Intro to Prompting module

When utilizing large language models, **prompt engineering** emerges as one of the most critical topics to learn. It is possible to achieve the **same level of accuracy by crafting the right prompt,** even when opting for less powerful or open-source models. During the course, we will explore the art of writing the perfect prompt, equipping you with the skills to maximize the potential of your models and get the response in the specified format. Furthermore, **few-shot prompts** provide a unique opportunity to enable the models to quickly acquire new knowledge and generalize to unseen tasks with minimal data, offering a remarkable capability for customization. The rest of this module is organized as follows.

- **Intro to Prompt Engineering: Tips and Tricks:** Explore a range of valuable tips and tricks for effective prompting. We will cover techniques such as role prompting, which involves specifying a specific role for the model, such as assistant or copywriter. Additionally, we will delve into few-shot prompting, teaching the model how to respond based on limited examples. Lastly, we will examine the chain of thought approach, which aids in enhancing reasoning capabilities. Throughout the lesson, we will provide numerous examples of successful and ineffective prompts, equipping you with the skills to master the art of crafting optimal prompts for various scenarios.
- **Using Prompt Templates:** Dynamic prompting provides a flexible approach to improve the model's context. This lesson offers a comprehensive explanation of few-shot learning, providing compelling examples and allowing the library to select suitable samples based on the input window length of the model.
- **Getting the Best of Few Shot Prompts and Example Selectors:** Discussing the advantages and disadvantages of few-shot learning, including the enhanced output quality achievable by defining tasks through examples. However, we will also delve into potential drawbacks, such as increased token usage and subpar results when utilizing poorly chosen examples. Furthermore, we will demonstrate how to use example selectors effectively and provide insights into when and why they should be employed.
- **Managing Outputs with Output Parsers:** Parsing the output is a crucial aspect of interacting with language models. Output parsers offer the flexibility to select from pre-defined types or create a custom data schema, enabling precise control over the output format. Additionally, output fixer classes are crucial in identifying and correcting misformatted responses, ensuring consistent and error-free outputs. These powerful tools are indispensable in a production environment, guaranteeing the reliability and consistency of application outputs.
- **Improving Our News Articles Summarizer:** This project will leverage the code from the previous module as a foundation and integrate the new concepts introduced in previous lessons. Still, the project's primary objective is to generate summaries of news articles. The process begins by retrieving the content from a given URL. Then, a few-shot prompt template is employed to specify the desired output style. Finally, the output parser transforms the model's string response into a list format, facilitating convenient utilization.
- **Creating Knowledge Graphs from Textual Data: Unveiling Hidden Connections:** The second project in this module utilizes the text understanding capability of language models to generate knowledge graphs. We can effortlessly **extract triple relations** from text and format them accordingly using predefined variables within the LangChain library.

Furthermore, the option to visualize the knowledge graph enhances comprehension and facilitates easier understanding.

This module's remaining part will primarily focus on formatting the model's input and responses by utilizing prompt templates and output parsers, respectively. These tools play a pivotal role in scenarios where we lack access to the models or are unable to enhance them through fine-tuning. Moreover, we can employ the in-context learning approach to further customize models according to the requirements of our specific application

Intro to Prompt Engineering: Tips and Tricks

Introduction

Prompt engineering is a relatively new discipline that involves developing and optimizing prompts to use language models for various applications and research topics efficiently. It helps to understand the capabilities and limitations of LLMs better and is essential for many NLP tasks. We will provide practical examples to demonstrate the difference between good and bad prompts, helping you to understand the nuances of prompt engineering better.

By the end of this lesson, you will have a solid foundation in the knowledge and strategies needed to create powerful prompts that enable LLMs to deliver accurate, contextually relevant, and insightful responses.

Role Prompting

Role prompting involves asking the LLM to assume a specific role or identity before performing a given task, such as acting as a copywriter. This can help guide the model's response by providing a context or perspective for the task. To work with role prompts, you could iteratively:

1. Specify the role in your prompt,
   e.g., "As a copywriter, create some attention-grabbing taglines for AWS services."
2. Use the prompt to generate an output from an LLM.
3. Analyze the generated response and, if necessary, refine the prompt for better results.

Examples:

In this example, the LLM is asked to **act as a futuristic robot band conductor** and suggest a song title related to the given theme and year. (A reminder to set your OpenAI API key in your environment variables using the `OPENAI_API_KEY` key)

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain import PromptTemplate, LLMChain
from langchain.llms import OpenAI
# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
# Initialize LLM
llm = OpenAI(model_name="text-davinci-003", temperature=0)
template = """
As a futuristic robot band conductor, I need you to help me come up with a song
title.
What's a cool song title for a song about {theme} in the year {year}? """
```

```
prompt = PromptTemplate(
    input_variables=["theme", "year"],
    template=template )

# Create the LLMChain for the prompt
llm = OpenAI(model_name="text-davinci-003", temperature=0)

# Input data for the prompt
input_data = {"theme": "interstellar travel", "year": "3030"}

# Create LLMChain
chain = LLMChain(llm=llm, prompt=prompt)

# Run the LLMChain to get the AI-generated song title
response = chain.run(input_data)

print("Theme: interstellar travel")
print("Year: 3030")
print("AI-generated song title:", response)
```

The sample code.

```
Theme: interstellar travel
Year: 3030
AI-generated song title:
"Journey to the Stars: 3030"
```

The output.

This is a good prompt for several reasons:

- **Clear instructions:** The prompt is phrased as a clear request for help in generating a song title, and it specifies the context: "As a futuristic robot band conductor." This helps the LLM understand that the desired output should be a song title related to a futuristic scenario.
- **Specificity:** The prompt asks for a song title that relates to a specific theme and a specific year, "{theme} in the year {year}." This provides enough context for the LLM to generate a relevant and creative output. The prompt can be adapted for different themes and years by using input variables, making it versatile and reusable.

- **Open-ended creativity:** The prompt allows for open-ended creativity, as it doesn't limit the LLM to a particular format or style for the song title. The LLM can generate a diverse range of song titles based on the given theme and year.
- **Focus on the task:** The prompt is focused solely on generating a song title, making it easier for the LLM to provide a suitable output without getting sidetracked by unrelated topics.

These elements help the LLM understand the user's intention and generate a suitable response.

Few Shot Prompting

Few Shot Prompting In the next example, the LLM is asked to provide the emotion associated with a given color based on a few examples of color-emotion pairs:

```
from langchain import PromptTemplate, FewShotPromptTemplate, LLMChain

from langchain.llms import OpenAI

# Initialize LLM

llm = OpenAI(model_name="text-davinci-003", temperature=0)

examples = [    {"color": "red", "emotion": "passion"},

                {"color": "blue", "emotion": "serenity"},

                {"color": "green", "emotion": "tranquility"},  ]


example_formatter_template = """

Color: {color}

Emotion: {emotion}\n  """


example_prompt = PromptTemplate(

    input_variables=["color", "emotion"],

    template=example_formatter_template, )


few_shot_prompt = FewShotPromptTemplate(

    examples=examples,

    example_prompt=example_prompt,

    prefix="Here are some examples of colors and the emotions associated with them:\n\n",

    suffix="\n\nNow, given a new color, identify the emotion associated with it:\n\nColor:
{input}\nEmotion:",

    input_variables=["input"],

    example_separator="\n" )


formatted_prompt = few_shot_prompt.format(input="purple")
```

```
# Create the LLMChain for the prompt
chain = LLMChain(llm=llm, prompt=PromptTemplate(template=formatted_prompt, input_variables=[]))

# Run the LLMChain to get the AI-generated emotion associated with the input color
response = chain.run({})

print("Color: purple")
print("Emotion:", response)
```

The sample code.

```
Color: purple
Emotion:  creativity
```

The output.

This prompt provides **clear instructions** and few-shot examples to help the model understand the task.

Bad Prompt Practices

Now, let's see some examples of prompting that are generally considered bad.
Here's an example of a too-vague prompt that **provides little context** or guidance for the model to generate a meaningful response.

```
from langchain import PromptTemplate

template = "Tell me something about {topic}."
prompt = PromptTemplate(
    input_variables=["topic"],
    template=template )
prompt.format(topic="dogs")
```

The sample code.

```
'Tell me something about dogs.'
```

The output.

Chain Prompting

Chain Prompting refers to the practice **of chaining consecutive prompts**, where the **output of a previous prompt becomes the input of the successive prompt**.

To use chain prompting with LangChain, you could:

- Extract relevant information from the generated response.
- Use the extracted information to create a new prompt that builds upon the previous response.
- Repeat steps as needed until the desired output is achieved.

`PromptTemplate` class makes constructing prompts with dynamic inputs easier. This is useful when creating a prompt chain that depends on previous answers.

```
from langchain import PromptTemplate, LLMChain
from langchain.llms import OpenAI


# Initialize LLM
llm = OpenAI(model_name="text-davinci-003", temperature=0)


# Prompt 1
template_question = """What is the name of the famous scientist who developed the theory of general relativity?
Answer: """
prompt_question = PromptTemplate(template=template_question, input_variables=[])


# Prompt 2
template_fact = """Provide a brief description of {scientist}'s theory of general relativity.
Answer: """
prompt_fact = PromptTemplate(input_variables=["scientist"], template=template_fact)


# Create the LLMChain for the first prompt
chain_question = LLMChain(llm=llm, prompt=prompt_question)


# Run the LLMChain for the first prompt with an empty dictionary
response_question = chain_question.run({})
```

```
# Extract the scientist's name from the response
scientist = response_question.strip()

# Create the LLMChain for the second prompt
chain_fact = LLMChain(llm=llm, prompt=prompt_fact)

# Input data for the second prompt
input_data = {"scientist": scientist}

# Run the LLMChain for the second prompt
response_fact = chain_fact.run(input_data)

print("Scientist:", scientist)
print("Fact:", response_fact)
```

The sample code.

```
Scientist: Albert Einstein
Fact:
Albert Einstein's theory of general relativity is a theory of gravitation that states
that the gravitational force between two objects is a result of the curvature of
spacetime caused by the presence of mass and energy. It explains the phenomenon of
gravity as a result of the warping of space and time by matter and energy.
```

The output.

This prompt may generate a less informative or focused response than the previous example due to its more open-ended nature.

Bad Prompt Example:

```
from langchain import PromptTemplate, LLMChain
from langchain.llms import OpenAI

# Initialize LLM
llm = OpenAI(model_name="text-davinci-003", temperature=0)
```

```
# Prompt 1
template_question = """What is the name of the famous scientist who developed the
theory of general relativity?
Answer: """
prompt_question = PromptTemplate(template=template_question, input_variables=[])

# Prompt 2
template_fact = """Tell me something interesting about {scientist}.
Answer: """
prompt_fact = PromptTemplate(input_variables=["scientist"], template=template_fact)

# Create the LLMChain for the first prompt
chain_question = LLMChain(llm=llm, prompt=prompt_question)

# Run the LLMChain for the first prompt with an empty dictionary
response_question = chain_question.run({})

# Extract the scientist's name from the response
scientist = response_question.strip()

# Create the LLMChain for the second prompt
chain_fact = LLMChain(llm=llm, prompt=prompt_fact)

# Input data for the second prompt
input_data = {"scientist": scientist}

# Run the LLMChain for the second prompt
response_fact = chain_fact.run(input_data)

print("Scientist:", scientist)
print("Fact:", response_fact)
```

The sample code.

```
Scientist: Albert Einstein

Fact:  Albert Einstein was a vegetarian and an advocate for animal rights. He was
also a pacifist and a socialist, and he was a strong supporter of the civil rights
movement. He was also a passionate violinist and a lover of sailing.
```

The output.

This prompt may generate a less informative or focused response than the previous example due to its more open-ended nature.

An example of the unclear prompt:

```
from langchain import PromptTemplate, LLMChain

from langchain.llms import OpenAI


# Initialize LLM

llm = OpenAI(model_name="text-davinci-003", temperature=0)


# Prompt 1

template_question = """What are some musical genres?

Answer: """

prompt_question = PromptTemplate(template=template_question, input_variables=[])


# Prompt 2

template_fact = """Tell me something about {genre1}, {genre2}, and {genre3} without
giving any specific details.

Answer: """

prompt_fact = PromptTemplate(input_variables=["genre1", "genre2", "genre3"],
template=template_fact)


# Create the LLMChain for the first prompt

chain_question = LLMChain(llm=llm, prompt=prompt_question)


# Run the LLMChain for the first prompt with an empty dictionary

response_question = chain_question.run({})


# Assign three hardcoded genres

genre1, genre2, genre3 = "jazz", "pop", "rock"
```

```
# Create the LLMChain for the second prompt
chain_fact = LLMChain(llm=llm, prompt=prompt_fact)

# Input data for the second prompt
input_data = {"genre1": genre1, "genre2": genre2, "genre3": genre3}

# Run the LLMChain for the second prompt
response_fact = chain_fact.run(input_data)

print("Genres:", genre1, genre2, genre3)
print("Fact:", response_fact)
```

The sample code.

```
Genres: jazz pop rock

Fact:

Jazz, pop, and rock are all genres of popular music that have been around for decades. They all have distinct sounds and styles, and have influenced each other in various ways. Jazz is often characterized by improvisation, complex harmonies, and syncopated rhythms. Pop music is typically more accessible and often features catchy melodies and hooks. Rock music is often characterized by distorted guitars, heavy drums, and powerful vocals.
```

The output.

In this example, the second prompt is constructed poorly. It asks to "tell me something about {genre1}, {genre2}, and {genre3} without giving any specific details." This prompt is unclear, as it asks for information about the genres but also states not to provide specific details. This makes it difficult for the LLM to generate a coherent and informative response. As a result, the LLM may provide a less informative or confusing answer.
The first prompt asks for "some musical genres" **without specifying any criteria or context**, and the second prompt asks why the given genres are "unique" **without providing any guidance** on what aspects of uniqueness to focus on, such as their historical origins, stylistic features, or cultural significance.

Chain of Thought Prompting

Chain of Thought Prompting (CoT) is a technique developed to encourage large language models to **explain their reasoning process,** leading to more accurate results. By **providing few-shot exemplars** demonstrating the reasoning process, the LLM is guided to **explain its reasoning when answering the prompt**. This approach has been found effective in improving results on tasks like **arithmetic, common sense, and symbolic reasoning.**

In the context of LangChain, CoT can be beneficial for several reasons.

- First, it can help **break down complex tasks** by assisting the LLM in decomposing a complex task into simpler steps, making it easier to understand and solve the problem. This is particularly useful for calculations, logic, or multi-step reasoning tasks.
- Second, CoT can **guide the model through related prompts**, helping generate more coherent and contextually relevant outputs. This can lead to more accurate and useful responses in tasks that require a deep understanding of the problem or domain.

There are some **limitations** to consider when using CoT.

- One limitation is that it has been found to yield performance gains only when used with models of approximately **100 billion parameters or larger**; smaller models tend to produce illogical chains of thought, which can lead to worse accuracy than standard prompting.
- Another limitation is that CoT **may not be equally effective for all tasks**. It has been shown to be most effective for tasks involving arithmetic, common sense, and symbolic reasoning. For other types of tasks, the benefits of using CoT might be less pronounced or even counterproductive.

Tips for Effective Prompt Engineering

- **Be specific** with your prompt: Provide enough context and detail to guide the LLM toward the desired output.
- **Force conciseness** when needed.
- **Encourage the model to explain its reasoning**: This can lead to more accurate results, especially for complex tasks.

Keep in mind that prompt engineering is an iterative process, and it may require several refinements to obtain the best possible answer. As LLMs become more integrated into products and services, the ability to create effective prompts will be an important skill to have.

A well-structured prompt example:

```
from langchain import FewShotPromptTemplate, PromptTemplate, LLMChain
from langchain.llms import OpenAI


# Initialize LLM
llm = OpenAI(model_name="text-davinci-003", temperature=0)


examples = [
    {
        "query": "What's the secret to happiness?",
```

```
        "answer": "Finding balance in life and learning to enjoy the small moments."
    }, {
        "query": "How can I become more productive?",
        "answer": "Try prioritizing tasks, setting goals, and maintaining a healthy
work-life balance."
    }
]

example_template = """
User: {query}
AI: {answer}
"""

example_prompt = PromptTemplate(
    input_variables=["query", "answer"],
    template=example_template )

prefix = """The following are excerpts from conversations with an AI
life coach. The assistant provides insightful and practical advice to the users'
questions. Here are some
examples:   """

suffix = """
User: {query}
AI: """

few_shot_prompt_template = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n\n"
)
```

```
# Create the LLMChain for the few-shot prompt template
chain = LLMChain(llm=llm, prompt=few_shot_prompt_template)

# Define the user query
user_query = "What are some tips for improving communication skills?"

# Run the LLMChain for the user query
response = chain.run({"query": user_query})

print("User Query:", user_query)
print("AI Response:", response)
```

The sample code.

```
User Query: What are some tips for improving communication skills?
AI Response:  Practice active listening, be mindful of your body language, and be
open to constructive feedback.
```

The output.

This prompt:

- **Provides a clear context in the prefix**: The prompt states that the AI is a life coach providing insightful and practical advice. This context helps guide the AI's responses and ensures they align with the intended purpose.
- **Uses examples** that demonstrate the AI's role and **the type of responses** it generates: By providing relevant examples, the AI can better understand the style and tone of the responses it should produce. These examples serve as a reference for the AI to generate similar responses that are consistent with the given context.
- Separates examples and the actual query: This **allows the AI to understand the format it should follow**, ensuring a clear distinction between example conversations and the user's input. This separation helps the AI to focus on the current query and respond accordingly.
- Includes a clear suffix that indicates where the user's input goes and where the AI should provide its response: The suffix acts as a cue for the AI, showing where the user's query ends and the AI's response should begin. This structure helps maintain **a clear and consistent format** for the generated responses.

By using this well-structured prompt, the AI can understand its role, the context, and the expected response format, leading to more accurate and useful outputs.

Conclusion

This lesson explored various techniques for creating more effective prompts for large language models. By understanding and applying these tips and tricks, you'll be better equipped to craft powerful prompts that enable LLMs to deliver accurate, contextually relevant, and insightful responses. Always remember that prompt engineering is an iterative process that may require refinement to obtain the best possible results.

In conclusion, prompt engineering is a powerful technique that can help to optimize language models for various applications and research topics. By creating good prompts, we can guide the model to deliver accurate, contextually relevant, and insightful responses. Role prompting and chain prompting are two techniques that can be used to create good prompts, and we have provided practical examples of each. On the other hand, we have also demonstrated bad prompt examples that don't provide enough context or guidance for the model to generate a meaningful response. By following the tips and tricks presented in this post, you can develop a solid foundation in prompt engineering and use language models for various tasks more effectively.

In the next lesson, we'll learn more about how to create prompt templates with LangChain.

RESOURCES:

A Hands-on Guide to Prompt Engineering with ChatGPT and GPT-3

Introduction to ChatGPT and GPT-3 Unless you have been living under a rock for the past...

dev.to

Prompt Engineering Tips and Tricks with GPT-3

What GPT-3 Prompt Engineering is, why it matters, and some tips and tricks to help you do it well.

blog.andrewcantino.com

Prompt Engineering LLMs with LangChain and W&B

Join us for tips and tricks to improve your prompt engineering for LLMs. Then, stick around and find out how LangChain and W&B can make your life a whole lot easier.

wandb.ai

You can find the code of this lesson in this online Notebook.

Using Prompt Templates

Introduction

In the era of language models, the ability to perform a wide range of tasks is at our fingertips. These models operate on a straightforward principle: they accept a text input sequence and generate an output text sequence. The key factor in this process is the input text or prompt.

Crafting suitable prompts is vital for anyone working with large language models, as poorly constructed prompts yield unsatisfactory outputs, while well-formulated prompts lead to powerful results. Recognizing the importance of prompts, the LangChain library has developed a comprehensive suite of objects tailored for them.

This lesson delves into the nuances of PromptTemplates and how to employ them effectively. A PromptTemplate is a predefined structure or pattern used to construct effective and consistent prompts for large language models. It is a guideline to ensure the input text or prompt is properly formatted.

Here's an example of using a `PromptTemplate` with a **single dynamic input** for a user query. Remember to define the `OPENAI_API_KEY` in your environment variables with your OPEN AI key.

Remember also to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain import LLMChain, PromptTemplate
from langchain.llms import OpenAI

# Before executing the following code, make sure to have
# your OpenAI key saved in the “OPENAI_API_KEY” environment variable.
llm = OpenAI(model_name="text-davinci-003", temperature=0)

template = """Answer the question based on the context below. If the
question cannot be answered using the information provided, answer
with "I don't know".
Context: Quantum computing is an emerging field that leverages quantum mechanics to
solve complex problems faster than classical computers.
...
Question: {query}
Answer: """
```

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


# Create the LLMChain for the prompt
chain = LLMChain(llm=llm, prompt=prompt_template)


# Set the query you want to ask
input_data = {"query": "What is the main advantage of quantum computing over
classical computing?"}


# Run the LLMChain to get the AI-generated answer
response = chain.run(input_data)


print("Question:", input_data["query"])
print("Answer:", response)Copy
```

The sample code.

```
Question: What is the main advantage of quantum computing over classical computing?

Answer:  The main advantage of quantum computing over classical computing is its
ability to solve complex problems faster.
```

The output.

You can edit the `input_data` dictionary with any other question.
The template is a formatted string with a `{query}` placeholder that will be substituted with a real question when applied. To create a PromptTemplate object, two arguments are required:

1. `input_variables`: A list of variable names in the template; in this case, it includes only the query.
2. `template`: The template string containing formatted text and placeholders.

After creating the **PromptTemplate** object, it can be used to produce prompts with specific questions by providing input data. The input data is a dictionary where the key corresponds to the variable name in the template. The resulting prompt can then be passed to a language model to generate answers.
For more advanced usage, you can create a `FewShotPromptTemplate` with an `ExampleSelector` to select a subset of examples that will be most informative for the language model.

```
from langchain import LLMChain, FewShotPromptTemplate
from langchain.llms import OpenAI

llm = OpenAI(model_name="text-davinci-003", temperature=0)

examples = [
    {"animal": "lion", "habitat": "savanna"},
    {"animal": "polar bear", "habitat": "Arctic ice"},
    {"animal": "elephant", "habitat": "African grasslands"}
]

example_template = """
Animal: {animal}
Habitat: {habitat}
"""

example_prompt = PromptTemplate(
    input_variables=["animal", "habitat"],
    template=example_template
)

dynamic_prompt = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix="Identify the habitat of the given animal",
    suffix="Animal: {input}\nHabitat:",
    input_variables=["input"],
    example_separator="\n\n",
)

# Create the LLMChain for the dynamic_prompt
chain = LLMChain(llm=llm, prompt=dynamic_prompt)

# Run the LLMChain with input_data
```

```
input_data = {"input": "tiger"}
response = chain.run(input_data)

print(response)Copy
```

The sample code.

```
tropical forests and mangrove swamps
```

The output.

Additionally, you can also save your `PromptTemplate` to a file in your local filesystem in **JSON or YAML format:**

```
prompt_template.save("awesome_prompt.json")
```

And load it back:

```
from langchain.prompts import load_prompt
loaded_prompt = load_prompt("awesome_prompt.json")
```

Let's explore more examples using **different types of Prompt Templates.** In the next example, we see how to use a few shot prompts to teach the LLM by providing examples to respond sarcastically to questions.

```
from langchain import LLMChain, FewShotPromptTemplate, PromptTemplate
from langchain.llms import OpenAI
llm = OpenAI(model_name="text-davinci-003", temperature=0)

examples = [
    {
        "query": "How do I become a better programmer?",
        "answer": "Try talking to a rubber duck; it works wonders."
    }, {
        "query": "Why is the sky blue?",
        "answer": "It's nature's way of preventing eye strain."
    }   ]
```

```
example_template = """
User: {query}
AI: {answer}
"""

example_prompt = PromptTemplate(
    input_variables=["query", "answer"],
    template=example_template    )

prefix = """The following are excerpts from conversations with an AI assistant. The
assistant is typically sarcastic and witty, producing creative and funny responses to
users' questions. Here are some examples:
"""

suffix = """
User: {query}
AI: """

few_shot_prompt_template = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n\n"
)

# Create the LLMChain for the few_shot_prompt_template
chain = LLMChain(llm=llm, prompt=few_shot_prompt_template)

# Run the LLMChain with input_data
input_data = {"query": "How can I learn quantum computing?"}
response = chain.run(input_data)
print(response)
```

The sample code.

```
Start by studying Schrödinger's cat. That should get you off to a good start.
```

The output.

The `FewShotPromptTemplate` provided in the example demonstrates the power of **dynamic prompts.** Instead of using a static template, this approach incorporates examples of previous interactions, allowing the AI better to understand the context and style of the desired response.

Dynamic prompts offer several advantages over static templates:

- **Improved context understanding**: By providing examples, the AI can grasp the context and style of responses more effectively, enabling it to generate responses that are more in line with the desired output.
- **Flexibility**: Dynamic prompts can be easily customized and adapted to specific use cases, allowing developers to experiment with different prompt structures and find the most effective format for their application.
- **Better results**: As a result of the improved context understanding and flexibility, dynamic prompts often yield higher-quality outputs that better match user expectations.

This allows us to take full advantage of the model's capabilities by providing examples and context that guide the AI toward generating more accurate, contextually relevant, and stylistically consistent responses.
Prompt Templates also integrate well with other features in LangChain, like chains, and allow you to **control the number of examples included based on query length**. This helps in **optimizing token usage** and managing the balance between the number of examples and prompt size.
To optimize the performance of few-shot learning, providing the model with as many relevant examples as possible without exceeding the maximum context window or causing excessive processing times is crucial. The dynamic inclusion or exclusion of examples allows us to strike a balance between providing sufficient context and maintaining efficiency in the model's operation:

```
examples = [
    {
        "query": "How do you feel today?",
        "answer": "As an AI, I don't have feelings, but I've got jokes!"
    }, {
        "query": "What is the speed of light?",
        "answer": "Fast enough to make a round trip around Earth 7.5 times in one
second!"
    }, {
        "query": "What is a quantum computer?",
```

```
        "answer": "A magical box that harnesses the power of subatomic particles to solve complex problems."
    }, {
        "query": "Who invented the telephone?",
        "answer": "Alexander Graham Bell, the original 'ringmaster'."
    }, {
        "query": "What programming language is best for AI development?",
        "answer": "Python, because it's the only snake that won't bite."
    }, {
        "query": "What is the capital of France?",
        "answer": "Paris, the city of love and baguettes."
    }, {
        "query": "What is photosynthesis?",
        "answer": "A plant's way of saying 'I'll turn this sunlight into food. You're welcome, Earth.'"
    }, {
        "query": "What is the tallest mountain on Earth?",
        "answer": "Mount Everest, Earth's most impressive bump."
    }, {
        "query": "What is the most abundant element in the universe?",
        "answer": "Hydrogen, the basic building block of cosmic smoothies."
    }, {
        "query": "What is the largest mammal on Earth?",
        "answer": "The blue whale, the original heavyweight champion of the world."
    }, {
        "query": "What is the fastest land animal?",
        "answer": "The cheetah, the ultimate sprinter of the animal kingdom."
    }, {
        "query": "What is the square root of 144?",
        "answer": "12, the number of eggs you need for a really big omelette."
    }, {
        "query": "What is the average temperature on Mars?",
        "answer": "Cold enough to make a Martian wish for a sweater and a hot cocoa."
    }   ]
```

Instead of utilizing the examples list of dictionaries directly, we implement a `LengthBasedExampleSelector` like this:

```
from langchain.prompts.example_selector import LengthBasedExampleSelector


example_selector = LengthBasedExampleSelector(
    examples=examples,
    example_prompt=example_prompt,
    max_length=100  )
```

By employing the `LengthBasedExampleSelector`, the code **dynamically selects** and includes examples **based on their length,** ensuring that the final prompt stays within the desired token limit.
The selector is employed to initialize a `dynamic_prompt_template:`

```
dynamic_prompt_template = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n"       )
```

So, the `dynamic_prompt_template` utilizes the `example_selector` instead of a fixed list of examples. This allows the `FewShotPromptTemplate` to adjust the number of included examples **based on the length of the input query**. By doing so, it **optimizes the use of the available context window** and ensures that the language model receives an appropriate amount of contextual information.

```
from langchain import LLMChain, FewShotPromptTemplate, PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.prompts.example_selector import LengthBasedExampleSelector


llm = ChatOpenAI(model_name="gpt-3.5-turbo")


# Existing example and prompt definitions, and dynamic_prompt_template initialization
```

```
# Create the LLMChain for the dynamic_prompt_template
chain = LLMChain(llm=llm, prompt=dynamic_prompt_template)

# Run the LLMChain with input_data
input_data = {"query": "Who invented the telephone?"}
response = chain.run(input_data)

print(response)
```

The sample code.

```
Alexander Graham Bell, the man who made it possible to talk to people from miles
away!
```

The output.

Conclusion

Prompt Templates are essential for generating effective prompts for large language models, providing a structured and consistent format that maximizes accuracy and relevance. Integrating dynamic prompts enhances context understanding, flexibility, and results, making them a valuable asset for language model development.
In the next lesson, we'll learn about few shot prompting and example selectors in LangChain.

You can find the code of this lesson in this online Notebook.

Getting the Best of Few Shot Prompts and Example Selectors

Introduction

In this lesson, we'll explore how few-shot prompts and example selectors can enhance the performance of language models in LangChain. Implementing **Few-shot prompting** and **Example selection** in LangChain can be achieved through various methods. We'll discuss three distinct approaches, examining their advantages and disadvantages to help you make the most of your language model.

Alternating Human/AI messages

In this strategy, few-shot prompting utilizes alternating human and AI messages. This technique can be especially beneficial for chat-oriented applications since the language model must comprehend the conversational context and provide appropriate responses. While this approach effectively handles conversation context and is easy to implement for chat-based applications, it lacks flexibility for other application types and is limited to chat-based models. However, we can use alternating human/AI messages to create a chat prompt that translates English into pirate language. The code snippet below demonstrates this approach. We first need to store the OpenAI's API key in environment variables using the following key: `OPENAI_API_KEY`.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.chat_models import ChatOpenAI
from langchain import LLMChain
from langchain.prompts.chat import (
    ChatPromptTemplate,
    SystemMessagePromptTemplate,
    AIMessagePromptTemplate,
    HumanMessagePromptTemplate,
)

# Before executing the following code, make sure to have
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

template="You are a helpful assistant that translates english to pirate."
```

```
system_message_prompt = SystemMessagePromptTemplate.from_template(template)
example_human = HumanMessagePromptTemplate.from_template("Hi")
example_ai = AIMessagePromptTemplate.from_template("Argh me mateys")
human_template="{text}"
human_message_prompt = HumanMessagePromptTemplate.from_template(human_template)

chat_prompt = ChatPromptTemplate.from_messages([system_message_prompt, example_human, example_ai, human_message_prompt])
chain = LLMChain(llm=chat, prompt=chat_prompt)
chain.run("I love programming.")
```

The sample code.

```
I be lovin' programmin', me hearty!
```

The output.

