```
# -*- coding: utf-8 -*-
"""Another copy of Final_ecom_opt.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1NLP95jBsC1KcKX7feIeuXB_9_J2zLq4r

# Jupyter code

## Importing Libraries
"""

# Data Manipulation
import pandas as pd
import numpy as np

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns

# NLP for text pre-processing
import nltk
import scipy
import re
from scipy import spatial
from nltk.tokenize.toktok import ToktokTokenizer
from nltk.corpus import stopwords
from nltk.tokenize import sent_tokenize, word_tokenize
```

```
from nltk.stem import PorterStemmer
tokenizer = ToktokTokenizer()

# other libraries
import gensim
from gensim.models import Word2Vec
import itertools
from sklearn.decomposition import PCA
from sklearn.feature_extraction. text import TfidfVectorizer
from sklearn.decomposition import PCA

# Import linear_kernel
from sklearn.metrics.pairwise import linear_kernel

# remove warnings
import warnings
warnings.filterwarnings (action = 'ignore')

data = pd.read_csv("/content/flipkart_com-ecommerce_sample.csv") # Changed the file path
data.head()

"""## Data Exploration - jupyter file

## Data Preprocessing
"""

data.isnull().sum()

#handling missing values
```

```
missing = pd.DataFrame(data.isnull().sum()).rename (columns = {0: 'missing' })
missing['percent'] = (missing['missing'] /len(data))*100
missing.sort_values ('percent', ascending = False)

# fing ing the redundant or duplicate rows and removingthem

duplicate_rows = data[data.duplicated()]

# Print the number of duplicate rows found
print(f"Number of duplicate rows: {duplicate_rows.shape[0]}")

# Display the duplicate rows (if any)
if duplicate_rows.shape[0] > 0:
    print("Duplicate rows found:")
    print(duplicate_rows)
else:
    print("No duplicate rows found.")

# Remove duplicate rows and keep the first occurrence
data_cleaned = data.drop_duplicates()

# Verify the result by checking the new shape of the dataset
print(f"Original dataset shape: {data.shape}")
print(f"Dataset shape after removing duplicates: {data_cleaned.shape}")

"""### Text preprocessing
```

There is a lot of unwanted information present in the text data. Let's clean it up. Text preprocessing tasks include

* Converting the text data to lowercase

* Removing/replacing the punctuations

* Removing/replacing the numbers

* Removing extra whitespaces

* Removing stop words

* Stemming and lemmatization

"""

```
#to lowercase
data['description'] = data['description'].str.lower()

#removing punctivations
data['description'] = data['description'].str.replace(r'[^\w\d\s]',' ')

#replacing whitespace between terms with a single space
data['description'] = data['description'].str.replace(r'\s+',' ')

#removing leading and trailing whitespace
data['description'] = data['description'].str.replace(r'^\s+|\s+?$','')

data['description'].head()

import nltk
nltk.download('stopwords')

#Removing stop words
stop=stopwords.words('english')
```

```
import re


pattern = r'\b(?:{})\b'.format('|'.join(stop))

data['description'] = data['description'].str.replace(pattern, '')


# Removing single characters

data['description'] = data['description'].str.replace(r'\s+', ' ')

data['description'] = data['description'].apply(lambda x: " ".join([word for word in str(x).split() if len(word) > 1]))


# Removing domain related stop words from description

specific_stop_words = ["rs", "flipkart", "buy", "com", "free", "day", "cash", "replacement", "guarantee", "genuine", "key", "feature", "delivery", "products", "product", "shipping", "online", "india", "shop"]

data['description'] = data['description'].apply(lambda x: " ".join(word for word in str(x).split() if word not in specific_stop_words))


data['description'].head()


"""### Visualizing the most occured words in corpus"""


import nltk

nltk.download('punkt')

nltk.download('punkt_tab')


#most frequent words after removing domain related stopwords


# Custom stopwords list (including 'rs' and other domain-specific terms)

custom_stopwords = stopwords.words('english') + ['rs', 'type','details','guarantee','product', 'products', 'delivery', 'shipping', 'cm','price', 'features']
```

```
# Concatenate all product descriptions into a single string

a = data['description'].str.cat(sep=' ')


# Tokenize the text

words = nltk.tokenize.word_tokenize(a)


# Filter out non-alphabetic words and stopwords (both generic and domain-specific)

words = [word for word in words if re.match(r'^[a-zA-Z]+$', word) and word.lower() not in
custom_stopwords]


# Create a frequency distribution of the remaining words

word_dist = nltk.FreqDist(words)


# Plot the top 10 most frequent words

plt.figure(figsize=(10, 6))

word_dist.plot(10, cumulative=False)


# Print the top 10 most frequent words

print(word_dist.most_common(10))


print(word_dist.most_common(15))


"""# Advanced Search Engine Using PyTerrier and Sentence-BERT


#Title of the Advanced Search Engine - "Hybrid Semantic Search Engine Using PyTerrier, SymSpell, and
Sentence-BERT for Enhanced Product Retrieval"


#### installing required libraries and making setup

"""
```

```
# Install necessary packages

!pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118

#developed by facebook ai reasearch lab, provides tools for building and training neural networks

#url is to download the CUDA version for GPU acceleration

!pip install -U sentence-transformers

#built on top of hugging face transofrmers

"""It simplifies the process of generating dense vector representations (embeddings) of sentences,

paragraphs, or documents. These embeddings can be used for tasks like semantic search, text similarity, clustering, and more.

used in models like BERT , RoBERTa, and DistilBERT. to work on NLP models"""


!pip install python-terrier

#It provides tools for indexing, querying, and evaluating retrieval systems

!pip install nltk

#Natural Language Toolkit (NLTK), ibraries for tokenization, stemming, lemmatization, part-of-speech tagging, parsing, and more.

!pip install scikit-learn

#simple and efficient tool for data mining and data analysis, built on NumPy, SciPy, and matplotlib

!pip install symspellpy

#an efficient spelling correction algorithm.


# Import libraries

import pandas as pd# data manipulation and analysis

import numpy as np #numerical computing in Python.

import string #contains a collection of string constants (e.g., punctuation characters, digits, letters).

import re #module provides support for regular expressions.

from nltk.corpus import stopwords #list of common words (e.g., "the", "and", "is") that are often removed from text because they don't contribute much to the meaning.

from nltk.tokenize import word_tokenize # splits text into individual words or tokens.
```

```
from nltk.stem.wordnet import WordNetLemmatizer #reduces words to their base or dictionary form (lemma).

import nltk

nltk.download('punkt')# for tokenization

nltk.download('wordnet') #for lemmatization

nltk.download('stopwords') #for stopword removal


!pip install --upgrade python-terrier

# Initialize PyTerrier -  for information retrieval (IR) tasks.

import pyterrier as pt

# Check if PyTerrier is already initialized. If not, initialize it

if not pt.started():  # Replace pt.started() with pt.init()

   pt.init()  # This line initializes PyTerrier


# Uninstall the current torchvision

!pip uninstall -y torchvision


# Reinstall torchvision specifying the CUDA version

!pip install torchvision --index-url https://download.pytorch.org/whl/cu118


# Import SentenceTransformer for embeddings

from sentence_transformers import SentenceTransformer

model = SentenceTransformer('sentence-transformers/bert-base-nli-mean-tokens')


# Import SymSpell for spelling correction

from symspellpy import SymSpell, Verbosity


"""#### Data preprocessing"""
```

```
# Load the dataset
df = pd.read_csv("/content/flipkart_com-ecommerce_sample.csv")

# Clean product category tree
df['product_category_tree'] = df['product_category_tree'].str.replace('>>', ',')
df['product_category_tree'] = df['product_category_tree'].str.replace('"', '')

# Drop unnecessary columns
df.drop(['product_url', 'image', "retail_price", "discounted_price",
        "is_FK_Advantage_product", "product_rating", "overall_rating", "product_specifications"],
       axis=1, inplace=True)

# Remove duplicate products
uniq_prod = df.copy()
uniq_prod.drop_duplicates(subset="product_name", keep="first", inplace=True)

# Define stopwords and punctuation
stop_words = set(stopwords.words('english'))
exclude = set(string.punctuation)
lem = WordNetLemmatizer()

# Function to clean text
def filter_keywords(doc):
    doc = doc.lower()
    stop_free = " ".join([i for i in doc.split() if i not in stop_words])
    punc_free = "".join(ch for ch in stop_free if ch not in exclude)
    word_tokens = word_tokenize(punc_free)  # Tokenize the text
    filtered_sentence = [(lem.lemmatize(w, "v")) for w in word_tokens]  # Lemmatize tokens
    return " ".join(filtered_sentence)
```

```
# Apply cleaning to relevant columns
# Convert the 'product_name' column to string before applying filter_keywords
uniq_prod['product'] = uniq_prod['product_name'].astype(str).apply(filter_keywords)
uniq_prod['brand'] = uniq_prod['brand'].astype(str).apply(filter_keywords)
uniq_prod['description'] = uniq_prod['description'].astype(str).apply(filter_keywords)

# Combine all keywords for each product
uniq_prod["keywords"] = (
    uniq_prod['product'] + " " +
    uniq_prod['brand'] + " " +
    uniq_prod['product_category_tree']
)

# Create a 'docno' column for recommendations
uniq_prod['docno'] = uniq_prod['product_name'].astype(str)

"""#### Spell Correction with SymSpell"""

# Download dictionaries for SymSpell
!wget https://raw.githubusercontent.com/mammothb/symspellpy/master/symspellpy/frequency_dictionary_en_82_765.txt
!wget https://raw.githubusercontent.com/mammothb/symspellpy/master/symspellpy/frequency_bigramdictionary_en_243_342.txt

# Initialize SymSpell
sym_spell = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)
```

```
# Load the pre-built dictionary
dictionary_path = "frequency_dictionary_en_82_765.txt"
bigram_path = "frequency_bigramdictionary_en_243_342.txt"

if not sym_spell.load_dictionary(dictionary_path, term_index=0, count_index=1):
    print("Dictionary file not found!")
