基于 MapReduce 的软件 Bug 分类

姓名：欧阳浩岚

班级：计算机五班

学号：18340133

日期：2021 年 1 月 25 号

1. 题目要求

1.1 题目

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，现在的分类标签大都是基于人工添加的，效率比较低。本项目通过爬取大量具有分类标签的 Bug，利用 MapReduce 分布式编程模型，实现分类算法，自动给 Bug 加上标签。

1.2 要求

1)、爬取至少 1000 个具有分类标签的 bug;

2)、采用 MapReduce 实现分类算法;

3)、测试验证算法的准确度;

4)、分析结果并得出结论;

5)、提交源码和报告，压缩后命名方式为：学号_姓名_班级

2. 解决思路

2.1 获取数据

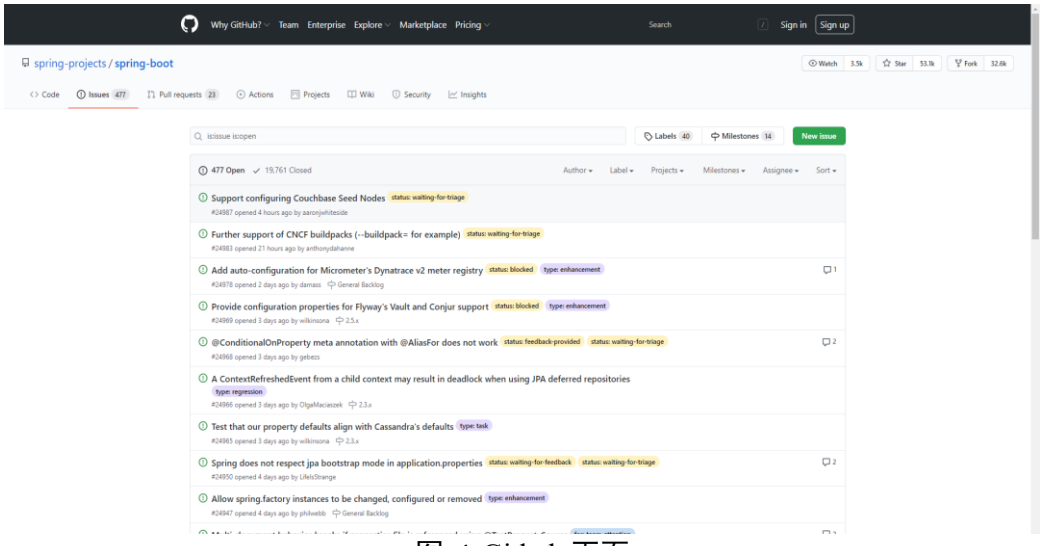


图 1 Github 页面

从 Github 页面可以看到，一个仓库主要包含 Code、Issues、Pull requests、Actions、Projects、Wiki、Security、Insights 这几部分。一般来说，Issues 是开发者给仓库的作者提交 Bug 的地方。

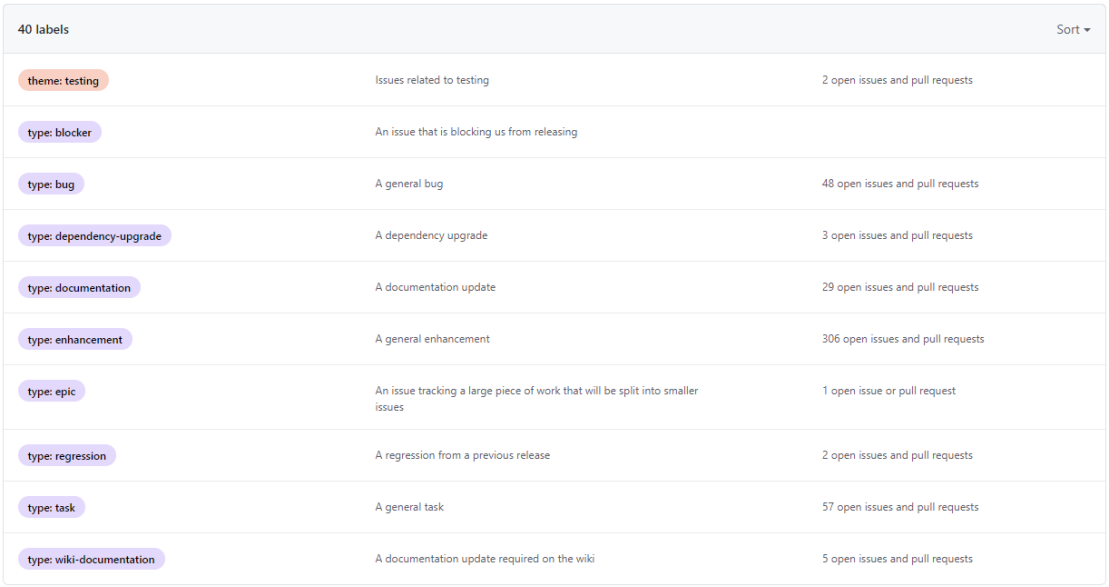


图 2 Issues 的类别

Github 的 Issues 中有很多人工标注的分类标签，我们的实验目的是：通过学习人工标注的分类标签，得到一个自动标注的分类算法。本次实验，我将通过爬取 Github 上的 Issues 作为数据集，实现软件 Bug 的自动分类算法。

因为是开发者自己提出的软件 Bug，不同开发者会有不同的 Bug 描述风格。以图 3 作为例子，Bug 的描述中除了包含有代码、一些无明确意义的数字串（经过哈希编码后的数值），可能还包含其他仓库开发者的讨论等等。这种没有统一格式的文本，很难利用程序进行自动识别。同时，不同代码仓库之间还有不同的 Bug 分类标签，但分类

算法一般只能在给定的有限个标签内进行分类。 如果进行跨软件仓库的分类，其性能可能大幅度下降。

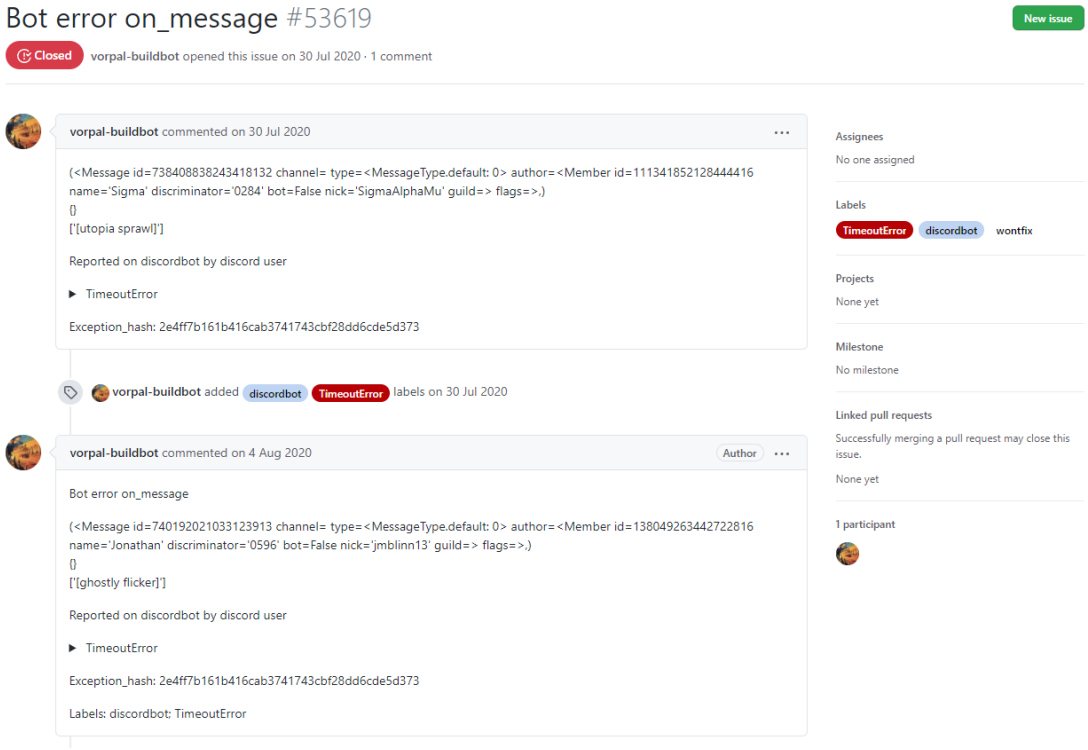


图 3 一个 Issues 具体例子

因此，如果得到稳定的数据源，需要满足以下条件：

① 对同一个或同一类的代码仓库里的 Issues 做分类任务

② Issues 的类别之间要有明确的区分度

③ 为了统一 Issues 的文本格式，只选用 Issues 的标题作为输入源

④ 选取的 Issues，标题必须能够体现 Bug 的类型

根据上面的考量，我最终使用的 Github 上比较热门的代码仓库 PyTorch 上的 Issues。

图 4 PyTorch 的 Issues 示例

2.2 MapReduce 实现分类算法

2.2.1 MapReduce

本实验使用的关键技术是 MapReduce 编程模型。MapReduce 是一个编程模型，也是一个处理和生成超大数据集的算法模型的相关实现。用户首先创建一个 Map 函数处理一个基于<key, value> pair 的数据集合，输出中间的基于<key, value> pair 的数据集合；然后再创建一个 Reduce 函数用来合并所有的具有相同中间 key 值的中间 value 值。

MapReduce 的通用流程是：

① 输入数据，将数据进行划分

② Mapper 处理输入的数据，将其变为<key, value>的形式

③ Shuffler 接收 Mapper 传递的数据，按照关键字进行排序

④ 一个 Reducer 接收相同 key 所有<key, value>键值对，并进行处理

⑤ 将 Reducer 的处理结果进行输出

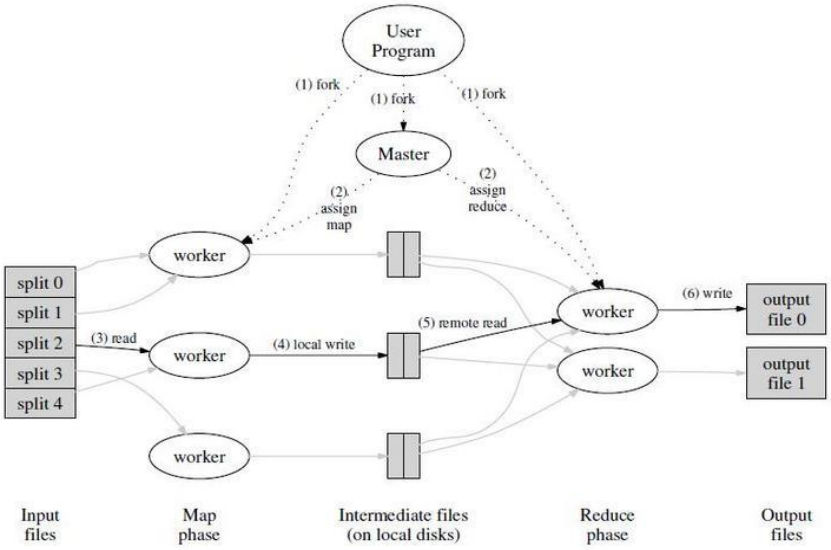


Figure 1: Execution overview

图 5 MapReduce 框架

2.2.2 HDFS

MapReduce 编程模型是一种分布式计算框架，它通过 GFS 分布式文件系统实现输入数据、中间缓存数据以及输出结果在分布式集群中的存储。实验中我使用的是开源软件 Hadoop，分布式文件系统为 HDFS。

HDFS 的全称是 Hadoop Distributed File System，是一种易于扩展的分布式文件系统。HDFS 由两部分组成：Namenode 负责管理整个文件系统信息，包括文件和文件夹结构，Datanode 负责存储文件所有的数据块，而文件的具体存放位置则记录在 Namenode 上。为了防止数据的丢失以提高分布式系统的容错性，HDFS 为每一份文件存储 3 个备份。

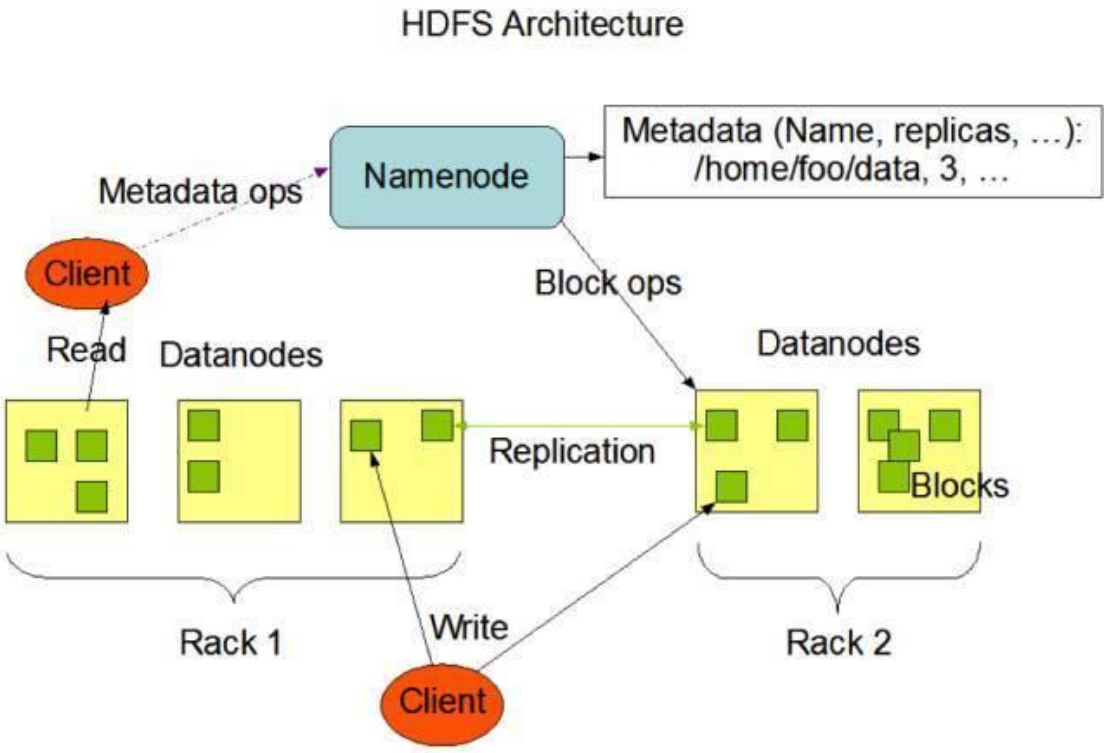


图 6 HDFS 架构

2.2.3 分类算法

由于输入数据都是文本数据，本次分类任务其实是一个基于文本的多分类任务。对于文本的向量化，我使用的是 TF-IDF 编码。具体的公式如下：

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

$$idf_i = \log_2 \frac{|D|}{|\{j: t_i \in d_j\}|}$$

其中，i 代表词典中第 i 个词，j 代表第 j 篇文档，$n_{i,j}$代表第 i 个词在第 j 篇文档地出现次数，$|D|$代表词典大小（即词的数量）。

整个算法流程：

① 利用爬虫得到数据集

② 使用 TF-IDF 编码向量化原始文本

③ 训练得到每一种分类的特征向量

④ 计算测试样本与每一种分类特征向量的余弦距离，求出最大距离对应的类别，该类别即分类器预测结果

根据上述流程，为了实现分类算法，我们总共需要使用两次 MapReduce：(1) 计算单词在所有数据集中出现的词频，用于 TF-IDF 编码的计算；(2) 训练时，求出每一种分类的特征向量。

不仅如此，经过爬虫得到的数据集，经过处理后会上传到 HDFS 分布式文件系统中，所有的 MapReduce 计算都会以 HDFS 文件系统作为存储媒介。

3. 实现细节

根据实现思路，总的流程图如图 7 所示：

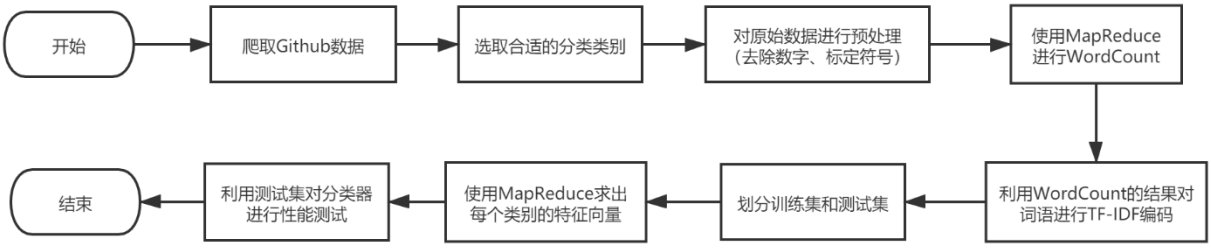


图 7 完整的流程图

3.1 爬取 Github 数据

具体的实现流程：

① 爬取代码仓库里所有的 Issue 类别

② 利用类别爬取所有的标题

③ 过滤含有中文字符的标题（中文英文混杂会加大识别的难度）

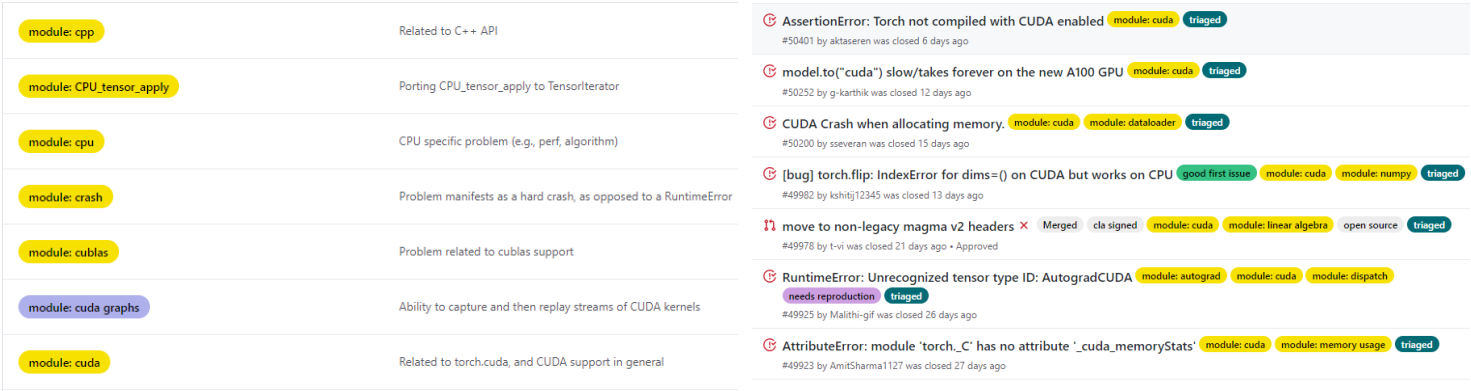


图 8 (左图) PyTorch Issues 的标签 (右图) 具体的 Issue 题目

Issue 的所有类别可以利用网址格式：https://github.com/pytorch/pytorch/labels?page=1，而具体类别的 Issue 题目则可以利用网址格式（以 module: cuda 为例子）：https://github.com/pytorch/pytorch/issues?q=label%3A%22module%3A+cuda%22+is%3Aclosed。

获取网页 html 文件并解析的代码（获取 label 和获取标题的代码类似）：

```
    # 请求获取 HTML
    html = requests.get(github_site + repository + label_site + f'?page
={i}')

    # 用 BeautifulSoup 解析 html
    obj = bf(html.text, 'html.parser')

    # 获取本页面所有 label
```

```
    label_list = obj.find_all(name='div', attrs={'class':'js-label-
list'})[0]
    for x in label_list.contents:
        if isinstance(x, str):
            continue

        label_name = ''.join(x.span.contents)
        if label_name.find('module') != -1:
            dataset[label_name] = []

    # 判断 Next Page 是否存在
    if len(obj.find_all(name='a', attrs={'class':"next_page"})) == 0:
        break
```

3.2 选取合适的分类类别

经过爬取数据与进一步统计，以下为 bug 数量前 20 的类别:

类别	数量
module: cuda	775
module: nn	650
module: autograd	512
module: tests	301
module: internals	250
module: bc-breaking	212
module: build	175
module: flaky-tests	174
module: third_party	149
module: binaries	124
module: sparse	112
module: android	97
module: bootcamp	93
module: nccl	62
module: infra	56
module: multiprocessing	51
module: type promotion	47
module: cublas	40
module: hub	40
module: tensorboard	32

从数量上看，前 10 的类别数量均超过了 100。为了避免因为类别数量差异巨大而导致分类有偏差，实验中我选取的是前 10 的类别，

也就是说，这是一个 10 分类任务。由于有些 bug 存在多标签，但为了简化问题，这里统一为单标签。

3.3 文本预处理

事实上，数字与标点符号对文本分类的影响很小，但对于机器识别的干扰却非常大。因此，我利用正则表达式去除了字母以外的所有符号。

实现代码:

```
def TextProcess(raw_text):
    # Keep Only English Text
    comp = re.compile('[^A-Z^a-z ]')

    # lower
    res = comp.sub(' ', raw_text).lower()
    return res
```

处理后的结果（示例）:

类别	文本
module: cuda	libtorch　put tensor from cpu to gpu is very slow
module: nn	print padding mode for conv modules if not zeros
module: autograd	gradient update of a sparse matrix results in a memory leak
module: tests	change pytorch tests to use non default cuda stream

3.4 WordCount

这部分的实现与 MapReduce 的经典示例 WordCount 程序完全一样。不同之处是，我采用了 Hadoop Streaming，利用 Python 代码来实现 Mapper 和 Reducer。Hadoop Streaming 是 Hadoop 提供的一种编程工具，允许用户用任何可执行程序和脚本作为 mapper 和 reducer 来完成 Map/Reduce 任务。

3.4.1 运行流程与代码

具体流程:

① 将数据上传到 HDFS

② Mapper 读取一行文本，将每个单词化为<word, 1>的形式输出

③ Reducer 读取<word, 1>，统计所有单词的次数

④ 从 HDFS 中下载结果

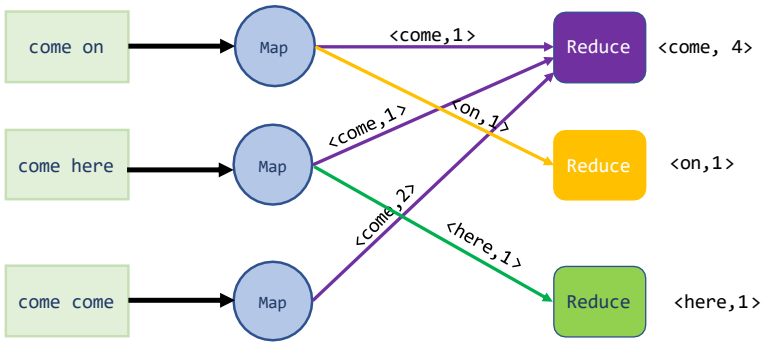


图 9 MapReduce 示意图

实现代码:

```
'''mapper.py'''
import sys

# input comes from STDIN
for line in sys.stdin:
    label, sentence = line.strip().split('\t', 1)

    # 将句子分割为词
    words = sentence.split()

    # 分割为<word, 1>
    for word in words:
```

```
        print(f'{word}\t{1}')


'''reducer.py'''
import sys

curr_word = None
curr_count = 0
word = None

for line in sys.stdin:
    # 输入格式 <word, count>
    word, count = line.strip().split('\t', 1)

    count = int(count)
    # 如果当前词与之前的词不同，则上一个单词计数完成
    if curr_word != word:
        if curr_word is not None:
            print(f'{curr_word}\t{curr_count}')
        curr_word = word
        curr_count = 0
    curr_count += count

# 最后一类词
if curr_word == word:
    print(f'{curr_word}\t{curr_count}')
```

3.4.2 实验结果

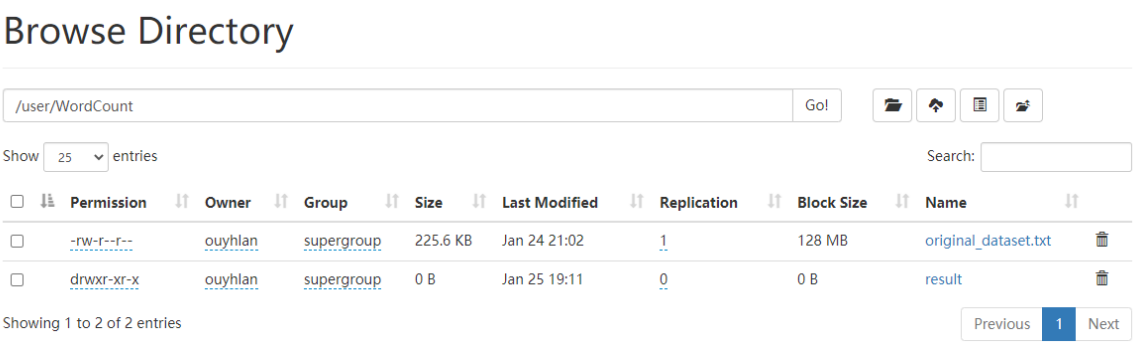
运行脚本代码:

```
hdfs dfs -mkdir /user/WordCount
hdfs dfs -put original_dataset.txt /user/WordCount
hadoop jar /home/ouyhlan/hadoop-3.3.0/share/hadoop/tools/lib/hadoop-
streaming-3.3.0.jar -D stream.non.zero.exit.is.failure=false \
-input "/user/WordCount" \
-output "/user/WordCount/result" \
-mapper "python3 ./mapper.py" -reducer "python3 ./reducer.py" \
-file ./*.py
hdfs dfs -get /user/WordCount/result ./
```

Hadoop 运行结果:

```
Map-Reduce Framework
        Map input records=3322
        Map output records=26992
        Map output bytes=223128
        Map output materialized bytes=277124
        Input split bytes=208
        Combine input records=0
        Combine output records=0
        Reduce input groups=3282
        Reduce shuffle bytes=277124
        Reduce input records=26992
        Reduce output records=3282
        Spilled Records=53984
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=154
        CPU time spent (ms)=3380
        Physical memory (bytes) snapshot=790777856
        Virtual memory (bytes) snapshot=2793775554560
        Total committed heap usage (bytes)=691535872
        Peak Map Physical memory (bytes)=301084672
        Peak Map Virtual memory (bytes)=1750236069888
        Peak Reduce Physical memory (bytes)=188674048
        Peak Reduce Virtual memory (bytes)=386238554112
```

hdfs 运行后结果：



3.5 文本向量化和划分训练集与测试集

3.5.1 文本向量化

按照思路里列出的公式进行计算即可，具体实现代码：

```
    # 计算所有词语 IDF 值
    doc_num = len(lines)
    vocab_idf = np.zeros(len(idx2vocab))

    for i in range(len(idx2vocab)):
        vocab_idf[i] = np.log2(doc_num/(vocab_stat[idx2vocab[i]] + 1))

    for i, line in enumerate(lines):
        encod = np.zeros(len(idx2vocab))
        label, sentence = line.strip().split('\t', 1)
```

```
        # 将句子分割为词
        words = sentence.split()

        # 分割为<word, 1>
        for word in words:
            encod[vocab2idx[word]] += 1

        # 计算 TF 值
        encod /= len(words)

        # 计算 TF-IDF 值
        encod *= vocab_idf
```

3.5.2 划分训练集与测试集

划分的方式按照每一个类别 80%的训练集和 20%的验证集。

实现代码:

```
fp_train, fp_test = open('TrainDataset', 'w'), open('TestDataset', 'w')
for label, sentences in dataset.items():
    train_num = int(len(sentences) * 0.8)
    for x in sentences[:train_num]:
        fp_train.write(f'{label}\t{x[0].tolist()}\n')

    for x in sentences[train_num:]:
        fp_test.write(f'{label}\t{x[0].tolist()}\t{x[1]}\n')
```

3.6 MapReduce 分类算法

3.6.1 运行流程与代码

分类算法的实现，主要是比较每一个类别提取出来的特征向量与待测试的样本之间的余弦距离。而特征向量的计算，则是通过将每一类训练集里的所有样本向量作平均操作。整个流程如图 10 所示，其中，#1 代表第一篇文档向量化后的结果，avg 即对括号里的向量进行平均操作。

具体流程:

① 将训练集数据上传到 hdfs 文件系统

② Mapper 将数据按照<class, vec>的方式进行输出

③ Reducer 读取 class 中所有的向量，对其进行平均操作，得到结果

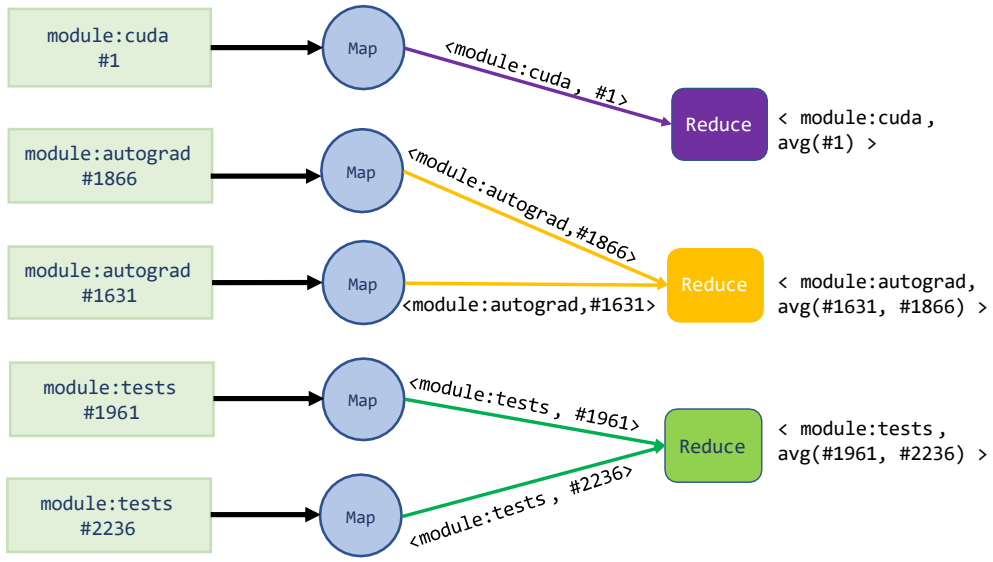


图 10 MapReduce 流程

实验代码:

```
'''mapper.py'''
import sys

# input comes from STDIN
for line in sys.stdin:
    label, encod = line.strip().split('\t', 1)
    label, encod = int(label), eval(encod)

    # <label, encoding>
    print(f'{label}\t{encod}')

'''reducer.py'''
import sys

curr_label = None
curr_count = 0
curr_encod = 0
label = None
```

```
for line in sys.stdin:
    label, encod = line.strip().split('\t', 1)
    label, encod = int(label), eval(encod)

    dim = len(encod)
    # 如果当前类别与之前的类别不同，则上一个类别计数完成
    if curr_label != label:
        if curr_label is not None:
            # 求平均数
            print(f'{curr_label}\t{[curr_encod[i] / curr_count for i in
 range(dim)]}')
        curr_label = label
        curr_count = 0
        curr_encod = [0] * dim
    curr_count += 1
    curr_encod = [curr_encod[i] + encod[i] for i in range(dim)]

# 最后一类
if curr_label == label:
    # 求平均数
    print(f'{curr_label}\t{[curr_encod[i] / curr_count for i in range(d
im)]}')
```

3.6.2 实验结果

运行脚本代码:

```
hdfs dfs -put TrainDataset /user/Train
hadoop jar /home/ouyhlan/hadoop-3.3.0/share/hadoop/tools/lib/hadoop-
streaming-3.3.0.jar -D stream.non.zero.exit.is.failure=false \
-input "/user/Train" \
-output "/user/Train/result" \
-mapper "python3 ./mapper.py" \
-reducer "python3 ./reducer.py" \
-file ./*.py
hdfs dfs -get /user/Train/result ./
```

