Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 20

Retrieval Augmented Generation (RAG)

Nicola Capuano and Antonio Greco

DIEM – University of Salerno

Outline

- Introduction to RAG
- Introduction to **LangChain**
- Building a RAG with **LangChain** and **HuggingFace**

Introduction to RAG

What is RAG?

LLMs can reason about **wide-ranging topics**, but:

- Their knowledge is limited to **data used during training**
- They cannot access **new information** introduced after training
- They cannot reason about **private or proprietary data**

Retrieval Augmented Generation (RAG) is a technique to augment the knowledge of large language models (LLMs) with **additional data**

RAG enables the creation of AI applications that can reason about **private data** and data introduced **after a model's cutoff date**

RAG Concepts

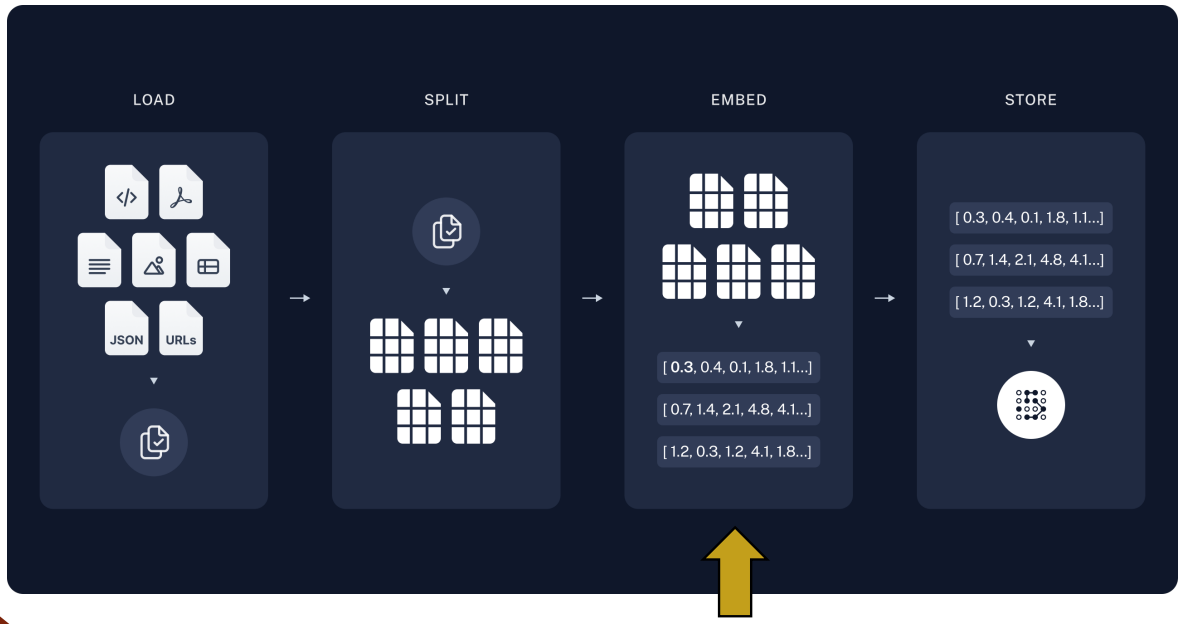
A typical RAG application has **two main components**:

- **Indexing:** a pipeline for ingesting data from a source and indexing it
  - This usually happens **offline**
- **Retrieval and generation:**
  - Takes the **user query** at run time
  - Retrieves the **relevant data** from the index
  - Generates a **prompt** that includes both the query with the retrieved data
  - Uses an LLM to **generate an answer**

Indexing

- **Load:** first we need to load our data
  - Popular RAG frameworks include **document loaders** for a variety of formats (PDF, CSV, HTML, Json, …) and sources (file system, Web, databases, …)
- **Split:** large documents are split into **smaller chunks**…
  - for **indexing** (smaller chunks are easier to search)
  - for **LLM usage** (smaller chunks fit within the model's finite context window)
- **Store:** We need somewhere to store and index our splits, so that they can be searched over later
  - This is often done using a **Vector Store**

