

电类工程导论 C 大作业: News Crucible

谢熠辰	詹欣宇	岳野
F1703014	F1703014	F1703014
517030910355	517030910358	517030910357

目录

1 基本介绍	3
1.1 主要功能	3
1.2 团队分工	3
2 数据获取	3
2.1 爬虫	3
3 文字搜索	4
3.1 文字索引	4
3.1.1 索引创建	7
3.2 相似文本推荐	10
3.2.1 TF-IDF 算法	10
3.2.2 具体实现	11
4 图片搜索	13
4.1 图片搜图片	13
4.1.1 感知哈希算法	14
4.1.2 SIFT 特征点匹配	18
4.2 图片搜文字	24
4.2.1 imageAI	24
4.2.2 ResNet-50	25
4.2.3 具体实现	26
4.2.4 与文字搜索的对接	26

5 Web 前端	28
5.1 项目运行环境	28
5.2 Web 框架: tornado	28
5.2.1 框架基本设置	28
5.2.2 路由设置	28
5.2.3 响应	29
5.2.4 tornado template	29
5.2.5 启动服务器	30
5.3 CSS/HTML 框架: Bootstrap 4	30
6 性能优化	31
7 结果展示	32
7.1 文字搜文字	32
7.2 文字搜图片	33
7.3 相关新闻推荐	34
7.4 图片搜图片	35
7.5 图片搜文字	36

1 基本介绍

1.1 主要功能

本次项目我们主要实现了新闻页面的多媒体检索工作，在数据抓取方面可以抓取各大新闻网站的页面内容，但由于时间因素限制本次仅抓取了澎湃新闻等页面作为展示和参考。

功能细节方面我们实现了以下功能：

- 文字检索新闻
- 图片检索新闻文字
- 图片检索图片
- 相似内容的新闻推荐
- 返回原网页的链接

用户在检索过程中可以实现对图片所对应的新闻内容的检索，也可以获得所检索内容的相似内容。这些功能在新闻检索中具有很高的应用价值。

在每项功能的实现过程中我们都使用了许多优化的或是前沿的方法，譬如“Elastic Search”，“tf-idf”，“SIFT”，“imageai” 等，这些方法共同融合完成了本次项目，而每个方面具体的实现会在下文中详细叙述。

1.2 团队分工

- (1) 谢熠辰：相似文本推荐、图片搜索
- (2) 詹欣宇：环境配置、Web 前端、性能优化
- (3) 岳野：爬虫、文字索引

2 数据获取

2.1 爬虫

在爬虫部分，我们主要针对新闻网站进行了爬取。以澎湃新闻网站为例，分析网页结构找到新闻分布的主要页面，对本页面的新闻链接进行爬取。对于每一个新闻页，主要的新闻内容

都在标签“newscontent”下，提取该标签的内容就可以过滤掉页面中的其他无关内容。我们使用了课上推荐的 BeautifulSoup 方法提取新闻页的主要内容以及对应该新闻页的标题图片并保存在本地。

为了在网站中保留原贴的 url，还创建了一个对应的文档，该文档中保存了每个抓取下来的页面的 url，可以链接回原帖进行查看。在爬虫的优化方面，使用了课程中优化好的多线程方法可以极大的加速页面的爬取效率。

将网页的源代码抓取成功后，为了成功建立倒排索引，还需要提取网页文字内容。对于每一个新闻页面，其格式分布非常规律，主要有以下几个重要标签：新闻标题、来源、日期、时间、新闻内容。此阶段依然使用的是 BeautifulSoup，将提取好的主要内容保存起来作为“newstxt”，以便下一步分的分词与索引。

至此，爬虫部分对于新闻内容的获取基本结束，我们得到了新闻的标题图片，新闻的标题、日期、来源等信息，以及新闻内容。

3 文字搜索

3.1 文字索引

文字索引部分我们抛弃了课程中主要介绍的 pyLucene 转而使用兼容性实用性更高的 ElasticSearch，因为 ES 可以更好地兼容 Python3 并且索引与查询的效率更高，实用功能也更丰富。

Elasticsearch 是一个分布式可扩展的实时搜索和分析引擎，一个建立在全文搜索引擎 Apache LuceneTM 基础上的搜索引擎。当然 Elasticsearch 并不仅仅是 Lucene 那么简单，它不仅包括了全文搜索功能，还可以进行以下工作：

- 分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。
- 实时分析的分布式搜索引擎。
- 可以扩展到上百台服务器，处理 PB 级别的结构化或非结构化数据。

我们知道 Elastic Search 的索引速度很快，比关系型数据库 B-tree 索引快，那么是如何做到的呢？

什么是 B-tree 索引？

数据结构老师教过我们，二叉树查找效率是 $\log N$ ，同时插入新的节点不必移动全部节点，所以用树型结构存储索引，能同时兼顾插入和查询的性能。因此在这个基础上，再结合磁盘的读

取特性(顺序读/随机读), 传统关系型数据库采用了 B-Tree/B+Tree 这样的数据结构:

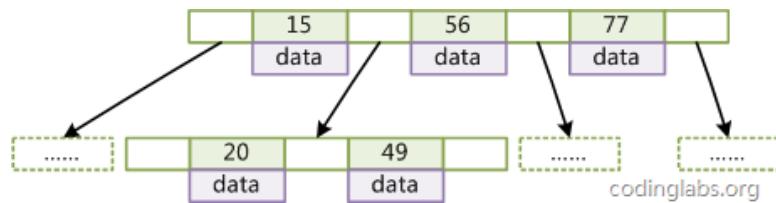


图 1: B-tree

为了提高查询的效率, 减少磁盘寻道次数, 将多个值作为一个数组通过连续区间存放, 一次寻道读取多个数据, 同时也降低树的高度。

什么是倒排索引?



图 2: Inverted-index

假设我们此次有如下几条数据, 简单起见仅展示部分索引信息:

ID	path	date	time	url
1	a.txt	2019-01-08	11:30	http://a.com
2	b.txt	2019-01-07	17:26	http://b.com
3	c.txt	2019-01-08	08:15	http://c.com

ID 是 Elasticsearch 自建的文档 id, 那么 Elasticsearch 建立的索引如下:

path:	
Term	Posting List
a.txt	1
b.txt	2
c.txt	3

date:	
Term	Posting List
2019-01-07	2
2019-01-08	[1,3]

time:	
Term	Posting List
08:15	3
11:30	1
17:26	2

Posting List

Elasticsearch 分别为每个 field 都建立了一个倒排索引，a.txt, http://a.com, 2019-01-08, 11:30 这些叫 term，而 [1,2] 就是 Posting List。Posting list 就是一个 int 的数组，存储了所有符合某个 term 的文档 id。

通过 posting list 这种索引方式似乎可以很快进行查找，比如要找 date = 2019-01-08 的文件，我们知道 id 是 1, 3 的文件。但是，如果这里有上千万的记录呢？如果是想通过 time 来查找呢？

Term Dictionary

Elasticsearch 为了能快速找到某个 term，将所有的 term 排个序，二分法查找 term， $\log N$ 的查找效率，就像通过字典查找一样，这就是 Term Dictionary。现在再看起来，似乎和传统数据库通过 B-Tree 的方式类似啊，为什么说比 B-Tree 的查询快呢？

