IR检索系统实验报告

计76 成镇宇 2017080068

实验环境

使用了在dockerhub上的`rn123/pylucene`Image

实验过程

1 分词标注

1-1 分词实现

分词用到的第三方库为thulac，先是用的python版本实现：

在命令行中输入

```
python main.py --training # 用python的thulac分词所有的语料库并进行索引搭建
python main.py --segmenting --limit 50000 # 用python的thulac分词所有的语料库，每个语料库只提取50000则新闻
```

但python的thulac真的花了好长时间，我的电脑带不动。后来换成了手动操作的c++，但也是非常的慢，就还是用python的了。

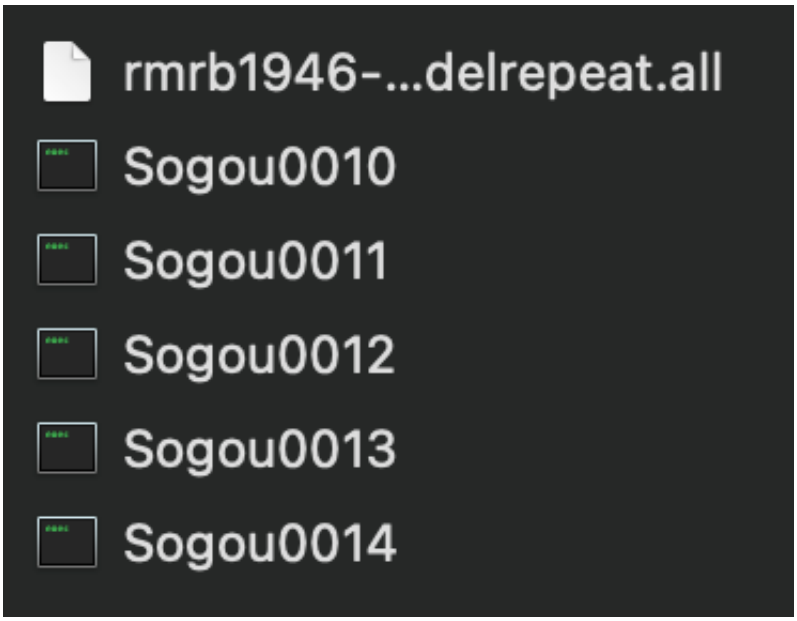
1-2 代码实现

```
from config import *
import os
import thulac

'''
  分词模块
'''
class SegProcessor:
  def __init__(self, segmentation_limit=LIMIT, training=False):
    self.training = training
```

```
    self.segmentation_limit = segmentation_limit
    self.file_loader()

  def file_loader(self):
    lac = thulac.thulac()
    with open(SEGMENTATION_FILE, 'w') as outfile:
      for file_name in os.listdir(CORPUS_DIR):
        if file_name == ".DS_Store":
          continue
        with open(CORPUS_DIR + file_name, "r") as f:
          print("Segmenting File [", CORPUS_DIR+file_name, "]")

          # 语料库是Sogou的情况下
          if "Sogou" in file_name:
            line_count = 0
            line = f.readline()
            while (line):
              terms = line.split(" ")
              for i in range(len(terms)):
                if terms[i] == '<N>':
                  terms[i] = '0'

              s = ""
              combined = s.join(terms) # merged a sentence
              segmented = lac.cut(combined, text=True) # segmentation process
              outfile.write("{}\n".format(segmented))

              # print(segmented)
              print("\r", str(line_count), " lines", end="", flush=True)
              line_count = line_count + 1
              if line_count > self.segmentation_limit and not self.training:
                f.close()
                break
              line = f.readline()

          # 语料库不是Sogou的情况下
          else:
            line = f.readline()
            line_count = 0
            while (line):
              segmented = lac.cut(line, text=True) # segmentation process
              outfile.write("{}\n".format(segmented))

              # print(segmented)
              print("\r", str(line_count), " lines", end="", flush=True)
              line_count = line_count + 1
              if line_count > self.segmentation_limit and not self.training:
                f.close()
                break
```

```
            line = f.readline()
        print()
```

1-3 语料库



2 索引建立

2-1 索引建立实现

该部分用于建立所有的文章的索引，对于语料库的每一个article当做一个Document。

建立索引用的Analyzer是whitespaceanalyzer，因为已经用thulac分析了词性分好词了，只需要根据空格实现即可。

之前的实现是直接把thulac生成的分词结果直接塞到lucene里面进行检索，但queryparse无法对`_`进行分析，所以就分成了两个来存储，一个field是文章内容，还有一个是词性。其中文章内容（context）的 fieldtype，需要搭建索引、存储、向量化；词性的（phrase）fieldtype，只需要存储。

2-2 代码实现

```
import os
import lucene
from config import *
from pathlib import Path

from java.nio.file import Paths
from org.apache.lucene.store import SimpleFSDirectory
from org.apache.lucene.analysis.core import WhitespaceAnalyzer
from org.apache.lucene.util import Version
```

```
from org.apache.lucene.index import IndexWriter, IndexWriterConfig,
IndexOptions
from org.apache.lucene.document import Document, Field, FieldType, StringField

'''
  索引搭建模块
'''
class Indexer:
  def __init__(self, path=INDEX_DIR, index_limit=LIMIT, training=False):
    self.index_limit = index_limit
    self.training = training

    p = Path(path)
    if not p.is_dir():
      os.mkdir(path)

    # 初始化lucene，准备好analyzer和writer
    lucene.initVM()
    indexdir = SimpleFSDirectory(Paths.get(path))
    analyzer = WhitespaceAnalyzer(Version.LATEST) # 由于thulac分词的时候已经实现了
用空格来表示不同的词，所以直接用空格分析器就可以。
    iwconf = IndexWriterConfig(analyzer)
    iwconf.setOpenMode(IndexWriterConfig.OpenMode.CREATE_OR_APPEND)
    index_writer = IndexWriter(indexdir, iwconf)

    self.Indexing(index_writer)

  def Indexing(self, writer):
    print("Indexing Segmented File [", SEGMENTATION_FILE, "]")
    with open(SEGMENTATION_FILE, 'r') as f:
      line_count = 0
      for line in f:
        # 建立 context 的 fieldtype，需要搭建索引、存储、向量化
        fieldtype_context = FieldType()
        fieldtype_context.setIndexOptions(IndexOptions.DOCS_AND_FREQS)
        fieldtype_context.setStored(True)
        fieldtype_context.setTokenized(True)

        # 建立 phrase 的 fieldtype，只需要保存
        fieldtype_phrase = FieldType()
        fieldtype_phrase.setStored(True)

        # 对分词好的内容进行处理，把词语和词性分开来存储
        processed_context, processed_phrase = self.process_line(line)

        doc = Document()
        # context field是用于记录文章的内容
        doc.add(Field('context', processed_context, fieldtype_context))
        # phrase field适用于记录文章每个词所对应的词性
```

```
        doc.add(Field('phrase', processed_phrase, fieldtype_phrase))

        # 把document写入索引库
        writer.addDocument(doc)

        # 跟踪程序运行情况用
        print("\r", str(line_count), " lines", end="", flush=True)
        line_count = line_count + 1
        if line_count > self.index_limit and not self.training:
          break

    writer.close()
    print()

  # 对分词好的内容进行处理，把词语和词性分开来存储
  def process_line(self, line):
    processed_context = []
    processed_phrase = []
    terms = line.split(' ')
    for index, term in enumerate(terms):
      splitted = term.split('_')
      if len(splitted) > 1:
        processed_context.append(splitted[0])
        processed_phrase.append(splitted[1])

    return ' '.join(processed_context), ' '.join(processed_phrase)
```

