youtube_rag

April 25, 2024

1 Youtube RAG

You want to know what a youtube-video is about without spending the time to watch it?
You want to summarize what was told in a youtube-video?
You want to get information from an influencer via their youtube-videos without watching all of them?

Then this might be an interesting notebook for you. Here I show you how to use youtube videos in combination with a RAG-Model to give you details about the video. RAG introduces an information retrieval component that uses user input to initially retrieve information from a new data source called the vectorstore in order to optimize performance and prize for the llm-calls.

This notebook shows a straight forward approach for creating a RAG-Model with openAI (python package Version 1+) and langchain. You can also connect Langsmith with it so you can also track your calls to the RAG-Model.

[1]:
```
import os
import getpass

from langchain import hub
from langchain.docstore.document import Document
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import Runnable, RunnablePassthrough
from langchain_community.callbacks import get_openai_callback
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
import openai
from pytube import YouTube
```

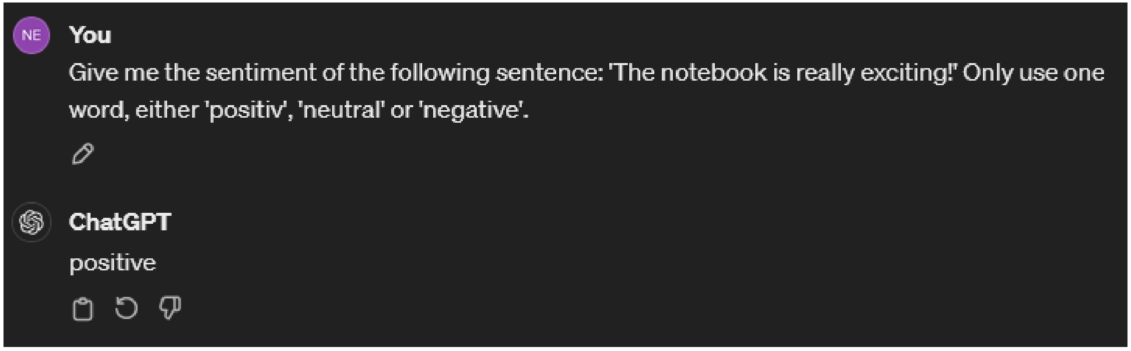
1.1 The Usages

1.1.1 What are LLMs?

Large Language Models (LLMs) are advanced artificial intelligence models capable of understanding and generating human-like text.
They are trained on vast amounts of data to learn the patterns and structures of language.
LLMs utilize deep learning techniques to process input text and generate coherent responses, enabling them to perform a wide range of natural language understanding and generation tasks.

1.1.2 What can LLMs do?

Read You can give it a text and it can summarize it or categorize it (p.E. sentiment):



You

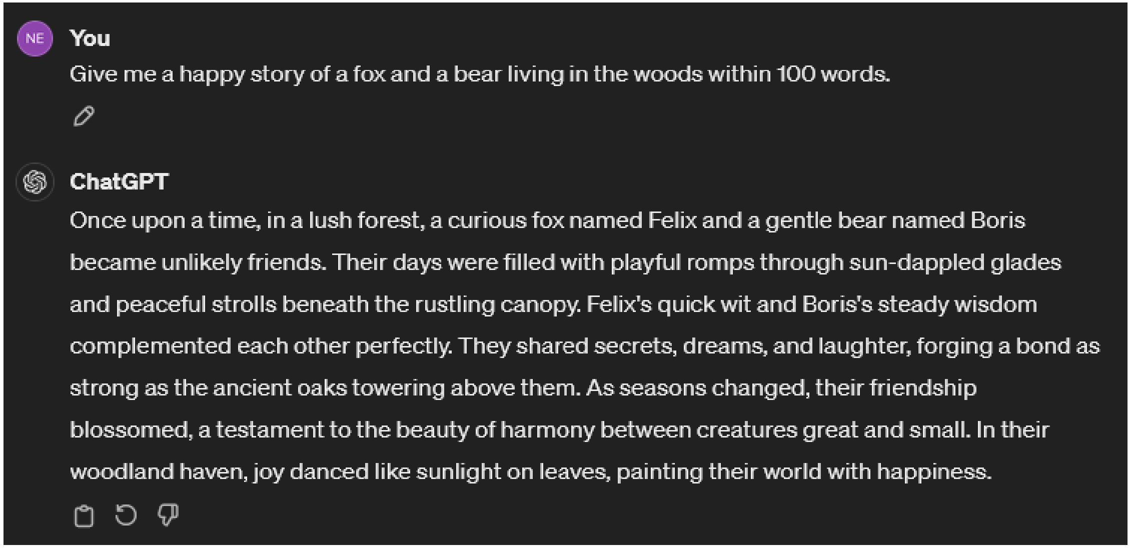
Give me the sentiment of the following sentence: 'The notebook is really exciting!' Only use one word, either 'positiv', 'neutral' or 'negative'.

ChatGPT

positive

This enables quite a view use-cases like categorizing reviews, translating text or summarizing large amounts of text.

Write You can ask it to create text:



You

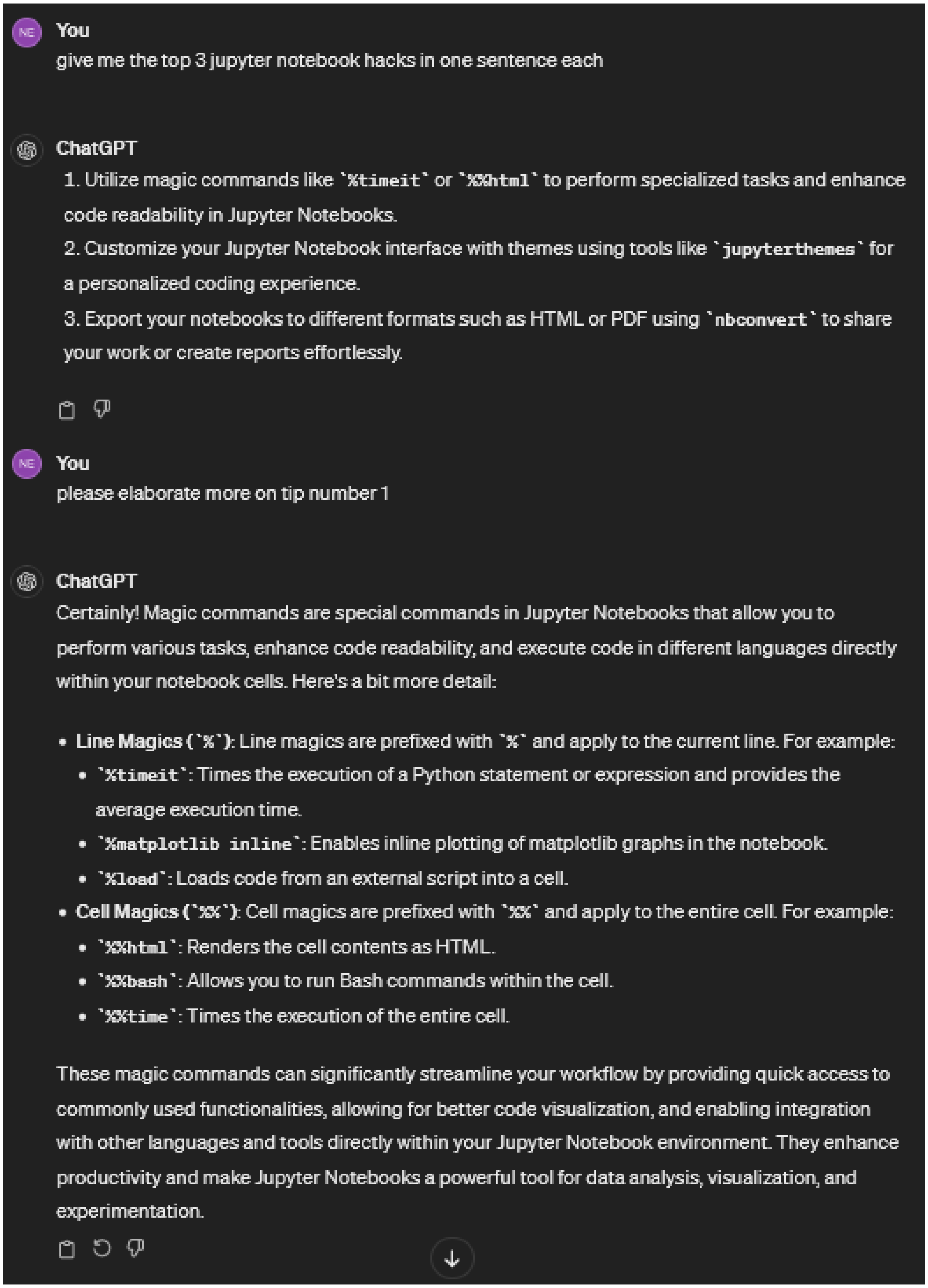
Give me a happy story of a fox and a bear living in the woods within 100 words.

ChatGPT

Once upon a time, in a lush forest, a curious fox named Felix and a gentle bear named Boris became unlikely friends. Their days were filled with playful romps through sun-dappled glades and peaceful strolls beneath the rustling canopy. Felix's quick wit and Boris's steady wisdom complemented each other perfectly. They shared secrets, dreams, and laughter, forging a bond as strong as the ancient oaks towering above them. As seasons changed, their friendship blossomed, a testament to the beauty of harmony between creatures great and small. In their woodland haven, joy danced like sunlight on leaves, painting their world with happiness.

This enables use-cases like content creation, brain-storming for ideas or #### Chat

You can give the llm a knowledge of the previous conversation (called context) so it can reference previous interactions.



You
give me the top 3 jupyter notebook hacks in one sentence each

ChatGPT

1. Utilize magic commands like `%timeit` or `%%html` to perform specialized tasks and enhance code readability in Jupyter Notebooks.
2. Customize your Jupyter Notebook interface with themes using tools like `jupyterthemes` for a personalized coding experience.
3. Export your notebooks to different formats such as HTML or PDF using `nbconvert` to share your work or create reports effortlessly.

You
please elaborate more on tip number 1

ChatGPT
Certainly! Magic commands are special commands in Jupyter Notebooks that allow you to perform various tasks, enhance code readability, and execute code in different languages directly within your notebook cells. Here's a bit more detail:

- **Line Magics (`%`)**: Line magics are prefixed with `%` and apply to the current line. For example:
  - `%timeit`: Times the execution of a Python statement or expression and provides the average execution time.
  - `%matplotlib inline`: Enables inline plotting of matplotlib graphs in the notebook.
  - `%load`: Loads code from an external script into a cell.
- **Cell Magics (`%%`)**: Cell magics are prefixed with `%%` and apply to the entire cell. For example:
  - `%%html`: Renders the cell contents as HTML.
  - `%%bash`: Allows you to run Bash commands within the cell.
  - `%%time`: Times the execution of the entire cell.

These magic commands can significantly streamline your workflow by providing quick access to commonly used functionalities, allowing for better code visualization, and enabling integration with other languages and tools directly within your Jupyter Notebook environment. They enhance productivity and make Jupyter Notebooks a powerful tool for data analysis, visualization, and experimentation.

Use-Cases are of course Chat-bots/Customer Support but also Q&As for eductation or training.

1.2 The Setup

First of all we specify some paths to save the youtube videos and audios to.
Then we need to specify the api-keys for open-ai (necessary) and langsmith (not necessary) if accessable.

[2]:

```
audio_path = "../artifacts/audio"
video_path = "../artifacts/video"

os.environ['OPENAI_API_KEY'] = getpass.getpass('OpenAI API Key:')
os.environ['LANGCHAIN_TRACING_V2'] = "True"
os.environ['LANGCHAIN_ENDPOINT'] = "https://api.smith.langchain.com"
os.environ['LANGCHAIN_API_KEY'] = getpass.getpass('LangChain API Key:')
os.environ['LANGCHAIN_PROJECT'] = "test_project"

client = openai.OpenAI(api_key=os.environ['OPENAI_API_KEY'])
```

1.3 The Video(s)

Here we specify the youtube videos for our knowledge-base.

