

# 目录

<b>1 项目介绍</b>	<b>2</b>
1.1 想法来源 . . . . .	2
1.2 基本思路 . . . . .	2
<b>2 实现过程</b>	<b>3</b>
2.1 数据爬取 . . . . .	3
2.2 数据预处理 . . . . .	3
2.3 数据索引 . . . . .	5
2.4 数据检索 . . . . .	7
2.5 网站前后端实现 . . . . .	9
<b>3 成果展示</b>	<b>12</b>
3.1 文本搜索 . . . . .	12
3.2 人脸搜索 . . . . .	13
<b>4 总结与分析</b>	<b>15</b>

# 1 项目介绍

## 1.1 想法来源

初入学交大, 在开学典礼以及各种班会、讲座上, 我被推荐了许许多多的重要的网站: 教务处网站, 学生办网站, 生活园区网站等等. 这些网站对于身处交大的学生而言, 每一个都含有必需且重要的信息, 所以我不得不常常去浏览, 以免错过某些通知公告.

诚然, 可以使用 RSS 订阅的方法来解决这个问题, 事实上, 我也是这么做的. 但除此之外, 有时候需要查看过往的通知, 而这些网站的搜索功能往往难以使用, 或者根本就形同虚设; 至于使用百度等通用搜索引擎, 由于其时效性与针对性不强, 用起来也颇有不便.

因此, 在构思本课程的大作业项目时, 我决定做一个针对交大的文本搜索引擎, 用以检索校内信息.

而关于多媒体信息检索, 考虑到校内的图片大多数为各种活动的新闻照片, 其中的主体是交大的师生们, 我便萌生了做一个人脸搜索引擎的想法, 可以找出一个人在哪些照片中出现.

## 1.2 基本思路

本项目主要分为两部分: 文本搜索与人脸搜索.

对于文本搜索, 首先爬取一定数量的相关网页并存储在本地, 对其进行数据处理, 使用 Elastic-Search 创建索引并实现检索功能; 对于人脸搜索, 从文本搜索的保存的网页中, 提取一定数量的图片并存储在本地, 剔除不含有人脸存在的图片后, 对每张图片, 检测其中的人脸并生成特征向量, 在检索时, 匹配相似度达到阈值的图片.

## 2 实现过程

### 2.1 数据爬取

在本课程前期的上机实验中, 我已经实现了基本的爬虫、解析工具类, 在此便可以直接使用. 其具有的多线程爬取, 线程池, 编码自动检测,BloomFilter 等功能, 在之前的实验报告中已较详细地说明, 这里就不再展开.

值得一提的是, 在进行本项目的过程中, 我实现了一些新的功能: 爬取范围限定, 网页的动态优先级.

爬取范围限定, 即是将爬取的网页限定在一定的范围内. 本项目中, 我限定仅爬取交大主页与新闻网, 以及电院相关的部分网站的内容. 至于编程实现, 采用的是最朴素的暴力匹配, 尽管效率较低, 但在爬虫应用中, 程序的主要耗时在于网络通信, 故简单的暴力匹配已经足够.

```
1 def __check_domain(self, url):
2     if not self.domains:
3         return True
4     current_domain = urlparse(url).netloc
5     return any(current_domain.find(x) != -1 for x in self.domains)
```

网页的动态优先级, 在初始时, 所有域名的优先级都为 100, 而每次爬取时, 从优先级队列中, 取出优先级最大的网页, 将该网页对应的域名优先级降低 10, 添加其外链进入优先级队列, 同时将其余所有域名的优先级提升 1. 编程实现的主要代码如下:

```
1 domain_to_priority[urlparse(url).netloc] -= 10
2 for k in domain_to_priority.keys():
3     domain_to_priority[k] = min(100, domain_to_priority[k] + 1)
```

这样的操作, 可以确保爬取网页的顺序相对均匀, 避免出现在一段时间内只爬取某一网站的现象, 对爬取对象而言, 也能一定程度减轻其压力, 实现爬虫的友善性.

至于图片数据的爬取, 在完成对网页数据的爬取之后, 从网页内容中提取出 `img` 标签对应的图片地址, 将其下载保存至本地即可. 这一步骤较为简单, 故在此不详述.

### 2.2 数据预处理

在完成数据的爬取之后, 面对约 10k 的数据量, 在对其进行索引操作之前, 我先做了一些预处理工作, 清理无用数据, 以便后续工作.

对于网页数据, 由于在爬取时我并没有严格的检查爬取内容是否为 HTML 文档, 实际爬取的数据中含有少量的非 HTML 文档(比如 Word 文档与 PDF 文件), 以及一些错误页面(比如 404 页面). 这些数据会干扰后续的索引与检索工作, 因此需要被清理.

对于非 HTML 文档, 分析爬取的网页, 我发现某些特定的 URL 是文件下载链接, 那么, 只需要从爬取的网页数据中, 找到这些 URL 对应的数据, 将其清理即可; 对于无意义的错误页面, 其主要特征在于文件大小很小(往往小于 2kb), 那么也很容易将其发现.

```

1 import os
2 total = 0
3 valid = 0
4 file_to_remove = []
5 with open("../data\\index.txt", mode="r") as original_index:
6     with open("../data\\new_index.txt", mode="w") as new_index:
7         for line in original_index:
8             total += 1
9             try:
10                 url, file = line.split()
11             except:
12                 continue
13             if not os.path.exists(file):
14                 continue
15             to_remove = url.find("fileUpload.action") != -1 or
16                 url.find("download.action") != -1
17             to_remove = to_remove or os.path.getsize(file) <= 2048:
18             if to_remove:
19                 file_to_remove.append(file)
20                 continue
21             new_index.write("{}\t{}\n".format(url, file))
22             valid += 1
23 print("{} / {}".format(valid, total))
24 confirm = input("There are {} invalid files, remove
25     all? ".format(len(file_to_remove)))
26 if confirm and confirm[0].lower() == 'y':
27     for file in file_to_remove:
28         os.remove(file)
29 print("All removed.")

```

对于图片数据，在后续步骤中，我将检测其中的人脸并提取特征，为了减少后续的工作量，需要对图片进行粗筛，去除那些明显不含有人脸的图片。在这一步，准确度和速度的权衡中，速度更为重要，因此我使用传统的机器学习方法，用 OpenCV 进行特征匹配：

```

1 import cv2
2 import sys
3 import os

4 faceCascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")

5 total, face_exist = 0, 0
6 images_with_faces = list()

```

```

7  data_dir = "..\\data"
8  image_dir = os.path.join(data_dir, "image_data")
9  image_index = os.path.join(data_dir, "image_index.txt")
10 new_image_index = os.path.join(data_dir, "new_image_index.txt")

11 with open(image_index, mode="r") as i_index:
12     with open(new_image_index, mode="w") as new_i_index:
13         for line in i_index:
14             try:
15                 image_url, image_file, page_url, html_file = line.split()
16             except:
17                 continue
18             total += 1
19             gray_img = cv2.imread(os.path.join(image_dir, image_file),
20                                   cv2.IMREAD_GRAYSCALE)
21             gray_img = cv2.equalizeHist(gray_img)
22             faces = faceCascade.detectMultiScale(
23                 gray_img,
24                 scaleFactor=1.1,
25                 minNeighbors=5,
26                 minSize=(10, 10)
27             )
28             if len(faces) > 0:
29                 print("\n{img}".format(img=image_file))
30                 face_exist += 1
31                 new_i_index.write(
32                     "{url}\t{file}\t{page}\t{faces}\n".format(
33                         image_url, image_file, page_url, len(faces)))
34             print("\r{}/{}/".format(face_exist, total), end="")

```

## 2.3 数据索引

对于网页数据的索引, 本课程之前的上机实验中, 采用的是 Lucene, 而在本项目中, 我使用了 ElasticSearch, 一个基于 Lucene 的更高层次抽象, 抛弃了繁冗的细节, 使用时更加简便. 同时在性能上, 也显著优于直接使用 Lucene. 编程实现的主要代码:

```

1  class HtmlDoc(Document):
2
3      title = Text(analyzer="ik_max_word", search_analyzer="ik_smart")
4      url = Keyword()
5      host = Keyword()

```

```

5      text = Text(analyzer="ik_max_word", search_analyzer="ik_smart")

6  class Index:
7      name = "qianmo"
8      settings = {
9          "number_of_shards": 2,
10     }

11 connections.create_connection(hosts=['localhost'])
12 HtmlDoc.init()
13 indexed_num = 0
14 with open(index_file, mode="r", encoding="utf8") as f:
15     for line in f:
16         url, file_path = line.split()
17         file_path = os.path.join(data_dir, file_path)
18         with open(file_path, mode="r") as html_file:
19             html_content = html_file.read()
20             text = html_to_text(html_content)
21             title = get_title(html_content, url)
22             host = urlparse(url).netloc
23             doc = HtmlDoc(title=title, text=text, url=url, host=host)
24             doc.save()
25             indexed_num += 1

```

可以看出,ElasticSearch 创建索引的过程较为简单方便. 至于上面代码中调用的函数, 相较于之前的上机实验, 我也做了一些针对性的改进, 但简洁起见, 不在此贴出其具体实现.

对于图片数据的索引, 先从每张图片中提取出可能的多个人脸, 然后为每个人脸生成特征向量并保存. 在上一步的数据预处理中, 我采用的是 OpenCV 中传统的特征匹配方法, 而在此, 为了更高的提取精度, 我使用了 MTCNN[2] 与 FaceNet[1].

MTCNN[2] 主要有 3 个网络结构:Proposal Net(P-Net), Refine Net(R-Net), Output Net(O-Net), 可以实现图片中较为精确地人脸检测与定位. 在本项目中, 对其输入一张可能存在人脸的图片, 其输出为多个矩形框选中的人脸,

FaceNet[1] 是 Google 的研究人员提出的一种深度学习人脸检测模型, 可以将一人脸映射为欧式空间中的固定维度向量. 在本项目中, 对其输入一张人脸, 则得到一个 512 维的人脸特征向量.

在实际创建索引时, 由于本项目中含有的人脸数在 15k 左右, 在运行 FaceNet 模型时, 需要设定合理的批尺寸 (Batch size). 如果批尺寸过大, 运行时会出现 OOM 错误; 而如果批尺寸过小, 则总的运行时间会很长. 批尺寸需要合理权衡, 最终我设定为 1k.

需要说明的是, 这一部分的代码我参考了 Github 上 FaceNet 的开源实现<sup>1</sup>, 并使用了其给出的训练好的模型. 由于这部分的代码较长, 简洁起见, 在此不贴出, 源代码已随报告附上.

---

<sup>1</sup>Face recognition using Tensorflow: <https://github.com/davidsandberg/facenet>

## 2.4 数据检索

对于网页数据, 其索引与检索都是借助于 ElasticSearch 实现. 那么, 实现网页数据的检索相对而言就十分简单了, 并且, 还可以轻松地实现关键词高亮功能.

```
1 def search_html(query_string, page=1, page_size=10):
2     search_from = page_size * (page - 1)
3     index = "qianmo"
4     doc_type = "doc"
5     fields = ["title", "text"]
6     tick = time.time()
7     response = es.search(
8         index=index,
9         doc_type=doc_type,
10        body={
11            "query": {
12                "multi_match": {
13                    "query": query_string,
14                    "fields": fields
15                }
16            },
17            "from": search_from,
18            "size": page_size,
19            "highlight": {
20                "pre_tags": ["<span class = \"highlight\">"],
21                "post_tags": ["</span>"],
22                "fields": {
23                    "title": {
24                        "no_match_size": 100
25                    },
26                    "text": {
27                        "fragment_size": 25,
28                        "number_of_fragments": 4,
29                        "no_match_size": 100
30                    }
31                }
32            }
33        }
34    )
35    tock = time.time()
36    time_used = tock - tick
37    result_num = response["hits"]["total"]
38    search_result = list()
```

```

39     for hit in response["hits"]["hits"]:
40         single_result = dict()
41         single_result["title"] = hit["highlight"]["title"][0]
42         single_result["content"] = "...".join(hit["highlight"]["text"])
43         single_result["url"] = hit["_source"]["url"]
44         search_result.append(single_result)
45     result_data = {
46         "time_used": time_used,
47         "total_num": result_num,
48         "page": page,
49         "search_result": search_result,
50         "query_string": query_string,
51     }
52     return result_data

```

对于图片数据, 在索引过程中, 我计算了所有图片中所有人脸的 512 维特征向量, 并保存了每张脸对应的相关信息. 那么, 在查询时, 给定一张图片, 需要做的就是从中定位出人脸, 提取其特征向量, 与已有的所有特征向量进行比对, 找出相似度超过阈值的那些人脸.

需要说明的是, 关于高维向量的匹配, 本课程介绍了局部敏感哈希 (LSH) 的方法, 其效率应当是高于朴素的暴力匹配. 然而在本项目中, 这一点并不成立, 原因在于: 我用 Python 实现的简单 LSH 运行效率不如 Numpy 中使用 C++ 编写的矩阵运算. 因此, 我最终采用的方法是: 将所有人脸的特征矩阵 ( $N \times 512$  矩阵) 与目标人脸的特征向量 (512 维) 相乘, 得到相似度向量 ( $N$  维), 从而找出匹配人脸.

```

1 def search_face(image_hash):
2     result_data = dict()
3     result_data["target_url"] = "/static/upload/" + image_hash + ".jpg"
4     result_data["search_result"] = list()
5     image_file = os.path.join(upload_dir, image_hash + ".jpg")
6     target_feature = extract_feature(image_file)
7     if target_feature is None:
8         print("No face detected!")
9         return result_data
10    similarity = np.dot(all_faces, target_feature.T)
11    sorted_index = np.argsort(similarity)
12    prev_similarity = None
13    for i in range(1, 21):
14        if similarity[sorted_index[-i]] <= similarity_threshold:
15            break
16        if prev_similarity == similarity[sorted_index[-i]]:
17            continue
18        prev_similarity = similarity[sorted_index[-i]]
19        face_info = face_info_list[sorted_index[-i]]

```

```

20     image_url, page_url = face_info.split()[0], face_info.split()[2]
21     page_title = get_page_title(page_url)
22     result_data["search_result"].append({
23         "url": image_url,
24         "page_url": page_url,
25         "page_title": page_title,
26     })
27     return result_data

```

在上面代码中, 涉及的具体函数 (如 `extract_feature`) 与索引过程类似, 在此略去其具体实现.

## 2.5 网站前后端实现

在完成了以上步骤后, 整合网站的前后端. 对于网页搜索, 我基本沿用了本课程中期整合的前端设计, 而仅仅将后端从 Lucene 替换为 ElasticSearch; 对于人脸搜索, 我做了比较大的改动.

首先是人脸检索的主页面, 需要由用户上传含有人脸的图片. 考虑到上传的易用性与美观性 (朴素的 HTML 文件上传元素不够美观且难以改造), 我使用了 Github 上开源项目, Uppy<sup>2</sup>, 一个优秀的前端文件上传插件, 支持用户通过本地文件、网络文件以及摄像头来上传图片.

```

1 const uppy = Uppy.Core({
2   debug: false,
3   autoProceed: true,
4   restrictions: {
5     maxFileSize: 2000000,
6     maxNumberOfFiles: 1,
7     minNumberOfFiles: 1,
8     allowedFileTypes: ['image/*']
9   }
10 })
11 .use(Uppy.Dashboard, {
12   inline: true,
13   target: '.DashboardContainer',
14   replaceTargetContent: true,
15   showProgressDetails: true,
16   note: 'Only 1 image file allowed, up to 2 MB',
17   height: 300,
18   width: 1000,
19   metaFields: [
20     { id: 'name', name: 'Name', placeholder: 'file name' },
21     { id: 'caption', name: 'Caption', placeholder: 'describe what the image
22       is about' }

```

<sup>2</sup>Uppy: <https://github.com/transloadit/uppy>

```

22 ],
23 browserBackButtonClose: true
24 })
25 .use(Uppy.Webcam, { target: Uppy.Dashboard })
26 .use(Uppy.XHRUpload, { endpoint: '/image-upload' })

```

当用户上传图片时, 网站后端接收到图片, 首先计算图片内容的哈希值, 与已有图片进行比对, 倘若存在, 便不再重复保存, 直接返回哈希值即可; 若不存在, 则在保存后返回哈希值. 而前端得到后端返回的哈希值, 便直接携带哈希值前往搜索结果页面.

```

1 class Upload:

2     def POST(self):
3         x = web.input()
4         if "files[]" not in x:
5             print("No file found!")
6             return None
7         image_content = x.get("files[]")
8         md5 = hashlib.md5()
9         md5.update(image_content)
10        image_hash = md5.hexdigest()
11        image_path = os.path.join(upload_dir, str(image_hash)+".jpg")
12        if not os.path.exists(image_path):
13            with open(image_path, mode="wb") as f:
14                f.write(image_content)
15        result_info = {
16            "hash": image_hash,
17        }
18        web.header('content-type', 'text/json')
19        return json.dumps(result_info)

```

```

1 uppy.on('upload-success', (file, resp, uploadURL) => {
2     window.location.href = "/search/face/" + resp.hash;
3 })

```

而在执行搜索时, 后端根据图片的哈希值, 找到保存的图片, 进行检索并返回结果即可, 这一部分在图片数据的检索2.4中已提及, 但值得一提的是, 在未经优化之前, 单张图片查询时间在 30 秒以上, 这显然是无法接受的, 因此我进行了一些简单的优化, 使得最终查询时间降至 500 毫秒以内.

经过分析, 未经优化之前, 查询的主要耗时在于 Tensorflow 加载 MTCNN 以及 FaceNet 的模型, 而在加载完成之后, 计算时间相对较短, 在 1 秒以内. 因此, 我采用了预先加载模型的方法, 在启动时就将完整的模型载入内存, 计算时不再重复载入, 这样大幅加快了查询速度.

```

1 # init
2 es = connections.create_connection(hosts=["localhost"])
3 upload_dir = "static/upload"
4 similarity_threshold = 0.6935
5 graph = tf.Graph()
6 with graph.as_default():
7     gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.6)
8     sess = tf.Session(config=tf.ConfigProto(gpu_options=gpu_options,
9         log_device_placement=False))
10    with sess.as_default():
11        pnet, rnet, onet = align.detect_face.create_mtcnn(sess, None)
12        facenet.load_model(facenet_model)

```

此外, 我将人脸对应信息也预先载入内存, 以空间换时间.

```

1 all_faces = np.load(face_feature_file)
2 with open(face_info_file, mode="r", encoding="utf8") as f:
3     face_info_list = f.readlines()

```

在经过这样的优化之后, 人脸搜索的查询时间在 1 秒左右. 更进一步的优化在于, 将部分信息放入 ElasticSearch, 查询时借助 ElasticSearch, 代替原先的顺序查找. 以查询图片对应的网页标题为例:

```

1 def get_page_title(page_url):
2     index = "qianmo"
3     doc_type = "doc"
4     response = es.search(
5         index=index,
6         doc_type=doc_type,
7         body={
8             "query": {
9                 "term": {
10                     "url": page_url,
11                 }
12             }
13         }
14     )
15     result_num = response["hits"]["total"]
16     if result_num == 0:
17         return "未找到标题"
18     return response["hits"]["hits"][0]["_source"]["title"]

```

至此, 优化基本完成, 查询时间可控制在 500 毫秒以内. 而搜索结果的前端展示, 我实现了瀑布流效果, 同时在每张图片上增加了悬停遮罩, 显示图片相关信息与链接. 具体效果参见成果展示3.2.

### 3 成果展示

#### 3.1 文本搜索



图 1: 网页搜索主页面

A screenshot of the search results page from the QianMo search engine. The search query '新年新气象' is entered in the search bar at the top. Below the search bar, a message indicates '找到 113 条结果,用时 0.0301 秒'. The results are displayed in three separate boxes. Each box contains a title, a snippet of text, and a link. The first result is '姜斯宪书记林忠钦校长新年贺词-上海交通大学新闻网', with the snippet: '姜斯宪书记林忠钦校长新年贺词...林忠钦全体师生医务人员、校友们、朋友们：大家新年好...2018的新年钟声已经敲响，在这辞旧迎新的美好时刻，...让我们满怀信心和期待，一起迎接新年新气象、新挑战！' and the link 'http://oldnews.sjtu.edu.cn/info/1002/1640579.htm'. The second result is '姜斯宪书记林忠钦校长新年贺词-上海交通大学新闻网', with the snippet: '姜斯宪书记林忠钦校长新年贺词...林忠钦全体师生医务人员、校友们、朋友们：大家新年好...2018的新年钟声已经敲响，在这辞旧迎新的美好时刻，...让我们满怀信心和期待，一起迎接新年新气象、新挑战！' and the link 'http://oldnews.sjtu.edu.cn/info/1017/1645514.htm'. The third result is '上海交通大学建校122周年 学生创新中心新启航[图]-上海交通大学新闻网', with the snippet: '全方位展示中心在校企合作深化、视觉形象升级、学生科协建设等各方面的新气象...姜斯宪书记林忠钦校长新年贺词' and the link 'http://oldnews.sjtu.edu.cn/info/1002/1646705.htm'. The background of the page features a grid pattern.

图 2: 网页搜索结果页面



图 3: 网页搜索结果页面

### 3.2 人脸搜索

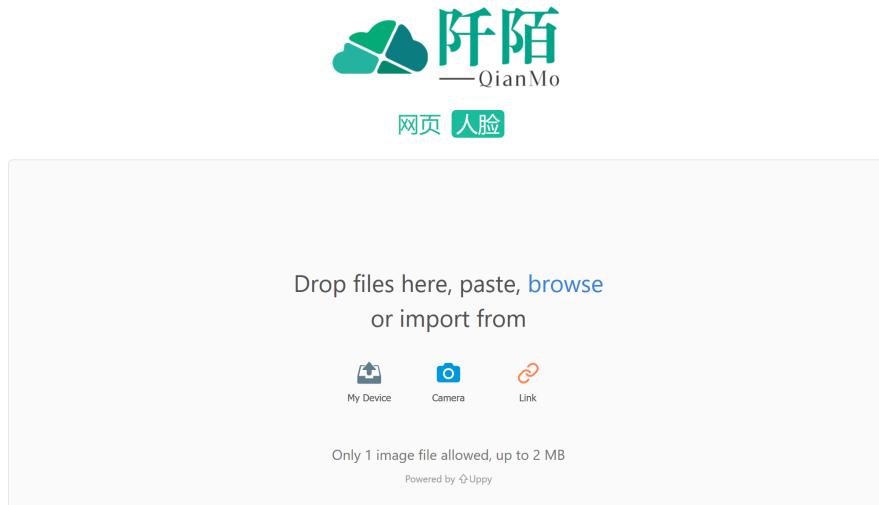


图 4: 人脸搜索主页面



图 5: 人脸搜索结果页面 (绿色边框图片为目标图片)

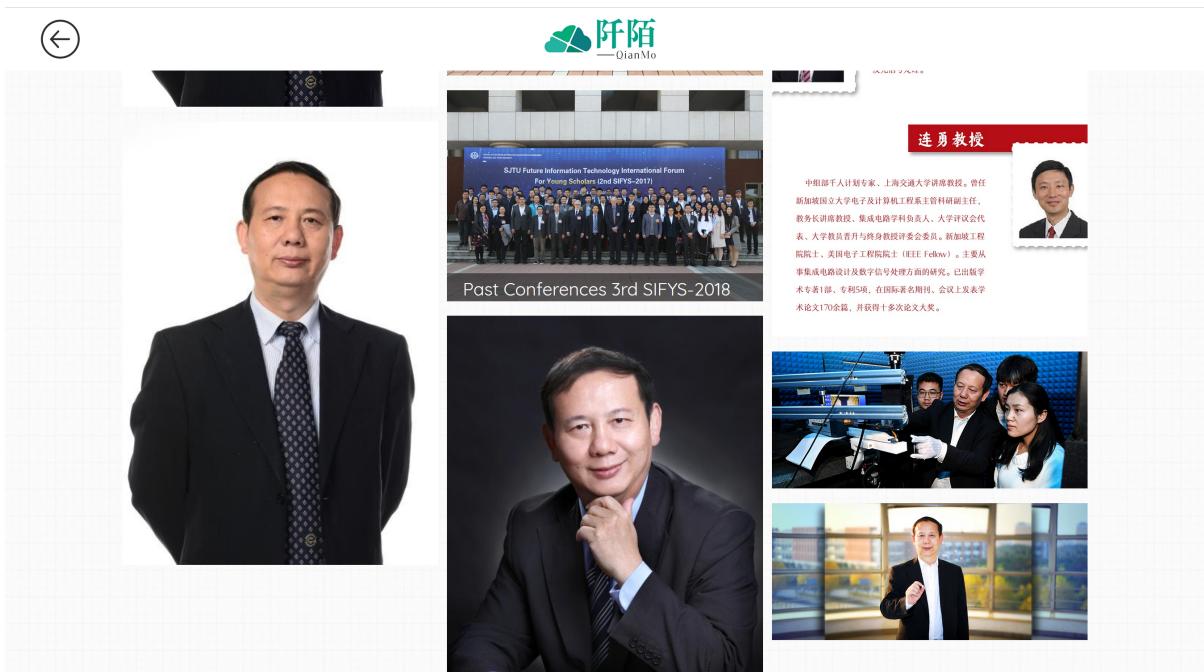


图 6: 人脸搜索结果页面

## 4 总结与分析

在本项目中,我综合运用了本学期所学的多媒体检索相关知识,同时结合深度学习相关知识,实现了一个简单的校内信息检索引擎.在这一过程中,我学习了ElasticSearch的基本概念以及使用方法,并将传统的计算机视觉方法与深度学习方法综合使用,收获颇丰.当然,除此之外,在网站开发的前后端,我都有新的收获,如何优化后端性能与美化前端.

关于一个人独立完成本项目,比起多人协作,诚然显得有些突兀.但在我看来,多人的协作是存在沟通成本的,包括代码的耦合与思路的耦合.那么,对于一个课程的大作业,总的工作量其实并不是很大,在单人可以完成且不会很吃力的范畴内,多人协作甚至可能会导致效率的降低,因而我选择了独自完成本项目.此外,这也给了我同时接触完整项目的的机会,包括信息的爬取与处理,网站的前端与后端,可以更加全面地锻炼自己的能力,我觉得是很有收获的.

## 参考文献

- [1] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.
- [2] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503, 2016.