

Ranking Documents Using TF-IDF (Assignment-02)



Session: 2021 – 2025

Submitted by:

Muhammad Yaqoob 2021-CS-118

Supervised by:

Dr Syed Khaldoon Khurshid

Department of Computer Science

University of Engineering and Technology UET Lahore

Table of Contents

1. Introduction.....	4
2. Objective.....	4
3. Document Collection	4
4. Real-World Scenario	4
5. Finding TF-IDF of a document.....	5
5.1. Clean data and Tokenize	5
5.2. Find TF.....	5
5.3. Find IDF.....	6
5.4. Calculate TF-IDF	6
6. Code TF-IDF.....	7
6.1 Load Documents	7
6.2. Clean data and Tokenize	7
6.3. Find TF.....	8
6.4. Find IDF.....	8
6.5. Find TF-IDF	9
7. Query Handling and Ranking	10
7.1. Preprocessing and Query TF-IDF Vector.....	10
7.2. Calculate Cosine Similarity	10
8. User Interface.....	12
9. Data Flow Diagram.....	13
7. Conclusion	13

Table Of Figures

Figure 1: Clean data and Tokenize.....	8
Figure 2: Finding TF for words.....	8
Figure 3: Finding IDF for words.....	9
Figure 4: Finding TF-IDF	10
Figure 5: User Interface	13
Figure 6: Dataflow Diagram	13

1. Introduction

The exponential growth of digital content has necessitated efficient information retrieval systems. Term Frequency-Inverse Document Frequency (TF-IDF) is a widely used method for ranking documents based on their relevance to a user query.

More importantly, TF-IDF is a technique which is used to find meaning of sentences consisting of words.

This assignment implements a TF-IDF and a ranking system that uses TF-IDF to process documents, calculate their relevance, and rank them in response to user input.

2. Objective

The objective of this assignment is to design and implement a document ranking system using the TF-IDF method. Main objective to this assignment to me is:

- How to find meaning of sentences and documents

3. Document Collection

The dataset comprises plain-text files stored in a directory. Each file represents a document, and its filename serves as its title. This modular design allows for flexibility in adding, modifying, or removing documents. A function reads and preprocesses these files to prepare them for the ranking process. Titles of the documents collected are written below. Each document has content.

- Artificial Intelligence
- Environmental Conservation
- Goods and Services
- Nature and Culture
- Pakistan's Evolution
- Pakistan's Global Links
- Renewable Energy
- Space Exploration

4. Real-World Scenario

Consider a scenario where a content marketing team is working on improving the visibility of their company's website in search engines. The team wants to analyze a set of blog articles to identify the most important keywords that resonate with their audience and optimize these articles for search engines like Google.

Problem:

Search engines rank web pages based on various factors, including the relevance of their content to a user's query. The team realizes that their current keyword identification process, which relies on raw word counts, is ineffective. Common words like "the," "is," and "and" frequently appear in their articles but do not add value to understanding the content's context. This makes it difficult to determine which keywords truly differentiate their content from competitors.

Solution:

The team employs TF-IDF (Term Frequency-Inverse Document Frequency) to address these challenges. TF-IDF helps highlight terms that are uniquely significant to their blog articles by considering:

- Term Frequency (TF): How often a term appears in a single article.
- Inverse Document Frequency (IDF): How unique a term is across a collection of articles.

Example:

Imagine the team has three blog articles:

- "10 Tips for a Beautiful Day Outdoors"
- "The Ultimate Guide to Hiking in Rainy Weather"
- "Discovering the Beauty of Remote Mountains"

The word "beautiful" appears twice in the first article and once in the third. However, it does not appear in the second article, making it more relevant to the theme of outdoor enjoyment and beauty. Meanwhile, common words like "the" and "in" appear frequently across all articles but hold no distinguishing value.

Using TF-IDF, the machine identifies "beautiful," "hiking," and "mountains" as highly relevant terms, guiding the team to focus on these for search engine optimization.

Real-World Impact:

With TF-IDF, the marketing team can:

- Enhance the relevance of their content by targeting high-value keywords.
- Improve their website's search engine rankings by aligning content with user intent.
- Save time by filtering out low-impact, frequently occurring words that do not add value.

By applying TF-IDF, the team bridges the gap between raw data and meaningful insights, making their content strategy data-driven and highly effective.

