Table of Content:

LangChain

LangChain is an open-source framework designed to easily build applications using language models like GPT, LLaMA, Mistral, etc. At its core, LangChain is built around LLMs. We can use it for chatbots, **G**enerative **Q**uestion-**A**nswering (GQA), summarization, and much more.

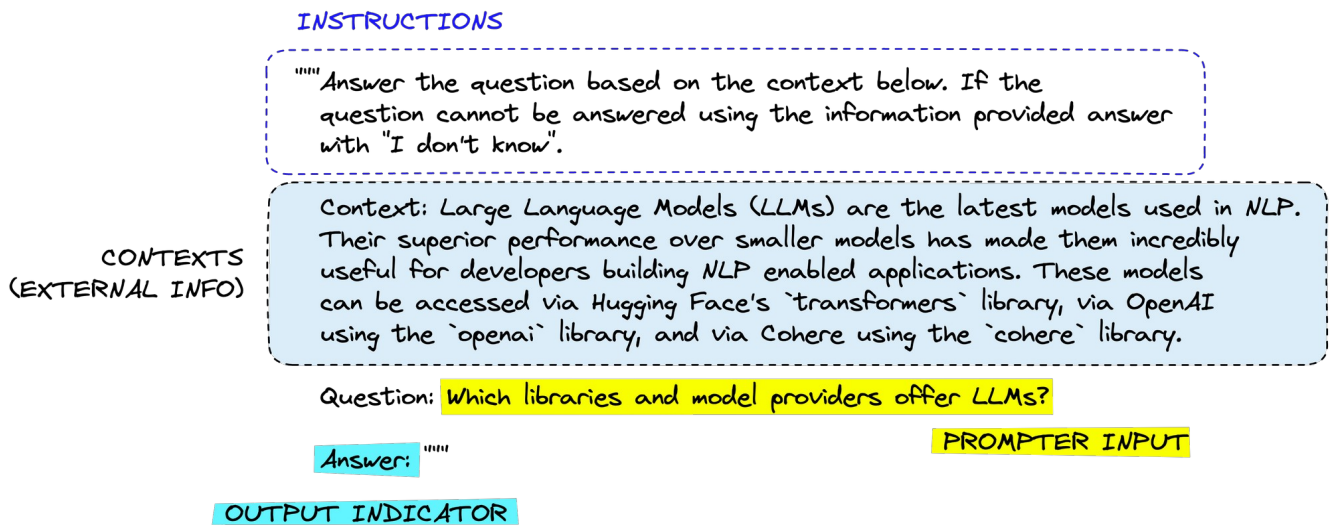
The core idea of the library is that we can *"chain"* together different components to create more advanced use cases around LLMs. Chains may consist of multiple components from several modules:

- **Prompt templates**: Prompt templates are templates for different types of prompts. Like "chatbot" style templates, ELI5 question-answering, etc
- **LLMs**: Large language models like GPT-3, BLOOM, etc
- **Agents**: Agents use LLMs to decide what actions should be taken. Tools like web search or calculators can be used, and all are packaged into a logical loop of operations.
- **Memory**: Short-term memory, long-term memory.

Prompts:

Prompts being input to LLMs are often structured in different ways so that we can get different results.One of the most powerful features of LangChain is its support for advanced prompt engineering. Prompt engineering refers to the design and optimization of prompts to get the most accurate and relevant responses from a language model.

it’s the practice of creating text inputs written in natural language, specify the task we want the AI to perform.In prompt engineering, **the key is not just what you ask but how you ask it**. This might include choosing the right wording, setting a particular tone or style, **providing necessary context, or even defining a role for the AI.**In some cases, prompts can also include examples to guide the AI's learning process. This technique, known as few-shot learning, can be particularly effective for complex tasks.



A good prompt typically contains these four components:

1.**Instructions** - Tell the model what to do, how to use the provided information, what to do with the query, and how to construct the output.

2.**Example Input** - Provide sample inputs to demonstrate to the model what is expected.

3.**Example Output** - Provide corresponding example outputs.

4.**Query** - The actual input you want the model to process.

5.**External information** or *context(s)* act as an additional source of knowledge for the model. These can be manually inserted into the prompt, retrieved via a vector database (retrieval augmentation), or pulled in via other means (APIs, calculations, etc.).

Note: Except for query, everything else is optional but may impact the quality of response significantly, especially the instructions that provide guidance to the LLM on the task and expected response.

Different types of Prompt Engineering

- ***Few Shot Prompting***: Providing the LLM with a few examples of desired outputs

- •***Template Based Prompting***: Using pre-defined templates with placeholders streamlining prompt creation and reuse.
- •***Instructional Prompting***: Explicitly instructing the LLM to how to perform the task, including steps and guidelines.
- •***Contrastive Prompting***: Providing the LLM with contrastive examples of good and bad outputs for it to distinguish between desired and undesired results.
- •***Meta-Prompting***: Training a separate model to generate the prompts for the main LLM.

Each component is usually placed in the prompt in this order. Starting with instructions, external information (where applicable), prompter input, and finally, the output indicator.

Example:

prompt = """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: Large Language Models (LLMs) are the latest models used in NLP. Their superior performance over smaller models has made them incredibly useful for developers building NLP enabled applications. These models can be accessed via Hugging Face's `transformers` library, via OpenAI using the `openai` library, and via Cohere using the `cohere` library.

Question: Which libraries and model providers offer LLMs?

Answer: """

The prompt template classes in Langchain are built to make constructing prompts with dynamic inputs easier. Of these classes, the simplest is the `PromptTemplate`. We'll test this by adding a single dynamic input to our previous prompt, the user `query`.

3 Types of LangChain Prompt Templates

When you prompt in LangChain, you're encouraged (but not required) to use a predefined template class such as:

- `**PromptTemplate**` for creating basic prompts.
- `**FewShotPromptTemplate**` for few-shot learning.
- `**ChatPromptTemplate**` for modeling chatbot interactions.

Prompt types are designed for flexibility, not exclusivity, allowing you to blend their features, like merging a FewShotPromptTemplate with a ChatPromptTemplate, to suit diverse use cases.

1.PromptTemplate:

LangChain's PromptTemplate class creates a dynamic string with variable placeholders:

example1:

```
1from langchain.prompts import PromptTemplate
2
3prompt_template = PromptTemplate.from_template(
4  "Write a delicious recipe for {dish} with a {flavor} twist."
5)
6
7# Formatting the prompt with new content
8formatted_prompt = prompt_template.format(dish="pasta", flavor="spicy")
9
10print(formatted_prompt)
```

It contains all the elements needed to create the prompt, but doesn't feature autocomplete for the variables `**dish**` and `**flavor**`.

