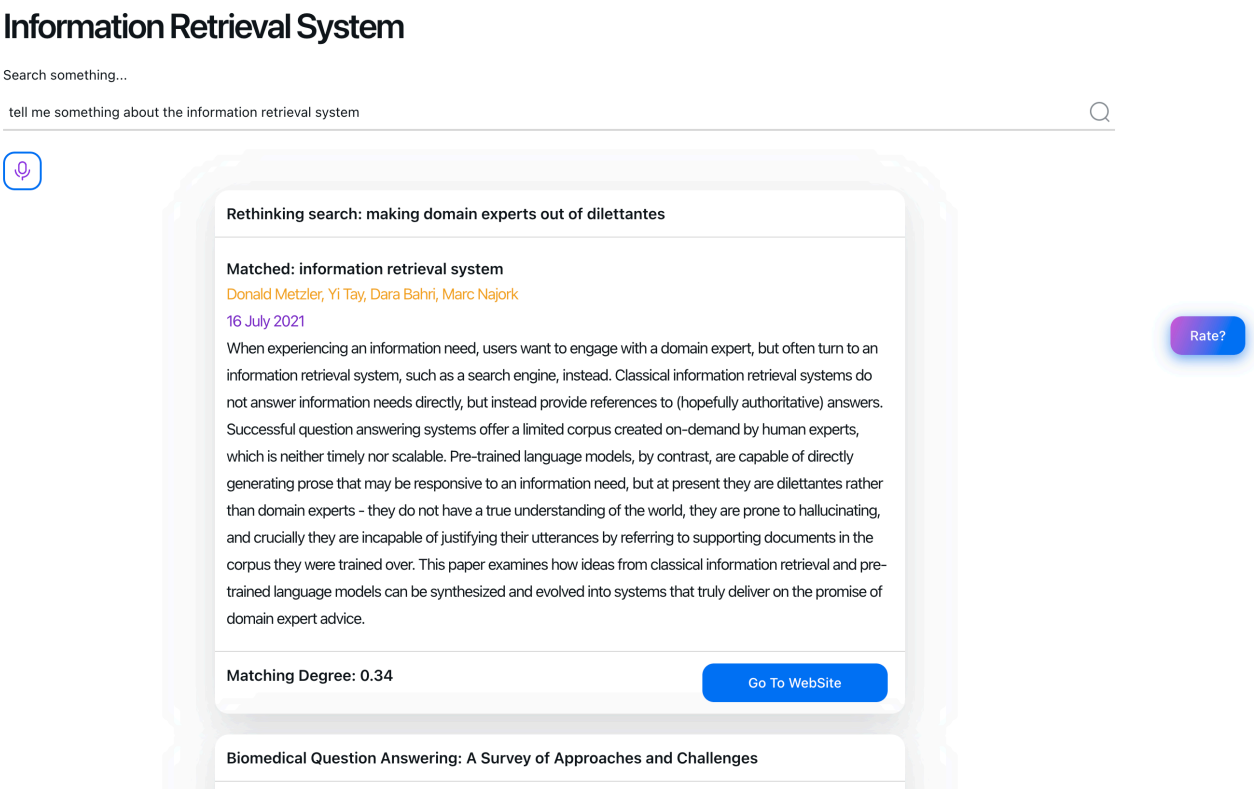
信息检索系统

姓名：何家豪

学号：2020211435

概览

本系统采用**倒排索引**的方式为1000篇源数据建立了索引（使用**TF-IDF算法**筛选关键词），并实现了基本的**向量空间检索模型**的匹配算法。在此基础之上，系统同时支持**用户主观评价检索结果**和**语音多媒体检索方式**，并设计了一个简约美观的可操作页面。该系统的主要使用场景截图如下：



准备工作

本系统的源数据是来自于**ACMDL**（Association for Computing Machinery Digital Library）的**1000篇关于人工智能领域的文章的摘要**。采用Scrapy爬虫实现爬取数据，实现过程的关键步骤如下：

定义数据结构

对于一篇摘要，本系统采集了它的如下信息：

- 标题
- 作者
- 发表日期
- 收录刊物
- 摘要原文
- 详情链接

在`ACMDL/ACMDL/items.py`中定义如下：

```
class AcmdlItem(scrapy.Item):
    title = scrapy.Field()
    date = scrapy.Field()
    author = scrapy.Field()
    pub = scrapy.Field()
    abstract = scrapy.Field()
    url = scrapy.Field()
```

爬虫程序要点

爬虫程序中的大部分流程比较简单，包括从网页上爬取html数据，从中解析出需要用的字段并保存到文件，在此不再详述，具体可见`ACMDL/ACMDL/spiders/spider.py`和`ACMDL/ACMDL/pipelines.py`。

在经过一开始的几次尝试之后，我发现ACMDL的官方网站的反爬虫措施设置的比较完善，仅仅采用传统的爬虫方式IP地址在几次爬取之后就会被封禁，针对这一点，我做了以下几点优化：