5. Finding TF-IDF of a document

- Clean data / Preprocessing: Normalize data (all lower case)
- Tokenize words with frequency
- Find TF for words
- Find IDF for words
- Vectorize vocab by finding TF-IDF

Let's cover an example of 3 documents:

- **Document 1** It is going to rain today.
- **Document 2** Today I am not going outside.
- **Document 3** I am going to watch the season premiere.

To find TF-IDF we need to perform the steps we laid out above, let's get to it.

5.1. Clean data and Tokenize

Table 1: Vocab of document

Word	Count
going	3
to	2
today	2
i	2
am	2
it	1
is	1
rain	1

5.2. Find TF

TF = (Number of repetitions of word in a document) / (Total number of words in a document)

Table 2: TF for the documents

Words	Document 1	Document 2	Document 3
going	0.16	0.16	0.12
to	0.16	0	0.12
today	0.16	0.16	0
i	0	0.16	0
am	0	0.16	0.12
it	0.16	0	0
is	0.16	0	0
rain	0.16	0	0

5.3. Find IDF

$IDF = \log[(\text{Number of documents}) / (\text{Number of documents containing the word} + 1)]$

Table 3: IDF for documents

Word	IDF Value	IDF Value
going	$\log(3/3)$	0
to	$\log(3/2)$	0.41
today	$\log(3/2)$	0.41
i	$\log(3/2)$	0.41
am	$\log(3/2)$	0.41
it	$\log(3/1)$	1.09
is	$\log(3/1)$	1.09
rain	$\log(3/1)$	1.09

Table 4: IDF Value and TF value of 3 documents.

Word	IDF Value	Words	Document 1	Document 2	Document 3
going	0	going	0.16	0.16	0.12
to	0.41	to	0.16	0	0.12
today	0.41	today	0.16	0.16	0
i	0.41	i	0	0.16	0
am	0.41	am	0	0.16	0.12
it	1.09	it	0.16	0	0
is	1.09	is	0.16	0	0
rain	1.09	rain	0.16	0	0

5.4. Calculate TF-IDF

$TF\text{-}IDF = TF * IDF$

We can use TF-IDF to compare results and use table to ask questions.

Words	going	to	today	i	am	it	is	rain
Document 1	0	0.07	0.07	0	0	0.17	0.17	0.17
Document 2	0	0	0.07	0.07	0.07	0	0	0

Document 3	0	0.05	0	0.05	0.05	0	0	0
---------------	---	------	---	------	------	---	---	---

We can easily see using this table that words like “it”, “is”, “rain” are important for document 1 but not for document 2 and document 3 which means Document 1 and 2&3 are different w.r.t talking about rain.

We can also say that Document 1 and 2 talk about something ‘today’, and document 2 and 3 discuss something about the writer because of the word ‘I’.

This table helps us to find similarities and non-similarities btw documents, words and more much better than any other approach.

6. Code TF-IDF

Challenge is to use these sentences and find words which provide meaning to these sentences using TF-IDF.

Let's begin

6.1 Load Documents

The `add_documents_from_directory` function loads all .txt files from a specified directory. Each file's content is stored, and a unique ID is generated using a hash of the file path.

```
def add_documents_from_directory(directory_path):
    """Add all text files from a directory to the documents list."""
    documents = [] # List to hold the contents of documents
    for file_name in os.listdir(directory_path):
        if file_name.endswith(".txt"): # Process only .txt files
            file_path = os.path.join(directory_path, file_name)
            with open(file_path, 'r', encoding='utf-8') as file:
                title = os.path.splitext(os.path.basename(file_path))[0] # Extract title from file name
                content = file.read()
                doc_id = hash(file_path) # Use a hash of the file path as a unique ID
                document_titles[doc_id] = title # Store the title with doc_id
                documents.append(content)
    return documents

# documents list with the documents loaded from a directory
directory_path = "./Documents_02/test"
documents = add_documents_from_directory(directory_path)
```

6.2. Clean data and Tokenize

Preprocessing involves removing punctuation, converting text to lowercase and splitting it into words. This ensures uniformity and reduces noise in data.

```
def preprocess(text):
    # Remove punctuation using regex
    text = re.sub(r'[^\w\s]', '', text) # Retain only words and spaces

    # Lowercase, split into words, and remove stopwords
```

```

words = text.lower().split()
return words

# Tokenized documents
tokenized_docs = [preprocess(doc) for doc in documents]

```

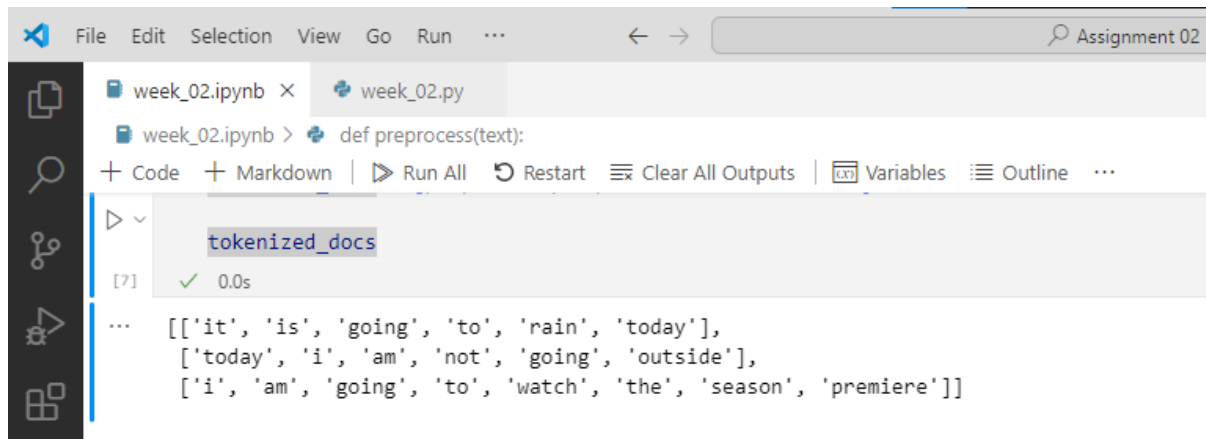


Figure 1: Clean data and Tokenize

6.3. Find TF

For counting the number of words in a document, I am using a python library named Counter.

```

from collections import Counter

# Step 3: Calculate term frequency (TF)
def compute_tf(doc_tokens):
    tf = Counter(doc_tokens)
    for term in tf:
        tf[term] /= len(doc_tokens)
    return tf

```

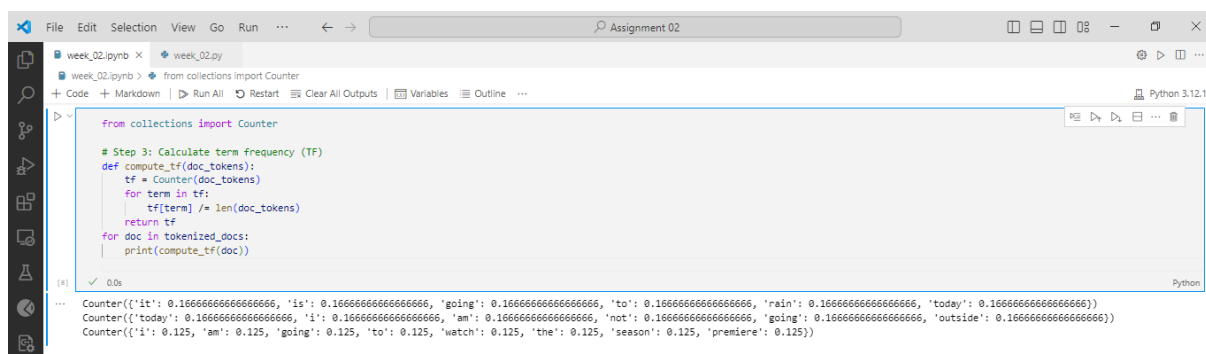


Figure 2: Finding TF for words

6.4. Find IDF

```

# Step 4: Calculate inverse document frequency (IDF)
def compute_idf(tokenized_docs):
    num_docs = len(tokenized_docs)
    idf = {}

```



```

all_terms = set(term for doc in tokenized_docs for term in doc)
for term in all_terms:
    doc_count = sum(1 for doc in tokenized_docs if term in doc)
    idf[term] = math.log(num_docs / doc_count)
return idf

idfs = compute_idf(tokenized_docs)
idfs