2.2.4 Retriever

2.2.4.1 一般查询

该部分用于对于建立好的索引进行查询，输入query，用分词器进行分析，然后通过searcher进行查询操作，由于查询量较大，而且每个语句起码包含一个搭配，所以查询的document个数设置为了100，然后返回的搭配为40个。

```
def search(self, term, window=2):
  self.hits = []
  index_list = []
  sort_para = term

  parser = QueryParser('text', self.analyzer)
  query = parser.parse(term)

  # Jump to multi-terms search if there are several words
  if self.multi_terms(query):
    self.search_multi_terms(query)
    return self.hits[:40]

  hits = self.searcher.search(query, 100).scoreDocs
```

```
  for hit in hits:
    index = []
    doc = self.searcher.doc(hit.doc)
    text = doc.get("text")
    self.hits.append(text)
    # save indexes of target term in each document
    terms = text.split()
    for i in range(len(terms)):
      if term == terms[i]:
        index.append(i)
    index_list.append(index)

  self.recover_sentence(index_list, window)
  hits_copy = self.hits
  self.hits = []
    # Add font tags for highlighting target terms, return to backend
  for hit in hits_copy:
    simpleHTMLFormatter = SimpleHTMLFormatter(prefixHTML, suffixHTML)
    highlighter = Highlighter(simpleHTMLFormatter, QueryScorer(query))
    highLightText = highlighter.getBestFragment(self.analyzer, 'text', hit)
    if highLightText is not None:
      self.hits.append(highLightText)
  print('search over')
  return self.hits[:40]
```

首先，通过之前定义的WhiteSpaceAnalyzer，对接收的query进行分析，因为我们只需要对单个关键词进行查询，没有空格，所以分析出来就是初始的关键词。

然后用searcher查询documents，上面的hits即为命中的document和它的一些相关信息。

接着通过get()函数对保存的句子进行提取，进行处理

句子中提取搭配

recover_sentence函数是在整个Retriever文件中最主要的部分。

```
def recover_sentence(self, indexs, window=2):
  hits_copy = self.hits
  self.hits = []
  for i in range(len(hits_copy)):
    terms = hits_copy[i].split()
    length = len(terms)

    for index in indexs[i]:
      combination = ''
      if index == 0:
        sentence = terms[0] + terms[1] if terms[1] not in punctuation else
terms[0]
        combination += sentence + ' '
```

```
      elif index == 1:
        sentence = terms[0] + terms[1] + terms[2] if terms[2] not in
punctuation else terms[0] + terms[1]
        combination += sentence + ' '
      elif index == length-2:
        sentence = terms[length-3] + terms[length-2] if terms[length-3] not in
punctuation else terms[length-2]
        combination += sentence + ' '
      elif index == length-3:
        sentence = terms[length-4] + terms[length-3] + terms[length-2] if
terms[length-4] not in punctuation else terms[length-3] + terms[length-2]
      elif index >= 2 and index <= length-4:
        combination = self.check_available(index, terms, window)

      # delete punc at first of sentence
      combination = self.replace_punc(combination)
      self.hits.append(combination)


  # delete duplicated datas and null data
  self.hits = list(set(self.hits))
  self.hits.sort(key=cmp_to_key(lambda x, y : self.compare(x, y)))
  if '' in self.hits:
    self.hits.remove('')
  self.hits = self.hits[:100]
  # print(self.hits)
```

对于常见搭配来说，一般都由2到3个词语组成，当然有一些例外甚至更长，但我们只考虑一般情况。

在命中了document之后，我将每个document中的target词语的index保存了起来，这样我就能知道它在句子中的那个位置。

在我的算法里，是以关键词的index为中心，窗口大小为2，即往前往后数各2个词，规定为常见搭配。

它大概能分为5种情况：

- 关键词在开头
- 在开头+1的位置
- 在末尾-2
- 在末尾-3
- 在大于开头+1 小于末尾-3

很容易会诧异，为什么是末尾-2和-3，是因为我在我的数据中发现，将document.split()之后，在末尾都为‘’，并且大部分都是句子，所以末尾-1相当于是真正的末尾，会有标点符号。

所以前四种情况判断特殊位置有没有标点符号，没有就组合成常见搭配存起来，关键是最后一种情况，我写了一个函数，为了支持后面的距离限制查询。

```
def check_available(self, index, terms, window=2):
  sentence = terms[index]
  length = len(terms)
```