Term Index

B-Tree 通过减少磁盘寻道次数来提高查询性能，Elasticsearch 也是采用同样的思路，直接通过内存查找 term，不读磁盘，但是如果 term 太多，term dictionary 也会很大，放内存不现实，于是有了 Term Index，就像字典里的索引页一样，A 开头的有哪些 term，分别在哪页，可以理解 term index 是一颗树：

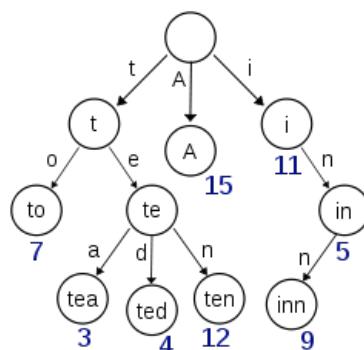


图 3: Term index

这棵树不会包含所有的 term，它包含的是 term 的一些前缀。通过 term index 可以快速地定位到 term dictionary 的某个 offset，然后从这个位置再往后顺序查找。

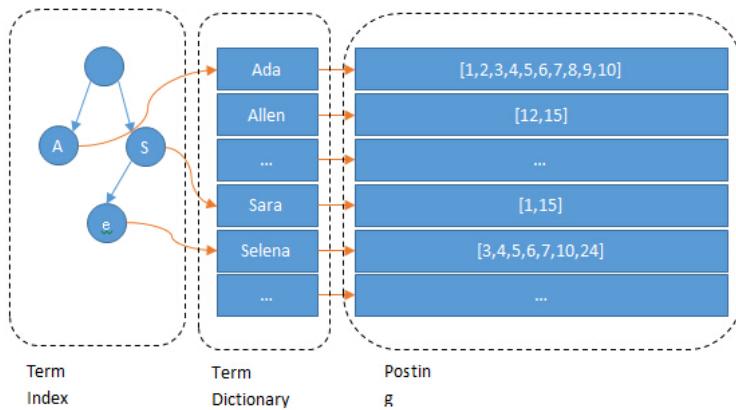


图 4: Index

所以 term index 不需要存下所有的 term，而仅仅是他们的一些前缀与 Term Dictionary 的 block 之间的映射关系，再结合 FST(Finite State Transducers) 的压缩技术，可以使 term index 缓存到内存中。从 term index 查到对应的 term dictionary 的 block 位置之后，再去磁盘上找 term，大大减少了磁盘随机读的次数。

3.1.1 索引创建

针对爬取到的数据创建索引，主要有以下八条：

1. date: 新闻日期
2. time: 新闻时间
3. title: 新闻标题
4. source: 新闻来源
5. path: 新闻内容在本地的储存路径
6. contents: 新闻内容
7. imgpath: 新闻图片的储存路径
8. url: 新闻原贴的地址

其中 3,4,6 在创建索引时需要分词，我们使用 ES 自配的“analysis-ik”分词器进行分词。在保存格式方面使用 ES 的索引格式一般为 text，对于 5,7 等不可分割内容使用“keyword”格式进行完全匹配。创建索引代码展示如下：

elasticsearch 索引的结构:

```
1 # 创建映射
2 _index_mappings = {
3     "mappings": {
4         self.index_type: {
5             "properties": {
6                 "title": {
7                     "type": "text",
8                     "index": True,
9                     "analyzer": "ik_max_word",
10                    "search_analyzer": "ik_max_word"
11                },
12                "contents": {
13                    "type": "text",
14                    "index": True,
15                    "analyzer": "ik_max_word",
16                    "search_analyzer": "ik_max_word"
17                },
18                "date": {
19                    "type": "date",
20                    "index": True
21                },
22                "time": {
23                    "type": "keyword",
24                    "index": True
25                },
26                "source": {
27                    "type": "text",
28                    "index": True,
29                    "analyzer": "ik_max_word",
30                    "search_analyzer": "ik_max_word"
31                },
32                "path": {
33                    "type": "keyword",
34                    "index": True
35                },
36                "imgpath": {
37                    "type": "keyword",
38                    "index": True
39                },
40                "url": {
41                    "type": "text",
42                    "index": True
43                }
44            }
45        }
46    }
47 }
```

在查询阶段，elasticsearch 还支持非常人性化的 highlight 功能，在搜索到的内容中可以自动实现搜索内容的高亮。对于前端的设计起到不小便利。

Index.py 作为函数库，create_index.py 与 Search.py 作为接口与前端进行对接。

3.2 相似文本推荐

新闻网站上常常包含相关新闻推荐板块，我们同样也实现了这一功能。它主要基于 TF-IDF 算法，通过计算内容相关度，推荐一些相似的新闻。

3.2.1 TF-IDF 算法

TF-IDF 是一种加权技术，采用一种统计方法，根据字词在文本中出现的次数和在整个语料中出现的文档频率来计算一个字词在整个语料中的重要程度。如果某个词或短语在一篇文章中出现的频率高，并且在其他文章中很少出现，则认为此词或者短语具有很好的类别区分能力，适合用来分类。

词频（TF）表示某一词条在当前文档中出现的频率，其计算公式如下所示：

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

以上式子中分子是特征词在文本 d_j 中的出现次数，而分母则是在文本 d_j 中所有字词的出现次数之和，计算词频可以避免文本长度对结果的影响。

逆向文件频率（IDF）反映了一个词在所有文档中的出现频率，是对一个词重要性的度量。如果一个词在很多的文本中出现，那么它的 IDF 值应该低。反之如果一个词在比较少的文本中出现，那么它的 IDF 值应该高。它的计算公式如下所示：

$$idf_{i,j} = \log \frac{|D|}{1 + |D_{t_i}|}$$

其中， $|D|$ 表示语料中文本的总数， $|D_{t_i}|$ 表示包含特征词的文本数量。为了避免特征词没有在语料中出现，也就是分母为 0 的极端情况，使用 $1 + |D_{t_i}|$ 作为分母。

最后，特征值的 $tfidf$ 值计算公式即为：

$$tfidf_{i,j} = tf_{i,j} \times idf_{i,j}$$

$tfidf$ 的值越大，表示该特征词对这个文本的重要性越大。

然后，我们根据每篇文本中的每个词的 TF-IDF 值，可以构建每篇文本的 TF-IDF 向量，进而计算两篇文本的余弦相似度，公式如下所示：

$$\cos \theta = \frac{A \cdot B}{|A| \times |B|}$$

其中， A 和 B 分别是表示两个文档的 TF-IDF 向量。余弦值越接近 1，就表明夹角越接近 0 度，也就是两个向量越相似，这就叫“余弦相似性”。

3.2.2 具体实现

为了减少搜索时间，将相似文本推荐的实现分为预处理和搜索两部分。

预处理部分

预处理部分主要是对语料库中的所有文本进行处理，建立词袋模型，然后再进一步建立 TF-IDF 模型，最后将建立好的模型放入 json 格式的文件中，从而在搜索部分可以直接载入，节省搜索时间。其中，分词使用的结巴分词，词袋模型和 TF-IDF 模型的建立使用了 gensim 库。

