Trigram Language Model

陈俊含 19307130180 2021.12.21

一、任务要求

1. 使用服务器上的wiki语料（/corpus/wiki）构建一个三元语言模型
2. 对于任意给出的w1 w2 w3，计算 P(w3 | w1, w2)
3. 加入 add-K smoothing
4. 支持对后续词的预测：输入前面的词，程序输出对后续词的预测

二、工程文件说明

1. Probability_Calculator（mapreduce部分）
    - main/
        - TLM.java：程序的执行入口，先执行WordCount获取单词种类数用于smoothing，再执行概率计算
        - TLMMap.java：map部分，找出所有的三元组并记录
        - TLMCombine.java：将map的输出值进行整理合并
        - TLMReduce.java：进一步整合数据，最后将value中的数量转化为概率
        - MyType.java：自定义reduce部分的输出格式（json格式），方便后端处理数据
    - WordCount/
        - Wordcount.java：计数程序的执行入口
        - WordcountMap.java：map部分，拆分成单个单词并计数
        - WordcountReduce.java：只取key部分，value设为1

2. Prediction（前后端部分，flask+vue）
    - dist/
        - static /：渲染html的css和js文件
        - index.html：前端页面，由frontend生成，可直接打开使用
    - frontend/src (只列出主要的文件夹)
        - assets/：静态文件所在地，如图片
        - components/
            - predict.vue：用于实现auto-complete框以及前后端的交互
        - router/
            - index.js：路由设计，将具体路由地址与components中的部件联系起来
        - App.vue：前端页面主体，包含logo与输入框
        - main.js：用于添加部件依赖，如auto-complete框与渲染文件

- output/：处理后的语料库
- app.py：后端文件，用于读取语料库与响应前端的需求

3. out/（mapreduce文件build出来的jar包，用于上传hadoop机群运行）

三、具体实现方式

1. Probability Calculator
   - 预先执行WordCount程序，查看输出文件的行数，即为语料库的单词种数，修改`TLM.java`中的`Size`变量为该值
   - 提交job后，hadoop机群首先会将目录下的所有语料文件切片，分配给不同的机器处理；对于每一个切片，由于使用了标准输入`TextInputFormat.class`格式，会自动按行生成键值对。key值为文本的偏移量，value为该行的内容；此key-value可以作为map程序的输入。

   - map阶段
     - 描述：将w1w2w3处理为w1w2 -> {w3: 1}
     -

	Key	Value
input	偏移量	一行语料
output	两个单词组成的词组（w1, w2）	词组后一个单词出现的次数 {w3: num, w4: num, ...}

     - 对于每一行语料，使用split函数将其转化为字符串数组，方便后续的操作
     - 遍历数组，取下标为`i`和`i+1`的词组成新的key值，即`s.append(str[i]).append(" ").append(str[i+1])`
     - 对于每个key值和其后继单词`str[i+2]`，先检测hashmap中有无key值的记录，有的话则更新value中的map值，没有的话则创建key值条目
     - 最后，输出hashmap中的所有内容：key值为单词长度为2前缀，value值为所有后继词及其出现次数

   - combine阶段
     - 对于所有的输入，合并具有相同key的value部分，加快运行效率
     - 如将 w1w2 -> {w3: 3, w4: 4} 和 w1w2 -> {w3: 1, w4: 2} 合并成 w1w2 -> {w3: 4, w4: 6}

   - reduce阶段
     -

	Key	Value
input	两个单词组成的词组（w1, w2）	map阶段输出value的一个list
output	两个单词组成的词组（w1, w2）	词组后一个单词出现的概率 {w3: p1, w4: p2, ...}

     - 前期工作与combine一致

- 对每一个key，需要额外统计其value中所有的单词频次的总和`sum`,用于后续的概率计算（用一个hashmap存起来）
- 最后，通过概率公式将value中所有的频次替换成概率，并输出
- add-K smoothing的概率公式：