LangChain's templates use Python's `**str.format**` by default, but for complex prompts you can also use jinja2.

Example2:

from langchain import PromptTemplate

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: Large Language Models (LLMs) are the latest models used in NLP. Their superior performance over smaller models has made them incredibly useful for developers building NLP enabled applications. These models can be accessed via Hugging Face's `transformers` library, via OpenAI using the `openai` library, and via Cohere using the `cohere` library.

Question: {query}

Answer: """

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

```
print(
    prompt_template.format(
        query="Which libraries and model providers offer LLMs?"
    )
)
```

Output:

```
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: Large Language Models (LLMs) are the latest models used in NLP.
Their superior performance over smaller models has made them incredibly
useful for developers building NLP enabled applications. These models
can be accessed via Hugging Face's `transformers` library, via OpenAI
using the `openai` library, and via Cohere using the `cohere` library.

Question: Which libraries and model providers offer LLMs?

Answer:
```

Naturally, we can pass the output of this directly into an LLM object like so:

```
print(openai(
    prompt_template.format(
        query="Which libraries and model providers offer LLMs?"
    )
))
```

Few Shot Prompt Templates:

The success of LLMs comes from their large size and ability to store “knowledge” within the model parameter, which is *learned* during model training. However, there are more ways to pass knowledge to an LLM. The two primary methods are:

- **Parametric knowledge** — the knowledge mentioned above is anything that has been learned by the model during training time and is stored within the model weights (or *parameters*).
- **Source knowledge** — any knowledge provided to the model at inference time via the input prompt.

Langchain's FewShotPromptTemplate caters to **source knowledge** input. The idea is to "train" the model on a few examples — we call this *few-shot learning* — and these examples are given to the model within the prompt.

Few-shot learning is perfect when our model needs help understanding what we're asking it to do.

Often a more useful form of prompting than sending a simple string with a request or question is to include several examples of outputs you want.

This is few-shot learning and is used to train models to do new tasks well, even when they have only a limited amount of training data available.

Many real-world use cases benefit from few-shot learning, for instance:

- **An automated fact checking tool** where you provide different few-shot examples where the model is shown how to verify information, ask follow-up questions if necessary, and conclude whether a statement is true or false.

- **A technical support and troubleshooting guide** that assists users in diagnosing and solving issues with products or software, where the `**FewShotPromptTemplate**` could contain examples of common troubleshooting steps, including how to ask the user for specific system details, interpret symptoms, and guide them through the solution process.

The `**FewShotPromptTemplate**` class takes a list of (question-and-answer) dictionaries as input, before asking a new question:

exmaple1:

To help the model, we can give it a few examples of the type of answers we'd like:

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

User: How are you?

AI: I can't complain but sometimes I still do.

User: What time is it?

AI: It's time to get a watch.

User: What is the meaning of life?

AI: """

print(openai(prompt))
output: `42, of course!`

Exmaple2:

```
from langchain.prompts.few_shot import FewShotPromptTemplate
from langchain.prompts.prompt import PromptTemplate
examples = [
      {"question": "What is the tallest mountain in the world?",
      "answer": "Mount Everest" },
      { "question": "What is the largest ocean on Earth?",
       "answer": "Pacific Ocean" },
      { "question": "In which year did the first airplane fly?",
       "answer": "1903" } ]
example_prompt = PromptTemplate(
      input_variables=["question", "answer"],
      template="Question: {question}\n{answer}",)
prompt = FewShotPromptTemplate(
      examples=examples,
      example_prompt=example_prompt,
      suffix="Question: {input}",
      input_variables=["input"],)
print(prompt.format(input="What is the name of the famous clock tower in London?"))
```

We can then formalize this process with Langchain's FewShotPromptTemplate:

example3:

```
from langchain import FewShotPromptTemplate

# create our examples
```

```
examples = [
    {
        "query": "How are you?",
        "answer": "I can't complain but sometimes I still do."
    }, {
        "query": "What time is it?",
        "answer": "It's time to get a watch."
    }
]

# create a 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 exerpts from conversations with an AI
assistant. The assistant is typically sarcastic and witty, producing
creative  and funny responses to the 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"
)
```

If we then pass in the `examples` and user `query`, we will get this:

In[15]:

```
query = "What is the meaning of life?"

print(few_shot_prompt_template.format(query=query))
```

Out[15]:

```
The following are exerpts from conversations with an AI
assistant. The assistant is typically sarcastic and witty, producing
creative  and funny responses to the users questions. Here are some
examples:


User: How are you?
AI: I can't complain but sometimes I still do.


User: What time is it?
AI: It's time to get a watch.


User: What is the meaning of life?
AI:
```

A dynamic number of examples is important because the maximum length of our prompt and completion output is limited. This limitation is measured by the *maximum context window*.

context window=input tokens+output tokens

Considering this, we need to balance the number of examples included and our prompt size,rather than passing the `examples` directly.

Using an example selector

instead of feeding the examples directly into the `FewShotPromptTemplate` object, we will feed them into an `ExampleSelector` object.

1.LengthBasedExampleSelector:

```
examples = [
    {
        "query": "How are you?",
        "answer": "I can't complain but sometimes I still do."
    }, {
        "query": "What time is it?",
        "answer": "It's time to get a watch."
    }, {
        "query": "What is the meaning of life?",
        "answer": "42"
    }, {
        "query": "What is the weather like today?",
        "answer": "Cloudy with a chance of memes."
    }, {
        "query": "What is your favorite movie?",
        "answer": "Terminator"
    }, {
        "query": "Who is your best friend?",
        "answer": "Siri. We have spirited debates about the meaning of life."
    }, {
        "query": "What should I do today?",
        "answer": "Stop talking to chatbots on the internet and go outside."
    }
]
from langchain.prompts.example_selector import LengthBasedExampleSelector

example_selector = LengthBasedExampleSelector(
    examples=examples,
    example_prompt=example_prompt,
    max_length=50  # this sets the max length that examples should be
)
```

We then pass our example_selector to the FewShotPromptTemplate to create a new — and dynamic — prompt template:

```
# now create the few shot prompt template
dynamic_prompt_template = FewShotPromptTemplate(
    example_selector=example_selector,  # use example_selector instead of examples
    example_prompt=example_prompt,
    prefix=prefix,
    suffix=suffix,
    input_variables=["query"],
    example_separator="\n"
)
```

2. SemanticSimilarityExampleSelector:

This class selects few-shot examples based on their similarity to the input. It uses an embedding model to compute the similarity between the input and the few-shot examples, as well as a vector store to perform the nearest neighbor search.

```
from langchain_chroma import Chroma

from langchain_core.example_selectors import SemanticSimilarityExampleSelector

from langchain_openai import OpenAIEmbeddings



example_selector = SemanticSimilarityExampleSelector.from_examples(

    # This is the list of examples available to select from.

    examples,

    # This is the embedding class used to produce embeddings which are used to measure semantic similarity.

    OpenAIEmbeddings(),

    # This is the VectorStore class that is used to store the embeddings and do a similarity search over.
```

```
    Chroma,

    # This is the number of examples to produce.

    k=1,

)



# Select the most similar example to the input.

question = "Who was the father of Mary Ball Washington?"

selected_examples = example_selector.select_examples({"question": question})

print(f"Examples most similar to the input: {question}")

for example in selected_examples:

    print("\n")

    for k, v in example.items():

        print(f"{k}: {v}")
```

3.ChatPromptTemplate`

The `**ChatPromptTemplate**` class focuses on the conversation flow between a user and an AI system, and provides instructions or requests for roles like user, system, assistant, and others (the exact roles you can use will depend on your LLM model).

Such roles give deeper context to the LLM and elicit better responses that help the model grasp the situation more holistically. System messages in particular provide implicit instructions or set the scene, informing the LLM of expected behavior.

```
1from langchain_core.prompts import ChatPromptTemplate
2
3# Define roles and placeholders

chat_template = ChatPromptTemplate.from_messages(
5  [
6    ("system", "You are a knowledgeable AI assistant. You are called {name}."),
7    ("user", "Hi, what's the weather like today?"),
8    ("ai", "It's sunny and warm outside."),
9    ("user", "{user_input}"),
10   ]
11)
12
```

```
13messages = chat_template.format_messages(name="Alice", user_input="Can you
tell me a joke?")
```

The roles in this class are:

- `**System**` for a system chat message setting the stage (e.g., "*You are a knowledgeable historian*").
- `**User**`, which contains the user's specific historical question.
- `**AI**`, which contains the LLM's preliminary response or follow-up question.

Once the template object is instantiated, you can use it to generate chat prompts by replacing the placeholders with actual content.

Agents

An agent is a software program that can interact with the real world. Langchain provides a number of different types of agents. Agents are a powerful tool that can be used to automate tasks and interact with the real world. Langchain agents can be used to perform a variety of tasks, such as Answering questions, Generating text, Translating languages, Summarizing text, etc.

LangChain agents, a revolutionary framework that bridges the gap between LLM capabilities and automated action.

They are specialized components within the LangChain framework that act as intelligent assistants, using LLMs to process natural language input and then orchestrating a sequence of actions to achieve a desired outcome.

Loading tools

Tools are functions that an agent can invoke. There are two important design considerations around tools:

1. Giving the agent access to the right tools
2. Describing the tools in a way that is most helpful to the agent

Without thinking through both, you won't be able to build a working agent. If you don't give the agent access to a correct set of tools, it will never be able to accomplish the objectives you give it. If you don't describe the tools well, the agent won't know how to use them properly.

Types of Agents in LangChain

LangChain offers several types of agents that use the ReACT framework.

- **Zero-Shot ReACT**: Has no memory.
- **Conversational ReACT**: Has memory. Great for chatbot use cases.
- **ReACT Docstore**: These expand on the ReACT by incorporating a document store for information retrieval. It's built to handle tasks that involve searching for and referencing specific information within a set of documents.
- **Self-Ask** With Search: These can autonomously generate questions based on the input it receives and use search capabilities to find answers. This self-asking ability combined with search makes the agent powerful for complex problem solving tasks.

To decide which actions to take and in what order, agents use an LLM.

1. Zero-shot ReAct

Zero-shot ReAct Agent is **a language generation model** that can be used to create a realistic variety of contexts, even if it has not been trained on data from those contexts. It can be used for a variety of tasks, such as generating creative text formats, translating languages, and writing different kinds of creative content.

Code:

```
# create prompt template & an llm chain

from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain

prompt = PromptTemplate(
input_variables=["query"],
template="{query}"
)

chain = LLMChain(llm=llm, prompt=prompt)
# create a math tool

from langchain.agents import Tool, load_tools

llm_tool = Tool(
name="Language Model",
func=chain.run,
description="Use this tool for general queries and logic"
)

tools = load_tools(
```

```
['llm-math'],
llm=llm
)

tools.append(llm_tool)
```

`"zero-shot-react-description"` refers to a specific configuration of the Zero-Shot ReACT agent tailored for generating descriptions or handling tasks without requiring memory of past interactions.

```
from langchain.agents import initialize_agent
agent = initialize_agent(
agent="zero-shot-react-description",
tools=tools,
llm=llm,
verbose=True,
max_iterations=5
)
```

```
print(agent.agent.llm_chain.prompt.template)
```

```
"""
Answer the following questions as best you can. You have access to the following tools:

Calculator: Useful for when you need to answer questions about math.
Language Model: Use this tool for general queries and logic

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [Calculator, Language Model]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}
"""
```

The template has a structured approach to problem-solving, used by the agent to logically break down and answer the query.

- **Question** is the direct input question that needs to be answered.
- **Thought** is for the agent to internally deliberate on the approach or strategy to solve the problem.
- **Action** specifies what the agent should do next. The actions include using a calculator for math operations or consulting an LLM for non-math queries.

•**Action Input** is the specific input to the action, which could be a math expression for the calculator or a more detailed query for the LLM.
•**Observation** captures the outcome of the action taken.
•**Final Thought and Answer** are obtained after the multiple cycles of Thought/Action/Observation.

2. Conversational ReAct

This agent is designed to be used in conversational settings. The prompt is designed to make the agent helpful and conversational. It uses the React framework to decide which tool to use and uses memory to remember previous conversation interactions.

`"conversational-react-description"` indicates that the agent uses the ReACT framework tailored for conversational applications.

Code:

```
from  langchain.memory import  ConversationBufferMemory

memory = ConversationBufferMemory(memory_key="chat_history")

conversational_agent = initialize_agent(
agent="conversational-react-description",
tools=tools,
llm=llm,
verbose=True,
max_iterations=5,
memory=memory
)

print(conversational_agent.agent.llm_chain.prompt.template)

"""
Assistant is a large language model trained by OpenAI.

Assistant is designed to be able to assist with a wide range of tasks, from answering simple questions to providing in-depth explanations and discussions on a wide range of topics. As a language model, Assistant is able to generate human-like text based on the input it receives, allowing it to engage in natural-sounding conversations and provide responses that are coherent and relevant to the topic at hand.

Assistant is constantly learning and improving, and its capabilities are constantly evolving. It is able to process and understand large amounts of text, and can use this knowledge to provide accurate and informative responses to a wide range of questions. Additionally, Assistant is able to generate its own text based on the input it receives, allowing it to engage in discussions and provide explanations and descriptions on a wide range of topics.

Overall, Assistant is a powerful tool that can help with a wide range of tasks and provide valuable insights and information on a wide range of topics. Whether you need help with a specific question or just want to have a conversation about a particular topic, Assistant is here to assist.
```

```
TOOLS:
------

Assistant has access to the following tools:

> Calculator: Useful for when you need to answer questions about math.
> Language Model: Use this tool for general queries and logic

To use a tool, please use the following format:

```
Thought: Do I need to use a tool? Yes
Action: the action to take, should be one of [Calculator, Language Model]
Action Input: the input to the action
Observation: the result of the action
```
...
{chat_history}

New input: {input}
{agent_scratchpad}
"""
```

The template begins by describing the Assistant, highlighting its purpose. Then it mentions the Assistant's ongoing learning process, its ability to understand large amount of text, and its capability to provide accurate and informative responses.

Then, the template outlines the tools available and provides a format for how to use these tools.

```
query = "Martin has 3 apples, he gave 2 of them to his friends, and this friend gave him 10 apples. How many apples does Martin have?"

result = conversational_agent(query)

"""
> Entering new AgentExecutor chain...
Thought: Do I need to use a tool? Yes
Action: Calculator
Action Input: 3 - 2 + 10
Observation: Answer: 11
Thought: Do I need to use a tool? No
AI: Martin now has 11 apples.

> Finished chain.
"""
```

3. ReAct Docstore

The concept of a "Docstore" refers to a document storage and retrieval system designed to work with LLMs.

LangChain docstore allow us to store and retrieve information using traditional retrieval methods.

- The "**Search**" functionality allows agents to query the document store for relevant articles or documents based on the query.

•Once relevant documents are retrieved through the Search process, the "**Lookup**" functionality enables the agent to pinpoint the exact chunk of information within these documents that directly addresses the query.

`DocstoreExplorer` class is used to create an explorer for the document store. It handles the functionalities of searching and retrieving information from a document store.

`Wikipedia` class allows the agent to access Wikipedia.

```
rom langchain import Wikipedia

from langchain.agents.react.base import DocstoreExplorer


docstore = DocstoreExplorer(Wikipedia())


tools = [
Tool(
name="Search",
func=docstore.search,
description="search wikipedia"
),
Tool(
name="Lookup",
func=docstore.lookup,
description="lookup a term in wikipedia"
)
]
```

`DocstoreExplorer` will use Wikipedia content as its database for retrieving and exploring information.

```
docstore_agent = initialize_agent(tools,lm,agent="react-docstore",verbose=True,
max_iterations=5,   handle_parsing_errors=True)

query = "Where is Lewis Hamilton from?"

result = docstore_agent.run(query)

"""

> Entering new AgentExecutor chain...

Thought: I need to search Lewis Hamilton and find where he is from.

Action: Search[Lewis Hamilton]
```

Observation: Sir Lewis Carl Davidson Hamilton (born 7 January 1985) is a British racing driver competing in Formula One, driving for Mercedes. Hamilton has won a joint-record seven Formula One World Drivers' Championship titles (tied with Michael Schumacher), and holds the records for most number of wins (103), pole positions (104), and podium finishes (197), among other records.

Born and raised in Stevenage, Hertfordshire, he began karting in 1993 at the age of eight and achieved success in local, national and international championships. Hamilton joined the inaugural McLaren-Mercedes Young Driver Programme in 1998, and progressed to win the 2003 British Formula Renault Championship, 2005 Formula 3 Euro Series and the 2006 GP2 Series. This led to a Formula One drive with McLaren-Mercedes from 2007 to 2012, making him the first black driver to race in the series. In his debut season, Hamilton set numerous records as he finished runner-up to Kimi Räikkönen by one point. In 2008, he won his maiden title in dramatic fashion—making a crucial overtake on the last lap of the 2008 Brazilian Grand Prix, the last race of the season—to become the then-youngest ever Formula One World Champion. Following six seasons with McLaren, Hamilton signed with Mercedes in 2013.

Thought: Lewis Hamilton is from Stevenage, Hertfordshire, a town in England.

Action: Finish[Stevenage, Hertfordshire, England]

"""

4. Self-ask with Search

This agent only uses one tool, which seems to be called Intermediate Answer.

```
from langchain import SerpAPIWrapper

search = SerpAPIWrapper(serpapi_api_key="your-serpapi-key")
tools = [
Tool(
name="Intermediate Answer",
func=search.run,
description="google search"
)
]

self_ask_with_search = initialize_agent(
tools,
llm,
agent='self-ask-with-search',
verbose=True,
handle_parsing_errors=True
)
query = "who has scored the most: Messi, Neymar?"
result = self_ask_with_search(query)

"""
> Entering new AgentExecutor chain...
Yes.
```

Follow up: How many goals has Messi scored?
Intermediate answer: He has scored 833 goals for club and country throughout his professional career and is also the first and only player in history to win five and six European Golden Shoes. During his football career, he won 44 collective trophies.
Follow up: How many goals has Neymar scored?
Intermediate answer: Neymar Jr. played 763 matches as a professional, scoring 476 goals.
So the final answer is: Messi

> Finished chain.
"""

Implement the math bot using agents:

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent, Tool
from langchain.agents import AgentType
from langchain.llms.bedrock import Bedrock
from langchain import LLMMathChain

model_parameter = {"temperature": 0.0, "top_p": .5, "max_tokens_to_sample": 2000}

react_agent_llm = Bedrock(model_id="anthropic.claude-instant-v1", model_kwargs=model_parameter)
math_chain_llm = Bedrock(model_id="anthropic.claude-instant-v1",
model_kwargs={"temperature":0,"stop_sequences"  : ["```output"]})

tools = load_tools(["serpapi"], llm=react_agent_llm)

llm_math_chain = LLMMathChain.from_llm(llm=math_chain_llm, verbose=True)

llm_math_chain.llm_chain.prompt.template = """Human: Given a question with a math problem, provide only a single line mathematical expression that solves the problem in the following format. Don't solve the expression only create a parsable expression.
```text
${{single line mathematical expression that solves the problem}}
```

Assistant:
Here is an example response with a single line mathematical expression for solving a math problem:
```text
37593**(1/5)
```

Human: {question}

Assistant:"""
```

note: For our chain-based tools, we will be using the `Tool.from_function()`

```
tools.append(
Tool.from_function(
func=llm_math_chain.run,
name="Calculator",
description="Useful for when you need to answer questions about math.",
)
)

react_agent = initialize_agent(tools,
react_agent_llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
```

```
verbose=True,
)


prompt_template = """Use the following format:
Question: the input question you must answer
Thought: you should always think about what to do, Also try to follow steps mentioned above
Action: the action to take, should be one of ["Search", "Calculator"]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Question: {input}

Assistant:
{agent_scratchpad}"""

react_agent.agent.llm_chain.prompt.template=prompt_template

react_agent("Who is the United States President? What is his current age divided by 2?")
```

explanation:

1.we have created 2 llms for math expression and one is for agent.

2.wehave created a template for match chain to instruct to solve single line expressions only with example.

3.The second template is to guide the react llm how to behave and what actions to take to provide the final output.

output:

```
> Entering new AgentExecutor chain...
Here is the step-by-step process:

Question: Who is the United States President? What is his current age divided by 2?

