

Wikipedia with LSA

——基于 Wikipedia 的搜索前端搭建

张可 16307100128
冯孟笛 16307100076
陈天予 16307130005
张言健 16300200020

1. 项目背景:

LSA: 潜在语义分析 (LSA) 是一种自然语言处理和信息检索技术, 旨在更好地理解文档语料库和这些文档中单词之间的关系。它试图将语料库提炼成一组相关概念。每个概念都捕获数据中的变化线索, 并且通常对应于语料库讨论的主题。

我们选取了 Wikipedia 于 2019-05-01 发布的数据, 将其中的 19, 197, 820 篇词条页面文章清洗出来, 对文章标号, 进行 LSA(潜在语义分析)检索。

数据来源: <https://dumps.wikimedia.org/enwiki/20190501/>

Index of /enwiki/

../	
20190320/	02-May-2019 01:28
20190401/	21-May-2019 01:34
20190428/	02-Jun-2019 01:27
20190501/	21-Jun-2019 01:33
20190520/	24-May-2019 02:26
20190601/	09-Jun-2019 19:22
20190620/	24-Jun-2019 06:26
latest/	24-Jun-2019 06:25

图 1: 下载目录

```
'<title>Anarchism</title>',
'<ns>0</ns>',
'<id>12</id>',
'<revision>',
'<id>894931025</id>',
'<parentid>894931010</parentid>',
'<timestamp>2019-04-30T22:23:44Z</timestamp>',
'<contributor>',
'<username>ClueBot NG</username>',
'<id>13286072</id>',
'</contributor>',
'<minor />',
'<comment>Reverting possible vandalism by [[Special:Contribs/Kill
also Positive?]] Thanks, [[WP:CBNG/ClueBot NG]]. (3622126) (Bot)</comment>
```

图 2: wiki 数据 (Anarchism 词条)

数据解压后的大小约为 72G。考虑到数据量较大, 我们将数据上传到 HDFS 上, 并搭建了由 3 台电脑组成的 Spark 集群对数据进行处理。

- 集群搭建于大数据本科实验室:新金博 1309 机房
访问方式: 连接校园网后 ssh hadoop@IP
IP 地址: 10.192.7.116/10.192.7.33/10.192.7.161

2. 实验环境：Spark 集群搭建

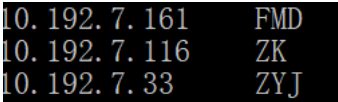
2.1 Hadoop 完全分布式配置

#Hadoop 目录: /usr/local/Hadoop/
#version:2.8.5

Step1:为三台机器添加 hadoop 用户，并对 hadoop 授予权限：sudo
chown -R hadoop ./hadoop

Step2: 配置三台机器互相免密登陆

#我们以三位小组成员：冯孟笛，张可，张言健的名字缩写命名了三个节点



10.192.7.161	FMD
10.192.7.116	ZK
10.192.7.33	ZYJ

图 3: vim /etc/hosts

互相免密登陆配置流程

```
cp id_rsa.pub authorized_keys#生成公钥密钥对
cp id_rsa.pub authorized_keys#在主节点执行
#分别在两个子节点上执行：
scp /root/.ssh/id_rsa.pub root@Mage1:/root/.ssh/id_rsa_Mage2.pub
scp /root/.ssh/id_rsa.pub root@Mage1:/root/.ssh/id_rsa_Mage3.pub
然后在主节点上，将拷贝过来的两个公钥合并到 authorized_keys 文件
#主节点上执行：
cat id_rsa_Mage2.pub>> authorized_keys
cat id_rsa_Mage3.pub>> authorized_keys
#最后用 scp 将主节点的 authorized_keys 替换子节点 authorized_keys
```

Step3:在三台机器的 hadoop 目录下配置 hadoop,并配置 master 与 slave

需要修改的配置文件有：

hadoop-env.sh	core-site.xml	hdfs-site.xml
yarn-env.sh	core-site.xml	mapred-site.xml
yarn-site.xml	slaves	

```

hadoop@bigDataLab32: $ /usr/local/hadoop/sbin/start-all.sh
This script is deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [FMD]
FMD: starting namenode, logging to /usr/local/hadoop/logs/hadoop-hadoop-namenode-bigDataLab32.out
ZK: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hadoop-datanode-bigDataLab31.out
ZYJ: starting datanode, logging to /usr/local/hadoop/logs/hadoop-hadoop-datanode-bigDataLab06.out
Starting secondary namenodes [0.0.0.0]
0.0.0.0: starting secondarynamenode, logging to /usr/local/hadoop/logs/hadoop-hadoop-secondarynamenode-bigDataLab32.out
starting yarn daemons
starting resourcemanager, logging to /usr/local/hadoop/logs/yarn-hadoop-resourcemanager-bigDataLab32.out
ZYJ: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hadoop-nodemanager-bigDataLab06.out
ZK: starting nodemanager, logging to /usr/local/hadoop/logs/yarn-hadoop-nodemanager-bigDataLab31.out

