COMP6714 Information Retrieval and Data Mining

Project Part 1 Report

Name: Wenke Yang

ZID: z5230655

1. TF-IDF index construction for Entities and Tokens

(i). In the initialisation phase, my plan is to store the normalised term frequencies of tokens and entities in self.tf_tokens and self.tf_entities:

```
        self.tf_tokens = defaultdict(dict)
        self.tf_entities = defaultdict(dict)
```

The structures of them are both a dictionary of dictionaries, the key of outer dictionary is token/entity, the value is a inner dictionary, the key of inner dictionary is document id, the value is normalised term frequency of given document.

The inversed document frequency of tokens and entities would be stored in two dictionaries as follows:

```
        self.idf_tokens = {}
        self.idf_entities = {}
```

The key of idf dictionary is a token/an entity, the value is the corresponding normalised idf of the token/entity.

Besides, the spacy English language model is loaded in the initialisation phase as well. It can be called by self.nlp(document) for further parsing.

```
        self.nlp = spacy.load("en_core_web_sm")
```

(ii). Build TF-IDF index, first, count tf for tokens and entities, then, calculate normalised tf and idf for them separately.

The step 1~4 below are implemented in the function:

```
            def _count_tf(self, documents)
```

1. Before calculating, the first task is to **Parse each document** to a list of tokens and entities using spacy.
2. Traversing the entire entity list to **count tf for entities**. At the same time, record a **single word entities dictionary**. The key is the single word entity, the value is the start index of this entity in the document. This dictionary is used in next step for avoiding calculating the tf of these single word entities as tokens.
3. [Note: Here I decided to calculate the term frequencies of entities first, because single word entities should not be counted as tokens, if tokens calculated first, there would be some extra deleting cost for calculation of token list.]
4. Then, **calculate tf for tokens.** Traversing the entire token list, check the is_stop and is_punct flags associated with each token, skip the stop words and punctuations while

counting. At same time using ent_iob flag of tokens and the single word entities dictionary to determine if the current token is a single word entity or not. The flag ent_iob determines if the token is inside an entity, if it's inside an entity, and its text and starting index appear in single word entities dictionary, then, skip it when counting tf. The calculated tf for tokens and entities are the return value of the function `_count_tf`, which will be passed to the step 5 functions.

The step 5 below is implemented in functions below for token and entity separately:

```
def _calc_tf_idf_token(self, token_tf_count, total_doc_no)
def _calc_tf_idf_entity(self, entity_tf_count, total_doc_no)
```

5. **Calculating normalised tf and idf for entities and tokens** according to the given formulae in the specification. Since formulae for calculating entity and token are different, two separate functions are written. These two functions has no return value, but updates the tf and idf dictionaries in `__init__` for calculating max score in later stage.

All functions mentioned are called in the same order as mentioned above in the given function to build the tf-idf index:

```
def index_documents(self, documents)
```

2. Split the Query into Entities and Tokens

(i). Step 1 in spec., enumerate all possible entities from query Q that can be found in DoE. This part is implemented in the function:

```
def _get_entities(self, tokens, DoE)
```

This function takes two arguments, tokens: a list of tokens(words) by **splitting the query by space**, and the given DoE. It returns two items, the **entities list** and **token count**. First, traversing the token list and r**ecord the number of occurence of each token** for Step 3. Then, **enumerate all possible combinations of tokens** in increasing order using the **`itertools.combinations`** python library, append each possible entity to the possible entities list. Finally, **traverse each entity in DoE**, if it appears in the possible entities list, add to final entities list, return this list after completed. This guarantees that the final entities list is a subset of DoE.

(ii). Step 2 in spec., enumerate all possible subsets of entity set gathered from Step 1. This part is implemented in the function:

```
def _get_subsets_ents(self, entities, token_count)
```

This function takes the two return items from _get_entities function in Step 1. It **returns a list of subsets of entities**. First, it **enumerates all possible subsets** of entities using the **`itertools.combinations`** python library. For each subset, if the length of subset is larger than one, it is possible to have token count exceeding issue, so we call the function in Step 3 to judge if the subset is qualified to be appended to the final subsets of entities list. As a conclusion, any subset enumerated with **length less or equal to one will be appended** to the final subsets list. In addition, subset with **length larger than one and not exceeding the token count** will also be appended to the final subsets list.

(iii). Step 3 in spec., filter subsets gathered in Step 2 by token count in Q. This part is implemented in the function:

```
    def _exceed_token_count(self, subset_ents, token_count)
```

This function takes one subset of entities from Step 2 and the token count of query Q from Step 1. It calculates the **token count of the given subset**, if any token in the subset **exceeds the token count of query, return True**, else False. The return value is used in Step 2 for judging if the subset enumerated is valid.

(iv). Step 4 in spec., split the query to entities and tokens according to filtered subsets of entities obtained from Step 3. This part is implemented in the function:

```
    def _create_splits(self, subsets_ents, tokens)
```

It takes the returned value of function `_get_subsets_ents`, a list of subsets of entities, and the list of tokens splitted from given query Q. This function creates a dictionary of dictionary to store all possible splits. For each subset of entities, create its corresponding list of tokens by removing the tokens appear in the subset from all tokens in query Q. The final return value is a dictionary with key as the index of split, value as a inner dictionary. Each inner dictionary has two keys, string 'tokens' and string 'entities', the values are a list of tokens and a list of entities of this split respectively.

All functions mentioned are called in the same order as mentioned above in the given function to split the query to tokens and entities:

```
    def split_query(self, Q, DoE)
```

3. Query Score Computation

All implementation for calculating the query score are in the given function:

```
    def max_score_query(self, query_splits, doc_id)
```

It takes two arguments, the first one is the return value from `split_query` function, a list of splits of query, each split has a token list and an entity list, the second argument is the document id i, used to calculate the tf-idf score. Traverse each split in query_splits, for each token in the token list, summing the tf-idf scores for tokens if it appears in document i in self.tf_tokens and self.idf_tokens (calculated at the First stage); for each entity in the entity list, if it appears in document i in self.tf_ entity and self.idf_entities (calculated at the First stage), summing the tf-idf score for the entities. Skip any token or entity not appear in document i in the tf/idf dictionary (i.e. the score is zero). Calculate the score for each split with given formula score_entity + 0.4 * score_token. Finally, return the maximum score and the corresponding split.