7 Ways to Employ LangChain Text Splitters for Enhanced Data Processing

LANGCHAIN LARGE LANGUAGE MODELS LLMS PYTHON

Introduction

In our previous article about LangChain Document Loaders, we explored how LangChain's document loaders facilitate loading various file types and data sources into an LLM application. Can we send the data to the LLM now? Not so fast. LLMs have limits on context window size in terms of token numbers, so any data more than that size will be cut off, leading to potential loss of information and less accurate responses. Even if the context size is infinite, more input tokens will lead to higher costs, and money is not infinite. So, rather than sending all the data to the LLM, it is better to send the data that is relevant to our query about the data. To achieve this, we need to split the data first, and for that, LangChain Text Splitters is required. Now, let's learn about LangChain Text Splitters.

Overview

1. **Understand the Importance of Text Splitters in LLM Applications:** Learn why text splitting is crucial for optimizing large language models (LLMs) and its impact on context window size and cost efficiency.
2. **Learn Different Methods of Text Splitting:** Explore various text-splitting techniques, including character count, token count, recursive splitting, HTML structure, and code syntax.
3. **Implement Text Splitters Using LangChain:** Learn to use LangChain's text splitters, including installing them, writing code to split text, and handling different data formats.
4. **Apply Semantic Splitting for Enhanced Relevance:** Use sentence embeddings and cosine similarity to identify natural breakpoints, ensuring semantically similar content stays together.

Table of contents

What are Text Splitters?

Text splitters split large volumes of text into smaller chunks so that we can retrieve more relevant content for the given query. These splitters can be applied directly to raw text or to document objects loaded using LangChain's document loaders.

Several methods are available for splitting data, each tailored to different types of content and use cases. Here are the various ways we can employ text splitters to enhance data processing.

Also read: A Comprehensive Guide to Using Chains in Langchain

Methods for Splitting Data

LangChain Text Splitters are essential for handling large documents by breaking them into manageable chunks. This improves performance, enhances contextual understanding, allows parallel processing, and facilitates better data management. Additionally, they enable customized processing and robust error handling, optimizing NLP tasks and making them more efficient and accurate. Further, we will discuss methods to split data into manageable chunks.

Pre-requisites

First, install the package using `'pip install langchain_text_splitters'`

By Character Count

The text is split based on the number of characters. We can specify the separator to use for splitting the text. Let's understand using the code. You can download the document used here: Free Strategy Formulation E-Book.

```
from langchain_community.document_loaders import UnstructuredPDFLoader from langchain_text_splitters import CharacterTextSplitter # load the data loader = UnstructuredPDFLoader('how-to-formulate-successful-business-strategy.pdf', mode='single') data = loader.load()
```

```
text_splitter = CharacterTextSplitter( separator="\n", chunk_size=500, chunk_overlap=0, is_separator_regex=False, )
```

This function splits the text where each chunk has a maximum of 500 characters. Text will be split only at new lines since we are using the new line (“\n”) as the separator. If any chunk has a size more than 500 but no new lines in it, it will be returned as such.

```
texts = text_splitter.split_documents(data) # Created a chunk of size 535, which is longer than the specified 500 # Created a chunk of size 688, which is longer than the specified 500 len(texts) >>> 73


for i in texts[48:49]: print(len(i.page_content)) print(i.page_content)
```

Output

As we can see, the above-displayed chunk has 688 characters.

Recursive

Rather than using a single separator, we use multiple separators. This method will use each separator sequentially to split the data until the chunk reaches less than chunk_size. We can use this to split the text by each sentence, as shown below.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter loader = UnstructuredPDFLoader('how-to-formulate-successful-business-strategy.pdf', mode='single') data = loader.load() recursive_splitter = RecursiveCharacterTextSplitter( separators=["\n\n", "\n", r"(?<=[.?!])\s+"], keep_separator=False, is_separator_regex=True, chunk_size=30, chunk_overlap=0) texts = recursive_splitter.split_documents(data) len(texts) >>> 293 # a few sample chunks for text in texts[123:129]: print(len(text.page_content)) print(text.page_content)
```

Output

As we can see, we have mentioned three separators, with the third one for splitting by sentence using regex.

By Token count

Both of the above methods use character counts. Since LLMs use tokens to count, we can also split the data by token count. Different LLMs use different token encodings. Let us use the encoding used in GPT-4o and GPT-4o-mini. You can find the model and encoding mapping here—GitHub link.

