Name: Andrew Hayes
Student ID: 21321503

CT4100

Name: Cathal Lawlor
Student ID: 21235456

Assignment 2: Query Term Suggestion

1 Question 1

Considering that we are given a query that a user submits to an information retrieval system, and the top N documents that are returned as relevant by the system, and that we want to suggest query terms to add to the query, the types of suggested terms that would be useful to the user are terms that split the set of returned documents; that is, we want to suggest terms to the user that narrow down their search. This requirement also contains the implicit requirement of a *diverse* set of suggested terms: there is little point in suggesting several terms with near-identical semantic meaning, e.g., "car", "automobile", "vehicle", as these will do little to narrow down and refine the set of returned documents. Although there can be utility in suggesting similar or indeed synonymous terms to expand the set of returned documents to contain documents which have similar contents even if they don't contain the specific query terms used, we will be focusing here on shrinking or narrowing down the set of returned documents to enhance specificity and remove irrelevant documents.

However, we must also consider that the terms we suggest must nonetheless be relevant to the query. While we could maximise the diversity of the terms by just randomly selecting words from a dictionary, this would not be useful to the user for their specific search. The key challenge here is to find terms which are maximally similar to the query terms, but are also maximally dissimilar from the other terms we are suggesting. One approach we could use to find such terms is **clustering**: a good clustering algorithm maximises intra-cluster similarity (thus grouping documents with similar contents into the same cluster) while minimising inter-cluster similarity (thus ensuring that the documents in each cluster are as dissimilar from the documents in other clusters as possible).

Our proposed approach to query term suggestion is as follows:

1. Run the query and retrieve the top N most relevant documents from the corpus.
2. Weight the terms in the returned documents using a slightly modified TF-IDF: while inverse document frequency is normally used to avoid giving high weights to terms that are common across the entire corpus, here we will be calculating it not for the entire corpus but just for the returned documents. Since the documents were returned by a search query, we know that they are similar to the terms in that query and therefore the terms with the highest frequencies would likely already be in the original query; by calculating the TF-IDF, we reduce the weights given to terms that are common across the N returned documents, and thus give higher weights to terms that distinguish the document in which they occur from the other returned documents.
3. With the TF-IDF calculated, the documents can be represented as term vectors and have their similarity measured using some metric such as cosine similarity. A clustering algorithm such as k-means can then be used to cluster the documents into a series of clusters that have maximal intra-cluster similarity (that is, each cluster contains documents that are as similar in content as possible) while also having minimal inter-cluster similarity (that is, the documents in each cluster are as distinct as possible from documents in other clusters). Because we want the clusters to be maximally dissimilar, a hard clustering approach such as k-means that only allows a document to belong to a single cluster is preferable.
4. Finally, for each cluster, identify the term which has the highest cumulative weight by adding together the weights assigned to a term by each document in that cluster; we want to identify the "definitive" term for each cluster. These highest-ranked terms are the terms which can then be suggested to the user to enhance their query. Because the clustering has been done using TF-IDF, it is unlikely that the highest-ranked term in one cluster will be occur frequently enough to be the highest ranked term in another cluster; however, for the sake of completeness & error-handling, if the highest-ranked term in a cluster is the highest-ranked term in another cluster, take the second-highest, then the third-highest, et cetera.

The length of the list of terms suggested to the user can be controlled either by setting the amount of clusters k or by only selecting the top n highest-weighted terms found from the clustering.

We have proposed a flat clustering approach using k-means because flat clustering best facilitates diversity among the clusters; however, a hierarchical approach to clustering could alternatively be utilised to suggest *sequences* of terms. For example, using a hierarchical clustering approach, we could suggest the two terms that cause the greatest split in the search space; if the user accepts one of these suggestions, we can then descend one level of the cluster hierarchy and suggest terms that further split the search space of that suggested term. However, hierarchical clustering is extremely computationally expensive due to its recursive nature, with the type of divisive hierarchical clustering that we are suggesting here having a time complexity of $O(2^n)$ which is likely unacceptably slow, especially for a vague query that returns a large number of documents.

2 Question 2

The problem of suggesting additional query terms to a user at run-time as they enter the query is an interesting contrast to the previous question: because we have not yet executed the query, we don't actually know what the documents returned will be like and therefore it is difficult to try and split this search space. We are also restrained from making Google-like auto-completion: the queries are stored unordered as a "bag of words" and therefore we cannot finish the user's sentences for them, just suggest terms that have previously occurred in similar queries. Furthermore, because the timestamp of a query is not stored, we cannot try to identify how users have refined similar queries. If the timestamp was stored, we could see what queries were made in quick succession, and if they were very similar we could infer that each subsequent query is a refinement on the preceding query, and that the final query in that short time frame was the optimal query that provided the best results.