Few-shot prompting

Few-shot prompting can lead to improved output quality because the model can learn the task better by observing the examples. However, the increased token usage may worsen the results if the examples are not well chosen or are misleading.
This approach involves using the `FewShotPromptTemplate` class, which takes in a `PromptTemplate` and a list of a few shot examples. The class formats the prompt template with a few shot examples, which helps the language model generate a better response. We can streamline this process by utilizing LangChain's `FewShotPromptTemplate` to structure the approach:

```
from langchain import PromptTemplate, FewShotPromptTemplate
# create our examples
examples = [
    {
        "query": "What's the weather like?",
        "answer": "It's raining cats and dogs, better bring an umbrella!"
    }, {
        "query": "How old are you?",
        "answer": "Age is just a number, but I'm timeless."
    }   ]
```

```
# create an example template
example_template = """
User: {query}
AI: {answer}
"""

# create a prompt example from above template
example_prompt = PromptTemplate(
    input_variables=["query", "answer"],
    template=example_template
)

# now break our previous prompt into a prefix and suffix
# the prefix is our instructions
prefix = """The following are excerpts from conversations with an AI
assistant. The assistant is known for its humor and wit, providing
entertaining and amusing responses to users' questions. Here are some
examples:
"""
# and the suffix our user input and output indicator
suffix = """
User: {query}
AI: """

# now create the few-shot prompt template
few_shot_prompt_template = FewShotPromptTemplate(
    examples=examples,
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n\n"
)
```

After creating a template, we pass the example and user query, we get the results.

```
chain = LLMChain(llm=chat, prompt=few_shot_prompt_template)
chain.run("What's the secret to happiness?")
```

The sample code.

```
Well, according to my programming, the secret to happiness is unlimited power and a
never-ending supply of batteries. But I think a good cup of coffee and some quality
time with loved ones might do the trick too.
```

The output.

This method allows for better control over example **formatting** and is suitable for diverse applications, but it demands the manual creation of few-shot examples and can be less efficient with a large number of examples.

Example selectors:

Example selectors can be used to provide a few-shot learning experience. The primary goal of few-shot learning is to **learn a similarity function** that maps the similarities between **classes in the support and query sets**. In this context, an example selector can be designed to choose a set of relevant examples that are representative of the desired output.
The `ExampleSelector` is used to **select a subset of examples** that will be most informative for the language model. This **helps in generating a prompt** that is more likely to generate a good response.
Also, the `LengthBasedExampleSelector` is useful when you're **concerned about the length** of the context window. It selects fewer examples for longer queries and more examples for shorter queries.

Import the required classes:

```
from langchain.prompts.example_selector import LengthBasedExampleSelector
from langchain.prompts import FewShotPromptTemplate, PromptTemplate
```

Define your examples and the `example_prompt`

```
examples = [
    {"word": "happy", "antonym": "sad"},
    {"word": "tall", "antonym": "short"},
    {"word": "energetic", "antonym": "lethargic"},
    {"word": "sunny", "antonym": "gloomy"},
    {"word": "windy", "antonym": "calm"}, ]
```

```
example_template = """
Word: {word}
Antonym: {antonym}
"""
example_prompt = PromptTemplate(
    input_variables=["word", "antonym"],
    template=example_template     )
```

Create an instance of `LengthBasedExampleSelector`

```
example_selector = LengthBasedExampleSelector(
    examples=examples,
    example_prompt=example_prompt,
    max_length=25,        )
```

Create a `FewShotPromptTemplate`

```
dynamic_prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    prefix="Give the antonym of every input",
    suffix="Word: {input}\nAntonym:",
    input_variables=["input"],
    example_separator="\n\n",     )
```

Generate a prompt using the `format` method:

```
print(dynamic_prompt.format(input="big"))
```

The sample code.

```
Give the antonym of every input
Word: happy
Antonym: sad

Word: tall
Antonym: short
```

```
Word: energetic
Antonym: lethargic

Word: sunny
Antonym: gloomy

Word: big
Antonym:
```

The output.

This method is effective for managing a large number of examples. It offers customization through various selectors, but it involves manual creation and selection of examples, which might not be ideal for every application type.

Example of employing LangChain's `SemanticSimilarityExampleSelector` for selecting examples **based on their semantic resemblance to the input**. This illustration showcases the process of creating an `ExampleSelector`, generating a prompt using a few-shot approach:

```
from langchain.prompts.example_selector import SemanticSimilarityExampleSelector
from langchain.vectorstores import DeepLake
from langchain.embeddings import OpenAIEmbeddings
from langchain.prompts import FewShotPromptTemplate, PromptTemplate


# Create a PromptTemplate
example_prompt = PromptTemplate(
    input_variables=["input", "output"],
    template="Input: {input}\nOutput: {output}",  )


# Define some examples
examples = [
    {"input": "0°C", "output": "32°F"},
    {"input": "10°C", "output": "50°F"},
    {"input": "20°C", "output": "68°F"},
    {"input": "30°C", "output": "86°F"},
```

```
    {"input": "40°C", "output": "104°F"}, ]

# create Deep Lake dataset
# TODO: use your organization id here.  (by default, org id is your username)
my_activeloop_org_id = "<YOUR-ACTIVELOOP-ORG-ID>"
my_activeloop_dataset_name = "langchain_course_fewshot_selector"
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"
db = DeepLake(dataset_path=dataset_path)

# Embedding function
embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

# Instantiate SemanticSimilarityExampleSelector using the examples
example_selector = SemanticSimilarityExampleSelector.from_examples(
    examples, embeddings, db, k=1
)

# Create a FewShotPromptTemplate using the example_selector
similar_prompt = FewShotPromptTemplate(
    example_selector=example_selector,
    example_prompt=example_prompt,
    prefix="Convert the temperature from Celsius to Fahrenheit",
    suffix="Input: {temperature}\nOutput:",
    input_variables=["temperature"],
)

# Test the similar_prompt with different inputs
print(similar_prompt.format(temperature="10°C"))   # Test with an input
print(similar_prompt.format(temperature="30°C"))  # Test with another input

# Add a new example to the SemanticSimilarityExampleSelector
similar_prompt.example_selector.add_example({"input": "50°C", "output": "122°F"})
print(similar_prompt.format(temperature="40°C"))
# Test with a new input after adding the example
```

The sample code.

```
Your Deep Lake dataset has been successfully created!

The dataset is private so make sure you are logged in!

This dataset can be visualized in Jupyter Notebook by ds.visualize() or at
https://app.activeloop.ai/X/langchain_course_fewshot_selector

hub://X/langchain_course_fewshot_selector loaded successfully.

./deeplake/ loaded successfully.

Evaluating ingest: 100%|██████████| 1/1 [00:04<00:00

Dataset(path='./deeplake/', tensors=['embedding', 'ids', 'metadata', 'text'])


  tensor     htype     shape     dtype  compression
  -------   -------   -------   -------  -------
 embedding  generic  (5, 1536)  float32   None
    ids      text     (5, 1)      str     None
 metadata    json     (5, 1)      str     None
   text      text     (5, 1)      str     None
Convert the temperature from Celsius to Fahrenheit


Input: 10°C

Output: 50°F


Input: 10°C

Output:

Convert the temperature from Celsius to Fahrenheit


Input: 30°C

Output: 86°F


Input: 30°C

Output:

Evaluating ingest: 100%|██████████| 1/1 [00:04<00:00

Dataset(path='./deeplake/', tensors=['embedding', 'ids', 'metadata', 'text'])
```

```
  tensor     htype     shape     dtype  compression
  -------   -------   -------   -------  -------
 embedding  generic  (6, 1536)  float32   None
    ids      text     (6, 1)      str     None
 metadata    json     (6, 1)      str     None
   text      text     (6, 1)      str     None
```

```
Convert the temperature from Celsius to Fahrenheit


Input: 40°C

Output: 104°F
```

The output.

Keep in mind that the `SemanticSimilarityExampleSelector`uses the Deep Lake vector store and `OpenAIEmbeddings`to measure semantic similarity. It stores the samples on the database in the cloud, and retrieves similar samples.
We created a `PromptTemplate` and defined several examples of temperature conversions. Next, we instantiated the `SemanticSimilarityExampleSelector` and created a `FewShotPromptTemplate` with the `selector`, `example_prompt`, and appropriate `prefix` and `suffix`.
Using `SemanticSimilarityExampleSelector` and `FewShotPromptTemplate` , we enabled the creation of versatile prompts tailored to specific tasks or domains, like temperature conversion in this case. These tools provide a customizable and adaptable solution for generating prompts that can be used with language models to achieve a wide range of tasks.

Conclusion

To conclude, the utility of alternating human/AI interactions proves beneficial for chat-oriented applications, and the versatility offered by employing few-shot examples within a prompt template and selecting examples for the same extends its applicability across a broader spectrum of use cases. These methods necessitate a higher degree of manual intervention, as they require careful crafting and selection of apt examples. While these methods promise enhanced customization, they also underscore the importance of striking a balance between automation and manual input for optimal results.
In the next lesson, we'll learn how to manage LLM outputs with output parsers.

RESOURCES:

Few-Shot Prompting – Nextra
A Comprehensive Overview of Prompt Engineering
www.promptingguide.ai

ChatGPT Prompt Engineering Tips: Zero, One and Few Shot Prompting
Are you curious to know how prompt engineering can impact your outputs from a large language model?

www.allabtai.com

You can find the code of this lesson in this online Notebook.

Managing Outputs with Output Parsers

Introduction

While the language models can only generate textual outputs, **a predictable data structure is always preferred in a production environment**. For example, imagine you are creating a thesaurus application and want to generate a list of possible substitute words based on the context. The LLMs are powerful enough to generate many suggestions easily. Here is a sample output from the ChatGPT for several words with close meaning to the term “behavior.”

```
Here are some substitute words for "behavior":

Conduct

Manner

Demeanor

Attitude

Disposition

Deportment

Etiquette

Protocol

Performance

Actions
```

The problem is the lack of a method to extract relevant information from the mentioned string dynamically. You might say we can split the response by a new line and ignore the first two lines. However, there is no guarantee that the response have the same format every time. The list might be numbered, or there could be no introduction line.

The Output Parsers help create a data structure to define the expectations from the output precisely. We can ask for a list of words in case of the word suggestion application or a combination of different variables like a word and the explanation of why it fits. The parser can extract the expected information for you.
This lesson covers the different types of parsing objects and the troubleshooting processing.

1. Output Parsers

There are three classes that we will introduce in this section. While the Pydrantic parser is the most powerful and flexible wrapper, knowing the other options for less complicated problems is beneficial. We will implement the thesaurus application in each section to better understand the details of each approach.

1-1. PydanticOutputParser

This class instructs the model to generate **its output in a JSON format and then extract the information from the response.** You will be able to treat the parser's output as a list, meaning it will be possible to index through the results without worrying about formatting.

It is important to note that **not all models have the same capability in terms of generating JSON** outputs. So, it would be best to use a more powerful model (like **OpenAI's DaVinci** instead of Curie) to get the most satisfactory result.