首先，我们设置了一个停用词表，在分词时忽略掉一些没有意义的词组或短语。

```
1 stop_words = 'stop_words.txt'  
2 stopwords = codecs.open(stop_words, 'r', encoding='utf8').readlines()  
3 stopwords = [w.strip() for w in stopwords]
```

此外，我们还设置了停用词性列表，主要包括：标点符号、连词、助词、副词、介词、时语素、‘的’、数词、方位词、代词。

```
1 stop_flag = ['x', 'c', 'u', 'd', 'p', 't', 'uj', 'm', 'f', 'r']
```

然后，对于每一篇文本，我们可以在读入内容后，使用结巴分词对其进行分词处理，并且去除掉其中的停用词。

```
1 def tokenization(filename):  
2     result = []  
3     with open(filename, 'r') as f:  
4         text = f.read()  
5         words = pseg.cut(text)  
6         for word, flag in words:  
7             if flag not in stop_flag and word not in stopwords:  
8                 result.append(word)  
9     return result
```

然后，我们进一步建立词袋模型。语料中的每一篇文本都是由一个个词构成的，把所有的词都放进一个词袋中，而不考虑它们的语义。这样我们可以构建出一个字典结构，key 为语料中出现的词语，value 为这个词的索引序号。

在这里，我们直接使用 gensim 库中的 corpora.Dictionary() 函数来实现。

```
1 dictionary = corpora.Dictionary(corpus)
```

之后，我们把根据词袋模型，按照每一篇文本中出现的词语的索引序号和出现次数，每一篇文本都变成一个稀疏向量。使用一个列表来表示一篇文本，每个列表中包含若干个元组，其中元组 (i, j) 表示索引序号为 i 的词语在这篇文本中出现了 j 次。

这里，我们借助 gensim 库中的 dictionary.doc2bow(doc) 来实现这一操作。

```
1 doc_vectors = [dictionary.doc2bow(text) for text in corpus]
```

之后，我们已经建立的词袋模型的基础上，按照前一部分原理中所述的公式计算出每篇文本中的每个词的 $tfidf$ 值，从而建立 TF-IDF 模型。这个操作使用 gensim 库中的 models.TfidfModel() 函数实现。

```
1 tfidf = models.TfidfModel(doc_vectors)
2 tfidf_vectors = tfidf[doc_vectors]
```

在预处理部分的最后，我们将词袋模型的词典和计算好的 TF-IDF 模型等内容一并保存到文件中，以便搜索时直接载入。

```
1 dictionary.save('ths_dict.dict')
2 tfidf.save("ths_tfidf.model")
3 with open("corpus.json", "w") as f:
4     json.dump(corpus, f)
```

搜索部分

搜索部分主要是载入之前预处理部分计算好的模型，并且对于给定的文本计算它和其它每篇文本的相似度，返回相似度较高的几篇文本。

首先，需要先载入模型，构建表示每一篇文本的稀疏向量，以进行进一步操作。

```
1 tfidf = models.TfidfModel.load("ths_tfidf.model")
2 dictionary = corpora.Dictionary.load('ths_dict.dict')
3 with open("corpus.json", 'r') as load_f:
4     corpus = json.load(load_f)
5 with open("file_list.json", "r") as load_f:
6     file_list = json.load(load_f)
7
```

```
8 doc_vectors = [dictionary.doc2bow(text) for text in corpus]
9 tfidf = models.TfidfModel(doc_vectors)
10 tfidf_vectors = tfidf[doc_vectors]
```

然后，用与预处理部分相同的方式，将给定的带搜索文本进行分词后，映射到词袋模型的稀疏向量中去。

```
1 query = tokenization(file)
2 query_bow = dictionary.doc2bow(query)
```

这样，我们就可以按照前一部分所述原理，使用 TF-IDF 模型计算 query 文本和其它文本的余弦相似度，这个通过 gensim 库中的 `similarities.MatrixSimilarity()` 函数实现。

```
1 index = similarities.MatrixSimilarity(tfidf_vectors)
2 sims = index[query_bow]
3 res = list(enumerate(sims))
```

最后，将结果按照相似度降序排序后，将除去它本身外，最相似的 10 个结果的文件名返回给前端。

```
1 res = sorted(res, key=lambda elem:elem[1], reverse=True)
2
3 files = []
4 for i in range(1, 11):
5     files.append(file_list[res[i][0]])
```

4 图片搜索

4.1 图片搜图片

图片搜索图片部分主要结合了感知哈希 (pHash) 算法和 SIFT 特征点匹配算法。首先使用感知哈希算法对相似图片进行初步筛选，为了尽可能弥补感知哈希算法对旋转等图片变换的不敏感性，我们使用 SIFT 特征点匹配算法对初步筛选出的图片进行逐一比对，找出最相似的若干张图片。

4.1.1 感知哈希算法

算法原理

感知哈希算法主要包括以下几个步骤：

- **缩小尺寸**

将图片缩小到 32×32 的尺寸，并且不要保持纵横比。这样我们就可以摒弃不同尺寸带来的图片差异，比较任意大小的图片，同时简化后续 DCT 的计算。

- **简化色彩**

将缩小后的 32×32 图片转化成灰度图像，降低维数，简化计算量。

- **计算 DCT**

离散余弦变换（DCT）是与傅里叶变换相关的一种变换，用于对信号和图像进行数据压缩，二维的 DCT 变换可以使用如下方式计算得到：

$$F(u, v) = c(u)c(v) \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} f(i, j) \cos\left[\frac{(i+0.5)\pi}{N}u\right] \cos\left[\frac{(j+0.5)\pi}{N}v\right]$$

$$c(u) = \begin{cases} \sqrt{\frac{1}{N}} & u = 0 \\ \sqrt{\frac{2}{N}} & u \neq 0 \end{cases}$$

其中， $f(i, j)$ 为原始的图像信号， $c(u)$ 可以认为是一个补偿系数，使得 DCT 变换矩阵为正交矩阵， $F(u, v)$ 是 DCT 变换后的系数。

也可以写作矩阵的形式：

$$\begin{aligned} F &= AfA^T \\ A(i, j) &= c(i) \cos\left[\frac{(j+0.5)\pi}{N}i\right] \end{aligned}$$

可以看出，以上的计算主要只讨论了二维图像是方阵的情况，而我们也是将其应用于转换后的 32×32 灰度图像。

- **缩小 DCT**

DCT 的结果是 32×32 大小的矩阵，我们将其缩放为 8×8 的矩阵，以进一步减小计算量。

- **计算平均值**

计算 DCT 矩阵中各元素的平均值。

- **DCT 值与均值比较**

将 8×8 的 DCT 矩阵的每个元素与平均值比较，大于平均值的设为 1，小于平均值的设为 0。这样处理后的结果只能反映图像各点对应的频率与平均值的相对比例，而不能表现真实值。但这样处理可以避免彩色图片调整为灰度等变换带来的影响，更好反映图片整体结构。

- **构造哈希值**

将最终的 DCT 矩阵中的各元素连接成二进制长整型，每四位用一个十六进制字符表示，这样就可以得到每张图片的哈希值，也就是每张图片的图片指纹。

在比较两张图片时，计算得到每张图片的哈希值，然后比较两个哈希值不同位的个数，也就是计算两张图片的**汉明距离** (Hamming distance 两个等长字符串之间的汉明距离是两个字符串对应位置的不同字符的个数)。汉明距离越小，那么两张图片就越相似。

具体实现

为了尽可能节省搜索时间，感知哈希算法的具体实现同样分为预处理和搜索两部分。

预处理部分

预处理部分需要计算图库中每一张图片的哈希值，也就是它的图片指纹，然后把计算结果保存在 json 文件中，以便搜索部分直接使用。

对于每一张图片，首先，将图片大小调整为 32×32 ，这一步直接使用 openCV 库中的 resize() 函数。

```
1 img=cv2.resize(img,(32,32),interpolation=cv2.INTER_CUBIC)
```

然后，对图片进行二维 DCT 变换，可以使用 openCV 库中的 cv2.dct() 函数。并且，截取生成的 32×32 DCT 矩阵转换为 8×8 矩阵。

```
1 h, w = img.shape[:2]
2 vis0 = np.zeros((h,w), np.float32)
3 vis0[:h,:w] = img
4
5 vis1 = cv2.dct(vis0)
6 vis1.resize(8,8)
```

然后，把二维的 DCT 矩阵变成一维，以便之后生成图片指纹。

```
1 img_list=flatten(vis1.tolist())
2
3 def flatten(x):
4     result = []
5     for el in x:
6         if isinstance(el, collections.Iterable) and not isinstance(el, str):
7             result.extend(flatten(el))
8         else:
9             result.append(el)
10    return result
```

计算转换后的一维 DCT 矩阵中元素的平均值，并且将每个元素与平均值比较，大于平均值的赋值为 1，小于平均值的赋值为 0。

```
1 avg = sum(img_list)*1./len(img_list)
2 avg_list = ['0' if i<avg else '1' for i in img_list]
```

然后，将上一步的结果转换为图片指纹。将每四位 01 字符用一个十六进制数来表示，作为哈希函数的结果返回。

```
1 return ''.join(['%x' % int(''.join(avg_list[x:x+4])),2) for x in range(0,8*8,4)])
```

最后，将图库中每张图片的文件名存入 file_list 列表，将每张图片的哈希函数结果存入 hash_list 列表，将这两个列表保存为 json 文件。

```
1 def dump_json(file_list,hash_list):
2     with open("hash_file.json", "w") as f:
3         json.dump(file_list, f)
4     with open("hash_hash.json", "w") as f:
5         json.dump(hash_list, f)
```

搜索部分

搜索部分需要先加载预处理部分保存在 json 文件中的数据，然后用和预处理相同的哈希函数生成给定图片的图片指纹。计算给定图片和图库中其它图片的汉明距离，找出图库中和给定图片最相似的一部分图片，以便之后使用 SIFT 特征点进行逐一比对。

首先，加载之前保存在 json 文件中的每张图片的文件名和每张图片的图片指纹，将其存在两个列表中。

```
1 def load_json():
2     with open("hash_file.json", "r") as f:
3         file_list = json.load(f)
4     with open("hash_hash.json", "r") as f:
5         hash_list = json.load(f)
6     return file_list,hash_list
```

然后，生成待搜索图片的哈希值，也就是它的图片指纹。

```
1 s = pHash(target)
```

然后，逐一比对待搜索图片和图库中的所有图片，计算它们的汉明距离，将比对结果存在列表中。

```
1 imgs = []
2 file_list, hash_list = load_json()
3 for i in range(len(file_list)):
4     img = {}
5     img["file"] = file_list[i]
6     img["hash"] = hash_list[i]
7     img["score"] = hammingDist(hash_list[i],s)
8     imgs.append(img)
```

汉明距离的计算首先需要保证两个字符串的长度相同，然后将图片指纹每位的十六进制数重新转换为四位二进制数（不足四位用 0 补足），计算两个图片指纹字符串对应位置不同的字符数，即为两个图片的汉明距离 $dist$ 。最后，按照 $score = 1 - \frac{dist}{64}$ 计算两张图片的相似度。

```
1 def hammingDist(s1, s2):
2     dist = 0
3     assert len(s1) == len(s2)
4     for ch1, ch2 in zip(s1, s2):
5         b1 = str(bin(int(ch1, 16)))[2:]
6         b1 = b1.zfill(4)
7         b2 = str(bin(int(ch2, 16)))[2:]
8         b2 = b2.zfill(4)
9         dist += sum([x != y for x, y in zip(b1, b2)])
10    out_score = 1 - dist * 1. / (8 * 8)
```

11

```
return out_score
```

然后，将待搜索图片和图库中各图片的比对结果按照相似度降序排序，取前 30%，将这些图片的文件名作为初筛的结果返回，将返回结果转换为集合，以提高之后的查找效率。

```
1 imgs = sorted(imgs, key=lambda elem:elem["score"], reverse=True)
2 res = [elem["file"] for elem in imgs]
3 return set(res[:int(len(res)*0.3)])
```

4.1.2 SIFT 特征点匹配

在使用感知哈希算法对相似图片进行初步筛选后，我们通过图像之间 SIFT 特征点的比对来进一步筛选，最终筛选出图库中和目标图像最相似的图片。

算法原理

SIFT 特征匹配算法可以处理两幅图像之间发生平移、旋转、仿射变换情况下的匹配问题，具有很强的匹配能力。SIFT 特征是图像的局部特征，对平移、旋转、尺度缩放、亮度变化、遮挡和噪声等具有良好的不变性，对视觉变化、仿射变换也保持一定程度的稳定性。

图像金字塔的构建

图像金字塔是一种以多分辨率来解释图像的结构，通过对原始图像进行多尺度像素采样的方式，生成多个不同分辨率的图像。把具有最高级别分辨率的图像放在底部，以金字塔形状排列，往上是一系列像素（尺寸）逐渐降低的图像。

获得图像金字塔一般包括两个步骤：

- 利用低通滤波器平滑图像

- 对平滑图像进行抽样（采样）放大图像使用上采样的方式，即在原有图像像素的基础上在像素点之间采用合适的插值算法插入新的元素。缩小图像采用下采样的方式，对于原有图像，间隔几个像素取样一次，再通过滤波器对新采集到的图像进行平滑。

坐标系的变换

SIFT 描述子之所以具有旋转不变的性质是因为在图像坐标系和物体坐标系之间变换。

SIFT 描述子把以关键点为中心的邻域内的主要梯度方向作为物体坐标系的 X 方向，因为该坐标系是由关键点本身的性质定义的，因此具有旋转不变性。

假设某关键点为 $I(x_0, y_0)$ ，对于它 $m \times m$ 领域内的每个点，计算其梯度方向和梯度强度：

$$m(x, y) = \sqrt{(I(x+1, y) - I(x-1, y))^2 + (I(x, y+1) - I(x, y-1))^2}$$

$$\theta(x, y) = \arctan \frac{I(x, y+1) - I(x, y-1)}{I(x+1, y) - I(x-1, y)}$$

梯度方向为 360 度，平均分成 36 个 bins，每个像素以 $m(x, y)$ 为权值为其所在的 bin 投票。最终权重最大的方向定位该关键点的主方向。

将关键点 16×16 邻域内的每个点的梯度方向由图像坐标系换算到物体坐标系，即

$$\theta'(x, y) = \theta(x, y) - \theta_0$$

其中 θ' 是相对于物体坐标系的梯度方向， θ 是相对图像坐标系的梯度方向， θ_0 是关键点的主方向。

由于两个坐标系中整数点并非相互对应，因此需要使用双线性插值计算对应点的值。

$$\theta(x', y') = \theta(x, y) \cdot dx2 \cdot dy2 + \theta(x+1, y) \cdot dx1 \cdot dy2 + \theta(x, y+1) \cdot dx2 \cdot dy1 + \theta(x+1, y+1) \cdot dx1 \cdot dy1$$

