# 【Day 6】 FAISS 資料庫與文本分割

## 1. 向量資料庫

### 1.1 FAISS

FAISS 是一個向量資料庫，用來儲存 embeddings（就是一些向量）。儲存的過程中，FAISS 可以採用一些特殊的演算法，例如 **IVF (Inverted File List)**、**PQ (Product Quantization)**、**HNSW (Hierarchical Navigable Small World)** 等等，來提升檢索的效率與擴展性。

以下範例展示如何在 LangChain 中建立一個 FAISS 向量資料庫：

```
import faiss

from langchain_google_genai import GoogleGenerativeAIEmbeddings
from langchain_community.docstore.in_memory import InMemoryDocstore
from langchain_community.vectorstores import FAISS
```

```
embeddings = GoogleGenerativeAIEmbeddings(model="models/gemini-embedding-001")

index = faiss.IndexFlatL2(len(embeddings.embed_query("test")))  # faiss.IndexIVF, faiss.IndexIVFPQ

vector_store = FAISS(
    embedding_function=embeddings,
    index=index,
    docstore=InMemoryDocstore(),
    index_to_docstore_id={},
)
```

### 1.2 Embedding

在語言模型與檢索任務中，**Embedding** 是將文字、圖片等資料轉換成向量的一種方式。

- **文字 → 向量**：例如句子 `"Hello World"` → `[0.12, -0.54, ..., 0.88]`
- **假設**：語意相近的文字，向量空間裡的距離也相近。

舉例：

- `"I like football"` 與 `"I enjoy soccer"` 的向量會比較接近。
- `"I like football"` 與 `"The stock market crashed"` 的向量則會距離很遠。

我們可以調用 `embeddings.embed_query("文字")` 拿到向量。

在 RAG 中，Retriever 會利用 Embedding 以及向量索引來找出最相關的文件。其中，**Embedding** 的品質會直接影響檢索效果與最終的 RAG 效果。在後續論文討論的部分，我們也會花不少篇幅介紹不同的 Embedding 方法與檢索策略。

## 1.3 Approximate Nearest Neighbor Search

當資料量變大時，直接對比所有向量（Brute Force）會非常耗時。

**ANN** 提供了一個折衷方案：犧牲一點點精確度，換取查詢效率提升大幅提升。

在 FAISS 中，可以建立：

```
faiss.IndexIVFFlat(...)
faiss.IndexIVFPQ(...)
faiss.IndexHNSWFlat(...)
```

## 1.4 LangChain FAISS 的安全隱患

在 FAISS 向量資料庫存在本地之後，會有兩個檔案

- `index.faiss` → 儲存向量索引
- `index.pkl` → 透過 `pickle` 儲存 Document 的 metadata 與內容

而在讀取 `pickle` 檔案時，有可能同時執行惡意的程式碼（python object 預留 `__reduce__` 方法，可以自定義 `unpickle` 時的行為）。因此建議僅讀取自己生成的檔案，不要載入來路不明的 `index.pkl`，或使用安全的序列化方式（例如 JSON 等）。

載入資料庫的時候，需要設置 `allow_dangerous_deserialization` 參數，提醒使用者注意。

```
# 危險操作：請只在自己可信任的檔案中使用

FAISS.load_local("vector_store", embeddings, allow_dangerous_deserialization=True)
```

# 2. Chunking

可供檢索的文本以 `chunk` 為單位，常見的策略包含 LangChain 中的 `RecursiveCharacterTextSplitter` 和 `CharacterTextSplitter` 。有時也可以利用資料本身的架構、層次來分塊，這樣更能保留語意完整度。我會利用範例資料集來練習基本的 chunking 方法。

```
from langchain_text_splitters import RecursiveCharacterTextSplitter, CharacterTextSplitter
from langchain.docstore.document import Document
```

## 2.1 `.Docx` & `.txt`

`CharacterTextSplitter` 會簡單的用給定的字符來把文字段落分開，這裡用 `"\n"` ，因為在資料庫中，`.Docx` 和 `.txt` 檔案是課堂筆記、工作經歷、履歷等簡單、段落式的文字敘述。