Indexing

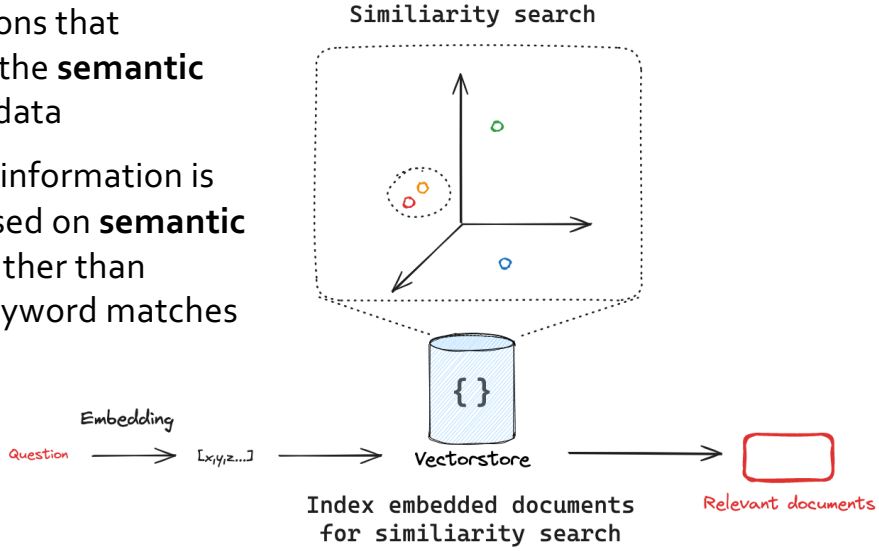


Vector stores represent splits with **vector representations (embeddings)**

Vector Stores

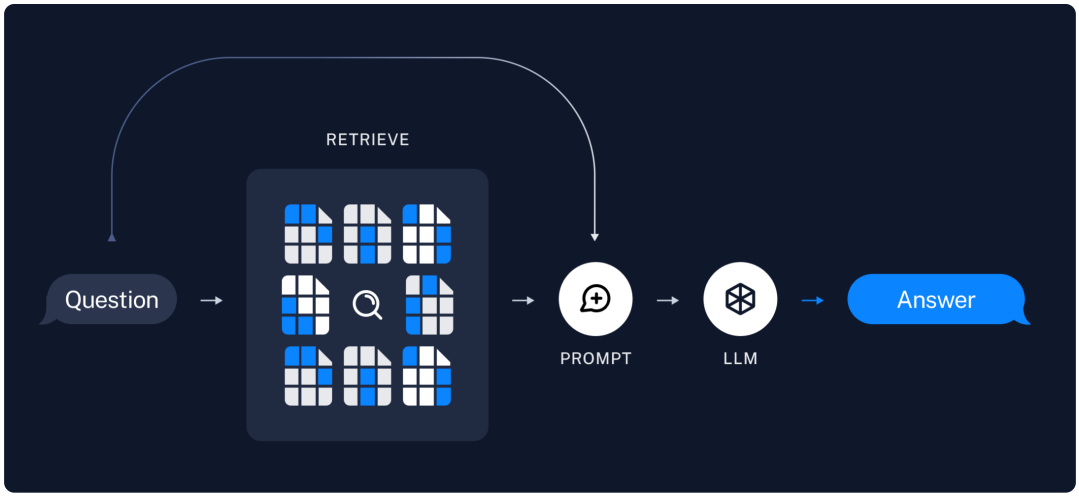
Specialized **data stores** that enable indexing and retrieving information based on **embeddings**

- **Recap:** embeddings are vector representations that encapsulate the **semantic meaning** of data
- **Advantage:** information is retrieved based on **semantic similarity**, rather than relying on keyword matches



Retrieval and Generation

- Given a user input, **relevant splits** are retrieved from storage
- A LLM produces an answer using a prompt that includes both the **question** with the **retrieved data**



Introduction to LangChain

LangChain

A framework built to **simplify the development of applications that utilize LLMs**

