信息检索系统

信息知识与获取作业2

授课教师	:	乔秀全
姓　　名	:	陈韵涵/黎昱彤
学　　号	:	2022211388/2022211414
班　　级	:	2022211311/2022211312
日　　期	:	2025/5/22

基本要求：自己动手设计实现一个信息检索系统，中、英文皆可，数据源可以自选，数据通过开源的网络爬虫获取，规模不低于100篇文档，进行本地存储。中文可以分词（可用开源代码），也可以不分词，直接使用字作为基本单元。英文可以直接通过空格分隔。构建基本的倒排索引文件。实现基本的向量空间检索模型的匹配算法。用户查询输入可以是自然语言字串，查询结果输出按相关度从大到小排序，列出相关度、题目、主要匹配内容、URL、日期等信息。最好能对检索结果的准确率进行人工评价。界面不做强制要求，可以是命令行，也可以是可操作的界面。提交作业报告、源代码和演示视频。__

扩展要求：鼓励有兴趣和有能力的同学积极尝试多媒体信息检索以及优化各模块算法。自主开展相关文献调研与分析，完成算法评估、优化、论证创新点的过程。

评分标准如下（按照100分计算）：

1、 完成基本的信息检索功能且有对环境和社会可持续发展影响的考虑，系统能够正常运行，并提交源代码和实验报告：60分；

2、 完成要求的信息检索功能且有对环境和社会可持续发展影响的考虑，系统能够正常运行，并按时提交源代码和实验报告：61~70分；

3、 在2的基础上，且实验报告撰写认真、思路清晰、表达准确：71~80分；

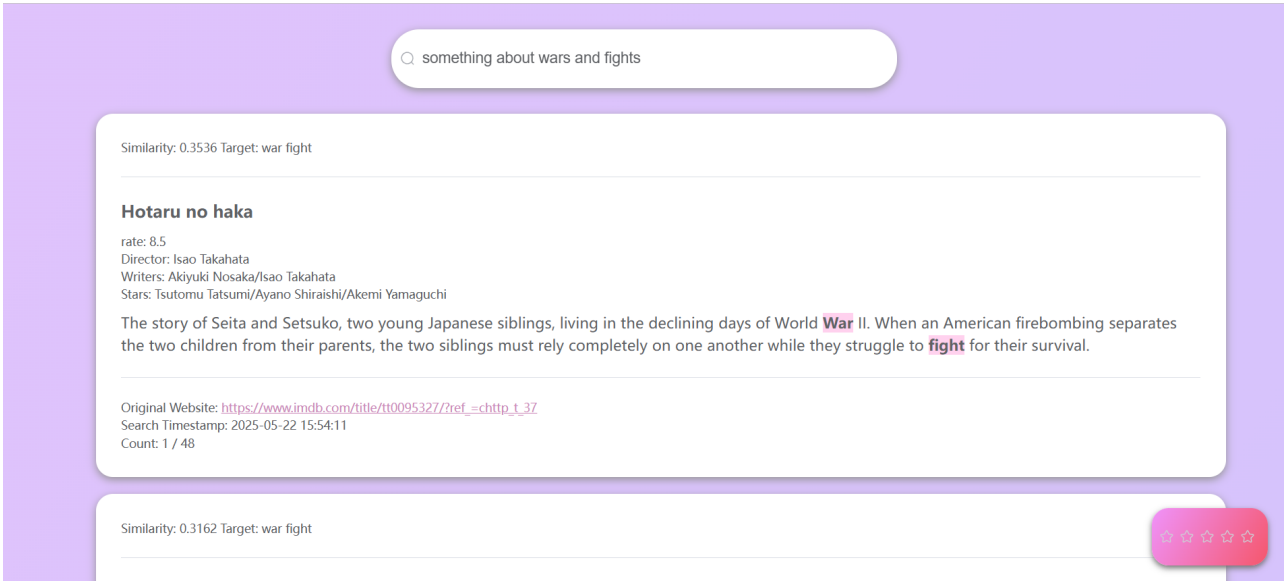
4、 在3的基础上，支持检索结果准确率人工评价：81~90分；

5、 在4的基础上，融入了自己的创新性思考、优化算法或对多媒体信息检索进行了尝试：91-100分。

1 使用说明

基本界面：

- 在搜索框中输入要搜索的文本，系统返回匹配的所有文档并高亮搜索中匹配到的关键词
- 文档内容包括原本的所有内容、搜索时间戳、匹配关键词、匹配相似度、文档计数



用户评分：

- 可以滑动右下角的卡片，点击后提交评分

2 数据准备

2.1 数据来源及类型

本系统的数据来源于 IMDB Top 250 Movies 榜单（IMDB Top 250 Movies），获取了共250篇文档，内容包含电影名称、电影评分、电影导演、电影编剧、电影明星、电影简介以及首页链接，以“序号.txt”的形式将文件存储在了爬虫项目的`source`目录下。

数据结构在爬虫项目的`IMDB\items.py`中定义如下：

```
class ImdbItem(scrapy.Item):
    title = scrapy.Field() # 电影名
    rate = scrapy.Field() # 评分
    summary = scrapy.Field() # 电影简介
    director = scrapy.Field() # 导演
    writers = scrapy.Field() # 编剧
    stars = scrapy.Field() # 明星
    url = scrapy.Field() # 详情页链接
```

2.2 爬虫程序要点

2.2.1 数据爬取

采用`xpath`的形式，获取到需要的数据，在`IMDB\spiders\imdbSpider.py`文件中体现。

主要流程为爬取首页的榜单，获取榜单中列表里的详情页url，进入详情页爬取数据，最后将数据构建成`ImdbItem`后交给`IMDB\pipelines.py`进行进一步处理。

主要难点：在详情页爬取时，电影简介的dom是动态加载出来的，也就是当用户将页面滚动到`Storyline`标题时，前端才向后端发送请求并动态构建出`div[data-testid="storyline-plot-summary"]`这个dom。因此需要在`SeleniumRequest`中添加script参数，手动将页面滚动后等待该dom加载出来再用xpath进行元素定位进而爬取数据。

```
def parse(self, response):
        li_list = response.xpath('//*
[@id="__next"]/main/div/div[3]/section/div/div[2]/div/ul/li')
        self.logger.info("Found %d movie rows", len(li_list))
        for item in li_list:
