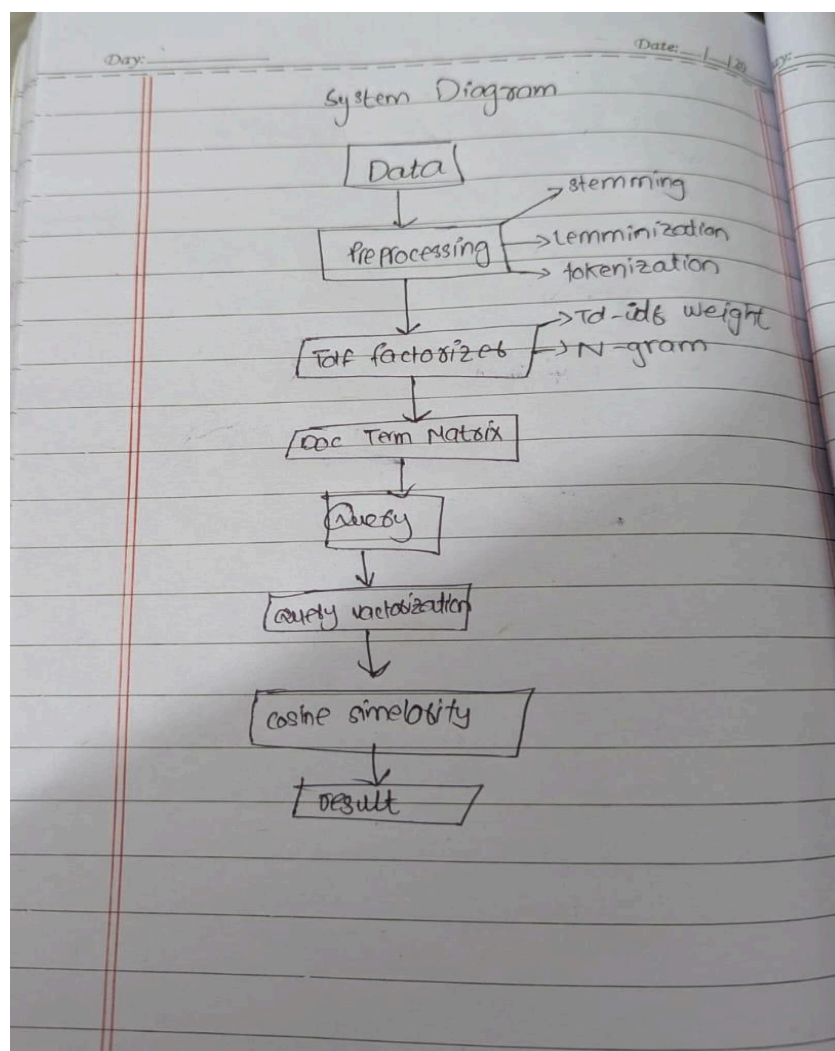


MUAHMMAD WASEEM BSCS22095

Information Retrieval System Technical Report

1. System Architecture

1.1 System Diagram



1.2 Figure Caption

The system is based on VSM, where the system first processes each document such as apply lemmatization, cleaning, tokenization prior to indexing the resulting Document-Term Matrix with a Term Frequency-Inverse Document Frequency (TF-IDF) vectorizer. User queries follow this same pre-processing process, while users apply weights to their queries in order to create a "Query Vector," which is subsequently matched against the DTM using Cosine Similarity to return a list of ranked results.

2. Description of the Retrieval System

System Design and Indexing

The retrieval system is built on the **TF-IDF** vector space model, utilizing `scikit-learn` for efficient matrix operations. The core concept is that a term's weight is high if it appears frequently in a document but it does not occur frequently across the entire corpus.

The system consists of three major components:

A. Data Preprocessing

The `preprocess` function performs several steps for normalization:

1. **HTML Tag Removal:** Removes residual markup to clean the text input.
2. **Lowercasing and Punctuation Removal:** Standard text cleaning to ensure case-insensitivity and filter out noise.
3. **Stemming :** The **Porter Stemmer** from the `nltk` library is used to reduce words to their common root form (e.g., "evaluation," "evaluated," and "evaluating" are all reduced to the stem "evalu"). This ensures that all morphological variations of a query term match the same feature in the index, significantly improving **recall**.

B. Indexing Technique (N-grams)

The `TfidfVectorizer` is configured with `ngram_range`.