if not sym_spell.load_bigram_dictionary(bigram_path, term_index=0, count_index=2):
    print("Bigram dictionary file not found!")

# Function to correct spelling
def correct_spelling(text):
    suggestions = sym_spell.lookup_compound(text, max_edit_distance=2)
    corrected_text = suggestions[0].term
    return corrected_text

# Apply spell correction to the keywords
uniq_prod["corrected_keywords"] = uniq_prod["keywords"].astype(str).apply(correct_spelling)

"""#### Indexing with PyTerrier"""

# Create a DataFrame for indexing
index_data = uniq_prod[['docno', 'corrected_keywords']]
index_data.columns = ['docno', 'text']

# Index the data
indexer = pt.DFIndexer("./index", overwrite=True)# creates an instance of a document frequency (DF) indexer
index_ref = indexer.index(index_data['text'], index_data['docno']) # This line indexes the documents, a reference to the created index.
```

```
# Retrieve documents using BatchRetrieve
retriever = pt.BatchRetrieve(index_ref, wmodel="BM25")

"""#### Semantic Search with Sentence-BERT"""

from sklearn.metrics.pairwise import cosine_similarity

# Function to compute semantic similarity
def compute_semantic_similarity(query, documents):
    query_embedding = model.encode([query])
    document_embeddings = model.encode(documents)
    similarities = cosine_similarity(query_embedding, document_embeddings).flatten()
    return similarities

"""#### Combined Search and Ranking"""

# Function to take user input and display results
def search_products(query):
    # Step 1: Spell correction
    corrected_query = correct_spelling(query)
    print(f"Corrected Query: {corrected_query}")

    # Step 2: Retrieve documents using BM25
    results = retriever.search(corrected_query)
    results = results.merge(uniq_prod, on='docno', how='left')

    # Step 3: Compute semantic similarity
    similarities = compute_semantic_similarity(corrected_query, results['corrected_keywords'].tolist())
```

```
    results['similarity_score'] = similarities

    # Step 4: Rank results by combining BM25 and semantic similarity
    results['final_score'] = results['score'] + results['similarity_score']
    ranked_results = results.sort_values(by='final_score', ascending=False)

    # Display top results
    return ranked_results[['product_name', 'brand', 'description', 'final_score']].head(10)

# Take user input
user_query = input("Enter your search query: ")
search_results = search_products(user_query)
print("\nSearch Results:")
search_results

"""**********

# Final code including soft computng Implementation -Fuzzy logic

**********
"""

# Install necessary packages
!pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118
!pip install -U sentence-transformers
!pip install python-terrier
!pip install nltk
!pip install scikit-learn
!pip install symspellpy
```

```
!pip install fuzzywuzzy
!pip install python-Levenshtein
!pip install spacy

# Download spaCy model
!python -m spacy download en_core_web_md

# # Uninstall the current torchvision
# !pip uninstall -y torchvision

# # Reinstall torchvision specifying the CUDA version
# !pip install torchvision --index-url https://download.pytorch.org/whl/cu118

# Import libraries
import pandas as pd
import numpy as np
import string
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem.wordnet import WordNetLemmatizer
import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')

# Initialize PyTerrier
import pyterrier as pt
if not pt.started():
```

```
    pt.init()

# Import SentenceTransformer for embeddings
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('sentence-transformers/bert-base-nli-mean-tokens')

# Import SymSpell for spelling correction
from symspellpy import SymSpell, Verbosity

# Import FuzzyWuzzy for fuzzy matching
from fuzzywuzzy import fuzz

# Import spaCy for query expansion
import spacy
nlp = spacy.load("en_core_web_md")

import nltk
nltk.download('punkt_tab')

# Load the dataset

df = pd.read_csv("/content/flipkart_com-ecommerce_sample.csv")

# Clean product category tree
df['product_category_tree'] = df['product_category_tree'].str.replace('>>', ',')
df['product_category_tree'] = df['product_category_tree'].str.replace('"', '')

# Drop unnecessary columns
df.drop(['product_url', 'image', "retail_price", "discounted_price",
```

```
        "is_FK_Advantage_product", "product_rating", "overall_rating", "product_specifications"],
       axis=1, inplace=True)

# Remove duplicate products
uniq_prod = df.copy()
uniq_prod.drop_duplicates(subset="product_name", keep="first", inplace=True)

# Define stopwords and punctuation
stop_words = set(stopwords.words('english'))
exclude = set(string.punctuation)
lem = WordNetLemmatizer()

# Function to clean text
def filter_keywords(doc):
    doc = doc.lower()
    stop_free = " ".join([i for i in doc.split() if i not in stop_words])
    punc_free = "".join(ch for ch in stop_free if ch not in exclude)
    word_tokens = word_tokenize(punc_free)  # Tokenize the text
    filtered_sentence = [(lem.lemmatize(w, "v")) for w in word_tokens]  # Lemmatize tokens
    return " ".join(filtered_sentence)

# Apply cleaning to relevant columns
uniq_prod['product'] = uniq_prod['product_name'].astype(str).apply(filter_keywords)
uniq_prod['brand'] = uniq_prod['brand'].astype("str").apply(filter_keywords)
uniq_prod['description'] = uniq_prod['description'].astype(str).apply(filter_keywords)

# Combine all keywords for each product
uniq_prod["keywords"] = (
    uniq_prod['product'] + " " +
```

```
    uniq_prod['brand'] + " " +
    uniq_prod['product_category_tree']
)

# Create a 'docno' column for recommendations
uniq_prod['docno'] = uniq_prod['product_name'].astype(str)

from sklearn.metrics.pairwise import cosine_similarity

# Function to expand query with related words using spaCy
def expand_query(query, topn=5):
    expanded_words = []
    doc = nlp(query)

    for token in doc:
        # Prioritize nouns over adjectives and ensure the token has a valid vector
        if token.pos_ == "NOUN" and token.has_vector:
            try:
                # Find the most similar words to the current noun
                similar_words = [
                    word.text for word in nlp.vocab
                    if word.has_vector and nlp(token.text).similarity(nlp(word.text)) > 0.7
                ][:topn]
                expanded_words.extend(similar_words)
            except KeyError:
                continue

    # Remove duplicates and filter out irrelevant words
    expanded_words = list(set(expanded_words))
```

```
    return " ".join(expanded_words)


# # Download dictionaries for SymSpell

# !wget https://raw.githubusercontent.com/mammothb/symspellpy/master/symspellpy/frequency_dictionary_en_82_765.txt

# !wget https://raw.githubusercontent.com/mammothb/symspellpy/master/symspellpy/frequency_bigramdictionary_en_243_342.txt


# Initialize SymSpell

sym_spell = SymSpell(max_dictionary_edit_distance=2, prefix_length=7)


# Load the pre-built dictionary

dictionary_path = "frequency_dictionary_en_82_765.txt"

bigram_path = "frequency_bigramdictionary_en_243_342.txt"


if not sym_spell.load_dictionary(dictionary_path, term_index=0, count_index=1):

    print("Dictionary file not found!")

if not sym_spell.load_bigram_dictionary(bigram_path, term_index=0, count_index=2):

    print("Bigram dictionary file not found!")


# Function to correct spelling

def correct_spelling(text):

    suggestions = sym_spell.lookup_compound(text, max_edit_distance=2)

    corrected_text = suggestions[0].term

    return corrected_text


# Apply spell correction to the keywords

uniq_prod["corrected_keywords"] = uniq_prod["keywords"].astype(str).apply(correct_spelling)
```

```
# Create a DataFrame for indexing
index_data = uniq_prod[['docno', 'corrected_keywords']]
index_data.columns = ['docno', 'text']

# Index the data
indexer = pt.DFIndexer("./index", overwrite=True)
index_ref = indexer.index(index_data['text'], index_data['docno'])

# Retrieve documents using BatchRetrieve
retriever = pt.BatchRetrieve(index_ref, wmodel="BM25")

# Function to compute fuzzy relevance score
def fuzzy_relevance(query, product_name, description, brand):
    name_similarity = fuzz.token_set_ratio(query, product_name)
    desc_similarity = fuzz.token_set_ratio(query, description)
    brand_similarity = fuzz.token_set_ratio(query, brand)
    partial_similarity = fuzz.partial_ratio(query, product_name + " " + description)
    return 0.3 * name_similarity + 0.3 * brand_similarity + 0.2 * desc_similarity + 0.2 * partial_similarity
    # return 0.5 * name_similarity + 0.3 * desc_similarity + 0.2 * partial_similarity

from nltk.corpus import wordnet

def infer_category(category):
    category_synsets = wordnet.synsets(category)
    hyponyms = set()

    for synset in category_synsets:
        for hyponym in synset.hyponyms():  # Get specific items under the category
```

```
            hyponyms.update(hyponym.lemma_names())

    return list(hyponyms)

