# HW4:MapRed


0. Open Source and Duplication
1. Refs
2. 问题重述
3. 实验过程
    3.1. 分词
        为什么要做分词
        jieba分词器
        分词pipeline
        具体实现
        另一条道路：AI分词
    3.2. MapReduce框架统计词频
        项目结构
        具体实现
    3.3. TOP-K词频求解
        选择什么算法？
        具体实现
    3.4. 运行注意事项
4. 运行结果


# 0. Open Source and Duplication

```
https://github.com/Liyanhao1209/BDL.git
```

hw4分支，包括了python脚本和java代码。

# 1. Refs

MapReduce编程实践(Hadoop3.1.3)_厦大数据库实验室博客 (xmu.edu.cn)

# 2. 问题重述

给定一个中文文档，需要先对其进行分词。随后，利用Hadoop自带的MapReduce框架，统计词频，输出词频为top K的单词。

# 3. 实验过程

三大步：

1. 分词
2. MapRed统计词频
3. 排序输出top K的单词

对于具体实现时的pipeline，我采用如下解决方案：

1. python + jieba text splitter
2. java + Hadoop jar dependency
3. Linux shell + awk + sort

## 3.1. 分词

### 为什么要做分词

英文文档几乎不需要做预处理，因为任意两个相邻单词间必有标点符号或者空格，所以很容易就能获得每个单词。但中文词语之间没有这样的分隔符。需要我们进行分词。

### jieba分词器

这里使用了jieba的分词器。jieba对中文文档分割提供了三种模式：

1. 精确模式/默认模式：对待分的句子进行全词匹配，找出所有可能的词语，然后依据词典中的词语频率计算出最可能的切分结果。
2. 全模式：全模式会将待分词的文本中所有可能的词语都进行匹配，因此会产生大量的冗余词语。在全模式下，jieba分词库会对待分的句子进行正向最大匹配，即从前往后扫描文本，找到最长的可能词语，然后将其切分出来，再从新的位置开始继续扫描。很显然我们本次的任务不适合用这种模式。
3. 搜索引擎模式：它在精确模式的基础上，对长词再次进行切分，以提高召回率。这种模式的分词结果既保证了准确性，又能够覆盖更多的词语，非常适合用于搜索引擎的

查询分词。虽然搜索引擎模式能够提高召回率，但也可能产生一些噪声词，所以也不适用于本次任务。

## 分词pipeline

而对于分词这个SubTask的pipeline，比较直观的有两种：

1. 先过滤标点符号，再分词
2. 先分词，再过滤标点符号

考虑到中文文本中的标点符号可以有效地凸显语义以及语句之间的分隔，因此分词工具能够根据句子的整体结构和语义进行分词，而不仅仅是基于标点。这样可能会得到更准确的分词结果。

所以最终我选择先分词，再过滤标点。

## 具体实现

在具体实现上，我采用正则表达式匹配所有标点符号，并替换为空字符串：

```
def remove_punctuation(text):
    # 定义一个正则表达式模式，匹配所有标点符号
    punctuation_pattern = r'[^\w\s]'
    # 使用re.sub替换所有匹配到的标点符号为空字符串
    no_punctuation_text = re.sub(punctuation_pattern, '', text)
    return no_punctuation_text
```

随后读取原始文本，利用jieba的cut函数，进行精确模式的分词：

```
    # 读取文件内容
    with open(original_path, "r", encoding="utf-8") as doc:
        content = doc.read()

        # 对内容进行分词
    seg_list = jieba.cut(content, cut_all=False)
    seg_words = list(seg_list)  # 将生成器转换为列表
```