```

图 4：配置成功后在 master（FMD）中 start-all 的运行效果

2.2 Spark 完全分布式配置

#Spark 目录： `/usr/local/spark/`

需要修改的配置文件有：

Spark-env.sh slaves.template

在三台机器的 Spark 目录下配置 spark，配置成功后 slave 上使用 jps 查看

进程会多出 Worker 节点

```

Last login: Wed Jun 20 20:10:10 2019
hadoop@bigDataLab32: $ jps
50722 NameNode
51159 ResourceManager
52119 Worker
50967 SecondaryNameNode
69801 Jps
51947 Master
2428 DataNode
53916 CoarseGrainedExecutorBackend

```

图 3-1：10.192.7.161

```

hadoop@bigDataLab06: $ jps
26912 SparkSubmit
113779 Jps
60882 Worker
75506 CoarseGrainedExecutorBackend
71492 NodeManager
97549 SparkSubmit
71342 DataNode

```

图 3-2：10.192.7.33

```

hadoop@bigDataLab31: $ jps
39172 SparkSubmit
49461 CoarseGrainedExecutorBackend
38455 DataNode
72584 Jps
71544 SparkSubmit
44202 SparkSubmit
19131 SparkSubmit
38686 NodeManager
77406 Worker

```

图 3-3：10.192.7.116

Worker
worker-20190623185545-10.192.7.116-38491
worker-20190623185545-10.192.7.33-45999
worker-20190623185545-10.192.7.161-42149

图 3-4：worker

图 5：Spark 集群运行状态

2.3 Pyspark Jupyter-notebook 配置

#anaconda 目录： `/home/hadoop/anaconda3`

Step1：在三台机器的 anaconda 目录下安装 anaconda，并配置 Pyspark

用 Jupyter-notebook 打开

修改 spark-env.sh 文件，添加如下内容：

```
export PYSPARK_PYTHON="/home/hadoop/anaconda3/bin/python3.7"
export PYSPARK_DRIVER_PYTHON="/home/hadoop/anaconda3/bin/ipython3"
export PYSPARK_DRIVER_PYTHON_OPTS="notebook"
alias pysbook="$SPARK_PATH/bin/pyspark"
```

Step2: 配置远程 Jupyter notebook 访问

方法一：使用 Xshell 配置监听端口实现 jupyter notebook 远程访问

方法二：生成 jupyter notebook 配置文件并配置为远程密码访问

在 10.192.7.116 机器上配置了 jupyter notebook 的远程访问，密码为 12345678 ,也可使用 xshell 远程访问 jupyter

2.4 Spark 集群使用指南

Step1: 远程登陆 IP: 10.192.7.116/10.192.7.161/10.192.7.33

需要在校园网环境下登陆

```
$ ssh Hadoop@10.192.7.116
```

Step2: 在 master 节点上启动集群

```
$ ssh FMD
$/usr/local/hadoop/sbin/start-all.sh
$/usr/local/spark/sbin/start-all.sh
$exit
```

Step3: 在 ZK worker 节点上通过 jupyter notebook 运行 pyspark

```
$/usr/local/spark/bin/pyspark
```

复制弹出的链接，将地址名 localhost 改为 10.192.7.116

提示后输入密码 12345678

Step4: 在 Jupyter notebook 中设置 spark cluster

```
$SparkConf.setMaster("spark://10.192.7.161:7077")
```

设置 master 节点 10.192.7.161:7077

根据需要设置 executor 和 driver 上的内存，以及处理器核的数目

SparkContext

Spark UI

Version

v2.4.3

Master

spark://10.192.7.161:7077

AppName

Bigdata_Spark_Cluster

```
from pyspark import SparkConf, SparkContext
conf=SparkConf()
    .setMaster('spark://10.192.7.161:7077')
    .set("spark.local.ip", "10.192.7.116")
    .set("spark.driver.host", "10.192.7.116")
    .setAppName('Bigdata_Spark_Cluster')
    .set('spark.executor.memory', '150g')
    .set('spark.driver.memory', '150g')
    .set("spark.executor.cores", '2')
    .set('spark.cores.max', 40)
    .set('spark.logConf', True)
    .set("spark.driver.maxResultSize", "150g")
sc.stop()
conf.get("spark.master")
```

图 4-1 集群运行结果

图 4-2 Jupyter Pyspark 集群设置

图 6：Pyspark SC 设置

3. 数据预处理

3.1 ID 标注

读取 HDFS 上的 enwiki.xml

```
log=sc.textFile("hdfs://10.192.7.161:9000/enwiki.xml")
```

定义函数 id,按<title></title>区分词条, 如果行中含有<title></title>

则 map 为列表[1,行文本, 标题], 否则 map 为列表[0, None]

目标: 标识词条分隔处, 提取标题, 方便后续对属于同一词条的行进行合并

```
Def id(line):
    try:
        title=re.findall("<title>(.*?)</title>", line, flags=0)
        if len(title)>0:
            l=list("".join(title))
            l1=[]
            for i in range(len(l)):
                if l[i].isupper() and (i>0):
                    l1.append(" ")
                l1.append(l[i])
            return [1,".join(l1)]
        else:
            return [0,None]
    except:
        return [0,None]
log=log.map(lambda x:(x,id(x)[0],id(x)[1]))
```

3.2 词条整合

- 去停用词,去除词性尾缀

定义 short 函数,对输入的文本进行 tf-idf 预处理,包括去除 HTML 标记 (strip_tags); 去除标点 (strip_punctuation), 去除数字 (strip_numeric),去除 is,in,to 等无用的词(remove_stopwords)。

```
Def short(log):

    log = log.lower()
    log = gensim.parsing.preprocessing.strip_tags(log)
    log = gensim.parsing.preprocessing.strip_punctuation(log)
    log = gensim.parsing.preprocessing.strip_numeric(log)

    log = gensim.parsing.preprocessing.remove_stopwords(log)

    stemmer = gensim.parsing.porter.PorterStemmer()
    log = stemmer.stem(log)
    log = gensim.parsing.preprocessing.strip_short(log, minsize=3)

    return log

data_clean = log.map(lambda x: [short(x[0]),x[1],x[2]])
```

- 文章标号

对 Tf-idf 预处理后的文本进行标号, 定义 give_index 函数, 定义全局变量 sum, 对词条标号进行累加, 作为文章 ID。

```
sum = 0.
def give_index(line):
    global sum
    if line[1] == 0:
        line[1] += sum
    else:
        line[1] += sum
        sum +=1
    return line
data_clean2= data_clean.map(give_index)
data_clean2.take(1)
```

- 词条行合并

定义 `combine` 函数，对属于同一词条的行合并为段落。由于 `map` 是顺序访问，定义全局变量 `title_before` 储存前一个访问到的词条标题，判断如果现在所在的行 `ID` 等于前一行的 `ID`，则字符串累加，否则判断为开启新词条。一旦判断旧词条结束，新词条开始，则返回上一词条的字符串累加结果，并赋值新的字符串，开始下一个字条的行合并过程。

```
I_str = ''
i_num=""
title_before=""
title_now=""
def combine(line):
    global i_str
    global i_num
    global result
    global title_before
    global title_now
    if line[0]== i_num:
        i_str=' '.join([i_str,line[1]])
        return ""
    elif line[2]:
        title_before=title_now
        title_now=line[2]
        i_num=line[0]
        tmp=i_str
        i_str=line[1]
        if len(title_before):
            return (i_num-1,tmp,title_before)
        else:
            return ""
    else:
        return ""
data4=data3.map(combine).filter(lambda x:len(x)>0)
data4.take(10)
```

4. 算法介绍：

4.1 概述：总体上，我们运用 `LSA`（潜在语义分析）算法对 `Wiki` 数据集进行处理并实现关键词搜索。所谓潜在语义分析，即通过 `SVD`（矩阵的奇异值分解），

