基于检索式的简易的问答系统

基于检索式的简易的问答系统，其中每一个样本数据是`(问题，答案)`对。系统的核心是当用户输入一个问题的时候，首先要找到跟这个问题最相近的已经存储在库里的问题，然后返回问题的索引，再根据索引找相应的答案即可。

https://zhuanlan.zhihu.com/p/165146107

任务流程

- 文本的读取： 需要从相应的文件里读取`(问题，答案)`
- 文本预处理： 清洗文本很重要，需要涉及到`停用词过滤`等工作，具体包括：

- 文本的表示： 表示一个句子非常关键，这里会涉及到`tf-idf`，`Glove`以及`BERT Embedding`
- 文本相似度匹配： 在基于检索式系统中一个核心的部分是计算文本之间的`相似度`，从而选择相似度最高的问题然后返回这些问题的答案
- 倒排表： 为了加速搜索速度，我们需要设计`倒排表`来存储每一个词与出现的文本
- 词义匹配：直接使用倒排表会忽略到一些意思上相近但不完全一样的单词，我们需要做这部分的处理。我们需要提前构建好`相似的单词`然后搜索阶段使用
- 拼写纠错：暂时不做
- 文档的排序： 最后返回结果的排序根据文档之间`余弦相似度`有关，同时也跟倒排表中匹配的单词有关

数据集

- `vocab.txt` 这里列了几万个英文常见的单词，可以用这个词库来验证是否有些单词被拼错
- `glove.6B`: 这里使用 d=200 的词向量
- `train-v2.0.json`: 这个数据包含了问题和答案的 pair， 但是以JSON格式存在，需要编写parser来提取出里面的问题和答案。

过程及代码

一、数据预处理

1.1 文本读取

读取给定的语料库，并把问题列表和答案列表分别写入到 qlist, alist 里

```
def read_corpus():
    qlist = []
    alist = []
    filename = 'train-v2.0.json'
```

```
    datas = json.load(open(filename,'r'))
    data = datas['data']
    for d in data:
        paragraph = d['paragraphs']
        for p in paragraph:
            qas = p['qas']
            for qa in qas:
                #print(qa)
                #处理is_impossible为True时answers空
                if(not qa['is_impossible']):
                    qlist.append(qa['question'])
                    alist.append(qa['answers'][0]['text'])
    assert len(qlist) == len(alist)  # 确保长度一样
    return qlist, alist
qlist,alist = read_corpus()
# print(qlist[:2], "\n", alist[:2])

# ['When did Beyonce start becoming popular?',
#  'What areas did Beyonce compete in when she was growing up?']
#  ['in the late 1990s', 'singing and dancing']
```

1.2 文本清洗

1. 停用词过滤(一般直接使用NLTK自带的);
2. 转换成小写字母;
3. 去掉一些无用的符号： 比如连续的感叹号， 或者一些奇怪的单词。
4. 对于数字的处理： 分词完只有有些单词可能就是数字比如44，415，把所有这些数字都看成是一个单词，这个新的单词我们可以定义为 "#number";
5. 去掉出现频率很低的词：比如出现次数少于10,20.... （阈值根据单词数的出现频率图来进行判断）

```
def lowerCase(ori_list):
    return [q.lower() for q in ori_list]

def tokenizer(ori_list):
    #分词时处理标点符号
    SYMBOLS = re.compile('[\s;\"\",.!?\\/\[\]\{\}\(\)-]+')
    new_list = []
    for q in ori_list:
        words = SYMBOLS.split(q.strip())
        new_list.append(' '.join(words))
    return new_list

# nltk中stopwords包含what等，但是在QA问题中，这算关键词，所以不看作关键词
# 第一次使用nltk中的stopword之前，先单独运行 nltk.download('stopwords') 来下载
def removeStopWord(ori_list):
    new_list = []
    restored = ['what','when','which','how','who','where']
    english_stop_words = list(set(stopwords.words('english')))#
    for w in restored:
        english_stop_words.remove(w)
    for q in ori_list:
        sentence = ' '.join([w for w in q.strip().split(' ') if w not in
                             english_stop_words])
        new_list.append(sentence)
    return new_list

# 根据thres筛选词表，小于thres的词去掉
```

```
def removeLowFrequence(ori_list,vocabulary,thres=3):
    new_list = []
    for q in ori_list:
        sentence = ' '.join([w for w in q.strip().split(' ') if vocabulary[w] >=
thres])
        new_list.append(sentence)
    return new_list

# 将数字统一替换,默认替换为#number
def replaceDigits(ori_list,replace = '#number'):
    DIGITS = re.compile('\d+')
    new_list = []
    for q in ori_list:
        q = DIGITS.sub(replace,q)
        new_list.append(q)
    return new_list

# 定义处理一个单词的总的函数: 使用参数来控制各项清洗功能
def handle_sentence(word_list, isLowCase=True, isStopWord=True,
isReplaceDigits=True):
    if isLowCase:
        word_list = lowerCase(word_list)
    word_list = tokenizer(word_list)
    if isStopWord:
        word_list = removeStopWord(word_list)
    if isReplaceDigits:
        word_list = replaceDigits(word_list)
    return word_list

new_qlist = handle_sentence(qlist)
print("new_qlist", new_qlist[:2])

# 清洗之前
# ['When did Beyonce start becoming popular?',
#  'What areas did Beyonce compete in when she was growing up?']
# 清洗之后
# ['when beyonce start becoming popular', 'what areas beyonce compete when
growing']
```

