```
import pandas as pd
import os
from sentence_transformers import SentenceTransformer
import json
from tqdm.auto import tqdm
import os
import sys
import ast
import re
import json
import cohere
import streamlit as st
import pandas as pd
from tqdm.auto import tqdm
import pinecone
from collections import Counter
import tiktoken
import openai
from transformers import AutoTokenizer
from sentence_transformers import SentenceTransformer
from langchain.text_splitter import RecursiveCharacterTextSplitter
import pinecone
from transformers import AutoTokenizer
from langchain.chains.conversation.memory import
ConversationBufferWindowMemory
from langchain.callbacks.base import BaseCallbackHandler
from langchain.chains.question_answering import load_qa_chain
from langchain.retrievers import PineconeHybridSearchRetriever
from langchain.callbacks.manager import CallbackManagerForRetrieverRun
from langchain.chains import ConversationalRetrievalChain
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from collections import Counter
```

Potentially interesting features for Secova

Define df (sample dataset in pandas)

```
df = pd.read_csv("secova.csv", names=['Column A', 'Column B',
'Summary', 'Main Text', 'Site'])
#df_processed = pd.read_csv("secova_processed.csv", names=['Column A',
'Column B', 'Summary', 'Main Text', 'Site'])
```

Preprocessing

Index

```
# Reset the index and assign it to the 'ID' column
df['ID'] = df.index
df = df[['ID'] + [col for col in df if col != 'ID']]
df = df.reset_index()
df['ID'] = df.index
df = df.iloc[:, 1:]
```

Nulls

Find nulls

```
# Use list comprehension to get the indices of null entries in the
specified column
null_indices_a = [index for index, value in enumerate(df['Number of
entries with null Column A'].isna()) if value]
null_indices_b = [index for index, value in enumerate(df['Number of
entries with null Column B'].isna()) if value]

# Print the list of null indices
print('Number of entries with null Column A:', len(null_indices_a))
print('Number of entries with null Column B:', len(null_indices_b))
```

This code is extracting specific information from rows of a DataFrame that have null or missing values, and writing this information to a text file.

```
result = []

for index in null_indices_a:
    row = df.loc[df['Index_Column'] == index]
    result.append(['Index: ' + str(index),
                   'Column with null entries: ' +
str(row['Column'].values[0])
                  ]
                 )
# Save to txt file
with open('nulls.txt', 'w') as f:
    for item in result:
        f.write("%s\n" % item)
```

This code is extracting specific information from rows of a DataFrame that have null or missing values, creating an HTML representation of this information with hyperlinks, and writing this HTML representation to a file.

```
result = []

for index in null_indices_a:
    row = df.loc[df['Index_Column'] == index]
    result.append([
        'Index: ' + str(index),
        'Column with null entries: ' + str(row['Column'].values[0]),
        'Site corresponding to null entry: ' +
str(row['Site'].values[0])
    ])

# Create an HTML representation of the result
html_result = '<html>\n<head><title>Nulls with
Hyperlinks</title></head>\n<body>\n'

for item in result:
    website_url = item[2].split(': ')[1]  # Extract the URL from
'Site: <URL>'
    html_result += f'<p>{item[0]}<br>{item[1]}<br><a
href="{website_url}">Website Link</a></p>\n'

html_result += '</body>\n</html>'

# Save the HTML representation to a text file
with open('nulls.html', 'w') as f:
    f.write(html_result)
```

Fill null Main Text with corresponding Summary (if Main Text is meant to be embedded)

```
def process_dataframe(df):
    num_list = []
    for index, row in df.iterrows():
        if pd.isnull(row['Summary']) and pd.isnull(row['Main Text']):
# both Summary and Main Text are null
            num_list.append(int(0))
        elif pd.notnull(row['Summary']) and pd.isnull(row['Main
Text']):  # Summary is not null and Main Text is null
            df.loc[index, 'Main Text'] = df.loc[index, 'Summary']
            num_list.append(int(1))
        elif pd.isnull(row['Summary']) and pd.notnull(row['Main
Text']):  # Summary is null and Main Text is not null
            num_list.append(int(2))
        elif pd.notnull(row['Summary']) and pd.notnull(row['Main
Text']):  # both Summary and Main Text are not null
            num_list.append(int(3))
    df['Text Source'] = num_list  # Text Source is used to keep track
of where the text came from
    return df

df=process_dataframe(df)
```

```
sum_n_mt_n = df['Text Source'].tolist().count(0)
sum_nn_mt_n = df['Text Source'].tolist().count(1)
sum_n_mt_nn = df['Text Source'].tolist().count(2)
sum_nn_mt_nn = df['Text Source'].tolist().count(3)
print('Percentage with Summary and Main Text null:',
round((sum_n_mt_n/df.shape[0])*100, 1),'% do dataset')
print('Percentage with Summary not null and Main Text null:',
round((sum_nn_mt_n/df.shape[0])*100, 1),'% do dataset')
print('Percentage with Summary null and Main Text not null:',
round((sum_n_mt_nn/df.shape[0])*100, 1),'% do dataset')
print('Percentage with Summary and Main Text not null:',
round((sum_nn_mt_nn/df.shape[0])*100, 1),'% do dataset')
print('Checksum:',
(sum_n_mt_n/df.shape[0]+sum_nn_mt_n/df.shape[0]+sum_n_mt_nn/df.shape[0
]+sum_nn_mt_nn/df.shape[0])*100, '%')

# remove rows where both Summary and Main Text are null
df = df[df['Text Source'] != 0]

# drop the Text Source column
df=df.drop(['Text Source'], axis='columns')
```

Memory

Memory can be useful in order to find the size of data to be embedded and therefore also the chunk sizes after slipting (since chunking will increase the size of the dataframe)

```
# List of column names you want to calculate string lengths for
columns_to_count = ['ID', 'Column A', 'Main Text', 'Site']

# Calculate the size of the selected columns in gigabytes
total_metadata_size = df[columns_to_count].memory_usage(deep=True,
index=False).sum()/1024**2
average_metadata_size=total_metadata_size/len(df)

print(f"Total metadata size: {total_metadata_size:.1f} MB")
print(f"Average metadata size: {average_metadata_size:.4f} MB")

size_per_dimension = 4  # 4 bytes per float32
size_per_vector = size_per_dimension * 1024  # 1024 dimensions per
vector
print(f"Size per vector {size_per_vector/1024**2: .4f} MB")
print(f"Total bytes size {(size_per_vector * len(df))/1024**2: .1f}
MB")
```

The following code gets the approximate maximum number of chunks in which Main Text can be splitted so the database data limit isn't surpassed, if all metadata in "columns_to_count" is to be upserted

```
max_database_size = 3000 # 3 GB

average_number_chunks=max_database_size/(size_per_vector *
len(df)/1024**2)
print(f"Average number of chunks {average_number_chunks:.1f}")
```

This memory code can also be applied after the splitting in order to check the new size of the dataframe and the data to be upserted

Splitting

In cases where chunks of text is too big, the whole document needs to be splitted in order for the chunks to have a smaller size, this way improving the quality of resulting Q&A chatbot

Defining the length function

```
tokenizer = tiktoken.get_encoding('cl100k_base')

def tiktoken_len(text):
    tokens = tokenizer.encode(
        text
    )
    return len(tokens)

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=400,  # optimal chunk size between 300-500 tokens;
chunksize includes overlap
    chunk_overlap=40,  # number of tokens overlap between chunks (20-
60 tokens)
    length_function = tiktoken_len,
    separators=['.\n', '\t', '\n\n', ';', '...', '. '] # list of
sentence separators
)

#'\n', '\t'  and '\n\n' and will typically result from scraping the
text from a website
```

Token statistics before splitting

```
token_counts = [tiktoken_len(context) for context in df['Main Text']]

print(f"Length of dataset: {len(df)}")
print(f"""Min num tokens: {min(token_counts)}
Avg num tokens: {int(sum(token_counts) / len(token_counts))}
Max num tokens: {max(token_counts)}""")
```

Text Splitting

```
# Apply text_splitter.split_text to 'Main Text' column
df['Main Text'] = df['Main Text'].apply(text_splitter.split_text)
```

```
# Use explode to create a new row for each string in the 'Main Text'
column
df = df.explode('Main Text')
```

Token statistics after splitting

```
token_counts = [tiktoken_len(context) for context in df['Main Text']]

print(f"Length of dataset: {len(df)}")
print(f"""Min num tokens: {min(token_counts)}
Avg num tokens: {int(sum(token_counts) / len(token_counts))}
Max num tokens: {max(token_counts)}""")
```

Post-splitting processing

```
# Reset the index after explode and assign it to the 'ID' column
df = df.reset_index()
df['ID'] = df.index
df = df.iloc[:, 1:]
```

