COMP6714 Information Retrieval and Web Search

Project 1 Part 1

Han YANG - z5140181

In this project, there are 3 main steps to compute the TF-IDF score: 1) TF-IDF index construction for entities and tokens, 2) split the query Q into lists of entities and tokens, 3) query score computation.

To construct **TF-IDF index** for entities and tokens, firstly I need to compute the term frequency of token t or entity e in a document D_j, $TF_{token}(t, D_j) = \# \ of \ times \ token \ t \ appears \ in \ D_j$, $TF_{entity}(e, D_j) = \# \ of \ times \ entity \ e \ appears \ in \ D_j$. To implement it, I used 2-D dictionary to store the term frequency of each document, the key of the first dimension is term (token t or entity e), and the key of the second dimension is document D_j, the value of the second dimension is the frequency of one term in one document. In my solution, I used the parsing results of Spacy (v2.1.8) to help me to compute the TF of entities and tokens (tokens are not stopwords and punctuations) for each document separately, I also stored the term frequency of each single-word entity at the same time, which is used to cut off the frequency of tokens that are single-word entities, since for single-word entities, we should not consider the corresponding token for computing the TF-IDF index of tokens.
Then, to de-emphasize the high frequency, I computed the normalized TF for tokens and entities, $TF_{norm\_token}(t, D_j) = 1.0 + \ln\left(1.0 + \ln\left(TF_{token}(t, D_j)\right)\right)$, $TF_{norm\_entity}(e, D_j) = 1.0 + \ln\left(TF_{entity}(e, D_j)\right)$. The last step is to compute the Inverse Document Frequency IDF for tokens and entities, $IDF_{token}(t) = 1.0 + \ln\left(\frac{total \ \# \ of \ docs}{1.0 + \# \ of \ docs \ containing \ token \ t}\right)$, $IDF_{entity}(e) = 1.0 + \ln\left(\frac{total \ \# \ of \ docs}{1.0 + \# \ of \ docs \ containing \ entity \ e}\right)$.

The second main step is to **split the query Q** into entities and tokens, in my solution, I defined a function called **is_entity_in(entity, Q)** to check if an entity presents in a set of tokens (e.g. query Q), in this function, I used a loop to iterate all words in the input entity to check if every word presents in Q **sequentially**, when I found one word, I would record its index (position) and pop out this word in Q, and in next iteration I would find the new word start from this recorded position, which can ensure the entity presents in the increasing order in Q. Once one word cannot be found, this function would return False, it means this entity dose not present in Q, otherwise, it would return True after all iterations.

To find all possible splits of query Q, firstly, I used this function to select all entities present in Q from DoE, and used this entity subset of DoE to enumerate all possible subsets of entities, which can save some computing cost in some cases. Then I computed all permutations for each subset, and for each permutation I also used function **is_entity_in(entity, Q)** to check if all tokens of it are present in Q and the token count of it dose not exceed the corresponding token in Q, once I found one valid permutation, I would record this subset of entities as entities and the rest of keywords in the query Q as tokens, and treated it as one possible split of the query Q, and I would not check other permutations for the same subset.

The last main step is to **compute the query score** of a document D_j, in this part, I used the corresponding TF-IDF index to compute the scores for tokens and entities for each query split, (k_i, e_i). $s_{i1} = \sum_{entity \epsilon e_i} TF_{norm\_entity}(entity, D_j) * IDF_{entity}(entity)$,
$s_{i2} = \sum_{token \epsilon k_i} TF_{norm\_token}(token, D_j) * IDF_{token}(token)$,
$score_i(\{k_i, e_i\},\ D_j) = s_{i1} + \lambda * s_{i2} (\lambda = 0.4)$. (When I computed these two scores (for tokens and entities), I will treat it as 0 for the cases where the term frequency (TF) for the token or entity is 0). Then return the maximum score among all splits as the final result.