Hadoop 运行结果：

```
Map-Reduce Framework
        Map input records=2655
        Map output records=2655
        Map output bytes=43896658
        Map output materialized bytes=43907290
        Input split bytes=184
        Combine input records=0
        Combine output records=0
        Reduce input groups=10
        Reduce shuffle bytes=43907290
        Reduce input records=2655
        Reduce output records=10
        Spilled Records=5310
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=248
        CPU time spent (ms)=3830
        Physical memory (bytes) snapshot=889065472
        Virtual memory (bytes) snapshot=3677073874944
        Total committed heap usage (bytes)=850395136
        Peak Map Physical memory (bytes)=303943680
        Peak Map Virtual memory (bytes)=1836371537920
        Peak Reduce Physical memory (bytes)=285372416
        Peak Reduce Virtual memory (bytes)=1027556696064
```

3.7 测试与结果分析

3.7.1 测试结果

以刚刚训练得到的特征向量作为每个类别的特征向量，利用测试集进行测试，得到下面的结果。

Module	cuda	nn	autograd	tests	internals	bc-breaking	build	flaky-tests	third_party	binaries	overall
Total	155	130	103	61	50	43	35	35	30	25	667
Correct	57	62	64	21	13	22	18	26	14	17	314
Accuracy	36.77%	47.69%	62.14%	34.43%	26%	51.16%	51.43%	74.29%	46.67%	68%	47.08%

图 11 分类器准确率

由于代码 Bug 这一类文本本身存在有大量的少见的专有名词，且文本的不规范性也是普遍存在。因此，虽然最终得到的分类器性能不佳，但还是比随机算法要高出不少，甚至在某些 Bug 类别处的准确

率能达到 74.29%。这一定程度上说明分类算法的有效性。

3.7.2 结果分析

Label		Prediction		Text
module: nn	0.4214	module: nn	0.4214	quantized conv d module
module: autograd	0.3769	module: autograd	0.3769	segmentation fault when calling autograd grad in autograd function backward
module: flaky-tests	0.3067	module: flaky-tests	0.3067	test proper exit is flaky
module: flaky-tests	0.2735	module: flaky-tests	0.2735	testquantizedops test adaptive avg pool d is flaky

图 13 预测正确的例子

Label		Prediction		Text
module: cuda	0.1203	module: autograd	0.1278	fix at view
module: cuda	0.0582	module: tests	0.1865	refactor and improve randperm tests
module: nn	0.1409	module: tests	0.1726	fix tests for multi head attention
module: autograd	0.0761	module: internals	0.1691	include c namespace into torch

图 12 预测错误的例子

图 12 和 13 分别是一些预测正确的例子和预测错误的例子。从正确的例子来看，当 Bug 标题出现对应类别的词汇时，测试例子与对应分类特征向量的余弦距离比较大。这说明，分类器的预测很大程度上依赖一些关键词。

然而，从错误的例子来看，这种基于关键词的预测可能会因为关键词提取错误而导致预测错误的发生。不仅如此，分类准确率低的原因也与标题意思含糊有关。对于预测错误的的第 1 个例子和第 4 个例子，其文本表达意思本身就含糊不清，这也导致分类错误发生的概率大大增大。

4. 遇到的问题

4.1 Java 路径和版本导致 Hadoop 环境配置失败

实验中，我使用的 Hadoop 版本为 3.3.0。使用 Java 15 的时候，Hadoop 会在 MapReduce 阶段出现一些奇怪的错误。而 Hadoop 官方推荐的 Java 版本为 Java 1.8。不仅如此，Java 的安装路径如果出现带空格的文件夹名字，Hadoop 也无法正常运行。

4.2 Hadoop Streaming 在 Windows 环境中无法运行

在环境配置完成后，我使用 Hadoop 官方提供的 example 进行测试，MapReduce 整个流程可以顺利完成。然而，在使用 Hadoop Streaming 时，则会出现 HADOOP_HOME 和 hadoop.home.dir 无法识别的问题。最终，我切换到 Windows Subsystem for Linux 的环境才成功搭建好 Hadoop。

5. 总结

本次实验我使用爬虫技术从网络中爬取 Github 的 Bug 信息，并利用 MapReduce 的计算框架对软件 Bug 进行自动分类。这实际上是一个大数据分析的编程场景。通过实践，我感受到了 MapReduce 计算框架带来的便利性，也理解了他频繁读写磁盘带来的性能损耗的问题。

利用在分布式系统课程学习到的分布式计算和分布式文件系统，我使用少量的数据模拟了整个大数据分析的流程。虽然最终分类器性能不算太理想，但利用分布式集群的优势，随着数据规模逐步增加，我相信分类算法的准确性会逐步提高的！