我們還可以指定 `chunk_size` 和 `chunk_overlap` 來控制 chunk 的大小與重疊（為了保持上下文關係）。這裡的 `chunk_size` 算的是字符數量，我們設置為 `256` 。

```
def get_docx_and_txt_chunks(text: str) -> list:
    splitter = CharacterTextSplitter(chunk_size=256, chunk_overlap=128, separator="\n")
    txt_chunks = splitter.split_text(text)
    return txt_chunks
```

## 2.2 `.md`

在資料庫裡面，有 `info_self_introduction.md` 這個檔案包含模擬人物的自我介紹，用經典的 markdown 架構撰寫。我們可以把 markdown 的 header 結構 （`#`、`##`、`###`）轉換為 `---h3---` 等標籤，接著再套用 `RecursiveCharacterTextSplitter` ，它會**遞回**式的依照給定的分隔符來分割文字，直到每一個 chunk 都小於 `chunk_size` 。

```
def get_md_chunks(text: str) -> list:
    text = text.replace("###", "---h3---").replace("##", "---h2---").replace("#", "---h1---").replace("\n\n", "\n")
    recursive_splitter_md = RecursiveCharacterTextSplitter(separators=["---h2---", "---h3---", "\n"], chunk_size=256)
    md_chunks = recursive_splitter_md.split_text(text)
    return md_chunks
```

## 2.3 .pdf

在**個人助手**的例子裡，`.pdf` 檔包含海報（poster）式的專案作品集，其中包含「情境」、「任務」、「行動」等子標籤，單純將文字取出並沒有辦法保有「子標題：內容」的語意結構，所以讓我們試著用一個 LLM 來整理亂中有序的文字檔：

```
from langchain_google_genai import GoogleGenerativeAI
from langchain.prompts import PromptTemplate
prompt_template = PromptTemplate.from_template("""
    請整理以下文字，用專案主題：<專案主題>\n**標題：<標題>內容：<內容>**\n\n**標題：<標題>內容：<內容>**\n\n ..... 的格式回傳。{pdf_text}""")
pdf_arranger = prompt_template | GoogleGenerativeAI(model="gemini-2.5-flash-lite")

def get_pdf_chunks(text: str) -> list:
    formatted_text = pdf_arranger.invoke({"pdf_text": text})
    splitter = RecursiveCharacterTextSplitter(separators=["\n\n", "\n"], chunk_size=256, chunk_overlap=128)
    pdf_chunks = splitter.split_text(formatted_text)
    return pdf_chunks
```

## 2.4 .json

最後，我們的資料中有一個**常見問答集**，像是：

```
[
  {
    "question": ...,
    "answer": ...,
```

```
    "category": ...
  },
]
```

因為剛好 QA 的長度都不長，所以實作上目前一個 QA pair （raw json string）就是一個 chunk。這當然有不少進步空間：

- 整理 json 字串，例如

> **question**: ...
> **answer**: ...
> **category**: ...

- 保留題目，只 Chunk 答案，例如：
  - 答案較長，將它分為 A1, A2, A3
  - chunk 為 Q+A1, Q+A2 和 Q+A3
  - 如此 chunk 有類似的長度，也保留 QA 的情境

# 3. vector store 操作

## 3.1 Documents

將每一個 chunk 包裝成 LangChain 的 `Document` 這樣可以整合 **metadata** 的訊息，我加入了 `source`、`chunk_index` 和 `last_update` 三個屬性，這樣方便以後更新資料庫。

```
# docs
Document(
    page_content=chunk,
        metadata={
     "source": file,
     "chunk_index": i,
     "last_update": time.time(),
     }
    )

# ids
f"{file}_{i}"  # i is chunk index
```

## 3.2 Add, save & delete

在處理完所有的 documents 之後，只要使用下方的代碼就可以將他們加入到 vector store 中然後儲存。有妥善的 **id** 跟 **metadata** 管理可以幫助我們在需要刪除或更新文件的時後更方便。

```
vector_store.add_documents(docs, ids=ids)
vector_store.save_local("../vector_store")
# vector_store.delete(ids=[...])
```