Remove escape sequences ('\n', etc.) and long empty spaces

```
df['Main Text'] = df['Main Text'].str.replace(r'\n|\n\n|\t', ' ',
regex=True)
df['Main Text'] = df['Main Text'].str.replace(r'  |   |    |    ', '
', regex=True)
df['Summary'] = df['Summary'].str.replace(r'\n|\n\n|\t\t', ' ',
regex=True)
df['Summary'] = df['Summary'].str.replace(r'  |   |    |    ', ' ',
regex=True)
```

It is good to remove the splitters in the beginning of each splitted chunk in order to have a better presentation. The prefix removing can be changed accordingly to the splitters used and their length in a string

```
def remove_prefixes(strings_list):
    for s in strings_list:
        # Check if the string starts with ". "
        if s.startswith(". "):
            # Remove the first two characters (dot and space)
            s = s[2:]
        elif s.startswith(" "):
            s = s[1:]
        elif s.startswith("; "):
            s = s[2:]
        elif s.startswith("... "):
            s = s[4:]
    return strings_list
```

```
df['Main Text'] = df['Main Text'].apply(remove_prefixes)

df.to_csv('secova_processed.csv', index=False, header=False)
```

The piece of code is meant to be used for law Main Texts. If, after splitting, there are chunks of Main Texts for the same Main Text, the name of the Main Text is identified and put in the beginning of each chunk, as well as '(Fortsetzung)', meaning that the specific chunk is a continuation of the Main Text mentioned.

In practice what happens is:

'Satzanfang' -> 'Rechtsartikel 30 (Fortsetzung) 'Satzanfang'

```
def create_main_text_dict(df):
    main_text_dict = {}
    for index, row in df.iterrows():
        words = row['Main Text'].split()
        if words[0] == 'Rechtsartikel' and row['ID'] not in
main_text_dict:
            main_text_dict[row['ID']] = ' '.join(words[:2])
    return main_text_dict

def modify_main_text(row, main_text_dict):
    if row['ID'] in main_text_dict:
        words = row['Main Text'].split()
        if words[0] != 'Main Text':
            new_words = [main_text_dict[row['ID']]]
            if row['is_duplicated']:
                new_words[0] += ' (Satzanfang)'
            new_words += words
            return ' '.join(new_words)
    return row['Main Text']

main_text_dict = create_main_text_dict(df)
df['is_duplicated'] = df.duplicated(subset='ID')
df['Main Text'] = df.apply(modify_main_text, axis=1,
args=(main_text_dict,))

# If you want to remove the temporary 'is_duplicated' column

df = df.drop(columns='is_duplicated')
df['Main Text'] = df['Main Text'].str.replace('[.;]', '', regex=True)
df.loc[:, 'Main Text'] = df['Main Text'].str.replace(r'\(Fortsetzung\)
,', '(Fortsetzung)', regex=True)
```

Embeddings

Sparse embeddings class

```
class SparseEncoder:
    def __init__(self, model_id):
        self.tokenizer = AutoTokenizer.from_pretrained(model_id)

    def build_dict(self, input_batch):
      # store a batch of sparse embeddings
      sparse_emb = []
      # iterate through input batch
      for token_ids in input_batch:
          # convert the input_ids list to a dictionary of key to
frequency values
          d = dict(Counter(token_ids))
          # remove special tokens and append sparse vectors to
sparse_emb list
          sparse_emb.append({key: d[key] for key in d if key not in
[101, 102, 103, 0]})
      # return sparse_emb list
      return sparse_emb

    def generate_sparse_vectors(self, batch_df):
      # create batch of input_ids
      inputs = self.tokenizer(
        batch_df, padding=True,
        truncation=True,
        max_length=512
      )['input_ids']
      # create sparse dictionaries
      sparse_embeds = self.build_dict(inputs)
      return sparse_embeds

    def encode_queries(self, query):
      sparse_vector = self.generate_sparse_vectors([query])[0]
      # Convert the format of the sparse vector
      indices, values = zip(*sparse_vector.items())
      return {"indices": list(indices), "values": list(values)}

model_id = 'bert-base-german-cased'
sparse_encoder = SparseEncoder(model_id)

# https://huggingface.co/bert-base-german-cased
```

Dense embeddings

```
embed = SentenceTransformer('bert-base-german-cased', device='cuda')
# 'cuda' (computing with help of graphics card) or 'cpu'
```

