Information Retrieval System Developmen

1. Dataset Selection and Preparation:

- Your first task is to select a dataset of text documents. This dataset should represent a collection of documents from a specific domain (e.g., news articles, research papers, or product descriptions).
- Ensure that the dataset is appropriately labeled or categorized.

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
df=pd.read_csv('Annotations.csv')
df
```

Out[2]:

	ID_Post	ID_Annotator	Category	Value
0	3326	1	ArgumentsUsed	0
1	3326	1	Discriminating	0
2	3326	1	Inappropriate	0
3	3326	1	OffTopic	0
4	3326	1	PersonalStories	0
...	...	...	...	...
58563	1003437	1	PossiblyFeedback	0
58564	1004625	1	PossiblyFeedback	1
58565	1006255	1	PossiblyFeedback	0
58566	1010868	2	PossiblyFeedback	0
58567	1010997	1	PossiblyFeedback	1

58568 rows × 4 columns

2. Data Preprocessing:

- The next step involves cleaning and preprocessing the text data. This includes tasks such as:
- Tokenization.
- Lowercasing.
- Removing punctuation.
- Handling missing values, if any.
- You should also create an inverted index, a data structure that maps terms (words) to their corresponding documents.

In [3]:
```
import nltk
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer



# Tokenization
tokens = word_tokenize('Category')

# Lowercasing
tokens = [word.lower() for word in tokens]

# Removing Punctuation
table = str.maketrans('', '', string.punctuation)
tokens = [word.translate(table) for word in tokens]

# Handling Missing Values (not applicable in this example)

# Stop Word Removal
stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word not in stop_words]

# Stemming
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]

# Lemmatization
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(word) for word in filtered_tokens]

# Normalization (not shown here, as it depends on specific use cases)

# Encoding and Vectorization (not shown here, as it depends on specific use cas

# Printing the results
print("Original Tokens:", tokens)
print("Filtered Tokens (Stopword Removal):", filtered_tokens)
print("Stemmed Tokens:", stemmed_tokens)
print("Lemmatized Tokens:", lemmatized_tokens)
```

```
Original Tokens: ['category']
Filtered Tokens (Stopword Removal): ['category']
Stemmed Tokens: ['categori']
Lemmatized Tokens: ['category']
```

In [5]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

# Create the TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer()

# Fit and transform the documents to TF-IDF vectors
tfidf_matrix = tfidf_vectorizer.fit_transform(df)

# Get the feature names (words)
feature_names = tfidf_vectorizer.get_feature_names_out()
```

3. User Query Interface:

- Create a user-friendly query interface that allows users to input search queries.
- Ensure that the interface can handle natural language queries.

In [4]:
```
!pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in c:\users\hania fatima\anaconda
3\lib\site-packages (1.2.1)
Requirement already satisfied: scipy>=1.3.2 in c:\users\hania fatima\anaconda
3\lib\site-packages (from scikit-learn) (1.11.1)
Requirement already satisfied: joblib>=1.1.1 in c:\users\hania fatima\anacond
a3\lib\site-packages (from scikit-learn) (1.1.1)
Requirement already satisfied: numpy>=1.17.3 in c:\users\hania fatima\anacond
a3\lib\site-packages (from scikit-learn) (1.23.5)
Requirement already satisfied: threadpoolctl>=2.0.0 in c:\users\hania fatima
\anaconda3\lib\site-packages (from scikit-learn) (2.2.0)

WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
WARNING: Ignoring invalid distribution -cipy (c:\users\hania fatima\anaconda3
\lib\site-packages)
```

In [7]:
```
# User Query Processing
user_query = "This is the second document."  # Replace with the actual user que

# Preprocess the user query using the same preprocessing steps as used for docu
# If you don't have a separate function, you can apply the preprocessing direct
# For example, assuming you tokenize and remove stopwords:
user_query_tokens = user_query.split()  # Tokenization
user_query_tokens = [word.lower() for word in user_query_tokens]  # Lowercasing
user_query_tokens = [word for word in user_query_tokens if word not in stop_wo

# Join the preprocessed tokens back into a single string
preprocessed_query = " ".join(user_query_tokens)

# Compute the TF-IDF vector for the query
query_vector = tfidf_vectorizer.transform([preprocessed_query])
```

4. Retrieval Algorithm:

- Implement the Vector Space Model (VSM) or Term Frequency-Inverse Document Frequency (TF-IDF) as your retrieval algorithm. These are beginner-friendly approaches.
- Your system should rank documents based on their relevance to user queries.

In [9]:
```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

def preprocess_query(query):
    # Tokenization and lowercasing
    tokens = query.lower().split()

    # Remove stopwords
    tokens = [word for word in tokens if word not in ENGLISH_STOP_WORDS]

    # Join tokens back into a single string
    preprocessed_query = " ".join(tokens)

    return preprocessed_query
```

5. Query Processing:

- Preprocess user queries similarly to how you processed documents.
- The system should compare the user query to the indexed documents and return a ranked list of relevant documents

In [11]:
```
# Preprocess user query
user_query = "This is the second document."  # Replace with the user's query
preprocessed_user_query = preprocess_query(user_query)

# Compute the TF-IDF vector for the query
query_vector = tfidf_vectorizer.transform([preprocessed_user_query])
```

In [12]:
```
from sklearn.metrics.pairwise import cosine_similarity

# Compute cosine similarity between the query vector and document vectors
cosine_similarities = cosine_similarity(query_vector, tfidf_matrix)

# Get a list of document indexes sorted by relevance
document_scores = list(enumerate(cosine_similarities[0]))
sorted_documents = sorted(document_scores, key=lambda x: x[1], reverse=True)

# Print the ranked documents
for idx, score in sorted_documents:
    print(f"Document {idx + 1}: {documents[idx]}, Score: {score:.2f}")
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[12], line 12
     10 # Print the ranked documents
     11 for idx, score in sorted_documents:
---> 12     print(f"Document {idx + 1}: {documents[idx]}, Score: {score:.2
f}")

NameError: name 'documents' is not defined
```

6. Evaluation:

- Define a set of test queries and relevant documents to evaluate the system's performance.
- Use common IR evaluation metrics like Precision, Recall, and F1-score to assess how well the system retrieves relevant documents.

In [13]:
```
test_queries = [
    "How does climate change impact ecosystems?",
    "Latest advances in artificial intelligence",
    "Healthy diet for weight loss",
    "How does blockchain technology work?",
    "Treatment options for diabetes",
]
```

In [14]:
```
# Define ground truth for each test query as a list of document indexes
ground_truth = {
    "How does climate change impact ecosystems?": [1, 3, 6],
    "Latest advances in artificial intelligence": [2, 4, 5],
    "Healthy diet for weight loss": [8, 10, 12],
    "How does blockchain technology work?": [14, 16, 18],
    "Treatment options for diabetes": [20, 22, 24],
}
```

In [15]:
```
from sklearn.metrics import precision_score, recall_score, f1_score

# Initialize lists to store evaluation results
precision_scores = []
recall_scores = []
f1_scores = []

# Iterate through test queries
for query in test_queries:
    # Retrieve documents for the query using your IR system (replace with your
    retrieved_documents = retrieve_documents(query)  # Implement this function

    # Get the ground truth for this query
    relevant_documents = ground_truth.get(query, [])

    # Calculate Precision, Recall, and F1-score
    precision = precision_score(relevant_documents, retrieved_documents)
    recall = recall_score(relevant_documents, retrieved_documents)
    f1 = f1_score(relevant_documents, retrieved_documents)

    # Append the scores to the lists
    precision_scores.append(precision)
    recall_scores.append(recall)
    f1_scores.append(f1)

# Calculate average scores
average_precision = sum(precision_scores) / len(precision_scores)
average_recall = sum(recall_scores) / len(recall_scores)
average_f1 = sum(f1_scores) / len(f1_scores)

# Print the average scores
print("Average Precision:", average_precision)
print("Average Recall:", average_recall)
print("Average F1-score:", average_f1)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[15], line 11
      8 # Iterate through test queries
      9 for query in test_queries:
     10     # Retrieve documents for the query using your IR system (replace
with your code)
---> 11     retrieved_documents = retrieve_documents(query)  # Implement this
function
     13     # Get the ground truth for this query
     14     relevant_documents = ground_truth.get(query, [])

NameError: name 'retrieve_documents' is not defined
```

In [ ]:
```

```