SIFT 描述子的计算

物体坐标系 16×16 的邻域分成 4×4 个块，每个块 4×4 个像素。在每个块内按照求主方向的方式把 360 度分成 8 个 bins，统计梯度方向直方图，最终每个块可生成 8 维的直方图向量，每个关键点可生成 $4 * 4 * 8 = 128$ 维的 SIFT 描述子。

对 128 维 SIFT 描述子 f_0 归一化得到最终的结果：

$$f = f_0 \cdot \frac{1}{|f_0|}$$

两个 SIFT 描述子 f_1 和 f_2 之间的相似度使用欧式距离表示：

$$d(f_1, f_2) = |f_1 - f_2|$$

，欧式距离越小就越相似。

具体实现

为了节省搜索时间，SIFT 算法同样被分为预处理和搜索两个部分。

预处理部分

在预处理部分，对于图库中的每张图片进行角点提取，找到每个角点的主方向，并且生成每幅图像每个角点的 128 维 SIFT 描述子。然后，再将这些保存在 json 文件中，这样搜索时可以之间载入 json 文件。

首先，对每一幅图片(灰度图片)提取角点，直接调用 openCV 中的角点检测函数 `cv2.goodFeaturesToTrack()`，返回图片中的角点列表。

```
1 for i in imglist:  
2     corners.append(cv2.goodFeaturesToTrack(i, maxCorners = corner_number,  
        qualityLevel = 0.01, minDistance = 10, blockSize = 3, k = 0.04)  
        [:,:,0,:,:].astype("int"))
```

然后，按照算法原理部分所述的方式，计算每张图片上各点的梯度方向和梯度强度，在梯度方向上减去一个极小的值以防止计算主方向时出现越界。

```

1 xdiff = img[1:-1, 2:] - img[1:-1, :-2]
2 ydiff = img[2:, 1:-1] - img[:-2, 1:-1]
3 grad = np.zeros([2, img.shape[0], img.shape[1]])
4 grad[0, 1:-1, 1:-1] = np.sqrt(xdiff * xdiff + ydiff * ydiff)
5 grad[1, 1:-1, 1:-1] = np.arctan2(ydiff, xdiff) - 0.0000000000000001

```

接着可以计算各个角点的主方向。

先将梯度方向 2π 平均分成 36 个 bins，根据每个像素点的梯度方向将其划分到不同的 bins 中。

```

1 directs = []
2 unit = np.pi * 2 / number
3 for grad in grads:
4     directs.append(((grad[1,...] + np.pi) / unit).astype('int'))

```

然后根据高斯函数计算出高斯模板，这将反映各个像素点到角点的距离对它确定角点主方向权值的影响，离中心 $(x, y)3\sigma$ 以外的高斯值均接近于 0。所以我们只需要考虑 $(6\sigma+1)*(6\sigma+1)$ 范围内的高斯值：

$$G(x_i, y_i, \sigma) = \frac{1}{2\pi\sigma} \exp\left(-\frac{(x_i - x)^2 + (y_i - y)^2}{2\sigma^2}\right)$$

```

1 def GSmodel(zgm):
2     size = int(round(3 * zgm) * 2 + 1)
3     half = size / 2
4     model = np.zeros((size, size))
5     zgm = zgm * zgm
6     for i in range(size):
7         for j in range(size):
8             model[i,j] = math.exp(-((half - i)*(half - i) + (half
9             - j)*(half - j))/2/zgm)/2/math.pi/zgm
10    return model

```

然后，邻域上的每个像素点以不同的权值为所在的 bins 投票，每个像素点的权值为 $I(x, y) \cdot G(x, y, \sigma)$ ，其中 $I(x, y)$ 为刚才计算出的该点的梯度强度， $G(x, y, \sigma)$ 为高斯函数计算结果，邻域大小为 $(6\sigma + 1) \times (6\sigma + 1)$ 。

```
1 gs = GSmodel(zgm)
```

```

2   r = (gs.shape[0] - 1)//2
3   for i in range(len(grads)):
4       directs.append([])
5       for x0,y0 in corners[i]:
6           wei_grad = grads[i][0, y0-r:y0+r+1, x0-r:x0+r+1] * gs
7           direct = np.bincount(pillar_directs[i][y0-r:y0+r+1, x0-r:x0+r
+1].ravel(), wei_grad.ravel(), minlength = 36)

```

需要注意的是，若某一方向梯度不是最大方向，但却能够达到主方向梯度的 80%，那么这个方向也可以认为是一个主方向，同样需要对它计算 SIFT 描述子。到此，完成了主方向的确定。

```

1 grad_limit = direct.max() * 0.8
2 for j in range(36):
3     if direct[j] >= grad_limit:
4         directs[i].append((x0, y0, math.pi * (j - 18) / 18.0))

```

最后，需要计算每个特征点的 128 维 SIFT 描述子。

先将特征点的 8×8 邻域由图像坐标系转换至物体坐标系，这个可以通过选择矩阵实现：

$$x' = x \cdot \cos \theta - y \cdot \sin \theta$$

$$y' = x \cdot \sin \theta + y \cdot \cos \theta$$

其中， θ 为旋转角，这里就是主方向，注意对于每个主方向都要如此操作。

```

1 for j in range(len(main_directs[i])):
2     pos_in_img[:, :, 0] = pos_in_corner[:, :, 0]*math.cos(main_directs[i][j]
] [2]) - pos_in_corner[:, :, 1]*math.sin(main_directs[i][j][2]) +
main_directs[i][j][0]
3     pos_in_img[:, :, 1] = pos_in_corner[:, :, 0]*math.sin(main_directs[i][j]
] [2]) + pos_in_corner[:, :, 1]*math.cos(main_directs[i][j][2]) +
main_directs[i][j][1]

```

物体坐标系 16×16 的邻域分成 4×4 个块，每个块 4×4 个像素。在每个块内按照求主方向的方式把 360 度分成 8 个 bins，每个像素以梯度大小为权值为其所在的 bin 投票，最终每个块可生成 8 维的直方图向量。然后再将每个块的八维直方图向量连接在一起，生产 128 维的 SIFT 描述子，并且将其归一化。

```

1 tmp = np.round(pos_in_img).astype("int")

```

```

2 grad = grads[i][:,:,1],tmp[:, :, 0]]
3 grad[1] = (grad[1] - main_directs[i][j][2] + np.pi) % (2*np.pi) - np.pi
4 transed = transDirect(grad[1], 8) + appendix
5 vector[j] = uniform(np.bincount(transed.ravel()), grad[0].ravel(), minlength =
128))

