对每一项$P(xj|ei)$求对数，变乘为加：

$$P(ei|x) = \log\big(P(ei)\big) + \sum_{j=1}^{K} \log\big(P(xj|ei)\big) \quad (10)$$

2.2 Hadoop简介

Hadoop 是 Apache 软件基金会所开发的并行计算框架与分布式文件系统，最核心的部分就是 HDFS 和 MapReduce。

2.2.1 HDFS

HDFS 是 Hadoop 分布式文件系统（Hadoop Distributed File System）的缩写，为分布式计算存储提供了底层支持。采用 JAVA 语言开发，可以部署在多种普通的廉价机器上，以集群处理数量积达到大型主机处理性能。

HDFS 采用 master/slave 架构。一个 HDFS 集

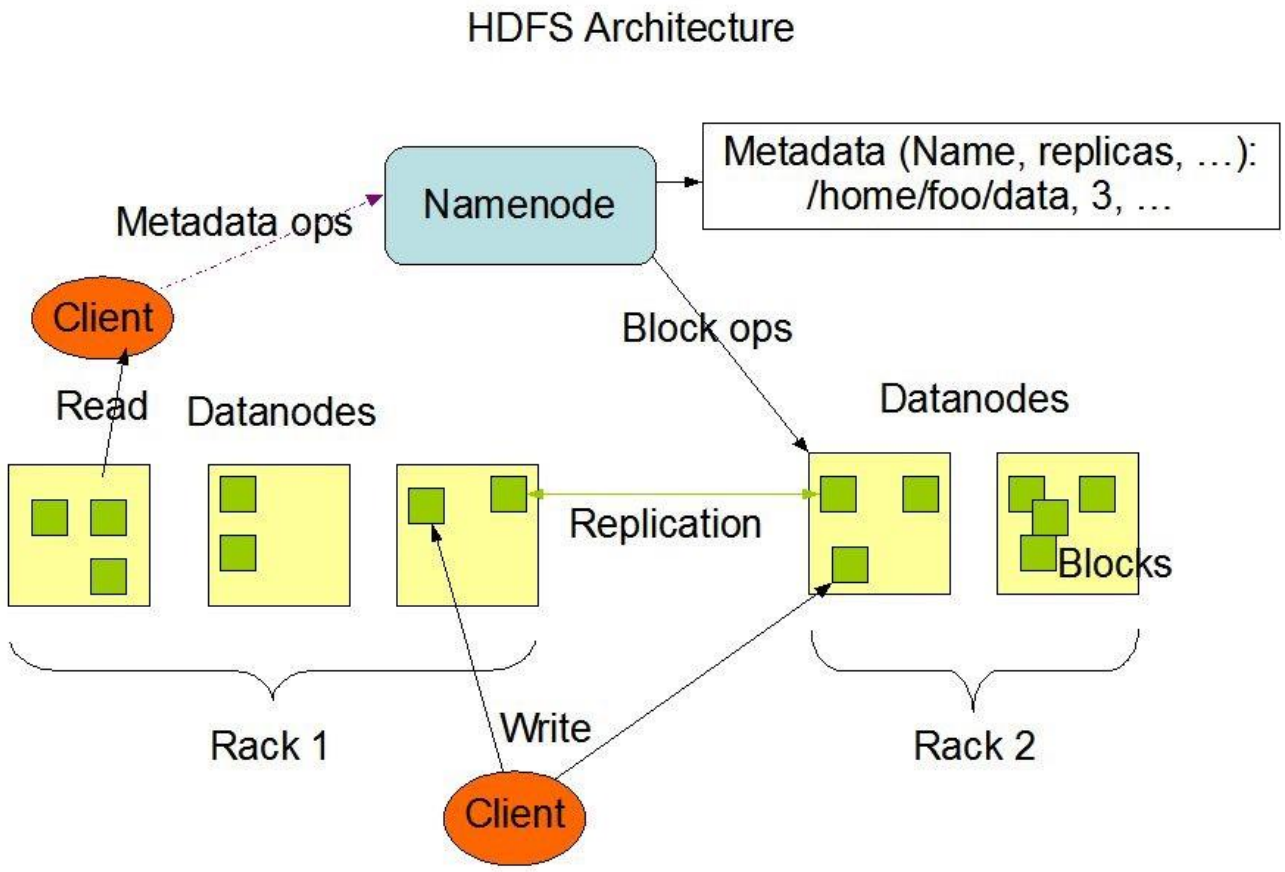

图 2. HDFS 文件系统架构图[3]

群包含一个单独的 NameNode 和多个 DataNode。NameNode 作为 master 服务，它负责管理文件系统的命名空间和客户端对文件的访问。NameNode 会保存文件系统的具体信息，包括文件信息、文件被分割成具体 block 块的信息、以及每一个 block 块归属的 DataNode 的信息。对于整个集群来说，HDFS 通过 NameNode 对用户提供了一个单一的命名空间。

DataNode 作为 slave 服务，在集群中可以存在多个。通常每一个 DataNode 都对应于一个物理节点。DataNode 负责管理节点上它们拥有的存储，它将存储划分为多个 block 块，管理 block 块信息，同时周期性的将其所有的 block 块信息发送给 NameNode。

HDFS 文件系统架构图如图 2 所示。文件写入时，Client 向 NameNode 发起文件写入的请求，NameNode 根据文件大小和文件块配置情况，返回给 Client 它所管理部分 DataNode 的信息，Client 将文件划分为多个 block 块，并根据 DataNode 的地址信息，按顺序写入到每一个 DataNode 块中。文件读取时，Client 向 NameNode 发起文件读取的请求。NameNode 返回文件存储的 block 块信息、及其 block 块所在 DataNode 的信息。Client 读取文件信息。

2.2.2 MapReduce 编程模型

MapReduce 是一种用于大规模数据集（大于 1TB）的并行计算编程模型，它的宗旨是让不懂并行编程的人也能写出并行代码。它的主要思想就是“Map（映射）”和“Reduce（规约）”，是借鉴了函数式编程语言和矢量编程语言的特性。

MapReduce 编程模型的数据流如图 3 所示。我们不深究 MapReduce 的具体底层实现，简单来说它就分为 3 个步骤：

1. Map(映射)
2. Shuffle (排序)
3. Reduce(规约)

为了让 MapReduce 程序运行，我们首先要将所有输入数据组织成键值对（key, value）的形式并存放在 hdfs 的文件系统里。

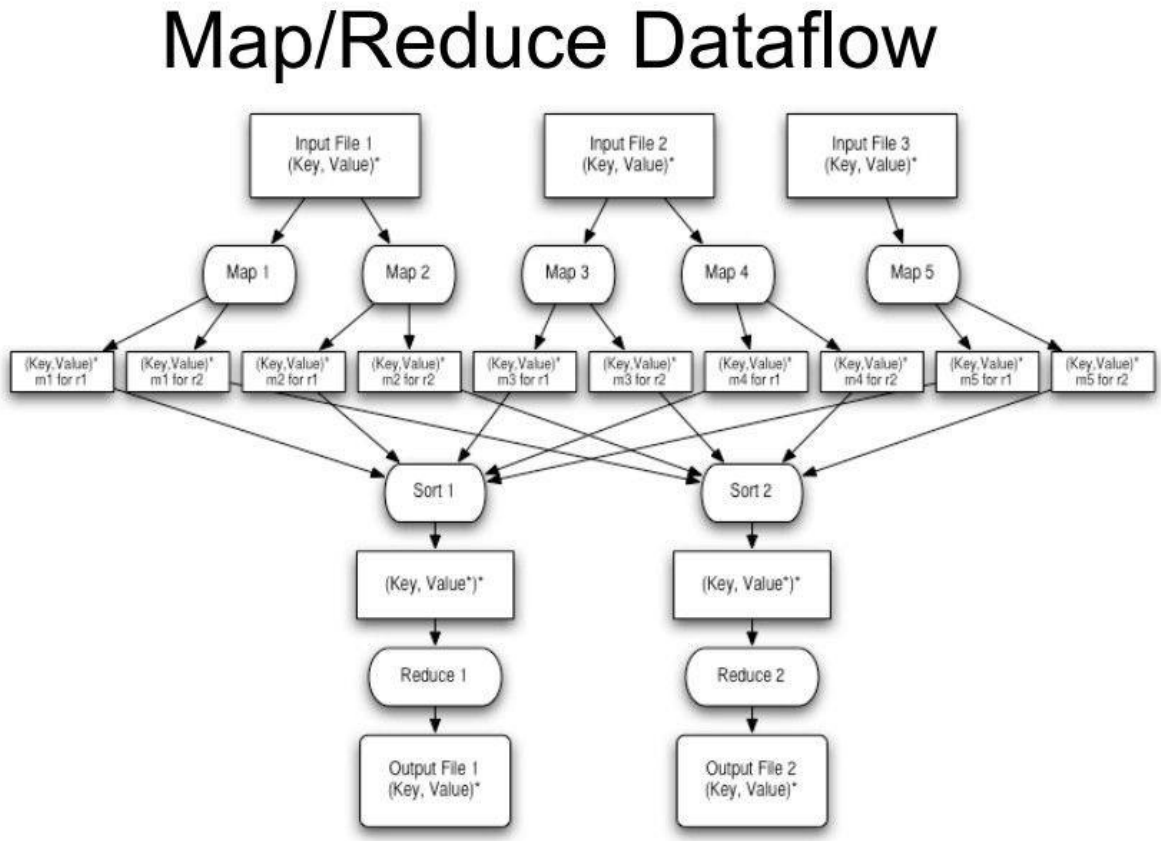

图 3. MapReduce 模型数据流[3]

一个常用的方法是将行作为一个 key，将行的内容作为 value，那么每一次，输入会按行切开成一个一个 split，每一个输入 split 会创建一个计算任务来调用我们定义好的 Map 函数进行相应的处理，最后将结果以新的(key， value)形式输出。具体如何定义 key 和 value 要取决于用户希望下一步作什么处理。所以 Map 这一步就是将输入映射成新的键值对输出。

在进入 Reduce 步骤之前，会先进行一次 Shuffle 操作。Shuffle 不需要用户去实现，它的主要功能就是对 Map 的输出进行一个排序，将相同键值的键值对合并到一起传给 Reduce。

Reduce 接收 shuffle 后的结果作为输入，在这里 Reduce 的输入实际上是(key, List<value>)，即将相同 key 的所有 value 合并成一个数组，这个数组整体作为一个 value，然后传入到 Reduce 中，那么在 Reduce 中，我们就可以一次性对同一个 key 的所有 value 进行处理，最后将结果还是以（key, value）的形式输出。

为了更好地理解 MapReduce 模型，一个实例如图 4 所示。这个实例是统计每个单词出现了多少次，在 Map 阶段对每行每遇到一个单词就输出一个(单词， 1)的键值对，在 reduce 阶段将同一个单词为 key 的 key-value 对的 value 值相加（因为这些 value 值都是 1），然后就可以得到 key 这个单词出现了多少次，最后输出。

2.3 使用python开发MapReduce程序

传统的 MapReduce 程序应该是用 JAVA 语言写的，但我们对 JAVA 不是很熟练，反而对 python 比较熟悉。经过查找资料，我们找到了使用 python 开发 MapReduce 程序的方法[5]。

Hadoop Streaming 提供了一个便于进行 MapReduce 编程的工具包，使用它可以基于一些脚本语言或者其他编程语言来实现 Map 和 Reduce 过程，从而充分利用 Haoop 并行计算框架的优势

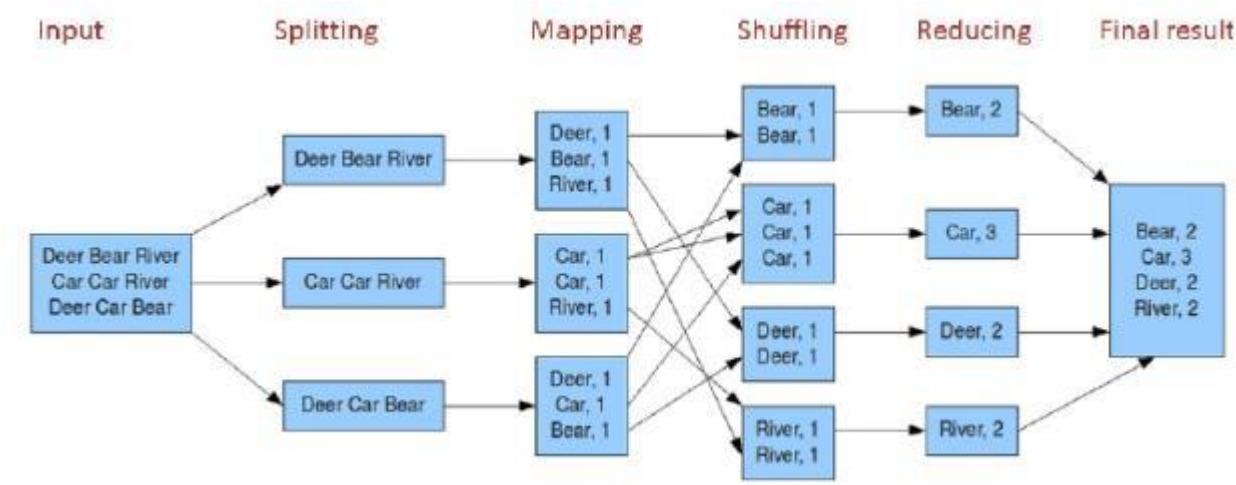

图 4. 利用 MapReduce 统计单词数[4]

来处理大数据。

Streaming 方式是基于 Unix 系统的标准输入输出来进行 MapReduce 任务的运行，任何支持标准输入输出特性的编程语言都可以使用 Streaming 方式来实现 MapReduce 程序。

在利用 Hadoop Streaming 实现的 MapReduce 程序中，我们姑且将实现 Map 功能和 Reduce 功能的程序成为 mapper 和 reducer。mapper 和 reducer 会从标准输入中读取用户数据，一行一行处理后发送给标准输出。mapper 任务运行时，它把输入切分成行并把每一行提供给可执行文件进程的标准输入。同时，mapper 收集可执行文件进程标准输出的内容，并把收到的每一行内容转化成(key，value)对，作为 mapper 的输出。 默认情况下，一行中第一个制表符（\t）之前的部分作为 key，之后的（不包括\t）作为 value。如果没有制表符，整行作为 key 值，value 值为 null。