将词和文档隐射到潜在语义空间，以去除如同义词、同时出现频率高的词对等，去除噪音，简化原始信息，使信息检索的准确率得以提升。

我们整个项目基本按照

数据预处理-->TF-IDF-->SVD-->计算文档相关矩阵-->实现关键词查询的流程处理。

步骤中的重要算法：

4.2 算法步骤：

Step1:TF-IDF

TF-IDF 顾名思义分为两部分：TF 即 Term Frequency，为词频，计算某个词在一篇文章中出现的频次；IDF 为 Inverse Document Frequency，可译作逆文本频率指数，在计算时，如果一个词出现的频率过高，则使它的重要度下降。TF 与 IDF 两个度量指标截然相反，一个随出现频率增大而增大，另一个随出现频率增大而减小。不难想象，只有一个词在其他文章中出现次数较少，唯独在一篇文章中频繁出现，它才会在这篇文章中得到 TF-IDF 高分。

因此综合考虑 TF 和 IDF 的 TF-IDF 算法，可以较为精确地给出某个词对某篇文章的关键程度，为实现关键词查询打下基础。

具体的计算公式为：

$$\begin{cases} tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \\ idf_i = \frac{|D|}{1 + |d \in D: t \in d|} \end{cases}$$

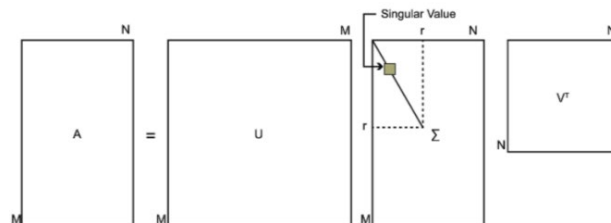
TF 的分子为某词在该篇文章中出现的次数，分母为文件中所有字词出现的次数之和。IDF 的分子为语料库中的文件总数，分母为包含该词语的文件数目（加 1 是为了避免分母为 0）。

将 TF 和 IDF 的结果相乘即可得到 $tfidf_{i,j} = tf_{i,j} \times idf_i$,在实际操作中, 经过 TF-IDF 步, 我们将得到一个文章 \times 词的矩阵。

Step2: SVD

SVD 即奇异值分解, 它建立在 EVD (特征矩阵分解) 的基础上。它将任意一个矩阵 A (大小为 $m \times n$) 分解为 $A = U \Sigma V^T$, 其中 U 是 $m \times m$ 的正交阵, 实际上是 $A^T A$ 的特征向量; Σ 是 $m \times n$ 的对角阵, 为其特征值; V 是 $n \times n$ 的正交阵, 实际上是 $A A^T$ 的特征向量。

下图更清晰地体现了 A 与 $U \Sigma V^T$ 的大小关系。



在算法中, 对 TF-IDF 矩阵做 SVD 分解, 实际上就是对文章 \times 词矩阵做分解, 取最具有代表性的一部分词, 过滤掉不重要的词。被过滤的词可能是同义词, 或是同时出现频率高的词, 总之进行 SVD 后文章 \times 词变小了。一方面提高查询时间, 精简存储空间, 另一方面可以提高查询的准确度, 是一个一举多得的步骤。

Step3: 计算文档/文档相关度矩阵

尝试 1: 小数据集 (前 10 万行)

我们首先在小规模数据集上实现了以上步骤, 即对文章内容预处理-->TF-IDF-->SVD-->文档/文档相关度矩阵-->关键词查询。随机进行关键词查询,

截图(Shift + Alt + A)

图 7：Geography 查询结果

尝试 2：大数据集（完整数据集，72G）

我们尝试将代码在完整数据集上运行，这着实耗费了很多时间，最终得出在单机上无法处理 72 个 G 数据的结论，因此我们搭建了 3 台计算机组成的 Spark 集群，计算时长总算降低到可以容忍的地步。

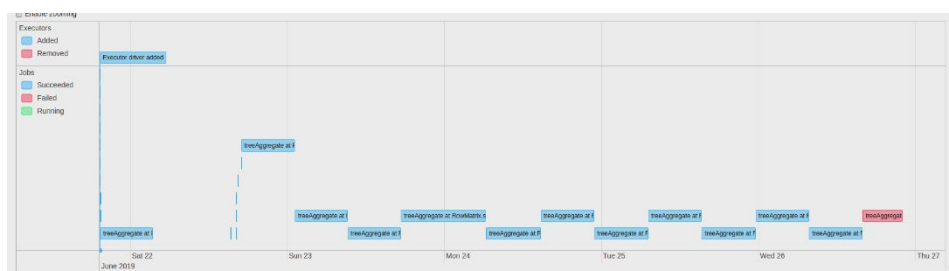


图 8：单机处理 72G 的数据记录

Spark 集群的搭建使每步 2600 次的迭代降低到 500 余次，对完整数据集的每次进行完整运算需要一天左右的时间（由于遇到内存不足等问题，完整跑完陆陆续续花费了 7 天时间），最终我们成功存储了 TF-IDF 和 ID 的结果，但可惜的是 SVD 的运行结果运行完成后未成功保存。