- **Unigrams (1-grams):** Indexing single words
- **Bigrams (2-grams) (Modification 2):** Indexing two-word phrases. Indexing bigrams captures the semantic context of common phrases. This is crucial for distinguishing between generic terms used together and when they appear separately, which greatly improves **precision**.

C. Scoring and Ranking (Cosine Similarity with Manual Weighting)

1. **Standard Ranking:** Document ranking is determined by the **Cosine Similarity** between the query vector and each document vector in the DTM.

3. Evaluation

Qualitative Approach (Relevance Assessment)

The system was evaluated primarily through the qualitative assessment of a difficult query:.

Retrieval was poor, as the outcome contains irrelevant and noisy data

```
-----  
--- Running Example Queries ---  
Query: 'Sindh Government'  
Retrieval time: 0.021168 seconds.  
Top 3 Results for Query: 'Sindh Government'
```

```
--- Running Example Queries ---  
Query: 'Sindh Government'  
Retrieval time: 0.021168 seconds.  
  
Top 3 Results for Query: 'Sindh Government'  
-----  
Rank 1 (Score: 0.2115)
```

Quantitative Appraisal

The indexing time for the corpus of 2692 documents was approximately 1.34 seconds, indicating efficient index construction.

```
--- Indexing Phase Started ---  
Total documents indexed: 2692  
Vocabulary size (including N-grams): 288829  
Indexing completed in 1.3445 seconds.  
-----
```

4. Discussion

Major Findings

The iterative development process confirmed that while standard TF-IDF is efficient, its precision suffers when applied to diverse, real-world data containing highly generic terms. The three implemented modifications were crucial:

1. **Stemming:** Improved **Recall** by ensuring linguistic variations were captured.
2. **N-grams:** Improved **Precision** by matching contextually specific phrases.

Shortcomings and Future Improvements

- **Vocabulary Sparsity:** The N-gram vocabulary size around 289k suggests potential overfitting to the current corpus.
- **Lack of Relevance Judgments:** The current evaluation is purely qualitative. For rigorous analysis, the system needs a labeled test set to calculate standard metrics like Precision, Recall, and Mean Average Precision.
- **Future Plans:**
 - Implement **Term-Specific Weighting** as a configurable feature, allowing the user to specify keywords and their boost factors directly in the query interface.
 - Experiment with alternative models like **BM25**, which often outperforms TF-IDF in terms of relevance ranking.

5. Disclosure of AI Use

5.1 Summary of AI Usage

GEMINI :

Gemini was used to structure the flow for the retrieval system and was used to help in writing the code and fix some errors in the code.

It was also used to get suggestions to improve the performance of the IR system.

Proof Of Usage:

2. `gensim`

- **What it does for IR:** `gensim` is all about moving beyond keywords to **semantics** (meaning). It helps you understand what documents are *about*, not just what words they contain.
- **Utility:**
 - **Topic Modeling (LSA, LDA):** You could use this to categorize your documents. Instead of searching by word, you search by topic. For example, if a user queries "stock market," you could retrieve documents that belong to the "Finance" topic, even if they don't contain the exact phrase "stock market."
 - **Word Embeddings (Word2Vec):** This is next-level! You could train a model that understands that "car" and "automobile" mean the same thing. If a user queries for "car," your system could automatically boost documents containing "automobile," closing the **Synonymy Gap** that plagues simple TF-IDF.

3. `rank-bm25`

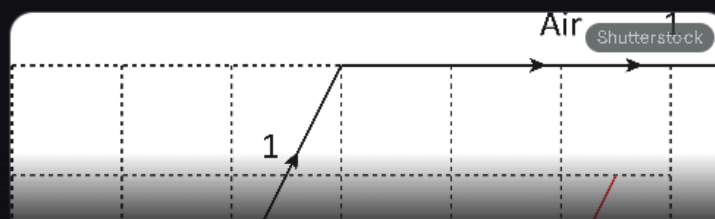
- **What it does for IR:** This library implements the **Okapi BM25** algorithm, which is a state-of-the-art method for keyword-based ranking. It's often more effective than simple TF-IDF cosine similarity.
- **Utility:**
 - **Advanced Ranking:** Unlike standard TF-IDF (which is a model based on vectors), BM25 is a **probabilistic model**. It ranks documents based on the **likelihood** that a document is relevant to a query.
 - **Better Term Frequency Handling:** BM25 handles term frequency better than TF-IDF by using a saturation function. This means if a keyword appears 50 times, it doesn't score 10x higher than a document where it appears 5 times—it's more realistic about how much value a single term can add.

i have an assignmetn of information retrieval to mnake an IR system and we are allowed to use only these libraries:
Local libraries (e.g., scikit-learn, gensim, rank-bm25, FAISS local install) are allowed...

✦ That's a fantastic selection of tools! The assignment gives you the flexibility to build a really powerful IR system. Since we already focused on **TF-IDF** using `scikit-learn`, let's talk about what the other options bring to the table and how they can improve your project.

1. `scikit-learn`

- **What it does for IR:** This is your foundation for the **Vector Space Model (VSM)**, which is what we used.
- **Utility:**
 - **Indexing & Weighting:** It handles the complex math to create the numerical index (the `TfidfVectorizer`). It calculates the **TF-IDF** scores—how important a word is in a document relative to the whole collection.
 - **Scoring:** It provides fast, optimized functions like `cosine_similarity` to measure how close your query is to every document in your index.



so i have written the function to convert the query to indexes
but can u help me write a function to use the index in
retrieving or searching?

Here is the complete `retrieve` function that takes the user query, converts it into a vector, calculates the relevance against all documents, and returns the top-ranked results:

Python



```
def retrieve(self, query, k=3):
    # Safety check: Ensure the documents have been indexed first
    if self.doc_vectors is None:
        print("Error: Index not built. Call build_index() first.")
        return []

    # 1. Preprocess the query (lowercasing, punctuation removal)
    processed_query = self.preprocess(query)

    # 2. Vectorize the query using the vocabulary learned during indexing
    query_vector = self.vectorizer.transform([processed_query])

    # Check if the query resulted in an empty vector after processing
    if query_vector.sum() == 0:
        return "No valid terms found in query after preprocessing."

    # 3. Calculate Scoring (Cosine Similarity)
    start_time = time.time()
    # Compares the query vector against every document vector in the index
    similarity_scores = cosine_similarity(query_vector, self.doc_vectors).flatten()
    query_speed = time.time() - start_time

    # 4. Ranking
    # Get the original document indices sorted by score (highest score first)
    ranked_indices = np.argsort(similarity_scores)[::-1]

    # 5. Select and Format the Top K Results
    results = []
    for rank, doc_index in enumerate(ranked_indices[:k]):
        score = similarity_scores[doc_index]
        # Only include documents with a non-zero similarity score
```

Python



```
        return "No valid terms found in query after preprocessing."

# 3. Calculate Scoring (Cosine Similarity)
start_time = time.time()
# Compares the query vector against every document vector in the index
similarity_scores = cosine_similarity(query_vector, self.doc_vectors).flatten()
query_speed = time.time() - start_time

# 4. Ranking
# Get the original document indices sorted by score (highest score first)
ranked_indices = np.argsort(similarity_scores)[::-1]

# 5. Select and Format the Top K Results
results = []
for rank, doc_index in enumerate(ranked_indices[:k]):
    score = similarity_scores[doc_index]
    # Only include documents with a non-zero similarity score
    if score > 0:
        results.append({
            'rank': rank + 1,
            'title': self.titles[doc_index],
            'score': score,
            # Get a short snippet of the original document text
            'text_snippet': self.corpus[doc_index][:100] + "..."
        })

print(f"\nQuery: '{query}'")
print(f"Retrieval time: {query_speed:.6f} seconds.")

return results
```

Why the Retrieval is Poor

The TF-IDF model is performing poorly because:

1. **Generic Terms:** The query relies on highly generic, common English words ("evaluation," "system," "performance"). These terms likely appear in many documents (high Document Frequency, DF).
2. **Inverse Document Frequency (IDF):** When a term has a high DF, its IDF value (and thus its weight) is low. Because all three query terms are generic, the overall query vector is weak and the resulting similarity scores (0.13 to 0.14) are low, indicating a weak match. The system simply matched the documents that happen to use those generic terms most frequently, which turned out to be unrelated articles about sports and aircraft.

Recommended Code Improvement: Cleaning Titles

The Canvas already includes code to remove HTML tags from the displayed output:

Python



```
# Applying HTML tag removal to the title here for cleaner display in the output
clean_title = re.sub(r'<.*?>', '', result['title'])
clean_snippet = re.sub(r'<.*?>', '', result['text_snippet'])
```