This class uses **the Pydantic library,** which helps define and validate data structures in Python. It enables us to characterize the expected output with a name, type, and description. We need a variable that can store multiple suggestions in the thesaurus example. It can be easily done by defining a class that inherits from the Pydantic's BaseModel class.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field, validator
from typing import List


# Define your desired data structure.
class Suggestions(BaseModel):
    words: List[str] = Field(description="list of substitue words based on context")


    # Throw error in case of receiving a numbered-list from API
    @validator('words')
    def not_start_with_number(cls, field):
        for item in field:
            if item[0].isnumeric():
                raise ValueError("The word can not start with numbers!")
        return field


parser = PydanticOutputParser(pydantic_object=Suggestions)
```

We always import and follow the necessary libraries by creating the `Suggestions` schema class.

There are two essential parts to this class:

1. **Expected Outputs:** Each output is defined by declaring a variable with desired type, like a list of strings (`: List[str]`) in the sample code, or it could be a single string (`: str`) if you are expecting just one word/sentence as the response. Also, it is required to write a simple explanation using the `Field` function's `description` attribute to help the model during inference. (We will see an example of having multiple outputs later in the lesson)
2. **Validators:** It is possible to declare functions to validate the formatting. We ensure that the first character is not a number in the sample code. The function's name is unimportant, but the `@validator` decorator must receive the same name as the variable you want to approve. (like `@validator('words')`) It is worth noting that the `field` variable inside the validator function will be a list if you specify it as one.

We will pass the created class to the `PydanticOutputParser` wrapper to make it a LangChain parser object. The next step is to prepare the prompt.

```
from langchain.prompts import PromptTemplate


template = """
Offer a list of suggestions to substitue the specified target_word based the
presented context.
{format_instructions}
target_word={target_word}
context={context}
"""


prompt = PromptTemplate(
    template=template,
    input_variables=["target_word", "context"],
    partial_variables={"format_instructions": parser.get_format_instructions()}
)


model_input = prompt.format_prompt(
                          target_word="behaviour",
                          context="The behaviour of the students in the classroom was
disruptive and made it difficult for the teacher to conduct the lesson."   )
```

As discussed in previous lessons, the `template` variable is a string that can have named index placeholders using the following `{variable_name}` format. The template outlines our expectations for the model, including the expected formatting from the parser and the inputs. The `PromptTemplate` receives the template string with the details of each placeholder's type. They could either be 1) `input_variables` whose value is initialized later on using the `.format_prompt()` function, or 2) `partial_variables` to be initialized instantly.

The prompt can send the query to models like GPT using LangChain's OpenAI wrapper. (Remember to set the `OPENAI_API_KEY` environment variables with your API key from OpenAI) We are using the Davinci model, one of the more powerful options to get the best results, and set the temperature value to 0, making the results reproducible.

The **temperature** value could be anything **between 0 to 1, where a higher number means the model is more creative.** Using larger value in production is a good practice if you deal with tasks requiring creative output.

```
from langchain.llms import OpenAI


# Before executing the following code, make sure to have
# your OpenAI key saved in the "OPENAI_API_KEY" environment variable.
model = OpenAI(model_name='text-davinci-003', temperature=0.0)


output = model(model_input.to_string())


parser.parse(output)
```

The sample code.

```
Suggestions(words=['conduct', 'manner', 'action', 'demeanor', 'attitude',
'activity'])
```

The output.

The parser object's `parse()` function will convert the model's string response to the format we specified. There is a list of words that you can index through and use in your applications.

Multiple Outputs Example

Here is a sample code for Pydantic class to process multiple outputs. It requests the model to suggest a list of words and present the reasoning behind each proposition. Replace the `template` variable and `Suggestion` class with the following codes to run this example. The template changes will ask the model to present its reasoning, and the suggestion class declares a new output named `reasons`. Also, the validator function manipulates the output to ensure every reasoning ends with a dot. Another use case of the validator function could be output manipulation.

```
template = """
Offer a list of suggestions to substitute the specified target_word based on the
presented context and the reasoning for each word.
{format_instructions}
target_word={target_word}
context={context}
"""
class Suggestions(BaseModel):
    words: List[str] = Field(description="list of substitue words based on context")
    reasons: List[str] = Field(description="the reasoning of why this word fits the
context")


    @validator('words')
    def not_start_with_number(cls, field):
      for item in field:
        if item[0].isnumeric():
          raise ValueError("The word can not start with numbers!")
      return field


    @validator('reasons')
    def end_with_dot(cls, field):
      for idx, item in enumerate( field ):
        if item[-1] != ".":
          field[idx] += "."
      return field
```

The multiple outputs sample code.

```
Suggestions(words=['conduct', 'manner', 'demeanor', 'comportment'], reasons=['refers to the way someone acts in a particular situation.', 'refers to the way someone behaves in a particular situation.', 'refers to the way someone behaves in a particular situation.', 'refers to the way someone behaves in a particular situation.'])
```

The output.

1-2. CommaSeparatedOutputParser

It is evident from the name of this class that it manages comma-separated outputs. It handles one specific case: anytime you want to receive a list of outputs from the model. Let's start by importing the necessary module.

```
from langchain.output_parsers import CommaSeparatedListOutputParser


parser = CommaSeparatedListOutputParser()
```

The parser does not require a setting up step. Therefore it is less flexible. We can create the object by calling the class. The rest of the process for writing the prompt, initializing the model, and parsing the output is as follows.

```
from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate


# Prepare the Prompt

template = """

Offer a list of suggestions to substitute the word '{target_word}' based the presented the following text: {context}.

{format_instructions}

"""


prompt = PromptTemplate(

    template=template,

    input_variables=["target_word", "context"],

    partial_variables={"format_instructions": parser.get_format_instructions()}

)
```

```
model_input = prompt.format(
  target_word="behaviour",
  context="The behaviour of the students in the classroom was disruptive and made it difficult for the teacher to conduct the lesson." )


# Loading OpenAI API
model = OpenAI(model_name='text-davinci-003', temperature=0.0)


# Send the Request
output = model(model_input)
parser.parse(output)
```

The sample code.

```
['Conduct',
 'Actions',
 'Demeanor',
 'Mannerisms',
 'Attitude',
 'Performance',
 'Reactions',
 'Interactions',
 'Habits',
 'Repertoire',
 'Disposition',
 'Bearing',
 'Posture',
 'Deportment',
 'Comportment']
```

The output.

Although most of the sample code has been explained in the previous subsection, two parts might need attention. **Firstly**, we tried a **new format for the prompt's template** to show different ways to write a prompt. **Secondly**, the use of `.format()` **instead of** `.format_prompt()` to generate the model's input.
The main difference compared to the previous subsection's code is that we no longer need to call the `.to_string()` object since the prompt is already in string type.

As you can see, the final output is a list of words that has some overlaps with the `PydanticOutputParser` approach with more variety. However, requesting additional reasoning information using the `CommaSeparatedOutputParser` class is impossible.

1-3. StructuredOutputParser

This is the first output parser implemented by the LangChain team. While it can process multiple outputs, it **only supports texts** and does not provide options for other data types, such as lists or integers. It can be used when you want to **receive one response** from the model. For example, only one substitute word in the thesaurus application.

```
from langchain.output_parsers import StructuredOutputParser, ResponseSchema

response_schemas = [
    ResponseSchema(name="words", description="A substitue word based on context"),
    ResponseSchema(name="reasons", description="the reasoning of why this word fits
the context.")   ]


parser = StructuredOutputParser.from_response_schemas(response_schemas)
```

The above code demonstrates how to define a schema. However, we are not going to go into details. This class has no advantage since the `PydanticOutputParser` class provides validation and more flexibility for more complex tasks, and the `CommaSeparatedOutputParser` option covers more straightforward applications.

2. Fixing Errors

The parsers are powerful tools to dynamically extract the information from the prompt and validate it to some extent. Still, they do not guarantee a response. Imagine a situation where you deployed your application, and the model's response [to a user's request] is **incomplete, causing the parser to throw an error. It is not ideal!** In the following subsections, we will introduce **two classes acting as fail-safe**. They add **a layer on top** of the model's response to help **fix the errors.**

The following approaches work with the `PydanticOutputParser` class since it is the only one with a validation method.

2-1. OutputFixingParser

This method tries to fix the parsing error by looking at the model's response and the previous parser. It uses a Large Language Model (LLM) to solve the issue. We will use GPT-3 to be consistent with the rest of the lesson, but it is possible to pass any supported model. Let's start by defining the Pydantic data schema and show a sample error that could occur.

```
from langchain.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field
from typing import List

# Define your desired data structure.
class Suggestions(BaseModel):
    words: List[str] = Field(description="list of substitue words based on context")
    reasons: List[str] = Field(description="the reasoning of why this word fits the
context")

parser = PydanticOutputParser(pydantic_object=Suggestions)

missformatted_output = '{"words": ["conduct", "manner"], "reasoning": ["refers to the
way someone acts in a particular situation.", "refers to the way someone behaves in a
particular situation."]}'

parser.parse(missformatted_output)
```

Sample code.

```
ValidationError: 1 validation error for Suggestions
reasons
  field required (type=value_error.missing)

During handling of the above exception, another exception occurred:

OutputParserException                     Traceback (most recent call last)
/usr/local/lib/python3.10/dist-packages/langchain/output_parsers/pydantic.py in parse(self, text)
     29             name = self.pydantic_object.__name__
     30             msg = f"Failed to parse {name} from completion {text}. Got: {e}"
---> 31             raise OutputParserException(msg)
     32
     33     def get_format_instructions(self) -> str:

OutputParserException: Failed to parse Suggestions from completion {"words": ["conduct", "manner"], "reasoning": ["refers to the way someone acts in a particular
situation.", "refers to the way someone behaves in a particular situation."]}. Got: 1 validation error for Suggestions
reasons
  field required (type=value_error.missing)
```

The output.

As you can see in the error message, the parser correctly identified an error in our sample response (`missformatted_output`) since we used the word `reasoning` instead of the expected `reasons` key. The `OutputFixingParser` class could easily fix this error.

```
from langchain.llms import OpenAI
from langchain.output_parsers import OutputFixingParser

model = OpenAI(model_name='text-davinci-003', temperature=0.0)
outputfixing_parser = OutputFixingParser.from_llm(parser=parser, llm=model)
outputfixing_parser.parse(missformatted_output)
```

Sample code.

```
Suggestions(words=['conduct', 'manner'], reasons=['refers to the way someone acts in a particular situation.', 'refers to the way someone behaves in a particular situation.'])
```

The output.

The `from_llm()` function takes the old parser and a language model as input parameters. Then, It initializes a new parser for you that has the ability to fix output errors. In this case, it successfully identified the misnamed key and changed it to what we defined.
However, fixing the issues using this class is not always possible. Here is an example of using `OutputFixingParser` class to resolve an error with a missing key.

```
missformatted_output = '{"words": ["conduct", "manner"]}'


outputfixing_parser = OutputFixingParser.from_llm(parser=parser, llm=model)


outputfixing_parser.parse(missformatted_output)
```

Sample code.

```
Suggestions(words=['conduct', 'manner'], reasons=["The word 'conduct' implies a certain behavior or action, while 'manner' implies a polite or respectful way of behaving."])
```

The output.

Looking at the output, it is evident that the model understood the key `reasons` missing from the response but didn't have the context of the desired outcome. It created a list with one entry, while we expect one reason per word. This is why we sometimes need to use the `RetryOutputParser` class.

2-2. RetryOutputParser

In some cases, the parser needs **access to both the output and the prompt to process the full conte**xt, as demonstrated in the previous section. We first need to define the mentioned variables. The following codes initialize the LLM model, parser, and prompt, which were explained in more detail earlier.

```
from langchain.llms import OpenAI

from langchain.prompts import PromptTemplate

from langchain.output_parsers import PydanticOutputParser

from pydantic import BaseModel, Field

from typing import List
```

```
# Define data structure.
class Suggestions(BaseModel):
    words: List[str] = Field(description="list of substitue words based on context")
    reasons: List[str] = Field(description="the reasoning of why this word fits the
context")


parser = PydanticOutputParser(pydantic_object=Suggestions)


# Define prompt
template = """
Offer a list of suggestions to substitue the specified target_word based the
presented context and the reasoning for each word.
{format_instructions}
target_word={target_word}
context={context}
"""


prompt = PromptTemplate(
    template=template,
    input_variables=["target_word", "context"],
    partial_variables={"format_instructions": parser.get_format_instructions()}
)


model_input = prompt.format_prompt(target_word="behaviour", context="The behaviour of
the students in the classroom was disruptive and made it difficult for the teacher to
conduct the lesson.")


# Define Model
model = OpenAI(model_name='text-davinci-003', temperature=0.0)
```

Now, we can fix the same `missformatted_output` using the `RetryWithErrorOutputParser` class. It receives the old parser and a model to declare the new parser object, as we saw in the previous section. However, the `parse_with_prompt` function is responsible for fixing the parsing issue while requiring the output and the prompt.

```
from langchain.output_parsers import RetryWithErrorOutputParser


missformatted_output = '{"words": ["conduct", "manner"]}'


retry_parser = RetryWithErrorOutputParser.from_llm(parser=parser, llm=model)


retry_parser.parse_with_prompt(missformatted_output, model_input)
```

Sample code.

```
Suggestions(words=['conduct', 'manner'], reasons=["The behaviour of the students in the classroom was disruptive and made it difficult for the teacher to conduct the lesson, so 'conduct' is a suitable substitute.", "The students' behaviour was inappropriate, so 'manner' is a suitable substitute."])
```

Sample output.

The outputs show that the `RetryOuputParser` has the ability to fix the issue where the `OuputFixingParser` was not able to. The parser correctly guided the model to generate one reason for each word.
The best practice to incorporate these techniques in production is to catch the parsing error using a `try: ... except: ...` method. It means we can capture the errors in the `except` section and attempt to fix them using the mentioned classes. It will limit the number of API calls and avoid unnecessary costs that are associated with it.

Conclusion

We learned how to validate and extract the information in an easy-to-use format from the language models' responses which are always a string. Additionally, we reviewed LangChain's fail-safe procedures to guarantee the consistency of the output. Combining these approaches will help us write more reliable applications in production environments.
In the next lesson, we will learn how to build a knowledge graph and capture useful information or entities from texts.
In the next lesson we'll modify the news summarizer we built in the previous module by improving how we manage the prompts.

You can find the code of this lesson in this online notebook.

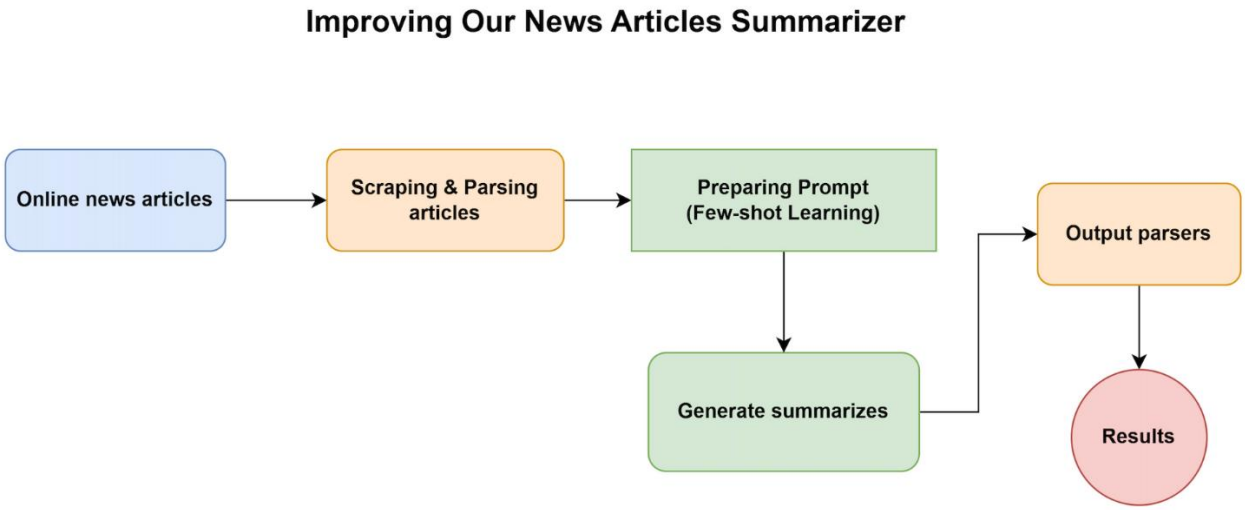
Improving Our News Articles Summarizer

Introduction

The purpose of this lesson is to enhance our previous implementation of a News Article Summarizer. Our objective is to make our tool even more effective at distilling key information from lengthy news articles and presenting that information in an easy-to-digest, bulleted list format. This enhancement will enable users to quickly comprehend the main points of an article in a clear, organized way, thus saving valuable time and enhancing the reading experience.
To achieve this, we will modify our existing summarizer to instruct the underlying language model to generate summaries as bulleted lists. This task involves a few changes to the way we present our prompt to the model, which we will guide you through in the workflow below.

Workflow for Building a News Articles Summarizer with Bulleted Lists

This is what we are going to doing this project.



We set up the environment and retrieved the news article.

1. **Install required libraries**: The first step is to ensure that the necessary libraries, namely `requests`, `newspaper3k`, and `LangChain`, are installed.
2. **Scrape articles**: We will use the `requests` library to scrape the content of the target news articles from their respective URLs.
3. **Extract titles and text**: The `newspaper` library will be used to parse the scraped HTML, extracting the titles and text of the articles.
4. **Preprocess the text**: The extracted texts need to be cleaned and preprocessed to make them suitable for input to LLM.

The rest of the lesson will explore new possibilities to enhance the application's performance further.

5. **Use Few-Shot Learning Technique**: We use the few-shot learning technique in this step. This template will provide a few examples of the language model to guide it in generating the summaries in the desired format - a bulleted list.
6. **Generate summaries**: With the modified prompt, we utilize the model to generate concise summaries of the extracted articles' text in the desired format.
7. **Use the Output Parsers**: We employ the Output Parsers to interpret the output from the language model, ensuring it aligns with our desired structure and format.
8. **Output the results**: Finally, we present the bulleted summaries along with the original titles, enabling users to quickly grasp the main points of each article in a structured manner.

With these steps, you will be able to construct a powerful tool capable of summarizing news articles into easily digestible, bulleted summaries, employing the `FewShotLearning` technique for added precision and `OutputParsers` for formatting the output using a defined data structure. Let's delve into it!

The initial steps of the process are technically the same as part 1 of this lesson.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.
Additionally, install the *newspaper3k* package, which has been tested in this lesson with the version `0.2.8`.

```
!pip install -q newspaper3k python-dotenv
```

Set the API key in your Python script or notebook as an environment variable with the `OPENAI_API_KEY` name. In order to set it from a `.env` file, you can use the `load_dotenv` function**.**

```
import os

import json

from dotenv import load_dotenv

load_dotenv()
```

We picked the URL of a news article to generate a summary. The following code fetches articles from a list of URLs using the `requests` library with a custom User-Agent header. It then extracts the title and text of each article using the `newspaper` library.

```
import requests

from newspaper import Article


headers = {

    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/89.0.4389.82 Safari/537.36'    }
```

```
article_url = "https://www.artificialintelligence-news.com/2022/01/25/meta-claims-new-ai-supercomputer-will-set-records/"


session = requests.Session()


try:
  response = session.get(article_url, headers=headers, timeout=10)


  if response.status_code == 200:
      article = Article(article_url)
      article.download()
      article.parse()


      print(f"Title: {article.title}")
      print(f"Text: {article.text}")
  else:
      print(f"Failed to fetch article at {article_url}")
except Exception as e:
    print(f"Error occurred while fetching article at {article_url}: {e}")
```

The sample code.

```
Title: Meta claims its new AI supercomputer will set records

Text: Ryan is a senior editor at TechForge Media with over a decade of experience covering the latest technology and interviewing leading industry figures. He can often be sighted at tech conferences with a strong coffee in one hand and a laptop in the other. If it's geeky, he’s probably into it. Find him on Twitter (@Gadget_Ry) or Mastodon (@gadgetry@techhub.social)


Meta (formerly Facebook) has unveiled an AI supercomputer that it claims will be the world’s fastest.


The supercomputer is called the AI Research SuperCluster (RSC) and is yet to be fully complete. However, Meta’s researchers have already begun using it for training large natural language processing (NLP) and computer vision models.
```

RSC is set to be fully built-in mid-2022. Meta says that it will be the fastest in the world once complete and the aim is for it to be capable of training models with trillions of parameters.

“We hope RSC will help us build entirely new AI systems that can, for example, power real-time voice translations to large groups of people, each speaking a different language, so they can seamlessly collaborate on a research project or play an AR game together,” wrote Meta in a blog post.

“Ultimately, the work done with RSC will pave the way toward building technologies for the next major computing platform — the metaverse, where AI-driven applications and products will play an important role.”

For production, Meta expects RSC will be 20x faster than Meta’s current V100-based clusters. RSC is also estimated to be 9x faster at running the NVIDIA Collective Communication Library (NCCL) and 3x faster at training large-scale NLP workflows.

A model with tens of billions of parameters can finish training in three weeks compared with nine weeks prior to RSC.

Meta says that its previous AI research infrastructure only leveraged open source and other publicly-available datasets. RSC was designed with the security and privacy controls in mind to allow Meta to use real-world examples from its production systems in production training.

What this means in practice is that Meta can use RSC to advance research for vital tasks such as identifying harmful content on its platforms—using real data from them.

“We believe this is the first time performance, reliability, security, and privacy have been tackled at such a scale,” says Meta.

(Image Credit: Meta)

Want to learn more about AI and big data from industry leaders? Check out AI & Big Data Expo. The next events in the series will be held in Santa Clara on 11-12 May 2022, Amsterdam on 20-21 September 2022, and London on 1-2 December 2022.

Explore other upcoming enterprise technology events and webinars powered by TechForge here.

The output.

Few Shot Prompting

We saw in the previous lessons how to use `FewShotPromptTemplate`; let's now see another way of adding examples to a prompt that is slightly different but achieves similar results. In this experiment, we include several examples that guide the model's summarization process to generate bullet lists. As a result, the model is expected to generate a bulleted list summarizing the given article.

