Lab 1: Information Retrieval Models

Course: Introduction to Information Retrieval

Topics: Boolean Model, Vector Space Model, Probabilistic Model (BM25)

Submission: Submit both .ipynb file and .ipynb converted to PDF

Submissions with following cases will get a zero

- Any compilation error in the notebook
- Missing output for any of the programming cells. There shoould be an output for every code cell

==========

Part A: Theoretical Questions (35 Marks)

==========

Q1: Define the Bag of Words model and explain its role in vector space representation.

Theoretical no code/program needed

Answer: The bag of words model represents documents as a collection of words. This stores the words as well as the frequency of the words from the document. Bag of words model is also unordered.

Example: a document containing the sentence "I love this this course", bag of words would be represented by {"love":1,"I":1,"this":2,"course":1}

It's role in vector space representation is that we can use the words and frequencies and store them in vectors, making them useful in mathematical calculations as normalization of a document, or applying algorithms to rank multiple documents.

Example:

Document: "I love this this course"

Vocabulary: ["I", "course", "this", "love"]

Document: [1,1,2,1]

One limitation is that it cannot differentiate between positive/ negative connotations in phrases such as "not good" since Bag of words is unordered. The unordering factor also ignores grammar.

Q2: Explain each component of TF-IDF formula.

Theoretical no code/program needed

Answer:

So, we know that TF-IDF(t,d) = TF(t,d) x IDF(t) = (Number of times term t appears in document d) x log(N / DF(t))

For the TF(t,d) portion, it is simply calculating the number of terms t in a specific document, this means if we have a corpus of documents, we can find a TF score for each document for a specific term

for the IDF(t) portion, it helps us understand how rare a term is across our corpus. This is because we divide the total number of documents by only the documents that contain term t. the log is more of a term to smooth the scaling, so rare terms may not be overweighting.

Now, when we multiple TF and IDF, the score we get tells us how important the term is in a specific document relative to the entire corpus. A high TF-IDF score tells us that the term appears more in the document relative to the corpus, and opposite happens when the TF-IDF score is low.

Q3: What is cosine similarity? Why is it preferred over Euclidean distance in IR?

Theoretical no code/program needed

Cosine similarity is used to measure the angle between two vectors, defined by:

$$Cosine(\theta) = \frac{(q \cdot d)}{\|\mathbf{q}\| \, \|\mathbf{d}\|}$$

Where q represents the query vector and d represents the document vector

The numerator (dot product) shows how "aligned" the two vectors are. The denominator removes the factor of vector magnitude. This is crucial because a query and document vectors will in most cases be very different magnitudes, meaning that the length of one vector will affect the similarity score, which we do not want, hence why we include the denominator.

The point mentioned above is why cosine similarity is preferred over euclidean distance since euclidean distance factors in the magnitude of vectors. We may have a document that is similar to a query but large in magnitude (docA), and another which is not as similar, but smaller in magnitude (docB). Euclidean distance would tell us that docA is "farther" from the query than docB, which would not be correct.

Q4: Discuss the impact of document length normalization on term weighting and ranking.

Theoretical no code/program needed

Document length normalization is needed when term weighting or ranking due to the varying size of documents. A document that is very long may have large frequencies of terms, but may not be a relevant document. When ranking documents, we would like to favour documents that have a higher ratio of term t compared to the document, rather than just the number of occurences of t.

We can normalize the frequency of terms in a document by dividing the term frequency of term t in document d by the length of d, this will normalize the frequencies to the range 0-1, giving an even comparison when weighting terms and ranking, rather than being biased towards large documents. Term density is rewarded in this scenario rather than term occurrence in a document.

In BM25, we normalize by dividing the length of document d by the average length of a document in the corpus (also we have b as a parameter).

Q5: What is the Probability Ranking Principle? How does it guide document ranking?

Theoretical no code/program needed

The probability ranking principle works by ranking documents based on the terms it contains that are also in the query, using logarithmic probability. The base algorithm calculates the score of a document based on how relevant it is. The score is found by:

1. for each term t in our query q, we divide the probability that t appears in relevant documents r, by the probability of t in non-relevant documents ¬r.
2. We find the logarithm score of the division
3. we sum the logarithmic score over all the terms in q to find our final score.

This way, because we find the probability of relevant vs non relevant documents, we can say that the higher the score, the higher estimated probability it has of being relevant to the users query.

We must note that this principle on its own ignores the frequency of a term appearing in a document, it only notices if it appears or not. Another limitation is the fact that it does not take into account document length. So a longer document may be favoured over a short document with a higher rate of occurence of a term.

Q6: Explain the intuition behind BM25 and how it improves over TF-IDF.

Theoretical no code/program needed

The BM25 is a probabilistic model used as a ranking function using probability to determine relevant documents given a specific query. The problem with TF-IDF is that is assumes that if a term is more frequently mentioned in a document, it is more important linearly. Another limitation is very long documents also contain multiple instances of a specific term, but that's just due to how long the document is, not because it is actually a relevant document.

BM25 fixes these issues. Firstly, the problem with higher term frequency leading to higher relevance is addressed by a smoothing function (with parameter k1), which allows higher scores for higher frequencies, but quickly levels out. This is because extra occurrences may not add as much information after a certain point.

BM25 also introduces document length normalization (found by dividing length of document d by the average length of a document). This allows for understanding that overly long documents may not be relevant, and it notices concise documents which are more related to our given query)

$BM25 = \sum_{t \in q} IDF(t) \times TFSaturation \times LengthNorm$

is the general formula of BM25, as we can see, it introduces the saturation or "smoothing" term for TF, and the normalization of documents

Q7: List two advantages and two limitations of probabilistic models in IR.

Theoretical no code/program needed

Advantages:

1. Probabilistic models adapts to feedback. It can use user input to improve its ranking and update its probabilities on relevant/ non-relevant documents.
2. The probabilistic model can incorporate frequency and document length to make its decisions. Other models may be naive in thinking that only term frequency itself matters, not accounting for the density of terms per document on relevance. Probabilistic models can fix that issue by introducing normalization and smoothing factors to account for overly long documents with terms appearing a lot of times.

Disadvantages:

1. Probabilistic models such as BM25 require parameter tuning (for k and b). This may take some time to determine which combination of parameters work best when given the scenario.
2. The initial results of a probabilistic model may not be accurate. This is because it has no previous knowledge on relevance. With user feedback, over time the model improves but it is not great in the beginning.

==========

Part B: Python Implementation Tasks

==========

65 Marks

Q8 (10 marks) Write a Python function that takes a list of documents (each document is a string) and returns an inverted index. The inverted index should be a dictionary where each term maps to a list of document IDs in which the term appears.

Example Input documents = ["the cat sat on the mat", "the dog sat on the log", "the cat chased the dog"]

✅ Expected Output { 'the': [0, 1, 2], 'cat': [0, 2], 'sat': [0, 1], 'on': [0, 1], 'mat': [0], 'dog': [1, 2], 'log': [1], 'chased': [2] }

In [1]:
```python
def build_inverted_index(documents):
    """
    Build an inverted index from a list of documents.

    Parameters:
    - documents (list of str): A list of raw text documents

    Returns:
    - dict: Inverted index {term: [doc_ids]}
    """
    output_dict = {}
    # Iterate over each document
    for idx, doc in enumerate(documents):
        # get list of words per document
        list_of_words = doc.split()
        # Iterate over words, and add index into dictionary value list
        for word in set(list_of_words):
            word = word.lower()
            if word not in output_dict:
                output_dict[word] = []
            output_dict[word].append(idx)
```

```
    return output_dict
```

In [2]:
```
documents = ["the cat sat on the mat", "the dog sat on the log", "the cat chased the dog"]
res = build_inverted_index(documents)
print(res)
```

```
{'mat': [0], 'on': [0, 1], 'cat': [0, 2], 'sat': [0, 1], 'the': [0, 1, 2], 'log': [1], 'dog': [1, 2], 'chased':
[2]}
```

Q9 (10 marks) Write a Python function that takes inverted index of documents as input and a query string, and returns the list of document IDs that satisfy the query. The query can use the operators AND, OR, and NOT between terms.