1. 在`settings.py`中设置请求 headers伪装：

   ```
   DEFAULT_REQUEST_HEADERS = {
      "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
      "Accept-Language": "en",
      "User-Agent": "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/55.0.2883.87 Safari/537.36",
      "Connection": "close"
   }
   ```

2. 在`settings.py`中设置下载延迟为2秒

   ```
   DOWNLOAD_DELAY = 2
   ```

3. 在`middlewares.py`中设置代理ip地址

   ```
   def process_request(self, request, spider):
       request.meta['Proxy'] = "http://"+"58.246.58.150:9002"
       return None
   ```

在设置完以上特殊设置之后，我成功的从ACMDL官方网站上爬取了近两年来按照被引量排序的前1000篇与人工智能相关的论文摘要，保存在`papers`目录下。

预处理文本信息

在构建索引表和文本向量化之前要先对文本进行一些预处理，本部分的内容全部在`preprocess.py`文件中。

初步处理

在进行文本分析之前要先去除其中的无用信息，比如标点符号，html标签等。然后将所有字母转换为小写方便后续处理。如下所示：

```
# 去除标点符号
text = re.sub("[^a-zA-Z]", " ", text)
# 转换成小写
text = text.lower()
# 去除标签
text = re.sub("&lt;/?.*?&gt", " &lt;&gt; ", text)
# 去除特殊符号和数字
text = re.sub("(\\d|\\W)+", " ", text)
```

设置停用词

在分析文本特征时，人们通常不希望将一些无意义的介词、连词、冠词纳入考虑，比如the, and, with等。所以在分析文本时要将这些词屏蔽掉，以便在后续处理时不将其列入考虑范围。本系统采用的是python中`stop_words`库中提供的停用词，并根据实际情况自行补充了一些停用词，如下所示：

```
# 停用词
stop_words = set(get_stop_words('en'))
new_words = ["using", "show", "result", "large", "also",
             "iv", "one", "two", "new", "previously", "shown", "based"]
stop_words = stop_words.union(new_words)
```

lemmatize处理

在分析文本时，我们希望将一个单词的不同词性和时态理解为一个单词，比如learn, learning, learned, learnt等。当出现一个词的不同形式时，要将其作为一个词看待。这个过程就叫做lemmatize。本系统采用的是`nltk`库中的`WordNetLemmatizer`实现的，如下所示：

```
# 字符串转为List
raw_text = text.split()
text = [lem.lemmatize(word) for word in raw_text if not word in stop_words]
```

最后再将分割开的词拼接回去统一管理在`corpus`中：

```
text = " ".join(text)
corpus.append(text)
```

建立倒排索引

选取关键词

在建立倒排索引表时，将1000篇文本中出现过的词全部作为关键词显然不合适，原因有二：

1. 词语太多，占用空间大
2. 有些词语不够**关键**。即使经过了停用词的初筛之后，也并不是所有词语都包含足够关键的语义信息。

所以需要一个评判标准来决定具体采用哪些词语来作为倒排索引表的关键词。本系统采用的是**TF-IDF算法**。一个词语在一个语料库中的一个文件的重要性如下：

$$TF-IDF_{i,j}=TF_{i,j}\cdot IDF_i \tag{1}$$

其中$TF_{i,j}$称为i这个词在第j篇文章中的**词频**。即：

$$TF_{i,j}=\frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

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：

$$IDF_i=\frac{|D|}{1+|\{j:t_i\in d_j\}|} \tag{3}$$

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：

$$importance_i=\sum_k TF-IDF_{i,k} \tag{4}$$

即一个词对于语料库中所有文档的TF-IDF值之和。本系统选用了TF-IDF值前500的词作为本语料库的关键词，并使用它们建立倒排索引表和文本向量，并将他们保存到了`key_words.json`文件中，如下所示：

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)
data = {'word': vectorizer.get_feature_names_out(),
        'tfidf': X.toarray().sum(axis=0).tolist()}
df = pd.DataFrame(data)
df.sort_values(by="tfidf", ascending=False, inplace=True)
key_words = df.head(500)['word'].to_list()
with open('key_words.json', 'x') as f:
    f.write(json.dumps(key_words))
```

建立倒排索引表

倒排索引表的结构示例如下：

```
{
  'word': {
    "778":[
      263, 502, ...
    ],
    "899":[
      3, 7...
    ],
  },
  ...
}
```

即每一个单词对应一个json对象，该对象以这个单词出现过的每篇文章编号为键，出现位置构成的列表为值。

构建过程的算法比较简单，最后的倒排索引表保存在`rev_index.json`文件中：

```
rev_index = {}
for i in range(1000):  # 对于每一篇文章
    for j in range(len(raw_corpus[i])):  # 中的每一个词
        word = raw_corpus[i][j]
        if word in key_words:  # 如果是关键词
            if not rev_index.get(word):
                rev_index[word] = {}
            word_index = rev_index[word]
            if not word_index.get(i):
                word_index[i] = []
            word_index[i].append(j) # 在其对应文件的索引中添加该出现位置
            rev_index[word] = word_index

with open("rev_index.json", "x") as f:
    f.write(json.dumps(rev_index, indent=4))
```

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 `text_vector.json` 中。

```
text_vector = []
for i in range(1000):  # 对于每一篇文章
    text_vector.append([]) # 创建他的向量
    for j in range(500):  # 对于每一个关键词
        if key_words[j] in clean_corpus[i]:  # 若本篇文章包含该关键词
            text_vector[i].append(1)  # 置1
        else:
            text_vector[i].append(0)  # 否则置0
with open('text_vector.json', 'x') as f:
    f.write(json.dumps(text_vector))
```

搜索匹配算法

在接收到用户输入的字符串后，系统会对其采用与预处理语料库中的信息同样的流程（停用词，lemmatize等）处理它，生成它对应的文本向量，然后在倒排索引表中查询出现在查询字符串中的关键词都出现在了哪些文章中，然后计算查询文本向量与这些文章的文本向量的余弦值，并以此作为评判标准对搜索结果进行排序并返回。具体处理步骤如下：

1. 生成查询字符串的文本向量

```
# 去除标点符号
text = re.sub("[^a-zA-Z]", " ", message)
# 转换成小写
text = text.lower()
# 去除标签
text = re.sub("&lt;/?.*?&gt", " &lt;&gt; ", text)
# 去除特殊符号和数字
text = re.sub("(\\d|\\W)+", " ", text)
# 字符串转为List
text = text.split()

# 停用词和lemmatize
text = [lem.lemmatize(word) for word in text if not word in stop_words]
vec = []
ret_list = []
for i in range(500):  # 对于所有的关键词
  if key_words[i] in text:  # 若第i个词出现在查询串中
    vec.append(1)  # 第i位置1
  else:
    vec.append(0)  # 否则第i位置0
```

2. 利用倒排索引获取相关文章，并计算与相关文本向量的余弦值

```
for words in text:  # 对于文本过滤后的每一个词
  if words in key_words:  # 如果这个词在关键词中
    for num in index[words]:  # 找到出现过这个词的文章
```

3. 计算文本向量之间的余弦值

```
i = eval(num)
if ret_info.get(i) == None:  # 如果返回值中还没有这篇文章
    md = np.dot(np.array(vec), np.array(text_vec[i])) /
      (np.linalg.norm(vec) * np.linalg.norm(text_vec[i]))  # 计算这篇文章的文本向量和检索字符串
的文本向量之间的余弦值
    ret_info[i] = {  # 把计算出的匹配度和匹配词初始化
      'md': round(md, 2),
      'match': ""
    }
    sort_md[i] = md  # 在排序索引数组中加入这篇文章的匹配度
if words not in ret_info[i]['match']:  # 如果尚未收录本词
  ret_info[i]['match'] += ' ' + words  # 在匹配词中加入该词
```

4. 将结果按照匹配度排序，补全细节（日期，作者，摘要等）并返回

```
sort_md = list(sort_md.items())  # 字典转为元组
sort_md = sorted(sort_md, key=lambda x: x[1], reverse=True)  # 排序
for t in sort_md:
  ret_obj = ret_info[t[0]]
  ret_obj.update(get_info(t[0]))  # 补全相关信息
  ret_list.append(ret_obj)  # 加入返回列表
  return ret_list
```

扩展功能

人工评价准确率

在每一次用户进行搜索之后，可以在页面的右侧点击“Rate”按钮对本次搜索进行评分。评分结果将会发送到后端进行保存，方便维护管理人员分析检索算法或数据源的不足。

多媒体信息检索

用户可以通过点击搜索框下方的语音按钮进行语音输入搜索。用户的语音输入会实时出现在搜索框中，当用户的一句话说完或再次点击语音按钮后，系统会开始搜索。

总结

本次实验首先让我复习了上学期Python课上所学的爬虫知识，在解决网站反爬虫的过程中，学习了许多绕过检测的手段。

本次实验最主要的是让我复习了课上讲过的信息检索相关知识（如：倒排索引，向量模型等），并且在自己探索的过程中学习到了很多其他的知识。在一开始构思该如何建立倒排索引的时候，并没有考虑到还有停用词和lemmatize方面的问题，但是在实践过程中发现了仅仅做最朴素的倒排索引效果和性能都不好，所以自己探索优化了一些细节，这个过程不但拓展了我在信息检索方面的知识，同时增强了我的学习与实践能力。

在开发用户评分功能和语音识别功能的时候，我也在网上查询了许多资料去探索实现方式。一开始我准备采用百度语音识别的API进行语音检索，但是后来发现识别效果不好，而且无法做到实时显示（用户边说话，输入框边出现字）。后来再进一步了解之后选择了Speechly框架的API，效果良好。

同时，虽然制作用户界面并非本课程的重点，但是我也出于个人兴趣，通过这次作业制作了一个简洁美观的用户界面，锻炼了自己的前端开发能力。

总体来说，本次实验虽然花费了超出我预期的时间去完成，但是也带给了我不菲的收获，带给了我以后解决类似问题的信心，对于我来说是一次完美的实践经历。