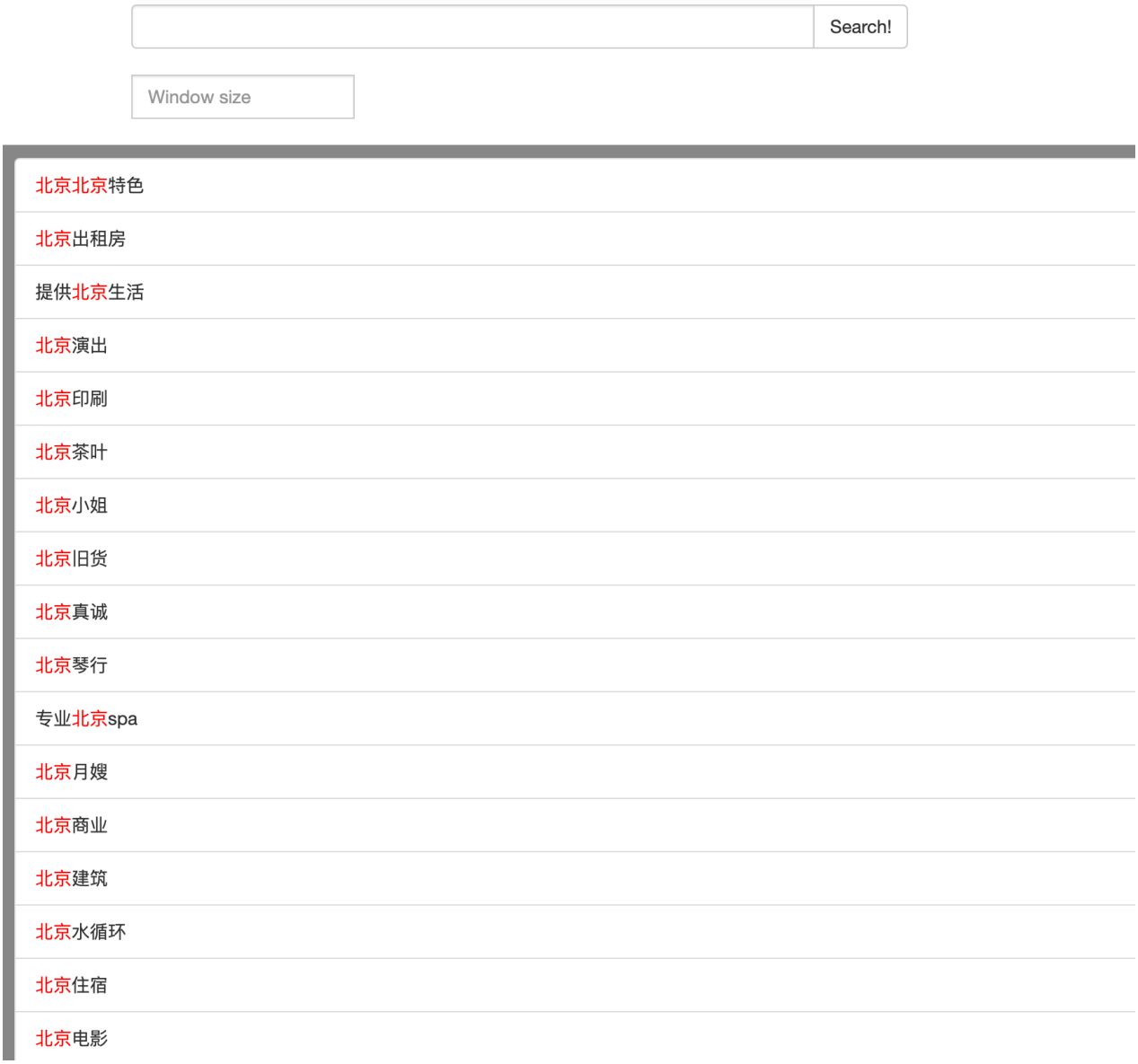
```
    left = index - window if index - window >= 0 else 0
    right = index + window if index + window <= length-1 else length-1
    l = index - 1
    r = index + 1
    while left < l and r < right:
      if terms[l] not in punctuation:
        temp = terms[l]
        temp += sentence
        sentence = temp
        l -= 1
      if terms[r] not in punctuation:
        sentence += terms[r]
        r += 1
      if terms[l] in punctuation and terms[r] in punctuation:
        break
    return sentence
```

默认距离window为2，用在一般查询。

该算法流程为：

1. 初始化两边临界值，window决定该参数；定位关键词
2. 从关键词左右各一位开始判断，若遇到词语，再往左或往右一个，往两边扩散；若遇到标点符号，该方向计数不变。
3. 退出条件为两边都到达临界值，或是两边都卡在标点符号的位置。

后来发现，分析出来的搭配有很多前面加‘的’，‘就’等常见搭配，于是我在第二部分加上了中文停用词，发现结果好了很多。



2.2.4.2 给定词性查询

查询方式：在搜索栏输入: 词语/词性

在前面有提到过，我将句子和词性分开保存，比如：

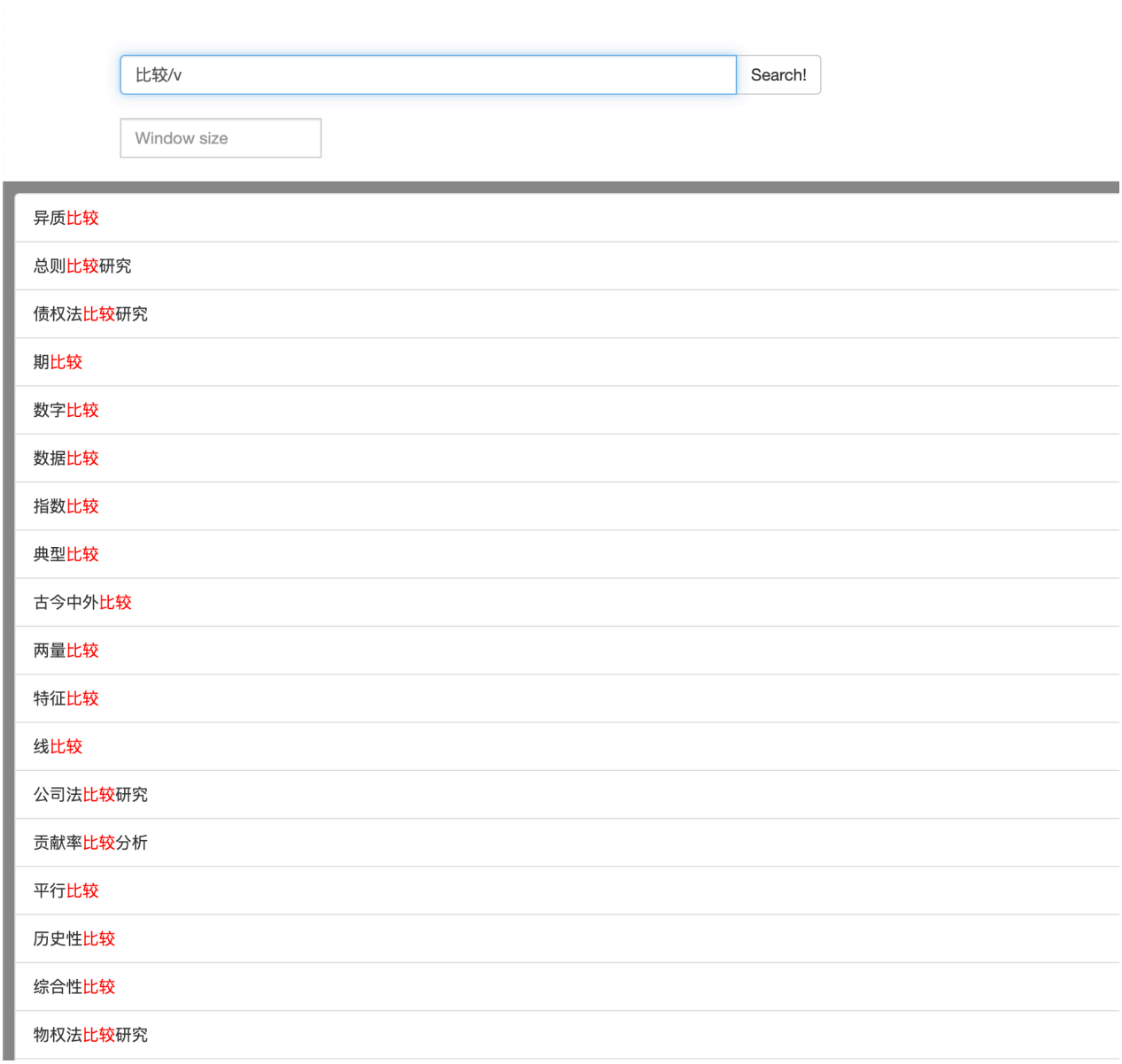
```
北京/ns 是/v 城市/n
-> text : '北京 是 城市'
   phrase : 'ns v n'
```