OpenAI embeddings

Helper function

```
OPENAI_API_KEY = 'your_api_key'
def get_embeddings_openai(text):
    try:
        response = openai.Embedding.create(
            input=text,
            model="text-embedding-ada-002"
        )
        response = response['data']
        return [x["embedding"] for x in response]
    except Exception as e:
        print(f"Error in get_embeddings_openai: {e}")
        raise
```

If you want to use the OpenAI embeddings, uncomment the line:

dense_embeds = embed.encode(context).tolist()

in the upsert code block to:

```
# # create dense vectors
# text_to_embed = []
# for text in context:
#     if tiktoken_len(text) > 8191:
#         id_big_main_text = batch_df.loc[batch_df['main_text'] ==
text, 'ID'].values[0]
#         with open('rest_combined_output.txt', 'a') as f:
#             f.write(f'main_text with too big token length to be
upserted: {id_big_text}\n')
#             f.write(f'IDs of vectors not upserted: {ids}\n')
#             continue
#     else:
#         text_to_embed.append(text)
# dense_embeds = embed.embed_documents(context)
```

This code block makes sure the text is not too big to be embedded by the OpenAI embedding model, which has a limit of input tokens

Index

```
PINECONE_API_KEY = 'your_api_key_here'
PINECONE_ENVIRONMENT = 'your_environment_here'

pinecone.init(
    api_key=PINECONE_API_KEY,
    environment=PINECONE_ENVIRONMENT
)
```

```
if len(pinecone.list_indexes()) == 0:
    #create the index
    pinecone.create_index(
        "secova",
        dimension = 1024,
        metric = "dotproduct",
        pod_type = "p1.x8"
    )

index=pinecone.Index('secova')  # connect the database to an index
pinecone.describe_index('secova')  # check the index

index.describe_index_stats()
```

Upserting (inserting or updating)

Asycronous upsert (faster than normal upsert) (without OpenAI embeddings)

```
# Define a function to split a list into chunks of n size
def chunks(lst, n):
    """Yield successive n-sized chunks from lst."""
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

# Set the batch size
batch_size = 100

# Open a Pinecone index with 30 threads
with pinecone.Index('index_name', pool_threads=30) as index:
    # Loop over the dataframe in batches
    for i in tqdm(range(0, len(df), batch_size)):
        # Find the end of the current batch
        i_end = min(i+batch_size, len(df))
        # Extract the current batch from the dataframe
        batch_df = df[i:i_end]
        # Create unique IDs for the batch
        ids = [str((df['ID'][x])) for x in range(i, i_end)]
        # Initialize an empty list to store metadata
        metadatas=[]
        # Loop over the batch to extract metadata
        for j in range(len(batch_df)):
            metadatas.append({
                'column_a': batch_df['Column A'].iloc[j],
                'main_text': batch_df['main_text'].iloc[j]
            })

        # Convert the 'main_text' column to a list
```

```
        context = batch_df['main_text'].tolist()

        # Generate sparse vectors from the context
        sparse_embeds =
sparse_encoder.generate_sparse_vectors(context)
        # Transform the sparse vectors to the desired format
        for j, sparse in enumerate(sparse_embeds):
            sparse_embeds[j] = {
            'indices': list(sparse.keys()),
            'values': [float(value) for value in sparse.values()]
        }

        # Check if any sparse vectors are empty and write them to a
file
        for sparse in sparse_embeds:
            if len(sparse['indices']) == 0 or len(sparse['values']) ==
0:
                with open('combined_output.txt', 'a') as f:
                    f.write(f'ID with empty sparse vector: {_id}\n')
                    continue

        # Generate dense vectors from the context
        dense_embeds = embed.encode(context).tolist()

        # Initialize an empty list to store vectors
        vectors=[]
        last_id = None
        try:
            # Loop over the IDs, sparse vectors, dense vectors, and
metadata
            for _id, sparse, dense, metadata in zip(ids,
sparse_embeds, dense_embeds, metadatas):
                last_id = _id
                # Check if the size of the vector is too large
                if sys.getsizeof(json.dumps([_id, sparse, dense,
metadata])) >= 40960:
                    # If the vector is too large, keep only the
context in the metadata
                    metadata = {'context': metadata.get('context')}
                    # Check again if the size of the vector is too
large
                    if sys.getsizeof(json.dumps([_id, sparse, dense,
metadata])) >= 40960:
                        # If the vector is still too large, write the
ID to a file and continue to the next iteration
                        with open('combined_output.txt', 'a') as f:
                            f.write(f'ID with too big context (not
upserted): {_id}\n')
                            continue
                else:
```

```
                    # If the data is not too large, append it to the
vectors list
                    vectors.append({
                        'id': _id,
                        'sparse_values': sparse,
                        'values': dense,
                        'metadata': metadata
                    })
            # Upsert the vectors in chunks and store the results
asynchronously
            async_results = [
                index.upsert(vectors=vectors_chunk, async_req=True)
                for vectors_chunk in chunks(vectors, 10)
            ]
            # Write the last ID upserted to a file
            with open('last_id.txt', 'a') as f:
                f.write(f"Last id upserted: {last_id}")
        except Exception as e:
            # If an error occurs, print the ID and the error details
            print(f"Error occurred while upserting the vector with id:
{last_id}")
            print(f"Error details: {str(e)}")
```

Upsert values that couldn't make it in the first upsert due to debug/size

This usually happens because vectors were too big. What is recommended is to split the chunks in yet smaller chunks

```
not_upsert = []

with open('combined_output.txt', 'r') as f:
    for line in f:
        if 'ID with too big context (not upserted):' in line:
            not_upsert.append(int(line.split()[-1]))
```

Make a subset of your original dataframe with the rows that were not upserted

```
# Assuming df is your original dataframe and 'ID' is your column
not_upsert_df = df[df['ID'].isin(not_upsert)]

print(f"Length of dataset: {len(not_upsert_df)}")
print(f"""Min num tokens: {min(token_counts)}
Avg num tokens: {int(sum(token_counts) / len(token_counts))}
Max num tokens: {max(token_counts)}""")

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=300,
    chunk_overlap=40,  # number of tokens overlap between chunks
```

```
    length_function = tiktoken_len,
    separators=['.\n', '\t', '\n\n', ';', '...', '. ']
)

# other separaters might be added in order to improve the splitting
```

Then, use the previous steps for post-splitting processing and also the same code block used in the first upsert

Update values

Here, the values to update are those that had a cut on the metadata in the first upsert. It is desired to add the values of 'Column A' where they were initally cut. This is just an example of updating vectors and can be highly costumizable

Create list of indexes to change

```
indexes_to_update = []

#last number for each line is the ID of the main_text
with open('combined_output.txt', 'r') as f:
    if 'Only context was kept as metadata for ID:' in line:
        indexes_to_update.append(int(line.split()[-1]))
```

Create dictionary with changes wanted

```
# You can create a dictionary as follows:
dict_list = []

# Here it is created a dictionary 'ID' : 'number ID' and 'Column A' :
'val
# However, this piece of code can be modified to create any dictionary
or operation you want
for number in indexes_to_update:
    temp_dict = {}
    temp_dict['ID'] = number
    temp_dict['column_a'] = df.loc[df['ID'] == number, 'Column
A'].iloc[0]
    dict_list.append(temp_dict)

# Now dict_list is a list of dictionaries you wanted
print(dict_list)
```

Fetch from database and modify

```
ids = [str(item['ID']) for item in dict_list]
# Fetch the vectors with ids in the list
fetched_vectors = index.fetch(ids=ids)

for item in dict_list:
    if str(item['ID']) in fetched_vectors['vectors']:
```

```
        # Format the fetch data appropriately
        fetched_vectors['vectors'][str(item['ID'])]['metadata']
['nome'] = item['nome']
```

Update

```
k = 0
vectors=[]

for id, vector in tqdm(fetched_vectors['vectors'].items()):
    k+=1
    vector_dict = {
        'id': id,
        'sparse_values': vector['sparse_values'].to_dict(),
        'values': vector['values'],
        'metadata': vector['metadata']
    }
    vectors.append(vector_dict)
    if k % 100 == 0:
        try:
            index.upsert(vectors)
            vectors=[]
        except Exception as e:
            with open('rest_combined_output.txt', 'a') as f:
            # extract the ID of the failed vector from the vector_dict
                f.write(f'Upsert failed for ID {id}\n')
```

Write to json if desired (extra)

```
# Convert FetchResponse to a dictionary
fetched_vectors_dict = {
    id: {
        'id': vector['id'],
        'sparse_values': {'indices': vector['sparse_values'].indices,
'values': vector['sparse_values'].values},
        'values': vector['values'],
        'metadata': vector['metadata']
    }
    for id, vector in fetched_vectors['vectors'].items()
}

# Write fetched_vectors_dict to a JSON file
with open('fetched_vectors.json', 'w') as f:
    json.dump(fetched_vectors_dict, f, indent=4)
```

Querying

Functions used in hybrid query

```
def hybrid_scale(dense, sparse, alpha: float):
    # check alpha value is in range
    if alpha < 0 or alpha > 1:
        raise ValueError("Alpha must be between 0 and 1")
    # scale sparse and dense vectors to create hybrid search vecs
    hsparse = {
        'indices': sparse['indices'],
        'values':  [v * (1 - alpha) for v in sparse['values']]
    }
    hdense = [v * alpha for v in dense]
    return hdense, hsparse

def hybrid_query(question, top_k, alpha, filter=None):
    # convert the question into a sparse vector
    sparse_vec = sparse_encoder.generate_sparse_vectors([question])[0]
    sparse_vec = {
        'indices': list(sparse_vec.keys()),
        'values': [float(value) for value in sparse_vec.values()]
    }

    # convert the question into a dense vector
    dense_vec = embed.encode(question).tolist()

    # scale alpha with hybrid_scale
    dense_vec, sparse_vec = hybrid_scale(
    dense_vec, sparse_vec, alpha
    )
    # query pinecone with the query parameters
    result = index.query(
        vector=dense_vec,
        sparse_vector=sparse_vec,
        top_k=top_k,
        include_metadata=True,
        filter=filter
    )

    # return search results as json
    return result
```

Query example

```
query = 'question_about_secova'
```

Search example

```
hybrid_query(query,
             top_k=3,
             alpha=0.5,
             filter={'date': {'$eq' : '09/13/2018'}}  # filter by date
```

```
if date is in metadata
)
```

Chatbot

```
COHERE_API_KEY = 'your_api_key_here'
co = cohere.Client(COHERE_API_KEY)
```

Code for formating documents

```
class Document:
    def __init__(self, page_content, metadata):
        self.page_content = page_content
        self.metadata = metadata

    def __repr__(self):
        return f"Document(page_content='{self.page_content}',
metadata={self.metadata})"
```

Code for reranking of documents. This is a method for improving Retrieval Augmented Generation (RAG).

```
def rerank(hybrid_scale, hybrid_query, query: str, top_k: int, top_n:
int, alpha: float, filter=None):
    results = hybrid_query(query, top_k=top_k, alpha=alpha,
filter=filter)
    # Filter results
    results_list = [match for match in results['matches']]
    docs_retrieved = []
    for result in results_list:
            doc = Document(result['metadata']['context'],
result['metadata'])
            docs_retrieved.append(doc)

    # Get contexts
    contexts = []
    for doc in docs_retrieved:
        contexts.append(doc.page_content)
    docs = {value: index for index, value in enumerate(contexts,
start=0)}
    i2doc = {docs[doc]: doc for doc in docs.keys()}

    # rerank
    rerank_docs = co.rerank(
        query=query, documents=docs, top_n=top_n, model="rerank-
multilingual-v2.0"
    )

    reranked_is = []
```

```
    for i, doc in enumerate(rerank_docs):
        rerank_i = docs[doc.document["text"]]
        reranked_is.append(rerank_i)
    formated_rerank_docs = []

    for i in reranked_is:
        formated_rerank_docs.append(docs_retrieved[i])

    return formated_rerank_docs
```

Code for checking rerank

```
def compare(hybrid_scale, hybrid_query, query: str, top_k: int, top_n:
int, alpha: float, filter=None):
    results = hybrid_query(query, top_k=top_k, alpha=alpha,
filter=filter)
    # Filter results
    results_list = [match for match in results['matches']]
    docs_retrieved = []
    for result in results_list:
            doc = Document(result['metadata']['context'],
result['metadata'])
            docs_retrieved.append(doc)
    contexts = []
    for doc in docs_retrieved:
        contexts.append(doc.page_content)
    #print(contexts)
    docs = {value: index for index, value in enumerate(contexts,
start=0)}
    i2doc = {docs[doc]: doc for doc in docs.keys()}
    # rerank
    rerank_docs = co.rerank(
        query=query, documents=docs, top_n=top_n, model="rerank-
multilingual-v2.0"
    )
    original_docs = []
    reranked_docs = []
    # compare order change
    for i, doc in enumerate(rerank_docs):
        rerank_i = docs[doc.document["text"]]
        print(str(i)+"\t->\t"+str(rerank_i))
        if i != rerank_i:
            reranked_docs.append(f"[{rerank_i}]\
n"+doc.document["text"])
            if i in i2doc:  # Check if 'i' exists in 'i2doc' before
accessing it
                original_docs.append(f"[{i}]\n"+i2doc[i])
    for orig, rerank in zip(original_docs, reranked_docs):
        print("ORIGINAL:\n"+orig+"\n\nRERANKED:\n"+rerank+"\n\n---\n")
```

