

Lab6 Learning to Rank 实验报告

10152130122 钱庭涵

point-wise.py

主函数中，调用 create_necessary_folders 函数，创建训练集所需文件夹。

```
540 if __name__ == '__main__':
541     # 创建训练集所需文件夹
542     create_necessary_folders('training')
```

跳转到 create_necessary_folders 函数，进行创建。

```
12 def create_necessary_folders(data):
13     if (not (os.path.exists('./' + data))):
14         os.mkdir('./' + data)
15     if (not (os.path.exists('./' + data + '/query'))):
16         os.mkdir('./' + data + '/query')
17     if (not (os.path.exists('./' + data + '/document'))):
18         os.mkdir('./' + data + '/document')
19     if (not (os.path.exists('./' + data + '/index_file'))):
20         os.mkdir('./' + data + '/index_file')
21     if (not (os.path.exists('./' + data + '/document_len_map'))):
22         os.mkdir('./' + data + '/document_len_map')
23     if (not (os.path.exists('./' + data + '/document_total'))):
24         os.mkdir('./' + data + '/document_total')
25     if (not (os.path.exists('./' + data + '/features'))):
26         os.mkdir('./' + data + '/features')
27     if (not (os.path.exists('./' + data + '/total_features'))):
28         os.mkdir('./' + data + '/total_features')
29     if data == 'training':
30         if (not (os.path.exists('./' + data + '/labels'))):
31             os.mkdir('./' + data + '/labels')
```

跳转回主函数，对训练集的 query 查询和 document 文档进行预处理。

```
544     # 训练集预处理
545     preprocess_query('training')
546     preprocess_document('training')
```

跳转到 `preprocess_query` 函数, 遍历训练集的所有 query 文件, 如果文件的后缀为'.pids', 说明这个文件内存储着该查询对应的候选文档 id, 打开并读取该文件, 在 training/query 文件夹中创建当前 topicid 对应的同名文件, 将读取的内容以新的格式 (每行一个 pid) 写入。

```
33 def preprocess_query(data):
34     query_path = '../2017TAR/' + data + '/extracted_data'
35
36     for a in os.walk(query_path):
37         document_files = a[2]
38         for document_file in document_files:
39             if document_file[-5:] == '.pids':
40                 file1 = open(query_path + '/' + document_file, 'r')
41                 lines = file1.readlines()
42                 file1.close()
43                 file2 = open('../' + data + '/query/' + document_file, 'w')
44                 for line in lines:
45                     tmp1 = re.split(' ', line.strip('\n'))
46                     file2.write(tmp1[1] + '\n')
47                 file2.close()
```

如果文件的后缀为'.title', 说明这个文件内存储着该查询对应的 term, 打开并读取该文件, 将读入的字符串按空格分隔存入列表, 列表中第一项为 topicid, 将列表中第二项至最后一项按空格分隔连接成新的字符串, 调用 `wordlist_pre_process` 函数对字符串进行预处理 (具体方式见下一条), 将处理后的字符串写入对应的文件。

```
49         if document_file[-6:] == '.title':
50             file3 = open(query_path + '/' + document_file, 'r')
51             lines = file3.read()
52             file3.close()
53             tmp2 = re.split(' ', lines.strip('\n'))
54             tmp3 = wordlist_pre_process(' '.join(tmp2[1:]))
55             file4 = open('../' + data + '/query/' + document_file, 'w')
56             file4.write(tmp3 + '\n')
57             file4.close()
```

`wordlist_pre_process` 函数中, 对字符串进行一系列处理 (全部转换成小写字母, 词条化, 去除停用词, 词性归并, 词干还原, 去除大部分无用符号和首尾空格), 返回预处理后的新字符串。

```
61 def wordlist_pre_process(sentence):
62     tokens = nltk.word_tokenize(sentence.lower())
63     lemmatizaer = WordNetLemmatizer()
64     stem_list = [PorterStemmer().stem(lemmatizaer.lemmatize(w)) for w in tokens \
65                  if (w not in stopwords.words('english'))]
66     tmp = re.split('[()();/|.,*"+=<> \\\?!^@_${}%{}-]+', ' '.join(stem_list))
67     tmp1 = ' '.join(tmp)
68     process_sentence = tmp1.replace("`", '').replace("'", "").strip(' ')
69     return process_sentence
```

每处理完一个 topic, 输出处理成功信息。

```
59     print(document_file, 'process succeed')
```

接下来对 document 进行预处理，**跳转到 preprocess_document 函数**，遍历训练集的所有 document 文件。因为存在子文件夹，因此跳过 for 循环的第一次进入，从第二次开始处理，根据路径获取 topicid 和 document 的 pid 列表。

```
71 def preprocess_document(data):
72     document_path = '../docs.' + data + '/topics_raw_docs'
73     count = 0
74     for a in os.walk(document_path):
75         if count == 0:
76             count = 1
77             continue
78         folder_tmp = re.split('/', a[0].replace('\\', '/'))
79         folder = folder_tmp[-1]
80         document_files = a[2]
```

创建存放当前 topic 的 document 文件夹。

```
82         if (not (os.path.exists('../' + data + '/document/' + folder))):
83             os.mkdir('../' + data + '/document/' + folder)
```

对于每一个 document 文件，调用 xml.etree.ElementTree 模块，用 xml 文件的方式打开 document 文件，使用 find 的方式找到 ArticleTitle 标签，获取该标签的文本，并调用 wordlist_pre_process 函数进行预处理。

使用 find 的方式找到 Abstract 标签，再使用 findall 的方式寻找其中所有 AbstractText 标签，获取它们的文本，并调用 wordlist_pre_process 函数进行预处理。有的 document 没有 Abstract，则 new_text 字符串为初始值空串。

