TTDS COURSEWORK 1*

s1968246

October 2019

1 Environment

This course Work is implemented on a Windows 10 Computer by Python whose processor is Intel i9 9900k with 32GB memory. The version of python and important packages are shown below:

Package	Version
python	3.7.4
numpy	1.17.2
scikit-learn	0.21.3
scipy	1.3.1
stemming	1.0.1

Tabell 1: Environment Dependencies

2 Tokenization

Given a collection(.xml file), the file is transformed into a standard xml file with xml version tag at the first line. And then, a parent tag named 'Sample' is added to second line(also '/Sample' tag is added to the last line). The standard xml file is saved using original xml file name with a 'standard_' prefix. Here, xml.etree.ElementTree is used to parse the xml file, stemming is used to get the stem of every single token. For every document in the collection, I simply use '+' to add the headline to the back of its document text. I use regular expression and re.sub to substitute the '&amp' with ' ', then replace non-word character with ' '. It is noted that '_' is not be removed. All texts are transformed into lower case. Then applying split() to the modified text to get all tokens. Disposing all the tokens which appear in the list of stop words. After all process mentioned above, using stemming to get every token's stem.

*The codes of all modules mentioned are shown in the appendix.

3 Inverted Index

I use a dictionary to store all the inverted index, whose key is the stems and its value represents the position list and document id in the form of dictionary too. It just only needs one iteration of all the texts, which have been pre-processed. Each time, appending the current stem's position to the index dictionary. Because the pre-processed texts were stored in a list of tuple. So when I iterate every stem, it is very easy to obtain the document id, stem and stem position(using list index in Python). Even I use append() inside the for-loop, it almost takes several seconds to build the whole inverted index(given pre-process texts).

```
# index_dic[stem]={'doc_id':positionList}
```

Then I use Python built-in functions file write to output the inverted index dictionary in the required format to index.txt. After indexing, each time I do the search, I just need to load the index from the index.txt.

4 Search Module

In this coursework, only two types of search is given, Boolean search and ranked retrieval. So I write the code to distinguish the type of search by the filename, using split('.')[1] to get the type. The Boolean search is slightly more complicated than the other. First I make several functions for single term or phrase query. That is split the query by ' AND ' or ' OR ', then use these functions to do single query. After getting the single query result, deciding whether it is necessary to get the difference set between it to the document id set(contains NOT then it is necessary). Then using Union or Intersection to merge single queries result. If there is no ' AND ' or ' OR ' in the query, just do the single query as mentioned(also need to check whether the term contains 'NOT ' prefix).

Given a Boolean search txt file, read in every line. Then get the query id, and the rest of current line is the query. If ' AND ' or ' OR ' is in the query, using method mentioned above to get the result. If not, directly using the function boolean_query() to get the single query result. Firstly, check and remove(if need) the 'NOT'. Then check whether the query starts with '#'(applied .strip() before check, insure all blanks are removed). If so, get the phrase search distance(using split function, split by '(') and two terms, then do proximity search. If not, check whether the query starts ends with '"' and also ends with '"'. If so, do phrase search. If not, then do single term query. In proximity search, only iterate the document which both terms' inverted index contains. Using index to iterate the position list in current document. In every iteration, just compare the difference to decide which index need to move forward. So does the phrase search.

When given a ranked search txt file, I get the document id and query as the Boolean search file does. Iterating from the term in the query, get the document frequency, then I iterate every term position list in every document which contains current term using inverted

index. Just use len() to get the list length. So that I could get the term frequency and document frequency. In each iteration of every term, I use a dictionary to store the accumulative TF-IDF of every document (in every iteration of terms just update the document's TF-IDF which contains at least one current term). The specific equation of TF-IDF is

$$Score(q,d) = \sum_{t \in q \cap d} w_{t.d}, \tag{1}$$

where q dotes a term and d dotes a document.

It is worth mentioning that every term in both type queries needs to be transformed in to the lower case. And then use the lower case form to get the stem form. Because this is what I do to the collection. The query terms need to do the same pre-process.

In the end, output the query result in required format into different file. The function is named output_query(). It is shown in the appendix.

5 Comments and Challenges

With respect to programming, there is no challenge at all. I just took some time to deal with several specific details. In the procedure of tokenization, I was intended to keep '-' and "' at the beginning. But then I found that a lot of numbers and weird word woulds appear in the index.txt such as '22-26' ,"mad' and 'cow"(it is 'mad cow' in the original document). So keeping them then tokenizing the documents may not be a good idea. But, when I simply removed all non-letter character, a lot of 'amp' appeared. I realized that '&amp' was a escape sequence(I learnt it when I am studying html). It turned out that there were 2205 '&amp's in the trec.5000.xml. So I need to delete all '&amp' before split on every non-letter character.

When I coded the function for proximity search, I initially combined phrase search and proximity search together, assuming that phrase search is a special case of proximity search at a distance of 1. Then I realized that it was wrong, the phrase search need to consider the order of two terms, but proximity search do not need to. That is why the coursework code slightly differs from my lab code.

I should have paid more attention the 'AND' and 'OR'. Because if I simply check whether a query contains 'AND' or 'OR', some words such as 'SCOTLAND' will be wrongly split into 'SCOTL' and 'AND'. So, when check whether a query contains 'AND' or 'OR'. We should not forget the blank next to 'AND' and 'OR'. Thus, ' AND ' and ' OR ' will be reasonable strings to represent the logic operator in the query.

Although I've done read some IR papers and done some information ranking experiments relating to online reviews. The implementing the index, Boolean search and ranking method by hand is still quite useful to check my understanding of every detail of the formula and procedure.

The creating inverted index takes a lot of time. the for-loop could use Numba to accelerate. Or using multiporcessing to process different parts of collection, then merging

the result to get the complete inverted index. Try not use list.append inside a loop, because append is really really slow!(Some classmates asked me why their programs were very slow even ran on trec.sample.xml)

In this coursework, I learn that when you do not ensure about your idea, just validate your idea using small data sets or run the experiments to check the result. Always check the intermediate result is a good habit because sometimes you will find some mistakes in your codes.

Appendices

A Tokenization

```
def load_stopword(stopword_path):
    stopword_list = []
    with open(stopword_path, 'r',encoding='utf-8') as f1:
        stopword_list = [str(current_word).strip() for current_word in
     f1]
    return stopword_list

def tokenisation_text(text):
    # simply remove every non-letter character
    #     del_punc = r'[^A-Za-z0-9_-]' # keep _ -
    #     del_punc = r'[^A-Za-z0-9_-\']' #keep _ - '
    del_punc = r'[\W]'  # keep _

    text_nopunc = re.sub('&amp', ' ', text)
    text_nopunc = re.sub(del_punc, ' ', text)

    tokenisation_list = text_nopunc.split()
    return tokenisation_list


def lower_word(word_list):
    return [current_word.strip().lower() for current_word in word_list
    ]


def token_lower_nostop_stem_list(all_text, stopword_list):
    token_list = tokenisation_text(all_text)
    token_lowerlist = lower_word(token_list)
    token_lowerlist_nostop = [str(current_word) for current_word in
    token_lowerlist if
                              str(current_word) not in stopword_list]
    stem_list = [stem(current_word) for current_word in
    token_lowerlist_nostop]

    return stem_list

def to_standard_xml(xml_file_path):
    new_file_name = 'standard_' + xml_file_path.split('/')[-1]

    with open(new_file_name,'w',encoding='utf-8') as new_file:
        with open(xml_file_path,'r') as original_file:
            new_file.write('<?xml version="1.0"?>\n')
            new_file.write('<sample>\n')
```

```
            for line in original_file:
                if(len(line.strip()) == 0):
                    continue
                new_file.write(line)
            new_file.write('</sample>')
    return new_file_name

def merge_text(text1,text2):
    merged_text = text1 + text2
    return merged_text

def xml_all_text(xml_file_path,stopword_list):
    # trec
    standard_xml_path = to_standard_xml(xml_file_path)
    tree = ET.parse(standard_xml_path)
    root = tree.getroot()
    all_text_list = [(child.find('DOCNO').text,
    token_lower_nostop_stem_list(merge_text(child.find('HEADLINE').
    text,child.find('TEXT').text), stopword_list)) for child in root.
    findall("./DOC")]

    return all_text_list
```

