

Data Collection:

Website source: <https://huggingface.co/datasets/BelR/scidocs-qrels>

Data link: <https://public.ukp.informatik.tu-darmstadt.de/thakur/BEIR/datasets/nfcorpus.zip>

```
import os

# function to ensure directory exists
def ensure_dir(file_path):
    directory = os.path.dirname(file_path)
    if not os.path.exists(directory):
        os.makedirs(directory)

# function to convert JSON to CSV
def json_to_csv(json_file_path, csv_file_path):
    ensure_dir(csv_file_path)
    if os.path.exists(json_file_path) and os.path.getsize(json_file_path) > 0:
        try:
            df = pd.read_json(json_file_path, lines=True)
            df.to_csv(csv_file_path, index=False)
            print(f"successfully converted {json_file_path} to {csv_file_path}")
        except ValueError as e:
            print(f"Error reading {json_file_path}: {str(e)}")
    else:
        print(f"File not found or is empty: {json_file_path}")

# function to convert TXT to CSV
def txt_to_csv(txt_file_path, csv_file_path):
    ensure_dir(csv_file_path)
    if os.path.exists(txt_file_path) and os.path.getsize(txt_file_path) > 0:
        try:
            df = pd.read_csv(txt_file_path, sep='\t', encoding='utf-8')
            df.to_csv(csv_file_path, index=False)
            print(f"successfully converted {txt_file_path} to {csv_file_path}")
        except Exception as e:
            print(f"Failed to convert {txt_file_path} to CSV: {str(e)}")
    else:
        print(f"File not found or is empty: {txt_file_path}")
```

Preprocessing:

Apply (Tokenization, Lowercasing, Stopword Removal, Stemming or Lemmatization)

To make data cleaned and improve evaluate of search engine

```
▼ preprocessing

# Initialize Porter Stemmer
stemmer = PorterStemmer()

stop_words = stopwords.words('english')

def preprocess(text):
    text = re.sub(r"http\S+", " ", text) # remove urls
    text = re.sub(r"RT ", " ", text) # remove rt
    text = re.sub(r"@[\w]*", " ", text) # remove handles
    text = re.sub(r"[\.\,\#\_\|\:\;\?\|\=\]", " ", text) # remove special characters
    text = re.sub(r'\t', ' ', text) # remove tabs
    text = re.sub(r'\n', ' ', text) # remove line jump
    text = re.sub(r'\s+', ' ', text).strip().lower()
    tokens = word_tokenize(text)
    return [stemmer.stem(token) for token in tokens if token not in stop_words]
```

Indexing:

Apply processing in column of text in data and use query to compare between it and doc of text

After apply processing

Indexing

```
# Load data
docs_df = pd.read_csv("/mnt/data/corpus.csv")
# Apply preprocessing to document content
docs_df['processed_content'] = docs_df['text'].apply(lambda x: ' '.join(preprocess(x)))
docs_df["docno"] = docs_df.index.astype(str)

# Build an inverted index
|
indexer = pt.DFIndexer("./index", overwrite=True)
index_ref = indexer.index(docs_df['processed_content'], docs_df['docno'])
index = pt.IndexFactory.of(index_ref)
```

```
meta = index.getMetaIndex()
inv = index.getInvertedIndex()
lex = index.getLexicon()

# List of terms to retrieve document frequency
terms = preprocess("Breast Using Diet") # Add your terms here

for term in terms:
    le = lex.getLexiconEntry(term)
    if le is None:
        print(f"Term '{term}' not found in the lexicon.")
        continue

    # Access the inverted index posting list for the term
    for posting in inv.getPostings(le):
        docno = meta.getItem("docno", posting.getId())
        print(f"Document: {docno}, Frequency: {posting.getFrequency()} for term: {term}")
```

Query Processing:

Get user information need as query and apply process and show document relevant to it and retrieved it

✓ Query Processing

```
def search(query, k=10):
    processed_query = preprocess(query)
    processed_query_str = " ".join(processed_query)
    tf = pt.BatchRetrieve(index, wmodel="TF_IDF")
    results = tf.transform(processed_query_str)
    # evaluation=pt.Evaluate(results,x)
    return results
ss=search("cancer")
```

<ipython-input-45-7fd625f9af64>:5: FutureWarning: .transform() should be passed a results = tf.transform(processed_query_str)

```
[ ] def expand_query(query, k=10):
    processed_query = preprocess(query)
    # Convert list of terms back to a string for searching
    processed_query_str = " ".join(processed_query)
    tf = pt.BatchRetrieve(index, wmodel="TF")
    rml_expander = pt.rml_expander(index, fb_terms=10, fb_docs=100)
    rml_exp = tf >> rml_expander
    expanded_query = rml_exp.search(query).iloc[0]["query"]
    expanded_terms = [term.split(" ")[0] for term in expanded_query.split()[1:]] # Extract only the terms
    return expanded_terms

expanded_terms = expand_query("Breast Using Biot")
print(expanded_terms)
```

Query expansion:

After apply relevant docs ranked them given similarity on meaning to show the informative docs first "using elmo" and bert

```
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
import pandas as pd

# function to calculate cosine similarity
def cosine_similarity(v1, v2):
    # Ensure the embeddings are summed across all tokens and reduced to a 1D array
    sum_v1 = np.sum(v1, axis=0)
    sum_v2 = np.sum(v2, axis=0)
    if sum_v1.ndim > 1:
        sum_v1 = np.sum(sum_v1, axis=0)
    if sum_v2.ndim > 1:
        sum_v2 = np.sum(sum_v2, axis=0)

    dot_product = np.dot(sum_v1, sum_v2)
    norm_v1 = np.linalg.norm(sum_v1)
    norm_v2 = np.linalg.norm(sum_v2)
    return dot_product / (norm_v1 * norm_v2)

# Load the elmo model
elmo = hub.load("https://tfhub.dev/google/elmo/2")

# Define the query sentence
query_sentence = "biomaterials"

# Assuming docs_df is your DataFrame with a "processed_content" column
# Limit the loop to the first five rows
similarities = []
for doc_content in docs_df["processed_content"].iloc[1:5]: # Process only the first five documents
    # Generate elmo embeddings for the query sentence and the document content
```

```

# Assuming docs_df is your DataFrame with a 'processed_content' column
# Limit the loop to the first five rows
similarities = []
for doc_content in docs_df['processed_content'].iloc[:5]: # Process only the first five rows
    # Generate ELMo embeddings for the query sentence and the document content
    query_embedding = elmo.signatures["default"](tf.constant([query_sentence]))["elmo_embeddings"]
    doc_embedding = elmo.signatures["default"](tf.constant([doc_content]))["elmo_embeddings"]

    # Calculate cosine similarity between the query and the document
    similarity_score = cosine_similarity(query_embedding, doc_embedding)
    similarities.append(similarity_score)

# Create a DataFrame to store similarity scores with corresponding documents
similarity_df = pd.DataFrame({
    'Document': docs_df['processed_content'].iloc[:5], # Include only the first five rows
    'Similarity Score': similarities
})

# Rank the documents based on similarity scores
similarity_df = similarity_df.sort_values(by='Similarity Score', ascending=False)

# Print the ranked DataFrame
print(similarity_df)

```

User Interface:

Apply user interface to website to satisfy the information need of user and retrieve docs ranked satisfy his requirements