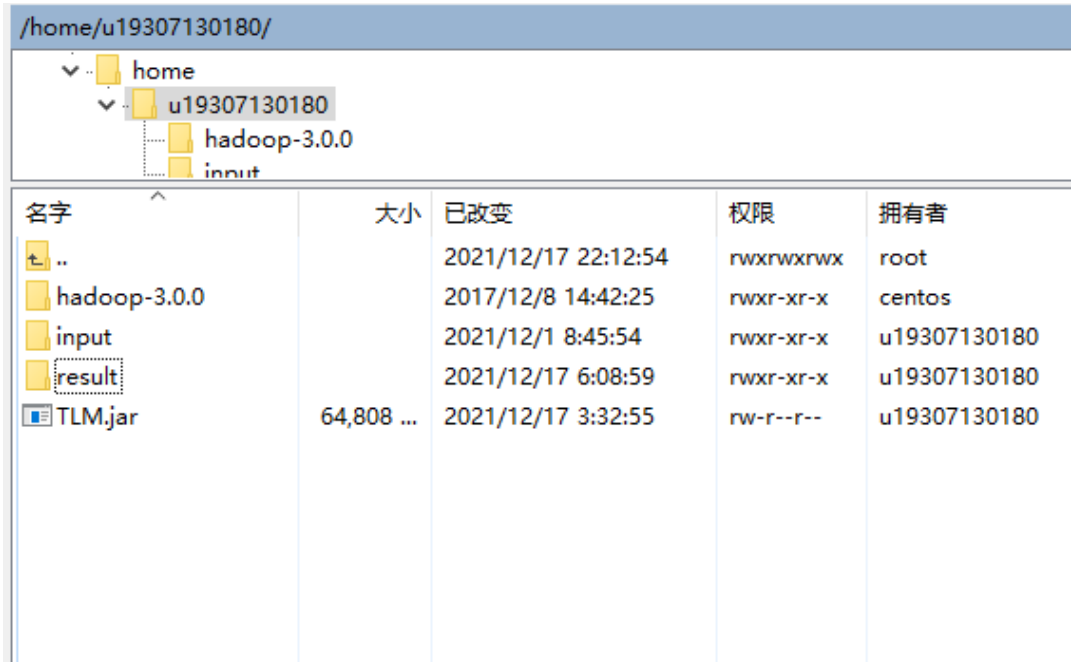
$$P(w3|w1,w2)=\frac{N(w1w2w3)+k}{N(w1w2)+k|V|}$$

N(w1w2w3)即value中的频次，N(w1w2)即该key对应的sum值，V即为wordcount中计算出来的单词种数，为方便取k=1

2. Prediction
   - 前端
     - 主体部分采用一个vue框架中的auto-complete输入框，实现数据的自动传输与获取
     - `handleSearch`：读取输入框中的内容，使用split函数将其处理成一个列表，若列表长度小于2则直接返回，否则调用work函数获取预测词
     - `work`：取列表中的最后两个单词，调用fetch方法将该前缀通过url传输给后端，然后将后端返回的数据经过适当处理后赋值给`dataSource`（预测词显示框）
   - 后端
     - 采用flask框架
     - 构造TLM类
       - 构建一个用于读取语料文件的接口
       - 给每个key值生成词频排序列表，记录在一个字典变量里
       - 实现一个可以从字典变量获取数据的接口
     - 设置跨域许可
     - 开始通过接口读取语料
     - 设置路由处理函数
       - 提交方式为`POST`
       - 接受前端发来的请求，提取其中携带的数据
       - 调用TLM模型接口搜索相应的预测单词并返回

四、运行方式

1. 训练Trigram Language Model模型
   - 将`out/artifacts/TLM/TLM.jar`通过WinSCP上传到自己的Hadoop主目录



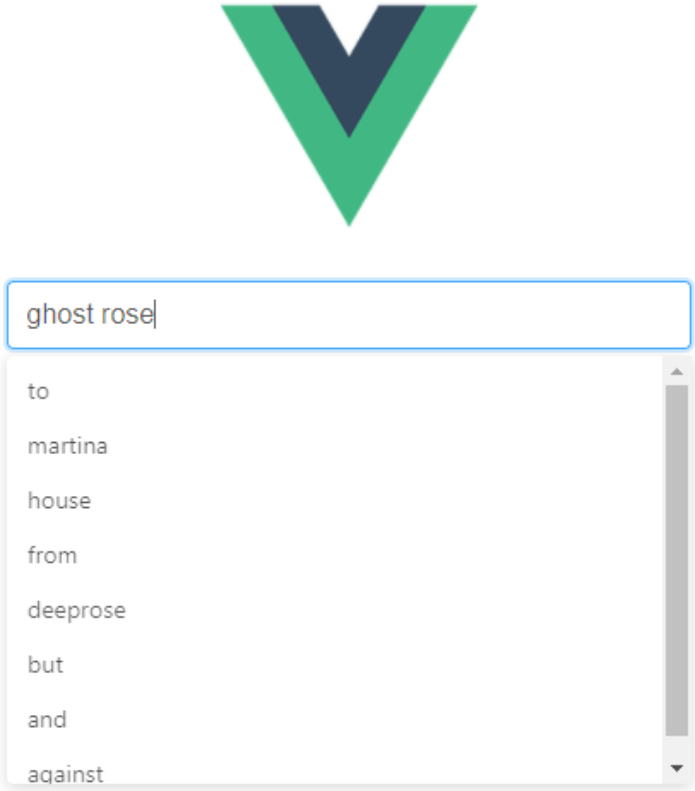
- 通过`ssh`命令连接Hadoop机群
- 执行指令`hadoop jar TLM.jar /corpus/wiki/ ./result`
- 耐心等待集群完成job