            page_url = item.xpath('./div/div/div/div/div[2]/div[1]/a/@href').get()
            detail_url = response.urljoin(page_url)
            yield SeleniumRequest(
                url=detail_url,
                callback=self.parse_movie,
                wait_time=3,
                wait_until=EC.presence_of_element_located((
                    By.CSS_SELECTOR,
```

```
                    'div[data-testid="storyline-header"]'
                )),
                cookies=self.cookies,
                meta={'url': detail_url},
                script="""
                    // 先滚到 Storyline 标题
                    document.querySelector('div[data-testid="storyline-header"]')
                            .scrollIntoView({behavior:'instant',block:'center'});
                    // 再等待剧情摘要出现在 DOM
                    return new Promise(resolve => {
                    const check = () => {
                        if (document.querySelector('div[data-testid="storyline-
plot-summary"]')) {
                            resolve();
                        } else {
                            setTimeout(check, 100);
                        }
                    };
                        check();
                    });
                """
            )
```

2.2.2 数据处理

用如下代码将爬取到的数据`item`转换为文本的形式存储在每一个序号命名的文件中：

```
class ImdbPipeline:
    i = 1
    def process_item(self, item, spider):
        dict_item = dict(item)
        with open('source/' + str(self.i) + '.txt', 'w', encoding='utf-8') as
file:
            file.write(dict_item['title'] + '\n')
            file.write('rate: ' + dict_item['rate'] + '\n')
            file.write(dict_item['director'] + '\n')
            file.write(dict_item['writers'] + '\n')
            file.write(dict_item['stars'] + '\n')
            file.write(dict_item['summary'] + '\n')
            file.write('url: ' + dict_item['url'] + '\n')

        self.i += 1

        return item
```

2.2.3 应对反爬的措施

- 使用"`SeleniumRequest`+无头浏览器"的方法进行数据爬取
  - 在`IMDB\settings.py`中设置无头模式和`SeleniumRequest`要用的中间件，还有默认请求头：

```
# 设置无头模式
SELENIUM_DRIVER_NAME = 'firefox'
SELENIUM_DRIVER_EXECUTABLE_PATH = 'C:/Users/cyh/venv/Scripts/geckodriver.exe'
SELENIUM_BROWSER_EXECUTABLE_PATH = None
```

```
SELENIUM_COMMAND_EXECUTOR = None
SELENIUM_DRIVER_ARGUMENTS = ['--headless']
# 设置中间件
DOWNLOADER_MIDDLEWARES = {
   "IMDB.middlewares.SeleniumMiddleware": 543,
}
# 设置请求头
DEFAULT_REQUEST_HEADERS = {
   'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.36 Edg/130.0.0.0',
   'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
   'Accept-Encoding': 'gzip, deflate, br, zstd',
   'Accept-Language': 'zh-CN,zh;q=0.9,en-US;q=0.8,en;q=0.7,en-GB;q=0.6,de;q=0.5,zh-TW;q=0.4',
}
# 其他设置
# Obey robots.txt rules
ROBOTSTXT_OBEY = False
# Configure a delay for requests for the same website (default: 0)
# See https://docs.scrapy.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs
DOWNLOAD_DELAY = 3
RANDOMIZE_DOWNLOAD_DELAY = True  # 随机化间隔时间
# The download delay setting will honor only one of:
CONCURRENT_REQUESTS_PER_DOMAIN = 16
#CONCURRENT_REQUESTS_PER_IP = 16
```

- 每次请求携带cookie：
  - 在`IMDB\spiders\imdbSpider.py`中设置cookie，并规定每次请求携带：

```
class ImdbspiderSpider(scrapy.Spider):
    name = "imdbSpider"
    allowed_domains = ["www.imdb.com"]
    cookies = {
        # 具体的cookie
    }

    # 每次请求携带cookie，其他请求也一致
    def start_requests(self):
        yield SeleniumRequest(url, callback=self.parse, cookies=self.cookies)
```

在完成以上配置后成功在网站爬取到250篇文档并进行存储。

3 预处理文本信息

具体代码在`sys\preprocess.py`中。

3.1 去除无用信息

只保留字母、字母全部小写。

```
lines = f.readlines()
line = lines[0] + lines[2] + lines[3] + lines[4] + lines[5]
line = re.sub('[^a-zA-Z]', ' ', line)              # 只保留字母
line = line.lower()                                # 全部小写
```

3.2 设置停用词及还原词型

- 停用词：去除 “the, is, at…” 等常见词， 本系统采用的是 `stop_words` 库 中提供的停用词
- 还原词型： 把单词还原到词典中的基本形（如 “running” → “run”）， 本系统采用的是 `nltk` 库中的 `WordNetLemmatizer` 实现的
- `clean_corpus`：用于构建向量空间模型
- `raw_corpus`：用于建立倒排序索引表

```
# 停用词
stop_words = set(get_stop_words('en'))
...
raw_text = line.split()
clean_line = [lem.lemmatize(word) for word in raw_text if not word in stop_words]
clean_corpus.append(clean_line)
raw_text = [lem.lemmatize(word) for word in raw_text]
clean_line = ' '.join(clean_line)
raw_corpus.append(raw_text)
corpus.append(clean_line)
```

3.3 建立倒排索引

- 正向索引:
  - 当用户发起查询时（假设查询为一个关键词），搜索引擎会扫描索引库中的所有文档，找出所有包含关键词的文档，这样依次从文档中去查找是否含有关键词的方法叫做正向索引。互联网上存在的网页（或称文档）不计其数，这样遍历的索引结构效率低下，无法满足用户需求。
- 倒排索引:
  - 为了增加效率，搜索引擎会把正向索引变为反向索引（倒排索引）即把“文档→单词”的形式变为“单词→文档”的形式。倒排索引具体机构如下: 单词1→文档1的ID；文档2的ID；文档3的ID…