```

其中，函数 `transDirect()` 将梯度方向 2π 平均分成 8 个 bins，根据邻域每个像素点的梯度方向将其划分到不同的 bins 中。

上文代码中变量 `appendex` 的值如下所示，借此可以一次性生成 128 维描述子，而不必分别计算每块的 8 维直方图再连接。

```

1 tmp1 = (np.arange(4*r) // r * 32).reshape(-1, 1)
2 tmp2 = np.arange(4*r) // r * 8
3 appendix = tmp1 + tmp2

```

最后，再将每张图片上每个特征点的 SIFT 描述子存储在 json 文件中，以便搜索时直接载入。

搜索部分

搜索部分首先加载之前保存的 json 文件，然后根据上一节中 pHash 的结果确定需要搜索的图片范围。将待搜索图片进行图像金字塔尺度变换，找出图片中的特征点并且计算特征点的 SIFT 描述子。将图像金字塔上的每幅图片与图库中的图片进行匹配，找出最相似的图片。

先加载预处理时存储的 json 文件，跳出所有在 pHash 筛选范围内的图片，将它们的文件名和特征点 SIFT 描述子保存在列表中。

```

1 with open("sift_chars.json", 'r') as load_f:
2     r_chars = json.load(load_f)
3 with open("sift_files.json", 'r') as load_f:
4     f_list = json.load(load_f)
5
6 scope = filtrate(cv2.imread(img_file, 0))
7 resource_chars = []
8 file_list = []
9 for i in range(len(r_chars)):
10     if f_list[i] in scope:
11         resource_chars.append(r_chars[i])
12         file_list.append(f_list[i])

```

然后，对待搜索图片进行缩放，构建图像金字塔。金字塔上的 4 张图片边长依次为原图的 2 倍、1 倍、0.5 倍、0.25 倍。

```

1 def getPyramid(img):
2     y_size, x_size = img.shape[:2]
3     imglist = []
4     imglist.append(cv2.resize(img, (x_size<<1, y_size<<1)))
5     for i in range(3):
6         imglist.append(cv2.resize(img, (x_size>>i, y_size>>i)))
7     return imglist

```

接着，对图像金字塔上的每张图片用和预处理部分相同的方式计算各特征点 128 维 SIFT 描述子。

```

1 chars, corners = getChar(imglist)

```

然后，计算待搜索图片和图库中图片的相似度。对于图库中的每一张图片，两两比较它和待搜索图片的特征点的 128 维 SIFT 描述子，找出和它的每个特征点最接近的待搜索图片中的两个特征点，得到该特征点和这两个点描述子的欧式距离。

```

1 for i in range(len(resources)):
2     for rvector in resources[i]:
3         min1, min2 = getmin2dis2(target, rvector)

```

```

1 def getmin2dis2(target, rvector):
2     tmp = np.zeros(target.shape[0]).reshape(-1,1)
3     resource = rvector + tmp
4     result = target - resource
5     result *= result
6     result = result.sum(axis = 1)
7     tmp = result.argmin()
8     min1 = math.sqrt(result[tmp])
9     result[tmp] = 10.0
10    min2 = math.sqrt(result.min())
11    return min1, min2

```

如果最小距离 d_1 和次小距离 d_2 满足： $d_1 \leq d_2 \cdot 0.75$ ，那么就认为这个特征点和与它的描述子欧式距离最小的特征点匹配上。

```

1 if min1 < min2 * 0.75:
2     match_numbers[i] += 1

```

需要注意的是，对于待搜索图片图像金字塔上的每幅图片都需要进行以上两步操作。

最后，找出图库中与待搜索图片（图像金字塔上的任一幅）匹配的特征点最多的若干张图片，作为最终的搜索结果。

```
1 rn = []
2 res_num = 10
3 for i in range(res_num):
4     tmp = match_numbers.argmax()
5     rn.append(tmp % len(resources))
6     match_numbers[:,tmp % len(resources)] = 0
```

4.2 图片搜文字

图片搜索文字部分的实现主要基于 Tensorflow 进行图片的目标检测和对象识别，通过深度学习训练出的神经网络模型在带搜索图片中定位对象并识别每个对象。这里我们主要借助了 python 中的 imageAI 库和 RetinaNet50 模型来实现这一功能。

4.2.1 imageAI

ImageAI 是一个 python 库，借助它可以使您使用较为简单的代码构建具有包含深度学习和计算机视觉功能的应用程序和系统。

它的可以将检测到的对象返回到数组中，然后利用数组中的数据在每个对象上绘制矩形标记来生成新图像。可以使用 RetinaNet 目标检测算法，并且可以调整模型性能和实时处理参数，从而实现提高准确率或者减少运行时间的目的。

检测效果如下图所示。

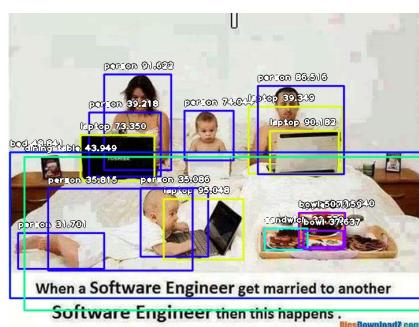


图 5: 检测结果

4.2.2 ResNet-50

ResNet 应用了残差学习的思想，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络只需要学习输入、输出差别的那一部分，简化学习目标和难度。避免信息传递的时候或多或少会存在信息丢失，损耗以及由此带来的梯度下降等问题。

在 ImageAI 中使用了 ResNet-50 模型，下面是网络模型的整体模型图。

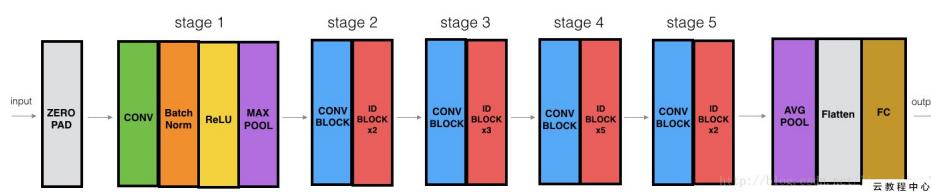


图 6: 网络结构

其中的 CONV 表示卷积层，Batch Norm 表示 Batch 归一化层，ID BLOCK 表示 Identity 块，由多个层构成，如下图所示。

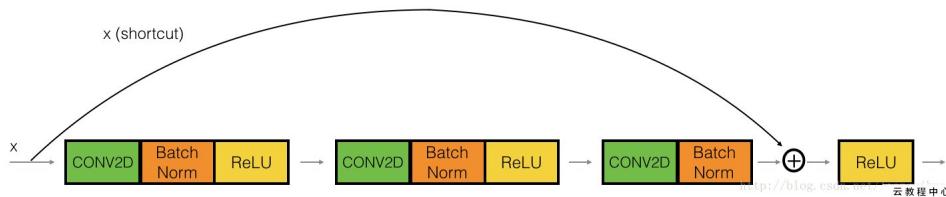


图 7: ID BLOCK

Conv BLOCK 表示卷积块，同样由多个层构成，如下图所示。

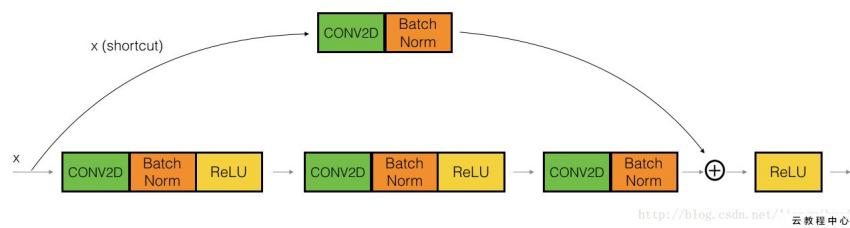


图 8: Conv BLOCK

为了使得 model 结构更加清晰，才提取出了 conv block 和 id block 两个“块”，分别把它们封装成函数。

4.2.3 具体实现

首先，需要导入 ImageAI.Detection 类，以及 Python os 类。

```
1 from imageai.Detection import ObjectDetection  
2 import os
```

然后，加载需要使用的模型。先借助 imageAI 库定义检测类，并且设置模型类型 RetinaNet，然后设置好模型路径。

```
1 execution_path = os.getcwd()  
2 detector = ObjectDetection()  
3 detector.setModelTypeAsRetinaNet()  
4 detector.setModelPath(os.path.join(execution_path , "models/  
    resnet50_coco_best_v2.0.1.h5"))
```

接下来，调用模型中的检测函数 `detector.detectObjectsFromImage()`，处理待搜索的图像，函数中的参数 `minimum_percentage_probability` 设置为 30，即识别概率大于 30% 的结果会被显示出来。

```
1 detections = detector.detectObjectsFromImage(input_image=in_file,  
                                              output_image_path=out_file, minimum_percentage_probability=30)
```

然后，将识别出的对象提取出来，这里我们只需要对象的名称，而抛弃它的位置信息。将其保存在列表中，完成目标检测和对象识别功能。

```
1 labels = []  
2 for eachObject in detections:  
3     label = translator(eachObject["name"])  
4     if label not in labels:  
5         labels.append(label)
```

4.2.4 与文字搜索的对接

图片中识别出的对象可以直接放入文字搜索部分的 Elastic Search 中，搜索对应的文本内容。

但是这里存在一个问题，即目标检测的结果为英文，但搜索文本主要为中文。所以，我们额外编写了一个翻译模块。

在翻译模块中，借助 python 的 urllib 库，通过浏览器的审查元素提取出有道翻译接口地址。

```
1 req_url = 'http://fanyi.youdao.com/translate?smartresult=dict&smartresult=rule'
```

在请求正文中，我们可以找到需要向服务器提交的数据内容。

```
1 Form_Date = {}
2 Form_Date['i'] = text
3 Form_Date['doctype'] = 'json'
4 Form_Date['form'] = 'AUTO'
5 Form_Date['to'] = 'AUTO'
6 Form_Date['smartresult'] = 'dict'
7 Form_Date['client'] = 'fanyideskweb'
8 Form_Date['salt'] = '1526995097962'
9 Form_Date['sign'] = '8e4c4765b52229e1f3ad2e633af89c76'
10 Form_Date['version'] = '2.1'
11 Form_Date['keyform'] = 'fanyi.web'
12 Form_Date['action'] = 'FY_BY_REALTIME'
13 Form_Date['typoResult'] = 'false'
```

使用 urllib.parse.urlencode() 方法将其转化为服务器可以处理的编码方式，然后将其提交。

```
1 data = parse.urlencode(Form_Date).encode('utf-8')
2 response = request.urlopen(req_url, data)
```

之后，读取以 json 格式返回的翻译结果，将其载入为列表，调取其中我们需要的中文内容。

```
1 html = response.read().decode('utf-8')
2 translate_results = json.loads(html)
3 translate_results = translate_results['translateResult'][0][0]['tgt']
```

然后，调用这个翻译模块，将图像识别出的英文结果翻译成为中文，放入 elasticsearch 的索引中进行搜索，从而实现图片搜索文字的功能。

5 Web 前端

5.1 项目运行环境

项目运行环境如下表所示：

操作系统	GNU/Linux
内核版本	Linux-4.19.12-arch1-1-ARCH-x86_64
发行版	Arch Linux
miniconda 版本	4.5.12
python 版本	3.6.5
elasticsearch 版本	6.3.1
OpenCV 版本	3.4.2
Tensorflow 版本	1.12.0(cpu)
tornado 版本	5.1.1

5.2 Web 框架：tornado

为了能够更好地兼容 Python 3，我们将 web 框架从 web.py 切换到了 tornado。在下文我们可以看到它们在处理请求和 web.py 有许多类似之处。

5.2.1 框架基本设置

在使用 tornado 框架时首先需要配置 static 和 template 路径：

```
1 settings = {
2     # "debug": True,
3     "static_path": os.path.join(os.path.dirname(__file__), "static"),
4     "template_path": os.path.join(os.path.dirname(__file__), "template")
5 }
```

5.2.2 路由设置

使用 tornado 框架时，首先需要配置路由信息，这样服务器在处理请求时可以针对不同的路由的请求给出不同的响应。

路由信息如下：

```
1 handlers = [
2     (r"/", IndexHandler),
3     (r"/s", SearchHandler),
4     (r"/i", SearchImageHandler),
5     (r"/v", ViewNewsHandler),
6     (r"/(favicon.ico)", tornado.web.StaticFileHandler, dict(path=settings
7         ['static_path']))
]
```

服务器在接收到请求后，会根据路由表中定义的路由，将请求交给对应的 handler 处理。

5.2.3 响应

在 handler 中，我们需要重载对应的 get 或者 post 方法，用 self.get_argument 或者 self.request.files.get 来获得 url/请求中包含的数据。

```
1 # for GET method
2 keyword = self.get_argument('keyword', '')
3
4 # for POST method (files)
5 img = self.request.files.get('image', None)
```

5.2.4 tornado template

tornado 框架下模板的使用方式和 web.py 下不同。

首先，需要使用 ... 来对表达式求值。示例如下：

```
1 {{ image_item['data']['url'] }}
```

随后，控制语句的形式是% ... %，包括 if、else、for 等控制语句以及临时变量的复制语句。示例如下：

(1) for 循环

```
1 {% for item in items %}
2     <!-- contents -->
3 {% end %}
```

(2) if 条件判断

```
1  {%- if flag %}  
2      <!-- contents -->  
3  {% else %}  
4      <!-- contents -->  
5  {% end %}
```

(3) set 临时变量赋值

```
1  {%- set temp = lvalue + rvalue %}
```

利用这些模板就可以根据搜索内容的不同和页码的不同生成对应的结果页面。

5.2.5 启动服务器

使用下面这段代码来启动服务器：

```
1 app = tornado.web.Application(handlers=handlers, **settings)  
2 http_server = tornado.httpserver.HTTPServer(app)  
3 http_server.listen(options.port)  
4 tornado.ioloop.IOLoop.instance().start()
```

5.3 CSS/HTML 框架：Bootstrap 4

为了使页面的布局更加方便，我们在前端使用了 Bootstrap 4 框架。使用 bootstrap 的 grid 系统可以大大减轻排版网页时的工作量，从而获得更加整齐美观的页面。