随后过滤标点，将结果写入到指定文件中：

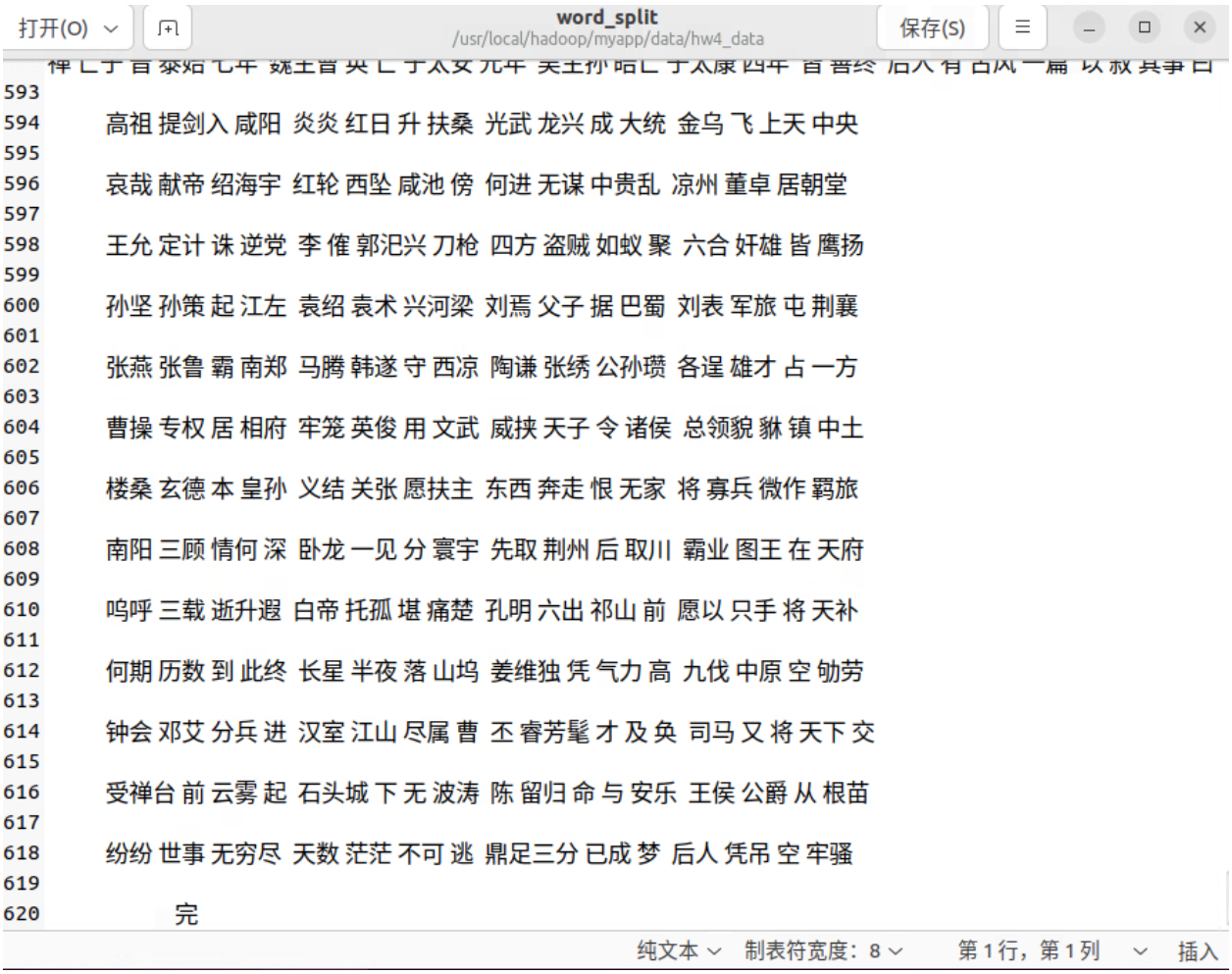
```
    # 输出分词结果到目标文件
    with open(target_path, "w", encoding='utf-8') as target:
        target.write(res)
```

而这次的文本是三国演义，其中包括了一些古文。因此我们要对一些古汉语常用词进行词典定义。如：

```
玄德 10000 n
事势 100 n
汝 100 n
蔡和 1000 n
受禅台 10000 n
邓艾 10000 n
钟会 10000 n
皇孙 1000 n
张鲁 10000 n
韩遂 10000 n
陶谦 10000 n
马腾 10000 n
公孙瓒 10000 n
袁术 10000 n
袁绍 10000 n
南郑 10000 n
```