```
from langchain_text_splitters import TokenTextSplitter text_splitter = TokenTextSplitter(encoding_name='o200k_base', chunk_size=50, chunk_overlap=0) texts = text_splitter.split_documents(data) len(texts) >>> 105
```

We can also use character text splitter methods along with token counting.

```
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder( encoding_name='o200k_base', separators=["\n\n", "\n", r"(?<=[.?!])\s+"], keep_separator=False, is_separator_regex=True, chunk_size=10, # chunk_size is number of tokens chunk_overlap=0) texts = text_splitter.split_documents(data) len(texts) >>> 279 for i in texts[:4]: print(len(i.page_content)) print(i.page_content)
```

Output

As shown, we can use token counting along with a recursive text splitter.

Among the three methods mentioned above, a recursive splitter with either character or token counting is better for splitting plain text data.

HTML

While the above methods are fine for plain text, If the data has some inherent structure like HTML or Markdown pages, it is better to split by considering that structure.

We can split the HTML page based on the headers

```
from langchain_text_splitters import HTMLHeaderTextSplitter, HTMLSectionSplitter headers_to_split_on = [ ("h1", "Header 1"), ("h2", "Header 2"), ("h3", "Header 3")] html_splitter = HTMLHeaderTextSplitter(headers_to_split_on, return_each_element=True) html_header_splits = html_splitter.split_text_from_url('https://diataxis.fr/') len(html_header_splits) >>> 37
```

Here, we split the HTML page from the URL by headers h1, h2, and h3. We can also use this class by specifying a file path or HTML string.

```
for header in html_header_splits[20:22]: print(header.metadata) >>> {'Header 1': 'Diátaxis¶'} {'Header 1': 'Diátaxis¶', 'Header 2': 'Contents¶'} # there is no h3 in this page.
```

Similarly, we can also split Markdown file text using headers with `MarkdownHeaderTextSplitter`

We can also split based on any other sections of the HTML. For that, we need HTML as a text.

```
import requests r = requests.get('https://diataxis.fr/') sections_to_split_on = [ ("h1", "Header 1"), ("h2", "Header 2"), ("p", "section"), ] html_splitter = HTMLSectionSplitter(sections_to_split_on) html_section_splits = html_splitter.split_text(r.text) len(html_section_splits) >>> 18 for section in html_section_splits[1:6]: print(len(section.page_content)) print(section)
```

Output

Here, we use h1, h2, and p tags to split the data in the HTML page.

Code

Since Programming languages have different structures than plain text, we can split the code based on the syntax of the specific language.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter, Language PYTHON_CODE = """ def add(a, b): return a + b class Calculator: def __init__(self): self.result = 0 def add(self, value): self.result += value return self.result def subtract(self, value): self.result -= value return self.result # Call the
```

```
function def main(): calc = Calculator() print(calc.add(5)) print(calc.subtract(2)) if __name__ == "__main__": main() """ python_splitter = RecursiveCharacterTextSplitter.from_language( language=Language.PYTHON, chunk_size=100, chunk_overlap=0) python_docs = python_splitter.create_documents([PYTHON_CODE]) python_docs
```

Output

Here, the Python code is split based on the syntax words like class, def, etc. We can find separators for different languages here – GitHub Link.

JSON

A nested json object can be split such that initial json keys are in all the related chunks of text. If there are any long lists inside, we can convert them into dictionaries to split. Let's look at an example.

```
from langchain_text_splitters import RecursiveJsonSplitter # Example JSON object json_data = { "company": { "name": "TechCorp", "location": { "city": "Metropolis", "state": "NY" }, "departments": [ { "name": "Research", "employees": [ {"name": "Alice", "age": 30, "role": "Scientist"}, {"name": "Bob", "age": 25, "role": "Technician"} ] }, { "name": "Development", "employees": [ {"name": "Charlie", "age": 35, "role": "Engineer"}, {"name": "David", "age": 28, "role": "Developer"} ] } ] }, "financials": { "year": 2023, "revenue": 1000000, "expenses": 750000 } } # Initialize the RecursiveJsonSplitter with a maximum chunk size splitter = RecursiveJsonSplitter(max_chunk_size=200, min_chunk_size=20) # Split the JSON object chunks = splitter.split_text(json_data, convert_lists=True) # Process the chunks as needed for chunk in chunks: print(len(chunk)) print(chunk)
```

Output

This splitter maintains initial keys such as company and departments if the chunk contains data corresponding to those keys.

Semantic Splitter

The above methods work based on the text's structure. However, splitting two sentences may not be helpful if they have similar meanings. We can utilize sentence embeddings and cosine similarity to identify natural break points where the semantic content of adjacent sentences diverges significantly.

Here are the Steps:

1. Split the input text into individual sentences.
2. Combine Sentences Using Buffer Size: Create combined sentences for a given buffer size. For example, if there are 10 sentences and the buffer size is 1, the combined sentences would be:
    - Sentences 1 and 2
    - Sentences 1, 2, and 3
    - Sentences 2, 3, and 4
    - Continue this pattern until the last combination, which will be sentences 9 and 10
3. Compute the embeddings for each combined sentence using an embedding model.
4. Determine sentence splits based on distance:
    - Calculate the cosine distance (1 – cosine similarity) between adjacent combined sentences
    - Identify indices where the cosine distance is above a defined threshold.
    - Join the sentences based on those indices

```
from langchain_community.document_loaders import WikipediaLoader from langchain_experimental.text_splitter import SemanticChunker from langchain_openai.embeddings import OpenAIEmbeddings # make sure to add OPENAI_API_KEY loader = WikipediaLoader(query='Generative AI', load_max_docs=1, doc_content_chars_max=5000, load_all_available_meta=True) data = loader.load() semantic_splitter = SemanticChunker(OpenAIEmbeddings(model='text-embedding-3-small'), buffer_size=1, breakpoint_threshold_type='percentile', breakpoint_threshold_amount=70) texts = semantic_splitter.create_documents([data[0].page_content]) len(texts) >>> 10 for text in texts[:2]: print(len(text.page_content)) print(text.page_content)
```

Output

The document is split into 10 chunks, and there are 29 sentences in it. We have these breakpoint threshold types available along with default values:

“percentile”: 95,

“standard_deviation”: 3,

“interquartile”: 1.5,

“gradient”: 95

If you’re interested in learning how embedding models compute embeddings for sentences, look for the next article, where we’ll discuss the details.

Conclusion

This article explored various text-splitting methods using LangChain, including character count, recursive splitting, token count, HTML structure, code syntax, JSON objects, and semantic splitter. Each method offers unique advantages for processing different data types, enhancing the efficiency and relevance of the content sent to LLMs. By understanding and implementing these techniques, you can optimize data for better accuracy and lower costs in your LLM applications.

Frequently Asked Questions

Q1. What are text splitters in LangChain?

Ans. Text splitters are tools that divide large volumes of text into smaller chunks, making it easier to process and retrieve relevant content for queries in LLM applications.

Q2. Why is it necessary to split text before sending it to an LLM?

Ans. Splitting text is crucial because LLMs have limits on context window size. Sending smaller, relevant chunks ensures no information is lost and lowers processing costs.

Q3. What methods are available for splitting text in LangChain?

Ans. LangChain offers various methods such as splitting by character count, token count, recursive splitting, HTML structure, code syntax, JSON objects, and semantic splitting.

Q4. How do I implement a text splitter in LangChain?

Ans. Implementing a text splitter involves installing the LangChain package, writing code to specify splitting criteria, and applying the splitter to different data formats.

Q5. What is semantic splitting, and when should it be used?

Ans. Semantic splitting uses sentence embeddings and cosine similarity to keep semantically similar content together. It’s ideal for maintaining the context and meaning in text chunks.

Article Url - https://www.analyticsvidhya.com/blog/2024/07/langchain-text-splitters/

Dsanr