除了网格系统，bootstrap 还提供了一些组件，使得网页更加美观。

我们使用的 Bootstrap 组件如下：

- 导航栏
- 表单 (Input Group, Form Group)
- 轮播 (Carousel)
- 警告 (Alert)
- jumbotron

- 分页栏 (Pagination)

最后的效果在 [7](#) (结果展示) 中有展示。

6 性能优化

1. 使用支持 avx 指令集的 CPU 版本 tensorflow

由于我们的计算机上的显卡不支持对深度学习的加速，因此我们只能选择 CPU 版本的 tensorflow。为了尽可能加快深度神经网络的速度，我们采用了支持 avx 指令集的 CPU 版本 tensorflow。这样 tensorflow 可以向量化部分运算，从而提高运行速度。

预先编译好的包来自<https://github.com/lakshayg/tensorflow-build>，使用下面的命令安装：

```
1 pip install --ignore-installed --upgrade ./tensorflow-1.12.0-cp36-cp36m-  
linux_x86_64.whl
```

2. 预先计算一部分静态的数据在图像搜索图像时，我们预先计算好了数据集内所有图片的 hash 值和 sift 特征，这样在比对目标图片时不需要重复计算，从而减少运行时间。我们存储的格式为 json 格式，解析起来比较快，而且作为文本文件比较容易压缩。

但是由于内存容量有限，我们不可能对任意大小的数据集都采用这种方式来加速。由于网站演示时的数据量不大，我们采用了这种加速的办法。

3. 将数据在启动时读入内存，避免频繁的硬盘读写由于预先计算了数据，因此我们需要在运行时将 json 读入内存。然而，这些文件动辄上 G，每搜索一次就要进行大量的硬盘读写操作，这样大量的时间就被消耗了。因此我们在启动服务器时就将这些数据一并载入，这样之后查询时就不需要再读取硬盘，从而加快响应的速度。

7 结果展示

7.1 文字搜文字

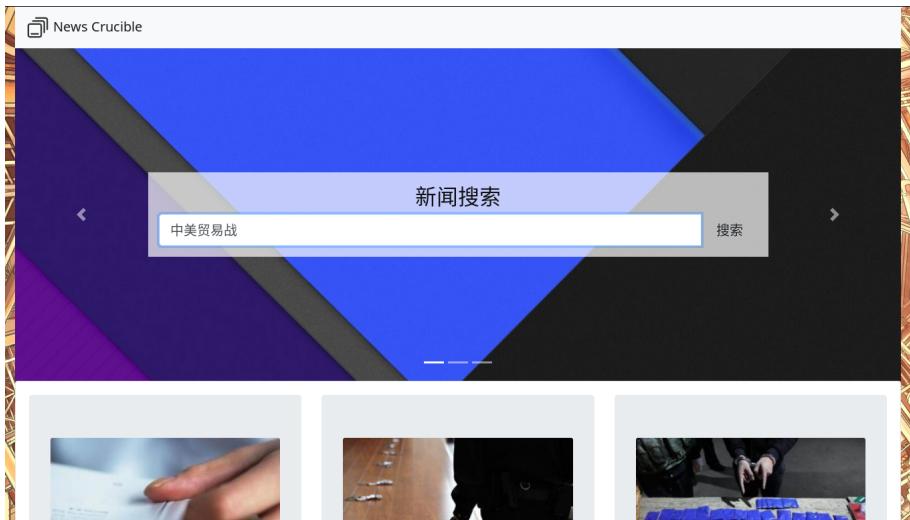


图 9: 搜索界面



图 10: 搜索结果

7.2 文字搜图片



图 11: 搜索界面

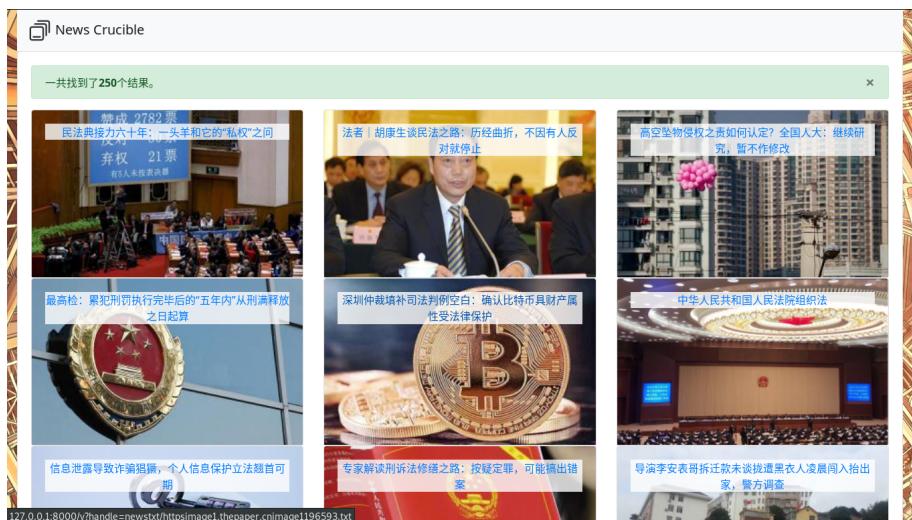


图 12: 搜索结果

7.3 相关新闻推荐



图 13: 新闻条目



图 14: 相似新闻

7.4 图片搜图片



图 15: 搜索界面



图 16: 上传的图片 (经过轻微旋转)

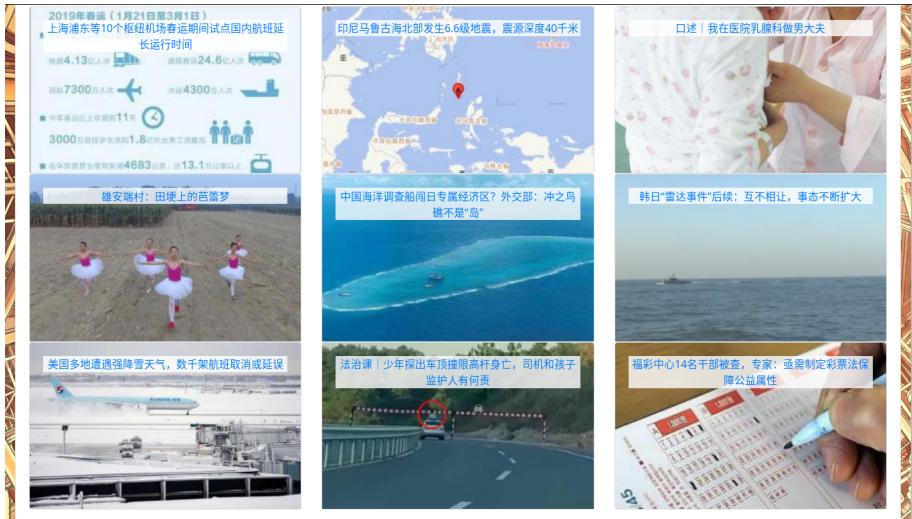


图 17: 搜索结果

7.5 图片搜文字

上传的图片与上一节相同。



图 18: 搜索结果 (1)



图 19: 搜索结果 (2)