Trigram Language Model	MAPREDUCE	default	0	Fri Dec 17 16:39:06 +0800 2021	Fri Dec 17 17:31:14 +0800 2021	FINISHED	SUCCEEDED

- 将Hadoop中`./result`里的文件通过WinSCP传输到本地目录`Prediction/output/`下
- 输出文件示例：

```
officialise his {"transfer": 1.9320292779716784E-7, "invitation": 1.9320292779716784E-7, "relationship":
officially allowed  {"capacity": 1.9319876588492327E-7, "online": 1.9319876588492327E-7, "its": 3.8639753
officially austin   {"has": 1.9320296512454637E-7}
officially cancel   {"their": 6.762095287389533E-7, "the": 7.728108899873751E-7, "josé": 1.93202722496843
officially carnaval {"ponceño": 2.8980439169575176E-7, "de": 1.9320292779716784E-7}
officially declared {"good": 1.93150422458604E-7, "with": 6.76026478605114E-7, "desegregated": 1.9315042
officially documented   {"all": 1.9320081882371033E-7, "with": 1.9320081882371033E-7, "no": 1.9320081882
officially enacted  {"on": 9.66010999580848E-7, "until": 1.9320219991616957E-7, "in": 8.694098996227631E-
officially encartar {"the": 1.9320296512454637E-7}
officially evicted  {"on": 2.898037757954341E-7, "between": 1.9320251719695605E-7, "as": 1.9320251719695
officially fadhéla  {"madani": 1.9320296512454637E-7}
officially fairgrounds  {"nashville": 1.9320296512454637E-7}
officially fanmade  {"for": 1.9320296512454637E-7}
officially filed    {"claim": 1.9319930711000497E-7, "with": 6.761975748850175E-7, "its": 2.897989606650
officially flung    {"open": 1.9320296512454637E-7}
```

2. 关联词预测
   - 在cmd中进入`./Prediction`
   - 运行指令`flask run`以启动后端
   - 打开`Prediction/dist/index.html`即可使用
   - 效果示意图：



五、总结

1. 基本掌握了mapreduce的代码框架以及Hadoop机群的操控方式，学习了vue和flask框架的使用。
2. 善用debuger：前后端交互这一块花费了很多时间，刚开始不懂得使用cmd和console进行调试，所以浪费了很多不必要的时间。
3. Reduce的task问题：最初没有设置reducetask数量，默认值为1，导致运算速度非常慢，而且输出文件太大很容易中途出现错误；后来将task数设置成了256，不到一小时就完成了job，但这也导致了一个很严重的问题——输出文件分成了256个文件，将它们下载到本地和用TLM模型读取时都带来了不小的麻烦。
4. vue与flask的交互
    - 分别在不同的端口监听，vue采用5000号端口，flask采用8088号端口
    - flask必须一直启动，确保其不停运作
    - vue有两种选择
        - `npm run dev`：可以实时响应前端代码的改变，适合调试时使用
        - `nom run build`：将前端打包生成html文件，可直接打开使用
    - 由于前后端分离，必须支持跨域
        - flask：`CORS(app, *support_credentials*=True)`
        - vue：`mode: 'cors'`
5. java的split函数：python系的语法是`split(' ')`即可自动处理所有的空格并分割，但java语法需要写成正则表达式`*String* str[] = line.split("\\s+")`，否则只会分割第一个空格。