- It provides a set of **building blocks** to incorporate LLMs into applications
- It connects to **third-party LLMs** (like OpenAI and HuggingFace), **data sources** (such as Slack or Notion), and **external tools**
- It enables the **chaining of different components** to create sophisticated workflows
- It supports several **use cases** like chatbots, document search, RAG, Q&A, data processing, information extraction …
- It has **open-source** and **commercial** components

Key Components

- **Prompt Templates:** Facilitate translating user input into language model instructions, supporting both string and message list formats
- **LLMs:** Third-party language models that accept strings or messages as input and return strings as output
- **Chat Models:** Third-party models using sequences of messages as input and output, supporting distinct roles within conversational messages
- **Example Selectors:** Dynamically select and format concrete examples in prompts to enhance model performance

Key Components

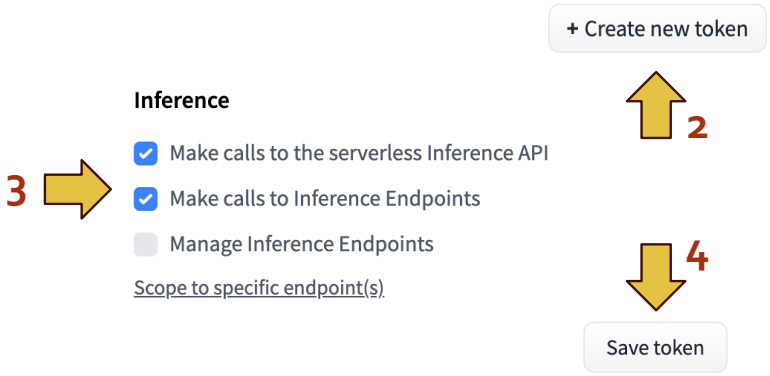
- **Output Parsers:** Convert model-generated text into structured formats (e.g., JSON, XML, CSV), supporting error correction and advanced features
- **Document Loaders:** Load documents from various data sources
- **Vector Stores:** Systems for storing and retrieving unstructured documents and data using embedding vectors
- **Retrievers:** Interfaces for document and data retrieval compatible with vector stores and other external sources
- **Agents:** Systems leveraging LLMs for reasoning to decide actions based on user inputs

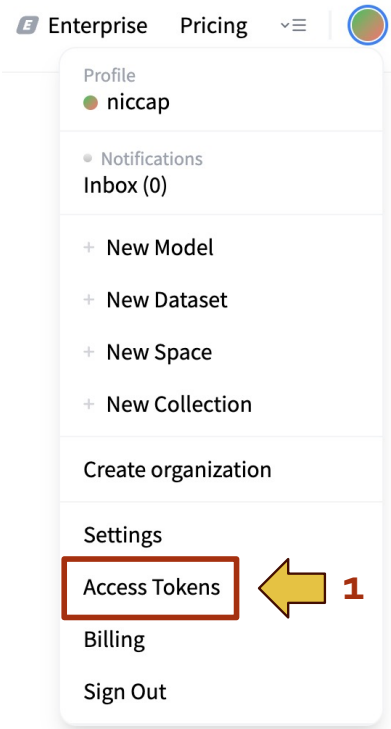
Installation

- Install core LangChain library
  `pip install langchain`
- Install community-contributed extensions
  `pip install langchain_community`
- Installs Hugging Face integration for LangChain
  `pip install langchain_huggingface`
- Install a library to load, parse, and extract text from PDF
  `pip install pypdf`
- Install the FAISS vector store (CPU version)
  `pip install faiss-cpu`