创建对应的 xml 文件，将 title 和 text 按照 xml 文件的格式写入，输出处理成功信息。

若以上步骤中有地方运行出错，则进入 except 异常，输出处理失败信息，无视该 document 文件。

```
85 for document_file in document_files:
86     try:
87         root = ET.parse(document_path + '/' + folder + '/' + document_file)
88         tmp1 = root.find('MedlineCitation').find('Article')
89         title = tmp1.find('ArticleTitle').text
90         new_title = wordlist_pre_process(title)
91         new_text = ''
92         tmp2 = tmp1.find('Abstract')
93         if tmp2 != None:
94             tmp3 = [f.text for f in tmp2.findall('AbstractText')]
95             text = ' '.join(tmp3)
96             new_text = wordlist_pre_process(text)
97         file1 = open('../' + data + '/document/' + folder + '/' + document_file + '.xml', 'w')
98         file1.write('<document>\n<title>' + new_title + '</title>\n<text>' + new_text + '</text>\n</document>')
99         file1.close()
100         print(folder, document_file, 'process succeed')
101     except:
102         print(folder, document_file, 'process fail')
```

跳转回主函数，对训练集建立索引，调用 os.walk()方法获取训练集 topic 列表。

```
548     # 训练集建立索引
549     for a in os.walk('../training/document'):
550         train_folder_list = a[1]
551         break
```

对于每一个 topic, 调用 document_index_create 函数建立其索引文件。

```
553     for folder in train_folder_list:
554         document_index_create('training', folder)
```

跳转到 document_index_create 函数, 创建存放当前 topic 的索引文件夹。

```
104 def document_index_create(data, folder):
105     if (not (os.path.exists('./' + data + '/index_file/' + folder))):
106         os.mkdir('./' + data + '/index_file/' + folder)
```

document_len_map 存放当前 topic 下每个 document 的长度, total_file_num 为 document 文件总数, block_num 为分块块数。

document_len_map 的字典结构为：

{document_pid: 文档长度, ...}

```
108     document_path = './' + data + '/document/' + folder
109     document_len_map = {}
110     total_file_num = 0
111     block_num = 0
112     start = 0
```

遍历当前 topic 的所有 document 文件, 按每 10000 个文件为一块进行分块处理, 最后不足 10000 个的算作一块, 调用 SPIIMI_Invert 函数, 传递参数 data 为数据类型 (训练集还是测试集), document_files[start:end] 为当前块中所有文件名列表, folder 为当前 topicid, document_len_map 记录这些文件的长度, block_num 统计块数, total_file_num 统计总词数。

```
114     for a in os.walk(document_path):
115         document_files = a[2]
116         while start < len(document_files):
117             end = start + 10000
118             if end > len(document_files):
119                 end = len(document_files)
120             document_len_map, block_num, total_file_num = SPIIMI_Invert(data, document_files[start:end], folder,
121                                                                           document_len_map, block_num, total_file_num)
122             start += 10000
123             block_num += 1
```

跳转到 **SPIMI_Invert** 函数，dictionary 字典用于存储该块中所有索引。对于块中每个文件，调用 xml.etree.ElementTree 模块打开，file_name 为当前 document 文件去除后缀的名字，即其 pid，读取并按空格分隔 title 标签和 text 标签对应的文本（如果有的话），存入 token_list 列表，document_len_map 记录当前 document 文件的长度，total_file_num 统计总词数。

```
173 def SPIMI_Invert(data, file_stream, folder, document_len_map, block_num, total_file_num):
174     dictionary = {}
175     for file in file_stream:
176         try:
177             tree = ET.parse('./' + data + '/document/' + folder + '/' + file)
178             root = tree.getroot()
179             file_name = file[:-4]
180
181             token_list = re.split('[ ]+', root.find('title').text)
182             tmp1 = root.find('text').text
183             if tmp1 != None:
184                 token_list += re.split('[ ]+', tmp1)
185             document_len_map[file_name] = len(token_list)
186             total_file_num += document_len_map[file_name]
```

dictionary 字典结构如下：（在未计算出词频时，词频位置先记录词数）

{token: {document_pid: 词频, ...}}

遍历当前文件中所有 token，如果 token 不在 dictionary 字典中，就将其添加到字典中，对应 document 文件的词频初始化为 1。

如果 token 在 dictionary 字典中，则判断 document 的 pid 是否在该 token 对应的字典中。若不在，初始化词频为 0。document 对应的词数加 1。

```
188         for i in range(len(token_list)):
189             if token_list[i] not in dictionary.keys():
190                 dictionary[token_list[i]] = {file_name: 1}
191             else:
192                 if file_name not in dictionary[token_list[i]].keys():
193                     dictionary[token_list[i]][file_name] = 0
194                     dictionary[token_list[i]][file_name] += 1
```

若以上步骤中有地方运行出错，则进入 except 异常，删除该 document 文件。

```
195     except:
196         os.remove('./' + data + '/document/' + folder + '/' + file)
```

把该块的索引分成 27 个索引，分别为 26 个英文字母开头的 token 和其他 token 对应的索引，即每个块生成 27 个索引文件。

```
198     token_dict_tmp = {'a': {}, 'b': {}, 'c': {}, 'd': {}, 'e': {}, 'f': {}, 'g': {},
199                       'h': {}, 'i': {}, 'j': {}, 'k': {}, 'l': {}, 'm': {}, 'n': {},
200                       'o': {}, 'p': {}, 'q': {}, 'r': {}, 's': {}, 't': {},
201                       'u': {}, 'v': {}, 'w': {}, 'x': {}, 'y': {}, 'z': {}, '_other': {}}
```

遍历 dictionary 字典中所有 token，对于每个 token 对应的各个文件中的词数，将其替换成词频。

如果当前 token 为空字符串，则添加到 token_dict_tmp['_other']对应的字典中。

否则，判断 token 的第一个字符为哪个英文字母，就添加到哪个英文字母对应的字典中去，若 token 的第一个字符不为英文字母，则添加到'_other'对应的字典中。

```
203 for token in dictionary.keys():
204     for file_name in dictionary[token].keys():
205         dictionary[token][file_name] = dictionary[token][file_name] / document_len_map[file_name] * 1.0
206     if token == '':
207         token_dict_tmp['_other'][token] = dictionary[token]
208     elif token[0] in token_dict_tmp.keys():
209         token_dict_tmp[token[0]][token] = dictionary[token]
210     else:
211         token_dict_tmp['_other'][token] = dictionary[token]
```

此时 27 个索引的字典已经建立完毕，遍历这 27 个字典，对每个字典按照 key 进行升序排序，创建对应的 json 文件，使用 json.dump 将 27 个字典分别写入对应的文件中，并 print 成功信息，最后返回文件长度字典 document_len_map、块数 block_num、文件总词数 total_file_num。

```
213 for key in token_dict_tmp.keys():
214     token_dict_tmp[key] = sort_dictionary(token_dict_tmp[key])
215     file1 = open('./' + data + '/index_file/' + folder + '/index_' + key + '_' + str(block_num) + '.json', 'w')
216     json.dump(token_dict_tmp[key], file1)
217     file1.close()
218     print(folder, 'index_' + key + '_' + str(block_num), 'succeed')
219
220 return document_len_map, block_num, total_file_num
```

字典排序函数 sort_dictionary 如下，先对字典中的 key 值进行升序排序，再创建一个新的字典，将未排序的字典按照排序后的 key 值，对应复制到新字典中，最后返回新字典。

```
166 def sort_dictionary(unsort_dict):
167     keys = sorted(unsort_dict.keys())
168     sort_dict = {}
169     for key in keys:
170         sort_dict[key] = unsort_dict[key]
171     return sort_dict
```

跳转回 document_index_create 函数，输出得到的总块数 block_num。

```
125 print(folder, 'block_num =', str(block_num))
```

把每块得到的相同字母开头（或其他符号）的索引合并到一起。index_dict 字典用于合并索引时存储。

```
127 index_dict = {}
```


遍历生成的所有索引，因为索引在文件夹中按名称排序，所以将 index_files 中每 block_num 个索引合并到一起。count 用于帮助统计读入的索引是否达到 block_num 个。

对于每个索引文件，使用 json.load() 读入文件中的字典，若 count 是 block_num 的倍数，则将读入的字典拷贝到 index_dict 中。

否则，对于读入的字典中的每个 key，若 key 不在 index_dict 的 keys 中，则将其添加到 index_dict 中；否则，更新 index_dict 中该 key 对应的文档名及词频。

输出当前索引加载成功信息，再将其删除，减少不必要的存储空间。

```
129 for b in os.walk('./' + data + '/index_file/' + folder):
130     index_files = b[2]
131     count = len(index_files)
132     for index_name in index_files:
133         file1 = open('./' + data + '/index_file/' + folder + '/' + index_name, 'r')
134         dict_tmp = json.load(file1)
135         file1.close()
136         if count % block_num == 0:
137             index_dict = dict_tmp.copy()
138             count -= 1
139         else:
140             for word in dict_tmp.keys():
141                 if word not in index_dict.keys():
142                     index_dict[word] = dict_tmp[word]
143                 else:
144                     index_dict[word].update(dict_tmp[word])
145             count -= 1
146         print(folder, index_name[:-5], 'load succeed')
147         os.remove('./' + data + '/index_file/' + folder + '/' + index_name)
```

若 count 是 block_num 的倍数，则说明合并的索引文件已达 block_num 个，对 index_dict 字典按照 key 进行升序排序，创建对应的 json 文件，使用 json.dump 将 index_dict 字典写入对应的文件中，并输出成功信息。

```
149 if count % block_num == 0:
150     index_dict = sort_dictionary(index_dict)
151     file2 = open('./' + data + '/index_file/' + folder + '/' + index_name[:-7] + '.json', 'w')
152     json.dump(index_dict, file2)
153     file2.close()
154     print(folder, index_name[:-7] + ' succeed')
```

将文档长度字典 document_len_map 和文档总词数 total_file_num 及文档总数量存储到相应的文件中，便于之后计算特征，输出存储成功信息。

```
158 file3 = open('./' + data + '/document_len_map/document_len_map_' + folder + '.json', 'w')
159 json.dump(document_len_map, file3)
160 file3.close()
161 file4 = open('./' + data + '/document_total/document_total_' + folder + '.txt', 'w')
162 file4.write('total_file_num = ' + str(total_file_num) + '\ntotal_file_len = ' + str(len(document_len_map)) + '\n')
163
164 print(folder, 'document_len_map total_file_num total_file_len succeed')
```

跳转回主函数，对于每一个 topic，调用 calculate_features 函数计算其特征。

```
556 # 训练集计算特征
557 for folder in train_folder_list:
558     calculate_features('training', folder)
```

跳转到 `calculate_features` 函数，创建用于存储特征的文件夹。

```
222 def calculate_features(data, folder):
223     if (not (os.path.exists('./' + data + '/features/' + folder))):
224         os.mkdir('./' + data + '/features/' + folder)
```

打开并加载文档长度字典 `document_len_map`、文档总词数 `total_file_num`、文档总数量 `total_file_len`，并计算文档平均长度 `avg_file_len`。

```
226 file1 = open('./' + data + '/document_len_map/document_len_map_' + folder + '.json', 'r')
227 docno_dict = json.load(file1)
228 file1.close()
229
230 file2 = open('./' + data + '/document_total/document_total_' + folder + '.txt', 'r')
231 lines = file2.readlines()
232 file2.close()
233 total_file_num = int(lines[0].strip('\n')[17:])
234 total_file_len = int(lines[1].strip('\n')[17:])
235 avg_file_len = total_file_len / total_file_num * 1.0
```

读入 `query` 查询文件的 token 列表 `query_list` 和待排序文档 `pid` 列表 `doc_list`。

```
237 file6 = open('./' + data + '/query/' + folder + '.title')
238 tmp1 = file6.read()
239 file6.close()
240 query_list = re.split(' ', tmp1.strip('\n'))
241
242 file7 = open('./' + data + '/query/' + folder + '.pids')
243 tmp2 = file7.readlines()
244 file7.close()
245 doc_list = [pid.strip('\n') for pid in tmp2]
```