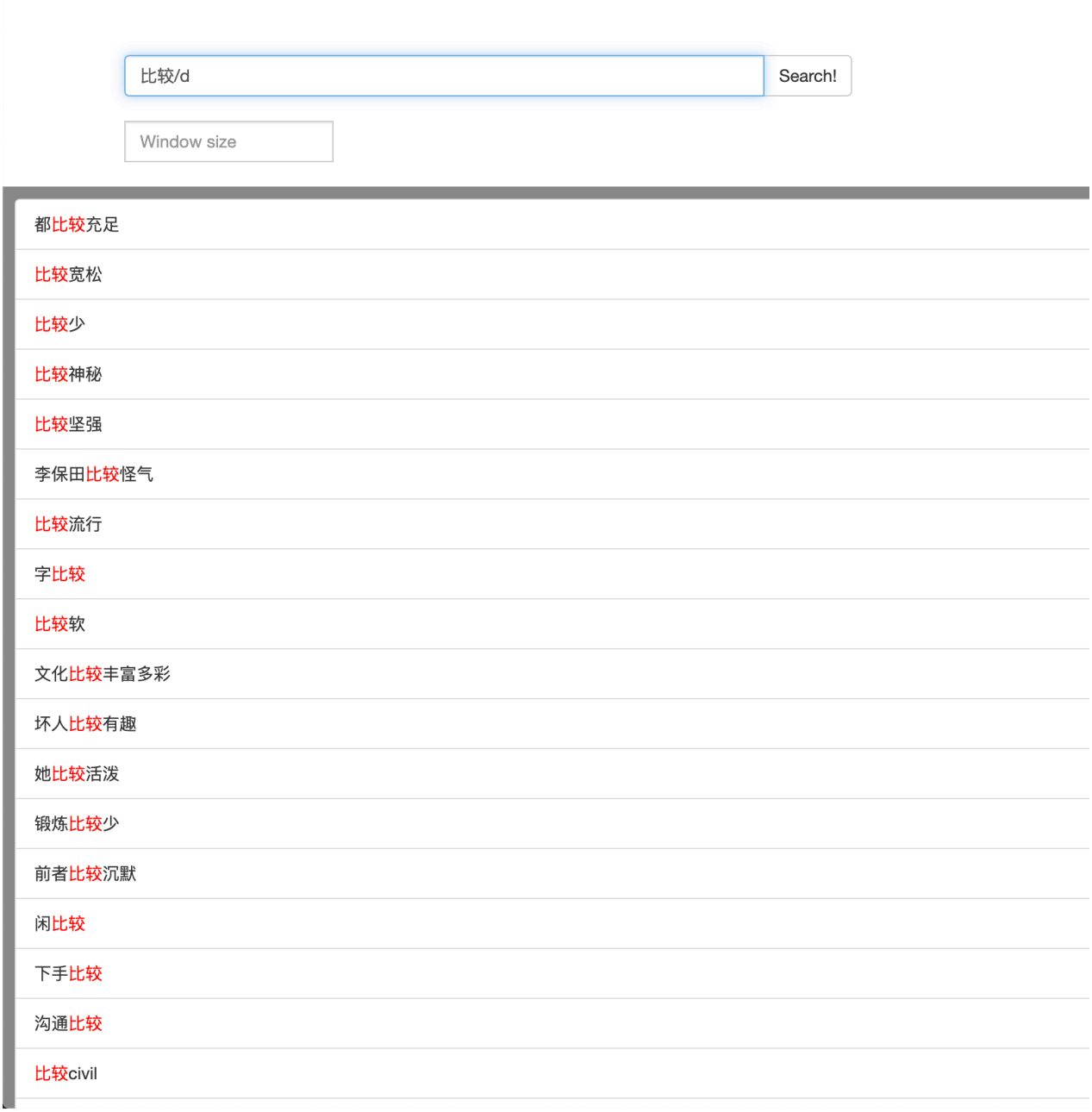
这么做的目的主要是，我们不需要用中文分词器重新进行分词，只需要按照空格分就好，但如果直接把已经处理好的语料库导进去建立索引，在后面查询的时候'/'是无法放进query里进行查询的，只能用词语进行查询，而直接分词的话会将'北京/ns'作为一个词，无法查询到'北京'。

得到输入的 词语/词性后，将它们分开存起来，然后通过searcher对词语进行查询。得到查询结果后，对于命中的每篇document，读取它的text field和phrase field，按照空格分词，然后对于每个对应关键词，判断词性与输入的词性是否一致，可能会有多个对应关键词，只要有一个词性不同，就选择丢弃该结果，最后存到self.hits里，之后的步骤与一般查询相似。

```
def search_phrase(self, term, phrase):
  print('Phrase search')
  self.hits = []
  index_list = []
  parser = QueryParser('text', self.analyzer)
  query = parser.parse(term)

  hits = self.searcher.search(query, 1000).scoreDocs
  if hits is None:
    return

  for hit in hits:
    index = []
    doc = self.searcher.doc(hit.doc)
    text = doc.get("text")
    phrases = doc.get("phrase")

    # processing with saved text and phrase
    terms = text.split()
    phrases = phrases.split()
    flag = 1 # this flag is judging for phrase in every target term in text
    index = [] # index number for searched term, maybe many terms
    for i in range(len(terms)):
      if term == terms[i]:
        index.append(i)
        if not phrase == phrases[i]:
          flag = 0
          break;
    if flag == 1:
      self.hits.append(text)
      index_list.append(index)
  self.recover_sentence(index_list)
  hits_copy = self.hits
  self.hits = []
  # add font tags for terms
  for hit in hits_copy:
    simpleHTMLFormatter = SimpleHTMLFormatter(prefixHTML, suffixHTML)
    highlighter = Highlighter(simpleHTMLFormatter, QueryScorer(query))
    highLightText = highlighter.getBestFragment(self.analyzer, 'text', hit)
    if highLightText is not None:
      self.hits.append(highLightText)

  return self.hits[:40]
```