Preliminary Steps

Obtain a Hugging Face **access token**



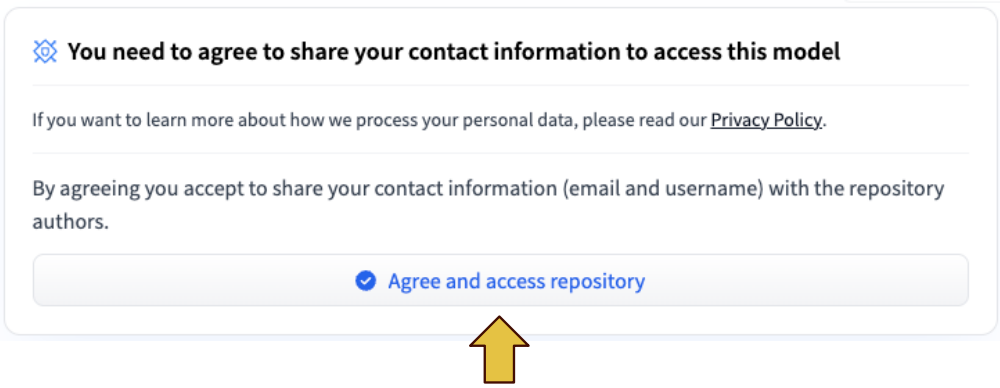


Copy your Access token

Preliminary Steps

Gain access to the **Mistral-7B-Instruct-v0.2** model by accepting the user license

https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2



Query a LLM Model

```python
from langchain_huggingface import HuggingFaceEndpoint
import os

os.environ["HUGGINGFACEHUB_API_TOKEN"] = "YOUR_API_TOKEN"

llm = HuggingFaceEndpoint(
    repo_id = "mistralai/Mistral-7B-Instruct-v0.2",
    temperature = 0.1
)

query = "Who won the FIFA World Cup in the year 2006?"
print(llm.invoke(query))
```

```
The FIFA World Cup in the year 2006 was won by the Italian national football team. They defeated France
in the final match held on July 9, 2006, at the Allianz Arena in Munich, Germany. The Italian team was
coached by Marcello Lippi and was led by the legendary goalkeeper Gianluigi Buffon. The team's victory
was significant as they had not won the World Cup since 1982. The final match ended in a 1-1 draw after
extra time, and the Italians won the penalty shootout 5-3. The winning goal in the shootout was scored by
Andrea Pirlo.
```

Query a LLM Model

```python
from langchain_huggingface import HuggingFaceEndpoint
import os

os.environ["HUGGINGFACEHUB_API_TOKEN"] = "YOUR_API_TOKEN"

llm = HuggingFaceEndpoint(
    repo_id = "mistralai/Mistral-7B-Instruct-v0.2",
    temperature = 0.1
)

query = "Who won the
print(llm.invoke(que
```

```
The FIFA World Cup in the y
in the final match held on
coached by Marcello Lippi a
was significant as they had
extra time, and the Italian
Andrea Pirlo.
```

NOTE: it would be better to save your **API key** in an **environment variable**

- on **macOS** or **Linux**:

  `export OPENAI_API_KEY="your_api_key_here"`

- on **Windows** with **PowerShell**

  `setx OPENAI_API_KEY "your_api_key_here"`

Prompt Templates

Predefined text structures used to create **dynamic and reusable prompts** for interacting with LLMs

```python
from langchain.prompts import PromptTemplate

template = "Who won the {competition} in the year {year}?"
prompt_template = PromptTemplate(
    template = template,
    input_variables = ["competition", "year"]
)

query = prompt_template.invoke({"competition": "Davis Cup", "year": "2018"})
answer = llm.invoke(query)

print(answer)
```

```
The Davis Cup in the year 2018 was won by Croatia. They defeated France in the final
held in Lille, France. The Croatian team was led by Marin Čilić and Borna Ćorić,
while the French team was led by Jo-Wilfried Tsonga and Lucas Pouille. Croatia won
the tie 3-2. This was Croatia's first Davis Cup title.
```

Introduction to Chains

Chains **combine multiple steps** in an NLP pipeline

- The **output** of each step becomes the **input** for the next
- Useful to automate **complex tasks**, and integrate external systems

```python
chain = prompt_template | llm
answer = chain.invoke({"competition": "Davis Cup", "year": "2018"})

print(answer)
```

```
The Davis Cup in the year 2018 was won by Croatia. They defeated France in the final
held in Lille, France. The Croatian team was led by Marin Čilić and Borna Ćorić,
while the French team was led by Jo-Wilfried Tsonga and Lucas Pouille. Croatia won
the tie 3-2. This was Croatia's first Davis Cup title.
```

Introduction to Chains

We want to **refine the LLM output** for future processing...

```
followup_template = """
task: extract only the name of the winning team from the following text
output format: json without formatting
example: {{"winner": "Italy"}}
### {text} ###
"""

followup_prompt_template = PromptTemplate(
    template = followup_template,
    input_variables = ["text"]
)

followup_chain = followup_prompt_template | llm

print(followup_chain.invoke({"text": answer}))
```

```
{"winner": "Croatia"}
```

Chaining all Together

```
from langchain_core.runnables import RunnablePassthrough

chain = (
    prompt_template
    | llm
    | {"text": RunnablePassthrough()}
    | followup_prompt_template
    | llm
)

print(chain.invoke({"competition": "Davis Cup", "year": "2018"}))
```

```
{"winner": "Croatia"}
```

This **forwards** the **output** of the previous step to the **next step** associated with a specific **dictionary key**

More on Chains

LCEL (LangChain Expression Language) is a syntax to create modular pipelines for chaining operations

- **Pipe Syntax:** The | operator allows chaining of operations
- LCEL supports **modular** and **reusable** components
- It can handle **branching logic** or **follow-up queries**

LangChain includes **Predefined Chains** for common tasks like question answering, document summarization, conversational agents, etc.

Documentation: https://python.langchain.com/docs/

Building a RAG with LangChain and GPT

Example Project

Building a RAG able to answer queries on some documents from the **Census Bureau** US

- An agency responsible for collecting statistics about the nation, its people, and its economy

We first download some **documents to be indexed**:

```
from urllib.request import urlretrieve

os.makedirs("us_census", exist_ok = True)
files = [
    "https://www.census.gov/content/dam/Census/library/publications/2022/demo/p70-178.pdf",
    "https://www.census.gov/content/dam/Census/library/publications/2023/acs/acsbr-017.pdf",
    "https://www.census.gov/content/dam/Census/library/publications/2023/acs/acsbr-016.pdf",
    "https://www.census.gov/content/dam/Census/library/publications/2023/acs/acsbr-015.pdf",
]
for url in files:
    file_path = os.path.join("us_census", url.split("/")[-1])
    urlretrieve(url, file_path)
```

Document Loaders

LangChain components used to **extract content** from diverse data sources:

- **TextLoader:** Handles plain text files
- **PyPDFLoader:** Extracts content from PDF documents
- **CSVLoader:** Reads tabular data from CSV files
- **WebBaseLoader:** Extracts content from web pages
- **WikipediaLoader:** Fetches Wikipedia pages
- **SQLDatabaseLoader:** Queries SQL databases to fetch and load content
- **MongoDBLoader:** Extracts data from MongoDB collections
- **IMAPEmailLoader:** Loads emails using the IMAP protocol from various providers
- **HuggingFaceDatasetLoader:** Fetches datasets from Hugging Face datasets library
- and many others…

Extract Content from PDFs

We can use **PyPDFLoader** to extract text from a single PDF document …

```
from langchain_community.document_loaders import PyPDFLoader

loader = PyPDFLoader("us_census/p70-178.pdf")
doc = loader.load()
print(doc[0].page_content[0:100]) # Print the first page (first 100 characters)
```

```
Occupation, Earnings, and Job
Characteristics
July 2022
P70-178
Clayton Gumber and Briana Sullivan
```

Extract Content from PDFs

Or we can use **PyPDFDirectory** to fetch a set of PDF documents from a folder …

```
from langchain_community.document_loaders import PyPDFDirectoryLoader

loader = PyPDFDirectoryLoader("./us_census/")
docs = loader.load()

print(docs[60].page_content[0:100]) # The 61st page (documents are concatenated)
print('\n' + docs[60].metadata['source']) # The source of the page 61st page
```

```
U.S. Census Bureau  19
insurance, can exacerbate the dif-
ferences in monetary compensa -
tion and w

us_census/p70-178.pdf
```

