Query Expansion

Assignment 2

CT4100

Liselotte Lichtenstein (24240597)

Jonas Michel(24238749)

25th November 2024

1 Question 1 (10 Marks)

Given a query that a user submits to an IR system and the top N documents that are returned as relevant by the system, devise an approach (high-level algorithmic steps will suffice)

to suggest query terms to add to the query. Typically, we wish to give a large range of suggestions to the users capturing potential intended query needs, i.e., high diversity of terms that may capture the intended query context/content.

The goal of the here designed query expansion (QE) heuristic is to enhance search results by adding terms that increase the specificity of a query, by proposing both relanvt and diverse terms for the user to select from. The objective is to design a interactive QE technique, by suggesting a range of terms, the user can then add to the query to refine the search results. Our solution integrates a global concept-based technique with a local pseudo-relevance feedback (PRF), combining relevance and diversity to generate more effective query suggestions [**azad2019query**]. Our approch is defined in high-level algorithmic steps below in Alorithm 1, and desribed in detail in the following.

Step 0 (Input and Output):
As prerequisite the document collection is preprocessed to have embeddings and clustering for the terms in the collection. This can be done offline to reduce runtime complexity. The input to the algorithm is the query terms q_{terms} and the document collection with embeddings and clustering. It has to be noted that during the algorithm we have to embedde the individual query terms as well, so the embedding models has to available. The output is predifined number of k terms that can be used to expand the original query.

Step 1 (Query Embedding):
The query needs to be embedded in order to further process it. This is done by embedde the whole query in the same space as the terms in the document collection. In our case we want to find later on terms similar to the whole query instead of just similar to the individual query terms. Therefore we get the embedding for each individual query term and apply mean pooling to get the embedding for the whole query [**chen2018enhancing**].

Step 2 (Get Relevant Terms):
After embedding the query terms, the next step is to extract relevant terms r_{terms}. There are two types of relevant terms, the most similar terms from the embedding space and the co-occurrence terms from the top N returned documents on the original query. This is done to ensure that we have semanticly similar terms as well as terms that co-occur with the original query terms. The semantically similar terms are selected from the embedding space to ensure a search within the entire vocabulary and therefore enhance diversity. The co-occurrence terms are selected from the top N returned documents on the original query, utilizing a pseudo-feedback (PRF) approach [**azad2019query**]. This allowed us to add relevant terms that might occur in the same documents but are not semantically similar. Each of the candiate terms is assigned a weight based on the co-occurrence with the original query terms and then the top m terms are selected as co-occurrence terms

[**azad2019query**].

Step 3 (Get Corresponding Clusters):
As we stated in Step 0, the collection vocalbulary has to be clustered. After getting the relevant terms, we can now get the corresponding clusters for these terms. This is done by selecting all clusters that contain the terms in r_{terms}.

Step 4 (Select Top k Clusters):
Since Step 3 can give us back a large number of clusters, we have to select a certain subset of k clusters to extraxt suggesting terms from. We want to select large clusters, since they seem relevant. But the clusters shoudl also be spead out in the embedding space to ensure diversity. These two measures are balanced in a score to ranke the clusters as defined by:

$$score = \alpha \cdot s + \beta \cdot d$$

where s is the normalised size of the cluster and d is the distance of the cluster centroid to the other cluster centriodes. The parameters α and β can be used to adjust the importance of size and distance. Based on this score the top k clusters are selected.

Step 5 (Extract Terms from Selected Clusters):
The selected top k clusters represent now the most relevant and diverse topics relating to the original query. In order for the user to select one of these topics we have to propose one term that is descriptive for the cluster. This is done by selecting the term that is closest to the centroid of the cluster [**rossiello2017centroid**, **khennak2019clustering**]. If this selected terms is already in the original query, the next closest term is selected.

Step 6 (Return QE Suggestions):
The terms that are selected in Step 5 are then returned as the query expansion suggestions. The number of returned terms is defined by the number of top clusters k. The number k is a restriction to ensure that the user has a limited number of terms to select from to be not overwhelmed with suggestions.

Reflection:
By precomputing the vocabulary embedding and clustering as well as the embedding model for the query runtime complexity is significantly reduced. Which allowes for a fast response time for the user.

In the case our query and the resulting releveant terms are so specidify that we only get one highlevel cluster (e.g., “jaguar engine horsepower” $\rightarrow$ cluster: “cars”). In that case we might want lowlevel clusters like the different engines or jaguar models. As a solution instead of just precomputing one clustering, multi-resolution clustering can be used to address different levels of specificity in the query [**lutov2019accuracy**]. The level of specificity would need to be seleted during runtime wihtin our algorithm before Step 3.

In general a lot of hyperparamters in our algorithm can be adjusted to the specific use case and also need to be examined in general studies to find the best values for them. Possible tunable parameters are, the number of similar terms n and co-occurrence terms

Algorithm 1 Query Expansion (QE) Suggestions

Input: Query terms q_{terms} and document collection with embeddings and clustering
Output: QE suggestion terms

Step 1: Get query embedding from q_{terms} (mean pooling)
Step 2: Get r_{terms} relevant terms (union of s_{terms} and c_{terms})
  Step 2.1: Get n most similar terms from embedding space (s_{terms})
  Step 2.2: Get m co-occurrence terms from top N returned documents on the original query (c_{terms})
Step 3: Get corresponding clusters for r_{terms}
Step 4: Select top k clusters (weighted with size and centriod distance)
Step 5: Get one descriptive term for each cluster (e.g., closest to centroid)
  Step 5.1: Check that the descriptive term is not in the original query
Step 6: Return the descriptive terms as QE suggestions

m to select, the number of clusters k to select, the weighting parameters α and β for the cluster selection, and the level of specificity for the possible multi-resolution clustering.