```

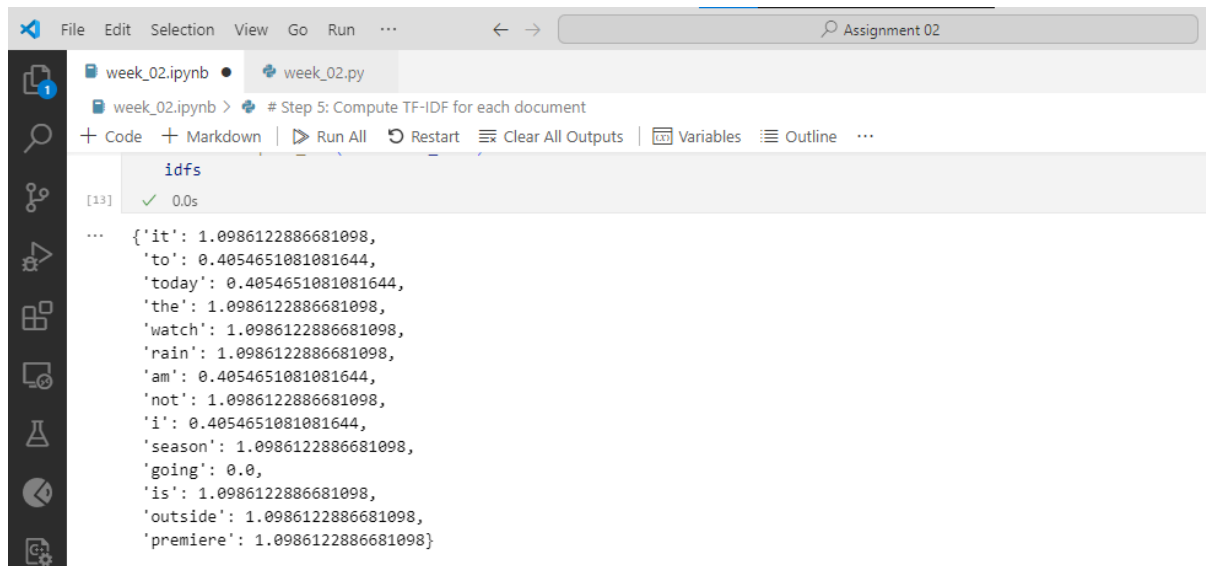


Figure 3: Finding IDF for words

6.5. Find TF-IDF

```

# Step 5: Compute TF-IDF for each document
def compute_tfidf(tf, idf):
    tfidf = {}
    for term, tf_value in tf.items():
        tfidf[term] = tf_value * idf.get(term, 0)
    return tfidf

# Compute IDF
tfidf_docs = [compute_tfidf(compute_tf(doc), idfs) for doc in tokenized_docs]
tfidf_docs

```

```

week_02.ipynb > # Step 5: Compute TF-IDF for each document
+ Code + Markdown | ▶ Run All | ⌂ Restart | 🗑 Clear All Outputs | 📄 Variables | 📖 Outline | ...
tfidf_docs
[14] ✓ 0.0s
... [{"it": 0.1831020481113516,
      'is': 0.1831020481113516,
      'going': 0.0,
      'to': 0.06757751801802739,
      'rain': 0.1831020481113516,
      'today': 0.06757751801802739},
      {'today': 0.06757751801802739,
       'i': 0.06757751801802739,
       'am': 0.06757751801802739,
       'not': 0.1831020481113516,
       'going': 0.0,
       'outside': 0.1831020481113516},
      {'i': 0.05068313851352055,
       'am': 0.05068313851352055,
       'going': 0.0,
       'to': 0.05068313851352055,
       'watch': 0.13732653608351372,
       'the': 0.13732653608351372,
       'season': 0.13732653608351372,
       'premiere': 0.13732653608351372}]

```

Figure 4: Finding TF-IDF

7. Query Handling and Ranking

- Normalize the query
- Tokenize the query
- Compute TF and TF-IDF values for the query
- Calculate Cosine Similarity
 - Compare the TF-IDF vector of the query with each document's TF-IDF vector.
 - Compute the cosine similarity between the query and each document.

Let's consider a query "It will Rain". And let's see how can we compute a Query Handling and Ranking.

7.1. Preprocessing and Query TF-IDF Vector

Word	Count	Query TF	IDF value	Query TF-IDF
it	1	0.33	1.09	0.36
is	1	0.33	1.09	0.36
rain	1	0.33	1.09	0.36

7.2. Calculate Cosine Similarity

As we know the query tf-idf vector is = [0.36, 0.36, 0.36]

Also, we already have the document tf-idf vectors:

Document 1 = [0.18, 0.18, 0.18]

Document 2 = [0, 0, 0]

Document 3 = [0, 0, 0]

Document 1:

$$\text{Dot Product} = (0.36 \times 0.18) + (0.36 \times 0.18) + (0.36 \times 0.18) = 0.194$$

$$\|\mathbf{A}\| = \sqrt{(0.36)^2 + (0.36)^2 + (0.36)^2} = 0.623, \quad \|\mathbf{B}\| = \sqrt{(0.18)^2 + (0.18)^2 + (0.18)^2} = 0.312$$

$$\text{Cosine Similarity} = \frac{0.194}{0.623 \times 0.312} = 0.9575$$

Document 2:

$$\text{Dot Product} = (0.36 \times 0) + (0.36 \times 0) + (0.36 \times 0) = 0$$

$$\|\mathbf{B}\| = \sqrt{(0)^2 + (0)^2 + (0)^2} = 0$$

$$\text{Cosine Similarity} = \frac{0}{0.623 \times 0} = 0$$

Document 3:

$$\text{Dot Product} = (0.36 \times 0) + (0.36 \times 0) + (0.36 \times 0) = 0$$

$$\|\mathbf{B}\| = \sqrt{(0)^2 + (0)^2 + (0)^2} = 0$$

$$\text{Cosine Similarity} = \frac{0}{0.623 \times 0} = 0$$

Results:

Document 1	0.9575
Document 2	0
Document 3	0

```
# Function to preprocess the query
def preprocess_query(query):
    return preprocess(query) # Assuming 'preprocess' is defined elsewhere

# Function to compute TF-IDF for the query
def compute_query_tfidf(query_tokens, idfs):
    query_tf = compute_tf(query_tokens) # Assuming 'compute_tf' is defined
    query_tfidf = compute_tfidf(query_tf, idfs) # Assuming 'compute_tfidf' is defined
    return query_tfidf

# Function to calculate cosine similarity between query and a document
def compute_cosine_similarity(query_tfidf, doc_tfidf):
    dot_product = sum(query_tfidf.get(term, 0) * doc_tfidf.get(term, 0) for term in query_tfidf)
    query_norm = math.sqrt(sum(value ** 2 for value in query_tfidf.values()))
    doc_norm = math.sqrt(sum(value ** 2 for value in doc_tfidf.values()))
```

```
    return dot_product / (query_norm * doc_norm) if query_norm and doc_norm else 0

# Function to rank documents based on cosine similarity
def rank_documents(query, tfidf_docs, idfs):
    query_tokens = preprocess_query(query)
    query_tfidf = compute_query_tfidf(query_tokens, idfs)

    # Compute cosine similarity for all documents
    rankings = []
    for i, doc_tfidf in enumerate(tfidf_docs):
        similarity = compute_cosine_similarity(query_tfidf, doc_tfidf)
        rankings.append((i, similarity))

    # Sort by similarity score in descending order
    rankings.sort(key=lambda x: x[1], reverse=True)
    return rankings
```