# print(get_hyponyms("furniture"))  # Output: ['sofa', 'bed', 'chair', 'table', etc.]

# Filter results by inferred category
def filter_by_category(results, query):
    category = infer_category(query)
    if category:
        results = results[results['product_category_tree'].str.contains(category, case=False)]
    return results

# # Function to handle search and display results
# def search_products_with_embeddings(query):
#     # Step 1: Spell correction
#     corrected_query = correct_spelling(query)
#     print(f"Corrected Query: {corrected_query}")

#     # Step 2: Expand query with related words
#     querys = ' '.join(infer_category(query))
#     expanded_query = query + " " + corrected_query + " " + expand_query(corrected_query)

#     # Step 3: Retrieve documents using BM25
#     results = retriever.search(expanded_query)
#     results = results.merge(uniq_prod, on='docno', how='left')

#     print(f"Initial results count: {len(results)}")  # Debugging check
```

```
#     # Check if results are less than 5 and expand query further
#     if len(results) < 5 and querys.strip():
#         print("Expanding query further for better results...")
#         additional_expansion = expand_query(querys)

#         if additional_expansion.strip():  # Ensure it's not empty
#             expanded_query += " " + additional_expansion
#             results = retriever.search(expanded_query)
#             results = results.merge(uniq_prod, on='docno', how='left')

#     print(f"Expanded Query: {expanded_query}")
#     print(f"Results count after expansion: {len(results)}")  # Debugging check

#     # Check if results are empty after BM25 retrieval
#     if results.empty:
#         print("No products found using BM25 retrieval.")
#         return suggest_alternatives(query)

#     # Step 4: Filter by category
#     results = filter_by_category(results, expanded_query)

#     # Check if results are empty after category filtering
#     if results.empty:
#         print("No products found after category filtering.")
#         return suggest_alternatives(query)

#     # Step 5: Compute semantic similarity
#     try:
#         query_embedding = model.encode([expanded_query])
```

```
#         document_embeddings = model.encode(results['corrected_keywords'].tolist())

#         if len(document_embeddings) == 0:
#             print("No valid embeddings found for the retrieved documents.")
#             return suggest_alternatives(query)

#         similarities = cosine_similarity(query_embedding, document_embeddings).flatten()
#         results['similarity_score'] = similarities
#     except Exception as e:
#         print(f"Error during semantic similarity computation: {e}")
#         return suggest_alternatives(query)

#     # Step 6: Compute fuzzy relevance
#     results['fuzzy_score'] = results.apply(
#         lambda row: fuzzy_relevance(expanded_query, row['product_name'], row['description'], row['brand']), axis=1
#     )

#     # Step 7: Filter results based on semantic similarity threshold
#     SEMANTIC_THRESHOLD = 0.6
#     results = results[results['similarity_score'] >= SEMANTIC_THRESHOLD]

#     if results.empty:
#         print("No products found after semantic filtering.")
#         return suggest_alternatives(query)

#     # Step 8: Rank results by combining BM25, semantic similarity, and fuzzy scores
#     results['final_score'] = (
#         results['score'] * 0.5 +
```

```
#         results['similarity_score'] * 0.3 +
#         results['fuzzy_score'] * 0.2
#     )
#     ranked_results = results.sort_values(by='final_score', ascending=False)

#     if ranked_results.empty:
#         print("Product not available.")
#         return suggest_alternatives(query)
#     else:
#         return ranked_results[['product_name', 'brand', 'description', 'final_score']].head(20)

# # Suggest alternative products if no match is found
# def suggest_alternatives(query):
#     suggestions = retriever.search(query)
#     if suggestions.empty:
#         return "No relevant products found."
#     else:
#         return suggestions.head(20)

# # Function to handle search and display results
# def search_products_with_embeddings(query):
#     # Step 1: Spell correction
#     corrected_query = correct_spelling(query)
#     print(f"Corrected Query: {corrected_query}")

#     # Step 2: Expand query with related words
#     # querys = ' '.join(infer_category(query))
#     expanded_query = query + " " + corrected_query + " " + expand_query(corrected_query)
```

```
#     # Step 3: Retrieve documents using BM25
#     results = retriever.search(expanded_query)
#     results = results.merge(uniq_prod, on='docno', how='left')
#     pre_semantic_results = results.copy();

#     print(f"Initial results count: {len(results)}")
#     # print(results.head(10))

#     # Step 4: Filter by category
#     results = filter_by_category(results, expanded_query)
#     print(f"Results count after ctegory filter: {len(results)}")

#     # Step 5: Compute semantic similarity
#     try:
#         query_embedding = model.encode([expanded_query])
#         document_embeddings = model.encode(results['corrected_keywords'].tolist())

#         if len(document_embeddings) == 0:
#             print("No valid embeddings found for the retrieved documents.")
#             return suggest_alternatives(query)

#         similarities = cosine_similarity(query_embedding, document_embeddings).flatten()
#         results['similarity_score'] = similarities
#     except Exception as e:
#         print(f"Error during semantic similarity computation: {e}")
#         return suggest_alternatives(query)

#     # Step 6: Compute fuzzy relevance
#     results['fuzzy_score'] = results.apply(
```

```
#         lambda row: fuzzy_relevance(expanded_query, row['product_name'], row['description'], row['brand']), axis=1
#     )

#     # Step 7: Filter results based on semantic similarity threshold
#     SEMANTIC_THRESHOLD = 0.6
#     results = results[results['similarity_score'] >= SEMANTIC_THRESHOLD]

#     # Step 8: Rank results by combining BM25, semantic similarity, and fuzzy scores
#     results['final_score'] = (
#         results['score'] * 0.5 +
#         results['similarity_score'] * 0.3 +
#         results['fuzzy_score'] * 0.2
#     )
#     ranked_results = results.sort_values(by='final_score', ascending=False)

#     print(f"Final results count: {len(ranked_results)}")

#     # 🔹 If final results are still < 5, expand the query further and repeat search
#     if len(ranked_results) < 5:
#       querys = ' '.join(infer_category(query))
#       if (querys.strip()):
#         print("Expanding query further for better results...")
#         additional_expansion = expand_query(querys)

#         if additional_expansion.strip():  # Ensure it's not empty
#             expanded_query += " " + additional_expansion
#             results = retriever.search(expanded_query)
#             results = results.merge(uniq_prod, on='docno', how='left')
```

```
#         # Repeat Steps 4-8 for updated results
#         results = filter_by_category(results, expanded_query)

#         try:
#             query_embedding = model.encode([expanded_query])
#             document_embeddings = model.encode(results['corrected_keywords'].tolist())

#             if len(document_embeddings) > 0:
#                 similarities = cosine_similarity(query_embedding, document_embeddings).flatten()
#                 results['similarity_score'] = similarities
#         except Exception as e:
#             print(f"Error during semantic similarity computation: {e}")
#             return suggest_alternatives(query)

#         results['fuzzy_score'] = results.apply(
#             lambda row: fuzzy_relevance(expanded_query, row['product_name'], row['description'], row['brand']), axis=1
#         )

#         results = results[results['similarity_score'] >= SEMANTIC_THRESHOLD]

#         results['final_score'] = (
#             results['score'] * 0.5 +
#             results['similarity_score'] * 0.3 +
#             results['fuzzy_score'] * 0.2
#         )
#         ranked_results = results.sort_values(by='final_score', ascending=False)
```

```
#         print(f"Results count after additional expansion: {len(ranked_results)}")


#     # Display results or suggest alternatives
#     if ranked_results.empty:
#         print("Product not available.")
#         if results.empty:
#           print("No products found after semantic filtering. Returning pre-semantic results.")
#           # return pre_semantic_results if not pre_semantic_results.empty else suggest_alternatives(query)
#           return suggest_alternatives(query) if not retriever.search(query).empty else pre_semantic_results
#         # return suggest_alternatives(query)
#         # return suggest_alternatives(query) if pre_semantic_results.empty else pre_semantic_results
#     else:
#         return ranked_results[['product_name', 'brand', 'description', 'final_score']].head(20)


# # Suggest alternative products if no match is found
# def suggest_alternatives(query):
#     suggestions = retriever.search(query)
#     if suggestions.empty:
#         return "No relevant products found."
#     else:
#         # Merge suggestions with uniq_prod to get product details
#         suggestions = suggestions.merge(uniq_prod, on='docno', how='left')
#         return suggestions[['product_name', 'brand', 'description']].head(30)


# # Function to handle search and display results
# def search_products_with_embeddings(query):
#     # Step 1: Spell correction
```

```
#     corrected_query = correct_spelling(query)
#     print(f"Corrected Query: {corrected_query}")

#     # Step 2: Expand query with related words
#     expanded_query = f"{query} {corrected_query} {expand_query(corrected_query)}"
#     print(expanded_query)

#     # Step 3: Retrieve documents using BM25
#     results = retriever.search(expanded_query)
#     results = results.merge(uniq_prod, on="docno", how="left")
#     pre_semantic_results = results.copy()

#     # Step 4: Filter by category
#     results = filter_by_category(results, expanded_query)
#     print(f"Results count after category filter: {len(results)}")

#     # Step 5: Compute semantic similarity
#     try:
#         query_embedding = model.encode([expanded_query])
#         document_embeddings = model.encode(results["corrected_keywords"].tolist())

