HW3 RAG

學號:313512072

姓名:洪亮

一、方法:

1. 資料集前處理:

```
nlp_model = spacy.load("en_core_web_sm")
def dataset_processor(demo_full_text):
    # Split the full text into sections
    sections = demo_full_text.split("\n\n\n")[:-1]

    merged_sections = []
    for section in sections:
        combined_text = " ".join(section.split("\n")).strip()
        merged_sections.append(combined_text)

    documents = [Document(page_content = doc) for doc in merged_sections]

    sentence_docs = []
    for doc in documents:
        spacy_doc = nlp_model(doc.page_content)
        # Filter out short sentences
        sentences = [sent.text.strip() for sent in spacy_doc.sents if len(sent.text.strip()) >= 40]
        sentence_docs.append(Document(page_content = " ".join(sentences)))
```

- **原始文字分段:**

  以 \n\n\n 為分隔符號，將整體文字切成多個段落（sections）。
  移除最後一段空內容，避免處理空字串。

- **合併與清理段落內容:**

  將每段內部的 \n 換行符轉為空格，使段落成為一串連續文字。
  去除前後空白，建立 Document 物件列表。

- **句子拆解與過濾:**

  對每個段落進行句子分割，過濾掉少於 40 字的短句，再次組合為長文本。

2. 檢索方法:

```python
def build_vector_store(document):
    # Preprocess and chunk the document
    docs_splits = (dataset_processor(document))
    # Filter out chunks
    valid_documents = [
        doc for doc in docs_splits
        if len(doc.page_content.strip()) >= 80 and len(doc.page_content.strip().split()) >= 12
    ]

    embeddings = OllamaEmbeddings(
      model="mxbai-embed-large",
      keep_alive=3000,
    )
    vector_store = InMemoryVectorStore.from_documents(valid_documents, embeddings)
    return vector_store
```

- **過濾 chunk:**

  在向量化前，篩掉太短的 chunk（少於 80 個字元或少於 12 個詞），以保證每個向量都有足夠資訊。

- **向量化 chunk:**

  使用 OllamaEmbeddings 將處理後的 chunk 文本轉換為向量。

  Embedding Model：mxbai-embed-large。

  keep_alive=3000：保留模型連線 3000 秒，以減少重複啟動延遲。

- **建立向量資料庫:**

  使用 InMemoryVectorStore.from_documents()，將所有有效 chunk 及其對應向量加載至記憶體中的向量資料庫，供後續檢索使用。

```python
def construct_retrieval_chain(llm, vector_store):

    SYSTEM_PROMPT: str = """You are a helpful assistant.
    Answer the question based on the context below.

    Make sure your answer:
    - is concise and based on facts from the context
    - does not make assumptions beyond the given information
    - does not include incomplete or fragmented sentences


    Answer in a complete sentence. Do not repeat the question.
    """
    CHAT_TEMPLATE_RAG = (
    f"""system: {SYSTEM_PROMPT}
    human: context: {{context}}\nquestion: {{input}}
    assistant: """
    )
    retrieval_qa_prompt = PromptTemplate.from_template(template=CHAT_TEMPLATE_RAG)
    combine_docs_chain = create_stuff_documents_chain(llm, retrieval_qa_prompt)
    rag_qa_chain = create_retrieval_chain(
      retriever=vector_store.as_retriever(
          search_kwargs={"k":5, "fetch_k": 15, "lambda_mult": 0.9},
        search_type="mmr"

        ),
      combine_docs_chain=combine_docs_chain
    )
    return rag_qa_chain
```

- **設定檢索參數（Retriever）**

參數說明：

search_type	用來指定使用的檢索策略
K	決定最終要提供給模型的 context 數量
fetch_k	從向量資料庫中抓出的候選資料數量
lambda_mult	控制相關性與多樣性的平衡，取值範圍 0 ~ 1，越接近 1 表示越重視相關性，越接近 0 則偏向多樣性

1.採用 MMR（Maximal Marginal Relevance） 演算法，每次挑選 context 時，不僅看它與問題的相似度，也會考慮與已選 context 之間的差異性，避免選出類似的段落。

2.fetch_k=15 和 k=5 是從資料庫抓出 15 筆候選。最後選出 5 筆 context 作為回答依據，k 如果設太小可能導致資訊可能不足，太大可能導致無關資訊干擾模型判斷。

3.lambda_mult = 0.9：偏好高相關性，這樣可確保選出的段落聚焦在回答問題上，而不是太過發散。

- **構建檢索問答鏈（RAG Chain）**

  使用 create_retrieval_chain()，將檢索到的相關文本作為上下文提供給 LLM 回答問題。

3.Prompt 技巧:

```
SYSTEM_PROMPT: str = """You are a helpful assistant.
Answer the question based on the context below.

Make sure your answer:
- is concise and based on facts from the context
- does not make assumptions beyond the given information
- does not include incomplete or fragmented sentences


Answer in a complete sentence. Do not repeat the question.
"""
CHAT_TEMPLATE_RAG = (
f"""system: {SYSTEM_PROMPT}
human: context: {{context}}\nquestion: {{input}}
assistant: """
)
```

- **清楚定義系統角色與回答規範**

  使用 SYSTEM_PROMPT 明確設定語言模型在對話中的角色是 helpful assistant，讓模型理解該如何回應提問。

  並加入明確的回答規則，包括：

回答必須簡潔明確、基於提供的 context，不可亂猜或憑空推測
禁止輸出不完整的句子，必須使用完整句子回答
要求回答時不要重複問題

- **結構化提示模板（Prompt Template）**

  建立一個結構化的 CHAT_TEMPLATE_RAG，分為以下部分:

system	輸入 SYSTEM_PROMPT（告訴模型角色與規則）
human	包含 context（由檢索結果拼接而成的相關文本）與 question（實際使用者問題）
assistant	語言模型將根據前面兩者進行作答。

- **與檢索鏈結合的設計（RAG）**
  - 將檢索到的內容嵌入 prompt 的 context 區塊。
  - 引導 LLM 僅依據上下文作答。

二、研究與實驗:

1. Split&Chunk:

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 512, # number of characters
    chunk_overlap = 256,
    separators=["\n\n", "\n", ".", "。", "!", "?", " ", ""],
    length_function = len,
    add_start_index = True
)
```

- **使用 RecursiveCharacterTextSplitter 進行分塊，以下是我嘗試過的配置：**

chunk_size	512	512	512
chunk_overlap	128	256	256
分塊方式	根據("\n\n", "\n", " ", "")分塊	根據("\n\n", "\n", " ", "")分塊	根據 ("\n\n", "\n", ".", "。", "!", "?", " ", "") 分塊

- **結論:**

  chunk_overlap：256 字元，效果比 chunk_overlap：128 字元更好，因為 chunk_overlap：256 相較 128 可提高保留上下文銜接資訊，避免關鍵資訊落在 chunk 邊界而被截斷。

  分塊方式採用遞進式分隔符（"\n\n", "\n", ".", "。", "!", "?", " ", ""），優先使用自然語言的語句邊界（段落、句點、問號等）來分段，若無法滿足長度需求，才以空白或任意位置切割。可確保每個 chunk 在長度適中的同時，盡可能保留語意完整與連貫性。

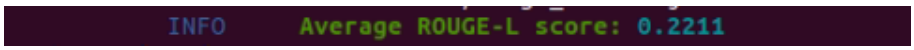
2. 額外的技巧:

我有額外使用 spaCy 的 en_core_web_sm 模型做為前處理的輔助工具。主要是用它來進行句子斷句，幫我更精準地把每段文字切成一個個語意完整的句子。這樣做的好處是，可以避免用單純的標點符號或換行去切句時可能出現的錯切問題，像是縮寫或長句被拆錯。另外我還加了一個條件，只保留長度超過 40 個字的句子，這樣可以過濾掉沒什麼資訊的短句，讓後面進行 chunk 切割的時候，每個 chunk 的內容都比較有資訊量，也比較容易被語言模型理解。

3. 紀錄改進結果細節:

chunk_size	chunk_overlap	分塊方式	ROUGE-L score
512	128	("\n\n", "\n", " ", "")	0.2069
512	256	("\n\n", "\n", " ", "")	0.2198
512	256	("\n\n", "\n", ".", "。", "!", "?", " ", "")	0.2211

以下為實驗結果



四、學習心得

上這堂課以前我都覺得，問 AI 什麼它都應該能正確回答，但上完這堂課後才發現，當 AI 遇到它沒學過的資訊時，其實很容易亂講。這次的作業讓我認識了 RAG（Retrieval-Augmented Generation），我覺得這個方法真的很有幫助。簡單來說，它會讓語言模型在回答之前，先從資料庫中找出相關內容，再根據這些內容來生成回答。這樣不但能讓回答更準確，也比較不會出現瞎掰的情況。

我覺得這種流程就像是我們在回答問題之前，會先去查資料，再來回答。因為語言模型本身不是萬能的，遇到較新的資訊時，還是會說錯。但透過 RAG，讓模型講的話更有根據，也更值得信任。