8. User Interface

A Command Line Interface (CLI) enables users to input queries and view ranked results or exit the program.

```
def main():
    print("Welcome to the TF-IDF Search Engine!")
    print("Enter your query to find the most relevant documents.")

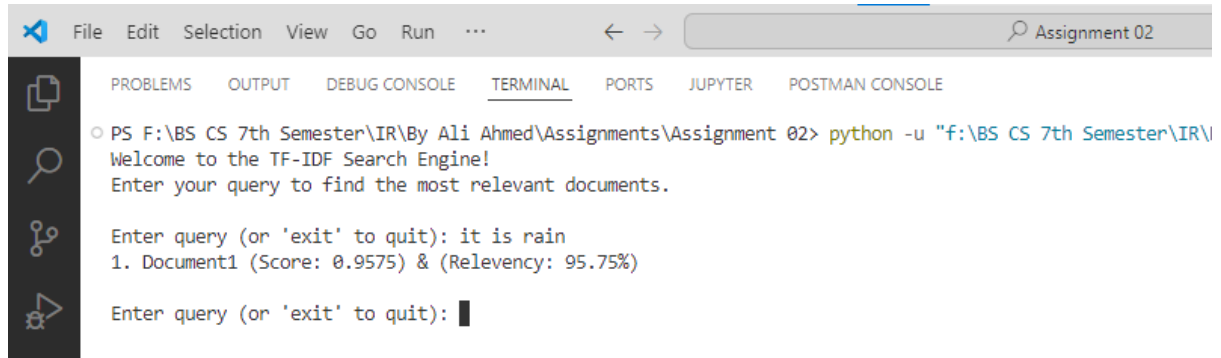
    while True:
        retried = False
        query = input("\nEnter query (or 'exit' to quit): ").strip()
        if query.lower() == "exit":
            print("Exiting the search engine. Goodbye!")
            break

        rankings = rank_documents(query)
        # print("\nDocument Rankings:")
        for rank, (doc_index, score) in enumerate(rankings, start=1):
            if score > 0:
                retried = True
                doc_id = list(document_titles.keys())[doc_index]
                document_title = document_titles[doc_id]

                print(f"[rank]. {document_title} (Score: {score:.4f}) & (Relevancy: {score * 100:.2f}%)")

            if not retried:
                print("\nSorry, no relevant documents found for your query. Please try again with different terms.")

# Run the CLI
if __name__ == "__main__":
    main()
```



```

PS F:\BS CS 7th Semester\IR\By Ali Ahmed\Assignments\Assignment 02> python -u "f:\BS CS 7th Semester\IR\
Welcome to the TF-IDF Search Engine!
Enter your query to find the most relevant documents.

Enter query (or 'exit' to quit): it is rain
1. Document1 (Score: 0.9575) & (Relevency: 95.75%)

Enter query (or 'exit' to quit):

```

Figure 5: User Interface

9. Data Flow Diagram

Here is the complete data flow diagram of the whole system.

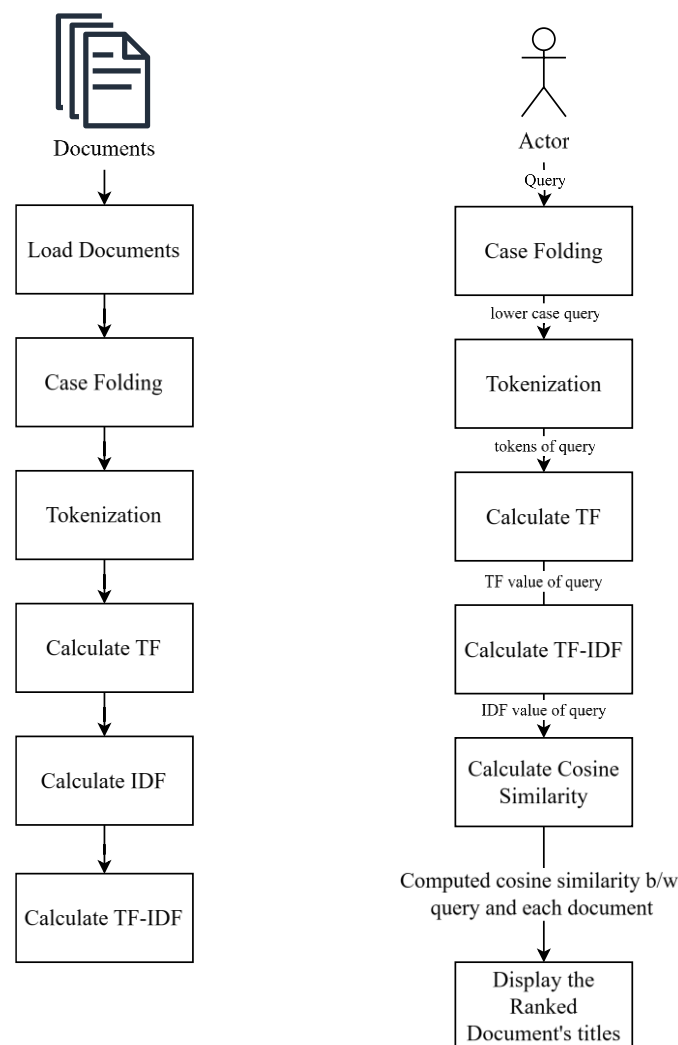


Figure 6: Dataflow Diagram

7. Conclusion

This document ranking system demonstrates the efficiency of TF-IDF for information retrieval. By normalizing text, weighting term importance, and ranking based on vector similarities, the system

effectively matches user queries with relevant documents. Future enhancements may include scalability to larger datasets.