# 分布式系统作业

## 期末大作业
## 基于 MapReduce 的软件 Bug 分类

姓名：曾慧蕾
班级：系统结构班
学号：21307358

# 一、题目描述

## 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。

## 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

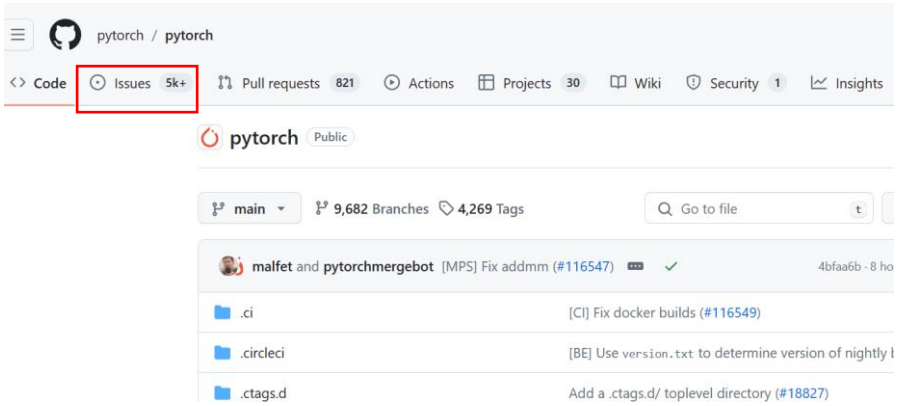
## 3、 参考实现

参考 Hadoop 或者 Spark 中的相关算法案例

# 二、数据选取

GitHub 仓库中有一栏 issue 可供用户在项目的仓库中提交 bug 报告，描述所遇到的问题、复现步骤和期望的行为。其他开发者可以通过查看这些 bug 报告来理解并修复软件中的问题，也可以参与讨论、提供解决方案或验证 bug 是否已被修复。由于这些分类标签都是人为添加的，不会采用统一的风格，因此使用程序对这些未处理的软件 bug 进行分类的难度很

大。而在本次 mapreduce 实验中，我的目的即是对这些 bug 进行分类。



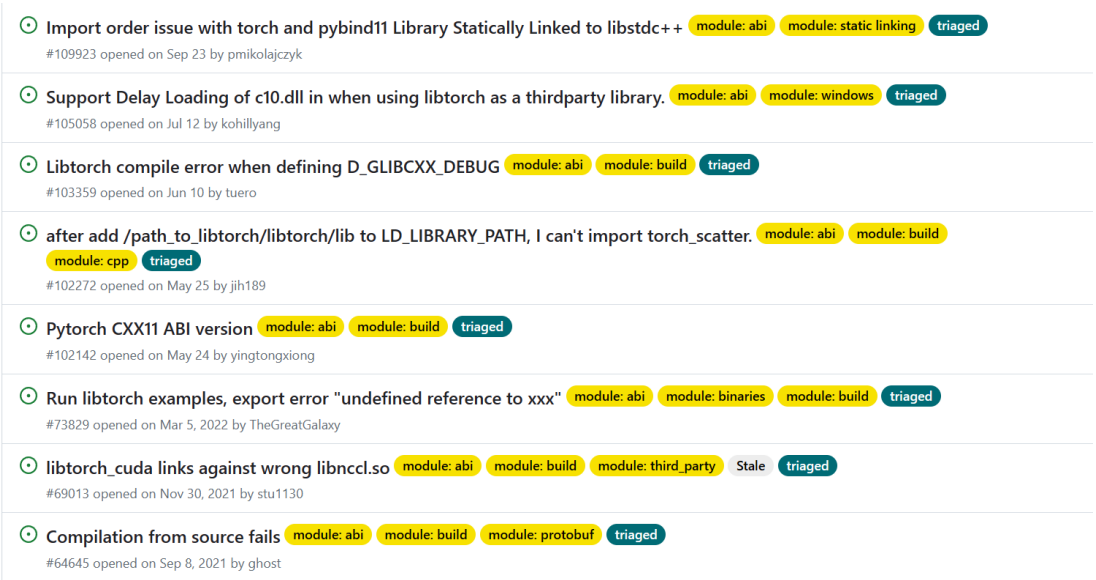

为了尽量减少无关因素的影响，对抓取的数据源要求如下：

1、 来自统一的仓库，尽量针对一个领域。如本实验使用 pytorch 库。

2、 数据源中的 bug 具有一定的格式和标准化，方便分类器处理和分析

3、 数据源的规模应该足够大，确保分类器训练数据充分且具有代表性。

4、 数据应该具有一定的平衡性，每个类别的 bug 数量应该差不多




以上图为例，要抓取的即是这些带标签的 bug。

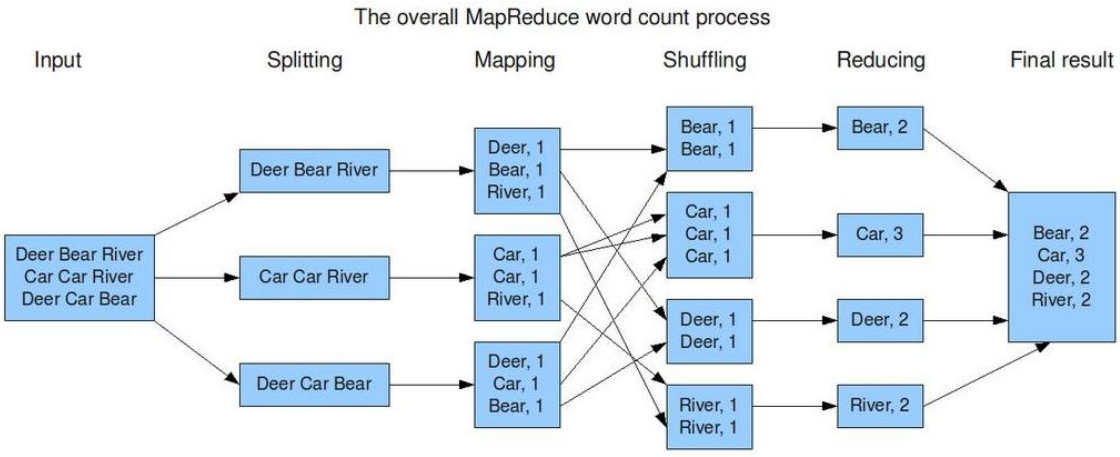
## 三、MapReduce 相关原理

MapReduce 是一个用于分布式计算的编程框架，可以帮助用户开发

基于 Hadoop 进行数据分析的应用程序。MapReduce 的核心功能是利用用户编写的代码与自带的组件形成一个完整的分布式运算程序，并在 Hadoop 集群上并发执行。

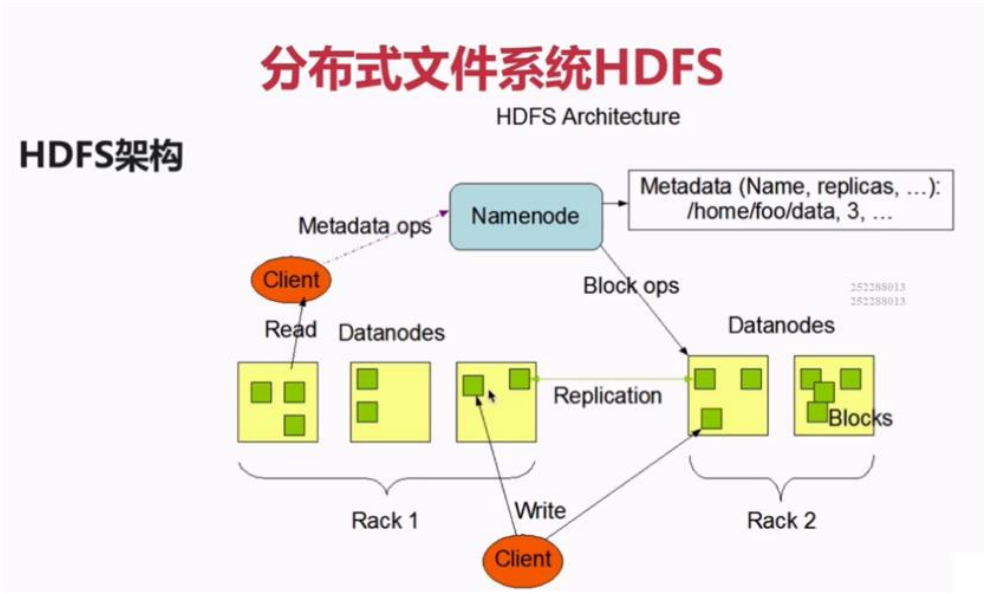
在 MapReduce 中，任务被分为两个阶段：Map 阶段和 Reduce 阶段。在 Map 阶段中，输入数据被划分为多个小块，每个小块由不同的计算节点处理。Mapper 会执行用户定义的逻辑代码，将输入数据转换为<key, value>的形式，并输出中间结果。这些中间结果会在 shuffle 阶段（无须用户编写）排序和分组，然后传递给 Reduce 阶段。在 Reduce 阶段中，相同 key 的中间结果会被传递到同一个计算节点进行处理。Reducer 同样会执行用户定义的逻辑代码，对中间结果进行归约操作，并生成最终的输出结果。最后，所有输出结果会被收集和整合起来，形成最终的计算结果。

下面是 mapreduce 里一个经典程序 wordcount 的流程图，该程序用于单词计数并输出结果，体现了 mapreduce 的流程与思想：分而治之。




在本实验中，我们除了编写 mapper 和 reducer，还要关注的部分是处理文件的存放位置：HDFS。HDFS 是 Hadoop 的分布式文件系统，用于存储和管理大规模数据，可以在廉价的普通机器上部署，构建成一个

能够处理大量数据的集群。




HDFS 的设计理念是将大文件切分成多个数据块，并将这些数据块分布存储在集群的不同节点上，以实现数据的并行处理和高可靠性。每个数据块都会有多个副本存储在不同的节点上，以防止数据丢失或故障。

在实验中，想要使用 mapreduce 方法需要先在 HDFS 集群上上传待处理的文件，再使用 Hadoop streaming 方法对数据进行处理。这一部分在解决方案中会进一步说明。

# 四、解决方案

- **数据获取**

**对于数据爬取部分**，在对 issue-label 进行粗略的翻阅后，我发现其中的 label 风格众多，但并不是所有 label 都与 title 关联明显。为了应对二中数据获取的要求，这里统一规定爬取 label 中含有“module：“的部分，一是风格统一，二是数量足够，很好地满足了对于数据的期望。

获取 label：

```python
def get_label():
    url = "https://github.com/pytorch/pytorch/labels"  # GitHub Bug页面的URL
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.124 Safari/5
    }  # 添加User-Agent头部，模拟浏览器请求

    for i in range(1, 5): # 经检查 共有8页
        # 为了格式统一 仅爬取头部为module的标签
        response = requests.get(url + f"?page={i}&q=module", headers=headers, timeout=10)
        if response.status_code == 200:
            print(f"页面{i}请求成功")
            soup = BeautifulSoup(response.text, "html.parser")
            labels = soup.find_all("a", class_ = "IssueLabel")
            for label in labels:
                label_name = label.get('data-name')
                if "module" in label_name and label_name not in original_data:
                    original_data[label_name] = []
                    # print(label_name)

            flag = soup.find_all("a", class_ = "next_page")
            if len(flag) == 0: # 不存在下一页
                break

        else:
            print(f"页面{i}请求失败，状态码：{response.status_code}")
            break
```

get_label() 遍历了 PyTorch 的 GitHub Issues 中前 4 页的标签，筛选出其标签名包含 "module" 的标签，并将它们存储到 original_data 字典中。

在获取 label 后，通过总结各 label 链接的特征，进一步爬取对应 label 中的 title：

```python
def get_title(): # 只爬取含module的
    base_url = "https://github.com/pytorch/pytorch/labels/"  # GitHub Bug页面的URL
    for label in original_data.keys():
        print(f"正在处理{label}标签")
        temp_label = label[8:]
        for i in range(1, 10): # 一个页面有25个标题，避免一个标签爬太多标题导致数据不平衡
            tail = f"module:%20{temp_label}?page={i}&q=is%3Aopen+label%3A%22module%3A+{temp_label}%22"
            url = base_url + tail
            response = requests.get(url, timeout=10)

            if response.status_code == 200:
                print(f"{label}: 页面{i}请求成功")
                soup = BeautifulSoup(response.text, "html.parser")
                titles = soup.find_all('a', class_='Link--primary v-align-middle no-underline h4 js-navigation-open markdown-title')
                for title in titles:
                    title_text = title.text.strip()
                    original_data[label].append(title_text)
                    print(title_text)

                flag = soup.find_all("a", class_ = "next_page")
                if len(flag) == 0: # 不存在下一页
                    break

            else:
                print(f"页面{i}请求失败，状态码：{response.status_code}")
                break
```

get_title() 遍历了 original_data 字典中所有的标签，爬取每个标签下前 10 页的标题，并将它们存储到对应标签的值列表中。

两个函数的实现都使用了 BeautifulSoup 库进行 HTML 解析，并模拟浏览

器请求获取页面内容。并通过添加 User-Agent 头来避免被防爬虫机制屏蔽。

最终粗略爬取的结果如下：

```
module: docs    Fix docstring errors in default_hooks.py, post_localSGD_hook.py, debugging_hooks.py, utils.py,
module: docs    Fix docstring errors in post_localSGD_optimizer.py, functional_sgd.py, _functional_collectives
module: docs    Fix docstring errors in spectral_ops_fuzz_test.py, simple_timeit.py, timer_interface.py, op_be
module: docs    Fix docstring errors in loss.py
module: docs    Fix docstring errors in nadam.py, radam.py, sgd.py, anomaly_mode.py, rprop.py, __init__.py, sw
module: docs    Fix docstring errors in _torch_docs.py, serialization.py, overrides.py, _utils.py
module: docs    Fix docstring errors in __init__.py, _tensor_docs.py, _meta_registrations.py, _tensor.py
module: docs    Fix docstring errors in _guards.py, _ops.py, _jit_internal.py, functional.py, _tensor_str.py,
module: docs    Fix docstring errors in _VF.py, _appdirs.py, hub.py, _classes.py, _storage_docs.py, _linalg_ut
module: docs    Requesting to add a section to the Installing C++ Distributions of PyTorch documentation for A
module: docs    ~ Docathon H2 2023 ~
module: docs    maximum Python version supported is not indicated
module: docs    Incorrect docstring / documentation for torch.nn.functional.scaled_dot_product_attention in 2.
module: docs    cuda/tf32 docs are outdated
```

**对于数据处理部分**，由原始的数据可见其中含有不少非英文字符以及大小写问题，这些数据会影响后续文本分类的结果，因此数据处理阶段的工作就是对 title 进行清洗，删去非英文字符以及将文本转化为小写，以及对原始数据进行训练集和测试集的划分。

```python
def clean_text(text):
    cleaned_text = re.sub('[^a-zA-Z\s]', ' ', text)
    cleaned_text = cleaned_text.lower()
    return cleaned_text
```

数据清理部分，使用 re 正则表达进行过滤。

```python
random.shuffle(data) # 随机打乱数据顺序

train_size = int(0.8 * len(data))
test_size = len(data) - train_size
train_data = sorted(data[:train_size])
test_data = sorted(data[train_size:])

with open('D:\\VSCode_code\\python\\file\\FBSxitong\\dataset\\train.txt', 'w', encoding='utf-8') as f: # i
    for line in train_data:
        f.writelines(line)
        f.writelines('\n')
print("训练集写入完毕！")

with open('D:\\VSCode_code\\python\\file\\FBSxitong\\dataset\\true_test.txt', 'w', encoding='utf-8') as f:
    for line in test_data:
        f.writelines(line)
        f.writelines('\n')
print("测试集(处理前)写入完毕！")

with open('D:\\VSCode_code\\python\\file\\FBSxitong\\dataset\\use_test.txt', 'w', encoding='utf-8') as f:
    for i, line in enumerate(test_data, start=1): # <label, <"Test", ID, title>>
        _, title = line.strip().split('\t')
        tail = f"{i} {title}"
        for label in labels:
            test_line = f"{label}\t{tail}"
            f.write(test_line)
            f.writelines('\n')
print("测试集(处理后)写入完毕！")
```

这里测试集分了两个，一个用于检验最后的预测结果，一个用于测试，其中每个 title 会附上记录到的所有 label，在后续计算该 title 属于当前 label 的概率。

最终，选择出来的数据总结如下：

| Title | Sum | Test | Train |
|---|---|---|---|
| autograd | 200 | 39 | 161 |
| cpp | 200 | 45 | 155 |
| cuda | 200 | 45 | 155 |
| docs | 200 | 44 | 156 |
| nn | 200 | 51 | 149 |
| numpy | 200 | 54 | 146 |
| onnx | 200 | 39 | 161 |
| optimizer | 200 | 39 | 161 |
| tests | 200 | 44 | 156 |

- **MapReduce**

在查阅其他使用 mapreduce 进行文本分类的博客后，我选择使用朴素贝叶斯分类算法来对本次任务进行分类。算法原理这里不再赘述。现在将重点放在 mapreduce 上，下面介绍我的设计思想。

Mapreduce 部分的思路如前文所述，使用分而治之的思想。为了体现分层思想及避免代码臃肿，我将本次任务分为两个 mapreduce 工作。

1、 mapreduce1：

对数据集进行统计，统计其中出现的 label 以及 word 数量并以键值对形式存储，为 mapreduce2 准备新的数据集。具体来说：

**Mapper：**分别统计单个的 label 以及 word，word 要求附在 label 后。

输入格式：<label，title>

输出格式：<label, <label, word, 1>>

<label, 1>

**Reducer：**对 mapper 输出的键值对进行合并统计并输出。

输入格式：<label, <label, word, 1>>

<label, 1>

输出格式：<label, <label, label_count, word, word_count>>

2、 mapreduce2：

根据 mapreduce1 提供的训练集信息训练文本分类器，并对测试集中的数据进行分类预测。

**Mapper：**根据输入的键值对格式进行解析，将词频信息和测试文档信息分别发射出去

输入格式：<label, <label, label_count, word, word_count>>

<label, <ID, title>>

输出格式：<label, <label, label_count, word, word_count>>

<label, <ID, title>>

**Reducer：**根据 mapper 提供的汇总信息，对测试集进行预测。

输入格式：<label, <label, label_count, word, word_count>>

<label, <ID, title>>

输出格式：<ID, <label, p>>

**Mapreduce1：**将输入的文本转换为适合进行贝叶斯分类的形式

Mapper：

```
import sys
#!/usr/bin/env python
for line in sys.stdin:
    # 去除首尾空格并分割单词
    label, title = line.strip().split('\t')
    title = title.split()

    # 对每个单词生成键值对
    for word in title:
        label_ = f"{label}"
        word_ = f"{word} 1"
        tail = f"{label_} {word_}"
        print(f"{label}\t{tail}")
    print(f"{label}\t1")
```

遍历每个单词，将标签和单词组成键值对；标签也单独组成一个键值对

Reducer：

```
label_record = {}
word_record = {}

for line in sys.stdin:
    label, value = line.strip().split('\t')
    value = value.split()
    if len(value) == 4:
        word = value[2]
        word_count = int(value[3])

        if label not in label_record:
            label_record.setdefault(label, 0)

        if word in word_record:
            word_record[word][0] += word_count
        else:
            value = [word_count, label]
            word_record.setdefault(word, value)

    else:
        label_count = int(value[0])
        if label in label_record:
            label_record[label] += label_count
        else:
            label_record.setdefault(label, label_count)
```

```
# word的数量应该远多于label，所以遍历word_record并输出
for key, value in word_record.items():
    label = value[1]
    word_count = value[0]
    word = key
    label_count = label_record[label]
    f1 = f"{label} {label_count} {word} {word_count}"
    res = f"{label}\t{f1}"

    print(res)
```

对 mapper 生成的键值对进行处理，统计每个标签和单词出现的次数。最终输出每个单词与对应标签的统计结果。

以管理员权限运行命令行，运行 Hadoop 集群，在 hdfs 中上传好数据

```
D:\hadoop-3.1.2\sbin>hdfs dfs -put D:\VSCode_code\python\file\FBSxitong\dataset\train.txt /input/train_data

D:\hadoop-3.1.2\sbin>hdfs dfs -put D:\VSCode_code\python\file\FBSxitong\dataset\use.txt /input/test_data
put: `D:/VSCode_code/python/file/FBSxitong/dataset/use.txt': No such file or directory

D:\hadoop-3.1.2\sbin>hdfs dfs -put D:\VSCode_code\python\file\FBSxitong\dataset\use_test.txt /input/test_data
```

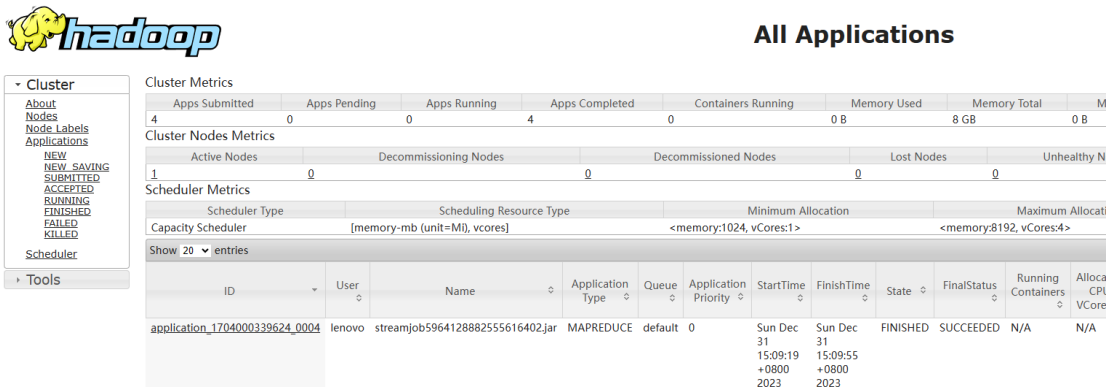
目录结构如下所示：

| Permission | Owner | Group | Size | Last Modified | Replication | Block Size | Name |
|---|---|---|---|---|---|---|---|
| drwxr-xr-x | lenovo | supergroup | 0 B | Dec 30 17:21 | 0 | 0 B | input |
| drwxr-xr-x | lenovo | supergroup | 0 B | Dec 30 17:20 | 0 | 0 B | output |
| drwx------ | lenovo | supergroup | 0 B | Dec 30 11:15 | 0 | 0 B | tmp |
| drwxr-xr-x | lenovo | supergroup | 0 B | Dec 30 17:10 | 0 | 0 B | usr |

然后运行脚本代码：

```
# mr1
hdfs dfs -rm -r /output/train_result
hadoop jar share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar ^
-mapper "python D:/VSCode_code/python/file/FBSxitong/train_mapresuce/mapper.py" ^
-reducer "python D:/VSCode_code/python/file/FBSxitong/train_mapresuce/reducer.py" ^
-input /input/train_data/train.txt ^
-output /output/train_result
```

监控页面：




运行日志:

```
Map-Reduce Framework
        Map input records=1600
        Map output records=17254
        Map output bytes=589523
        Map output materialized bytes=624043
        Input split bytes=200
        Combine input records=0
        Combine output records=0
        Reduce input groups=9
        Reduce shuffle bytes=624043
        Reduce input records=17254
        Reduce output records=2781
        Spilled Records=34508
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=227
        CPU time spent (ms)=6682
        Physical memory (bytes) snapshot=755343360
        Virtual memory (bytes) snapshot=1248346112
        Total committed heap usage (bytes)=579338240
        Peak Map Physical memory (bytes)=303136768
        Peak Map Virtual memory (bytes)=463814656
        Peak Reduce Physical memory (bytes)=176214016
        Peak Reduce Virtual memory (bytes)=367820800
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=129241
File Output Format Counters
        Bytes Written=118552
```