Instead, the best we can do is suggest terms that we think might go with the query that the user has entered, either because of the query's similarity to other queries in the database, or because of the user's similarity to other users in the database. There are three primary ways in which such a system could be of use to a user:

- Saving the user typing: the system could suggest terms that the user was going to enter anyway, but that the user now doesn't have to manually type, not dissimilar to Google's auto-complete, just without the word-order;
- Helping to refine vague queries: if the user is searching for a term such as "jaguar" and the system suggests adding the terms "cat" and "car", the query can be refined before the search has even been executed;
- Even if none of the suggested terms are relevant to the query, it can still provide utility to the user by showing them that the system isn't understanding what they are trying to search for. For example, if a classicist entered the search term "Paris", looking for information on the prince of Troy of the same name in *The Illiad*, and all the suggested terms were related to the capital city of France and not ancient Greek poetry, the user would quickly understand that they need to add more specificity to their search as the system has not understood what they are looking for. In this way, we can kind of get away with not suggesting terms that are very specific to a certain type of query, as even the unrelated query term suggestions can provide information on how to refine the query to a human user, without ever running the query.

The other primary issue is that all this needs to happen live, as we're making these suggestions before the user has even executed the search, and so time is of the essence. All this being said, our proposed approach is as follows:

1. Before run-time, analyse the set of queries that each user has made using some similarity metric such as cosine similarity, and add the users into a series of soft clusters of users who make similar queries. Soft clustering is important here as it allows a user to belong to a number of clusters instead of just one. In this way, a user who works as a front-end software engineer could belong to a "front-end" cluster *and* a "software development" cluster based off the types of searches they make and thus could be recommended the terms from the types of searches that are made by both front-end designers and back-end engineers.
2. Then, at run-time when a user has entered some search terms, search the database for similar queries made by users in the same cluster as the user in question, using some metric such as cosine similarity to identify similar queries. The returned list of similar historical queries can be limited either by just selecting the top n most similar queries or by specifying a minimum level of similarity, or both.

   If the user is new to the system, and thus does not have enough historical queries to be accurately clustered, or just has not yet been clustered, just search the entire query history instead. While this is less efficient and will be less accurate, there is nothing that can really be done about this without guessing which clusters a user should belong to.
3. For the set of similar queries obtained, identify the top n most common terms across the queries that do not occur in the original query that the user has entered. Suggest these terms to the user to add to their query.

Advantages of this approach include:

- The clustering of the users occurs before run-time, reducing run-time computation and increasing speed & responsiveness.
- Clustering the users allows us to limit the number of queries we have to search through to find similar queries, and increases the likelihood that the queries to which we are comparing the user's query are relevant.
- The soft clustering approach allows users to belong to multiple clusters, and therefore we aren't "pigeon-holing" users and thus restricting our understanding of what types of searches they might make.
- The system can handle a user not yet belonging to a query or not having made enough searches to have been accurately clustered.

Disadvantages of this approach include:

- The clusters need to be pre-computed in advance, and need to be updated regularly to ensure that the clustering is as accurate as possible.
- If a user makes an uncharacteristic search for whatever reason, it is unlikely that relevant similar queries will be found in the clusters to which that user belongs. This can be handled by falling back on searching the entire query history if there are too few historical queries found that meet a minimum similarity requirement.
- Because we just search for similar queries made by similar users, there is no guarantee of diversity among the terms that we suggest to the user. While we know that the suggested terms will be similar to the user's query, we also know that the suggested terms will also probably be similar to each other and thus not do much to split the search space. Therefore, this approach to query term suggestion may be more useful as a form of auto-complete than as a way of refining vague queries. However, as mentioned previously, we can get away with this to a certain extent, as the user can infer from the fact that the suggested terms are not relevant to their intended search that they need to add their own terms to the query to narrow it down, although this is not entirely ideal.
- Because we are trying to suggest terms before the query is submitted, it is likely that query terms entered do not constitute a complete query. Therefore, when we compare these query terms to historical queries, we are comparing an incomplete query to complete queries, which may have unintended consequences.
- Because the historical queries are not timestamped, we cannot consider only recent queries and are forced to search through the entire historical record to find similar queries, which could become quite slow as the database grows and could negatively influence our term suggestions, as older queries are less likely to be relevant. However, there's not much that we can do to mitigate this, and it's primarily just a flaw in the database design.
- Another consequence of the lack of timestamping is that we cannot distinguish old queries from new queries when clustering users. While this problem is offset by our choice of soft clustering as users won't just belong to a single cluster, users will nonetheless be added to clusters based off potentially very old queries. A user who used to work as a developer and made searches related to that role who has since been promoted to a manager will still get term suggestions related to their past as a developer, as the system cannot distinguish their old development-related queries from their more recent management-related queries.