[illegible]

图 9.Svd 保存终止

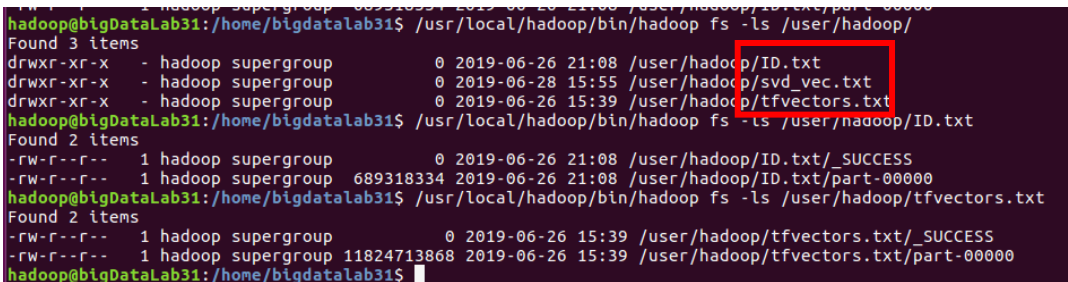


图 10: ID, TF-IDF, SVD 文件存储 (HDFS)

由于 rdd 文件是分布存储，所以需要先划分成一个 partition 再另存为 txt

```
$svd_vec.repartition(1).saveAsTextFile
```

HDFS 文件目录概览

ID.txt	储存了文章标号 ID 以及对应的文章标题，储存在在 HDFS 的 /user/Hadoop/ID.txt/part-00000
tfvectors.txt	储存了文章 tf-IDf 的分数，储存在在 HDFS 的 /user/Hadoop/tfvectors.txt/part-00000
Svd-vec	储存了文章 SVD 降维后的结果，可惜目前存储未成功
ID.txt 和 tfvectors.txt 可以横向匹配	

好在尽管只完成了 TF-IDF，我们依旧可以进行基础的查询。我们对数据集随机进行关键词查询

查询流程代码展示

```
tf=HashingTF(10000)
Item=tf.indexof("targetword")
```

以“Key”为例

```
In [59]: item=tf.indexOf("Key")
print(item)

3743
```

图 11:HashingTF 转化效果

在 hdfs 的 TF-IDF 存储结果 (tfvector.txt) 按 index 查询这个词，(此处以 key

为例) 的 tf-idf 分数

预处理:从 HDFS 中读取 tfvector.txt,并将每一行从字符串形式转为[(index, value), ID]的形式, 命名为 tfidf_index 的 rdd 文件。

```
Def qtf(line)#定义查询 tf-idf 函数代码
    Index=line[0][0]
    Value=line[0][1]
    Id=line[1]
    global item
    if str(item) in index:
    try:
        return (Id,value[i])
    except:
        return ""
    else:
        return ""
target=tfidf_index.map(qtf).filter(lambda: len(line)>0).take(10)
```

```
In [68]: target
Out[68]: [(3118, 8287.0),
(6210, 0.0),
(79116, 6.158271599204199),
(208272, 4.505964251405044),
(115074, 0.008465792131588505),
(115892, 7.080800966493641),
(212598, 15.49772290042952),
(13355, 27.64935691337979),
(7300, 6.740148743818815),
(201743, 4.6936137951390345e-07)]
```

图 12: tf-idf 查询结果(文章 ID, tf-idf 分数)

将 tf-idf 查询结果与文章标题进行匹配

预处理: 读取 HDFS 中的 ID.txt 文件并处理字符串 map 为每行 (id, title)

的形式,保存为名称为 id 的 rdd 文件

展示搜索结果 top10 的词条标题

```
l=[i[0] for i in target]
id.filter(lambda line:int(line[0]) in l).take(10)
```

尝试 3: 标题优先-小数据集

在前三次尝试中,我们发现查询的关键词有时甚至没有出现在相关度前十的文章

标题中，这与我们平时使用搜索引擎查找关键词时得到的结果不太一样。这是由于在进行 TF-IDF 时并没有加上标题的原因。但不难想象，哪怕将标题加入文章内容一起进行 TF-IDF，也不会使标题中的词的 TF-IDF 分数提高很多（因为这就相当于将其出现频次增加 1 而已）。