六、附录（主要代码文件）

1. TLMMap.java

```
public class TLMMap extends Mapper<LongWritable, Text, Text, MyType> {

    public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
        String line = value.toString();
        String str[] = line.split("\\s+");  //注意不能简单用“ ”，不然只会分离第一
个空格
        HashMap<Text, MyType> map = new HashMap<Text, MyType>();

        for(int i=0; i<str.length-2; ++i) {
            StringBuilder s = new StringBuilder();
            s.append(str[i]).append(" ").append(str[i+1]);
            Text ab = new Text(s.toString());
            Text c = new Text(str[i+2]);

            if(!map.containsKey(ab)) {
                map.put(ab, new MyType());
            }

            int val = (map.get(ab).containsKey(c) ? ((IntWritable)
map.get(ab).get(c)).get(): 0);
            map.get(ab).put(c, new IntWritable(val + 1));
        }

        for(Entry<Text, MyType> entry : map.entrySet()) {
            context.write(entry.getKey(), entry.getValue());
        }
    }
}
```

2. TLMReduce.java

```
public class TLMReduce extends Reducer<Text, MyType, Text, MyType> {

    static long num;
    public void reduce(Text key, Iterable<MyType> values, Context context)
throws IOException, InterruptedException{
        num = context.getConfiguration().getLong("num", 0);
        MyType map = new MyType();
        MapWritable sums = new MapWritable();

        for(MyType value : values){
            for(Entry<Writable, Writable> entry : value.entrySet()) {
                Text k = (Text) entry.getKey();
                double val = (map.containsKey(k) ? ((DoubleWritable)
map.get(k)).get(): 0);
                map.put(k, new DoubleWritable(val + ((IntWritable)
entry.getValue()).get()));
                int v = (sums.containsKey(k) ?
((IntWritable)sums.get(k)).get() : 0);
                sums.put(k, new IntWritable(v + ((IntWritable)
entry.getValue()).get()));
```

```
            }
        }

        map.replaceAll((k, v) -> new DoubleWritable((((DoubleWritable)
v).get() + 1) / (((IntWritable)sums.get(k)).get() + num)));

        context.write(key, map);
    }
}
```

3. app.py

```
from flask import Flask, request
import json
from pathlib import Path
from typing import List
from flask_cors import CORS

class TLM(object):
    def __init__(self) -> None:
        super().__init__()
        self.__dict = {}

    def _read(self, filepath: Path) -> None:
        with open(filepath, 'r', encoding='utf-8') as file:
            for line in file:
                line_ = line.strip('\n').split('\t', 2)
                key = line_[0]
                dict_ = json.loads(line_[1])
                sorted_list = [key for key, value in sorted(dict_.items(),
key = lambda kv:(kv[1], kv[0]), reverse=True)]
                self.__dict[key] = sorted_list

    def getList(self, words: str) -> List[str]:
        return self.__dict.get(words, [])

app = Flask(__name__)
CORS(app, support_credentials=True)
if __name__ == '__main__':
    app.run()

tlm = TLM()

for i in range(0, 10):
    p = "F:/output/part-r-0000" + str(i)
    tlm._read(Path(p))
for i in range(10, 50):
    p = "F:/output/part-r-000" + str(i)
    tlm._read(Path(p))
for i in range(100, 256):
    p = "F:/output/part-r-00" + str(i)
    tlm._read(Path(p))

@app.route('/predict', methods=['POST'])
def predict():
    data = json.loads(request.data)
    res_list = tlm.getList(data['prefix'])
```

```
    reply = {'data': res_list}
    return json.dumps(reply, ensure_ascii=False)
```

4. predict.vue

```
<template>
  <a-auto-complete
    :dataSource="dataSource"
    style="width: 400px"
    size="large"
    @search="handleSearch"
    placeholder="please input..."
  />
</template>

<script>
let list

function work (target) {
  const prefix = list[list.length - 1] === '' ? list[list.length - 3] + ' '
+ list[list.length - 2] : list[list.length - 2] + ' ' + list[list.length -
1]
  fetch('http://localhost:5000/predict', {
    method: 'POST',
    mode: 'cors',
    body: JSON.stringify({
      prefix: prefix
    })
  }).then(function (data) {
    return data.json()
  }).then(function (d) {
    const result = d.data
    const data = []
    result.forEach(row => { data.push(row) })
    target(data)
  })
}

export default {
  data () {
    return {
      dataSource: []
    }
  },

  methods: {
    handleSearch (value) {
      this.dataSource = []
      list = value.split(' ')
      if (list.length < 2) return
      work(data => (this.dataSource = data))
    }
  }
}
</script>
```