Helper fiunction for chatbot

```
class StreamHandler(BaseCallbackHandler):
    def __init__(self, container, initial_text="",
display_method='markdown'):
        self.container = container
        self.text = initial_text
        self.display_method = display_method

    def on_llm_new_token(self, token: str, **kwargs) -> None:
        self.text += token
        display_function = getattr(self.container,
self.display_method, None)
        if display_function is not None:
            # Wrap the text in a div with a custom background color
            colored_text = f'<div style="background-color: #a1b6b7;
padding: 10px; border-radius: 5px;">{self.text}</div>'
            display_function(colored_text, unsafe_allow_html=True)
        else:
            raise ValueError(f"Invalid display_method:
{self.display_method}")
```

Streamlit app

No chat history nor memory

```
def run_chatbot_app():
    # Initialize Pinecone
    @st.cache_resource
    def connect_to_pinecone():
        pinecone.init(api_key="your_api_key",
environment="your_environment")
        return pinecone.Index('secova')
    index = connect_to_pinecone()

    # prompt template where {human_input} is the user query and
{context} is the retrieved document
    template = """Instructions:

                Context:
                {context}

                Human: {human_input}
                """

    prompt = PromptTemplate(
        input_variables=["human_input", "context"],
        template=template
    )
```

```
    # Streamlit app
    st.title('ChatSecova')

    query = st.text_input("Stelle eine frage", key='input')

    if st.button('Suchen') or 'input' in st.session_state:
        if 'input' in st.session_state:
            query = st.session_state.input
        else:
            query = st.session_state.input = ''

        if query:
            query = query.lower()
            # Chatbot
            st.subheader('Antwort:')

            # llm setup
            chat_box = st.empty()
            stream_handler = StreamHandler(chat_box,
display_method='write')
            llm = ChatOpenAI(temperature=0,
openai_api_key=OPENAI_API_KEY, model_name="gpt-3.5-turbo-16k",
                     callbacks=[stream_handler], streaming=True
            )

            # Fetch results with minimum score of relevancy
            results = hybrid_query(query, top_k=10, alpha=1)
            #filter by score (when alpha=1 score goes from 0 to 1)
            filtered_list = [match for match in results['matches'] if
match['score'] > 0.8]
            combined_docs_chain = []
            for result in filtered_list:
                doc = Document(result['metadata']['context'],
result['metadata'])
                combined_docs_chain.append(doc)

            # Create the chain
            chain = load_qa_chain(llm, chain_type="stuff",
prompt=prompt)
            response = chain.run(input_documents=combined_docs_chain,
question=query, human_input=query)

            # Display reference expanders
            for doc in combined_docs_chain:
                content = doc.page_content
                nome = doc.metadata['nome']
                with st.expander(f"{nome}"):
                    st.markdown(content)
```

Chatbot with chat history

```
def run_chatbot_app():
    # Initialize Pinecone
    @st.cache_resource
    def connect_to_pinecone():
        pinecone.init(api_key="your_api_key",
environment="your_environment")
        return pinecone.Index('secova')
    index = connect_to_pinecone()

    # prompt template where {human_input} is the user query and
{context} is the retrieved document
    template = """Instructions:

                Context:
                {context}

                Human: {human_input}
                """

    prompt = PromptTemplate(
        input_variables=["human_input", "context"],
        template=template
    )

    st.title('ChatSecova')

    # Set a default model
    if "openai_model" not in st.session_state:
        st.session_state["openai_model"] = "gpt-3.5-turbo-16k"

    # Initialize chat history
    if "messages" not in st.session_state:
        st.session_state.messages = []

    if "references" not in st.session_state:
        st.session_state.references = []

    # Initialize subheaders
    if "subheaders" not in st.session_state:
        st.session_state.subheaders = []

    # Display messages and subheaders
    for message, subheader in zip(st.session_state.messages,
st.session_state.subheaders):
        st.markdown(subheader)
        with st.chat_message(message["role"]):
            if message['role'] == 'user':
                st.markdown(message['content'])
            elif message['role'] == 'assistant':
```

```
                colored_text = f'<div style="background-color:
#a1b6b7; padding: 10px; border-radius:
5px;">{message["content"]}</div>'
                st.markdown(colored_text, unsafe_allow_html=True)

    for referece in st.session_state.references:
        with st.expander(referece['nome']):
            st.markdown(referece['content'])

    st.markdown('### Frage:')
    if query := st.chat_input("Stelle eine frage"):
        st.session_state.subheaders.append('#### Frage:')
        # Add user message to chat history
        st.session_state.messages.append({"role": "user", "content":
query})
        with st.chat_message("user"):
            st.markdown(query)

        st.markdown('### Antwort:')
        with st.chat_message("assistant"):
            st.session_state.subheaders.append('#### Antwort:')
            # Fetch results with minimum score of relevancy
            results = hybrid_query(query, top_k=5, alpha=0.5)
            #when alpha=0.5 the score result goes from 0 to approx 40
            filtered_list = [match for match in results['matches'] if
match['score'] > 30]
            combined_docs_chain = []
            for result in filtered_list:
                doc = Document(result['metadata']['context'],
result['metadata'])
                combined_docs_chain.append(doc)

            query = query.lower()

            # llm setup
            model=st.session_state["openai_model"]
            chat_box = st.empty()
            stream_handler = StreamHandler(chat_box,
display_method='write')
            llm = ChatOpenAI(temperature=0,
openai_api_key=OPENAI_API_KEY, model_name=model,
                     callbacks=[stream_handler], streaming=True
            )

            # Create the chain
            chain = load_qa_chain(llm, chain_type="stuff",
prompt=prompt)
            response = chain.run(input_documents=combined_docs_chain,
question=query, human_input=query)
```

```
            st.session_state.messages.append({"role": "assistant",
"content": response})

            # Display reference expanders
            # Assuming there is a metadata with the name of a certain
document/article
            for doc in combined_docs_chain:
                content = doc.page_content
                name = doc.metadata['name']
                with st.expander(f"{name}"):
                    st.markdown(content)
                st.session_state.references.append({"name": name,
"content": content})
```

Chatbot with conversational memory

```
def run_chatbot_app():
    # Initialize Pinecone
    @st.cache_resource
    def connect_to_pinecone():
        pinecone.init(api_key="your_api_key",
environment="your_environment")
        return pinecone.Index('secova')
    index = connect_to_pinecone()

    # retriever
    hs_retriever = PineconeHybridSearchRetriever(
        embeddings=embed, sparse_encoder=sparse_encoder, index=index,
top_k=4, alpha=0.5
    )

    # conversational memory
    conv_mem = ConversationBufferWindowMemory(
        memory_key = 'chat_history',
        k = 5,  # number of previous turns to remember
        return_messages = True
    )

    # prompt template where {human_input} is the user query and
{context} is the retrieved document
    template = """Instructions:

                Context:
                {context}

                Human: {human_input}
                """

    prompt = PromptTemplate(
        input_variables=["human_input", "context"],
```

```
        template=template
    )

    # Streamlit app
    st.title('ChatSecova')

    # Input for the user to enter their query
    query = st.text_input('Stelle eine frage')

    if st.button('Suchen'):
        # Chatbot
        st.subheader('Antwort:')
        # llm setup
        chat_box = st.empty()
        stream_handler = StreamHandler(chat_box,
display_method='write')
        llm = ChatOpenAI(temperature=0, openai_api_key=OPENAI_API_KEY,
model_name="gpt-3.5-turbo-16k",
                callbacks=[stream_handler], streaming=True
        )

        # Create the chain
        chain = ConversationalRetrievalChain.from_llm(
            llm=llm,
            retriever=hs_retriever,
            chain_type="stuff",
            condense_question_prompt=prompt,
            memory=conv_mem
        )

        # Call the chain with the query
        output = chain({"question": query})

        # Get the chatbot's response
        response = output['answer']


combine_docs_chain=PineconeHybridSearchRetriever._get_relevant_documen
ts(
            self=hs_retriever,
            run_manager=CallbackManagerForRetrieverRun,
            query=query
        )

        # Display reference expanders
        for doc in combine_docs_chain:
            content = doc.page_content
            nome = doc.metadata['name']
            with st.expander(f"{name}"):
                st.write(content)
```

Possibility: make with chat history and chat memory