对词语'比较'进行了查询（有动词和副词形式）



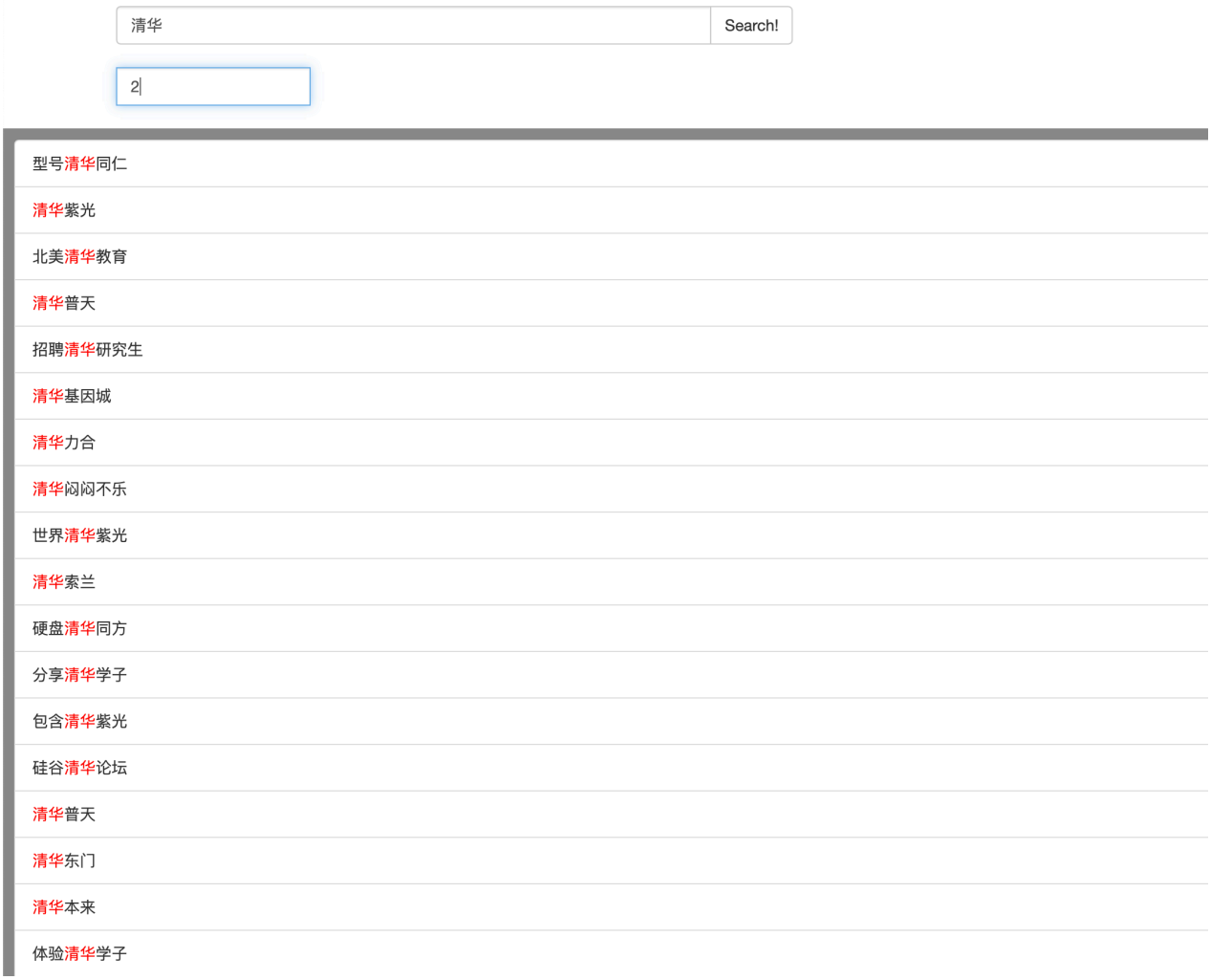


可以发现效果良好，词性分的很清楚。

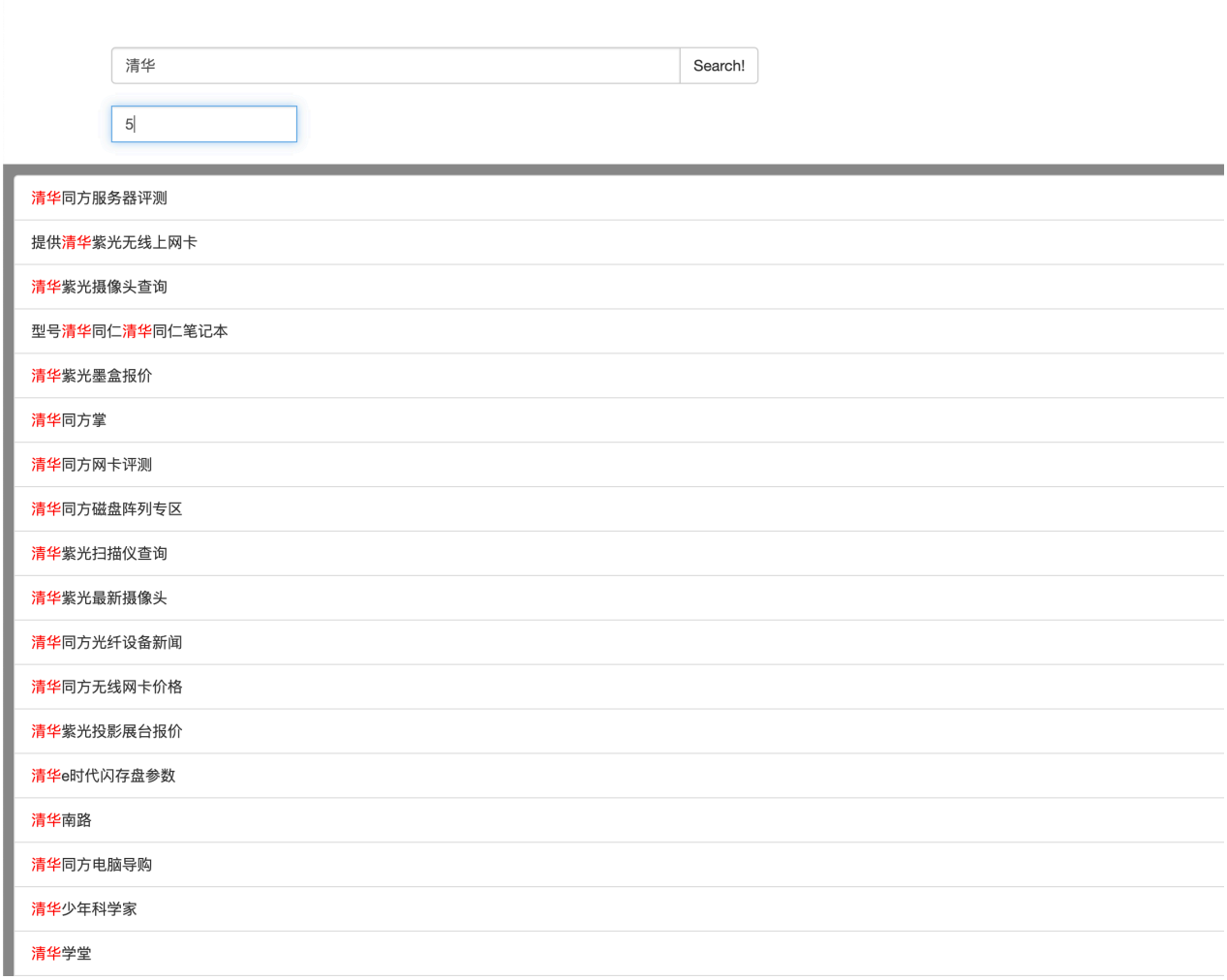
2.2.4.3 给定关键词的距离限制查询

在前面2.2.4.1提到的check_available函数，有一个window参数，它相当于是可以调的距离限制，也就是窗口大小，只要调整窗口大小即可完成对于给定关键词的距离限制查询。

window = 2:


清华