于是我们考虑给标题赋权，但这也不好处理。如何合理增大其 TF-IDF 分数是一个难题。于是我们决定在进行查询时，先筛选出标题中含有该关键词的文章。尽管这样做会忽略标题中不包含该关键词，但关键词在文章中频繁出现，与关键词有一定关联的文章，但我们猜测最终相关度前十的查询结果必然比不进行这一步要高很多。

为了进行这一实验，我们随机构造了一个包含 50 个单词的列表（包括名词、专有名词、动词、形容词等），如下图所示。

```
'phone','university','environment','heart','key','deadline','world','politics','imagination','prison','for  
give','artificial','reflection','health','stress','passion','kingdom','ration','individual','mountain','consc  
ious','prepare','sunshine','attitude','darkness','prosper','stochastic','lemology','parasite','hepatitis  
'memory','strech','graphics','offender','stowaway','bullish','obituary','eden','intelligence','pipe','  
military','plastic','civilization','compass','character','mansion','dream','dynasty','emperor','laser'
```

图 13：单词列表

取出标题包含其中任意一个词的文章，对由这些文章构成的数据集再进行 TF-IDF-->SVD-->文档/文档相关度矩阵-->实现查询的步骤。

对结果进行随机查询，查询结果如下

```
print(topDocsForTerm('environment'))  
  
index of term 4823  
[('Environment variable', 4.19572046104269462e-15), ('Environment movement', 3.71945104610573651e-15)], ('Environmental history', 3.49361594  
104751829e-15), ('Environmental law', 3.49197518437517491e-15), ('Environmental medicine', 3.08135914157113479e-15), ('Lists of environmenta  
l topics', 2.41584614397419375e-15), ('Environmental manager', 2.41584614397419375e-15), ('Environmental skepticism', 2.18523731845914398e-1  
5), ('Environmental philosophy', 1.71731242341234676e-15), ('Environmental Principles and Policies', 1.46128347361539649e-15)]
```

图 14 查询结果

尝试 4：标题优先-大数据集

将尝试 3 中类似的步骤应用到更大的数据集中。在查询时将算法改为筛选出关键词出现在标题中的文章，再对相关度进行降序排列。但由于时间关系，这一步

没有计算到最后。

代码：尝试 1-3 的代码请见 `whole_dataset.py` 和 `small_dataset.py`，它们并没有很大区别，只是一个在单机运行，一个在集群运行。标题优先的代码请见 `title_first.py`。这些代码文件都包含了详细注释，且保证可以顺利运行。

5.UI

5.1 前端设计

模式设计

我们在前后端的接口设计上采用任务驱动模式，避免了多次查询时“链接-查询”的复杂轮询结构。

实现细节

利用 `flask` 作简单 `router` 和任务触发时的请求调度。为了使 `server` 与事物查询语言的统一，我们在请求调度的 `spark` 接口的语言上也使用 `python`。同时我们还利用 `jinja2` 模板引擎控制结构动态渲染生成搜索结果部分的详细信息。

接口设计

我们在查询的结果上提供了两个维度的信息，一个是查询的时间，一个是查询结果的标题，以及还有一个是得到此结果的概率，我们在查询过程中 `js` 端的 `sparkCal` 部分提供了参数 `nums` 来控制显示搜索结果的数量，更加贴近实际文章查询场景。

还可以拓展的地方是后端缓存，由于时间原因，没有实现后端缓存。对于单次搜索，查询任务始终是一个单个查询词。如果需要拓展，对接缓存数据库我们还需要在 `sparkCal` 函数上做一些修改，这是可以实现的。同时，

我们在改进模板渲染时的数据部分是还可以采用字典结构，因为这对于前后端接口间的 json 文件传递有好的支持，容易适配其它高级前端框架。

以下是我们的初始界面：

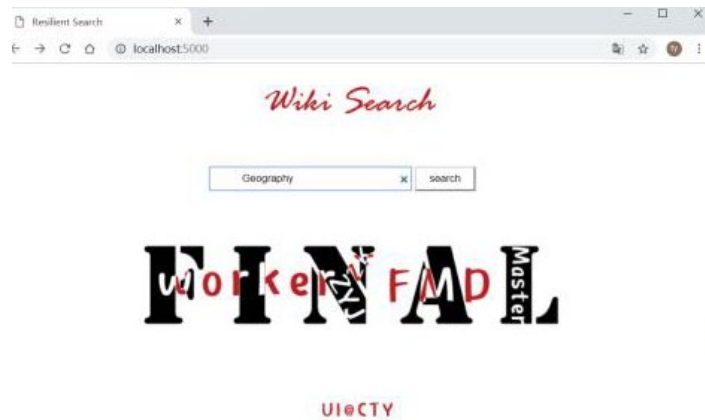


图 15：初始界面效果图

事件驱动

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Resilient Search</title>
</head>