#         if not len(document_embeddings):
#             print("No valid embeddings found for the retrieved documents.")
#             return suggest_alternatives(query, pre_semantic_results)

#         results["similarity_score"] = cosine_similarity(query_embedding, document_embeddings).flatten()
#     except Exception as e:
#         print(f"Error during semantic similarity computation: {e}")
#         return suggest_alternatives(query, pre_semantic_results)
```

```
#     # Step 6: Compute fuzzy relevance
#     results["fuzzy_score"] = results.apply(
#         lambda row: fuzzy_relevance(expanded_query, row["product_name"], row["description"], row["brand"]),
#         axis=1,
#     )


#     # Step 7: Filter results based on semantic similarity threshold
#     SEMANTIC_THRESHOLD = 0.6
#     results = results[results["similarity_score"] >= SEMANTIC_THRESHOLD]


#     # Step 8: Rank results by combining BM25, semantic similarity, and fuzzy scores
#     results["final_score"] = (
#         results["score"] * 0.5 + results["similarity_score"] * 0.3 + results["fuzzy_score"] * 0.2
#     )
#     ranked_results = results.sort_values(by="final_score", ascending=False)
#     print(f"Final results count: {len(ranked_results)}")


#     # Step 9 If final results are still < 5, expand the query further and repeat search
#     if len(ranked_results) < 5:
#         inferred_category = " ".join(infer_category(query))
#         if inferred_category.strip():
#             print("Expanding query further for better results...")
#             additional_expansion = expand_query(inferred_category)
```

```
#         if additional_expansion.strip():
#             expanded_query += f" {additional_expansion}"
#             results = retriever.search(expanded_query).merge(uniq_prod, on="docno", how="left")

#             # Repeat Steps 4-8 for updated results
#             results = filter_by_category(results, expanded_query)

#             try:
#                 query_embedding = model.encode([expanded_query])
#                 document_embeddings = model.encode(results["corrected_keywords"].tolist())

#                 if len(document_embeddings) > 0:
#                     results["similarity_score"] = cosine_similarity(query_embedding, document_embeddings).flatten()
#             except Exception as e:
#                 print(f"Error during semantic similarity computation: {e}")
#                 return suggest_alternatives(query, pre_semantic_results)

#             results["fuzzy_score"] = results.apply(
#                 lambda row: fuzzy_relevance(expanded_query, row["product_name"], row["description"], row["brand"]),
#                 axis=1,
#             )

#             results = results[results["similarity_score"] >= SEMANTIC_THRESHOLD]

#             results["final_score"] = (
#                 results["score"] * 0.5 + results["similarity_score"] * 0.3 + results["fuzzy_score"] * 0.2
#             )
```

```
#                ranked_results = pd.concat([ranked_results, results.sort_values(by="final_score",
ascending=False)])


#            print(f"Results count after additional expansion: {len(ranked_results)}")


#     # Display results or suggest alternatives
#     if len(ranked_results)<5:
#         if ranked_results.empty:
#             print("Product not available.")
#             if results.empty:
#                 print("No products found after semantic filtering. Returning pre-semantic results.")
#             print(ranked_results[["product_name", "brand", "description", "final_score"]].head(20))
#             return suggest_alternatives(query, pre_semantic_results)
#         return suggest_alternatives(query, pre_semantic_results)

#     # # return ranked_results[["product_name", "brand", "description", "final_score"]].head(20)

# # if len(ranked_results) < 5:
# #     print(ranked_results[["product_name", "brand", "description", "final_score"]].head(20))

# #     if ranked_results.empty:
# #         print("Product not available.")

# #         if results.empty:
# #             print("No products found after semantic filtering. Returning pre-semantic results.")

# #         return suggest_alternatives(query, pre_semantic_results)
```

```
# #     return suggest_alternatives(query, pre_semantic_results)


# # Suggest alternative products if no match is found
# def suggest_alternatives(query, pre_semantic_results):
#     suggestions = retriever.search(query).merge(uniq_prod, on="docno", how="left")
#     if suggestions.empty:
#         return pre_semantic_results[["product_name", "brand", "description"]].head(30)

#     return suggestions[["product_name", "brand", "description"]].head(30)

# Function to handle search and display results
def search_products_with_embeddings(query):
    # Step 1: Spell correction
    corrected_query = correct_spelling(query)
    print(f"Corrected Query: {corrected_query}")

    # Step 2: Expand query with related words
    expanded_query = f"{query} {corrected_query} {expand_query(corrected_query)}"
    print(expanded_query)

    # Step 3: Retrieve documents using BM25
    results = retriever.search(expanded_query).merge(uniq_prod, on="docno", how="left")
    pre_semantic_results = results.copy()

    # Step 4: Filter by category
    results = filter_by_category(results, expanded_query)
    print(f"Results count after category filter: {len(results)}")
```

```
    # Step 5: Compute semantic similarity
    try:
        query_embedding = model.encode([expanded_query])
        document_embeddings = model.encode(results["corrected_keywords"].tolist())

        if not len(document_embeddings):
            print("No valid embeddings found for the retrieved documents.")