`k` 和 `b` 是在 BM25 方法中用来归一化约束的，防止某个词的词频过大，在这里取 `k` 为 1.5，`b` 为 0.75，并创建 TF-IDF、BM25、VSM 三种不同排序方式的结果文件。

```
247 k = 1.5
248 b = 0.75
249 file3 = open('./' + data + '/features/' + folder + '/TF-IDF.res', 'w')
250 file4 = open('./' + data + '/features/' + folder + '/BM25.res', 'w')
251 file5 = open('./' + data + '/features/' + folder + '/VSM.res', 'w')
```


query_tf 字典用于记录 query_list 中各 token 的出现次数，score_tf_idf、score_bm25、score_VSM 三个字典用于存放当前 topicid 下三种方法各自的不同文档的得分，doc_w 字典用于记录该 token 在该 document 中的 tf-idf 值，其键的值为文档 pid。对于待排序文档列表 doc_list 中每个文档，初始化其得分为 0。

```
253     query_tf = {}
254     score_tf_idf = {}
255     score_bm25 = {}
256     score_VSM = {}
257     doc_w = {}
258     for doc in doc_list:
259         score_tf_idf[doc] = 0.0
260         score_bm25[doc] = 0.0
261         score_VSM[doc] = 0.0
262         doc_w[doc] = {}
```

遍历 query_list 中的每个 token，query_tf 字典做统计，调用 query_index 函数，获取当前 token 即 query_list[i] 在索引中的 value 值。

```
264     for i in range(len(query_list)):
265         if query_list[i] not in query_tf:
266             query_tf[query_list[i]] = 1
267         else:
268             query_tf[query_list[i]] += 1
269
270     token_dict = query_index(query_list[i], data, folder)
```

跳转到 query_index 函数，根据参数 token 的首字母，判断应该调用哪个索引文件，再读取该索引，返回该 token 在索引中的 value 值，返回的字典结构如下：

{document_pid: 词频, ...}

若该 token 并不在索引中，则返回 -1。

```
330 def query_index(token, data, folder):
331     if token[0] >= 'a' and token[0] <= 'z':
332         path = './' + data + '/index_file/' + folder + '/index_' + token[0] + '.json'
333     else:
334         path = './' + data + '/index_file/' + folder + '/index__other.json'
335     file1 = open(path, 'r')
336     index_dict = json.load(file1)
337     file1.close()
338     if token in index_dict.keys():
339         return index_dict[token]
340     else:
341         return -1
```

跳转回 `calculate_features` 函数，若 `token_dict` 为 -1，则说明该 `token` 并不在索引中，`continue` 直接进入下一次循环。

否则，计算 TD-IDF 和 BM25 两种方法中需要用到的 `idf` 值。

对于包含该 `token` 的所有文档，判断该文档 `pid` 是否在待排序列表中，若在，则分别计算并叠加 TF-IDF 和 BM25 两种方法的 `score` 得分。将对应 `document` 的词频 `tf` 值存入 `doc_w` 字典中，这里 `doc_w` 字典的键的值为文档 `pid`。

```
271     if token_dict == -1:
272         continue
273
274     idf_t = math.log(total_file_num / len(token_dict) * 1.0)
275     idf_qi = math.log((total_file_num - len(token_dict) + 0.5) / (len(token_dict) + 0.5) * 1.0)
276
277     for docno in token_dict.keys():
278         if docno in doc_list:
279             score_tf_idf[docno] += token_dict[docno] * idf_t
280             score_bm25[docno] += idf_qi * (token_dict[docno] * (k + 1) / (
281                 token_dict[docno] + k * (1 - b + b * docno_dict[docno] / avg_file_len * 1.0)) * 1.0)
282             doc_w[docno][query_list[i]] = token_dict[docno]
```

使用 `query_tf` 字典中的出现次数计算得到 `query` 向量。

```
284     query_vector = []
285     for i in range(len(query_list)):
286         query_vector.append(query_tf[query_list[i]] / len(query_list) * 1.0)
```

对于 `doc_w` 字典中的所有候选文档，计算各自的 `document` 向量。`if_0` 变量初始化为 0。

遍历 `query_list` 中所有 `token`，若当前 `document` 包含当前 `token`，则在 `document` 向量中存入对应的 `tf` 值，`if_0` 赋值为 1；若不包含，则在 `document` 向量中存入 0。

判断 `if_0` 是否为 0，若为 0，则说明该 `document` 向量的所有参数都为 0，那么得分置为 0；若不为 0，则调用 `conine_score` 函数计算其与 `query` 向量的相似度，并得到该文档的得分。

```
288     for docno in doc_w.keys():
289         doc_vector = []
290         if_0 = 0
291         for i in range(len(query_list)):
292             if query_list[i] in doc_w[docno]:
293                 if_0 = 1
294                 doc_vector.append(doc_w[docno][query_list[i]])
295             else:
296                 doc_vector.append(0)
297         if if_0 == 0:
298             score_VSM[docno] = 0.0
299         else:
300             score_VSM[docno] = conine_score(query_vector, doc_vector)
```

跳转到 **conine_score** 函数，计算两个向量的余弦相似度并返回给 score_VSM 字典。

```
343 def conine_score(u, v):
344     fenzi = 0
345     fenmu1 = 0
346     fenmu2 = 0
347     for i in range(len(u)):
348         fenzi += u[i] * v[i]
349         fenmu1 += u[i] * u[i]
350         fenmu2 += v[i] * v[i]
351     return fenzi / (math.sqrt(fenmu1) * math.sqrt(fenmu2)) * 1.0
```

跳转回 **calculate_features** 函数，按照得分降序的顺序，对当前 topicid 下三种方法各自的结果进行排序，得到各自排序后的元组列表结果。

```
302 sort_score_tf_idf = sorted(score_tf_idf.items(), key=lambda x: x[1], reverse=True)
303 sort_score_bm25 = sorted(score_bm25.items(), key=lambda x: x[1], reverse=True)
304 sort_score_VSM = sorted(score_VSM.items(), key=lambda x: x[1], reverse=True)
```

按照 DOC_PID RANK SCORE 的格式，将结果写入对应的结果文件，并同时输出到控制台，rank 从 1 开始排名。

如：4278185 1 5.7910793023957945

```
306 rank = 1
307 for item in sort_score_tf_idf:
308     print('TF-IDF ' + item[0] + ' ' + str(rank) + ' ' + str(item[1]))
309     file3.write(item[0] + ' ' + str(rank) + ' ' + str(item[1]) + '\n')
310     rank += 1
311
312 rank = 1
313 for item in sort_score_bm25:
314     print('BM25 ' + item[0] + ' ' + str(rank) + ' ' + str(item[1]))
315     file4.write(item[0] + ' ' + str(rank) + ' ' + str(item[1]) + '\n')
316     rank += 1
317
318 rank = 1
319 for item in sort_score_VSM:
320     print('VSM ' + item[0] + ' ' + str(rank) + ' ' + str(item[1]))
321     file5.write(item[0] + ' ' + str(rank) + ' ' + str(item[1]) + '\n')
322     rank += 1
```

关闭三个文件指针，并输出成功信息。

```
324 file3.close()
325 file4.close()
326 file5.close()
327
328 print(folder, 'calculate features succeed')
```

跳转回主函数，对于每一个 topic，调用 combine_features 函数将三种方法得到的特征合并。

```
560 # 训练集合并特征
561 for folder in train_folder_list:
562     combine_features('training', folder)
```

跳转到 `combine_features` 函数，对于当前 `topicid`，打开并读入相应的三种特征文件。

```
353 def combine_features(data, folder):
354     file1 = open('./' + data + '/features/' + folder + '/TF-IDF.res', 'r')
355     tmp1 = file1.readlines()
356     file1.close()
357     file2 = open('./' + data + '/features/' + folder + '/BM25.res', 'r')
358     tmp2 = file2.readlines()
359     file2.close()
360     file3 = open('./' + data + '/features/' + folder + '/VSM.res', 'r')
361     tmp3 = file3.readlines()
362     file3.close()
```

`Features_dict` 存储每个文档 `pid` 对应的特征列表。

```
364 features_dict = {}
365 for i in range(len(tmp1)):
366     tmp1_list = re.split(' ', tmp1[i].strip('\n'))
367     tmp2_list = re.split(' ', tmp2[i].strip('\n'))
368     tmp3_list = re.split(' ', tmp3[i].strip('\n'))
369     if tmp1_list[0] not in features_dict.keys():
370         features_dict[tmp1_list[0]] = [tmp1_list[1], tmp1_list[2], 0, 0, 0, 0]
371     else:
372         features_dict[tmp1_list[0]][0] = tmp1_list[1]
373         features_dict[tmp1_list[0]][1] = tmp1_list[2]
374     if tmp2_list[0] not in features_dict.keys():
375         features_dict[tmp2_list[0]] = [0, 0, tmp2_list[1], tmp2_list[2], 0, 0]
376     else:
377         features_dict[tmp2_list[0]][2] = tmp2_list[1]
378         features_dict[tmp2_list[0]][3] = tmp2_list[2]
379     if tmp3_list[0] not in features_dict.keys():
380         features_dict[tmp3_list[0]] = [0, 0, 0, 0, tmp3_list[1], tmp3_list[2]]
381     else:
382         features_dict[tmp3_list[0]][4] = tmp3_list[1]
383         features_dict[tmp3_list[0]][5] = tmp3_list[2]
```

对于每一个文档的特征，按照 `DOC_PID` `TF-IDF_RANK` `TF-IDF_SCORE` `BM25_RANK` `BM25_SCORE` `VSM_RANK` `VSM_SCORE` 的格式，写入对应文件中。

如：8809426 1328 0.04900872034820591 1383 3.903382449109586e-06 1283
0.40824829046386296

关闭文件指针，并输出成功信息。

```
385 file4 = open('./' + data + '/total_features/' + folder, 'w')
386 for key in features_dict.keys():
387     file4.write(key + ' ' + features_dict[key][0] + ' ' + features_dict[key][1] + ' ' + features_dict[key][2]
388     + ' ' + features_dict[key][3] + ' ' + features_dict[key][4] + ' ' + features_dict[key][5] + '\n')
389 file4.close()
390 print(folder, 'features combine succeed')
```

跳转回主函数，调用 `get_label` 函数从训练集标准答案中获取相关性标签。

```
564 # 训练集获得01标签
565 get_label('training')
```

跳转到 get_label 函数, 打开并读入训练集标准答案, label_dict 字典用于存储每个 topicid 对应的文档 pid 和 01 相关性。

```
392 def get_label(data):
393     file1 = open('../2017TAR/' + data + '/qrels/qrel_abs_train', 'r')
394     lines = file1.readlines()
395     label_dict = {}
396     for line in lines:
397         tmp1 = line.strip('\n').strip(' ')
398         tmp2 = re.split('[ ]+', tmp1)
399         if tmp2[0] not in label_dict.keys():
400             label_dict[tmp2[0]] = []
401             label_dict[tmp2[0]].append(tmp2[2] + ' ' + tmp2[3] + '\n')
```

对于 label_dict 里的每个 topicid, 创建并写入相应的文件, 输出成功信息。

```
403     for topic in label_dict.keys():
404         file2 = open('../' + data + '/labels/' + topic, 'w')
405         for label in label_dict[topic]:
406             file2.write(label)
407         file2.close()
408         print(topic, 'get label succeed')
```

跳转回主函数, 对于所有 topicid, 调用 logistic 函数进行逻辑回归, 得到打分函数。

```
567     # 训练集逻辑回归
568     logistic('training', train_folder_list)
```

跳转到 logistic 函数, dataX 用于存储特征, dataY 用于存储标签, 对于所有文档, 按照每行的顺序, 将每行的 6 个特征添加到 dataX 的一行中, 将每行的相关性标签添加到 dataY 的一行中。

```
410 def logistic(data, folder_list):
411     dataX = []
412     dataY = []
413
414     for folder in folder_list:
415         file1 = open('../' + data + '/total_features/' + folder, 'r')
416         TrainLines = file1.readlines()
417         file1.close()
418         for line in TrainLines:
419             tmp1 = re.split(' ', line.strip('\n'))
420             tmp1 = tmp1[1:]
421             tmp2 = [float(elem) for elem in tmp1]
422             dataX.append(tmp2)
423
424         file2 = open('../' + data + '/labels/' + folder, 'r')
425         lines = file2.readlines()
426         file2.close()
427         for line in lines:
428             tmp3 = re.split(' ', line.strip('\n'))
429             dataY.append(float(tmp3[1]))
```

alpha 为设置的梯度的阈值，count 为迭代次数，损失函数 loss_function 初始化为 0，data_num 是训练数据总行数，x_length 是训练数据总列数，即特征个数。

```
431     alpha = 0.0001
432     count = 2000
433     loss_function = 0
434     data_num = len(dataX)
435     x_length = len(dataX[0])
```

计算每列数据的最大值和最小值，对每列数据进行归一化。

```
437     max_data = []
438     min_data = []
439     for i in range(x_length):
440         max_data.append(max([line[i] for line in dataX]))
441         min_data.append(min([line[i] for line in dataX]))
442
443     for line in dataX:
444         for i in range(x_length):
445             line[i] = line[i] / (max_data[i] - min_data[i]) * 1.0
```

theta 权重都初始化为 1，并输出相应成功信息。

```
447     theta = [1 for i in range(x_length)]
448
449     print('train data process succeed, start calculate theta')
```

进行多次迭代，更新权重。

```
451     while True:
452         for i in range(data_num):
453             thetaTx = 0
454             for j in range(x_length):
455                 thetaTx += theta[j] * dataX[i][j]
456             h_theta = 1. / (1 + math.exp(-thetaTx))
457             for j in range(x_length):
458                 theta[j] += alpha * (dataY[i] - h_theta) * dataX[i][j]
459
460         for i in range(data_num):
461             thetaTx = 0
462             for j in range(x_length):
463                 thetaTx += theta[j] * dataX[i][j]
464             h_theta = 1. / (1 + math.exp(-thetaTx))
465             if h_theta < 1:
466                 loss_function += dataY[i] * math.log(abs(h_theta)) + (1 - dataY[i]) * math.log(abs(1 - h_theta))
467         loss_function = - loss_function / data_num * 1.0
468
469         count -= 1
470         print('count =', count)
471         if count == 0:
472             break
```


输出 theta 到控制台，创建并打开相应的文件，将 theta 权重写入，并输出逻辑回归成功信息。

```
474     print(theta)
475     file3 = open('./' + data + '/theta.predict', 'w')
476     for i in range(x_length):
477         if i != x_length - 1:
478             file3.write(str(theta[i]) + ' ')
479         else:
480             file3.write(str(theta[i]) + '\n')
481     file3.close()
482     print('logistic regressor succeed')
```

跳转回主函数，对测试集进行相同的操作：创建文件夹、预处理、建立索引、计算特征、合并特征。

```
570     # 创建测试集所需文件夹
571     create_necessary_folders('testing')
572
573     # 测试集预处理
574     preprocess_query('testing')
575     preprocess_document('testing')
576
577     # 测试集建立索引
578     for a in os.walk('./testing/document'):
579         test_folder_list = a[1]
580         break
581
582     for folder in test_folder_list:
583         document_index_create('testing', folder)
584
585     # 测试集计算特征
586     for folder in test_folder_list:
587         calculate_features('testing', folder)
588
589     # 测试集合并特征
590     for folder in test_folder_list:
591         combine_features('testing', folder)
```

对于测试集中每个 topic，调用 prediction 函数进行预测并重新排序。

```
593     # 测试集预测
594     for folder in test_folder_list:
595         prediction('training', 'testing', folder)
```

跳转到 **prediction 函数**，pids 用于存储待排序文档 pid，testX 用于存储特征，testY 用于存储标签，对于所有文档，按照每行的顺序，将每行的文档 pid 添加到 pids 的一行中，将每行的 6 个特征添加到 testX 的一行中。

```
484 def prediction(train, test, folder):
485     pids = []
486     testX = []
487     testY = []
488     file1 = open('./' + test + '/total_features/' + folder, 'r')
489     TestLines = file1.readlines()
490     file1.close()
491     for line in TestLines:
492         tmp1 = re.split(' ', line.strip('\n'))
493         pids.append(tmp1[0])
494         tmp1 = tmp1[1:]
495         tmp2 = [float(elem) for elem in tmp1]
496         testX.append(tmp2)
```

test_num 是测试数据总行数，x_length 是测试数据总列数，即特征个数。

```
498     test_num = len(testX)
499     x_length = len(testX[0])
```

计算每列数据的最大值和最小值，对每列数据进行归一化。

```
501     max_testdata = []
502     min_testdata = []
503     for i in range(x_length):
504         max_testdata.append(max([line[i] for line in testX]))
505         min_testdata.append(min([line[i] for line in testX]))
506
507     for line in testX:
508         for i in range(x_length):
509             line[i] = line[i] / (max_testdata[i] - min_testdata[i]) * 1.0
```

打开并读入 theta 权重函数，根据 theta 计算得到每行对应的阈值 h_theta，添加到 testY 中。

```
511     file2 = open('./' + train + '/theta.predict', 'r')
512     tmp3 = file2.read()
513     file2.close()
514     tmp4 = re.split(' ', tmp3.strip('\n'))
515     theta = [float(elem) for elem in tmp4]
516
517     for i in range(test_num):
518         thetaTx = 0
519         for j in range(x_length):
520             thetaTx += theta[j] * testX[i][j]
521         h_theta = 1. / (1 + math.exp(-thetaTx))
522         testY.append(h_theta)
```

创建 predict_tuples 列表，每行存储一个 (topicid, 文档 pid, 相关性概率) 元组，并按照概率降序对整个列表进行排序。

```
524 predict_tuples = []
525 for i in range(test_num):
526     predict_tuples.append((folder, pids[i], testY[i]))
527 sort_predict_tuples = sorted(predict_tuples, key=lambda x: x[2], reverse=True)
```

创建相应的结果文件。

按照 TOPIC_ID Q0 DOC_ID RANK SCORE RUN_ID 的格式, 将结果写入对应的结果文件, 并同时输出到控制台, rank 从 1 开始排名, 最后输出成功信息。

