

Langchain

[LangChain: Models, Prompts and Output Parsers](#)

[LangChain: Memory](#)

[Chains in LangChain](#)

[LangChain: Q&A over Documents](#)

[LangChain: Evaluation](#)

[LangChain: Agents](#)

Overview



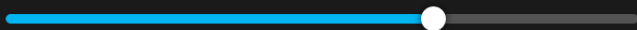
Open-source development framework for LLM applications

Python and JavaScript (TypeScript) packages

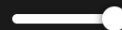
Focused on composition and modularity

Key value adds:

1. Modular components
2. Use cases: Common ways to combine components

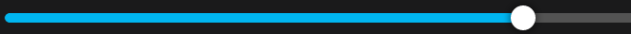


2:07 / 3:06



Components

- **Models**
 - LLMs: 20+ integrations
 - Chat Models
 - Text Embedding Models: 10+ integrations
- **Prompts**
 - Prompt Templates
 - Output Parsers: 5+ implementations
 - Retry/fixing logic
 - Example Selectors: 5+ implementations
- **Indexes**
 - Document Loaders: 50+ implementations
 - Text Splitters: 10+ implementations
 - Vector stores: 10+ integrations
 - Retrievers: 5+ integrations/implementations
- **Chains**
 - Prompt + LLM + Output parsing
 - Can be used as building blocks for longer chains
 - More application specific chains: 20+ types
- **Agents**
 - Agent Types: 5+ types
 - Algorithms for getting LLMs to use tools
 - Agent Toolkits: 10+ implementations
 - Agents armed with specific tools for a specific application



2:35 / 3:06



LangChain: Models, Prompts and Output Parsers

****Study Notes: Lesson One - Models, Prompts, and Parsers****

In this lesson, we delve into the concepts of models, prompts, and parsers in the context of language processing and understanding using OpenAI's language models. We will explore how these concepts interplay and contribute to creating efficient and effective applications.

****1. Models:****

- Models refer to the underlying language models that power various natural language processing tasks. These models are capable of understanding and generating text in a human-like manner.
- In the context of OpenAI's models, we often use models like GPT (Generative Pre-trained Transformer) 3.5 Turbo, which is designed to generate coherent and contextually relevant text.

****2. Prompts:****

- Prompts are inputs provided to language models to generate specific types of outputs. They guide the models in producing desired responses or completing tasks.
- The style of creating prompts involves crafting text that communicates the desired task or inquiry to the model.
- Prompts can be formulated in various ways to achieve different outcomes, including translations, question answering, summarization, etc.

****3. Parsers:****

- Parsers involve taking the output generated by language models and structuring it into a more organized and usable format.
- Parsing is especially useful when dealing with complex outputs that need to be interpreted, stored, or processed further.
- Parsing allows us to extract relevant information from the model's output and convert it into a structured data format.

****Key Concepts and Applications:****

- ****LangChain Abstractions:**** LangChain introduces a set of abstractions that simplify the process of interacting with language models, making it easier to create applications that utilize models effectively.
- ****Reusable Models:**** Applications built using Large Language Models (LLMs) often involve reusing models to repeatedly prompt for desired outputs.
- ****Building Applications:**** The process involves prompting a model, parsing its output, and utilizing the parsed data for downstream processing.
- ****Prompt Engineering:**** Effective prompt engineering involves specifying prompts clearly to achieve desired outcomes. Language style, specific instructions, and input variables play a role in crafting effective prompts.

- **Translation Example:** An example scenario involves translating a customer's message from English Pirate to polite American English. The process entails setting styles, crafting prompts, and utilizing prompt templates to achieve consistency and ease of use.

Why use prompt templates?

```
prompt = """
Your task is to determine if
the student's solution is
correct or not.
```

```
To solve the problem do the following:
- First, work out your own solution to the problem.
- Then compare your solution to the student's solution
and evaluate if the student's solution is correct or not.
...
Use the following format:
Question:
...
question here
...
Student's solution:
...
student's solution here
...
Actual solution:
...
steps to work out the solution and your solution here
...
Is the student's solution the same as actual solution \
just calculated:
...
yes or no
...
Student grade:
...
correct or incorrect
...

Question:
...
{question}
...
Student's solution:
...
{student_solution}
...
Actual solution:
...
"""
```

Prompts can be long and detailed.

Reuse good prompts when you can!

LangChain also provides prompts for common operations.

- **LangChain's Chat Prompt Template:** LangChain provides tools like chat prompt templates to create reusable prompts. It abstracts away complex prompt creation and allows for easy modification and reuse.

LangChain output parsing works with prompt templates

```
EXAMPLES = ["""
Question: What is the elevation range
for the area that the eastern sector
of the Colorado orogeny extends into?

Thought: need to search Colorado orogeny, find
the area that the eastern sector of the Colorado
orogeny extends into, then find the elevation range
of the area.

Action: Search[Colorado orogeny]

Observation: The Colorado orogeny was an
episode of mountain building (an orogeny) in
Colorado and surrounding areas.

Thought: It does not mention the eastern sector.
So I need to look up eastern sector.
Action: Lookup[eastern sector]

...

Thought: High Plains rise in elevation from
around 1,800 to 7,000 ft, so the answer is 1,800 to
7,000 ft.

Action: Finish[1,800 to 7,000 ft]""",
]
```

LangChain library
functions parse the
LLM's output
assuming that it will
use certain keywords.

Example here uses
Thought, Action,
Observation as
keywords for Chain-
of-Thought
Reasoning. (ReAct)



9:38 / 18:23


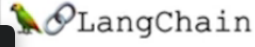


- **Output Parsing:** Output parsing is critical for making sense of model-generated content. LangChain's output parser helps extract meaningful information from the output and format it in a structured manner.
- **Example: Product Review Parsing:** An example demonstrates how LangChain's output parser can be used to extract specific details from a product review and format them as structured data (JSON).
- **Formatting Instructions:** LangChain's parser provides format instructions that guide the language model's output, making it easier to parse and use in downstream applications.
- **Practicality:** LangChain's abstractions and tools simplify the process of using language models in real-world applications, reducing the need for extensive prompt engineering and output processing.

****Conclusion:****

In this lesson, we explored the core concepts of models, prompts, and parsers. These concepts are essential for building applications that utilize language models effectively. With LangChain's abstractions, prompt templates, and output parsers, developers can streamline the process of interacting with language models and create powerful and efficient applications that understand and generate human-like text.

LangChain: Memory

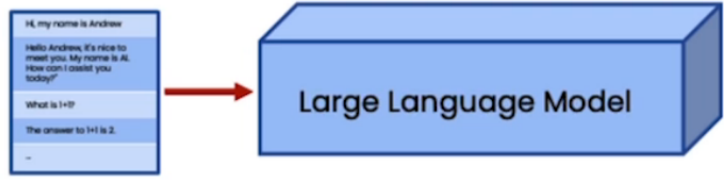


Memory


Large Language Models are 'stateless'

- Each transaction is independent

Chatbots appear to have memory by providing the full conversation as 'context'



LangChain provides several kinds of 'memory' to store and accumulate the conversation



In this section, we explore the concept of memory within the context of language models, specifically focusing on LangChain's sophisticated memory management options. Memory is crucial for maintaining context and facilitating conversational flow, particularly when building applications like chatbots that require continuous interaction.

****1. Background: Memory and Conversation Flow****

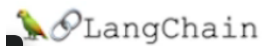
- When interacting with language models, they don't inherently retain memory of prior interactions or conversations.

- This limitation becomes apparent when building applications like chatbots, where maintaining a conversational history is vital for coherent and context-aware interactions.

****2. Introducing Memory Management with LangChain****

- LangChain addresses the issue of memory by providing various options for managing conversation history and context.
- Memory management involves retaining and using past conversation segments to guide the language model's responses.

****3. Types of Memory in LangChain:****



Memory Types

ConversationBufferMemory

- This memory allows for storing of messages and then extracts the messages in a variable.

ConversationBufferWindowMemory

- This memory keeps a list of the interactions of the conversation over time. It only uses the last K interactions.

ConversationTokenBufferMemory

- This memory keeps a buffer of recent interactions in memory, and uses token length rather than number of interactions to determine when to flush interactions.

ConversationSummaryMemory

- This memory creates a summary of the conversation over time.



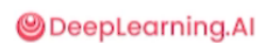
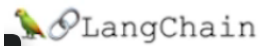
15:17 / 17:04



- **ConversationBufferMemory:** This memory type stores an ongoing conversation, allowing for the retrieval of prior interactions. It maintains a record of the entire conversation, making it easy to access context.

- **ConversationBufferWindowMemory:** Unlike the previous memory type, this variant only retains a window of recent conversation, which can be controlled using the "k" parameter. It helps manage memory size while retaining recent context.

- **ConversationSummaryBufferMemory:** This type leverages the language model to generate a summary of the conversation history. It's particularly useful when memory limitations exist, as it provides a concise overview of the conversation.



Additional Memory Types

Vector data memory

- Stores text (from conversation or elsewhere) in a vector database and retrieves the most relevant blocks of text.

Entity memories

- Using an LLM, it remembers details about specific entities.

You can also use multiple memories at one time.

E.g., Conversation memory + Entity memory to recall individuals.

You can also store the conversation in a conventional database (such as key-value store or SQL)



15:39 / 17:04



- **Vector Data Memory:** This advanced memory type utilizes word and text embeddings to store and retrieve relevant blocks of text. It's beneficial for applications that involve complex data retrieval based on embeddings.

- **Entity Memory:** This memory type is used to remember details about specific entities, such as individuals, for reference during interactions.

4. Working with Memory in LangChain:

- LangChain supports the integration of multiple memory types, allowing developers to combine them for different applications.
- The chosen memory type depends on the application's requirements, such as maintaining conversation history or recalling information about specific entities.
- Memory management helps enhance interactions by providing contextual information for language models, making them more conversational and context-aware.

5. Practical Usage:

- LangChain's memory management is particularly useful when creating applications like chatbots that require continuous conversation flow and context retention.
- Memory can be crucial for maintaining user engagement and providing relevant responses based on prior interactions.

Conclusion:

In this section, we explored how LangChain tackles the challenge of memory management in language models. Memory plays a pivotal role in maintaining conversational context and enhancing interactions, especially in applications like chatbots. The various memory types offered by LangChain allow developers to choose the most suitable approach for their specific use cases, resulting in more dynamic and contextually aware language model interactions.

Chains in LangChain

****Study Notes: Key Building Blocks of LangChain: The Chain****

In this lesson, we delve into the fundamental building block of LangChain, known as the "chain." The chain combines a large language model (LLM) with a prompt, and it serves as the foundation for creating sequential operations on text or other data. This concept enables the creation of complex operations by linking multiple chains together, allowing data to flow through a sequence of steps.

****1. Introduction to the Chain:****

- The chain is an essential component of LangChain that facilitates sequential operations on input data.
- Combines a large language model (LLM) with a prompt, creating a unit that can process and transform data.

****2. Using Data with Chains:****

- Loading environment variables and data is a common initial step.
- Data can be organized in structures like pandas DataFrames, allowing for batch processing of inputs.

****3. LLM Chain:****

- The simplest chain type combines an LLM with a prompt.
- An LLM initializes with a high temperature to encourage diverse output.
- A prompt template is defined, taking in variables and generating text that guides the LLM.
- The LLM chain executes the prompt with the LLM, producing contextual outputs based on inputs.

Sequential Chains



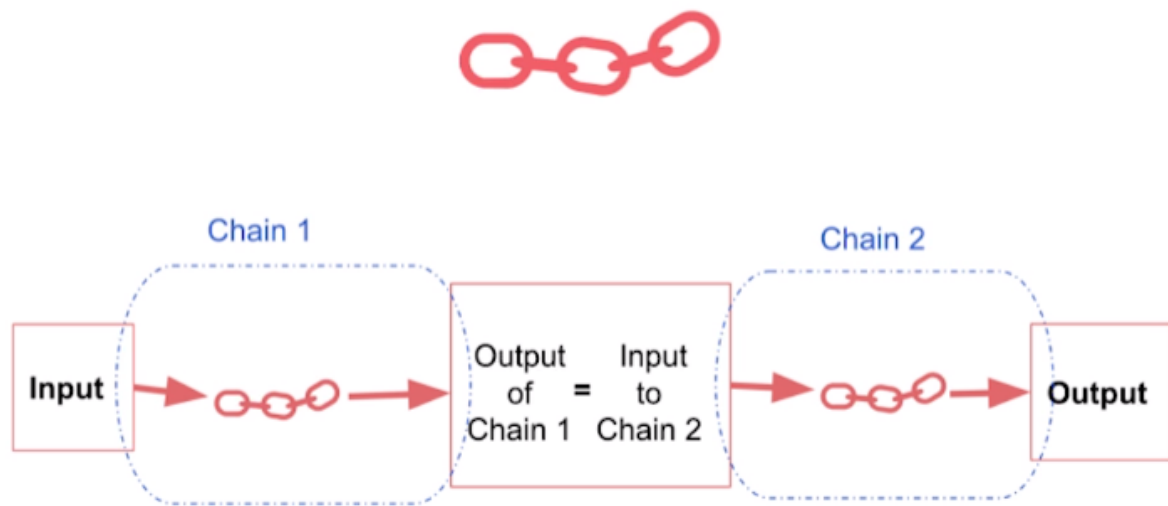
Sequential chain is another type of chains.

The idea is to combine multiple chains where the output of the one chain is the input of the next chain.

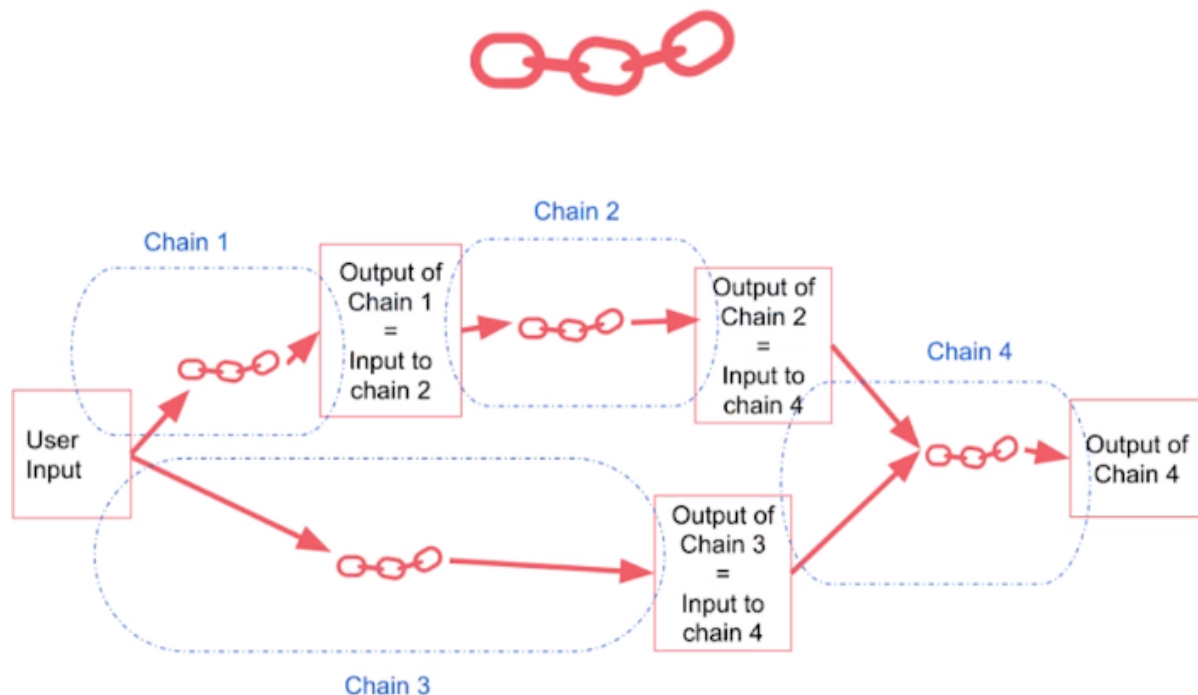
There is two type of sequential chains:

1. SimpleSequentialChain: Single input/output
2. SequentialChain: multiple inputs/outputs

Simple Sequential Chain



Sequential Chain



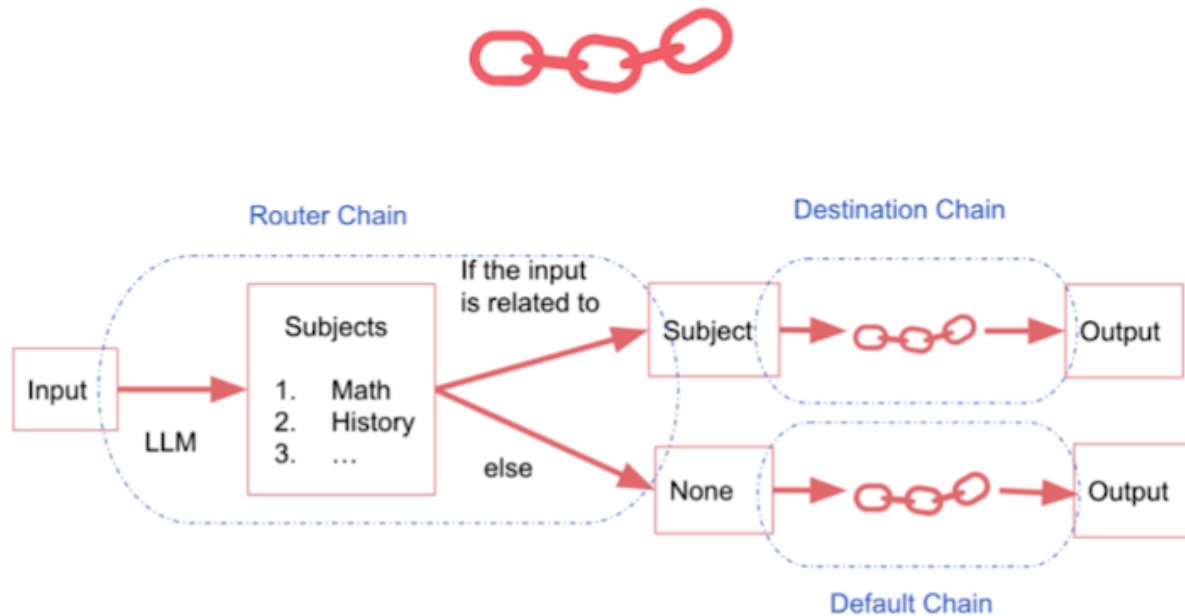
****4. Sequential Chains:****

- Simple Sequential Chain: Utilized when subchains require a single input and output.
- Sequential chains run multiple subchains one after another, forming a sequence of operations.
- Inputs from previous steps are passed to subsequent chains for processing.

****5. Sequential Chains with Multiple Inputs and Outputs:****

- Regular Sequential Chain: Handles cases where subchains require multiple inputs or outputs.
- Each step of the chain takes inputs from previous steps and returns corresponding outputs.
- Variables must be carefully aligned between steps to avoid key errors.

Router Chain



6. Routing Inputs to Chains:

- Router Chains: Used to route inputs to specific subchains based on conditions.
- A multi-prompt chain is introduced for routing among prompt templates.
- LLM Router Chain: Uses an LLM to decide which subchain to pass inputs to.
- Router Output Parser: Converts LLM outputs into dictionaries to guide further processing.

7. Constructing a Chain:

- Language model initialization and destination chains are defined.
- Templates for subchains are created, specifying task details and output format.
- The router chain is constructed using the LLM, templates, and output parser.
- An overall chain combines the router chain, destination chains, and a default chain for unmatched inputs.

8. Practical Usage of Chains:

- Chains can be used to process and transform data in complex ways.
- Router chains are valuable for directing inputs to specialized subchains based on conditions.
- Chained operations enable dynamic and tailored data processing.

Conclusion:

In this lesson, we explored the concept of the "chain" in LangChain, which serves as the fundamental building block for creating sequential operations on data. By combining large language models (LLMs) with prompts, chains enable a sequence of steps that can process,

transform, and route data. These chains provide a versatile tool for developing applications that require complex data processing, context-awareness, and specialized routing based on input conditions. The flexibility and power of chains make them an essential component of LangChain's toolkit.

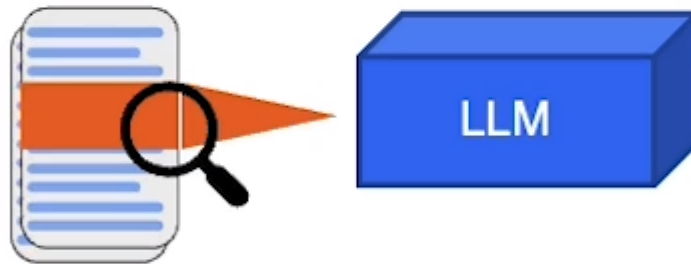
LangChain: Q&A over Documents

****Study Notes: Question Answering Using Language Models and Vector Stores****

****Introduction:****

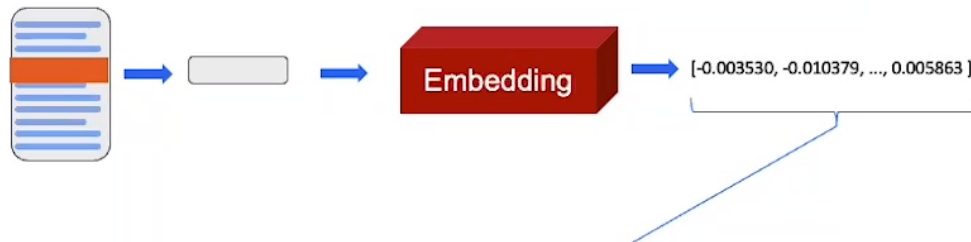
- Complex applications using LLM (Large Language Models) involve answering questions about documents.
- LLMs help users understand and access information within documents, even beyond their training data.
- Key components include embedding models, vector stores, and LLMs.

LLM's on Documents



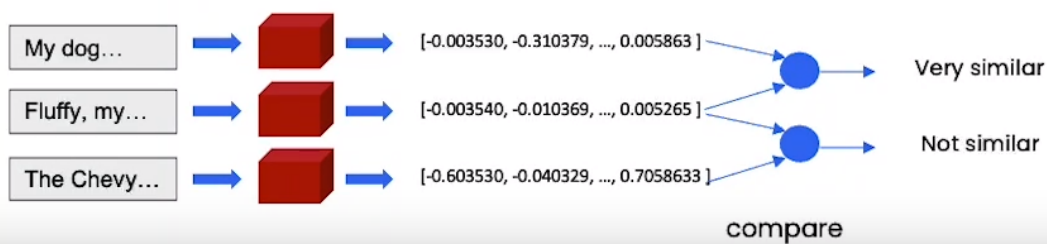
LLM's can only inspect a few thousand words at a time.

Embeddings

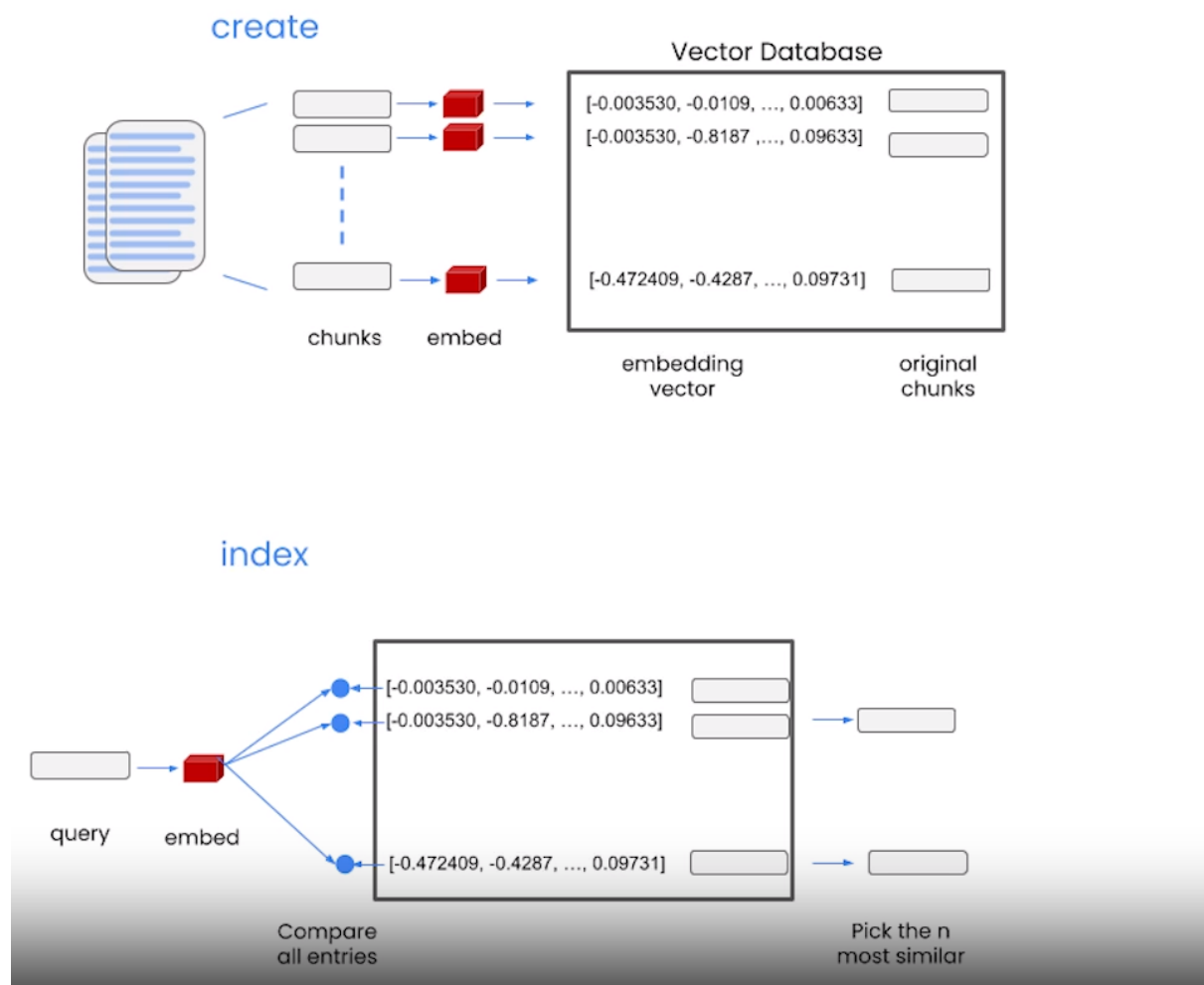


- Embedding vector captures content/meaning
- Text with similar content will have similar vectors

- 1) My dog Rover likes to chase squirrels.
- 2) Fluffy, my cat, refuses to eat from a can.
- 3) The Chevy Bolt accelerates to 60 mph in 6.7 seconds.



Vector Database



****Process Overview:****

1. ****Embeddings:**** Create numerical representations of text capturing semantic meaning, allowing comparison of text pieces in a vector space.
2. ****Vector Stores:**** Store embedding representations. Break large documents into smaller chunks for efficient processing.
3. ****Vector Database Creation:**** Chunks from documents are embedded and stored in a vector store, creating a vector database.
4. ****Query and Retrieval:**** Incoming queries are embedded, compared to database vectors, and the most similar documents are retrieved.
5. ****LLM Generation:**** Retrieved documents are passed to an LLM for text generation to answer queries.

****Detailed Steps:****

1. ****Document Loading:****
 - Load documents, often from CSV, containing content to be queried.

2. **Embedding Creation:**

- Use embeddings to transform text into numerical vectors.
- Similar content results in similar vectors.
- Embeddings capture semantic meaning.

3. **Vector Store Creation:**

- Populate a vector store with chunks of text from documents.
- Create an embedding for each chunk.
- Store embeddings in the vector store.

4. **Query Processing:**

- Create an embedding for the incoming query.
- Compare the query embedding to database vectors.
- Retrieve the most similar documents.

5. **LLM Response:**

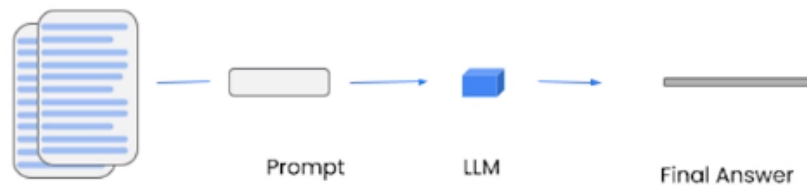
- Pass the retrieved documents to an LLM.
- Generate responses that answer the query.

LangChain for Question Answering:

- **LangChain:** A framework for building complex language processing pipelines.
- Combine retrieval and LLM-based question answering using a chain.
- Create retrievers to fetch documents and LLMs to generate responses.

Methods for Combining Documents:

Stuff method



Stuffing is the simplest method. You simply stuff all data into the prompt as context to pass to the language model.

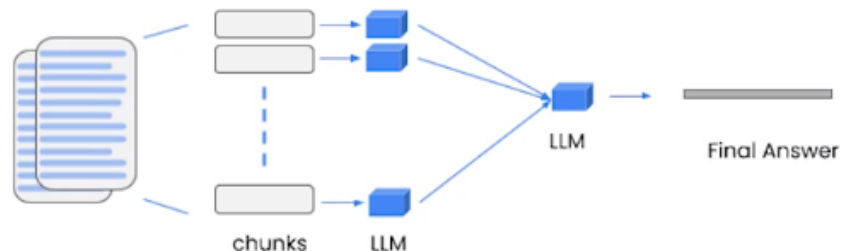
Pros: It makes a single call to the LLM. The LLM has access to all the data at once.

Cons: LLMs have a context length, and for large documents or many documents this will not work as it will result in a prompt larger than the context length.

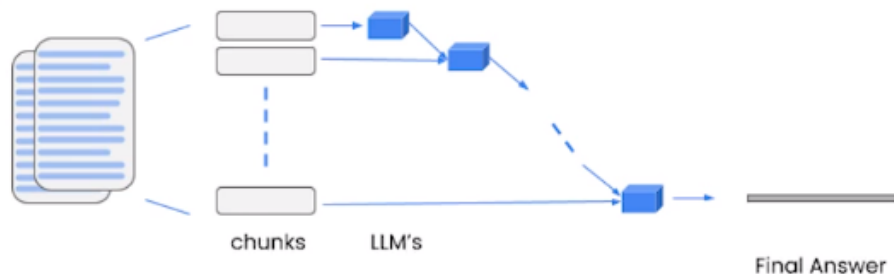
- ****Stuff Method:**** Combine all documents into a single prompt for the LLM. Simple and efficient..

3 additional methods

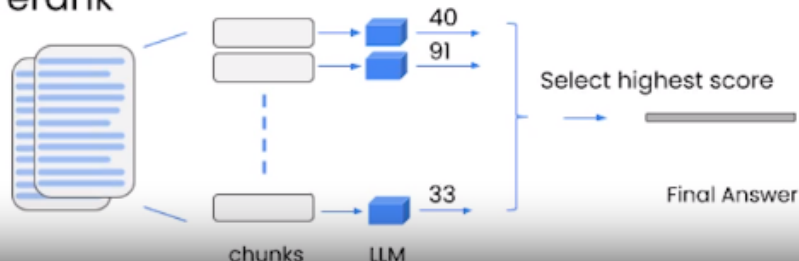
1. Map_reduce



2. Refine



3. Map_rerank



- **Map-Reduce Method:** Divide documents into chunks, query each chunk separately with LLM, and summarize results.
- **Refine Method:** Iteratively build up an answer, refining it with each additional document.
- **Map-Rerank Method:** Call LLM for each document, assign a relevance score, and select the highest-scoring document.

Conclusion:

- Question answering with LLMs involves embedding, vector stores, retrieval, and LLM response generation.
- Different methods like "stuff," "map-reduce," "refine," and "map-rerank" offer varying trade-offs for combining documents and generating answers.
- LangChain provides a flexible framework to streamline complex language processing tasks.

These study notes summarize the process of using Large Language Models (LLMs) to answer questions based on documents, integrating embeddings, vector stores, retrieval methods, and LLM-based response generation. The notes also touch on different methods for combining documents and generating answers, highlighting the flexibility of the LangChain framework.

LangChain: Evaluation

****Study Notes on Evaluating LLM-based Applications:****

When developing complex applications using Language Models (LLMs), it's important to evaluate their performance effectively. Evaluating LLM-based applications involves assessing their accuracy, understanding how changes affect implementation, and employing tools to aid in this process. This study note covers key concepts related to evaluating LLM-based applications.

****1. Importance of Evaluation:****

- Evaluating LLM-based applications helps determine their performance and accuracy.
- Changes in implementation, LLMs, or other components should be evaluated to ensure improvements and avoid regressions.

****2. Manual Example Creation:****

- Manually creating example question-answer pairs for evaluation is a straightforward but time-consuming approach.
- Analyze data to generate relevant examples and ground truth answers.

****3. Automated Example Generation:****

- Language models can help automate the generation of question-answer pairs for evaluation.
- Using a dedicated chain, such as the QA generation chain in LangChain, examples can be automatically created.

****4. Debugging with langchain.debug:****

- The langchain.debug setting allows in-depth examination of a chain's execution.
- Print detailed information about each step in the chain's execution, including input, output, and interactions with language models.

****5. Evaluation Chain:****

- The evaluation process involves comparing predicted answers with actual ground truth answers.
- The QA evaluation chain in LangChain can be utilized to assess the correctness of LLM-generated responses.

****6. Challenges in String Comparison:****

- Evaluating text responses is challenging due to the variability in acceptable answers.
- String matching and exact matching are often inadequate for LLM-based applications' evaluation.

****7. Evaluating with Language Models:****

- Using a language model to evaluate LLM-generated responses provides a more nuanced assessment.
- Language models can understand semantic similarity, allowing for meaningful evaluation.

****8. LangChain Evaluation Platform:****

- The LangChain Evaluation Platform offers a user interface to visualize and analyze the evaluation process.
- Track inputs, outputs, and execution steps, and even create and manage example datasets for evaluation.

****9. Building Datasets for Evaluation:****

- Creating datasets of example question-answer pairs is crucial for ongoing evaluation.
- The Evaluation Platform facilitates the creation and management of datasets for systematic evaluation.

In conclusion, evaluating LLM-based applications involves manual and automated example creation, debugging with `langchain.debug`, comparing responses with ground truth answers using evaluation chains, and utilizing language models for nuanced assessment. The LangChain Evaluation Platform further aids in visualizing and managing the evaluation process, allowing for comprehensive and ongoing evaluation of LLM-based applications.

LangChain: Agents

****Study Notes on LangChain's Agents Framework:****

LangChain's Agents framework enhances the utility of large language models (LLMs) by enabling them to act as reasoning engines rather than mere knowledge stores. This study note provides an overview of key concepts related to LangChain's Agents framework.

****1. LLM as a Reasoning Engine:****

- LLMs can be thought of as reasoning engines, utilizing background knowledge and new information to answer questions and make decisions.
- LangChain's Agents framework leverages this capability to create intelligent applications.

****2. Introduction to Agents:****

- Agents in LangChain enable LLMs to interact with tools and external data sources for problem-solving and decision-making.
- Agents act as intermediaries between LLMs and various tools, allowing them to perform specific tasks.

****3. Power of Agents:****

- Agents are a powerful and emerging aspect of LangChain, allowing integration with various tools and data sources.
- They offer the potential to create sophisticated and context-aware applications.

****4. Initializing Agents:****

- To begin using Agents, environment variables are imported, and necessary packages like "duckduckgo-search" and "wikipedia" are installed.
- Relevant methods, classes, and modules are imported from LangChain.

****5. Agent Initialization:****

- Agents are initialized using the `initialize_agent()` method, which takes language models, tools, and agent types as parameters.
- The agent type defines the agent's behavior and characteristics.

****6. Handling Parsing Errors:****

- Setting `handle_parsing_errors=True` ensures that LLM outputs are properly parsed using output parsers for downstream use.

****7. Using Agents for Information Retrieval:****

- Agents can recognize when to use specific tools for information retrieval.
- For example, asking about a recent event not known during training will prompt the agent to use a search tool like DuckDuckGo.

****8. Utilizing Wikipedia Tool:****

- Agents can leverage built-in tools like Wikipedia for retrieving information.
- When asking about historical figures or entities, agents recognize the need to refer to Wikipedia for detailed answers.

****9. Custom Tools with AgentType**:**

- Agents can be extended with custom tools to access external data sources.
- The `@tool` decorator is used to create custom tools that can be connected to the agent.
- A detailed docstring guides the agent on when and how to use the tool.

****10. Incorporating Custom Tools**:**

- Custom tools are integrated with agents by including them in the list of existing tools during agent initialization.

****11. Demonstrating Custom Tool Usage**:**

- Agents can utilize custom tools to perform specific tasks.
- A custom tool that retrieves the current date is demonstrated, where the agent recognizes the need to use the custom tool and provides a relevant response.

****12. Conclusion on Agents**:**

- Agents represent a cutting-edge and experimental aspect of LangChain.
- They empower LLMs to act as reasoning engines, connect with various tools, and make informed decisions based on context and data sources.