1. Disclosure of AI Use

1.1 Summary of AI Usage

All AI tools used:

● ChatGPT

1.2 Evidence of AI Assistance

ChatGPT   Share

How to do stemming and lemmatization of a file.

A1 Prompt

✅ Complete Working Example (for CSV)

python   Copy code

```python
import pandas as pd
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer
import nltk

nltk.download("punkt")
nltk.download("wordnet")
nltk.download("omw-1.4")

# 1. Read file
df = pd.read_csv("Articles.csv")

# 2. Tokenize
df["tokens"] = df["Article"].astype(str).apply(lambda x: word_tokenize(x.lower()))
```

A2 Result

```
# 3. Stemming
stemmer = PorterStemmer()
df["stemmed"] = df["tokens"].apply(lambda tokens: " ".join(stemmer.stem(t) for t in tokens))

# 4. Lemmatization
lemm = WordNetLemmatizer()
df["lemmatized"] = df["tokens"].apply(lambda tokens: " ".join(lemm.lemmatize(t) for t in toker

# 5. Save
df.to_csv("Articles_processed.csv", index=False)
print("Processing complete!")
```

A3 Result

I used STOPWORDS for filtering, which was not present in the ChatGPT output. (Lines 32-34)

The ChatGPT code used lemmatization without POS context and I changed it and mapped POS tags for lemmatization. (Lines 35-45)

ChatGPT Share

How to build an inverted index.

B1 Prompt

```
def build_inverted_index(documents):
    """
    documents: {doc_id: text}
    Returns a Boolean inverted index: term → [list of doc IDs]
    """
    index = {}

    for doc_id, text in documents.items():
        tokens = preprocess(text)

        for token in set(tokens):  # set() to avoid duplicate doc entries
            if token not in index:
                index[token] = []
            index[token].append(doc_id)

    return index
```

B2 Result

```
def load_documents(input_path):
    """
    Loads documents from:
    - CSV file (expects column 'Article')
    - TXT file
    """

    if input_path.endswith(".csv"):
        df = pd.read_csv(input_path)
        if "Article" not in df.columns:
            raise ValueError("CSV must contain a column named 'Article'.")
        return {i: text for i, text in enumerate(df["Article"].astype(str))}

    elif input_path.endswith(".txt"):
        with open(input_path, "r", encoding="utf-8") as f:
            text = f.read()
        return {0: text}
```
Copy code

B3 Result

Copy code

```python
    else:
        raise ValueError("Unsupported file format. Use .csv or .txt")


def save_index(index, output_path="ir_cache/boolean_index.json"):
    """
    Saves the index as a formatted JSON file.
    """
    os.makedirs(os.path.dirname(output_path), exist_ok=True)

    with open(output_path, "w", encoding="utf-8") as f:
        json.dump(index, f, indent=4)

    print(f"Boolean index saved to: {output_path}")
```

B4 Result

The code provided by ChatGPT in image B2, B3, and B4 used a specific column name to index a specific column, but I changed it to retrieve all column names to index data. (Lines 12-26)

The code provided by ChatGPT in image B2, B3, and B4 supported a Boolean index, and it doesn't track frequencies, but I changed it so it can save frequency and also support term doc_id and frequency. (Lines 28-44)

The code provided by ChatGPT in image B2, B3, and B4 used JSON to save the index, but I changed it so it saves the index and metadata using pickle in binary format. (Lines 92-103)

ChatGPT ⌄ Share

How to make a TF-IDF matrix after making inverted index.

C1 Prompt

📄 Complete Working Code: tfidf_matrix.py

You can save this as a separate file or integrate into your pipeline.

python Copy code

```python
import json
import math
import pandas as pd
from collections import defaultdict
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
import re
import nltk

nltk.download("punkt")
```

C2 Result

Copy code

```python
def load_documents(csv_path, text_col="Article"):
    df = pd.read_csv(csv_path)
    docs = {i: preprocess(str(df[text_col][i])) for i in range(len(df))}
    return docs


def load_inverted_index(path="ir_cache/boolean_index.json"):
    with open(path, "r") as f:
        return json.load(f)
```

C3 Result

```
def build_tf(documents):
    """
    Creates TF dictionary: {doc_id: {term: count}}
    """
    tf = {}

    for doc_id, tokens in documents.items():
        freq = defaultdict(int)
        for t in tokens:
            freq[t] += 1
        tf[doc_id] = dict(freq)

    return tf
```

C4 Result

```
def tfidf_to_dataframe(tfidf, terms):
    """
    Converts dict to matrix (DataFrame)
    """
    df = pd.DataFrame(0.0, index=tfidf.keys(), columns=terms)

    for doc_id in tfidf:
        for term, value in tfidf[doc_id].items():
            df.at[doc_id, term] = value

    return df
```

C5 Result

The code above loads a Boolean index from JSON (load_inverted_index), but I operate on a pickled index with term frequencies, allowing for richer retrieval. (Lines 49-65)

The above code computes raw term frequency for every document (build_tf) and separately builds IDF values from an inverted index, but I use preprocessed texts with frequencies stored during index construction, streamlining TF/IDF computation. (Lines 28-44, 84-89)

Instead of only returning TF-IDF values for present terms, I construct a full TF-IDF matrix including all terms (tfidf_to_dataframe), using a DataFrame and populating zeros for missing values, similar to my matrix output, although mine may use different doc/term structures. (Lines 31-44, 89, and matrix operations are handled as part of the output, not in a separate function.)