随后到 hdfs 中指定的路径中获得 mapreduce1 输出结果即可。部分展示如下

```
module: autograd    module: autograd 186 with 236
module: autograd    module: autograd 186 csr 5
module: autograd    module: autograd 186 matrix 11
module: autograd    module: autograd 186 in 284
module: autograd    module: autograd 186 inference 7
module: autograd    module: autograd 186 mode 39
module: autograd    module: autograd 186 throws 8
module: autograd    module: autograd 186 an 47
module: autograd    module: autograd 186 exception 5
module: autograd    module: autograd 186 max 17
module: autograd    module: autograd 186 pool 9
module: autograd    module: autograd 186 d 91
module: autograd    module: autograd 186 indices 5
module: autograd    module: autograd 186 self 7
module: autograd    module: autograd 186 shouldn 2
module: autograd    module: autograd 186 t 68
```

**Mapreduce2：**对测试集进行贝叶斯文本分类任务。

**Mapper：**

```python
import sys
#!/usr/bin/env python

for line in sys.stdin:
    label, title = line.strip().split('\t')
    print(f"{label}\t{title}")
```

简单的输出，在 shuffle 中进行不显示的排序。

Reducer：

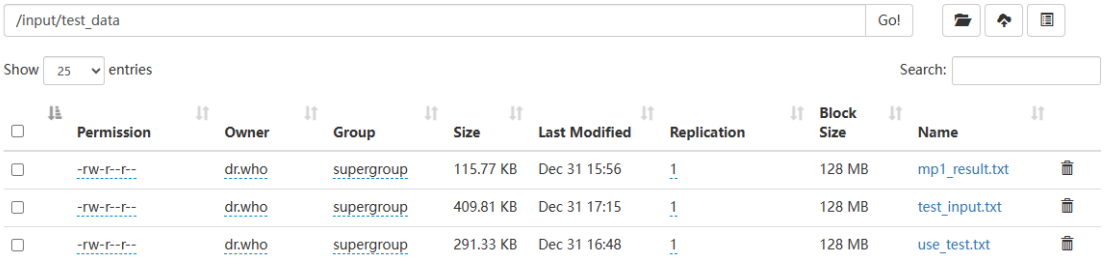
```python
labels = {}
test_id = ""
test_title = ""
probabilities = []
total = 0

# 先训练
for line in sys.stdin:
    label, value = line.strip().split("\t")
    value = value.split()
    if value[0] == "module:": # 训练数据
        _, label, label_count, word, word_count = value[0], value[1], value[2], value[3], value[4]
        label_count = int(label_count)
        word_count = int(word_count)
        if label not in labels:
            labels[label] = {'total': 0}

        labels[label]['total'] = label_count # label出现次数
        total += label_count
        labels[label][word] = labels[label].get(word, 0) + word_count # label下word出现次数
    else:
        continue
```

```python
# 后测试
for line in sys.stdin:
    label, value = line.strip().split("\t")
    value = value.split()
    if value[0] != "module:":  # 测试数据
        test_id = value[0]
        test_title = value[1:]
        # 计算test_title属于当前label的概率
        p = get_p(test_title, label[8:], labels, total)
        probabilities.append((test_id, label, p))
    else:
        continue

pro_set = {}
for prob in probabilities:
    id, label, p = int(prob[0]), prob[1], prob[2]
    if id not in pro_set:
        pro_set[id] = []
    pro_set[id].append((label, p))
    pro_set[id] = sorted(pro_set[id], key=lambda x:x[1], reverse=True) # 把概率最高的标签摆在最前面

for id, value in pro_set.items():
    output = f"{test_id}\t{value[0][0]}"
    print(output)
```

```python
def get_p(test_title, label, labels, total, k=1): # k是平滑系数
    label_prob = labels[label]['total'] / total # 计算P(label)

    # 分割test_title，计算每个词出现的概率
    words = test_title[:]
    word_probs = [((labels[label].get(word, 0) + k) / (labels[label]['total'] + k * len(labels[label]))) for word in words]

    p_test_label = math.prod(word_probs)  # 计算P(test_title | label)
    p_label_test = p_test_label * label_prob  # 计算P(label | test_title) = P(test_title | label) * P(label)

    return p_label_test
```

进行文本分类

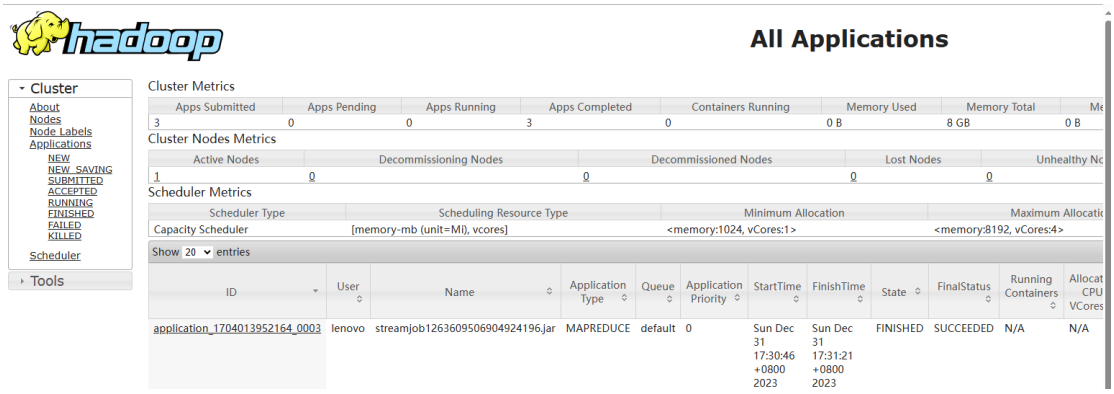
将 mapreduce1 中获得的结果与要使用的测试集进行合并处理后，上传到 hdfs 中为 mapreduce2 预测使用。




运行以下脚本代码：

```
# mr2
hdfs dfs -rm -r /output/test_result
hadoop jar share/hadoop/tools/lib/hadoop-streaming-3.1.2.jar ^
-mapper "python D:/VSCode_code/python/file/FBSxitong/test_mapreduce/mapper.py" ^
-reducer "python D:/VSCode_code/python/file/FBSxitong/test_mapresuce/reducer.py" ^
-input /input/test_data/test_input.txt ^
-output /output/test_result
```

监控如下：




运行日志如下：

```
Map-Reduce Framework
        Map input records=6381
        Map output records=6381
        Map output bytes=413352
        Map output materialized bytes=426207
        Input split bytes=208
        Combine input records=0
        Combine output records=0
        Reduce input groups=9
        Reduce shuffle bytes=426207
        Reduce input records=6381
        Reduce output records=9
        Spilled Records=12762
        Shuffled Maps =2
        Failed Shuffles=0
        Merged Map outputs=2
        GC time elapsed (ms)=235
        CPU time spent (ms)=6385
        Physical memory (bytes) snapshot=746647552
        Virtual memory (bytes) snapshot=1240903680
        Total committed heap usage (bytes)=575143936
        Peak Map Physical memory (bytes)=276807680
        Peak Map Virtual memory (bytes)=419880960
        Peak Reduce Physical memory (bytes)=196333568
        Peak Reduce Virtual memory (bytes)=406016000
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=423746
File Output Format Counters
        Bytes Written=1994879
```

到 hdfs 中获取运行结果，进行分析。

# 五、实验结果

```python
with open(test_path, "r", encoding="utf-8") as f:
    test_data = f.readlines()

labels = {}
for line in test_data:
    label, _ = line.strip().split('\t')
    labels[label] = {'total': 0, 'correct': 0}

with open(predict_path, "r", encoding="utf-8") as f:
    predict_data = f.readlines()

for line in predict_data: # <ID, <label, p>>
    test_id, predict_label = line.strip().split('\t')
    true_label, _ = test_data[int(test_id)-1].strip().split('\t')

    if true_label not in labels:
        labels[true_label] = {'total': 0, 'correct': 0}

    labels[true_label]['total'] += 1
    sum_total += 1
    if true_label == predict_label:
        labels[true_label]['correct'] += 1
        sum_correct += 1

for label, info in labels.items():
    total = info['total']
    correct = info['correct']
    accuracy = correct / total if total != 0 else 0
    print(f"{label}: Total: {total}, Correct: {correct}, Accuracy: {accuracy:.4f}")

print(f"ALL: Total:{sum_total}, Correct:{sum_correct}, Accuracy:{(sum_correct / sum_total):.4f}")
```

使用以上代码对 mapreduce2 得到的结果与 true_test_data.txt 中的数据比对，运行后得到结果如下：

```
PS D:\VSCode_code\python\file\FBSxitong> python -u "d:\VSCode_co
module: autograd: Total: 39, Correct: 17, Accuracy: 0.4359
module: cpp: Total: 45, Correct: 14, Accuracy: 0.3111
module: cuda: Total: 45, Correct: 16, Accuracy: 0.3556
module: docs: Total: 44, Correct: 15, Accuracy: 0.3409
module: nn: Total: 51, Correct: 23, Accuracy: 0.4510
module: numpy: Total: 54, Correct: 21, Accuracy: 0.3889
module: onnx: Total: 39, Correct: 16, Accuracy: 0.4103
module: optimizer: Total: 39, Correct: 14, Accuracy: 0.3590
module: tests: Total: 44, Correct: 14, Accuracy: 0.3182
ALL: Total:400, Correct:150, Accuracy:0.3750
PS D:\VSCode_code\python\file\FBSxitong>
```

由图可知经两次 mapreduce 运行出来的结果平均正确率达到了 37.5%。由于本次训练数据集仅为 2000，每个标签下训练集仅有 200，不能涵盖更多与标签有关

的关键词。在这个数据规模下到达 30%以上的正确率，个人认为效果是可观的。也说明了本次任务使用 mapreduce 进行分类是成功的。

# 六、遇到的问题及解决方法

1、 hadoop 监控页面一直不可访问

在刚开始搭建 Hadoop 环境时遇到的问题，表现为 hdfs 可访问但 Hadoop 监控页面不可见。一开始我以为是 8088 端口监听服务的问题，在排查后无奈重装 Hadoop 才解决。最后复盘时发现是 hdfs-site.xml 部分的配置问题。最后配置如下才解决：

```
<configuration>
 <property>
    <name>dfs.replication</name>
    <value>1</value>
 </property>
 <property>
    <name>dfs.namenode.name.dir</name>
    <value>/D:/hadoop-3.1.2/data/namenode</value>
 </property>
 <property>
      <name>dfs.datanode.data.dir</name>
      <value>/D:/hadoop-3.1.2/data/datanode</value>
 </property>
<property>
  <name>dfs.permissions</name>
  <value>false</value>
</property>

</configuration>
```

2、 在执行 mapreduce 任务时一直报错如下：

```
2023-12-30 19:50:26,020 INFO mapreduce.Job: Task Id : attempt_1703932698898_0003_m_000001_2, Status : FAILED
Error: java.lang.RuntimeException: PipeMapRed.waitOutputThreads(): subprocess failed with code 2
        at org.apache.hadoop.streaming.PipeMapRed.waitOutputThreads(PipeMapRed.java:325)
        at org.apache.hadoop.streaming.PipeMapRed.mapRedFinished(PipeMapRed.java:538)
        at org.apache.hadoop.streaming.PipeMapper.close(PipeMapper.java:130)
        at org.apache.hadoop.mapred.MapRunner.run(MapRunner.java:61)
        at org.apache.hadoop.streaming.PipeMapRunner.run(PipeMapRunner.java:34)
        at org.apache.hadoop.mapred.MapTask.runOldMapper(MapTask.java:465)
        at org.apache.hadoop.mapred.MapTask.run(MapTask.java:349)
        at org.apache.hadoop.mapred.YarnChild$2.run(YarnChild.java:174)
        at java.security.AccessController.doPrivileged(Native Method)
        at javax.security.auth.Subject.doAs(Subject.java:422)
        at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1729)
        at org.apache.hadoop.mapred.YarnChild.main(YarnChild.java:168)
```

在网上查询说是分配的容器内存不够大，才导致的任务中断。

解决：打开 mapred-site.xml 文件，配置如下：

```
<configuration>
  <property>
     <name>mapreduce.job.tracker</name>
     <value>hdfs://master:8020</value>
     <final>true</final>
  </property>
  <property>
     <name>mapreduce.framework.name</name>
     <value>yarn</value>
  </property>

  <property>
    <name>mapreduce.map.memory.mb</name>
    <value>2048</value>
  </property>

  <property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>4096</value>
  </property>

</configuration>
```

然后重启 Hadoop 集群，让新的内存配置生效。

3、在 Hadoop 集群运行正常，代码正确无误的情况下 mapreduce 任务一直报错

查询资料得知 mapreduce 对文本内容敏感，需要将其中多余的空行手动删除。

4、脚本代码中调用 hdfs 上的 py 文件一直报错，后面改用了本地的 py 文件才解

决。

# 总结

经本实验后我对于分布式计算和大数据处理的理解有了进一步的加深。我了解了 MapReduce 作为分布式计算模型，通过将大规模数据分成小块，能够在多台计算机上并行处理，充分利用集群中的计算资源，快速高效地处理大量数据。对于 Hadoop 集群及其命令也有了一定的了解。

并且，我对于爬虫技术也有了一定的收获，学会了网页解析、数据请求和处理、数据存储、反爬虫策略应对的基本知识，能够高效地获取互联网上的数据，并结合其他技术进行数据分析和应用。

对于未来，我期望能够进一步探索和应用分布式计算和大数据处理技术。随着数据规模的不断增大，传统的串行计算方法已经无法满足需求，因此深入了解和掌握分布式计算的理论和实践是非常重要的。我希望能够继续学习和研究相关技术，探索更高效、可扩展的大数据处理方案，并将其应用于解决实际问题。