这样的人名或短词等，前面的词语，后面是词频，词频越高提取该词的可能越大，n是词性，名词。

分词结果如下：




至此，分词子任务完成。

## 另一条道路：AI分词

我们可以调用chatGLM的API，让AI来帮助我们分词，不过这个总感觉不太好，有点作弊。但是对于现代文的分词的效果确实不错：

```
client = ZhipuAI(api_key=api_key)
messages = [{"role": "user",
             "content": f"你是一名中文的专家。下面给你一
response = client.chat.completions.create(
    model="glm-4",
    messages=messages,
)
```

但chatGLM对于古文的理解能力很有限，我试了试发现分词效果很差，最终没有选用这种方式。

## 3.2. MapReduce框架统计词频

### 项目结构

1. Hadoop版本：3.1.3
2. 项目依赖：首先确保你下载了hadoop3.1.3。项目依赖以jar包的形式存在，包括：
    a. hadoop/share/hadoop/common下的所有jar包
    b. hadoop/share/hadoop/common/lib下的所有jar包
    c. hadoop/share/hadoop/mapreduce下的所有jar包
    d. hadoop/share/hadoop/mapreduce/lib下的所有jar包

这里备注一点，厦大林子雨老师给的依赖可能有点不全，我按照他的配有些依赖库找不到，所以自己摸索了一个项目依赖配置，如果你根据他的配不出来，可以参考我的。

### 具体实现

MapReduce是一个框架，他接收用户传入的Map函数、Reduce函数（可能还有Combiner函数、Partition函数等）。

在作为开发者应用这个框架时，将代码结构拆分为以下3部分：

1. 配置模块
2. Map函数
3. Reduce函数

其中配置模块给MapReduce框架传入用户编写的Map函数和Reduce函数，随后框架热加载用户的程序并运行，得到结果返回给用户。

在配置模块中，传递了这些参数：

1. Map函数所在的类（Java没有函数指针的概念，而是采用反射机制将类定义作为函数指针的替代品，JVM在类路径中寻找指定的类定义，然后解析其中的Map函数/Reduce函数。笨重的java。）
2. Reduce函数所在的类

3. 文本的输入输出路径

其中，前两个写死成我们自己写的类就行了。第三个在命令行给出：

```
public class WordCount {
    public static void main(String[] args) throws IOException, I
        Configuration conf = new Configuration();
        String[] otherArgs = (new GenericOptionsParser(conf, arg
        if(otherArgs.length < 2) {
            System.err.println("Usage: wordCount <in> [<in>...]
            System.exit(2);
        }
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        for(int i = 0; i < otherArgs.length - 1; ++i) {
            FileInputFormat.addInputPath(job, new Path(otherArgs
        }
        FileOutputFormat.setOutputPath(job, new Path(otherArgs[o
        System.exit(job.waitForCompletion(true)?0:1);
    }
}
```

那么这里的：

```
  job.setCombinerClass(IntSumReducer.class);
```

是什么呢？在Google有关MR的论文中提到：

## 4.3 Combiner Function

In some cases, there is significant repetition in the intermediate keys produced by each map task, and the user-specified *Reduce* function is commutative and associative. A good example of this is the word counting example in Section 2.1. Since word frequencies tend to follow a Zipf distribution, each map task will produce hundreds or thousands of records of the form `<the, 1>`. All of these counts will be sent over the network to a single reduce task and then added together by the *Reduce* function to produce one number. We allow the user to specify an optional *Combiner* function that does partial merging of this data before it is sent over the network.

The *Combiner* function is executed on each machine that performs a map task. Typically the same code is used to implement both the combiner and the reduce functions. The only difference between a reduce function and a combiner function is how the MapReduce library handles the output of the function. The output of a reduce function is written to the final output file. The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

Partial combining significantly speeds up certain classes of MapReduce operations. Appendix A contains an example that uses a combiner.