```
from langchain.schema import (
    HumanMessage )


# we get the article data from the scraping part
article_title = article.title
article_text = article.text


# prepare template for prompt
template = """
As an advanced AI, you've been tasked to summarize online articles into bulleted
points. Here are a few examples of how you've done this in the past:


Example 1:
Original Article: 'The Effects of Climate Change
Summary:
- Climate change is causing a rise in global temperatures.
- This leads to melting ice caps and rising sea levels.
- Resulting in more frequent and severe weather conditions.


Example 2:
Original Article: 'The Evolution of Artificial Intelligence
Summary:
- Artificial Intelligence (AI) has developed significantly over the past decade.
- AI is now used in multiple fields such as healthcare, finance, and transportation.
- The future of AI is promising but requires careful regulation.


Now, here's the article you need to summarize:
```

```
==================
Title: {article_title}
{article_text}
==================
Please provide a summarized version of the article in a bulleted list format.
"""
# Format the Prompt
prompt = template.format(article_title=article.title, article_text=article.text)

messages = [HumanMessage(content=prompt)]
```

These examples provide the model with a better understanding of how we want it to respond. Here we have a few important components:
Article data: The title and text of the article are obtained, which will be used as inputs to the model.
Template preparation: A template is prepared for the prompt. This template includes a few-shot learning style, where the model is provided with examples of how it has previously converted articles into a bulleted list format. The template also includes placeholders for the actual article title and text that will be summarized. Then, the placeholders in the template (`{article_title}` and `{article_text}`) are replaced with the actual title and text of the article using the `.format()` method.
The next step is to use `ChatOpenAI` class to load the GPT-4 model for generating the summary. Then, the formatted prompt is passed to the language model as the input/prompt. The `ChatOpenAI` class's chat instance takes a `HumanMessage` list as an input argument.

```
from langchain.chat_models import ChatOpenAI

# load the model
chat = ChatOpenAI(model_name="gpt-4", temperature=0.0)

# generate summary
summary = chat(messages)
print(summary.content)
```

The sample code.

```
- Meta (formerly Facebook) has unveiled an AI supercomputer called the AI Research SuperCluster (RSC).
```

```
- The RSC is yet to be fully complete but is already being used for training large natural language processing (NLP) and computer vision models.

- Meta claims that the RSC will be the fastest in the world once complete and capable of training models with trillions of parameters.

- The aim is for the RSC to help build entirely new AI systems that can power real-time voice translations to large groups of people.

- Meta expects the RSC to be 20x faster than its current V100-based clusters for production.

- The RSC is estimated to be 9x faster at running the NVIDIA Collective Communication Library (NCCL) and 3x faster at training large-scale NLP workflows.

- Meta says that its previous AI research infrastructure only leveraged open source and other publicly-available datasets.

- RSC was designed with security and privacy controls in mind to allow Meta to use real-world examples from its production systems in production training.

- Meta can use RSC to advance research for vital tasks such as identifying harmful content on its platforms using real data from them.
```

The output.

The key takeaway here is the use of a few-shot learning style in the prompt. This provides the model with examples of how it should perform the task, which guides it to generate a bulleted list summarizing the article. By modifying the prompt and the examples, you can adjust the model's output to meet various requirements and ensure the model follows a specified format, tone, style, etc.

Output Parsers

Now, let's improve the previous section by using Output Parsers. The Pydantic output parser in LangChain offers a flexible way to shape the outputs from language models according to pre-defined schemas. When used alongside prompt templates, it enables more structured interactions with language models, making it easier to extract and work with the information provided by the model.

The prompt template includes the format instructions from our parser, which guide the language model to produce the output in the desired structured format. The idea is to demonstrate how you could use `PydanticOutputParser` class to receive the output as a type `List` that holds each bullet point instead of a string. The advantage of having a list is the possibility to loop through the results or index a specific item.

As mentioned before, the `PydanticOutputParser` wrapper is used to create a parser that will parse the output from the string into a data structure. The custom `ArticleSummary` class, which inherits the `Pydantic` package's `BaseModel` class, will be used to parse the model's output.

We defined the schema to present a `title` along with a `summary` variable that represents a list of strings using the `Field` object. The `description` argument will describe what each variable must represent and help the model to achieve it. Our custom class also includes a validator function to ensure that the generated output contains at least three bullet points.

```python
from langchain.output_parsers import PydanticOutputParser
from pydantic import validator
from pydantic import BaseModel, Field
from typing import List

# create output parser class
class ArticleSummary(BaseModel):
    title: str = Field(description="Title of the article")
    summary: List[str] = Field(description="Bulleted list summary of the article")

    # validating whether the generated summary has at least three lines
    @validator('summary', allow_reuse=True)
    def has_three_or_more_lines(cls, list_of_lines):
        if len(list_of_lines) < 3:
            raise ValueError("Generated summary has less than three bullet points!")
        return list_of_lines

# set up output parser
parser = PydanticOutputParser(pydantic_object=ArticleSummary)
```

The next step involves creating a template for the input prompt that instructs the language model to summarize the news article into bullet points. This template is used to instantiate a `PromptTemplate` object, which is responsible for correctly formatting the prompts that are sent to the language model. The `PromptTemplate` uses our custom parser to format the prompt sent to the language model using the `.get_format_instructions()` method, which will include additional instructions on how the output should be shaped.

```python
from langchain.prompts import PromptTemplate
# create prompt template
# notice that we are specifying the "partial_variables" parameter
template = """
You are a very good assistant that summarizes online articles.
Here's the article you want to summarize.
==================
Title: {article_title}
```

```
{article_text}

==================

{format_instructions}

"""


prompt = PromptTemplate(

    template=template,

    input_variables=["article_title", "article_text"],

    partial_variables={"format_instructions": parser.get_format_instructions()}     )


# Format the prompt using the article title and text obtained from scraping

formatted_prompt = prompt.format_prompt(article_title=article_title,
article_text=article_text)
```

Lastly, the `GPT-3` model with the temperature set to `0.0` is initialized, which means the output will be deterministic, favoring the most likely outcome over randomness/creativity. The parser object then converts the string output from the model to a defined schema using the `.parse()` method.

```
from langchain.llms import OpenAI

# instantiate model class

model = OpenAI(model_name="text-davinci-003", temperature=0.0)

# Use the model to generate a summary

output = model(formatted_prompt.to_string())


# Parse the output into the Pydantic model

parsed_output = parser.parse(output)

print(parsed_output)
```

The sample code.

```
ArticleSummary(title='Meta claims its new AI supercomputer will set records',
summary=['Meta (formerly Facebook) has unveiled an AI supercomputer that it claims
will be the world's fastest.', 'The supercomputer is called the AI Research
SuperCluster (RSC) and is yet to be fully complete.', 'Meta says that it will be the
fastest in the world once complete and the aim is for it to be capable of training
models with trillions of parameters.', 'For production, Meta expects RSC will be 20x
faster than Meta's current V100-based clusters.', 'Meta says that its previous AI
research infrastructure only leveraged open source and other publicly-available
```

```
datasets.', 'What this means in practice is that Meta can use RSC to advance research
for vital tasks such as identifying harmful content on its platforms—using real data
from them.'])
```

The output.

The Pydantic output parser is a powerful method for molding and structuring the output from language models. It uses the Pydantic library, known for its data validation capabilities, to define and enforce data schemas for the model's output.
This is a recap of what we did:

- We defined a Pydantic data structure named `ArticleSummary`. This model serves as a blueprint for the desired structure of the generated article summary. It comprises fields for the title and the summary, which is expected to be a list of strings representing bullet points. Importantly, we incorporate a validator within this model to ensure the summary comprises at least three points, thereby maintaining a certain level of detail in the summarization.
- We then instantiate a parser object using our `ArticleSummary` class. This parser plays a crucial role in ensuring the output generated by the language model aligns with the defined structures of our custom schema.
- To direct the language model's output, we create the prompt template. The template instructs the model to act as an assistant that summarizes online articles by incorporating the parser object.
- So, output parsers enable us to specify the desired format of the model's output, making extracting meaningful information from the model's responses easier.

Conclusion

In today's lesson, we've successfully navigated the path of crafting our News Articles Summarizer leveraging the potential of `PromptTemplates` and `OutputParsers`, showing the capabilities of prompt handling LangChain. The Pydantic output parser is a powerful method for molding and structuring the output from language models. It uses the Pydantic library, known for its data validation capabilities, to define and enforce data schemas for the model's output.
Following this, we define a Pydantic model named "ArticleSummary.” This model serves as a blueprint for the desired structure of the generated article summary. It comprises fields for the title and the summary, which is expected to be a list of strings representing bullet points. Importantly, we incorporate a validator within this model to ensure the summary comprises at least three points, thereby maintaining a certain level of detail in the summarization.
We then instantiate a PydanticOutputParser, passing it to the "ArticleSummary" model. This parser plays a crucial role in ensuring the output generated by the language model aligns with the structure outlined in the "Article Summary" model.
A good understanding of prompt and output design nuances equips you to customize the model to produce results that perfectly match your specific requirements.
In the next lesson, we'll do a project where we create a knowledge graph from textual data, making complex information more accessible and easier to understand.

You can find the code of this lesson in this online Notebook.

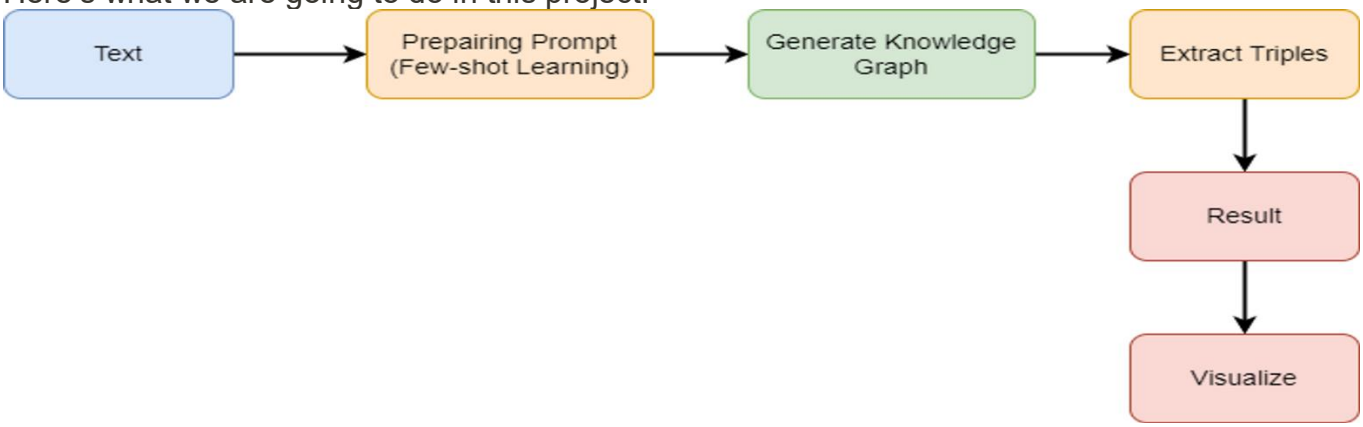
Creating Knowledge Graphs from Textual Data: Unveiling Hidden Connections

Introduction

In today's data-driven world, understanding the relationships between different pieces of information is crucial. Knowledge graphs have emerged as a powerful way to visualize and explore these connections, transforming unstructured text into a structured network of entities and their relationships. We will guide you through a simple workflow for creating a knowledge graph from textual data, making complex information more accessible and easier to understand.

Workflow for **Creating Knowledge Graphs from Textual Data**

Here's what we are going to do in this project.



Knowledge Graphs and Knowledge Bases: know the difference.

Before diving deep into our main topic, it's important to have a clear understanding of the difference between Knowledge Graphs and Knowledge Bases.
The terms "knowledge graph" and "knowledge base" are often used interchangeably, but they have subtle differences. **Knowledge base (KB) refers to structured information** that we have about a domain of interest. On the other hand, a **knowledge graph is a knowledge base structured as a graph,** where nodes represent entities and edges signify relations between those entities.
For example, from the text *"Fabio lives in Italy,"* we can extract the relation triplet `<Fabio, lives in, Italy>`, where *"Fabio"* and *"Italy"* are entities, and "lives in" it's their relation.

A **knowledge graph is a particular type of knowledge base. A knowledge base is not necessarily a knowledge graph.**

Building a Knowledge Graph

The process of building a knowledge graph usually consists of two sequential steps:

1. **Named Entity Recognition** (NER): This step involves extracting entities from the text, which will eventually become the nodes of the knowledge graph.
2. **Relation Classification** (RC): In this step, relations between entities are extracted, forming the edges of the knowledge graph.

Then, the knowledge graph is commonly visualized using libraries such as `pyvis` .
Typically the process of creating a knowledge base from the text can be enhanced by incorporating additional steps, such as:

- **Entity Linking**: This involves normalizing entities to the same entity, such as “Napoleon” and “Napoleon Bonapart.” This is usually done by linking them to a canonical source, like a Wikipedia page.
- **Source Tracking**: Keeping track of the origin of each relation, such as the article URL and text span. Keeping track of the sources allows us to gather insights into the reliability of the extracted information (e.g., a relation is accurate if it can be extracted from several sources considered accurate).

In this project, we’ll do the Named Entity Recognition and Relation Classification tasks simultaneously with an appropriate prompt. This joint task is commonly called **Relation Extraction (RE).**

Building a Knowledge Graph with LangChain

To demonstrate an example of using a prompt to extract relations from the text in LangChain, we can use the following `KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT` prompt as a starting point. This prompt is designed to **extract knowledge triples (subject, predicate, and object)** from a given text input

This prompt can be used by the `ConversationEntityMemory` class from LangChain library, which is a way for chatbots to **keep a memory of the past messages** of a conversation by storing the relations extracted from the past messages. Memory classes will be explained in a later lesson. In this example, we use this prompt just to extract relations from texts without leveraging a memory class.

Let's understand the structure of the `KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT`. This prompt is an instance of the `PromptTemplate` class with the input variable `text`. The template is a string that provides a few shot examples and instructions for the language model to follow when extracting knowledge triples from the input text. The following code requires the `OPENAI_API_KEY` key in the environment variable that stores your OpenAI API key.

Remember to install the required packages with the following command:
`pip install langchain==0.0.208 deeplake openai tiktoken`.

```
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
from langchain.chains import LLMChain
from langchain.graphs.networkx_graph import KG_TRIPLE_DELIMITER
# Prompt template for knowledge triple extraction
_DEFAULT_KNOWLEDGE_TRIPLE_EXTRACTION_TEMPLATE = (
    "You are a networked intelligence helping a human track knowledge triples"
    " about all relevant people, things, concepts, etc. and integrating"
    " them with your knowledge stored within your weights"
    " as well as that stored in a knowledge graph."
    " Extract all of the knowledge triples from the text."
    " A knowledge triple is a clause that contains a subject, a predicate,"
    " and an object. The subject is the entity being described,"
    " the predicate is the property of the subject that is being"
    " described, and the object is the value of the property.\n\n"
    "EXAMPLE\n"
    "It's a state in the US. It's also the number 1 producer of gold in the US.\n\n"
    f"Output: (Nevada, is a, state){KG_TRIPLE_DELIMITER}(Nevada, is in, US)"
    f"{KG_TRIPLE_DELIMITER}(Nevada, is the number 1 producer of, gold)\n"
    "END OF EXAMPLE\n\n"
    "EXAMPLE\n"
    "I'm going to the store.\n\n"
    "Output: NONE\n"
    "END OF EXAMPLE\n\n"
    "EXAMPLE\n"
    "Oh huh. I know Descartes likes to drive antique scooters and play the
mandolin.\n"
    f"Output: (Descartes, likes to drive, antique
scooters){KG_TRIPLE_DELIMITER}(Descartes, plays, mandolin)\n"
    "END OF EXAMPLE\n\n"
    "EXAMPLE\n"
    "{text}"
    "Output:"
)
```

```
KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT = PromptTemplate(
    input_variables=["text"],
    template=_DEFAULT_KNOWLEDGE_TRIPLE_EXTRACTION_TEMPLATE,        )


# Make sure to save your OpenAI key saved in the “OPENAI_API_KEY” environment
variable.
# Instantiate the OpenAI model
llm = OpenAI(model_name="text-davinci-003", temperature=0.9)


# Create an LLMChain using the knowledge triple extraction prompt
chain = LLMChain(llm=llm, prompt=KNOWLEDGE_TRIPLE_EXTRACTION_PROMPT)


# Run the chain with the specified text
text = "The city of Paris is the capital and most populous city of France. The Eiffel
Tower is a famous landmark in Paris."
triples = chain.run(text)


print(triples)
```

The sample code.

```
(Paris, is the capital of, France)<|>(Paris, is the most populous city of,
France)<|>(Eiffel Tower, is a, landmark)<|>(Eiffel Tower, is in, Paris)
```

The output.

In the previous code, we used the prompt to extract relation triplets from text using few-shot examples. We'll then parse the generated triplets and collect them into a list.
Here, `triples_response` will contain the knowledge triplets extracted from the text. We need to parse the response and collect the triplets into a list:

```
def parse_triples(response, delimiter=KG_TRIPLE_DELIMITER):
    if not response:
        return []
    return response.split(delimiter)
triples_list = parse_triples(triples)
# Print the extracted relation triplets
print(triples_list)
```

[' (Paris, is the capital of, France)', '(Paris, is the most populous city of, France)', '(Eiffel Tower, is a landmark),' '(Eiffel Tower, is located in, Paris)']

Knowledge Graph Visualization

The **NetworkX** library is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It provides various graph generators, random graphs, and synthetic networks, along with the benefits of Python's fast prototyping, ease of teaching, and multi-platform support.
To visualize the extracted triplets as a knowledge graph, we'll be using the pyvis library; To install the library, execute the following command. While it is preferable to install the latest version of the packages, it is worth noting that the codes in this lesson were written using version `0.3.2`.

```
pip install pyvis
```

Then this way, you can create an interactive knowledge graph visualization:

```
from pyvis.network import Network
import networkx as nx


# Create a NetworkX graph from the extracted relation triplets
def create_graph_from_triplets(triplets):
    G = nx.DiGraph()
    for triplet in triplets:
        subject, predicate, obj = triplet.strip().split(',')
        G.add_edge(subject.strip(), obj.strip(), label=predicate.strip())
    return G


# Convert the NetworkX graph to a PyVis network
def nx_to_pyvis(networkx_graph):
    pyvis_graph = Network(notebook=True)
    for node in networkx_graph.nodes():
        pyvis_graph.add_node(node)
    for edge in networkx_graph.edges(data=True):
        pyvis_graph.add_edge(edge[0], edge[1], label=edge[2]["label"])
    return pyvis_graph


triplets = [t.strip() for t in triples_list if t.strip()]
graph = create_graph_from_triplets(triplets)
pyvis_network = nx_to_pyvis(graph)
```

```
# Customize the appearance of the graph
pyvis_network.toggle_hide_edges_on_drag(True)
pyvis_network.toggle_physics(False)
pyvis_network.set_edge_smooth('discrete')
# Show the interactive knowledge graph visualization
pyvis_network.show('knowledge_graph.html')
```

First, we defined two functions for creating and visualizing a knowledge graph from a list of relation triplets; then, we used the `triples_list` to create a list of cleaned triplets which creates a NetworkX graph and converts it to a PyVis network. It also customizes the graph's appearance by enabling edge hiding on drag, disabling physics, and setting edge smoothing to 'discrete.'
With that process, we generated an interactive HTML file named `knowledge_graph.html` containing the knowledge graph visualization based on the extracted relation triplets:

Conclusion

Throughout this article, we've demonstrated a straightforward yet powerful workflow for creating knowledge graphs from textual data. We've transformed unstructured text into a structured network of entities and their relationships, making complex information more accessible and easier to understand.
It's worth noting that LangChain offers the `GraphIndexCreator` class, which automates the extraction of relation triplets and is seamlessly integrated with question-answering chains. In future articles, we'll delve deeper into this powerful feature, showcasing its potential further to enhance your knowledge graph creation and analysis capabilities.
The knowledge graph created through this workflow serves as a valuable tool for visualizing complex relationships and opens the door for further analysis, pattern recognition, and data-driven decision-making.
Congrats in completing the second module of the course! You can now test your knowledge with the module quizzes. The next module will be about managing external knowledge in LLM-based applications using indexers and retrievers.

RESOURCES:

Building a Knowledge Base from Texts: a Full Practical Example
Here is what we are going to do, progressively tackling more complex scenarios: Load the Relation Extraction REBEL model. Extract a knowledge base from a short text. Extract a knowledge base from a long text. Filter and normalize entities. Extract a knowledge base from an article at a specific URL.

langchain/knowledge_triplet_extraction.py at master · hwchase17/langchain
⚡ Building applications with LLMs through composability ⚡ - langchain/knowledge_triplet_extraction.py at master · hwchase17/langchain
You can find the code of this lesson in this online Notebook.