# LLM Apps

## RAG Technique: Advanced Concepts

# Splitters: advanced concepts

- CharacterTextSplitter
  - Uses a criterion to split the text into fragments.
  - For example, the newline character: “/n”

- RecursiveCharacterTextSplitter
  - Uses one or several criteria to split the text into fragments.
  - For example, paragraph break and line break: “/n/n” and “/n”
  - This means it will first separate the fragments following the paragraph break criterion and then, if any of the fragments is larger than the established size limit, it will separate the fragments exceeding the maximum size using the second criterion (line break).

# RetrievalQA chain: advanced concepts

- chain_type = Stuff vs MapReduce:
  - Stuff simply aggregates all results of the semantic search, converts them into a prompt, and with that prompt makes a single call to the LLM.
  - The problem with Stuff arises when the aggregated result exceeds the context window limit.
  - MapReduce makes a call to the LLM for each result of the semantic search. When it has all the answers, it makes a new call to the LLM and asks it to design a single final answer considering all the previous responses.
  - The problem with MapReduce is that by making more calls to the LLM, it is more expensive than Stuff.
- return_source_documents = True
  - The answer will include as metadata the sources used to compose it.
- chain_type_kwargs = {“prompt”: prompt_template}
  - To associate a prompt template that, for example, tells chatGPT to respond based on the context and if it doesn't find an answer in the document to respond “I don't know”.

# Vector databases: advanced concepts (1)

- Creating a database associated with a private document each time the app is run is unnecessary if the private document has not changed. Besides, it is time-consuming and costly. It's better to save it in a file and create a function that creates it only at the beginning.

```
embedding = OpenAIEmbedding()

def create_vector_db():
    loader = CSVLoader(…)
    documents = loader.load()
    vectordb = FAISS.from_documents(documents, embedding)
    vectordb.save_local(“my_vector_database”)

vectordb_file_path = “my_vector_database”
```

# Vector databases: advanced concepts (2)

- At the end of the file, we add this code that will create the database if the file is being run as the main program and not being imported as a module in another script.

```
if __name__ == “__main__”:
    create_vector_db()
```

- Now, when creating the chain, we just have to retrieve the saved database:

```
vectordb = FAISS.load_local(vectordb_file_path, embedding)
```

# Other interesting tips

- langchain.debug = True