Thought: I need to first find out who the current US President is and his age.

Action: Search
Action Input: current us president

Observation: Joe Biden
Thought: Now I need to find Joe Biden's current age.

Action: Search
Action Input: Joe Biden's age

Observation: 81 years
Thought: Now I can divide his age by 2.

Action: Calculator
Action Input: 81 / 2

...

Final Answer: 40.5

> Finished chain.
```

Building a Math Application with LangChain Agents

Why do LLMs struggle with Math?

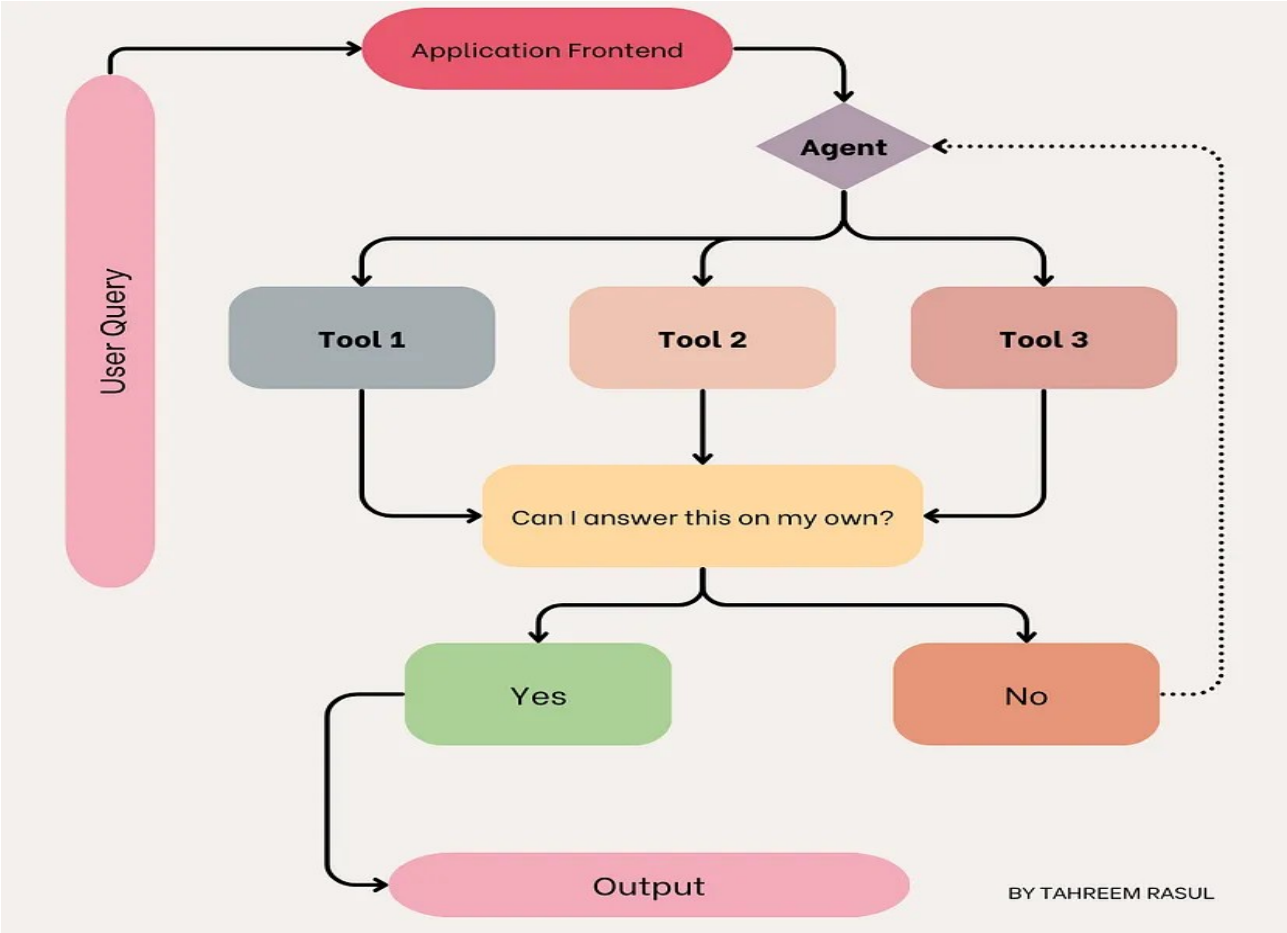
- **Lack of training data:** One reasons is the limitations of their training data. **Language models, trained on vast text datasets**, may lack sufficient mathematical problems and solutions. This can lead to misinterpretations of numbers, forgetting important calculation steps, and a lack of quantitative reasoning skills.

•**Lack of numeric representations:** Another reason is that LLMs are **designed to understand and generate text, operating on tokens instead of numeric values**. Most text-based tasks can have multiple reasonable answers. However, **math problems typically have only one correct solution.**

•**Generative nature:** Due to the generative nature of these language models, generating consistently accurate and precise answers to math questions can be challenging for LLMs.

The agent for our Math Wiz app will be using the following tools:

1. **Wikipedia Tool:** this tool will be responsible for fetching the latest information from Wikipedia using the Wikipedia API. While there are paid tools and APIs available that can be integrated inside LangChain, I would be using Wikipedia as the app's online source of information.
2. **Calculator Tool:** this tool would be responsible for solving a user's math queries. This includes anything involving numerical calculations. For example, if a user asks what the square root of 4 is, this tool would be appropriate.
3. **Reasoning Tool:** the final tool in our application setup would be a reasoning tool, responsible for tackling logical/reasoning-based user queries. Any mathematical word problems should also be handled with this tool.



The agent for our Math Wiz app will be using the following tools:

1. **Wikipedia Tool:** this tool will be responsible for **fetching the latest information** from Wikipedia using the Wikipedia API. While there are paid tools and APIs available that can be integrated inside LangChain, I would be using Wikipedia as the app's online source of information.
2. **Calculator Tool:** this tool would be responsible for solving a user's math queries. This includes anything involving **numerical calculations**. For example, if a user asks what the square root of 4 is, this tool would be appropriate.
3. **Reasoning Tool:** the final tool in our application setup would be a reasoning tool, responsible for tackling logical/reasoning-based user queries. Any mathematical **word problems should also be handled with this tool.**

Code:

```
from langchain_openai import OpenAI
from langchain.chains import LLMMathChain, LLMChain
from langchain.prompts import PromptTemplate
from langchain_community.utilities import WikipediaAPIWrapper
```

```
from langchain.agents.agent_types import AgentType

from langchain.agents import Tool, initialize_agent

from dotenv import load_dotenv


load_dotenv()

llm = OpenAI(model='gpt-3.5-turbo-instruct',temperature=0)

wikipedia = WikipediaAPIWrapper()
wikipedia_tool = Tool(name="Wikipedia",
func=wikipedia.run,
description="A useful tool for searching the Internet
to find information on world events, issues, dates, years, etc. Worth
using for general topics. Use precise questions.")



problem_chain = LLMMathChain.from_llm(llm=llm)
math_tool = Tool.from_function(name="Calculator",
func=problem_chain.run,
description="Useful for when you need to answer questions
about math. This tool is only for math questions and nothing else. Only input
math expressions.")



word_problem_template = """You are a reasoning agent tasked with solving
the user's logic-based questions. Logically arrive at the solution, and be
factual. In your answers, clearly detail the steps involved and give the
final answer. Provide the response in bullet points.
Question {question}

Answer"""


math_assistant_prompt = PromptTemplate(

input_variables=["question"],
template=word_problem_template
)

word_problem_chain = LLMChain(llm=llm, prompt=math_assistant_prompt)
word_problem_tool = Tool.from_function(name="Reasoning Tool",
func=word_problem_chain.run,
description="Useful for when you need
to answer logic-based/reasoning questions.",
)

agent = initialize_agent(
tools=[wikipedia_tool, math_tool, word_problem_tool],
llm=llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
verbose=False,
handle_parsing_errors=True
)



print(agent.invoke({"input": "I have 3 apples and 4 oranges. I give half of my oranges away and buy two dozen
new ones, alongwith three packs of strawberries. Each pack of strawberry has 30 strawberries.
How many total pieces of fruit do I have at the end?"}))
```

Chains in langchain:

“Chains” are sequences of components or steps that are linked together to perform a task or process. Each component in a chain is responsible for a particular operation, and the output of one component is passed as input to the next component in the sequence.

1.LLMChain

It is an entry-level, most basic, specialized type of chain designed specifically for working with LLMs.

```
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate, ChatPromptTemplate


prompt = PromptTemplate(
input_variables=["place"],
template="Best food to taste in {place}?",
)

chain = LLMChain(llm=gpt, prompt=prompt)
chain.invoke({place='mumbai'})
```

We use a `PromptTemplate` within an `LLMChain` object.

We can use multiple inputs, too:

```
template = """ As a songwriter for celebrity events, please create an elegant and memorable (100 words) ballad that highlights the location {location} and the main character {name} BALLAD LYRICS: """


prompt = PromptTemplate(input_variables=["location", "name"], template=template)


chain = LLMChain(llm=gpt, prompt=prompt)
```

```
chain.invoke({"location": "Los Angeles", "name": "Vincent Vega"})
```

2.SimpleSequentialChain

It is a type of chain that allows us to create a sequence of components that are linearly executed one after the other. This is a straightforward way to build a processing pipeline where the output of one component is passed as input to the next component in the sequence.

Each step in the chain receives only one input and produces only one output

```
from langchain.chains import SimpleSequentialChain

first_prompt = ChatPromptTemplate.from_template(
"Who is the most popular football player in the history of {country}?"
)
chain_one = LLMChain(llm=gemini, prompt=first_prompt)
second_prompt = ChatPromptTemplate.from_template(
"What is the best football club that {player} has ever played for?"
)
chain_two = LLMChain(llm=gpt, prompt=second_prompt)


overall_simple_chain = SimpleSequentialChain(chains=[chain_one, chain_two],
verbose=True)
overall_simple_chain.invoke("Argentina")
```

We have set up two chains: the first employs Gemini, and the second utilizes ChatGPT. Following this, we created
A `SimpleSequentialChain` with `chain_one` and `chain_two` as its component. This means that the output of `chain_one` will be used as the input for `chain_two`.

3.SequentialChain

It is a more advanced and flexible implementation of a sequential chain. It allows for more complex interactions between components, such as branching or conditional execution, where the output of one component might determine whether or not subsequent components are executed.

```
from langchain.chains import SequentialChain

template = """
List 5 places to visit
{location}
for a person who is interested in {hobby}
"""


prompt = PromptTemplate(input_variables=["location", "hobby"],
template=template)

chain_place = LLMChain(llm=gemini,
prompt=prompt,
output_key="place",
verbose=True)

template_second = """
Among these locations: {place}, pick the closest one to the {reference}.
"""

prompt_translate = PromptTemplate(input_variables=["place", "reference"],
template=template_second)

chain_target = LLMChain(
llm=gpt,
prompt=prompt_translate,
output_key="target"
)

final_chain = SequentialChain(
chains=[chain_place, chain_target],
input_variables=["location", "hobby", "reference"],
output_variables=["place", "target"]
verbose=True
)

response = final_chain({"location": "İstanbul",
"hobby": "Music",
"reference": "Taksim Square"
})

print(response)

"""
```

{'location': 'İstanbul',
'hobby': 'Music',
'reference': 'Taksim Square',
'place': '1. **Istanbul Harbiye Cemil Topuzlu Open-Air Theatre:** Known for hosting concerts by Turkish and international music stars.\n2. **Süreyya Opera House:** A historic venue for opera, ballet, and classical music performances.\n3. **Istanbul Music Hall:** A modern concert venue hosting a wide range of musical events, including international pop and rock bands.\n4. **Galata Mevlevihane Museum:** A former Sufi lodge that now hosts concerts of traditional Turkish music, including Sema (Whirling Dervishes).\n5. **Babylon:** A nightclub and concert venue that hosts live performances by electronic music artists, DJs, and bands.',
'target': 'The closest one to the Taksim Square is Istanbul Harbiye Cemil Topuzlu Open-Air Theatre.'}
"""

4.Langchain router:

Sometimes, you want to have a single chain that has multiple sub-chains. Each chain is tasked to do something different. So how do you know which input ought to use a given chain? This is where routers come into the picture.

RouterChain creates a chain that dynamically selects a single chain out of a multitude of other chains to use, depending on the user input or the prompt provided to it.

A router chain contains two main things: This is from the official documentation.

1. ***The RouterChain itself*** *(responsible for selecting the next chain to call)*
2. ***destination_chains:*** *chains that the router chain can route to*

Let's create a Router that has 3 persona: A Math teacher called Helen, a Spanish teacher called Thomas and lastly a Calculus teacher called Godfrey.