2 Question 2 (10 Marks)

Consider the following scenario: a company search engine is employed to allow people to search a large repository. All queries submitted to the system are recorded. A record that contains the id of the user and the terms in the query is stored. The order of the terms is not stored and neither is any timestamp. Each entry in this record is effectively an id and a set of terms.

The designers of the search engine, decide to use this information to develop an approach to make query term suggestions for users, i.e., at run time, once a user has entered their query terms, the system will suggest potential extra terms to add to the query.

Given the data available, outline an approach that could be adopted to generate these suggested terms. A brief outline is sufficient to capture the main ideas in your approach. The designers of the system wish to take into account previous queries and any similarities between users. Identify the advantages and disadvantages of your approach (briefly).

The goal is to develop a system that suggests additional terms for a query based on the recorded history of user queries. The approach incorporates three levels of scoring, which utilize the frequency and co-occurrence of terms in the query history at different granularities.

Outline of the Approach

1. **Frequent Searches by the User:** For the current user, analyze their past query history. If terms t_1 and t_2 frequently co-occur with t_3 in this user's queries, t_3 should be suggested as an additional term. This is personalized to the user and gives priority to their individual search patterns.

2. **Frequent Searches by Similar Users:** If the current user has insufficient query history, cluster users based on their query behaviors. Clustering can be done offline using techniques such as k-means, hierarchical clustering, or graph-based methods. For instance:

   - Represent each user's query history as a vector of term frequencies or co-occurrence statistics.
   - Compute similarities using measures like cosine similarity or Jaccard similarity.
   - Group users into clusters where members have similar search behaviors.

   Then, suggest terms based on the aggregated query patterns of users in the same cluster as the current user.

3. **Global Query Patterns:** For new users or those with no history (e.g., users who have deleted their data), leverage global query data. Identify frequently co-occurring terms in the entire dataset of historical queries and suggest terms based on these patterns. This provides a fallback mechanism when personalized or cluster-based data is unavailable.

Advantages

- **Personalization:** Levels 1 and 2 allow for tailored suggestions based on the user's behavior and that of similar users.
- **Adaptability:** Level 3 ensures the system works for new users or users with no stored history, maintaining functionality for all scenarios.
- **Scalability:** Offline clustering of users ensures that the system remains efficient during query processing at runtime.

Disadvantages

- **Cold Start for New Terms:** Rare or novel terms may not be suggested effectively, as they may not appear frequently in the dataset.
- **New Positions:** Inheriting old search history and user profiles can lead to entirely irrelevant suggestions, for example, if a user changes their position within the company.
- **Computational Complexity:** Clustering users and maintaining similarity measures can be computationally expensive as the dataset grows.
- **Privacy Concerns:** While clustering and global statistics are anonymized, using individual query histories (Level 1) could raise privacy issues if not handled appropriately.

Score that consists of three components, one based on the users search history, one basedon the search history of similar users and one based on the global search history. BY adjusting the weights of these components the score of the term can be adjusted.

$$score(t) = \alpha \cdot user\_history(t) + \beta \cdot similar\_users(t) + \gamma \cdot global\_history(t)$$

where α, β and γ are the weights for the three components and the three scores are based on the frequency a term occurs given the entered query terms in the respective context. These individual term scores for each context are calculated as follows:

- absolute frequency of the term in combination with entered query terms (t_q)
- proportion of the term in the context of (t_q)
- ...

Since the contains the first and second as well, occurences in the own users history and similar users history get an additional boost. This third component adds a globel context to the term score and depending on its weight can give 'new to the user' terms a chance to be suggested. It also provides a fallback for the case that a user enters unusual terms

that are not in the history of the user or similar users or the case of a user being new and having no or very limited history.

Two users are considered similar if they not only use the same search terms but also use them in similar combinations - have similar queries. For each two users a score is calculated as the average query similarity between two users.

$$\text{UserSimilarity}(A, B) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} w_i \cdot v_j \cdot \text{QuerySimilarity}(q_{Ai}, q_{Bj})}{\sum_{i=1}^{n} \sum_{j=1}^{m} w_i \cdot v_j}$$

Users that surpass a certain threshold are considered similar users. The threshold can be adjusted to the specific use case. The weights w_i and v_j are used to adjust the importance of the individual queries in the calculation of the similarity score. The QuerySimilarity function is used to calculate the similarity between two queries and can be calculated by applying metrics like the Jaccard similarity or cosine similarity on the query representations. The similarity score is then normalised by the length of the queries.

two users are considered similar if they are part of the same cluster. In order to cluster the users, they are represented as a vector of:

- the embedding of all used terms set (looses query information)
- the frequency of the terms in the user log (looses order information) (different frequency metrics (e.g. number of queries, overall num of uses, normalised frequencies, ...))
- the combined (pooling/concatenation/...) query embeddings (query embeddings could be sentence based or pooling of term based)
- ...

These vector representations of the users are then clustered using a clustering algorithm like k-means or hierarchical clustering. The number of clusters can be adjusted to the specific use case. The term score is then calculated in the context of the user logs of users that share the same cluster as the user that entered the query.