<body background={{url_for('static', filename='pics/background.png')}} ;background-size:100%
100%;>
  <form method="POST" action="/search" style=" width:380px; margin:30px auto; margin-left:-
190px;font-size: 14px; position: absolute;left: 50%;top: 20%">
    <input type="search" name="keyWord" style="width: 260px;height: 30px; float left;
padding: 0 0 0 40px;">
    <input type="submit" value="search" style=" width:78px; height: 32px; float right;
background: white; color: black; text-align: center;">
  </form>
</body>
</html>
```

页面生成

当我们点击搜索按钮时，我们的前端将向后端发送请求，来返回结果

```
<!DOCTYPE html>
```

```

<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Search Result</title>
</head>

<body>
  <div id="container">
    <div id="hd" class="ue-clear">
      <form method="POST" action="/search" style=" width:380px; margin:30px
auto;font-size: 14px;">
        <input type="search" name="keyWord" value="{{keyWord}}" style="width:
260px;height: 30px; padding: 0 0 0 40px; font-size: 14px">
        <input type="submit" value="search" style=" width:78px; height: 32px;
background: white; color: black; text-align: center; font-size: 14px">
      </form>
    </div>

    <div class="resultInfo" style=" width:380px; margin:30px auto;font-size: 12px;">
      <p>
        <span class="info">totally: <span>{{nums}}</span></span>

      </p>
      <p>
        <span class="info">cost: <span>{{time}} s</span></span>

      </p>
    </div>

    <div class="resultList" >
      {% for search in search_result %}
      <div class="result" style=" width:380px; margin:30px auto;font-size: 14px;">
        <div class="title">
          <p><a style="color: red; text-align: left;">{{search.title}}</a></p>
          <p><a style="color: blue; text-align: right;">{{search.relation}}</a></p>
        </div>
      </div>
      {% endfor %}
    </div>
  </div>
</body>

```

以下是我们触发搜索事件之后的查询界面，结果包含了查询到的条数（默认显示最相关的前十条）、查询消耗时间、查询到的文章标题及分数。展示如

下:

Geography	search
-----------	--------

totally: 10
cost: 3.267024278640747 s

Geography of American Samoa
8.526375659428468e-18

Argentina/ Transnational Issues
5.597313882608661e-18

The Plague
3.9429794628871456e-18

Albania/ Transnational Issues
3.3871955581610277e-18

A priori and a posterior knowledge
3.2625233519856373e-18

图 16: 查询效果

5.2 后端工作

Flask 事件响应

后端使用 Flask 来相应前端, 在前端发来请求后, 后端 Flask 调用 python

函数 sparkCal 来获取结果:

```
# app.py
from flask import Flask, render_template, request
@app.route('/search', methods=['GET', 'POST'])
def search():

    # #test for json pass
    # result = sparkCal()
    # return jsonify(result)

    if request.method == 'POST':
        keyWord = request.form['keyWord']
        # # test for vue json
        # result = sparkCal(keyWord)
        # return jsonify(result)
        start_time = time.time()
        result = sparkCal(keyWord)
        end_time = time.time()
        nums = len(result)
        search_result = []
        for item in result:
            search_result.append({'title': item[0], 'relation': item[1]})
        time = end_time - start_time
```

```
        return render_template("result.html", search_result=search_result,
                                nums=nums, keyWord=keyWord, time=time)
```

连接 spark 服务器&读取已保存的 SVD 结果

每次调用函数 `sparkCal`，都会对 Spark 服务器进行链接，建立 `SparkContext`，同时载入我们之前保存好的 `npv` 文件，得到结果：