在某些情况下，每个map任务产生的中间键都会有很大的重复。用户指定的Reduce函数是可交换的和关联的。一个很好的例子是第2.1节中的词频统计示例。由于词频往往遵循齐普夫分布，每个map任务将会产生数百或数千条形如<the, 1>的记录。所有这些计数将通过网络发送到单个reduce任务，然后通过Reduce函数加在一起产生一个数字。我们允许用户指定一个可选的Combiner函数，在数据发送到网络之前对其进行部分合并。

换句话说，Combiner就是Reducer，只是为了减少网络的负担，所以在传输数据前，**先对词频进行一次部分合并**，例如中间文件中的数据为：

```
a 1
a 1
a 1
a 1
a 1
```

那么combiner会将其合并为：

```
a 5
```

然后再发送，这样文件的大小就会变小，网络负担也会变少。

而对于Map函数，则是遍历当前文本的split，将其中的每个单词取出，并写入到中间文件中：

```
    public void map(Object key, Text value, Mapper<Object, Text,
        StringTokenizer itr = new StringTokenizer(value.toStrin
        while(itr.hasMoreTokens()) {
            this.word.set(itr.nextToken());
            context.write(this.word, one);
        }
    }
```

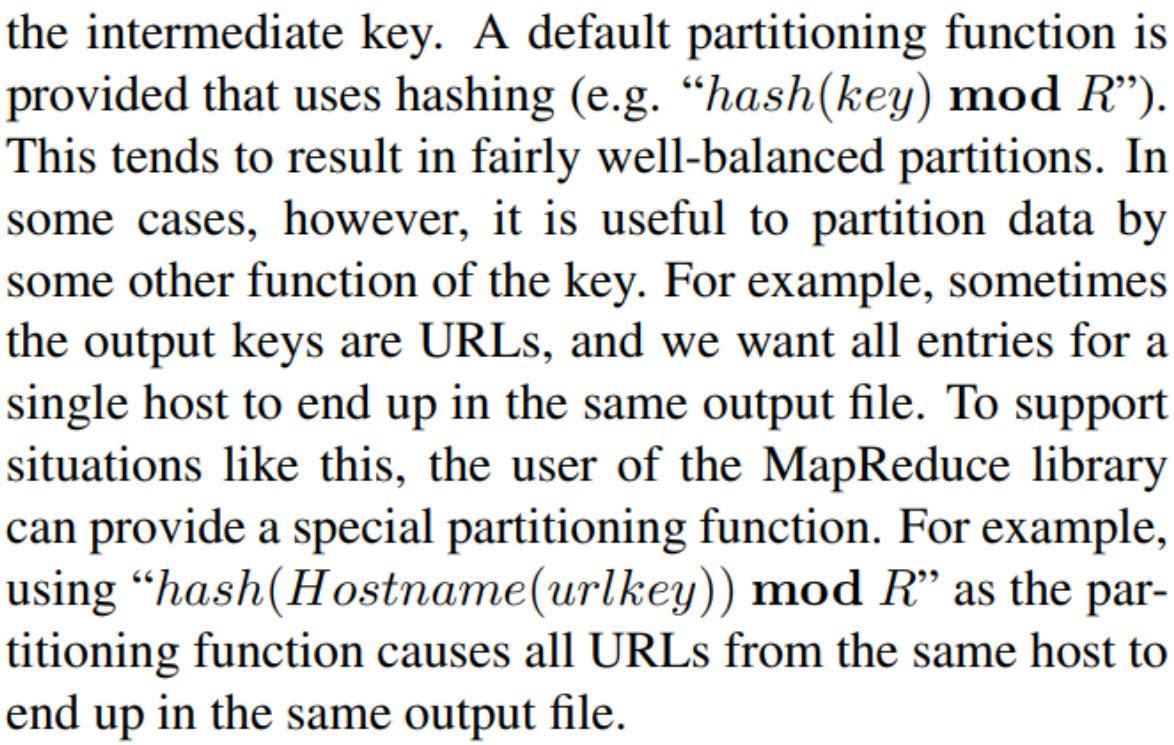
而对于Reduce函数，则是将每个键具有的词频求和加起来，写到临时结果文件。

```
    public void reduce(Text key, Iterable<IntWritable> values, R
        int sum = 0;
        IntWritable val;
        for(Iterator<IntWritable> i$ = values.iterator(); i$.has
            val = i$.next();
        }
        this.result.set(sum);
```

```
            context.write(key, this.result);
        }
```