Text Splitters

LangChain components used to **break large text** documents into **smaller chunks**

- **CharacterTextSplitter:** Splits text into chunks based on a character count
- **RecursiveCharacterTextSplitter:** Attempts to split text intelligently by using hierarchical delimiters, such as paragraphs and sentences
- **TokenTextSplitter:** Splits text based on token count rather than characters
- **HTMLHeaderTextSplitter:** Splits HTML documents by focusing on headers (e.g., <h1>, <h2>) to create meaningful sections from structured web content
- **HTMLSectionSplitter:** Divides HTML content into sections based on logical groupings or structural markers
- **NLPTextSplitter:** Uses NLP to split text into chunks based on semantic meaning
- and many others…

Split Text in Chunks

We use **RecursiveCharacterTextSplitter** to obtain…

- Chunks of about **700** characters
- Overlapped for about **50** characters

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 700,
    chunk_overlap = 50,
)
chunks = text_splitter.split_documents(docs)

print(chunks[0].page_content) # The first chunk
```

```
Health Insurance Coverage Status and Type
by Geography: 2021 and 2022
American Community Survey Briefs
ACSBR-015
...
```

Split Text in Chunks

Printing the **average document length** before and after splitting…

```
len_chunks = len(chunks)
avg_docs = sum([len(doc.page_content) for doc in docs])//len(docs)
avg_chunks = sum([len(chunk.page_content) for chunk in chunks])//len(chunks)

print(f'Before split: {len(docs)} pages, with {avg_docs} average characters.')
print(f'After split: {len(chunks)} chunks, with {avg_chunks} average characters.')
```

```
Before split: 63 pages, with 3840 average characters.
After split: 398 chunks, with 624 average characters.
```

Index Chunks

Embeddings are used to index text chunks, enabling semantic search and retrieval

- We can use **pre-trained embedding models** from Hugging Face
- **Bi-Directional Generative Embeddings** (BGE) models excel at **capturing relationships between text pairs**
- **BAAI/bge-small-en-v1.5** is a lightweight embedding model from the Beijing Academy of Artificial Intelligence
- Suitable for tasks requiring **fast generation**

https://huggingface.co/BAAI/bge-small-en-v1.5

Index Chunks

```python
from langchain_community.embeddings import HuggingFaceBgeEmbeddings
import numpy as np

embedding_model = HuggingFaceBgeEmbeddings(
    model_name = "BAAI/bge-small-en-v1.5",
    encode_kwargs = {'normalize_embeddings': True} # useful for similarity tasks
)

# Embed the first document chunk
sample_embedding = np.array(embedding_model.embed_query(chunks[0].page_content))

print(sample_embedding[:5])
print("\nSize: ", sample_embedding.shape)
```

```
[-0.07508391 -0.01188472 -0.03148879  0.02940382  0.05034875]

Size:  (384,)
```

Vector Stores

Enable **semantic search and retrieval** by indexing and querying embeddings of documents or text chunks:

- **FAISS:** Open-source library for efficient similarity search and clustering of dense vectors, ideal for local and small-to-medium datasets
- **Chroma:** Lightweight and embedded vector store suitable for local applications with minimal setup
- **Qdrant:** Open-source vector database optimized for similarity searches and nearest-neighbor lookup
- **Pinecone:** Managed vector database offering real-time, high-performance semantic search with automatic scaling
- and many others…

Vector Stores

We use the **Facebook AI Similarity Search** (**FAISS**)

```
from langchain_community.vectorstores import FAISS

# Generate the vector store
vectorstore = FAISS.from_documents(chunks, embedding_model)

# Save the vector store for later use...
vectorstore.save_local("faiss_index")

# To load the vector store later...
# loaded_vectorstore = FAISS.load_local("faiss_index", embedding_model)
```

Querying the Vector Store

We can use the vector store to **search for chunks relevant to a user query**

```
query = """
What were the trends in median household income across
different states in the United States between 2021 and 2022.
"""

matches = vectorstore.similarity_search(query)

print(f'There are {len(matches)} relevant chunks.\nThe first one is:\n')
print(matches[0].page_content)
```