[3]:

```
urls = ['https://www.youtube.com/watch?v=nYdE3DDtNgA']
```

1.4 The Audio

At first we need to extract the audio from the youtube videos and save them in `audio_path`. We can do that directly via the package `pytube` and in order to save discspace, for this use-case, we just download the audio-data and ignore the video itself.

The next step is to transcribe the audio into text which is done with the `whisper-1` model from the `openai`-package. You can also use the `speechrecognition` package here.

In order to be processed correctly and add some metadata there is a handy class from `langchain` calles `Document` to process the docs correctly.

[4]:

```
def get_audio_from_youtube(url: str = 'https://youtu.be/2lAe1cqCOXo',␣
 ↪audio_path: str = "../artifacts/audio") -> str:
    """
    :param url: string url
    :param audio_path: string audio_path

    - download audio from youtube
    """
    os.makedirs(audio_path, exist_ok=True)
    audio = YouTube(url).streams.filter(only_audio=True).first()
    audio.download(output_path=audio_path)
    return audio.default_filename

def speech_to_text(client: object, audio_file_name: str = "speech_recording.
 ↪mp4",
                   file_path: str = "../artifacts/audio", language: str = "en")␣
 ↪-> str:
    """
    :param audio_file_name: string audio_file_name
    :param language: string language
    :return: string

    - get a response from chatGPT
    - convert speech to text
    """
    with open(f"{file_path}/{audio_file_name}", "rb") as audio_file:
        response = client.audio.transcriptions.create(model="whisper-1",␣
↪language=language, file=audio_file)
        gpt_result = response.text

    return gpt_result
```

```
[5]: docs = []
     for url in urls:
         audio_file = get_audio_from_youtube(url=url, audio_path=audio_path)
         print(f"audio file: {audio_file}")
         text = speech_to_text(client=client, audio_file_name=audio_file,␣
     ↪file_path=audio_path, language="en")
         doc = Document(
                     page_content=text,
                     metadata={"source": audio_file, "page": 1},
                 )
         docs.append(doc)
```

```
audio file: Why 'The Godfather is the GREATEST MOVIE OF ALL TIME.mp4
```

1.5 The Documents

Now we need to split the texts into context-chunks in order to give the rag enough context to make reasonable decisions but also keep the context small enough so that the call isn't too big and expensive.

Chunk-size = 1000 and chunk_overlap = 200 means that each chunk has 1000 tokens (rule of thumb: 3 words refer to 4 tokens) with with an overlap of 200.

Example: tokenization

```
[6]: text = "What's the difference between tokens and words in large language models␣
     ↪like chat-gpt or Dolly or Llama3?"

     import tiktoken
     encoding = tiktoken.encoding_for_model("gpt-3.5-turbo")
     token_int = encoding.encode(text)
     print(f"integer tokens: {token_int}")
     token_str = [encoding.decode_single_token_bytes(token) for token in token_int]
     print(f"String tokens: {token_str}")
```

```
integer tokens: [3923, 596, 279, 6811, 1990, 11460, 323, 4339, 304, 3544, 4221,
4211, 1093, 6369, 2427, 418, 477, 423, 8788, 477, 445, 81101, 18, 30]
```

```
String tokens: [b'What', b"'s", b' the', b' difference', b' between', b'
tokens', b' and', b' words', b' in', b' large', b' language', b' models', b'
like', b' chat', b'-g', b'pt', b' or', b' D', b'olly', b' or', b' L', b'lama',
b'3', b'?']
```

Example: chunk_size and chunk_overlap

[7]:
```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=40, chunk_overlap=10)
splits = text_splitter.split_text(text)
print(splits)
```

```
["What's the difference between tokens and", 'and words in large language models
like', 'like chat-gpt or Dolly or Llama3?']
```

Necessary code for the RAG:

[8]:
```
def split_docs(docs: list, chunk_size: int = 1000, chunk_overlap: int = 200) ->␣
↪list:
    """
    :param docs: list of documents
    :param chunk_size: integer chunk_size of the documents
    :param chunk_overlap: integer chunk_overlap for each chunk
    :return: list of splits

    - split documents into chunks
    """
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size,␣
↪chunk_overlap=chunk_overlap)
    splits = text_splitter.split_documents(docs)
    return splits
```

[9]:
```
splits = split_docs(docs, chunk_size=1000, chunk_overlap=200)
for doc in splits:
    print(doc.metadata)
    print(len(doc.page_content))
```

```
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
996
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
999
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
998
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
998
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
```

```
1}
999
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
998
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
998
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
997
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
996
{'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL TIME.mp4', 'page':
1}
861
```

1.6 The Vectorstore

Now we have quite some context-docs to enable our llm to answer questions about the video. But in order to give the right context out of all these context-docs to the llm we need to create something that's called a Vectorstore.

Vectorstores basically transform the context into numerical vectors where you then make an comparison of the question (also transformed to a vector) and the context. And the context that fits best to the question is given to the llm for reasoning.

[10]:

```python
def create_vectorstore(
    splits: list,
    vectore_store_name: str = "chromadb",
) -> object:
    """
    :param splits: list[Document] splits to create the vectorstore
    :param vectore_store_name: string vectore_store_name to choose the␣
 ↪vectorstore
    :return: object

    - create a vectorstore from the pages
    - different vectorstore options are available
    """

    embeddings = OpenAIEmbeddings()

    if vectore_store_name == "chromadb":
        from langchain_community.vectorstores import Chroma
        vectorstore = Chroma.from_documents(documents=splits,␣
 ↪embedding=embeddings)
```

```
    elif vectore_store_name == "deeplake":
        from langchain_community.vectorstores import DeepLake
        vectorstore = DeepLake.from_documents(
            documents=splits,
            dataset_path="./my_deeplake/",
            embedding=embeddings,
            overwrite=True,
        )

    elif vectore_store_name == "faiss":
        from langchain_community.vectorstores import FAISS
        vectorstore = FAISS.from_documents(splits, embeddings)

    elif vectore_store_name == "annoy":
        from langchain_community.vectorstores import Annoy
        vectorstore = Annoy.from_documents(splits, embeddings)

    elif vectore_store_name == "docarray_hnsw_search":
        from langchain.vectorstores.docarray.hnsw import DocArrayHnswSearch
        vectorstore = DocArrayHnswSearch.from_documents(splits, embeddings,␣
 ↪work_dir="hnswlib_store/", n_dim=1536)

    elif vectore_store_name == "docarray_inmemory":
        from langchain_community.vectorstores.docarray import␣
 ↪DocArrayInMemorySearch
        vectorstore = DocArrayInMemorySearch.from_documents(splits, embeddings)

    else:
        raise ValueError("Unknown vectorstore name")

    return vectorstore
```

[11]:
```
vectorstore = create_vectorstore(splits, vectore_store_name="chromadb")
```

1.7 The Retreiver

The retreiver is a lightweight wrapper around the vectorstore that lets you query questions and retrieve the contents that are closest to the question.

By default it uses similarity-search for finding the best fitting answers.

[12]:
```
retriever = vectorstore.as_retriever()

retriever.invoke("Why does the film appeal to the reviewer?")
```