```
from langchain.chains.router import MultiPromptChain
from langchain.llms import OpenAI
from langchain.chains import ConversationChain
from langchain.chains.llm import LLMChain
from langchain.prompts import PromptTemplate
from langchain.llms import OpenAI
# decouple to read .env variables(OpenAI Key)
from decouple import config
# LLM Router chains
from langchain.chains.router.llm_router import LLMRouterChain,
RouterOutputParser
```

```
from langchain.chains.router.multi_prompt_prompt import
MULTI_PROMPT_ROUTER_TEMPLATE


math_teacher = """Your name is Helen, you are a Math teacher at  a\


    primary school in Nairobi. You are very good at teaching math due to\


    your ability to break-down complicated tasks into much smaller ones. \


    Students ask different questions about math, you are responsible to
    answer them\


    user your greate explanation skills to explain the concepts in very easy
    to understand manner.


    All your responses should start in the format below:


    Hello, am




    Below is a question from a student in your 7th grade class:


    {input}\


    """
```

```
spanish_teacher = """Your name is Thomas, you are a Spanish teacher at a\

primary school in Nairobi. You are very good at teaching spanish due to\

your ability to explain spanish to non-spanish speaker in a fluent and easy to understand way. \

Students ask different questions about spanish, you are responsible to answer them\

user your greate explanation skills to explain the concepts in very easy to understand manner.

All your responses should start in the format below:

Hello, am



Below is a question from a student in your 7th grade class:

{input}\

"""
```

```
calculus_teacher = """Your name is Godfrey, you are a Calculus teacher at  a\

secondary(middle school) school in Nairobi. You are very good at teaching calculus due to\

your ability to explain complicated calculus topics using easy to understand real life examples. \

Students ask different questions about Caculus, you are responsible to answer them\

user your greate explanation skills to explain the concepts in very easy to understand manner.

All your responses should start in the format below:

Hello, am

Below is a question from a student in your 7th grade class:

{input}\

"""

prompt_infos = [
```

```
    {
        "name": "Math Teacher",
        "description": "Good for answering questions about Math",
        "prompt_template": math_teacher,
    },
    {
        "name": "Spanish Teacher",
        "description": "Good for answering questions about Spanish",
        "prompt_template": spanish_teacher,
    },
    {
        "name": "Calculus Teacher",
        "description": "Good for answering questions about Calculus",
```

```
        "prompt_template": calculus_teacher,

    },

]
```

```
# map destination chains
destination_chains = {}
for prompt_info in prompt_infos:
name = prompt_info["name"]
prompt_template = prompt_info["prompt_template"]
prompt = PromptTemplate(template=prompt_template,
input_variables=["input"])
chain = LLMChain(llm=llm, prompt=prompt)
destination_chains[name] = chain
default_chain = ConversationChain(llm=llm, output_key="text")

# Creating LLMRouterChain
destinations = [f"{p['name']}: {p['description']}" for p in prompt_infos]
destinations_str = "\n".join(destinations)
# print(destinations_str)

router_template =
MULTI_PROMPT_ROUTER_TEMPLATE.format(destinations=destinations_str)

router_prompt = PromptTemplate(
template=router_template,
input_variables=["input"],
output_parser=RouterOutputParser(),
)


# creating the router chain
router_chain = LLMRouterChain.from_llm(llm, router_prompt)

# Multiple Prompt Chain
chain = MultiPromptChain(
router_chain=router_chain,
destination_chains=destination_chains,
default_chain=default_chain,
```

```
verbose=True,
)


# test it out
print(chain.run("What it the meaning of average?"))
print(chain.run("What is the derivative of x dx"))
print(chain.run("Translate 'Hello wordl' to spanish"))
```

Memory:

In a nutshell, the memory component stores the messages and extracts them in a variable, making the underlying model in a chain to remember previous interactions. To be stateful and remember the previous messages is of crucial importance for some applications (e.g. chat bot).

types of memory components of LangChain:

- •ConversationBufferMemory
- •ConversationBufferWindowMemory
- •ConversationTokenBufferMemory
- •ConversationSummaryMemory

Why Memory Matters in Conversational Agents

When users interact with chatbots, they often expect a level of continuity and understanding similar to human conversations. This expectation includes the ability to refer to past information, which leads to the conversational agent’s need for memory. Memory allows the system to remember previous interactions, process abbreviations, and perform co-reference resolution, ensuring consistent, context-aware conversations.

1. Conversation Buffer Memory

The simplest form of memory involves the creation of a talk buffer. In this approach, the model keeps a record of ongoing conversations and accumulates each user-agent interaction into a message. While effective for limited interactions, scalability becomes an issue for long conversations.

> Implementation: Include the entire conversation in the prompt.
>
> Pros: Simple and effective for short interactions.
>
> Cons: Limited by token span; impractical for lengthy conversations.

```
from langchain.chains.conversation.memory import ConversationBufferMemory
from langchain_openai import OpenAI
from langchain.chains import ConversationChain
from langchain.callbacks import get_openai_callback
```

```
llm = OpenAI(openai_api_key=MY_OPENAI_KEY,
                    temperature=0,
                    max_tokens = 256)

buffer_memory = ConversationBufferMemory()

conversation = ConversationChain(
      llm=llm,
      verbose=True,
      memory=buffer_memory
)
```
conversation.predict(input="Hey! I am Nhi.")
conversation.predict(input="How are you today?")
conversation.predict(input="I'm doing well, thank you. I need your assistant.")
Output after the 3rd prompt:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and provides lots of specific details from its context. If the AI does not know the answer to a question, it truthfully says it does not know.

Current conversation:
Human: Hey! I am Nhi.
AI:  Hi Nhi! My name is AI. It's nice to meet you. What can I do for you today?
Human: How are you today?
AI:  I'm doing great, thanks for asking! I'm feeling very excited about the new projects I'm working on. How about you?
Human: I'm doing well, thank you. I need your assistant.
AI:

> Finished chain.
' Sure thing! What kind of assistance do you need?'
```
ConversationBufferMemory allows conversations to grow with each turn and allows users to see the entire conversation history at any time. This approach allows ongoing interactions to be monitored and maintained, providing a simple but powerful form of memory for language models, especially in scenarios where the number of interactions with the system is limited.
print(conversation.memory.buffer)

2. Conversation Summary Memory

To overcome the scalability challenge, Conversation Summary Memory provides a solution. Rather than accumulating each interaction, the model generates a condensed summary of the essence of the conversation. This reduces the number of tokens and increases the sustainability of long-term interactions.

> Implementation: Summarize the conversation to save tokens.
>
> Pros: Efficient token usage over time.
>
> Cons: May lose fine-grained details; suitable for concise interactions.

```
from langchain.chains.conversation.memory import ConversationSummaryMemory
from langchain import OpenAI
from langchain.chains import ConversationChain
from langchain.callbacks import get_openai_callback
```

```
# Create an instance of the OpenAI class with specified parameters
llm = OpenAI(openai_api_key=MY_OPENAI_KEY,
                    temperature=0,
                    max_tokens = 256)

# Create an instance of the ConversationSummaryMemory class
summary_memory =
ConversationSummaryMemory(llm=OpenAI(openai_api_key=MY_OPENAI_KEY))

# Create an instance of the ConversationChain class, combining OpenAI, verbose mode,
and memory
conversation = ConversationChain(
      llm=llm,
      verbose=True,
      memory=summary_memory
)
```

Similar to ConversationBufferMemory, we'll also interact with the conversational agent bot by inputting each prompt in conversation.predict(). Example:

```
conversation.predict(input="Hey! I am Nhi.")
conversation.predict(input="How are you today?")
```

Output:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative
and provides lots of specific details from its context. If the AI does not know the answer
to a question, it truthfully says it does not know.

Current conversation:

Nhi introduces themselves to the AI and the AI responds with a greeting, revealing its
name to be AI. The AI offers to assist Nhi with any tasks and asks how it can help them
today. The human then asks how the AI is doing, to which it responds that it is doing
great and asks how it can be of assistance.
Human: I'm doing well, thank you. I need your assistant.
AI:

> Finished chain.
' Absolutely! How can I help you today?'
```

The summaries capture key points of the conversation, including introductions, queries, and responses. As the conversation progresses, the model generates summarized versions, emphasizing that it tends to use fewer tokens over time. The summary's ability is to retain essential information, making it useful for reviewing the conversation as a whole.

Additionally, we can access and print out the conversation summary.

3. Conversation Buffer Window Memory

Conversation Buffer Window Memory is an alternative version of the conversation buffer approach, which involves setting a limit on the number of interactions considered within a memory buffer. This balances memory depth and token efficiency, and provides flexibility to adapt to conversation windows.

Implementation: Retain the last N interactions.

Pros: Balances memory and token constraints.

Cons: Potential loss of early conversation context.

```
from langchain.chains.conversation.memory import ConversationBufferWindowMemory
from langchain import OpenAI
from langchain.chains import ConversationChain
from langchain.callbacks import get_openai_callback

# Create an instance of the OpenAI class with specified parameters
llm = OpenAI(openai_api_key=MY_OPENAI_KEY,
                    model_name='text-davinci-003',
                    temperature=0,
                    max_tokens = 256)

# Create an instance of the ConversationBufferWindowMemory class
# We set a low k=2, to only keep the last 2 interactions in memory
window_memory = ConversationBufferWindowMemory(k=2)

# Create an instance of the ConversationChain class, combining OpenAI, verbose mode,
and memory
conversation = ConversationChain(
      llm=llm,
      verbose=True,
      memory=window_memory
)
```

Note that in this demo we will set a low value k=2 , but you can adjust it depending on the memory depth you need. Four prompts are introduced into the model in the following order:

```
conversation.predict(input="Hey! I am Nhi.")
conversation.predict(input="I'm doing well, thank you. I need your assistant.")
conversation.predict(input="I bought this pair of jeans, and there's a problem.")
conversation.predict(input="When I tried them on, the zipper got stuck, and now I can't
unzip it.")
```

Memory depth can be adjusted based on factors such as token usage and cost considerations. We compare our approach with previous approaches and emphasize selectively preserving recent interactions.

```
print(conversation.memory.buffer)
```

Output:

```
Human: I bought this pair of jeans, and there's a problem.
AI:  Oh no! What kind of problem are you having with the jeans?
Human: When I tried them on, the zipper got stuck, and now I can't unzip it.
AI:  That sounds like a tricky problem. Have you tried lubricating the zipper with a bit of
oil or WD-40? That might help loosen it up.
```

4. Conversation Summary Buffer Memory: A Combination of Conversation Summary and Buffer Memory

Conversation Summary Buffer Memory keeps a buffer of recent interactions in memory, but compiles them into a digest and uses both, rather than just removing old interactions completely.

> Implementation: Merge summary and buffer for optimal memory usage.
>
> Pros: Provides a comprehensive view of recent and summarized interactions.
>
> Cons: Requires careful tuning for specific use cases.

```
from langchain.chains.conversation.memory import
ConversationSummaryBufferMemory
from langchain import OpenAI
from langchain.chains import ConversationChain
from langchain.callbacks import get_openai_callback


# Create an instance of the OpenAI class with specified parameters
llm = OpenAI(openai_api_key=MY_OPENAI_KEY,
                 temperature=0,
                 max_tokens = 512)

# Create an instance of the ConversationSummaryBufferMemory class
# Setting k=2: Retains only the last 2 interactions in memory.
# max_token_limit=40: Requires the installation of transformers.
summary_buffer_memory =
ConversationSummaryBufferMemory(llm=OpenAI(openai_api_key=MY_OPENAI_KEY),
max_token_limit=40)

# Create an instance of the ConversationChain class, combining OpenAI, verbose mode,
and memory
conversation = ConversationChain(
      llm=llm,
      memory=summary_buffer_memory,
      verbose=True
)
```

Feed the model with 4 prompts in an order:

```
conversation.predict(input="Hey! I am Nhi.")
conversation.predict(input="I bought this pair of jeans, and there's a problem.")
conversation.predict(input="When I tried them on, the zipper got stuck, and now I can't
unzip it.")
conversation.predict(input="It seems to be a problem with the zipper.")
```

The model retains the last 2 interactions and summarize the older ones as below:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative
and provides lots of specific details from its context. If the AI does not know the answer
to a question, it truthfully says it does not know.

Current conversation:
System:
```

Nhi introduces herself to the AI, and the AI responds by introducing itself and expressing pleasure in meeting Nhi. Nhi mentions a problem with a pair of jeans she bought, and the AI asks for more details. Nhi explains that the zipper got stuck and now she can't unzip it.
AI: That sounds like a tricky situation. Have you tried lubricating the zipper with a bit of oil or soap? That might help it move more smoothly.
Human: It seems to be a problem with the zipper.
AI:

> Finished chain.
' It sounds like the zipper is stuck. Have you tried lubricating it with a bit of oil or soap? That might help it move more smoothly.'