如: CD007431 0 4278185 1 0.04060672446668173 10152130122_point-wise

```
529 file3 = open('./' + test + '/10152130122_钱庭涵_point-wise.res', 'a')
530 rank = 1
531 for i in range(test_num):
532     print(sort_predict_tuples[i][0], '0', sort_predict_tuples[i][1], str(rank), str(sort_predict_tuples[i][2]),
533           '10152130122_point-wise')
534     file3.write(sort_predict_tuples[i][0] + ' 0 ' + sort_predict_tuples[i][1] + ' ' + str(rank) + ' ' + str(
535         sort_predict_tuples[i][2]) + ' 10152130122_point-wise\n')
536     rank += 1
537 file3.close()
538 print(folder, 'prediction succeed')
```

Result :

对测试集文档 pid 重新排序后的结果为 10152130122_钱庭涵_point-wise.res

 10152130122_钱庭涵_point-wise.res 2018/7/1 20:59 Compiled Resou... 8,124 KB

```
10152130122_钱庭涵_point-wise.res x
1  CD007431 0 1826546 1 0.004733930597254791 10152130122_point-wise
2  CD007431 0 10537383 2 0.004203739942664555 10152130122_point-wise
3  CD007431 0 6617177 3 0.004070059292182798 10152130122_point-wise
4  CD007431 0 15830972 4 0.004063654012957143 10152130122_point-wise
5  CD007431 0 2975062 5 0.004045838863799197 10152130122_point-wise
6  CD007431 0 6458028 6 0.00393594875225486 10152130122_point-wise
7  CD007431 0 1670454 7 0.003908479838099452 10152130122_point-wise
8  CD007431 0 8297630 8 0.0037642617574479963 10152130122_point-wise
9  CD007431 0 2940266 9 0.003667102410306726 10152130122_point-wise
10 CD007431 0 8124277 10 0.003607826912670675 10152130122_point-wise
```

evaluation.py

主函数中，打开并读入测试集标准答案，rele_dict 用于存储每个 topicid 对应的相关文档总数和相关文档列表，字典结构为：

`{topicid: {'rele_num': 相关文档个数, 'rele_list': [pid1, pid2, ...]}, ...}`

同时按照旧的评测程序所需的标准答案格式，生成旧的格式的标准答案 qrels_testing.res，以便于使用旧的评测程序进行评测。

```
52 if __name__ == '__main__':
53     qrel_path = '../2017_test_qrels/qrel_abs_test.txt'
54     res_path = './testing/10152130122_钱庭涵_point-wise.res'
55
56     file1 = open(qrel_path, 'r')
57     lines = file1.readlines()
58     file1.close()
59     file0 = open('./qrels_testing.res', 'w')
60     rele_dict = {}
61     for line in lines:
62         tmp1 = line.strip('\n').strip(' ')
63         tmp2 = re.split(' ', tmp1)
64         file0.write(tmp2[0] + ' ' + tmp2[1] + ' ' + tmp2[2] + ' ' + tmp2[3] + '\n')
65         if tmp2[0] not in rele_dict.keys():
66             rele_dict[tmp2[0]] = {'rele_num': 0, 'rele_list': []}
67         if int(tmp2[3]) == 1:
68             rele_dict[tmp2[0]]['rele_num'] += 1
69             rele_dict[tmp2[0]]['rele_list'].append(tmp2[2])
70     file0.close()
71     print('qrel dictionary succeed')
```

打开并读入测试集重排后的结果，res_dict 用于存储每个 topicid 对应的文档 pid、文档排名、文档相关性得分，字典结构为：

`{topicid: {'文档名': 'rank': 排名, 'rate': 得分}, ...}, ...}`

```
73     file2 = open(res_path, 'r')
74     lines = file2.readlines()
75     file2.close()
76     res_dict = {}
77     for line in lines:
78         tmp1 = line.strip('\n')
79         tmp2 = re.split(' ', tmp1)
80         if tmp2[0] not in res_dict.keys():
81             res_dict[tmp2[0]] = {}
82             res_dict[tmp2[0]][tmp2[2]] = {'rank': int(tmp2[3]), 'rate': float(tmp2[4])}
83     print('result dictionary succeed')
```

调用 eval_map 函数计算 MAP 指标并输出。

```
85     MAP = eval_map(rele_dict, res_dict)
86     print('MAP =', MAP)
```

跳转到 **eval_map** 函数，计算每个 topic 的 MAP 指标，返回所有 MAP 的平均值。

```
6  def eval_map(rele_dict, res_dict):
7      map_list = []
8      for topic in rele_dict.keys():
9          map = 0
10         count = 0
11         for i in range(len(rele_dict[topic]['rele_list'])):
12             pid = rele_dict[topic]['rele_list'][i]
13             if pid in res_dict[topic].keys():
14                 count += 1
15                 map += count / res_dict[topic][pid]['rank'] * 1.0
16             map = map / rele_dict[topic]['rele_num']
17         map_list.append(map)
18
19     avg = sum(map_list) / len(map_list) * 1.0
20     return avg
```

跳转回主函数，调用 eval_ndcg 函数，计算不同 p 值的 NDCG 评测指标并输出。

```
88     NDCG = eval_ndcg(res_dict, 10)
89     print('NDCG@10 =', NDCG)
90     NDCG = eval_ndcg(res_dict, 20)
91     print('NDCG@20 =', NDCG)
92     NDCG = eval_ndcg(res_dict, 50)
93     print('NDCG@50 =', NDCG)
94     NDCG = eval_ndcg(res_dict, 100)
95     print('NDCG@100 =', NDCG)
96     NDCG = eval_ndcg(res_dict, 200)
97     print('NDCG@200 =', NDCG)
```

在 **get_rank** 函数中，按照相关性概率从小到大，将其均分为 1~5 五个等级。

```
22  def get_rank(rate):
23      if rate < 0.2:
24          return 1
25      elif rate < 0.4:
26          return 2
27      elif rate < 0.6:
28          return 3
29      elif rate < 0.8:
30          return 4
31      else:
32          return 5
```