可能会疑惑，为什么敢对所有的key统一求和，不怕key不一样吗？对于这个问题，有两种认知基础：

1. 不知道Partitioner的存在：MapReduce框架有一个Partition函数，在Map函数向中间文件写入数据时，会保证相同的key写到同一个文件中。这一点在Google有关mr的论文中提到了：



the intermediate key. A default partitioning function is provided that uses hashing (e.g. "$hash(key)$ **mod** $R$"). This tends to result in fairly well-balanced partitions. In some cases, however, it is useful to partition data by some other function of the key. For example, sometimes the output keys are URLs, and we want all entries for a single host to end up in the same output file. To support situations like this, the user of the MapReduce library can provide a special partitioning function. For example, using "$hash(Hostname(urlkey))$ **mod** $R$" as the partitioning function causes all URLs from the same host to end up in the same output file.

2. **知道Partitioner的存在**：这也是我自己好奇的一个问题，因为Google提出的设想是使用Hash来进行单词与文件的映射的，众所周知利用取模计算Hash值会导致**哈希冲突**，也就是多个key写入到了同一个文件。例如中间文件1中有：

```
a a a a
d d d
g g
```

假设我们总共有三个中间文件，那么a,d,g取余3之后的值都是1，那么a、d、g都将写入中间文件1中。现在我们再简单地进行累加就不对了，就变成了：

```
<a,4>
<d,3>
<g,2>
```

不过在运行代码时，我没有遇到这种情况，对此我有一个猜测：那就是**Hadoop MR框架的中间文件数量充分多**，多到不会发生哈希冲突。假设我们计算出来100个哈希值，但是MR框架给了100个中间文件，这样就不会有哈希冲突了。

总之，Partitioner确保了Reducer在任务进行时，输入KV对的key全部一致。因此这个问题就不用担心了。

## 3.3. TOP-K词频求解

### 选择什么算法？

由于这个任务比较简单，我就用shell脚本做了。

一般对于TOP-K问题，有很多求解方法：

1. 排序，时间复杂度O(nlgn)
2. 大小根堆维护长度为K的窗口：时间复杂度O(nlgK)
3. 桶排序：时间复杂度O(n)，空间复杂度O(max_ele)

一般来说**综合能力上大小根堆最优**，桶排序别看时间比较牛，但一旦有一个超大无比的树，就得有对应那么多的桶，对于硬件一般的机器根本负担不起；排序就是个比较trivial的算法。

按理说我这里应该选择大小根堆的，不过这还要看**具体需求**，如果n不是1e10以上的数据，用排序跑nlgn的复杂度，一般来说2s内都可以解决掉的，所以没必要省这点复杂度了。

### 具体实现

这里我用Linux的Shell脚本做了，其实写起来很简单：

```
#!/bin/bash

# 检查参数数量
if [ "$#" -ne 3 ]; then
    echo "Usage: $0 <input_file_path> <k> <output_file_path>"
    exit 1
fi

input_file="$1"
k="$2"
output_file="$3"

# 使用awk读取文件，使用sort排序，使用head获取前k个，最后输出到文件
awk '{print $2, $1}' "$input_file" | sort -nrk 1 | head -n "$k"

echo "Top $k <单词,词频> 键值对已输出到 $output_file"
```

