Information Retrieval – Assignment 5

Ramtin Samavat

Task 1: IR Models

Question 1

No, this is not possible with a term document matrix. The term document matrix only contains information about how frequently each term occurs in a document and is not sufficient enough to search for terms in the same ordering as in the query. To be able to provide the capability of searching for phrases, we need to store the positional information of each term in the document.

Question 2

No, we cannot provide a ranking of documents using a term-document matrix with Boolean values. The matrix does not give information about how many times each term appears in a document, only whether they appear or not. Therefore, we can only determine whether a document is relevant to a query, but not how relevant it is. To rank documents, we need to store either the raw term frequency or the term frequency combined with inverse document frequency. This allows us to know how important a term in the query is in a document and use that information to rank documents by their relevance.

Question 3

No, it is not a wise choice to consider. The problem with term-document matrices, is that they are often sparse, meaning that most documents contain only a small subset of the total vocabulary, leaving the majority of matrix entries as zeroes. A better solution when implementing the term-documents matrix, is to store only the non-zero entries. We can also consider using an inverted index, where each term contains a list of documents that contain them. However, an inverted index is not strictly a term-document matrix, but it is functionally equivalent for most retrieval tasks.

Question 4:

In a recall-oriented task we want to retrieve all relevant documents, even if some irrelevant items are included. In a precision-oriented task, on the other hand, we want the top results to be highly relevant, even if it means some relevant documents are missed. A search functionality for files and folders in an operating system is most likely a precision-oriented task, because the users are often searching for specific files or folders and expect the most relevant results. However, in a case where the users prioritize all potentially relevant files or folders and cannot afford to miss a single relevant result, the task becomes recall oriented.

Since an operating system can contain millions of files with many unique terms, using a standard term-document matrix is not practical. However, it can be an appropriate choice if we only store the non-zero entries in the matrix. Another and more efficient solution is to implement an inverted index.

Question 5

No, this is not possible with only a term-document matrix, because it does not contain information about relationships between terms. To implement suggestive query completion, we need to know how terms are correlated, which can be represented in a term-term correlation matrix. In such a matrix, each entry measures how often two terms co-occur, quantifying how strongly related they are.

Task 2: Boolean retrieval

a) Boolean term-document matrix

	hbase	open	source	key	value	store	dynamodb	proprietary	redis	popular	not
d1	1	1	1	1	1	1	0	0	0	0	0
d2	0	0	0	1	1	1	1	1	0	0	0
d3	0	0	0	0	0	0	0	0	1	1	0
d4	0	0	0	0	0	0	0	0	1	1	1

b) Find result for given query:

a) q = key ∧ value ∧ store → result set: {d1, d2}

b) q = (key ∧ value ∧ store) ∧ (not ∨ popular) → result set: {} – *no documents match*

Task 3: Evaluation metrics

Relevant documents in total: 4

a) Precision@5 = (Number of relevant documents in the top k results) / k

- Team 1: 3 / 5 = 0.6
- Team 2: 2 / 5 = 0.4

b) Recall@5 = (Number of relevant documents in top k) / (Total number of relevant documents)

- Team 1: 3 / 4 = 0.75
- Team 2: 2 / 4 = 0.50

c) Mean Average Precision (MAP) = 1 / | R | * sum(precision at each relevant rank)

- Team 1:
  - Relevant positions (positions for relevant docs): 1, 3, 5, 8.
  - Formula: (Number of relevant documents) / Position
  - Rank 1: 1 / 1 = 1.00
  - Rank 3: 2 / 3 = 0.67
  - Rank 5: 3 / 5 = 0.60
  - Rank 8: 4 / 8 = 0.50
  - MAP = (1.00 + 0.67 + 0.60 + 0.50) / 4 = 0.69
- Team 2:
  - Relevant positions: 2, 5, 6.
  - Rank 2: 1 / 2 = 0.50
  - Rank 5: 2 / 5 = 0.40
  - Rank 6: 3 / 6 = 0.50
  - MAP = (0.50 + 0.40 + 0.50) / 4 = 0.35

d) Mean Reciprocal Rank (MRR) = 1 / (Position of the first relevant document)

- Team 1: 1 / 1 = 1.00
- Team 2: 1 / 2 = 0.50

Task 4: Local Association and Query Expansion

a) Construct the unnormalized association matrix.

Step 1: Build a term-document matrix (TDM) where each entry is the raw term frequency.

	Oslo	To	Bergen	Trondheim	Stockholm	Helsinki
d1	1	2	2	1	0	0
d2	0	2	1	0	1	1
d3	2	2	0	0	1	1

Step 2: Find the local association matrix by multiplying TDM with TDM transformed.

Since the rows in the TDM are documents and the columns are terms, LAM = $TDM^T \times TDM$.

Association matrix:

	Oslo	To	Bergen	Trondheim	Stockholm	Helsinki
Oslo	5	6	2	1	2	2
To	6	12	6	2	4	4
Bergen	2	6	5	2	1	1
Trondheim	1	2	2	1	0	0
Stockholm	2	4	1	0	2	2
Helsinki	2	4	1	0	2	2

b) Query expansion

Query = “Oslo”

Based on the association matrix, the query should be expanded with the terms Bergen, Stockholm, or Helsinki. These terms have the highest correlation values with “Oslo” and are therefore most likely to be associated. The query term itself and non-meaningful stop words (like “to”) are excluded from consideration.

Task 5: Text indexing

a) Construct an inverted list for given sentence

The mind is wandering around the mind and does not mind that every mind does wander around.

Vocabulary (without stopwords) = [mind, wandering, around, does, not, that, every, wander]

Inverted list:

Vocabulary	Occurrences (position in sentence)
mind	[5, 34, 52, 68]
wandering	[13]
around	[23, 85]
does	[43, 73]
not	[48]
that	[57]
every	[62]
wander	[78]

b) Difference between vocabulary trie and suffix trie

A trie is a tree-based data structure used for efficiently storing and retrieving words. In a trie, each node represents a character, and the path between nodes forms words. To retrieve a word, we start at the root node and follow the path character by character until we find the word.

A vocabulary trie is a prefix-based dictionary of words. Each branch represents a shared prefix between words. For example, words like "to", "top", and "toy" share the prefix "to", which is stored once. A suffix trie, on the other hand, stores all suffixes of a text. For example, the word "banana" has the suffixes "banana", "anana", "nana", "ana", "na", and "a", and they are all stored in the suffix trie. Since the suffix trie stores a lot more information, it takes up more memory space compared to a prefix trie, but it allows efficient pattern matching of substrings.