```
There are 4 relevant chunks.
The first one is:

Comparisons
The U.S. median household income
...
```

RAG Prompt Template

Must integrate retrieved **context** with a user **question**

- A placeholder **`context`** is used to dynamically inject retrieved chunks
- A placeholder **`question`** is used to specify the user query
- Explicit instructions are provided for handling the information

```
template = """
Use the following pieces of context to answer the question at the end.
Please follow the following rules:
1. If you don't know the answer, don't try to make up an answer. Just say "I can't find the final answer".
2. If you find the answer, write the answer in a concise way with five sentences maximum.

{context}

Question: {question}

Helpful Answer:
"""

prompt_template = PromptTemplate(template = template, input_variables = ["context", "question"])
```

Vector Store as a Retriever

To be used in chains, a vector store must be wrapped within a **retriever** interface

- A retriever takes a text query as **input** and provides the most relevant information as **output**

```
# Create a retriever interface on top of the vector store
retriever = vectorstore.as_retriever(
    search_type = "similarity", # use cosine similarity
    search_kwargs = {"k": 3} # use the top 3 most relevant chunks
    )
```

Custom RAG Chain

Then we can define our **RAG chain**...

```python
from langchain_core.runnables import RunnablePassthrough

# Helper function to concatenate the retrieved chunks
def format_docs(docs):
    return "\n\n".join(doc.page_content for doc in docs)

my_rag_chain = (
    {
        "context": retriever | format_docs,
        "question": RunnablePassthrough()
    }
    | prompt_template
    | llm
)
```

Builds a **dictionary** where each value is obtained as the result of a **sub-chain**

Predefined Chains

LangChain include ready-to-use **chains that handle common tasks** with minimal setup

- **LLMChain:** Executes a single prompt using a language model and returns the output
- **RetrievalQAChain:** Combines a retriever and an LLM to answer questions based on retrieved context
- **AnalyzeDocumentChain:** Extracts insights, structured data, or key information from documents
- **SequentialChain:** Executes a series of chains sequentially, passing outputs from one as inputs to the next
- **ConditionalChain:** Executes different chains based on conditions in the input or intermediate outputs
- and many others...

Predefined RAG Chain

We can use a predefined **RetrievalQA** chain instead of our custom chain

```
from langchain.chains import RetrievalQA

# Create a RetrievalQA chain
retrievalQA = RetrievalQA.from_chain_type(
    llm = llm,
    retriever = retriever,
    chain_type = "stuff", # concatenate retrieved chunks
    chain_type_kwargs = {"prompt": prompt_template},
    return_source_documents = True
)
```

Querying the RAG

We can now use the RAG for **question answering**

```
query = """
What were the trends in median household income across
different states in the United States between 2021 and 2022.
"""

# Call the QA chain with our query.
result = retrievalQA.invoke({"query": query})
print(result['result'])
```

```
Five states, including Alabama, Alaska, Delaware, Florida, and Utah, experienced a statistically
significant increase in real median household income from 2021 to 2022. Conversely, 17 states showed
a decrease. For 28 states, the District of Columbia, and Puerto Rico, there was no statistically
significant difference in real median household income between the two years.
```

Querying the RAG

Which **chunks** have been used to **generate the answer**?

```
sources = result['source_documents']
print(f'{len(sources)} chunks have been used to generate the answer.')

for doc in sources:
    print(f"\n------- from: {doc.metadata['source']}, page: {doc.metadata['page']}\n\n{doc.
    page_content}")
```

```
3 chunks have been used to generate the answer.

------- from: us_census/acsbr-017.pdf, page: 1

Comparisons
The U.S. median household income
in 2022 was $74,755, according
Figure 1.
Median Household Income in the Past 12 Months in the United States: 2005–2022
...
```

Natural Language Processing and Large Language Models

Corso di Laurea Magistrale in Ingegneria Informatica

Lesson 20

Retrieval Augmented Generation (RAG)

Nicola Capuano and Antonio Greco

DIEM – University of Salerno