因此我们使用 python 开发 MapReduce 代码的思路就清晰了。我们只需要用 python 实现 mapper 和 reducer 即可。在每一个代码中，使用 python 的 sys.stdin 来读入标准输入的数据，使用 sys.stdout 来输出数据就可以了，当然对于输入和输出数据都要处理成“key \t value”这样的形式。最后我们直接调用 Hadoop Streaming 工具包来运行我们的 python 程序即可。

3 模型设计

3.1 朴素贝叶斯分类算法变体

根据公式 8 和公式 9，我们整理得到使用朴素贝叶斯分类算法进行软件 bug 分类需要计算的中间变量表：

表 1. 朴素贝叶斯分类需计算的中间变量表

变量名	意义
N	训练集文章总数
NWei[label][xj]	训练集中标签为 label 的标题中单词 xj 的出现次数
NW[label]	训练集中标签为 label 的 title 中包含了几个单（不去重）
Nei[label]	训练集中标签为 label 的标题总数
totalwords	训练集中一共有多少个不同的单词

对于一个标签为 label 的标题中的所有单词可以分成三类：

A. 在训练集中标签为 label 的标题里出现过

B. 在训练集中标签为 label 的标题里没现过

C. 在训练集所有标题里都没有出现过。

对于 C 类的单词我们按传统的做法直接舍弃，当它没有出现过。对于 A 类和 B 类单词我们按照公式 9 计算相应概率。

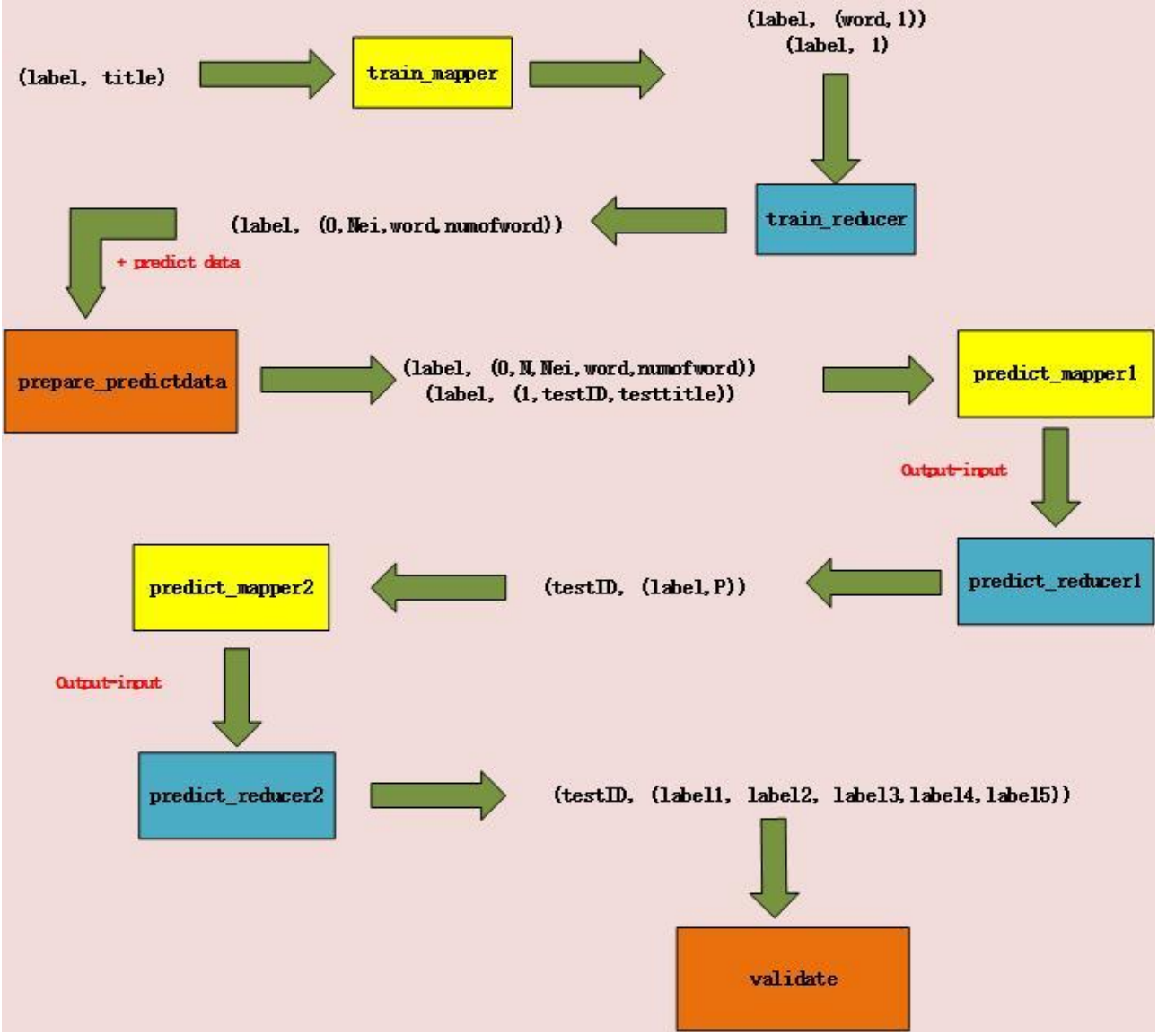

图 5. 基于 MapReduce 的朴素贝叶斯分类算法总体框图

现在问题是平滑公式中的 totalwords 是比较难以统计的，就算我们能够统计，这个数字也是比较大，那么计算出来的概率就会很小，连乘之后容易溢出，我们可以将其改为 N,一方面 N 远没有 totalwords 大，另一方面 N 也比较容易知道方便编程。同时这样更改不会影响预测概率之间的相对大小，也就不会影响正确性，这条式子我们称为朴素贝叶斯分类算法变体：

$$P(xj|ei) = \frac{NW_{ei}(xj)+alpha}{NW_{ei}+alpha*N} \quad (11)$$

3.2 基于MapReduce的朴素贝叶斯分类算法

由于 Github 上一条 issue 可以有多个标签，由于我们目前无法做到多标签分类，于是我们将问题简化，将每个 issue 的标签设为其第一个标签。基于此我们设计的基于 MapReduce 的朴素贝叶斯分类算法总体框图如图 5 所示。

我们经过分析，认为无法只使用一对 MapReduce 程序完成训练和分类，因此我们将训练和预测过程分成不同的 MapReduce 程序去执行，如图 5 所示，我们的代码总有 train_mapper、train_reducer、prepare_predictdata、predict_mapper1、predict_reducer1、predict_mapper2、predict_reducer2、validate 共 8 个部分，下面我们每一个部分逐一介绍。

3.2.1 训练过程----train_mapper 和 train_reducer

我们将训练数据按（label，title）的形式预先

整理到一个文本中，训练过程主要是统计每种label有多少个训练样本以及每个单词在某个label的样本中出现了多少次。

在train_mapper程序中，遍历每一个文本，对每一文本中的每个单词，输出键值对(label，(word, 1))，意识是标签为label的title中，单词word出现了1次，最后遍历完一个文本的所有单词之后，输出键值对(label，1)，意思是标签为label的title出现了1次，代码如下：

```
import sys
"""
输入:
label  title
输出:
<label, <word, 1>>
<label, 1>
Nei[label]: 标签为label的title数
"""
for row in sys.stdin:
    label, title = row.split('\t\t') # 我这里数据以两个制表符分割key和value
    title = title.split()
    flag = False
    for word in title:
        key_value = "{}\t\t{}".format(label, word + ' ' + str(1))
        print(key_value)
    print("{}\t\t{}".format(label, str(1)))
```

在train_reducer程序中，对train_mapper提供的相同label的键值对统一处理。这里输入会有两种情况，分别为(label，(word, 1))和(label，1)，我们可以通过判断value包含多少个元素来区分这两种输入，对于第一种输入，我们可以用来计算标签为label的title中单词word出现了多少次，对于第二种输入，可以用来统计标签为label的title数。

```
import sys
"""
输入:
<label, <word, 1>>
<label, 1>
输出:
<<label>, <0, Nei, word, numofword>>
统计标签为label的title中 单词word的出现次数
"""
wordHash = dict()
# 键值为（label, word）的字典，统计标签为label的title中单词word出现的次数
titleHash = dict()
# 键值为label的字典，统计标签为label的title个数
for row in sys.stdin:
    key, value = row.split('\t\t')
    value = value.split()
    if len(value) == 1:
        if key not in titleHash:
            titleHash[key] = 0
        titleHash[key] += int(value[0])
    elif len(value) == 2:
        newkey = key + ' ' + value[0]
        if newkey not in wordHash:
            wordHash[newkey] = 0
        wordHash[newkey] += int(value[1])

for k,v in wordHash.items():#得到输出
    k = k.split(' ')
    key = k[0]
    value = str(0) + ' ' + str(titleHash[key]) + ' ' + k[1] + ' ' + str(v)
    print(key + "\t\t" + value)
```

上面reducer的代码输出键值对为(label, (0, Nei, word, numofword))，它的意思是，标签为label的title一共有Nei个，在这里面单词word一共出现了numofword次。这里value中还有一个0，这里这个0是用来表示这个键值对是训练数据的键值对。后面会使用1来表示需要预测的数据的键值对。

3.2.2 准备需要预测的数据----prepare_predictdata

我们的算法是按照训练预测分离，其中训练的模型可以复用到不同测试数据上的方式设计的。这一步prepare_predictdata是一个本地程序，的作用是整合训练数据和需要预测的数据到一个文本中作为预测MapReduce程序的输入，这里预测的数据你可以随意替换成不同的，比较灵活。这一步会得到如下输出：

A. (label, (0, Nei, word, numofword))

B. (label, (1, testID, testtitle))

其中前者如3.2.1中所述，后者的意义是第testID条需预测数据的内容是testtitle，1表示它是测试数据，请注意，这里这个label不是它真实的标签，而是希望程序计算它属于这个label的概率，也就是说假设我们一个有n种标签，那么对于每一条测试数据，我们会得到n个键值对：

(label_i, (1, testID, testtitle))　　i =0,1,2…n

这一步不是MapReduce过程，代码不在这里展示，可见附件中的MapReduce_code文件夹下的prepare_predictdata.py。

3.2.3 预测过程1----predict_mapper1和predict_reducer1

这一步是求出每条需预测数据属于每种label的概率。这一步中mapper的输出=输入，我们的目的是希望利用mapper之后的shuffle过程对数据先按照键值作一个排序，将相同label的训练数据和测试数据归到一起之后再做处理：

```
import sys
"""
输入:
<<label>, <0, N, Nei, word, numofword>>
<<label>, <1, testID, testtitle>>
输出 = 输入，目的是理由shuffle先排个序

"""
for row in sys.stdin:
    row = row.strip()
    key, value = row.split('\t\t')
    key_value = "{}\t{}".format(key, value)
    print(key_value)
```

在 predict_reducer1 这一步里，根据接收的已经排好序的相同 label 的训练数据和测试数据，用训练数据的 value 计算表 1 中的变量，最后根据公式 11 计算测试数据属于某个 label 的概率，得到输出键值对(testID, (label, p))，即第 testID 条数据的标签是 label 的概率为 p：

```
import sys
"""
NWei: dict 套 dict, NWei[label][xi] 表示训练集中标签为label的title中单词xi出现了几次（不去重）
NW: a dict,  NW[label] 表示训练集中标签为label的title中包含了几个单词（不去重）
Nei: a dict, Nei[label] 表示训练集中标签为label的title总数
N:  训练集title总数
wordset: a set 记录训练集中出现过的单词

输入:
<<label>, <0, N, Nei, word, numofword>>
<<label>, <1, testID, testtitle>>
输出:
<<testID>, <label, p>> 第testID条测试数据的标签是label的概率为p
"""
```

```
wordset = set()
testtitle = []
N = 0
Nei = dict()
NWei = dict()
NW = dict()
alpha = 1.0
for row in sys.stdin:
    row = row.strip()
    key, value = row.split('\t')
    value = value.split()
    value[0] = int(value[0])
    if value[0] == 1:
        testtitle.append(row)  # 先保存好测试数据，后续再预测
    elif value[0] == 0:
        N = int(value[1])
        word = value[3]
        wordset.add(word)    # 更新word集合
        if key not in Nei:
            Nei[key] = 0
            NW[key] = 0
            NWei[key] = dict()
        Nei[key] = int(value[2])    # 更新Nei、NW、NWei
        NW[key] += int(value[4])
        if word not in NWei[key]:
            NWei[key][word] = 0
        NWei[key][word] += int(value[4])
```

```
# 注意这一步，由于不是每种label的title都包含了wordset的所有单词，
# 所以这里要将没出现的单词次数设为0，因为测试title有可能有这些词
for label in Nei:
    for word in wordset:
        if word not in NWei[label]:
            NWei[label][word] = 0


# 计算第testID条title 属于 label的概率
for row in testtitle:
    label, value = row.split('\t')
    value = value.split()
    testID = int(value[1])
    title = value[2:]
    px = 1.0
    pe = float(Nei[label]) / N  # 计算p(ei)
    for word in title:
        if word not in wordset:
            px *= (float(0) + alpha) / float((NW[label] + alpha * N))
        else:
            dx = float(NWei[label][word]) + alpha
            dy = float((NW[label] + alpha * len(title)))
            px *= dx/dy  #计算p(x|ei)
    p = px * pe #当前title属于label的概率
    output = "{}\t\t{} {}".format(str(testID), label, str(p))
    print(output)
```

3.2.4 预测过程 2----predict_mapper2 和 predict_reducer2

在这一步中，目的是求出对于每一条需要预测的数据，概率最高的 5 个标签是什么。与前一步相同，这里的 mapper 也不做任何操作，目的也是希望先利用 shuffle 排序后再做处理。

在 reducer 中，我们的输入是(testID, (label, p))，这里因为经过了 shuffle 之后，我们可以收集到每一条需预测数据，它们属于每一种 label 的概率是多少，我们将它们记录下来，然后排序，求出概率最高的 5 个 label 输出即可。输出格式为(testID, (label1,label2,label3,label4,label5))。这一步 reduce 操作的代码如下：

```
import sys
"""
输入:
<<testID>, <label, p>>
输出: 每个title 概率最高的5个类别
<<test_ID>,  <label1, label2, label3, label4, label5>>
"""
total = 0
predict = dict()
for row in sys.stdin:
    testID, value = row.split('\t\t')
    value = value.split()
    if testID not in predict:
        predict[testID] = dict()
    predict[testID][value[0]] = float(value[1])
    # predict[testID][label]
    # testID这条数据属于label的概率

for testID in predict:
    # 对概率排序
    p = sorted(predict[testID].items(), key=lambda d:d[1], reverse=True)
    output = "textid: {}\t\t{} {} {} {} {}".format(
        testID, p[0][0], p[1][0], p[2][0], p[3][0], p[4][0])
    print(output)
```

在得到最终预测结果之后，加入我们知道预测数据的真实标签，那么我们可以使用 validate 这个程序去验证正确率。

4 实验

4.1 数据集

本次实验的用到的数据从 Github 上的 vscode 项目的 issues 里面爬取已经加上标签的软件 bug。原始数据一共有 3110 条，但其中存在一些 label 只出现了几次的数据，出现次数太少的话可以视为是噪声数据，我们去掉了 label 出现不足 10 次的数据，最后我们得到了 2785 条数据，其中 1935 条数据用于训练，850 条数据用于测试，label 种类为 32 种。

另外，由于原始数据每一条 issue 可能属于多个软件 bug 标签，并且这个标签数目不固定，这样不固定数目的多标签分类还没有什么适合并行

编程的方法来实现，因此我们采用了将这个多标签多分类问题转化成了单一标签多分类问题，将每条 issue 的第一个软件 bug 标签作为它的真实标签。我们将重点放在学习 MapReduce 的编程模式上。

4.2 评价指标

前面提到了，每个 issue 可能有多个标签，但是我们只取了一个，而人们在手工打标签的时候不会按照最相关的顺序打标签，所以有可能我们预测出来的 label 中，概率最高的 label 不是我们之前保留的那个 label，因此，我们参照了行人重识别领域的评价指标，采用 rank-1 正确率和 rank-5 正确率来评价我们的模型。

rank-1 正确率就是取概率最高的预测 label 来跟我们保留的真实 label 比较从而计算正确率。对于 rank-5 正确率，对每一条待预测数据，假设我们预测出的概率最高的 5 个 label 中存在一个 label 与我们保留的真实 label 相同，我们就认为这个数据预测正确。

4.3 实验环境

由于机器条件限制，我们的实验没有使用真实的分布式集群，我们使用 VMware 虚拟机在一台电脑上安装了三个 Ubuntu16.04 系统，利用虚拟机的 NAT 网络设置将三个虚拟机以及我们的本机置于同一个网段下，这样我们可以将这三个系统部署成一个分布式集群，我们称之为虚拟分布式集群，然后在该虚拟集群上面安装 Hadoop 来跑 MapReduce 并行代码。并且由于和本机在一个网段下，我们可以在本机上通过 ssh 工具登陆它们，而避免了虚拟机和本机之间麻烦的的键鼠切换。具体设置方法可以查看我们 Environment Setting 文件夹里的设置说明。

4.4 实验结果

为了验证我们的 MapReduce 代码是否编写正确，我们首先实现了一个单机版的程序在本地进行实验，以便后续作对比实验。单机版程序保存在 local_code 文件夹下的 classifier_local.py 中。实验结果如图 6 所示：

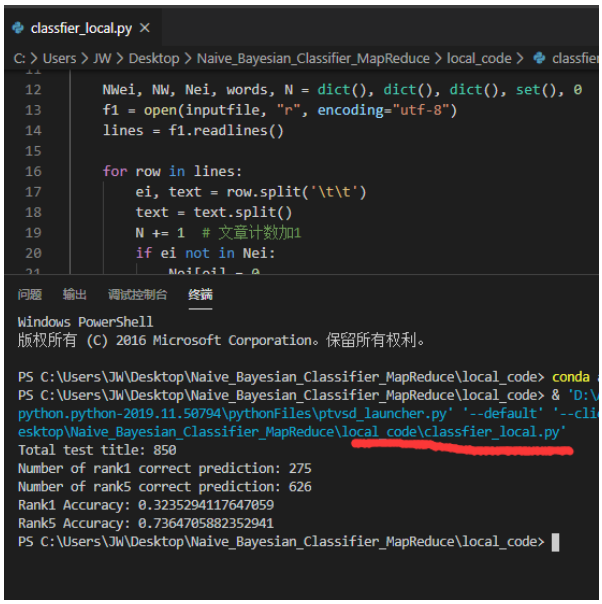

图 6. 单机版贝叶斯分类结果

接着我们在虚拟分布式集群上运行 MapReduce 版程序，实验结果如图 7 所示：

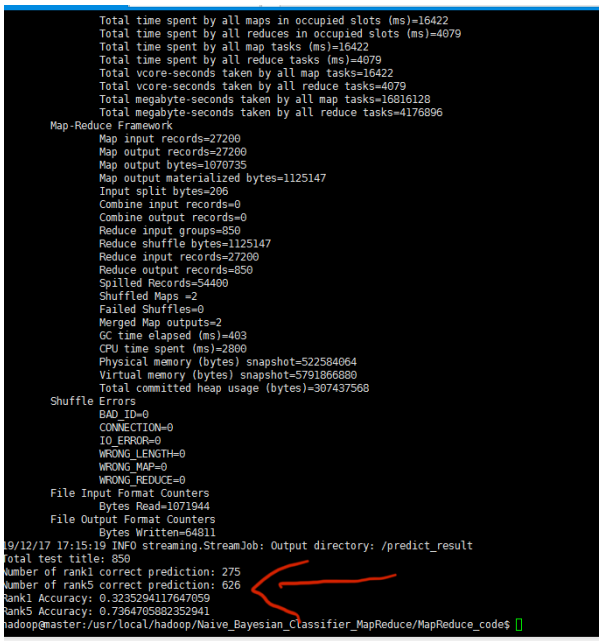

图 7. MapReduce 版贝叶斯分类结果

从图 6 和图 7 可以看出，单机版贝叶斯分类结果和 MapReduce 版分类结果是一样的，这证明我们的分布式代码是正确的。从结果中我们可以看到，rank-1 正确率是 32%，这一点是意料之中，原因如 4.2 中所述。但是，rank-5 正确率达到了 73%，这说明在概率最高的 5 种 label 中，我们找到跟之前保留的真实 label 一样的 label 的可能性

较高，而 label 的总类是达到了 32 种的，能在前 5 个就找到正确的 label，说明我们的分类器的性能是很好的。

4.5 实验中遇到的问题

最开始我们认为在将数据输送到 Map 过程的时候就会将所有相同的键值的所有数据归在一起计算，所以我们一开始在 Map 过程中进行了统计类计算，这导致最后结果是不正确的。原因是 Map 的输入是将输入数据分成一个一个 split，然后对每一个 split 创建一个计算任务来进行分布式计算，这些 split 是相同键值的数据，但是不一定是该键值的所有数据，有可能两个 split 的数据加起来才是该键值的所有数据。因此我们在编写 MapReduce 程序的时候一定不能在 Map 过程中进行像最终统计这样的计算，否则会导致结果错误。只有在 Reduce 过程中，才会对相同键值的所有数据一起处理。

另外，在设计 MapReduce 版的朴素贝叶斯分类程序的时候，由于该算法设计到的需要计算的变量较多，只用一次 MapReduce 完成该任务基本上是不可能的。一开始我们没有想到要拆成几次 MapReduce 的做法，在这一步卡了很久。后面想到了这个做法，再将所需变量一个个写出来，才想出每一步 MapReduce 应该执行什么功能。所以我们以后如果还要编写 MapReduce 程序，不需要执着于一个 MapReduce 完成所有任务，可以拆分过程，分而治之。

最后，当我们代码运行出错的时候，Hadoop 的屏幕输出不会告诉我们在哪一步错了，所以一开始我们每次遇到问题，只能肉眼反反复复地排查代码，但后面发现错误信息可以在 Hadoop 安装目录下的 logs 文件夹中的 userlogs 文件里找到。

5 总结

本次大作业我们针对目前 Github 仓库上的 issues 栏目的软件 bug 标签需要人为添加而导致效率低下的问题，设计了基于 MapReduce 的朴素贝叶斯分类算法，并且在虚拟分布式小集群上运行了我们的代码。实验表明，我们的分布式代码的结果与本地单机版代码的结果是一样，说明我们的代码编写正确。此外，分类性能也在我们的预期之内，rank-5 准确率达到了一个比较好的数值。

通过这次大作业，我们学习了 MapReduce 编程模型，对它的程序编写方法有了比较清晰的了解，同时我们也感受了什么是分布式的计算，学到了原来没有接触过的知识。

本次大作业的遗憾之处在于我们没有找到一个良好的可以改写成 MapReduce 形式的支持多标签多分类的算法，希望以后有机会能够改进。

参 考 文 献

[1] 《朴素贝叶斯》，百度百科，https://baike.baidu.com 2019,11,01

[2] 《贝叶斯定理》，百度百科，https://baike.baidu.com 2019,09,26

[3] 《hdfs》，百度百科，https://baike.baidu.com 2019,11,23

[4] 《MapReduce 工作流程最详细解释》，简书，https://www.jianshu.com/p/461f86936972 2018,10,18

[5] 《Python 实现 Hadoop MapReduce 程序》, cnblog, https://www.cnblogs.com/chushiyaoyue/p/5713177.html 2016,07,28