Search!
2
型号清华同仁
清华紫光
北美清华教育
清华普天
招聘清华研究生
清华基因城
清华力合
清华闷闷不乐
世界清华紫光
清华索兰
硬盘清华同方
分享清华学子
包含清华紫光
硅谷清华论坛
清华普天
清华东门
清华本来
体验清华学子

window = 5:



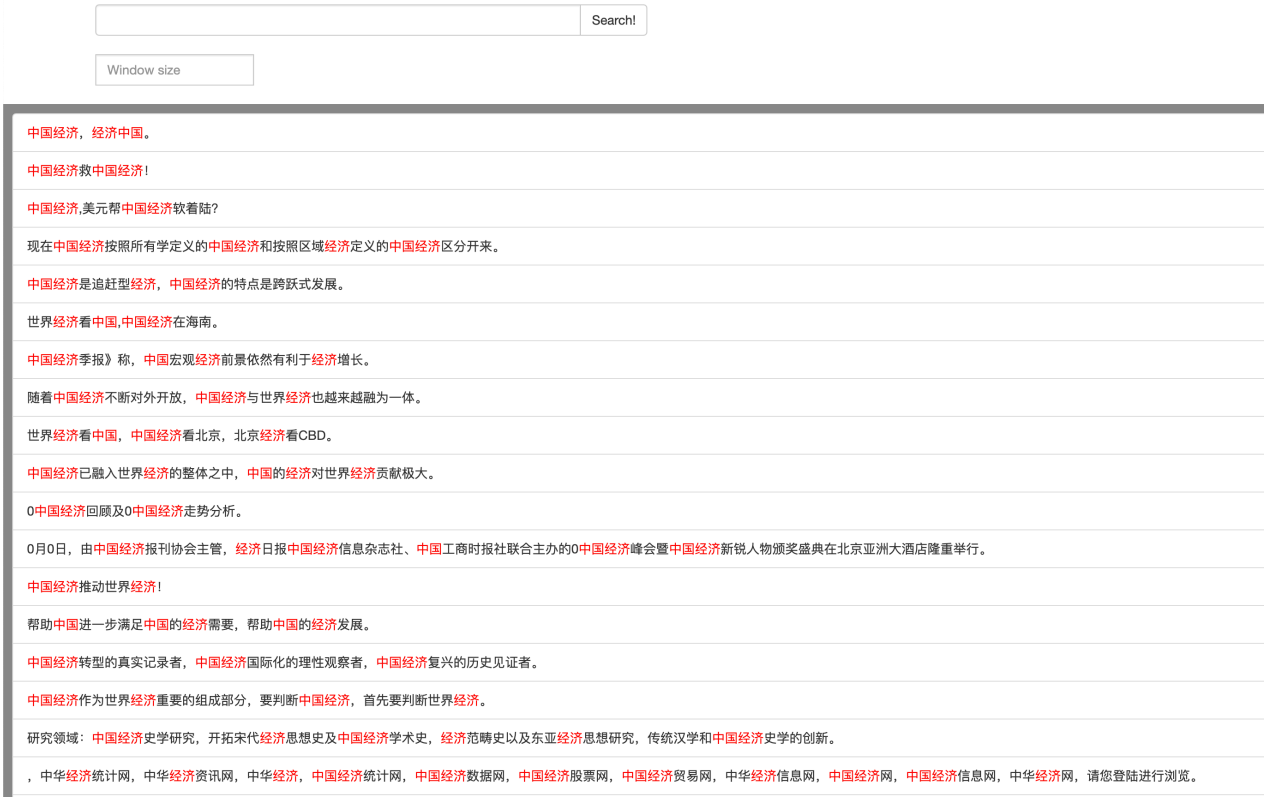
不过会发现，当窗口等于5时，也有一些距离短的搭配，是因为我的算法遇到标点符号或是停用词就停止，不管窗口多大，遇到标点符号就不算是搭配了，所以还是会出现一些限制距离外的一些搭配