B Inverted Index

```
def inverted_index(all_text_list):
    '''
    return: # index dic[stem]={'doc_id':position}
    '''
    index_dic = defaultdict(dict)

    for current_doc_tupple in all_text_list:
        current_doc_id = current_doc_tupple[0]
        current_doc_text_list = current_doc_tupple[1]

        for index in range(len(current_doc_text_list)):
            current_stem = current_doc_text_list[index]
            if (current_doc_id not in index_dic[current_stem].keys()):
                index_dic[current_stem][current_doc_id] = [index + 1]
            else:
                index_dic[current_stem][current_doc_id].append(index +
     1)

    return index_dic

def output_index(index_dic,file_path):
```

```
    sorted_index_dic_list = sorted(index_dic.items(),key=lambda x:x[0],reverse=False)
    with open(file_path,'w',encoding='utf-8') as file_1:
        for current_tupple in sorted_index_dic_list:
            current_stem = current_tupple[0]
            current_stem_doc_position_dic = current_tupple[1]
            file_1.write(current_stem)
            file_1.write(':\n')
            for current_doc_id,current_doc_id_position_list in current_stem_doc_position_dic.items():
                file_1.write('\t')
                file_1.write(str(current_doc_id))
                file_1.write(': ')
                current_doc_id_position_list_new = map(lambda x: str(x), current_doc_id_position_list)
                # print(','.join(current_doc_id_position_list_new))
                file_1.write(','.join(current_doc_id_position_list_new))
                file_1.write('\n')
            file_1.write('\n')
    return

def load_inverted_index(index_path):
    result_inverted_index = {}
    with open(index_path,'r') as f1:
        for line in f1:
            line = line.strip()
            if(line.endswith(':')):
                current_stem = line.replace(':','')
                result_inverted_index[current_stem] = {}
                continue

            if(len(line)==0):
                continue

            temp_split_list = line.split(': ')
            current_doc_id,str_position_list = temp_split_list[0],temp_split_list[1]
            current_position_list = [int(current_position) for current_position in str_position_list.split(',')]
            result_inverted_index[current_stem][current_doc_id] = current_position_list
    return result_inverted_index
```

C Boolean Search

```
def get_doc_id_set(current_inverted_dic):
```

```
    doc_id_set = set()
    for current_stem,current_doc_position_dic in current_inverted_dic.
    items():
        for current_doc_id in current_doc_position_dic.keys():
            doc_id_set.add(str(current_doc_id))
    return doc_id_set

def query_word(current_inverted_dic,current_word,is_not=0):
    current_word_stem = stem(current_word.strip().lower())
    if(is_not):
        for current_index_stem,current_index_stem_position in
    current_inverted_dic.items():
            if(current_word_stem == current_index_stem):
                doc_id_set = get_doc_id_set(current_inverted_dic)
                stem_doc_list = set(current_index_stem_position.keys()
    )
                return list(doc_id_set.difference(stem_doc_list))
    else:
        for current_index_stem,current_index_stem_position in
    current_inverted_dic.items():
            if(current_word_stem == current_index_stem):
                return list(current_index_stem_position.keys())
#     raise RuntimeError('Query not found.')
    return []

# OR
def union_list(a,b):
    result_list = list(set(a).union(set(b)))
    result_list.sort(key=lambda i:int(i))
    return result_list

# AND
def intersection_list(a,b):
    result_list = list(set(a).intersection(set(b)))
    result_list.sort(key=lambda i:int(i))
    return result_list

#proximity search
def proximity_query(doc_word_pos1,doc_word_pos2,current_distance,
    is_phrase = 0):
    result_list = []
    for current_docid_1,current_positionlist_1 in doc_word_pos1.items
    ():
        if(current_docid_1 not in doc_word_pos2.keys()):
            continue
        current_positionlist_2 = doc_word_pos2[current_docid_1]
        i = 0
        j = 0