```
def sparkCal (term):
    import pyspark
    import numpy as np
    from pyspark import SparkConf, SparkContext
    from pyspark.mllib.feature import HashingTF
    import numpy as np

    def topDocsForTerm(term):
        try:
            index = tf.indexOf(term)
            # print("index of term", index)
            term_row = V[index]
        except:
            print("Term doesn't exist")
            return

        cosine_sim = np.dot(U, np.dot(S, term_row))
        indeces = np.argsort(cosine_sim).tolist()
        indeces.reverse()

        return list(zip(titles[indeces[:10]], cosine_sim[indeces]))

    conf=(SparkConf().set("spark.local.ip","10.192.7.116"))
    sc=SparkContext(conf=conf)

    US_normalized = np.load("US_normalized.npy")
    V = np.load("V.npy")
    S = np.load("S.npy")
    U = np.load("U.npy")
    titles = np.load("titles.npy")
    tf=HashingTF(50000)
    res = topDocsForTerm(term)

    sc.stop()
```

```
return res
```

以上 npy 文件我们都在 SVD 分解后保存。

```
np.save("US_normalized",US_normalized)
np.save("V",V)
np.save("U",U)
np.save("S",S)
np.save("titles",titles)
```

6.工作总结

6.1 功能实现

- ✓ 完成了在三台主机的小型 spark 集群搭建与分布式部署,并实现了远程访问使用 jupyter notebook 在 pyspark 上启动 spark 集群.
- ✓ 对 72G 的 wikipedia 数据完成了 TF-IDF 的计算, SVD 矩阵的计算, 并将 TF-IDF 结果(tfvectors.txt), 与文章 ID(ID.txt),svd 结果(svd_vec.txt) 保存在 HDFS 中。
- ✓ 开发了查询的文档查询流程; 实现了基于文档标题, 词频, TF-IDF 抽取结果, SVD 降维结果的综合抽取, 以改进 TF-IDF 与 SVD 的文档查询结果。
- ✓ 实现了所给词的查询最相关的文档的功能, 并且在 Web 页面中实现了前端查询与前端展示。

6.2 反思与改进

不足 1: 没有实现后端缓存

改进: 如果需要拓展, 对接缓存数据库我们还需要在 sparkCal 函数上做一些修改, 这是可以实现的。

不足 2: 前后端的接口文件的的可拓展性支持

改进: 目前我们的存储只是以 python 的包 numpy 自带的格式来存贮, 这意味

着在前端读取的时候必须经过，我们在改进模板渲染时的数据部分是还可以采用字典结构，因为这对于前后端接口间的 json 文件传递有好的支持，容易适配其它高级前端框架。

不足 3：我们的文字查询仅仅局限于一个词

改进：还可以进一步通过多个词来计算与文章相似度的距离，实现多个词的综合查询，此外，还可以通过多个词的组合成的语义来进行语义表示用以查询

不足 4：我们的词必须是要在词库中出现才能搜索

改进：通过引入拼写检查的方式可以保证我们不至于因为拼错一个字母而搜索失败

7. 小组分工

Spark 集群搭建：张可，冯孟笛

大数据集文档查询后端算法实现(72G):张可

小数据集文档查询后端算法实现:张言健

算法优化与设计：冯孟笛

UI 前端:陈天予

报告撰写：张可/冯孟笛/张言健

8. 参考文献

Spark 集群搭建：

1. Hadoop 完全分布式集群搭建

<https://blog.csdn.net/superman404/article/details/83591324>

2. 2.0-Spark 完全分布式集群安装

<https://blog.csdn.net/yyl424525/article/details/77428541>

3. Hadoop2.8.1 完全分布式环境搭建

<https://www.cnblogs.com/pcxie/p/7747317.html>

大数据集文档查询后端算法实现(72G):

1. jupyter 中使用 pyspark 连接 spark 集群

<https://blog.csdn.net/u013129944/article/details/80107214>

2. MLlib - Feature Extraction and Transformation

<https://spark.apache.org/docs/1.2.0/mllib-feature-extraction.html>

3. Write rdd as textfile using apache spark

<https://stackoverflow.com/questions/30993655/write-rdd-as-textfile-using-apache-spark>

小数据集文档查询后端算法实现(72G):

1. Understanding Wikipedia with Latent Semantic Analysis

https://github.com/dbaikova/Wikipedia_LSA/blob/master/WikipediaAnalysisWithLSA.ipynb

算法优化与设计

1. Singular value decomposition

https://en.wikipedia.org/wiki/Singular_value_decomposition

2. LSA 及 SVD 介绍

<https://blog.csdn.net/u013395878/article/details/51741706>