2.2.4.4 给定多个关键词进行查询

在这个部分，我发现输入多个关键词，lucene会查询与他们相关的文档，而且每个document都有包含许多短语，即搭配，所以我直接给出了整个句子。

```
def search_multi_terms(self, query):
  print('Multiterms search')
  hits = self.searcher.search(query, 100).scoreDocs
  for hit in hits:
    doc = self.searcher.doc(hit.doc)
    text = doc.get("text")
    terms = text.split()
    sentence = ''
    for term in terms:
      sentence += term
        # add tags
    simpleHTMLFormatter = SimpleHTMLFormatter(prefixHTML, suffixHTML)
    highlighter = Highlighter(simpleHTMLFormatter, QueryScorer(query))
    highLightText = highlighter.getBestFragment(self.analyzer, 'text',
sentence)
    if highLightText is not None:
```

```
        self.hits.append(highLightText)
```



2.2.5 flask

```
# coding=utf-8
from indexer import *
from retriever import *
from pathlib import Path
import os
from flask import Flask, render_template, request, redirect, url_for

#INDEX_DIR = '/root/IR_system/flask_app/data'
INDEX_DIR = '/Users/kim/Desktop/Git/IRsystem/flask_app/data/'
CORPUS_DIR = '/Users/kim/Desktop/corpus/'
app = Flask(__name__, instance_relative_config=True)

# initialize model
lucene.initVM()
retriever = Retriever(INDEX_DIR)


# a simple page that says hello
@app.route('/')
def hello():
  return redirect(url_for('home'))

@app.route('/home')
```

```
def home():
  return render_template('home.html')

@app.route('/search_phrase')
def search_phrase(query, retriever):
  query = query.split('/')
  term = query[0]
  phrase = query[1]
  hits = retriever.search_phrase(term, phrase)
  if hits:
    return hits
  else:
    return ''

@app.route('/search', methods=['GET', 'POST'])
def search():
  query = request.args.get('query')
  print(query)
  if query == '':
    return render_template('nothing.html')
  window = 2 if request.args.get('window') is '' else
int(request.args.get('window'))
  if '/' in query:
    hits = search_phrase(query, retriever)
  else:
    hits = retriever.search(query, window)
  print(len(hits))
  return render_template('result.html', hits=hits) if hits != '' else
render_template('nothing.html')

if __name__ == '__main__':
  # initialize indexer
  print('Initializing...')
    # 第一次建立索引需要去掉下面一行的注释
  # Indexer(INDEX_DIR)
  app.run()
```

flask初始化代码中，主要工作为：

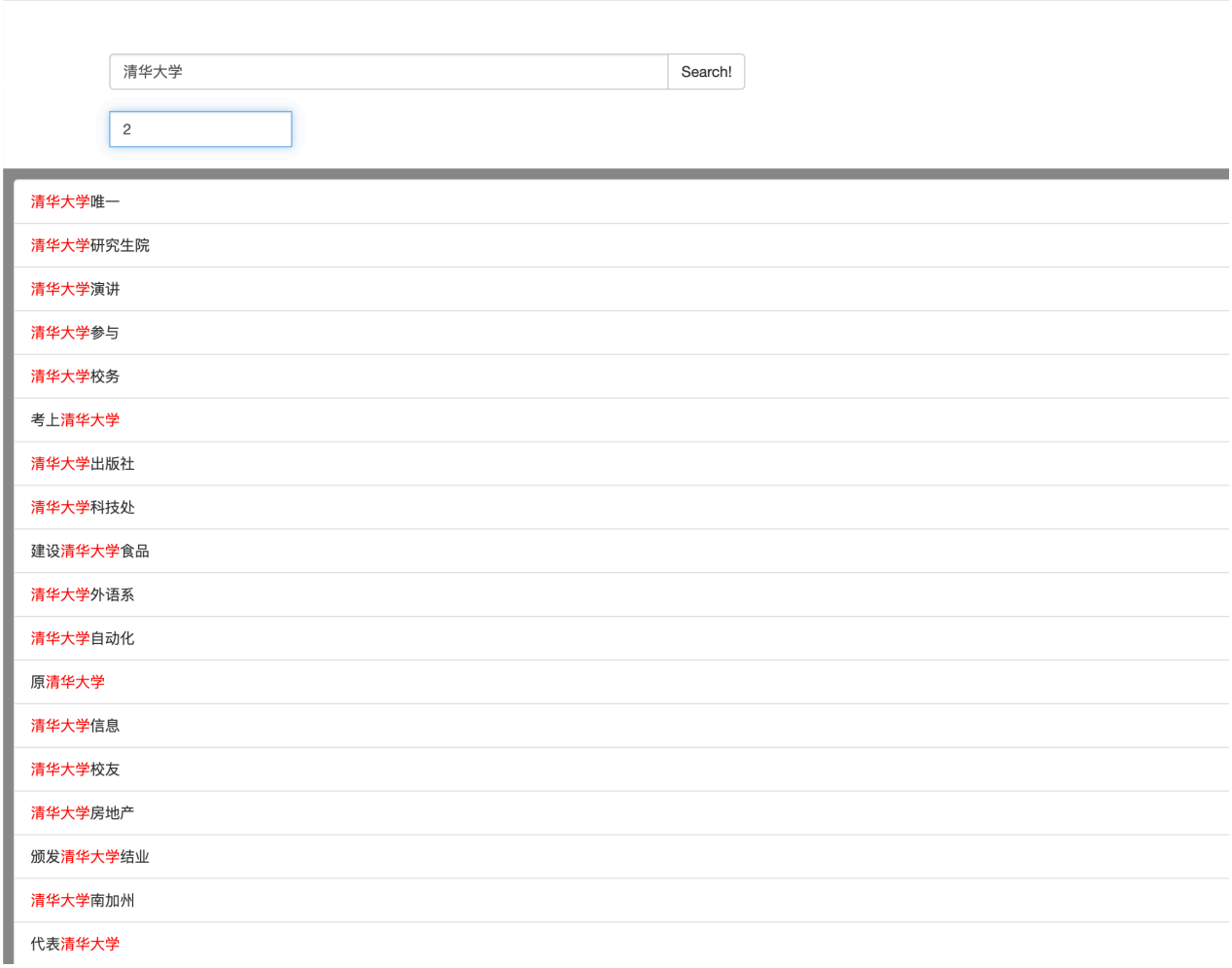
- 建立索引（只需要建立一次）
- 初始化Retriever
- 建立路由，进行查询

先是用query进行判断，是否有'/'，有则为词性查询，没有则为其他查询。这部分是在flask进行判断。最后用模板展示结果内容。

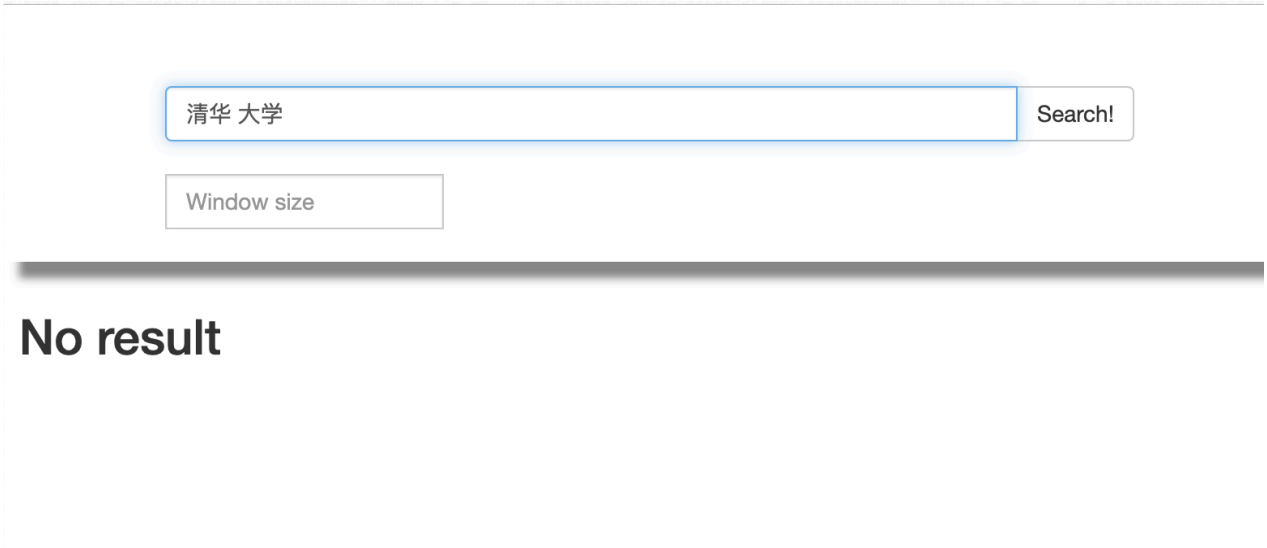
3. 分词与不分词的差别

查询词：清华大学， 清华 大学，大学清华，大学 清华 (窗口大小默认为2)

首先对清华大学进行了查询：



然后查询‘清华 大学’时，出现了问题，返回了空值。



然后搜了‘大学清华’和‘大学 清华’，结果都一样：

大学清华 Search!

Window size

清华，眼前这个大学就是清华?

清华、同济大学；

清华等名牌大学学习。

清华可是国际名牌大学。

北大清华，二流大学?

清华中学”、“清华小学”，还叫什么大学！

清华紫光主要推荐菲律宾雅典耀大学和国父大学。

大学经济学学士，清华EMBA。

北大、清华沦为二流大学?

深圳大学城势比北大清华?

北大清华正在沦为二流大学！

现代气息充溢着清华校园，清华展示出一流大学的魅力。

大学读香港的大学好还是读清华北大好?

转帖]香港大学将北大、清华扫为二流大学?

公司的面授教室位于清华科技园，地处五道口，紧邻清华东门，地理环境极为优越：清华、北大、中科院、语言文化大学、地质大学、林业大学、农业大学等著名高校围绕周边；

清华，中国最好的大学。

香港大学的确比北大清华好。

为清华发展成为大学初创基础。

大学 清华 Search!

Window size

清华，眼前这个大学就是清华?

清华、同济大学；

清华等名牌大学学习。

清华可是国际名牌大学。

北大清华，二流大学?

清华中学”、“清华小学”，还叫什么大学！

清华紫光主要推荐菲律宾雅典耀大学和国父大学。

大学经济学学士，清华EMBA。

北大、清华沦为二流大学?

深圳大学城势比北大清华?

北大清华正在沦为二流大学！

现代气息充溢着清华校园，清华展示出一流大学的魅力。

大学读香港的大学好还是读清华北大好?

转帖]香港大学将北大、清华扫为二流大学?

公司的面授教室位于清华科技园，地处五道口，紧邻清华东门，地理环境极为优越：清华、北大、中科院、语言文化大学、地质大学、林业大学、农业大学等著名高校围绕周边；

清华，中国最好的大学。

香港大学的确比北大清华好。

为清华发展成为大学初创基础。

通过调试以及分析，我发现该问题是处在lucene自带的analyzer中。

analyzer是在建立索引或查询文档时用到的分析器，它有许多种，其中，我在建立索引时用的是WhiteSpaceAnalyzer，关键是我在查询时分析query用的是SmartChineseAnalyzer。

SmartChineseAnalyzer，顾名思义，是专门对中文进行分词的分析器，它会根据词典对文本进行分词。而'清华大学'这个名称，是一个学校名称，固有名词，所以搜索'清华大学'的时候，它会正常给出搭配；但搜索'清华 大学'的时候，分析器会将它看做一个词，而不是'清华'，'大学'。它会有文档命中，但按照我定义的搭配，该算法会将它看做是多词条搜索，但是分析器将它作为一个词，发生冲突，所以不会给出任何搭配，但原文档还是有命中的。当然，'大学'，'清华'按照这个顺序的话，它必定分为两个词，识别为多词条查询，给出搭配。于是可发现分词与不分词的差别之处。

4. 实验总结

通过本次实验比较深入的了解了倒排文件结构系统，并学习了如何用第三方工具建立索引，以及分析返回的结果，也提高了我对python编程的能力

本次实验遇到的问题不少，最主要是pylucene的环境配置不太友好，api文档只有java lucene的，虽然官方说java api文档中的 api都可以用，但实际上想用的时候还是发现很多问题，比如没有文档中所说的类，或是对于不同平台Directory类该用哪些，所以感觉java版的可能更亲民一些。

5. 一些实验中的测试

实验中遇到的：

1. 搭建pylucene环境的时候遇到了很多的问题，特别是在java的配置上，由于我的电脑上有java8和java12，所以总是会出问题，花了两天多的时间，后来一个docker image就解决了。

1. 一开始的索引策略是：用SmartChineseAnalyzer，导入两个文件的语料库：一个是分好词的，一个是原始数据，导入两个field，建立索引时间是本次实验索引建立时间的三倍，所有没有取该方法。
2. SmartChineseAnalyzer无法直接对分好的语料（带词性）直接进行分词并建立索引，它会直接把词性也一起放进来进行分词，用了stopwords也没啥效果。
3. 有一种叫StringField，将Field代替为它，但实际上没有太大优化效果。