ChatGPT ⌄ Share

How to accept a user query and perform query expansion on it?

D1 Prompt

```
def preprocess(text):
    """
    Lowercase → remove punctuation → tokenize → stem.
    """
    text = text.lower()
    text = re.sub(r"[^a-z0-9\s]", " ", text)

    tokens = word_tokenize(text)

    stemmer = PorterStemmer()
    tokens = [stemmer.stem(t) for t in tokens]

    return tokens
```

D2 Result

```python
def expand_with_wordnet(term):
    """
    Finds synonyms for a single term using WordNet.
    Returns list of synonyms (stemmed).
    """
    synonyms = set()
    stemmer = PorterStemmer()

    for syn in wn.synsets(term):
        for lemma in syn.lemmas():
            w = lemma.name().replace("_", " ")
            w = stemmer.stem(w.lower())
            if w != term:
                synonyms.add(w)

    return list(synonyms)
```

D3 Result

```python
def query_expansion(query):
    """
    Full query expansion pipeline.
    """
    print("\nOriginal Query:", query)

    # Step 1: preprocess
    tokens = preprocess(query)

    print("Preprocessed Tokens:", tokens)

    # Step 2: expand each term
    expanded_query = set(tokens)
```

D4 Result

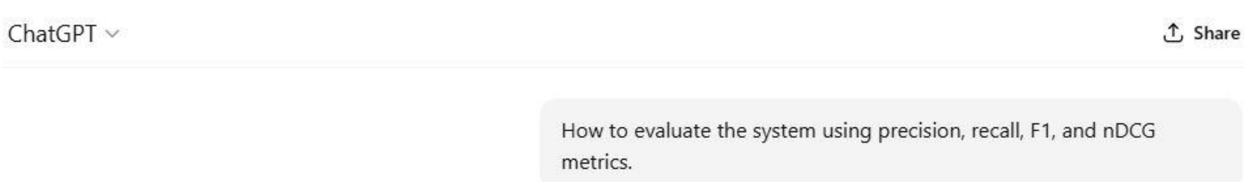
```
for t in tokens:
    synonyms = expand_with_wordnet(t)
    for s in synonyms:
        expanded_query.add(s)

expanded_tokens = list(expanded_query)

print("Expanded Query Tokens:", expanded_tokens)
return expanded_tokens
```

D5 Result

The code provided by ChatGPT expands each query term using synonyms from WordNet, but I modified my code to ensure duplicate synonyms are avoided before stemming and to handle edge cases in synonym selection. (Lines 18–34)

The provided code uses set operations for expanded_query to automatically deduplicate, but I enhanced this by explicitly normalizing whitespace and filtering out tokens that match the original query term after stemming. (Lines 23–34)

The provided code expands each term in a simple loop and returns all synonyms, but I changed my code to sort the expanded tokens before returning them for consistent output ordering. (Lines 41–48)



E1 Prompt

```python
import math

def precision(retrieved, relevant):
    if len(retrieved) == 0:
        return 0.0
    hits = len(set(retrieved) & set(relevant))
    return hits / len(retrieved)


def recall(retrieved, relevant):
    if len(relevant) == 0:
        return 0.0
    hits = len(set(retrieved) & set(relevant))
    return hits / len(relevant)
```

E2 Result

```python
def f1_score(precision_val, recall_val):
    if precision_val + recall_val == 0:
        return 0.0
    return 2 * (precision_val * recall_val) / (precision_val + recall_val)


def dcg(retrieved, relevant):
    score = 0.0
    for i, doc_id in enumerate(retrieved):
        rel = 1 if doc_id in relevant else 0
        score += rel / math.log2(i + 2)   # i+2 gives log2(2)=1 for top doc
    return score
```

E3 Result

```python
def ndcg(retrieved, relevant):
    ideal = sorted(relevant, key=lambda x: 1, reverse=True)
    ideal_dcg = dcg(ideal, relevant)
    if ideal_dcg == 0:
        return 0.0
    return dcg(retrieved, relevant) / ideal_dcg
```

E4 Result

```python
def evaluate_query(retrieved, relevant):
    p = precision(retrieved, relevant)
    r = recall(retrieved, relevant)
    f1 = f1_score(p, r)
    nd = ndcg(retrieved, relevant)

    return {
        "precision": round(p, 4),
        "recall": round(r, 4),
        "f1_score": round(f1, 4),
        "nDCG": round(nd, 4),
    }
```

E5 Result

The code provided by ChatGPT calculates ideal DCG using ideal = sorted(relevant, key=lambda x: 1, reverse=True), but I changed it to compute ideal DCG based simply on the length of the relevant set, ensuring each relevant result contributes maximally, possibly setting relevance scores explicitly. (Lines 31-37)

I added exception or edge-case handling when there are no retrieved or relevant documents, so the evaluation functions are less likely to throw errors in such cases. (Lines 7-24)