            return suggest_alternatives(query, pre_semantic_results)

        results["similarity_score"] = cosine_similarity(query_embedding, document_embeddings).flatten()
    except Exception as e:
        print(f"Error during semantic similarity computation: {e}")
        return suggest_alternatives(query, pre_semantic_results)

    # Step 6: Compute fuzzy relevance
    results["fuzzy_score"] = results.apply(
        lambda row: fuzzy_relevance(expanded_query, row["product_name"], row["description"],
row["brand"]),
        axis=1,
    )

    # Step 7: Filter results based on semantic similarity threshold
    SEMANTIC_THRESHOLD = 0.6
    results = results[results["similarity_score"] >= SEMANTIC_THRESHOLD]

    # Step 8: Rank results by combining BM25, semantic similarity, and fuzzy scores
    results["final_score"] = (
        results["score"] * 0.5 + results["similarity_score"] * 0.3 + results["fuzzy_score"] * 0.2
    )
```

```
    ranked_results = results.sort_values(by="final_score", ascending=False)
    print(f"Final results count: {len(ranked_results)}")

    # Step 9: Expand query if results < 5
    if len(ranked_results) < 5:
        inferred_category = " ".join(infer_category(query))
        if inferred_category.strip():
            print("Expanding query further for better results...")
            additional_expansion = expand_query(inferred_category)

            if additional_expansion.strip():
                expanded_query += f" {additional_expansion}"
                results = retriever.search(expanded_query).merge(uniq_prod, on="docno", how="left")

                # Repeat Steps 4-8 for updated results
                results = filter_by_category(results, expanded_query)

                try:
                    query_embedding = model.encode([expanded_query])
                    document_embeddings = model.encode(results["corrected_keywords"].tolist())

                    if len(document_embeddings) > 0:
                        results["similarity_score"] = cosine_similarity(query_embedding,
document_embeddings).flatten()
                except Exception as e:
                    print(f"Error during semantic similarity computation: {e}")
                    return suggest_alternatives(query, pre_semantic_results)

                results["fuzzy_score"] = results.apply(
```

```
            lambda row: fuzzy_relevance(expanded_query, row["product_name"], row["description"],
row["brand"]),
            axis=1,
        )

        results = results[results["similarity_score"] >= SEMANTIC_THRESHOLD]

        results["final_score"] = (
            results["score"] * 0.5 + results["similarity_score"] * 0.3 + results["fuzzy_score"] * 0.2
        )

        ranked_results = pd.concat([ranked_results, results.sort_values(by="final_score",
ascending=False)])

    print(f"Results count after additional expansion: {len(ranked_results)}")

# Step 10: Display results or suggest alternatives
if len(ranked_results) > 5:
  return ranked_results[["product_name", "brand", "description", "final_score"]].head(20)
elif ranked_results.empty:
    print("Product not available.")
    if results.empty:
        print("No products found after semantic filtering. Returning pre-semantic results.")
    return suggest_alternatives(query, pre_semantic_results)
elif len(ranked_results) < 5:
    res = pd.concat(ranked_results, suggest_alternatives(query, pre_semantic_results))
    return res[["product_name", "brand", "description", "final_score"]].head(20)

# Suggest alternative products if no match is found
```

```
def suggest_alternatives(query, pre_semantic_results):
    suggestions = retriever.search(query).merge(uniq_prod, on="docno", how="left")
    # if suggestions.empty:
    if len(suggestions) < 10:
        final_results = pd.concat([suggestions, pre_semantic_results])
        if len(final_results) == 0:
            return "No Results Found"
        return final_results[["product_name", "brand", "description"]].head(30)

    return suggestions[["product_name", "brand", "description"]].head(30)


# def suggest_alternatives(query, pre_semantic_results):
#     suggestions = retriever.search(query).merge(uniq_prod, on="docno", how="left")
#     # if suggestions.empty:
#     if len(suggestions) < 10:
#         suggestions = suggestions if not suggestions.empty else pd.DataFrame(columns=pre_semantic_results.columns)
#         final_results = pd.concat([suggestions, pre_semantic_results])
#         return final_results[["product_name", "brand", "description"]].head(30)
#     else:
#         return suggestions[["product_name", "brand", "description"]].head(30) if not suggestions.empty else "No Results Found"


# Take user input
user_query = input("Enter your search query: ")
search_results = search_products_with_embeddings(user_query)
print("\nSearch Results:")
search_results
```