```

```
45          while((i <= len(current_positionlist_1) - 1) and (j <= len(
    current_positionlist_2) - 1)):
46              if(is_phrase):
47                  if(int(current_positionlist_1[i]) > int(
    current_positionlist_2[j]) - current_distance):
48                      j+=1
49                      continue
50                  elif(int(current_positionlist_1[i]) < int(
    current_positionlist_2[j]) - current_distance):
51                      i+=1
52                      continue
53                  else:
54                      result_list.append(current_docid_1)
55                      break
56              else:
57                  if(int(current_positionlist_1[i]) > int(
    current_positionlist_2[j]) + current_distance):
58                      j+=1
59                      continue
60                  elif(int(current_positionlist_1[i]) < int(
    current_positionlist_2[j]) - current_distance):
61                      i+=1
62                      continue
63                  else:
64                      result_list.append(current_docid_1)
65                      break
66      return result_list
67
68  #phrase
69  def phrase_query(current_inverted_dic,query_phrase):
70      phrase_list = [ stem(current_word.lower().strip()) for
    current_word in query_phrase.replace('"','').split()]
71      result_list = proximity_query(current_inverted_dic[phrase_list
    [0]],current_inverted_dic[phrase_list[1]],1,is_phrase=1)
72      return result_list
73
74  def boolean_query(current_inverted_dic, current_query_word):
75      is_NOT = 0
76      #whether contains NOT
77      if(current_query_word.startswith('NOT')):
78          is_NOT = 1
79          current_query_word = current_query_word.replace('NOT','').
    strip()
80
81      if(current_query_word.startswith('"') and current_query_word.
    endswith('"')):
82          result_list = phrase_query(current_inverted_dic,
    current_query_word)
83
```

```
84      elif(current_query_word.startswith('#')):
85          current_query_word_list = current_query_word.split('(')
86          current_word_distance = int(current_query_word_list[0].replace
        ('#','').strip())
87          current_query_stem_list = [stem(temp_word.strip()) for
        temp_word in current_query_word_list[1].replace(')','').split(',')
        ]
88          result_list = proximity_query(current_inverted_dic[
        current_query_stem_list[0]],current_inverted_dic[
        current_query_stem_list[1]],current_word_distance)
89      else:
90          result_list = query_word(current_inverted_dic,
        current_query_word)
91
92      # if contains NOT, get the difference set: doc_id_set -
        result_list
93      if(is_NOT):
94          doc_id_set = get_doc_id_set(current_inverted_dic)
95
96          result_list = list(doc_id_set.difference(set(result_list)))
97
98      return result_list
```

D TFIDF

```
1  from collections import defaultdict
2  from math import log10
3
4  def tf_idf_weight(current_inverted_dic,query_phrase,is_stop=0,
       stop_word_path=None):
5      #doc_tfidf['doc_id'] = tf_idf
6      doc_tfidf = defaultdict(int)
7
8      # N
9      doc_number = len(get_doc_id_set(current_inverted_dic))
10
11     if(is_stop == 0):
12         stem_word_list = [ stem(current_query_word.strip().lower())
       for current_query_word in query_phrase.split()]
13     else:
14         stopword_list = load_stopword(stop_word_path)
15         stem_word_list = [ stem(current_query_word.strip().lower())
       for current_query_word in query_phrase.split() if
       current_query_word.strip().lower() not in stopword_list]
16
17     if(len(stem_word_list) == 0 or (len(list(set(stem_word_list).
       intersection(set(current_inverted_dic.keys()))))) == 0 )):
```

```
        raise RuntimeError('All query word not found!')

    for current_stem in stem_word_list:
        #current_stem_doc_tf_count['doc_id'] = count
        current_stem_doc_tf_count = {}

        if(current_stem not in current_inverted_dic.keys()):
            continue

        for current_doc_id,current_doc_pos in current_inverted_dic[current_stem].items():
            current_stem_doc_tf_count[current_doc_id] = len(current_doc_pos)

        current_stem_df = len(current_inverted_dic[current_stem].keys())

        for current_doc_id,current_stem_doc_tf in current_stem_doc_tf_count.items():
            doc_tfidf[current_doc_id] += (1 + log10(current_stem_doc_tf)) * log10(doc_number/current_stem_df)

    sorted_doc_tfidf = sorted(doc_tfidf.items(), key = lambda x:x[1], reverse = True)

    return sorted_doc_tfidf
```

E Output result

```
def output_query(query_result_dic,is_boolean=1):
    if(is_boolean):
        with open('results.boolean.txt','w',encoding='utf-8') as oq_file:
            for current_query_id,current_query_result_list in query_result_dic.items():
                for current_doc_id in current_query_result_list:
                    oq_file.write('%d 0 %d 0 1 0\n'%(int(current_query_id),int(current_doc_id)))
    else:
        with open('results.ranked.txt','w',encoding='utf-8') as oq_file:
            for current_query_id,current_query_result_list in query_result_dic.items():
                for current_docid_tfidf_tuple in current_query_result_list:
```

```
11                      oq_file.write('%d 0 %d 0 %.4f 0\n'%(int(
    current_query_id),int(current_docid_tfidf_tuple[0]),
    current_docid_tfidf_tuple[1]))
12
13      return
```