1.3 频率可视化

```
# 首先先来看一下总词数
word_total = list()
for q in new_qlist:
    # 这里的q就是指的每一句话
    for w in q.split(' '):
        # w指的是每一个单词
        word_total.append(w)

word_total_unique = set(word_total)
# 输出总单词数
word_total.remove('')
print("word_total: ", len(word_total))
print("word_total_unique", len(word_total_unique))

# word_total:  562229
# word_total_unique 37731
```

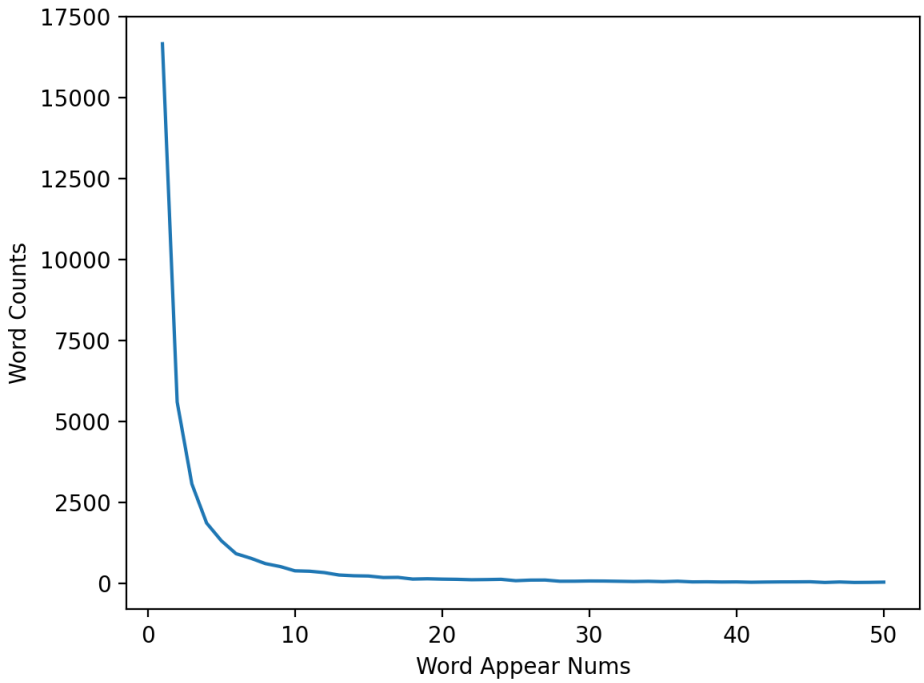
统计一下qlist中出现1次，2次，3次... 出现的单词个数，

然后画一个plot. 这里的x轴是单词出现的次数（1，2，3，..)， y轴是单词个数

```
# step1: 先统计词频
# dict_word_count:key：单词，value：词的出现次数
# 不用Counter的话，开始需要进行初始化
dict_word_count = {l:0 for l in word_total}
for value in word_total:
    dict_word_count[value] +=1

# step2: 再根据词频统计出现1,2,3...n次的单词的个数
# 需要先把set保存，以此来作为字典的key
word_count_set = sorted(list(set(dict_word_count.values())))
dict_appear_count = {s:0 for s in word_count_set}
for w, v in dict_word_count.items():
    dict_appear_count[v] += 1

# step3: 绘制出现次数的图，x轴为出现的次数；y轴为出现次数的单词数量；
x_data = list(dict_appear_count.keys())
y_data = list(dict_appear_count.values())
fig = plt.figure()  #设置画布
ax1 = fig.add_subplot(111)
# 看前50个
k = 50
plt.plot(x_data[:k], y_data[:k])
ax1.set_xlabel(u'Word Appear Nums')
ax1.set_ylabel(u'Word Counts')
plt.show()
```



由此图可以看出词库中的一些单词经常使用，而绝大部分单词不常用

由频率图来对词表进行筛选,把出现少的且数目比较大的（靠近 y 轴）的词删掉，大概为 3

二、文本表示

做完必要的文本处理之后就需要想办法表示文本了，这里有几种方式:

- 使用`tf-idf vector`
- 使用embedding技术如`word2vec`，`bert embedding`等 下面我们分别提取这三个特征来做对比。

2.1 使用 TF-IDF 制作词向量

把`qlist`中的每一个问题的字符串转换成`TF-IDF`向量, 转换之后的结果存储在`X`矩阵里。`X`的大小是：`N* D`的矩阵。 这里`N`是问题的个数（样本个数），`D`是扩展的维度。

```
vectorizer = TfidfVectorizer()  # 定一个tf-idf的vectorizer
X_tfidf = vectorizer.fit_transform(new_qlist)   # 结果存放在X矩阵
```

2.2 使用 GloVe + average_pooling

这里下载训练好的： https://nlp.stanford.edu/projects/glove/ （下载`glove.6B.zip`），并使用`d=200`的词向量（200维），通过`average pooling`来实现句子的向量。

```
def loadEmbedding(filename):
    #加载glove模型，转化为word2vec，再加载word2vec模型
    word2vec_temp_file = 'word2vec_temp.txt'
    glove2word2vec(filename,word2vec_temp_file)
    model = KeyedVectors.load_word2vec_format(word2vec_temp_file)
    return model

emb  =  loadEmbedding('glove.6B.200d.txt')

# 这是 X*D的矩阵，这里的X是词典库的大小， D是词向量的大小。
# 需要从文本中读取来给定的每个单词的词向量
# 用average pooling的方法求算每个句子的句子向量
def computeGloveSentenceEach(sentence,embedding):
    #查找句子中每个词的embedding,将所有embedding进行加和求均值
    emb = np.zeros(200)
    # 将每个句子分成单独的词
    words = sentence.strip().split(' ')
    for w in words:
        if w not in embedding:
            #没有lookup的即为unknown
            w = 'unknown'
        emb += embedding[w]
    return emb / len(words)

def computeGloveSentence(qlist,embedding):
    #对每一个句子进行求均值的embedding
    X_w2v = np.zeros((len(qlist),200))
    for i, q in enumerate(qlist):
        # 这里的q是指的每一个句子
        X_w2v[i] = computeGloveSentenceEach(q,embedding)
    return X_w2v

X_w2v =   computeGloveSentence(new_qlist, emb)
# 初始化完emb之后就可以对每一个句子来构建句子向量了，这个过程使用average pooling来实现
```

2.3 基于BERT

```
sentence_embedding = np.ones((len(qlist),768))

#加载Bert模型，model，dataset_name,须指定
bert_embedding = BertEmbedding(model='bert_12_768_12',
dataset_name='wiki_multilingual_cased')

#查询所有句子的Bert  embedding
all_embedding = bert_embedding(qlist,'sum')
for i in range(len(all_embedding)):
    sentence_embedding[i] = np.sum(all_embedding[i][1],axis = 0) /
                                                    len(q.strip().split(' '))
    if i == 0:
        print(sentence_embedding[i])

X_bert =  sentence_embedding
# 每一个句子的向量结果存放在X_bert矩阵里。行数为句子的总个数，列数为一个句子embedding大小。
```

三. 相似度匹配及搜索

在这部分里，需要把每一个输入跟知识库里的每一个问题做一个相似度计算，从而得出最相似的问题。时间复杂度其实很高，所以需要结合倒排表来获取相似度最高的问题，从而获得答案。

首先定义好返回 topk 的函数，这里用`优先队列`作为数据结构进行返回：

```
def get_least_numbers_big_data(alist, k):
    max_heap = []
    length = len(alist)
    # 当k传入的不满足范围时，返回为空
    if not alist or k <= 0 or k > length:
        return
    k -= 1
    for e in alist:
        if len(max_heap) <= k:
            heapq.heappush(max_heap, e)
        else:
            heapq.heappushpop(max_heap, e)
    return max_heap
```

3.1 TF-IDF + 余弦相似度

将输入的句子转化为 TF-IDF 向量与库中的句子做余弦相似度匹配，最后返回topk。

```
def get_top_results_tfidf_noindex(query):
    # TODO 需要编写
    """
    给定用户输入的问题 query，返回最有可能的TOP 5问题。这里面需要做到以下几点：
    1. 对于用户的输入 query 首先做一系列的预处理(上面提到的方法)，然后再转换成tf-idf向量（利用上面的vectorizer)
    2. 计算跟每个库里的问题之间的相似度
    3. 找出相似度最高的top5问题的答案
    """
    input_seq = query
    input_vec = vectorizer.transform([input_seq])
```

```
    result = list(cosine_similarity(input_vec, X_tfidf)[0])
    top_values = sorted(get_least_numbers_big_data(result, 5), reverse=True)

    top_idxs = []
    len_result = len(result)
    dict_visited = {}
    for value in top_values:
        for index in range(len_result):
            if value == result[index] and index not in dict_visited:
                top_idxs.append(index)
                dict_visited[index] = True

    top_idxs = top_idxs[:5]

    return [alist[i] for i in top_idxs]  # 返回相似度最高的问题对应的答案，作为TOP5答案


# 测试
# line = 'hello world'
# get_top_results_tfidf_noindex(line)
```

因为循环了所有库里的query，速度会比较慢，为了优化这个过程，可使用一种数据结构叫做`倒排表`。使用倒排表我们可以把单词和出现这个单词的文档做关键。 之后假如要搜索包含某一个单词的文档，即可以非常快速的找出这些文档。 在这个QA系统上，我们首先使用倒排表来快速查找包含至少一个单词的文档，然后再进行余弦相似度的计算，即可以大大减少`时间复杂度`。

3.2 创建倒排序

```
'''
创建字典有两种方式--
1. 一种是开始就初始化，然后直接添加：inverted_idx =  {value:[] for value in
word_total_unique};
2. 另一种是，开始只定义dict()，然后通过if判断是否在词典中，在的话+1，否则赋值；
'''
inverted_idx = {word:[] for word in word_total_unique}
# 定一个一个简单的倒排表，是一个map结构。 循环所有qlist一遍就可以
for index, sentence in enumerate(new_qlist):
    for word in sentence.strip().split():
        inverted_idx[word].append(index)
inverted_idx.pop('')
# print(inverted_idx)
```

3.3 语义相似度

两个单词比如car, auto这两个单词长得不一样，但从语义上还是类似的。如果只是使用倒排表我们不能考虑到这些单词之间的相似度，这就导致如果搜索句子里包含了`car`, 则我们没法获取到包含auto的所有的文档。所以我们希望把这些信息也存下来。那这个问题如何解决呢？ 其实也不难，可以提前构建好相似度的关系，比如对于`car`这个单词，一开始就找好跟它意思上比较类似的单词比如top 10，这些都标记为`related words`。所以最后就可以创建一个保存`related words`的一个`map`. 比如调用`related_words['car']`即可以调取出跟`car`意思上相近的TOP 10的单词。

那这个`related_words`又如何构建呢？ 在这里我们仍然使用`Glove`向量，然后计算一下俩俩的余弦相似度。之后对于每一个词，存储跟它最相近的top 10单词，最终结果保存在`related_words`里面。 这个计算需要发生在离线，因为计算量很大，复杂度为`O(V*V)`， V是单词的总数。代码放在`related.py`的文件里，然后结果保存在`related_words.txt`里。 在使用的时候直接从文件里读取就可以了，不用再重复计算。

```
# 读取语义相关的单词
def get_related_words(file):
    dict_related = {}
    for line in open(file, mode='r', encoding='utf-8'):
        item = line.split(",")
        word, si_list = item[0], [value for value in item[1].strip().split()]
        dict_related[word] = si_list
    return dict_related

related_words = get_related_words('related_words.txt')
# 直接放在文件夹的根目录下，不要修改此路径。
```