跳转到 `eval_ndcg` 函数, 计算每个 topic 的 NDCG 指标, 返回所有 NDCG 的平均值。


```
34 def eval_ndcg(res_dict, p):
35     NDCG_list = []
36     for topic in res_dict.keys():
37         DCG = 0
38         IDCG = 0
39         i = 1
40         for pid in res_dict[topic]:
41             rel = get_rank(res_dict[topic][pid]['rate'])
42             if i <= p:
43                 DCG += (math.pow(2, rel) - 1) / math.log(1+i, 2) * 1.0
44                 IDCG += (math.pow(2, rel) - 1) / math.log(1 + i, 2) * 1.0
45                 i += 1
46         NDCG = DCG / IDCG * 1.0
47         NDCG_list.append(NDCG)
48
49     avg = sum(NDCG_list) / len(NDCG_list) * 1.0
50     return avg
```

Result :

```
MAP = 0.2836859976102207
NDCG@10 = 0.05208310056568291
NDCG@20 = 0.08070303012690405
NDCG@50 = 0.14784750443322986
NDCG@100 = 0.2261831494789407
NDCG@200 = 0.31034843038565174
```

Trec 评测：

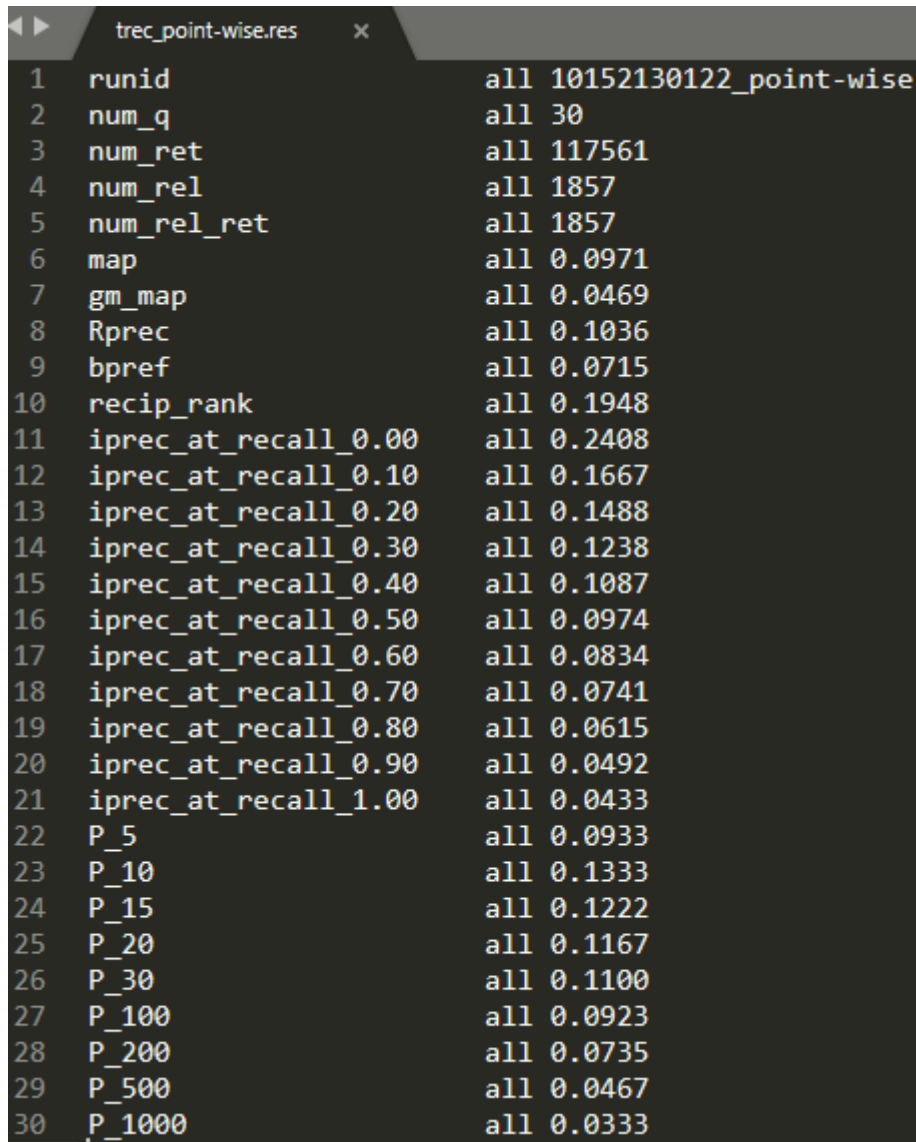
使用 trec_eval 进行评测的结果为 trec_point-wise.res

 trec_point-wise.res

2018/7/1 21:02

Compiled Resou...

2 KB



| | | |
|----|----------------------|----------------------------|
| 1 | runid | all 10152130122_point-wise |
| 2 | num_q | all 30 |
| 3 | num_ret | all 117561 |
| 4 | num_rel | all 1857 |
| 5 | num_rel_ret | all 1857 |
| 6 | map | all 0.0971 |
| 7 | gm_map | all 0.0469 |
| 8 | Rprec | all 0.1036 |
| 9 | bpref | all 0.0715 |
| 10 | recip_rank | all 0.1948 |
| 11 | iprec_at_recall_0.00 | all 0.2408 |
| 12 | iprec_at_recall_0.10 | all 0.1667 |
| 13 | iprec_at_recall_0.20 | all 0.1488 |
| 14 | iprec_at_recall_0.30 | all 0.1238 |
| 15 | iprec_at_recall_0.40 | all 0.1087 |
| 16 | iprec_at_recall_0.50 | all 0.0974 |
| 17 | iprec_at_recall_0.60 | all 0.0834 |
| 18 | iprec_at_recall_0.70 | all 0.0741 |
| 19 | iprec_at_recall_0.80 | all 0.0615 |
| 20 | iprec_at_recall_0.90 | all 0.0492 |
| 21 | iprec_at_recall_1.00 | all 0.0433 |
| 22 | P_5 | all 0.0933 |
| 23 | P_10 | all 0.1333 |
| 24 | P_15 | all 0.1222 |
| 25 | P_20 | all 0.1167 |
| 26 | P_30 | all 0.1100 |
| 27 | P_100 | all 0.0923 |
| 28 | P_200 | all 0.0735 |
| 29 | P_500 | all 0.0467 |
| 30 | P_1000 | all 0.0333 |