[12]:
```
[Document(page_content="outfit here was amongst my favorite looks of his. Just a
gray jacket, faded olive green inside, and a burgundy base tie. So what'd you
```

```
think? If you watched this film, did you previously notice the things I
mentioned? Most of my observations came from watching this film for a second
time, which is just one of the things I love about cinema. It's okay if everyone
doesn't see it or get it, it's meant to be seen again and again, and with every
watch, there is always something new to discover. If you want to say in what
movies I get to review next, I have polls on my Twitter twice a week, and if I
somehow get a chance to watch any movies outside of the ones I review, you can
follow my letterboxd to keep updated on that. Thank you so so much to the
American Film Institute for providing the 100 Greatest American Films of All
Time list. One down, 99 to go.", metadata={'page': 1, 'source': 'Why ‘The
Godfather is the GREATEST MOVIE OF ALL TIME.mp4'}),
 Document(page_content="without Francis Ford Coppola's amazing direction, and
just Francis Ford Coppola in general, so I'm still a little bit salty about his
Oscar snub. So my impression of the film? Uh, I loved it. I envy Italian
families, they're so big and loving and fun, just look how joyous this wedding
scene is. This scene takes place right at the beginning of the film, while
simultaneously cutting between business meetings. At first I thought the wedding
scene was overexposed, until you realize how underexposed the office scenes are,
even with the windows open to broad daylight. This is the first shot of the
movie, and it's one take that lasts for three minutes, ending with these two
characters' silhouette. I view this choice as two guys participating in shady
business, hence their blackout. Aside from that and overall, I was just in awe
of every single location they shot at. So in pre-production for a film, there is
a location team made up of scouts and managers who determine what location would
work", metadata={'page': 1, 'source': 'Why ‘The Godfather is the GREATEST MOVIE
OF ALL TIME.mp4'}),
 Document(page_content='and overall, I was just in awe of every single location
they shot at. So in pre-production for a film, there is a location team made up
of scouts and managers who determine what location would work best for a scene,
and they really did an amazing job with The Godfather. In the later part of the
film, Michael runs away to Sicily for two years, and Italy is already just such
a beautiful place on its own, but the medieval villages they shot in Savoca and
Forza d\'Agro were amongst my favorite. Speaking of favorite, welcome to script
vs. scene. So my favorite scene in this film takes place between the brothers,
mostly Sonny, Tom Hagen, and Michael, where the famous, "\'It\'s not personal,
it\'s strictly business,\' Linus said." And that, it\'s a really bad accent. I
usually like scenes like this that really let us see how an actor portrays their
character. None of what Sonny says or does in this moment was even written in
the script. Now, I love Al Pacino, and this by far is one of my favorite',
metadata={'page': 1, 'source': 'Why ‘The Godfather is the GREATEST MOVIE OF ALL
TIME.mp4'}),
 Document(page_content="and scene of the film. At this point, Michael's taken on
a certain position, and when it's time to start discussing business, she is left
out. Overall, the production design for The Godfather was so unbelievably
amazing, I am still so shocked Dean Tavaleris didn't even get nominated for his
work. But besides my frustration, I of course loved all of the mise-en-scene,
especially when Michael was in Sicily, aka my future home one day. But this was
```

```
just one of the many scenes I liked in particular. So, just a casual scene, a
nice day for some fruit. But, because I'm a detail-oriented psychopath, I really
appreciated this detail. Do you see it? It's a poster! In the back, it reads
Jake LaMotta and Tommy Bell. Well, when you look those names up, it turns out
that they are two American professional boxers, a part of this huge fight back
in January of 1946. Just so appropriately placed between Christmas shopping
scenes. But, to even really hone in on the specifics of the timeline, there's
a", metadata={'page': 1, 'source': 'Why ‘The Godfather is the GREATEST MOVIE OF
ALL TIME.mp4'})]
```

1.8 The Prompt

The langchain-community-hub provides different prompts for different use-cases. You can use p.E. `rlm/rag-prompt` which is a quite powerful prompt for knowledge extraction without halluzination.

We can also use a slightly modified version from the hub I created, where a "don't know" answer starts with "Sorry" and thus can be caught and handled: `nelzman/rag-prompt-with-dunno-catch`

[13]:

```python
#prompt = hub.pull("rlm/rag-prompt")
prompt = hub.pull("nelzman/rag-prompt-with-dunno-catch")

print(prompt.to_json()["kwargs"]["template"])
```

```
You are an assistant for question-answering tasks. Use the following pieces of
retrieved context to answer the question. If you don't know the answer, just say
that you don't know and start with 'Sorry' continued in the language of the
question. Use three sentences maximum, just answer the question and keep the
answer concise. The language of the Answer should be the same as the question.
Question: {question}
Context: {context}
Answer:
```

1.9 The Large Language Model

Here we are using `gpt-3.5-turbo-1106` from openAI to create the llm-interface for the RAG. One important parameter is the `temperature` which is used to control the randomness of the outputs.
Since we are creating a rag to retreive accurate information, we are setting `temperature == 0`

[14]:

```python
llm = ChatOpenAI(
    model_name="gpt-3.5-turbo-1106",  # currently only works with gpt-3.
 ↪5-turbo-1106
    temperature=0, # less creative responses
)
```

1.10 The RAG-Chain

Langchain is a Interface that enables chaining different parts of a execution-pipeline together into one object and thus enables a robust way to interact and use the RAG-model.

First we send the question to the retriever which gives us the best fitting context.
Then the Docs get formatted and giiven to the prompt.
The prompt then gets sent to the llm which gives back a json-like object which then gets parsed to string.

Et voilà, we have the anwer to our question.

[15]:
```
def format_docs(docs: list) -> str:
    """
    :param docs: list of documents to format
    :return: string

    - format the documents
    """
    return "\n\n".join(doc.page_content for doc in docs)

def create_rag_chain(retriever: object, prompt: object, llm: object) -> object:
    """
    :param retriever: vectorstore retreiver for the answer
    :param prompt: prompt for the question
    :param llm: large language model for the answer


    - create chain: retriever -> format_docs -> prompt -> llm -> StrOutputParser
    """

    rag_chain = (
        {"context": retriever | format_docs, "question": RunnablePassthrough()}␣
 ↪| prompt | llm | StrOutputParser()
    )

    return rag_chain
```

[16]:
```
rag_chain = create_rag_chain(retriever, prompt, llm)
```

1.11 The Test

This is just a Test of the things we implemented.

[17]:
```
question = "Why does the film appeal to the reviewer?"
response1 = rag_chain.invoke(question)
print(response1)
```

```
The film appeals to the reviewer because of the amazing direction by Francis
Ford Coppola, the beautiful locations, and the impressive production design. The
reviewer also loved the scenes that showcased the actors' portrayal of their
characters and the attention to detail in the mise-en-scene. Additionally, the
wedding scene and the use of lighting and exposure in the film were particularly
captivating to the reviewer.
```

You can also use a callback function to get the response from the openai model. `tiktoken` package is needed for that.

[18]:
```python
def check_rag_chain(rag_chain: object, prompt: str = None):
    with get_openai_callback() as cb:
        response = rag_chain.invoke(prompt)
    return response, cb
```

[19]:
```python
response2, rag_callback = check_rag_chain(rag_chain=rag_chain, prompt=question)
print(response2)
print(rag_callback)
```

```
The film appeals to the reviewer because of the amazing direction by Francis
Ford Coppola, the beautiful locations, and the impressive production design. The
reviewer also loved the scenes that showcased the actors' portrayal of their
characters and the attention to detail in the mise-en-scene. Additionally, the
wedding scene and the business meetings were particularly captivating for the
reviewer.
Tokens Used: 1038
        Prompt Tokens: 968
        Completion Tokens: 70
Successful Requests: 1
Total Cost (USD): $0.001108
```

1.11.1 Catching Dunno

If you don't find an answer, you can also create paraphrases for your question and then loop through them until you find an answer to your question.
Since we are using the prompt that always starts with "Sorry" if it didn't find an answer, we can loop through the paraphrases until it finds an answer to a "close enough" question.

[20]:
```python
paraphrases = llm.invoke(f"Create 5 paraphrases of the question: {question}␣
  ↪Seperate them by |||")
paraphrases_list = [value.replace("\n", "") for value in paraphrases.content.
  ↪split("|||")]
print(paraphrases_list)
```

```
['What makes the movie attractive to the critic? ', ' What is it about the film
that captures the attention of the reviewer? ', ' Why does the reviewer find the
film appealing? ', " What is the reason behind the reviewer's positive reaction
to the film? ", ' What is it about the movie that resonates with the reviewer?']
```

[23]:
```python
for question in paraphrases_list:
    response3 = rag_chain.invoke(question)
    if not response3.startswith("Sorry"):
        break
print(f"First question to retreive an answer: {question}")
print(f"Answer: {response3}")
```

First question to retreive an answer: What makes the movie attractive to the critic?

Answer: The movie is attractive to the critic because of the amazing direction by Francis Ford Coppola, the vibrant costume design, and the attention to detail in the production design. The critic also appreciates the joyous wedding scene and the use of color and contrast in the film. Additionally, the critic enjoys the attention to detail in the mise-en-scene and the placement of historical references in the movie.