3.4 利用倒排表搜索

使用倒排表先获得一批候选问题，然后再通过余弦相似度做精准匹配，这样一来可以节省大量的时间。搜索过程分成两步：

- 使用倒排表把候选问题全部提取出来。首先，对输入的新问题做分词等必要的预处理工作，然后对于句子里的每一个单词，从`related_words`里提取出跟它意思相近的top 10单词， 然后根据这些top词从倒排表里提取相关的文档，把所有的文档返回。 这部分可以放在下面的函数当中，也可以放在外部。
- 然后针对于这些文档做余弦相似度的计算，最后排序并选出最好的答案。

这里以 TF-IDF 作为例子：

```
def get_handled_input_seq(query):
    result = []
    for word in query.split():
        word = handle_sentence(word.split())
        if word != None:
            result += word
    return result

# 检查输入的问题并返回处理过的问题tf-idf用，返回为字符串
def check_query(query):
    input_seq = get_handled_input_seq(query)
    return ' '.join(input_seq)

# 利用倒排表和同义词获取相关的预料库中问题的序号
def get_related_sentences(query):
    # 得到的是分词过后每句话的列表
    input_seq = get_handled_input_seq(query)
    # 定义相关词list
    si_list = []
    for word in input_seq:
        # 得到每句话的词
        if word in related_words:
            for value in related_words[word]:
                si_list.append(value)

    total_list = input_seq
    for word in si_list:
        total_list.append(word)
    sentence_list = []
    for word in total_list:
        # 如果word在倒排表里
        if word in inverted_idx:
```

```
            sentence_list.extend(inverted_idx[word])
    return list(set(sentence_list))
def getTopIndexByResult(result):
    top_idxs = []
    top_values = sorted(get_least_numbers_big_data(result, 5), reverse=True)
    len_result = len(result)
    dict_visited = {}
    for value in top_values:
        for index in range(len_result):
            if value == result[index] and index not in dict_visited:
                top_idxs.append(index)
                dict_visited[index] = True
    return top_idxs



def get_top_results_tfidf(query):
    """
    给定用户输入的问题 query, 返回最有可能的TOP 5问题。这里面需要做到以下几点:
    1. 利用倒排表来筛选 candidate （需要使用related_words).
    2. 对于候选文档，计算跟输入问题之间的相似度
    3. 找出相似度最高的top5问题的答案
    """
    # 将query转成字符串
    query = check_query(query)
    if query == "":
        print_format("please input a effect question","")
        return None
    # 得到返回的序号
    sentence_list = get_related_sentences(query)

    top_idxs = []  # top_idxs存放相似度最高的（存在qlist里的）问题的下表

    # 将输入的query转化为tf-idf向量
    input_seq = query
    input_vec = vectorizer.transform([input_seq])

    is_use_s_l = len(sentence_list) > 0

    if is_use_s_l == True:
        X_tfidf_si = []
        for id in sentence_list:
            X_tfidf_si.append(X_tfidf[id].toarray()[0])
        X_tfidf_si = np.array(X_tfidf_si)
        # csr_matrix根据行列索引到稀疏矩阵里的值
        result = list(cosine_similarity(input_vec, csr_matrix(X_tfidf_si))[0])
    else:
        result = list(cosine_similarity(input_vec, X_tfidf)[0])

    top_idxs = getTopIndexByResult(result)

    if is_use_s_l == True:
        top_idxs = [sentence_list[idx] for idx in top_idxs[:5]]
    else:
        top_idxs = top_idxs[:5]

    return [new_qlist[i] for i in top_idxs]  # 返回相似度最高的问题对应的答案，作为TOP5
答案
```

```
def test_output(question_list, iscorrect=False):
    for question in question_list:
        if question.strip() == "":
            print("Your question is empty")
            continue
        print(get_top_results_tfidf(question))

question_list = ["What time is it now ? ",
                 "What's the weather like today ?"]
test_output(question_list)
```

输出结果为

```
['the state of what is now Maharashtra', 'Chaeronea', 'Brazil', 'Manchester',
'Glatz (now Kłodzko, Poland) in Silesia']
['mild and changeable with abundant rainfall and a lack of temperature
extremes', 'hot and muggy', 'mild and also wet', '3400 BCE', 'severe
thunderstorms']

['现在的马哈拉施特拉邦'、'查罗内亚'、'巴西'、'曼彻斯特'、'西里西亚的格拉茨（现在的科兹科，波
兰）'"
["温和多变，雨量充沛，缺乏极端温度"，"炎热闷热"，"温和潮湿"，"公元前3400年"，"严重雷暴"]
```

四. 拼写纠错

暂时不做