3.3.1 TF–IDF 算法调研

主要目的：提供一个评判标准，来判断具体采用哪些词语来作为倒排索引表的关键词。

一个词语在语料库中的重要性如下:

$$TF-IDF_{i,j} = TF_{i,j} \cdot IDF_i$$

其中$TF_{i,j}$称为i这个词在第j篇文章中的词频。即:

$$TF_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

其中$n_{i,j}$是i这个词在第j篇文章出现的次数，$\sum_k n_{k,j}$是第j篇文章中的总词数，而IDF_i指的是：

$$IDF_i = \frac{|D|}{1 + |\{j : t_i \in d_j\}|}$$

其中$|D|$指的是语料库中文档的总量，分母指的是出现该词语的文档数量+1（+1是为了防止没有任何文档包含该词）。

该标准合理的原因是：如果有一个词，他只在第i篇文档中的出现率很高，但是同时又不经常在其他文档中出现，那么它对于第i篇文章来说就是很重要的一个特征。这种情况下该词的对于第i篇文章的TF值和IDF值都很大。其他情况，比如：

1. 一个词尽管出现频繁，但是他在所有文档里都出现频繁，那么它并不是那么重要（特征不明显，比如the, a, we等）
2. 一个词在所有文档中都不怎么出现（生僻词）。

都会导致TF-IDF值变低。

在本系统中评价一个词的重要程度的标准是：

$$importance_i = \sum_k TF - IDF_{i,k}$$

即一个词对于语料库中所有文档的TF-IDF值之和

3.3.2 TF–IDF 关键词提取

本系统采用`TF-IDF`算法，用以评判具体需要采用哪些词语来作为倒排序索引表的关键词，具体使用了python中`sklearn.feature_extraction.text`这个库来实现：

```
vectorizer = TfidfVectorizer()          # 初始化向量器
X = vectorizer.fit_transform(corpus)    # 建立词汇表

data = {
  'word': vectorizer.get_feature_names_out(),     # 词汇表中的所有词
  'tfidf': X.toarray().sum(axis=0).tolist()       # 对每一列（每个词）求和，得到该词在所有文档中的总 TF-IDF 值
}
df = pd.DataFrame(data)
df.sort_values(by="tfidf", ascending=False, inplace=True) # 降序排序
key_words = df.head(500)['word'].to_list()                # 选取前500词
with open('json/key_words.json', 'w') as f:
    f.write(json.dumps(key_words))
```

大致思路：

- `TfidfVectorizer`：自动做词袋模型、计算TF–IDF 。
- 把矩阵 `X` 转数组后，按列求和，得到每个词在所有文档中的总TF–IDF分数。
- 按分数降序，取前 500 个词 。
- 最后将关键词存储到`json\key_words.json`文件中。

3.4 建立索引表

索引表被存在`json/reverse_index.json`中

```python
# 建立倒排序索引表
reverse_index = {}
for i in range(1, 251):
    for j in range(len(raw_corpus[i - 1])):
        word = raw_corpus[i - 1][j]
        if word in key_words:
            if not reverse_index.get(word):
                reverse_index[word] = {}
            word_index = reverse_index[word]
            if not word_index.get(i):
                word_index[i] = []
            word_index[i].append(j) # word为第i篇文章的第j个词
            reverse_index[word] = word_index
with open('json/reverse_index.json', 'w') as f:
    f.write(json.dumps(reverse_index))
```

大致思路：

- 依次获取`i.txt`的raw_corpus中的单词，对于在`key_words`里面的单词：
  - 为其构建一个键值对：key为该文章名，value为数组，具体为该词在`raw_corpus[i]`中的索引

索引表结构如下：

```
{
  "word": {
    "1": [
      1, 2
    ],
    "2": [
      1, 2
    ]
  }
}
```

表示单词“word”出现在1.txt的第1、2索引处，出现在2.txt的1、2索引处。

3.5 文本向量化

文本向量化阶段就是要为每一个文本创建一个向量来表示它。根据500个关键词，第i个关键词存在于这篇文本中， 则该文本对应的向量的第i维置1，否则置0，并将该文本向量空间保存到`json\text_vector.json`中。

结果 `text_vector` 是一个 `1000 × 500` 的二值矩阵：

- 每行代表一篇文档
- 每列代表一个全局关键词
- 值为 1/0 表示该关键词是否出现在该文档中

```
# 关键词构建向量空间模型
text_vector = []
for i in range(250):
    text_vector.append([])
    for j in range(500):
        if key_words[j] in clean_corpus[i]:
            text_vector[i].append(1)
        else:
            text_vector[i].append(0)
with open('json/text_vector.json', 'w') as f:
    f.write(json.dumps(text_vector))
```

4 用户查询

具体代码在`sys\main.py`中。

4.1 基本功能

4.1.1 查询处理

处理用户查询并返回相关结果

- 文本预处理
  - 和`process.py`相同去除无用信息
  - 分词，去除停用词

```
line = re.sub("[^a-zA-Z]", " ", message)
line = line.lower()
words = line.split()
words = [lem.lemmatize(w) for w in words if not w in stop_words]
```

- 拼写纠正
  - 调用`correct_spelling(words)`
- 构建查询向量
  - 与文档向量维度相同的500维二元向量
  - 出现为1，否则为0

```
query_vec = []
for i in range(500):
    if key_words[i] in words:
        query_vec.append(1)
    else:
        query_vec.append(0)
```

- 执行检索
  - 处理每个出现在`key_words`中的查询词
  - 通过倒排索引找到包含该词的文档
  - 首次处理计算余弦相似度，记录匹配的关键词

```
  for w in words:
    if w in key_words:
        if w in reverse_index:
            for id in reverse_index[w]:
                doc_id = int(id)
                if doc_id not in ret_info:
                    # 计算相似度
                    doc_vec = text_vector[doc_id - 1]
                    query_np = np.array(query_vec)
                    doc_np = np.array(doc_vec)
                    query_magn = np.linalg.norm(query_np)
                    doc_magn = np.linalg.norm(doc_np)
                    if query_magn > 0 and doc_magn > 0:
                        dot_product = np.dot(query_np, doc_np)
                        cos_sim = dot_product / (query_magn * doc_magn)
                    else:
                        cos_sim = 0
                    ret_info[doc_id] = {
                        'sim': round(cos_sim, 4),
                        'match': ""
                    }
                    sort_sim[doc_id] = cos_sim
                if w not in ret_info[doc_id]['match']:
                    ret_info[doc_id]['match'] += " "+ w
```

- 按相似度降序排序

```
sort_sim = list(sort_sim.items())
sort_sim.sort(key=lambda x: x[1], reverse=True)
```

- 构建结果向量表并返回

```
ret_list = []
for doc_id in sort_sim:
    result = ret_info[doc_id[0]]
    result.update(get_info(doc_id[0]))
    ret_list.append(result)
```

4.2 扩展功能

4.2.1 拼写纠错

这是一个附加功能，对用户输入进行纠错，后续进行模糊匹配

```
def correct_spelling(words):
    corrected = []
    correction_info = {}
    for w in words:
        if w not in key_words and len(w) > 3:
            if w not in spell:
                correct_w = spell.correction(w)
                # 修正后在关键词列表中才采用
                if correct_w != w and correct_w in key_words:
                    corrected.append(correct_w)
                    correction_info[w] = correct_w
```

```
                else:
                    corrected.append(w)
            else:
                corrected.append(w)
        else:
            corrected.append(w)
    return corrected, correction_info
```

- 只对不在关键词列表且长度大于3的单词进行拼写检查
- 修正后在关键词列表中才采用

4.2.2 用户评分

在每一次用户进行搜索之后，可以在页面的右侧滑动卡片对本次搜索进行评分。评分结果将会发送到后端进行保存，具体文件位于项目的`rate.txt`，方便维护管理人员分析检索算法或数据源的不足。

4.3 API

搜索

```
@app.route('/api/search', methods=['GET'])
def search():
    query = request.args.get('q', '')
    if not query.strip():
        return jsonify({"error": "Query cannot be empty"}), 400
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    results, corrections = handle_query(query)
    return jsonify({
        "total": len(results),
        "timestamp": timestamp,
        "results": results,
        "corrections": corrections,
        "has_corrections": len(corrections) > 0
    })
```

评分

```
@app.route('/api/rate', methods=['POST'])
def save_rate():
    data = request.json
    if not data or 'query' not in data or 'rate' not in data:
        return jsonify({"error": "缺少必要的字段"}), 400
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    rate_file.write(f"查询: {data['query']}, 评分: {data['rate']}, 时间:
{timestamp}\n")
    rate_file.flush()
    return jsonify({"success": True, "message": "评价已记录"})
```