IR_V2_w_Hybrid_Search

October 8, 2024

1 IR project v2 with (Optional) Hybrid Search

Overview of the Project -Environment Setup

-Data Loading and Preprocessing

-Generating Embeddings with Vertex AI

-Building the Vector Store with FAISS

-Implementing the Information Retrieval System

-Example Query and Retrieval

1.0.1 1. Environment Setup

Install Required Libraries Ensure you have the necessary libraries installed:

[ ]:
```
!pip install google-cloud-storage
!pip install faiss-cpu
!pip install PyPDF2
!pip install pandas
!pip install scikit-learn  # For TfidfVectorizer
!pip install langchain
```

Import necessary libraries:

[4]:
```
import os
import re
import pandas as pd
from google.cloud import storage
from PyPDF2 import PdfReader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.schema import Document
from sklearn.feature_extraction.text import TfidfVectorizer
import faiss
import numpy as np
```

1.0.2 2. Data Loading and Preprocessing

Download PDF from Google Cloud Storage We start by downloading the PDF file from Google Cloud Storage.

[5]:
```
def download_blob(bucket_name, source_blob_name, destination_file_name):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(source_blob_name)
    try:
        blob.download_to_filename(destination_file_name)
        print(f"Downloaded {source_blob_name} to {destination_file_name}.")
    except Exception as e:
        print(f"Error downloading {source_blob_name}: {e}")
```

Set your bucket and file names:

[6]:
```
bucket_name = "benefit_books_3"
source_blob_name = "WF_benefits_book.pdf"
destination_file_name = "/tmp/WF_benefits_book.pdf"

download_blob(bucket_name, source_blob_name, destination_file_name)
```

```
Downloaded WF_benefits_book.pdf to /tmp/WF_benefits_book.pdf.
```

Extract and Preprocess Text Next, we extract text from the PDF and preprocess it.

[7]:
```
def preprocess_pdf(pdf_path):
    text = ""
    with open(pdf_path, "rb") as file:
        reader = PdfReader(file)
        if reader.is_encrypted:
            try:
                reader.decrypt("")
            except:
                raise ValueError("Failed to decrypt PDF file.")
        for page in reader.pages:
            page_text = page.extract_text()
            if page_text:
                text += page_text
    return text

def preprocess_text(text):
    # Clean up the text
    text = re.sub(r'\s+', ' ', text).strip()
    return text

raw_text = preprocess_pdf(destination_file_name)
clean_text = preprocess_text(raw_text)
```

Split Text into Chunks We split the text into manageable chunks for embedding.

[8]:
```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,␣
↪chunk_overlap=200)
```

```
chunks = text_splitter.split_text(clean_text)
```

Create Document Objects

[9]:
```
documents = [Document(page_content=chunk) for chunk in chunks]
```

1.0.3 3. Generating Embeddings with Vertex AI

Initialize the Embedding Model

Note: If you are running this in an environment that doesn't require explicit authentication (like Colab with a logged-in account), you don't need to set up GOOGLE_APPLICATION_CREDENTIALS or initialize aiplatform.

[10]:
```
from vertexai.preview.language_models import TextEmbeddingModel

# Initialize the embedding model
embedding_model = TextEmbeddingModel.from_pretrained("textembedding-gecko@001")
```

Generate Dense Embeddings

[12]:
```
def get_dense_embeddings(texts):
    embeddings = []
    batch_size = 5  # Adjust based on API limits
    for i in range(0, len(texts), batch_size):
        batch_texts = texts[i:i+batch_size]
        response = embedding_model.get_embeddings(batch_texts)
        embeddings.extend([embedding.values for embedding in response])
    return embeddings

texts = [doc.page_content for doc in documents]
dense_embeddings = get_dense_embeddings(texts)
```

Generate Sparse Embeddings (Optional) Sparse embeddings can be useful for hybrid search. https://github.com/GoogleCloudPlatform/generative-ai/blob/main/embeddings/hybrid-search.ipynb

[13]:
```
vectorizer = TfidfVectorizer()
vectorizer.fit(texts)

def get_sparse_embedding(text):
    tfidf_vector = vectorizer.transform([text])
    return {
        "values": tfidf_vector.data.tolist(),
        "dimensions": tfidf_vector.indices.tolist()
    }

sparse_embeddings = [get_sparse_embedding(text) for text in texts]
```

1.0.4 4. Building the Vector Store with FAISS

Initialize FAISS Index

[14]:
```
embedding_dim = len(dense_embeddings[0])  # Should be 768 for␣
↪'textembedding-gecko'
index = faiss.IndexFlatL2(embedding_dim)
```

Add Embeddings to Index

[15]:
```
embedding_matrix = np.array(dense_embeddings).astype('float32')
index.add(embedding_matrix)
```

Save the Index and Documents

[16]:
```
faiss.write_index(index, 'faiss_index.index')

# Save documents and sparse embeddings
df = pd.DataFrame({
    'content': texts,
    'sparse_embedding_values': [emb['values'] for emb in sparse_embeddings],
    'sparse_embedding_dims': [emb['dimensions'] for emb in sparse_embeddings]
})
df.to_csv('documents.csv', index=False)
```

1.0.5 5. Implementing the Information Retrieval System

Query Function

[18]:
```
def query_index(query, k=5):
    # Generate dense embedding for the query
    query_embedding = embedding_model.get_embeddings([query])[0].values
    query_vector = np.array([query_embedding]).astype('float32')

    # Generate sparse embedding for the query (optional)
    query_sparse_embedding = get_sparse_embedding(query)

    # Search the FAISS index
    distances, indices = index.search(query_vector, k)

    # Retrieve the corresponding documents
    results = []
    for idx in indices[0]:
        idx = int(idx)
        content = df.iloc[idx]['content']
        distance = distances[0][list(indices[0]).index(idx)]
        results.append({
            'content': content,
            'distance': distance,
```

```
            'sparse_embedding': {
                'values': df.iloc[idx]['sparse_embedding_values'],
                'dimensions': df.iloc[idx]['sparse_embedding_dims']
            }
        })
    return results
```

[19]:
```
query = "What dental benefits are available?"
results = query_index(query, k=5)
```

[20]:
```
for i, result in enumerate(results):
    print(f"Result {i+1}:")
    print(f"Content: {result['content'][:200]}...")  # Show first 200 characters
    print(f"Distance: {result['distance']}\n")
```

```
Result 1:
Content: 3-4 Cost 3-4 How the Delta Dental coverage options work 3-4
Pretreatment estimate 3-4 What the Delta Dental coverage options cover 3-6 Your
dental benefits and costs at a glance 3-6 Frequency limits 3…
Distance: 0.4681614339351654

Result 2:
Content: option 2 Employee Care accepts all relay service calls, including 711.
Information about premiums HR Services & Support site Chapter 3: Dental Plan 3-2
The information in this chapter - along with app…
Distance: 0.5025160312652588

Result 3:
Content: eligible covered services and the maximum benefits payable under the
plan. "What the Delta Dental coverage options cover" section starting on page
3-6 for detailed benefits and coverage information. Y…
Distance: 0.5220946073532104

Result 4:
Content: and when coverage begins 3-3 Changing or canceling coverage 3-3 When
coverage ends 3-4 Cost 3-4 How the Delta Dental coverage options work 3-4
Pretreatment estimate 3-4 What the Delta Dental coverage …
Distance: 0.5399929285049438

Result 5:
Content: Wachovia Dental Program. The total lifetime orthodontia benefits paid
per person, combined with any other orthodontia benefits paid under the Wells
Fargo dental plan or the former Wachovia Dental Prog…
Distance: 0.5410927534103394
```

Conclusion In this lecture, we've built an IR system that:

Processes a PDF from Google Cloud Storage. Splits the text into chunks and preprocesses it. Generates embeddings using Vertex AI's TextEmbeddingModel. Builds a FAISS index for efficient similarity search. Implements a query function to retrieve relevant text based on user input. This system can be extended and integrated into applications where efficient text retrieval is needed.

[ ]: