Retrieval

Retrieval is the centerpiece of our retrieval augmented generation (RAG) flow.

Let's get our vectorDB from before.

Vectorstore retrieval

In [ ]:
```
import os
import openai
import sys
sys.path.append('../..')

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file

openai.api_key  = os.environ['OPENAI_API_KEY']
```

In [ ]:
```
#!pip install lark
```

Similarity Search

In [ ]:
```
from langchain.vectorstores import Chroma
from langchain.embeddings.openai import OpenAIEmbeddings
persist_directory = 'docs/chroma/'
```

In [ ]:
```
embedding = OpenAIEmbeddings()
vectordb = Chroma(
    persist_directory=persist_directory,
    embedding_function=embedding
)
```

In [ ]:
```
print(vectordb._collection.count())
```

In [ ]:
```
texts = [
    """The Amanita phalloides has a large and imposing epigeous (aboveground
    """A mushroom with a large fruiting body is the Amanita phalloides. Some
    """A. phalloides, a.k.a Death Cap, is one of the most poisonous of all k
]
```

In [ ]:
```
smalldb = Chroma.from_texts(texts, embedding=embedding)
```

In [ ]:
```
question = "Tell me about all-white mushrooms with large fruiting bodies"
```

In [ ]:
```
smalldb.similarity_search(question, k=2)
```

In [ ]:
```
smalldb.max_marginal_relevance_search(question,k=2, fetch_k=3)
```

Addressing Diversity: Maximum marginal relevance

Last class we introduced one problem: how to enforce diversity in the search results.

`Maximum marginal relevance` strives to achieve both relevance to the query *and diversity* among the results.

In [ ]:
```
question = "what did they say about matlab?"
docs_ss = vectordb.similarity_search(question,k=3)
```

In [ ]:
```
docs_ss[0].page_content[:100]
```

In [ ]:
```
docs_ss[1].page_content[:100]
```

Note the difference in results with `MMR`.

In [ ]:
```
docs_mmr = vectordb.max_marginal_relevance_search(question,k=3)
```

In [ ]:
```
docs_mmr[0].page_content[:100]
```

In [ ]:
```
docs_mmr[1].page_content[:100]
```

Addressing Specificity: working with metadata

In last lecture, we showed that a question about the third lecture can include results from other lectures as well

To address this, many vectorstores support operations on `metadata`.

`metadata` provides context for each embedded chunk.

In [ ]:
```
question = "what did they say about regression in the third lecture?"
```

In [ ]:
```
docs = vectordb.similarity_search(
    question,
    k=3,
    filter={"source":"docs/cs229_lectures/MachineLearning-Lecture03.pdf"}
)
```

In [ ]:
```
for d in docs:
    print(d.metadata)
```

In [ ]:

```

```

Addressing Specificity: working with metadata using self-query retriever

But we have an interesting challenge: we often want to infer the metadata from the query itself.

To address this, we can use `SelfQueryRetriever`, which uses an LLM to extract:

1. The `query` string to use for vector search
2. A metadata filter to pass in as well

Most vector databases support metadata filters, so this doesn't require any new databases or indexes.

In [ ]:

```
from langchain.llms import OpenAI
from langchain.retrievers.self_query.base import SelfQueryRetriever
from langchain.chains.query_constructor.base import AttributeInfo
```

In [ ]:

```
metadata_field_info = [
    AttributeInfo(
        name="source",
        description="The lecture the chunk is from, should be one of `docs/c
        type="string",
    ),
    AttributeInfo(
        name="page",
        description="The page from the lecture",
        type="integer",
    ),
]
```

In [ ]:

```
document_content_description = "Lecture notes"
llm = OpenAI(temperature=0)
retriever = SelfQueryRetriever.from_llm(
    llm,
    vectordb,
    document_content_description,
    metadata_field_info,
    verbose=True
)
```

In [ ]:

```
question = "what did they say about regression in the third lecture?"
```

You will receive a warning about predict_and_parse being deprecated the first time you executing the next line. This can be safely ignored.

In [ ]:

```
docs = retriever.get_relevant_documents(question)
```

In [ ]:
```
for d in docs:
    print(d.metadata)
```

Additional tricks: compression

Another approach for improving the quality of retrieved docs is compression.

Information most relevant to a query may be buried in a document with a lot of irrelevant text.

Passing that full document through your application can lead to more expensive LLM calls and poorer responses.

Contextual compression is meant to fix this.

In [ ]:
```
from langchain.retrievers import ContextualCompressionRetriever
from langchain.retrievers.document_compressors import LLMChainExtractor
```

In [ ]:
```
def pretty_print_docs(docs):
    print(f"\n{'-' * 100}\n".join([f"Document {i+1}:\n\n" + d.page_content f
```

In [ ]:
```
# Wrap our vectorstore
llm = OpenAI(temperature=0)
compressor = LLMChainExtractor.from_llm(llm)
```

In [ ]:
```
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=vectordb.as_retriever()
)
```

In [ ]:
```
question = "what did they say about matlab?"
compressed_docs = compression_retriever.get_relevant_documents(question)
pretty_print_docs(compressed_docs)
```

Combining various techniques

In [ ]:
```
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=vectordb.as_retriever(search_type = "mmr")
)
```

In [ ]:
```
question = "what did they say about matlab?"
compressed_docs = compression_retriever.get_relevant_documents(question)
pretty_print_docs(compressed_docs)
```

Other types of retrieval

It's worth noting that vectordb as not the only kind of tool to retrieve documents.

The `LangChain` retriever abstraction includes other ways to retrieve documents, such as TF-IDF or SVM.

In [ ]:
```
from langchain.retrievers import SVMRetriever
from langchain.retrievers import TFIDFRetriever
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
```

In [ ]:
```
# Load PDF
loader = PyPDFLoader("docs/cs229_lectures/MachineLearning-Lecture01.pdf")
pages = loader.load()
all_page_text=[p.page_content for p in pages]
joined_page_text=" ".join(all_page_text)

# Split
text_splitter = RecursiveCharacterTextSplitter(chunk_size = 1500,chunk_overl
splits = text_splitter.split_text(joined_page_text)
```

In [ ]:
```
# Retrieve
svm_retriever = SVMRetriever.from_texts(splits,embedding)
tfidf_retriever = TFIDFRetriever.from_texts(splits)
```

In [ ]:
```
question = "What are major topics for this class?"
docs_svm=svm_retriever.get_relevant_documents(question)
docs_svm[0]
```

In [ ]:
```
question = "what did they say about matlab?"
docs_tfidf=tfidf_retriever.get_relevant_documents(question)
docs_tfidf[0]
```

In [ ]: