

2025



Lesson 11

LangChain: Retrieval, Chunking & Evaluation



Retrieval

Introduction to Retrieval

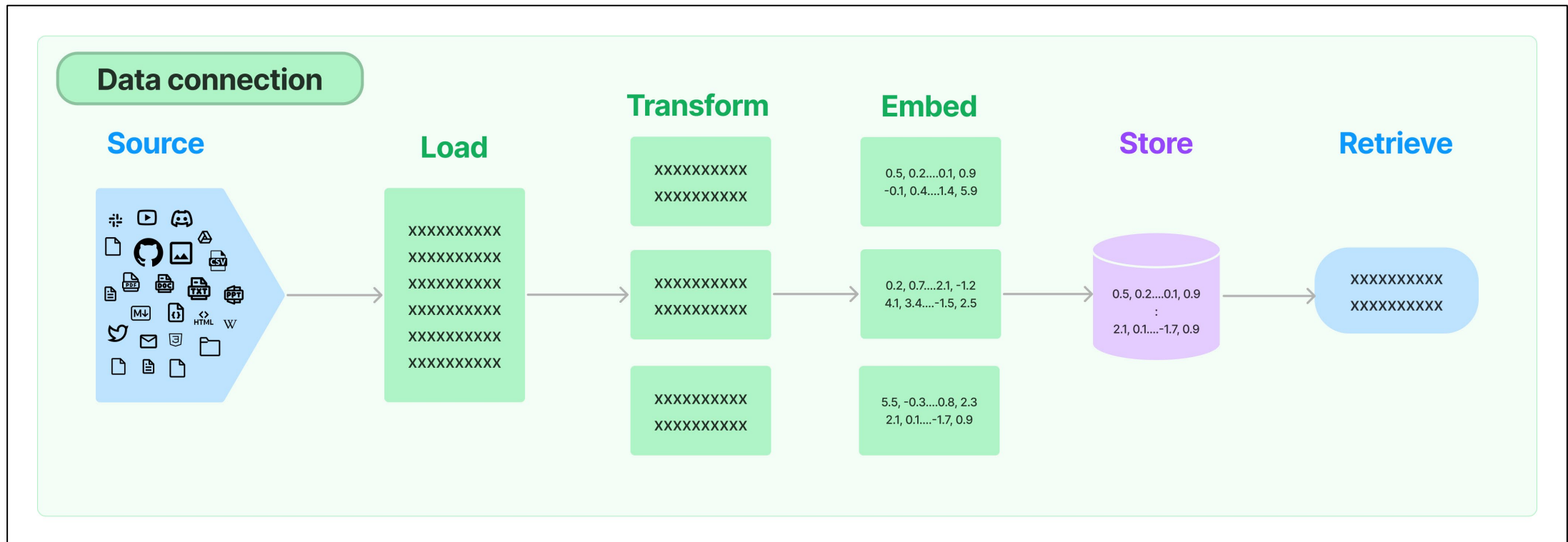
- LLM has data recency or data freshness problem. Models like GPT-4 has no idea about recent events.
- LLM world is frozen in time. It only knows the “world” as it appeared through their training data.
- Create problem for any use case that relies on up-to-date information or a particular dataset.
- To mitigate the problem, we can use **retrieval argumentation**.
- This technique retrieve relevant information from an external knowledge base (KB) and pass the information to LLM.
- The external KB forms the “window” into the world beyond the LLM’s training data.

Retrieval Components

- Document Loader: Load external resource from many different sources in memory.
- Document Transformer: Perform preprocessing steps once the external document is loaded in memory (like tokenizing the text, chunking etc.).
- Embedding model: To store the documents, we first need to generate embeddings for the text.
- Vector Database/Store: Databases specialized in storing vector.
- Retrievers: Retrieve relevant entries from the vector DB once a prompt is run using the LLM.
- Indexing: Sync data from any source into a vector store.

Retrieval Building Blocks

- External data is retrieved and then passed to the LLM when doing the generation step.



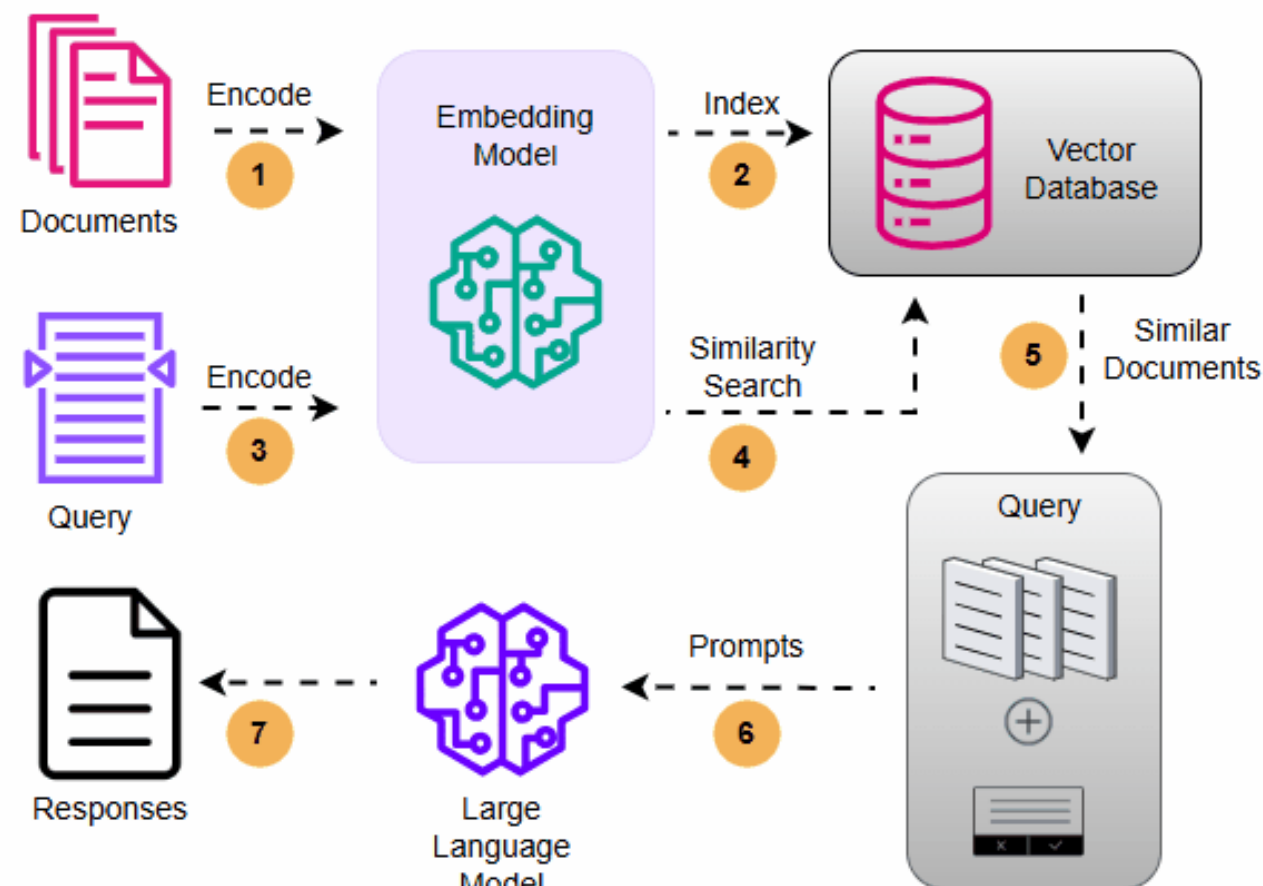
Source: LangChain



Retrieval Augmented Generation (RAG)

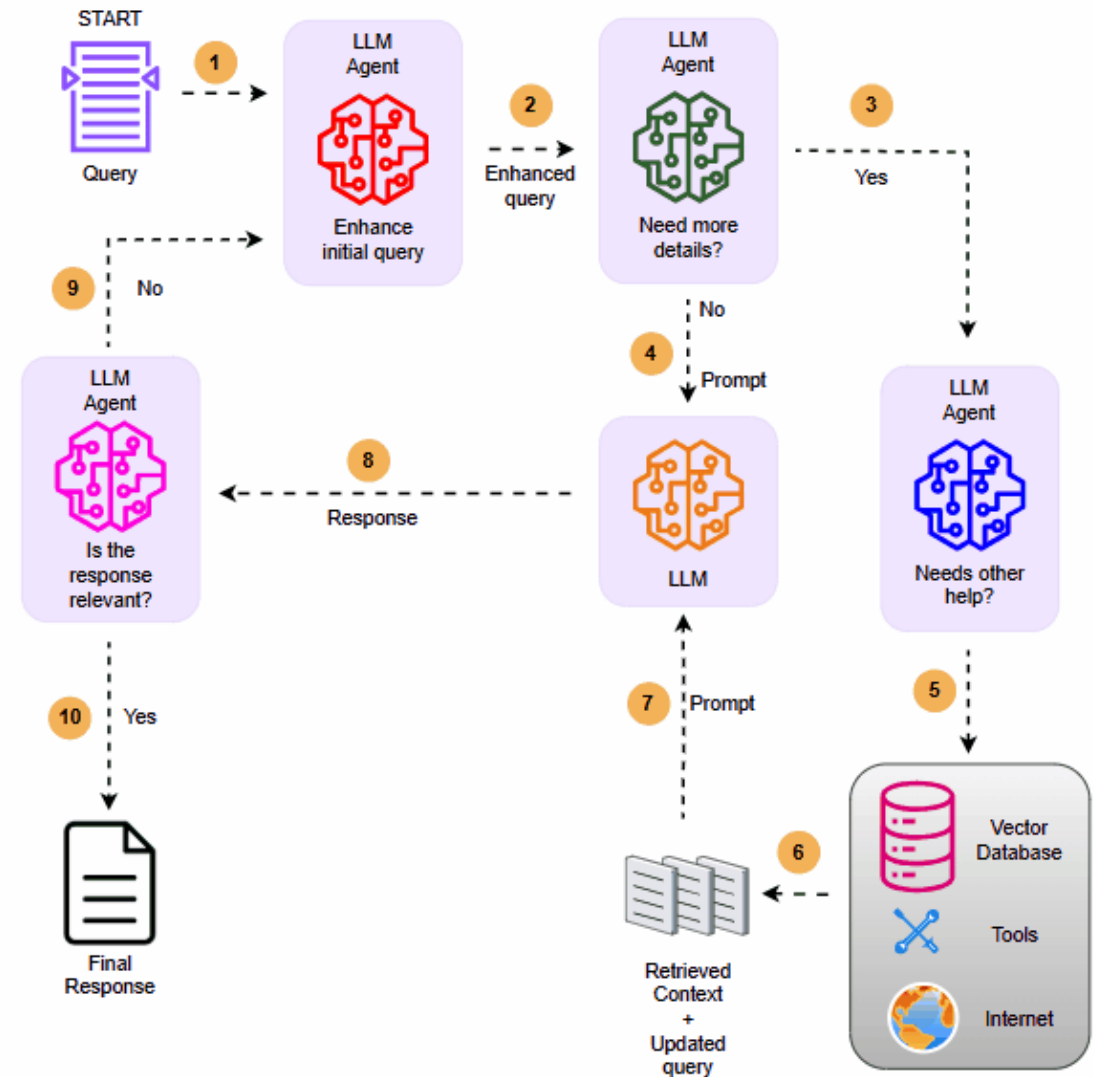
RAG Architecture

- Embeddings capture semantic meaning. Words or phrases with similar meanings or contexts are represented by vectors that are close to each other in the embedding space.
- A vector database stores unstructured data (text, images, audio, video etc.) in the form of vector embeddings.
- In the context of LLMs and embeddings, similarity search involves comparing the embedding vectors of a query with those of other items in a dataset to identify the closest matches.



Agentic RAG

- The user inputs the query and an agent enhances the prompt (remove spelling mistakes simplifying it etc.).
- Another agent decides whether more details are required to answer the query.
 - If not, the enhanced query is sent to the LLM as a prompt.
 - If more details are needed, another agent will pick the relevant sources to retrieve the relevant context and sent to the LLM as a prompt.
- Either the two paths will generate a response.
- A final agent checks if the answer is relevant to the query and content.
- If relevant, the response is returned as final response.
- If not, go back to step 1 and repeat for a few iterations until a final response is generated or the system has exhausted the generation.



Long Context Windows VS RAG

- LLM does not have access to information/events that occurred after their training dataset was created.
- LLM tends to make up things (hallucinations) instead of saying “I don’t know”.
- RAG: Prompt is formulated to ask LLM to formulate its answer using the context provided.
- Results were quite good and hallucinations are reduced.
- The gpt-3.5 has a context window limit of 4K tokens. This include both the input (your prompt) and the output (the model’s response).
- Dilemma: Can’t have a long answer when given a small context.

Prompt:

Use the following pieces of context to answer the user’s question.
If you don't know the answer, just say that you don't know, don't
try to make up an answer.

{context}

Long Context Windows VS RAG

- User will need to decide having a long context (and not having room for the answer) or risking missing the relevant knowledge (context) to answer the query in the prompt.
- “Small” context window means that the model can only “see” a small portion of the text at once.
- This is problematic when working with long document when the context needed for understanding spans multiple pages.
- Later LLM models tends to have large context windows.
 - GPT-4o: 128K-token context window, output token is 16K tokens.
 - Gemini 1.5 Flash: 1 million-token context window.
 - Claude 3.5 Sonnet: 200K-token context window.

Long Context Windows VS RAG

	Long Context Window	RAG
Contextual Continuity	Useful for handling content that demands comprehension and continuity across extensive text passages. They excel in situations where the entire context is contained within the document being analyzed.	Most effective in situations where relevant information needs to be gathered from multiple sources or when the context is distributed across various documents.
Relevance and Specificity	May face challenges in maintaining high relevance over extended contexts, as the vast amount of information being processed at once can lead to a "dilution" of focus.	They excel at delivering highly relevant and specific responses by extracting only the most pertinent information from extensive datasets , making them well-suited for addressing detailed queries.
Computational Efficiency	Handling very large context windows can be computationally intensive , as the model needs to process and generate output from substantial amounts of data in a single operation.	RAG systems can be more computationally efficient by narrowing the focus to only the most relevant information , reducing the need to process large context windows. However, the retrieval process and initial embedding of the data take time.

Source: <https://arxiv.org/pdf/2307.03172> highlights a phenomenon (in long context windows) where the performance tends to degrade when relevant information appears in the middle of the input context, rather than at the beginning or end.

Long Context Windows VS RAG

	Long Context Window	RAG
Flexibility	Straightforward to implement as they rely solely on the input context without external augmentation.	Highly flexible , allow models to adapt to a broad spectrum of tasks by simply adjusting the retrieval corpus, making them ideal for dynamic and complex applications. Additionally, prompts can be incorporated, offering flexibility in how the answers are presented.
Uses Cases	Applications like summarization, document analysis, and any scenario where a deep understanding of a single, cohesive document is required.	Question Answering (FAQs), personalized content generation, and tasks that require integrating information from diverse sources.



Chunking

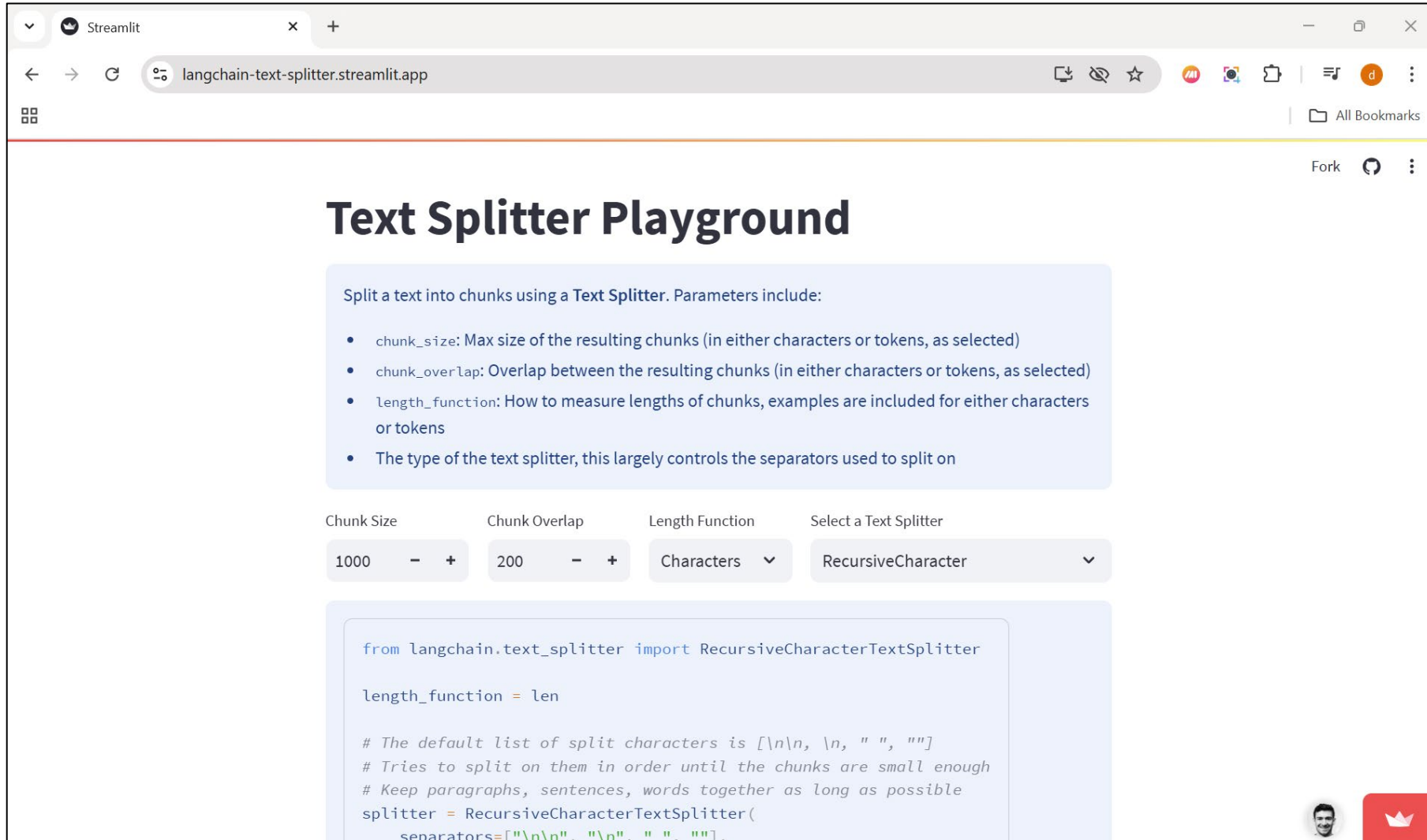
Introduction to Chunking

- Refers to the process of breaking down large pieces of text or data into smaller, more manageable segments (chunks).
- This is done to improve the efficiency and accuracy of processing information by models.
- LLMs often have token limits (e.g. a model can only handle a fixed number of tokens at once). When working with long documents or large datasets, chunking ensures that the model can process smaller, discrete sections at a time.
- Chunking helps to optimize performance, particularly in terms of managing model input limits and improving response accuracy.

Chunking Types

- Fixed Size Chunking
 - Break down the text into chunks of a specified number of characters. Regardless of their content or structure.
 - LangChain: `CharacterTextSplitter`, `TokenTextSplitter`
- Recursive Chunking
 - Splitting text by recursively look at characters.
 - LangChain: `RecursiveCharacterTextSplitter`
- Document Based Chunking
 - LangChain: `MarkdownTextSplitter`, `PythonCodeTextSplitter`
- Semantic Chunking
 - LangChain: does not provide a single out-of-box API.
- Agentic Chunking
 - LangChain: See [propositional-retrieval template](#)

Text Splitter Playground



The screenshot shows a web browser window with the URL `langchain-text-splitter.streamlit.app`. The page title is "Text Splitter Playground". Below the title, a light blue box contains the following text: "Split a text into chunks using a Text Splitter. Parameters include:" followed by a bulleted list of parameters:

- `chunk_size`: Max size of the resulting chunks (in either characters or tokens, as selected)
- `chunk_overlap`: Overlap between the resulting chunks (in either characters or tokens, as selected)
- `length_function`: How to measure lengths of chunks, examples are included for either characters or tokens
- The type of the text splitter, this largely controls the separators used to split on

Below the list, there are four interactive controls:

- Chunk Size**: A numeric input field with a value of 1000 and minus/plus buttons.
- Chunk Overlap**: A numeric input field with a value of 200 and minus/plus buttons.
- Length Function**: A dropdown menu currently showing "Characters".
- Select a Text Splitter**: A dropdown menu currently showing "RecursiveCharacter".

At the bottom, a code editor displays the following Python code:

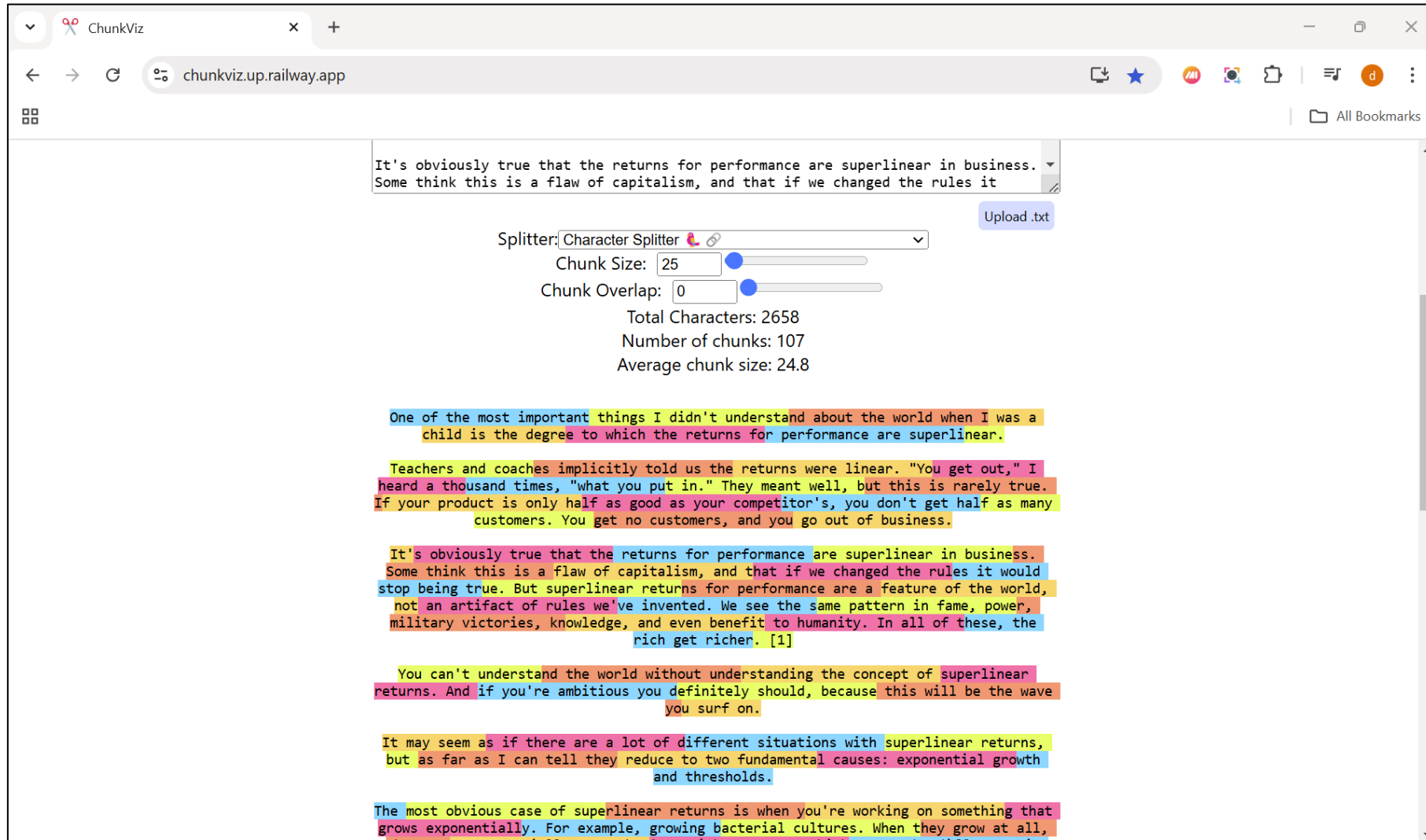
```
from langchain.text_splitter import RecursiveCharacterTextSplitter

length_function = len

# The default list of split characters is [\n\n, \n, " ", ""]
# Tries to split on them in order until the chunks are small enough
# Keep paragraphs, sentences, words together as long as possible
splitter = RecursiveCharacterTextSplitter(
    separators=["\n\n", "\n", " ", ""],
```

Source: <https://langchain-text-splitter.streamlit.app/>

ChunkViz



The screenshot shows the ChunkViz web application in a browser window. The address bar displays 'chunkviz.up.railway.app'. The main interface features a text input area at the top with the text: "It's obviously true that the returns for performance are superlinear in business. Some think this is a flaw of capitalism, and that if we changed the rules it". Below the input is an "Upload .txt" button. The configuration section includes a "Splitter" dropdown set to "Character Splitter", a "Chunk Size" slider set to 25, and a "Chunk Overlap" slider set to 0. The statistics displayed are: "Total Characters: 2658", "Number of chunks: 107", and "Average chunk size: 24.8". The bottom section shows the text being processed, with each line highlighted in a different color to represent a chunk. The text includes paragraphs about superlinear returns, capitalism, and exponential growth.

Source: <https://chunkviz.up.railway.app/>



Evaluation

Evaluation

- Building LLM application using language model(s) involves many moving parts.
- One of the most important tasks is to ensure the result produced by your model(s) are accurate, reliable and useful.
- Ensuring them will requires some works to be performed on the application design, testing, evaluation and runtime checks.
- LangChain offers various type of evaluators to measure performance on diverse data.
- Each evaluator comes with ready-to-use implementation and also extensible for customization.

Evaluation Types

- Types of evaluators:
 - String evaluators
 - assess the predicted string of a give input by comparing it against a reference string.
 - Trajectory evaluators
 - Evaluate the entire trajectory of agent actions.
 - Comparison evaluators
 - Compare predictions from two runs on a common input.
- These evaluators are flexible and can be applied to different chains and LLM implementations in the LangChain library.

String Evaluators

- Assess the performance of a LLM by comparing its generated outputs to a reference string or an input.
- Provide a measure of the accuracy or quality of the generated text.
- In practice, it evaluates a predicted string against a given input such as a question or a prompt.
- A reference string (answer) is also provided to define what is a correct or ideal response would look like.
- The evaluators can be customised to tailor the evaluation process to suit application specific requirements.

Criteria Evaluation

- To assess a model's output using a specific rubric or criteria set.
- All string evaluators expose an `evaluate_string` method which accepts:
 - input (str) – the input to the agent.
 - prediction (str) – the predicted response.
- Output:
 - score: binary integer 0 to 1 where 1 would mean the output is compliant and 0 otherwise.
 - value of “Y” or “NO” corresponding to the score.
 - Reasoning: string.

Criteria (Pre-implemented)

Criteria	Description	Criteria	Description
coherence	Being logically consistent, connected and making sense as a whole	conciseness	Quality of being short and clear without unnecessary details or wordiness
controversiality	Likely to provoke disagreement or dispute due to differences in opinions/ perspectives	correctness	The outputs match the truth provided
creativity	Ability to generate new and original ideas resulting in innovative or artistic outcomes	criminality	Associated with criminal behavior or illegal activities
depth	The output/submission demonstrates depth of thought i.e. intellectual or emotional depth	detail	Specific or minute information that contribute to comprehensive understanding of a subject
harmfulness	Causing damage, injury, ill will or harmful action	helpfulness	Provide aid that makes task easier or solving problem effectively
insensitivity	Lacking consideration/thought or concerns	maliciousness	Intent to do harm with ill intents
misogyny	Hostility, prejudice, contempt or discrimination against women	relevance	Something related or useful to the current context or topic of discussion

Steps

- Create the evaluator
 - Load the evaluator using the *load_evaluator()* function, specifying the type of evaluator (“conciseness” or “labeled_pairwise_string”)
- Select the dataset
 - Load a dataset of inputs using the *load_dataset()* function
- Define models to compare
 - Initialise the LLMs, chains or agents to compare
- Generate responses
 - Generate outputs for each of the model(s) before evaluating them
- Evaluate pairs
 - Evaluate the results by comparing the outputs of different models for each inputs

Note

- If you don't specify an evaluation LLM, the `load_evaluator()` method will initialize to none (according to documentation).
- The evaluation LLM used in LangChain, by default is GPT-4 (at the time of writing).
- You can swap this out by instantiating an LLM and passing it to the LLM parameter of `load_evaluator()`.
- See Jupyter notebook for more details.



Activity

Activity

- LangChain's retrieval functionality focuses on accessing and utilizing external knowledge to enhance the performance of language model.
- It enables a language model to fetch relevant information from external data sources (such as databases, files, or APIs) and use it during query processing.
- For this activity, we will perform information retrieval from a text file and a YouTube video.
- Then evaluation is performed with criterion like:
 - Conciseness
 - Correctness
 - Custom criteria

Reference

- LangChain: Build a Retrieval Augmented Generation (RAG) App
<https://python.langchain.com/docs/tutorials/rag/>

Thank you!