Sample Inverted Index

index = {'the': [0, 1, 2], 'cat': [0, 2], 'sat': [0, 1], 'on': [0, 1], 'mat': [0], 'dog': [1, 2], 'log': [1], 'chased': [2]}

Example Queries and Expected Outputs

Query 1: "cat AND dog"

Documents containing both 'cat' and 'dog'

cat → [0, 2], dog → [1, 2]

Intersection → [2]

Expected Output: [2]

Query 2: "cat OR dog"

Union of documents with 'cat' or 'dog'

cat → [0, 2], dog → [1, 2]

Union → [0, 1, 2]

Expected Output: [0, 1, 2]

Query 3: "cat AND NOT dog"

cat → [0, 2], dog → [1, 2]

NOT dog → all docs except [1, 2] → [0]

Intersection → [0]

Expected Output: [0]

Query 4: "sat OR chased"

sat → [0, 1], chased → [2]

Union → [0, 1, 2]

Expected Output: [0, 1, 2]

Query 5: "mat AND NOT sat"

mat → [0], sat → [0, 1]

NOT sat → [2]

Intersection → mat ∩ NOT sat → []

Expected Output: []

In [3]:
```
def query_inverted_index(index, query):
    """
    Evaluate a Boolean query using an inverted index.

    Parameters:
    - index (dict): Inverted index {term: [doc_ids]}
    - query (str): Boolean query using 'AND', 'OR', 'NOT'
```

```
    Returns:
    - list of int: Sorted list of document IDs that satisfy the query
    """
    # Split query into list of words
    tokens = query.lower().split()
    every_word = set(word for doc in index.values() for word in doc)

    # Retrieve indices of terms mentioned
    index_list = []
    for t in tokens:
        if t not in ("and","or","not"):
            index_list.append(index.get(t, []))

    # Logic output
    output = []
    if "and" in tokens:
        if "not" in tokens:
            output = list(set(index_list[0]) & (every_word - set(index_list[1])))
        else:
            output = list(set(index_list[0]) & (set(index_list[1])))

    elif "or" in tokens:
        if "not" in tokens:
            output = list(set(index_list[0]) | (every_word - set(index_list[1])))
        else:
            output = list(set(index_list[0]) | (set(index_list[1])))
    return output
```

In [4]:
```
# Show results for all provided test cases
index = {'the': [0, 1, 2], 'cat': [0, 2], 'sat': [0, 1], 'on': [0, 1], 'mat': [0], 'dog': [1, 2], 'log': [1], '
print(query_inverted_index(index,"cat AND dog"))
print(query_inverted_index(index,"cat OR dog"))
print(query_inverted_index(index,"cat AND NOT dog"))
print(query_inverted_index(index,"sat OR chased"))
print(query_inverted_index(index,"mat AND NOT sat"))
```

```
[2]
[0, 1, 2]
[0]
[0, 1, 2]
[]
```

Vector Space Model (20 marks)

Q10. (20 marks) Implement a TF-IDF vectorizer from scratch for a small corpus of 3 documents (corpus) and 1 query.

In [5]:
```
import math
def compute_tf_idf(corpus, query):

    # Prepare TF-IDF vectors for each document and the query following the document TF-IDF document posted on D
          # Print all TF-IDF vectors

    # Part 1: Find Term frequency for each document and term, including corpus
    freq_dict  = {}
    for idx, doc in enumerate(corpus):
        list_words = doc.split()
        for word in list_words:
            if word not in freq_dict:
                freq_dict[word] = [0] * len(corpus)
            freq_dict[word][idx] += 1

    # Part 2: IDF
    N = len(corpus)
    idf_vector = []
    for word in freq_dict.keys():
        DF_term = sum(1 for num in freq_dict[word] if num != 0)
        idf = round(math.log(N / DF_term),3)
        idf_vector.append(idf)

    # Part 3: TF-IDF
    tf_idf_vec = []
    for idx, nun in enumerate(corpus):
        doc_vec = []
        idf_idx = 0
        for word in freq_dict.keys():
            tf_idf = freq_dict[word][idx] * idf_vector[idf_idx]
            doc_vec.append(tf_idf)
            # This increments the index of the idf vector so we don't keep multiplying by 1 value only
            idf_idx += 1
```

```
        tf_idf_vec.append(doc_vec)

    # Part 4: print all TF-IDF vectors:
    print("Words", freq_dict.keys())
    for idx,vec in enumerate(tf_idf_vec):
        print("Document", idx+1, "TF-IDF", tf_idf_vec[idx])

    # Part 5: Cosine Similarity
    # 5a: prepare document and query vectors, get rid of stopwords
    vocabulary = freq_dict.keys()
    stopwords = ["the","on"]
    vocabulary = [w for w in vocabulary if w not in stopwords]

    document_vectors = []
    for doc in corpus:
        words = doc.split()
        vector = [words.count(term) for term in vocabulary]
        document_vectors.append(vector)

    query_vector = [query.split().count(term) for term in vocabulary]

    # 5b: calculate similarity between query and each document
    cosine_similarities = []
    for idx,doc_vec in enumerate(document_vectors):
        dot_prod = sum(x*y for x,y in zip(doc_vec, query_vector))
        doc_norm = math.sqrt(sum(x*x for x in doc_vec))
        query_norm = math.sqrt(sum(x*x for x in query_vector))
        cs = round(dot_prod / (doc_norm*query_norm),3)
        cosine_similarities.append((f"Document {idx+1} Cosine Similarity",cs))

    cosine_similarities.sort(key=lambda x: x[1], reverse=True)

    return cosine_similarities
```

Q11. (5 marks) Compute cosine similarity between the query and each document (at least three). Show the ranked results.

In [6]:
```
corpus = ["the cat sat on the mat", "the dog sat on the log", "the cat chased the dog"]
compute_tf_idf(corpus,'cat dog')
```

```
Words dict_keys(['the', 'cat', 'sat', 'on', 'mat', 'dog', 'log', 'chased'])
Document 1 TF-IDF [0.0, 0.405, 0.405, 0.405, 1.099, 0.0, 0.0, 0.0]
Document 2 TF-IDF [0.0, 0.0, 0.405, 0.405, 0.0, 0.405, 1.099, 0.0]
Document 3 TF-IDF [0.0, 0.405, 0.0, 0.0, 0.0, 0.405, 0.0, 1.099]
```

Out[6]:
```
[('Document 3 Cosine Similarity', 0.816),
 ('Document 1 Cosine Similarity', 0.408),
 ('Document 2 Cosine Similarity', 0.408)]
```

Probabilistic Model – BM25

Q12. (15 marks) Implement a simplified BM25 scoring function for a small corpus. Using the formula in slides:

In [7]:
```
def bm25_score(query, corpus, k=1.5, b=0.75):
    # Returns BM25 scores for each document
    # Part 1: initialize constants (average length of documents)
    N = len(corpus)
    doc_split = [doc.split() for doc in corpus]
    avg_doc_len = sum(len(doc) for doc in doc_split) / N

    # Part 2: Find frequency for terms in corpus
    freq_dict = {}
    for idx,docs in enumerate(doc_split):
        for word in docs:
            if word not in freq_dict:
                freq_dict[word] = [0] * len(corpus)
            freq_dict[word][idx] += 1

    # Part 3: Find bm25 scores
    bm25_scores = []
    query_words = query.split()

    for idx, doc in enumerate(doc_split):
        score = 0
        len_doc = len(doc)

        # we only calculate terms that are an element of query q
        for q_term in query_words:
```

```
            if q_term in freq_dict:
                # Find IDF
                DF_term = sum(1 for num in freq_dict[q_term] if num != 0)
                idf = math.log(N / DF_term)
                # Number of occurences of term in document
                f = freq_dict[q_term][idx]
                # bm25 score
                score += idf*((f*(k+1))/(f+k*(1-b+b*(len_doc/avg_doc_len))))
        bm25_scores.append((f"Document {idx+1} BM25 Score",round(score,4)))

    bm25_scores.sort(key=lambda x: x[1], reverse=True)
    return bm25_scores
```

Q13. (5 marks) Compare BM25 scores with TF-IDF scores for the same query and corpus. Discuss the differences.

In [8]:
```
bm25_score("cat dog", corpus)
```

Out[8]:
```
[('Document 3 BM25 Score', 0.8563),
 ('Document 1 BM25 Score', 0.395),
 ('Document 2 BM25 Score', 0.395)]
```

We notice that the rankings of BM25 and TF-IDF seem to rank quite similarly. BM25 algorithm accounts for document length, so it will perform really well compared to TF-IDF when given varying sizes of documents, with different ratios of terms appearing. TF-IDF did not have any normalization factors, although Cosine similarity did.