Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

CT4IOO

2024–11–01

Assignment 1: Information Retrieval

1 Question 1

1.1 Indexing Structure for a Sparse Term-Document Matrix

One of the key factors that must be considered when choosing an appropriate indexing structure for a term-document matrix is the sparsity of the matrix, as (according to Zipf's law) most terms will occur quite rarely in the corpus and not occur at all in most documents, resulting in the majority of indices in the term-document matrix containing a NULL value. Another key factor that must be considered is lookup speed: typically, we will be trying to find the documents that are most relevant to a given query or vector of terms, so we want to be able to quickly find a given term in the matrix and the documents in which that term has the highest weight.

One data structure that addresses these factors is the **inverted index**. At a high level, this is a data structure which consists of the list of all terms in the corpus, where each term in the list points to a list of tuples (called a *posting list*) containing the identifier of each document in which the term occurs and the weight of said term in the document. This completely circumvents the issue of storing a large volume of NULL weight values, as we only store a weight for a document which contains the given term.

If the term list was implemented as a hash table with a suitable hash function yielding minimal collisions, where each term in the corpus is a key pointing to a posting list value, the time complexity of retrieving the list of documents in which that term occurs would be $O(1)$ in the general case. Provided the posting list was implemented as a list of document-weight pairs, sorted by decreasing order of weight, it would then also be an $O(k)$ operation to retrieve the top k documents for which that term is relevant, with k being a fixed integer that does not scale with the list size n. Therefore, searching for the most relevant documents for a term or calculating which documents are most relevant to a query vector would be extremely fast & efficient.

1.2 Algorithm to Calculate the Similarity of a Document to a Query

Assuming that the both the query and the document are supplied in full as just a string of terms:

```
"""
Input:
    query_terms: an array of terms in the user query, suitably pre-processed (e.g., stemmed, lemmatised)
    doc_id: an integer identifying the document in the inverted index
    inverted_index: a hash table of terms and tuples consisting of the doc_id and the weight of the term
    ↪  in that document
"""
def similarity(query_terms, doc_id, inverted_index):
    query_vector = {}  # dictionary to store weights of terms in the query
    doc_vector   = {}  # dictionary to store weights of terms in the document

    # calculate the term frequency for each term in the query
    for term in query_terms:
        # initialise to 1 if not already present in vector, otherwise increment
        query_vector[term] = query_vector.get(term, 0) + 1

    # normalise the query weights
    for term in query_vector:
        query_vector[term] = query_vector[term] / len(query_terms)

    # Step 2: Retrieve document term weights from the inverted index
    # for each query term, find the term in the inverted index, if present
    for term in query_terms:
        if term in inverted_index:

            # find the weight of the term in the given document, if present and add to doc_vector
            for (doc, weight) in inverted_index[term]:
```

```
                if doc == doc_id:
                    doc_vector[term] = weight

    # calculate the dot product of the query vector and document vector
    dot_product = 0
    for term in query_vector:
        if term in doc_vector:
            dot_product += query_vector[term] * doc_vector[term]

    # calculate the magnitudes of the query and document vectors
    total_squared_query_weights = 0
    for weight in query_vector.values():
        total_query_weights += weight^2
    query_magnitude = sqrt(total_squared_query_weights)

    total_squared_doc_weights = 0
    for weight in doc_vector.values():
        total_doc_weights += weight^2
    doc_magnitude = sqrt(total_squared_doc_weights)

    # calculate cosine similarity
    return (dot_product / (query_magnitude * doc_magnitude))
```

Listing 1: Algorithm to Calculate the Similarity of a Document to a Query

As can be seen from the above algorithm, calculating the similarity of a specific document in the corpus to a query is not a particularly efficient operation using the inverted index: finding the tuple pertaining to the given document in the postings list for a query term is an $O(n)$ operation in the worst case, and n could be potentially billions of documents depending on the corpus in question; it would most likely be computationally cheaper to just ignore the inverted index and recompute the weights of each term in the document. However, I still maintain that the inverted index is a good choice for term-document matrix, as I assume that general searching of the corpus for the most similar documents to a query is the ordinary use case of such a data structure.

2 Similarity of a Given Query to Varying Documents

For a document $D_1 = \{\text{Shipment of gold damage in a fire}\}$ and a query $Q = \{\text{gold silver truck}\}$, and assuming that we are only considering the similarity of the query & document as weighted vectors in the vector space model, then $\text{sim}(Q, D_1)$ should be relatively low as the query and the document only share one term. Since no term is repeated in either the query or the document, each term should have equal weight. For each of the following augmentations on D_1:

a) $D_1 = \{\text{Shipment of gold damaged in a fire. Fire.}\}$: the inclusion of an additional term "fire" increases the weight of the term "fire" in determining the meaning of the document. Since Q does not contain the term "fire", the $\text{sim}(Q, D_1)$ will be reduced.

b) $D_1 = \{\text{Shipment of gold damaged in a fire. Fire. Fire.}\}$: the inclusion of two additional instances of the term "fire" further increases the weight of the term "fire" in determining the meaning of the document, and thus further reduces $\text{sim}(Q, D_1)$.

c) $D_1 = \{\text{Shipment of gold damaged in a fire. Gold.}\}$: the repetition of the term "gold" in D_1 increases the weight of the term in determining the meaning of the document, and since the term "gold" also appears in Q, $\text{sim}(Q, D_1)$ will be increased compared to the unaltered document.

d) $D_1 = \{\text{Shipment of gold damaged in a fire. Gold. Gold.}\}$: the double repetition of the term "gold" in D_1 further increases the weight of the term in determining the meaning of the document, and since the term "gold" also appears in Q, $\text{sim}(Q, D_1)$ will be further increased.

However, a human reviewer of the above similarity scores might argue that further repetition of terms in the augmented documents does little to affect the meaning of the document, and so one could consider using the logarithm of the term frequency to reduce the significance of each additional occurrence of a term.

3 Context-Based Weighting Scheme for Scientific Articles

The two additional features I have chosen to include in my context-based weighting scheme are:

- **Citation count:** a somewhat obvious choice, as citation count is a measure of the number of times the article in question has been referenced by another publication, and thus is a good indicator of how influential the article is. Including the citation count in the weighting scheme will prioritise returning more influential articles, and increases the likelihood that returned articles will be of use to the searcher. However, since it is unlikely that the $n + 1^{\text{th}}$ citation when $n = 3000$ holds the same importance as the $n + 1^{\text{th}}$ citation when $n = 5$, the logarithm of the citation count should be used instead of the raw citation count. Since the citation count may be zero, we ought to add 1 to the citation count before calculating the logarithm, as $\log(0) = -\infty$; while we do want to assign a negative bias to low citation counts, I think $-\infty$ is probably *too* negative.
- **Years since publication:** the inclusion of the citation count in the weighting scheme could cause an undesirable bias that favours older articles, as newer articles may have a low citation count simply because enough time hasn't elapsed since their publication for them to have been cited by other publications. This is especially undesirable for scientific papers, where one would imagine that more recent & up-to-date research articles would be of greater importance (generally speaking) than older articles. This can be counteracted via the inclusion of a negative bias based on the number of years since publication: the older the article, the greater the reduction. However, subtracting some value from the similarity score could cause the similarity score to become negative, particularly in the case of very old papers that are very dissimilar to the query. To maintain positive similarity scores for the sake of simplicity, I instead chose to incorporate the years-since-publication as a negative exponent on a positive number so that the resulting value is never negative, but shrinks as exponentially as the documents get older.

With these two features in mind, my proposed weighting scheme would be as follows:

$$S_i = \alpha \cdot \text{tf-idf} + \beta \cdot \log(C_i + 1) + e^{-\gamma Y_i}$$

where:

- i is the document in question.
- S_i is the significance of the document i.
- α, β, & γ are tuning parameters that control the influence of the tf-idf, citation count, & years since publication on the similarity score, respectively.
- C_i is the citation count for document i.
- Y_i is the number of years since document i was published.

References

[1] David A. Grossman and Ophir Frieder. *Information Retrieval: Algorithms & Heuristics*. 2nd Edition. Springer, 2004. DOI: 10.1007/978-1-4020-3005-5.