主要是awk和sort命令的使用。

## 3.4. 运行注意事项

1. 确保你将分词后的文本文件传到了hdfs里：

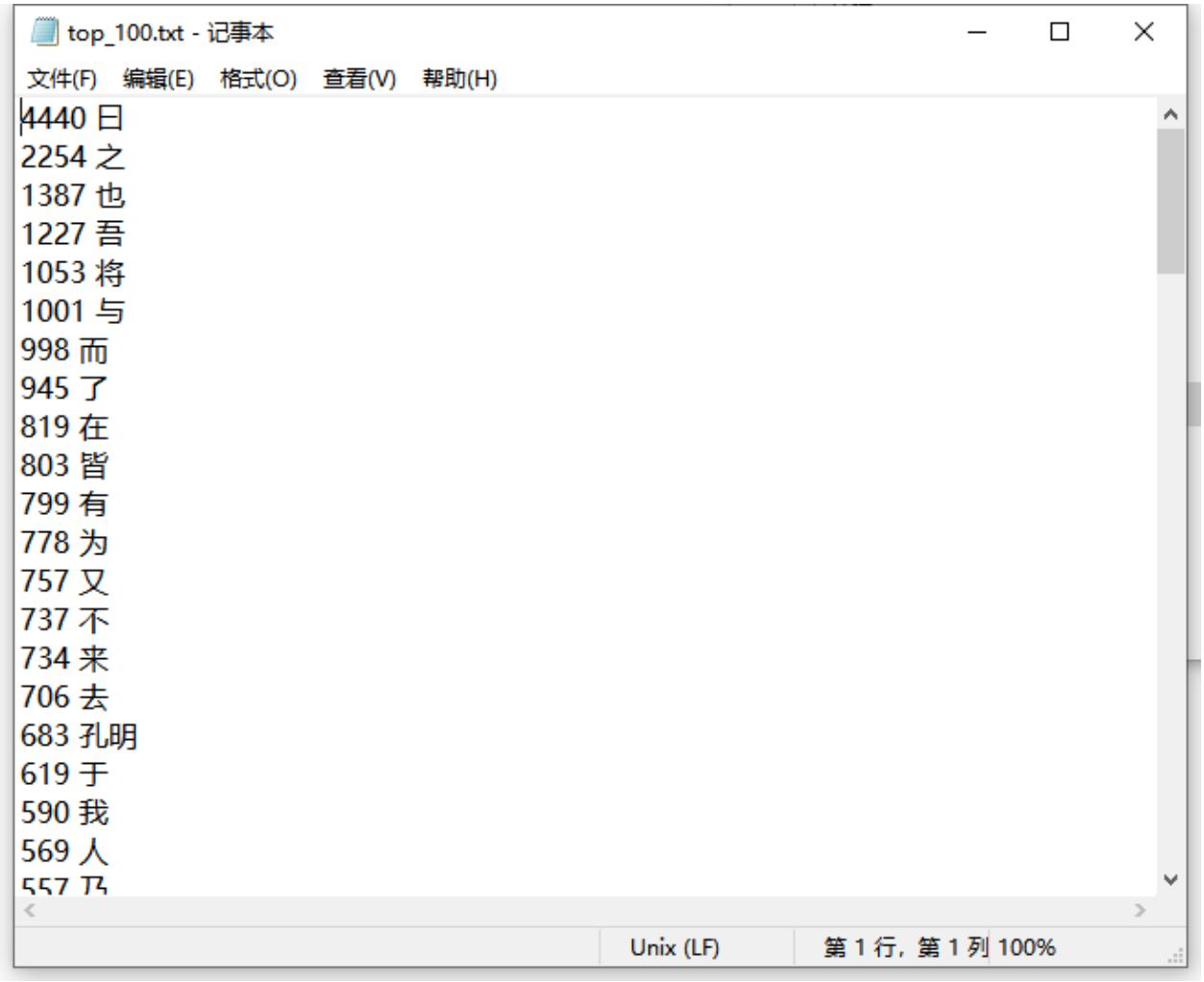
```
hdfs dfs -put "你的文件路径" "hdfs上的文件路径"
```

2. 确保你的hdfs中，没有和你命令行里传入参数相同的输出路径。例如你要求把输出写到hdfs里的output文件夹下，但你已经有个这个文件夹了，他再创建就重名报错。所以每次都要把这个output文件夹删一下，我建议把这组命令写到shell脚本

```
hdfs dfs -rm -r /BigDataHomework/output
hadoop jar ../hw4.jar /BigDataHomework/input /BigDataHomework/ou
```

# 4. 运行结果

如图下图所示：


top_100.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
4440 日
2254 之
1387 也
1227 吾
1053 将
1001 与
998 而
945 了
819 在
803 皆
799 有
778 为
757 又
737 不
734 来
706 去
683 孔明
619 于
590 我
569 人
Unix (LF)
第 1 行，第 1 列 100%