Implement agentic chunking to optimize LLM inputs with Langchain and watsonx.ai

What is Agentic chunking?

The way language models process and segment text is changing from the traditional static approach, to a better, more responsive process. Unlike traditional fixed-size chunking , which chunks large documents at fixed points, agentic chunking employs AI-based techniques to analyze content in a dynamic process, and to determine the best way to segment the text.

Agentic chunking makes use of AI-based text-splitting methods, recursive chunking, and chunk overlap methods, which work concurrently to polish chunking ability, preserving links between notable ideas while optimizing contextual windows in real time. With agentic chunking, each chunk is enriched with metadata to deepen retrieval accuracy and overall model efficiency. This is particularly important in RAG applications applications , where segmentation of data can directly impact retrieval quality and coherence of the response. Meaningful context is preserved in all the smaller chunks, making this approach incredibly important to chatbots, knowledge bases, and generative ai (gen ai) use cases. Frameworks like Langchain or LlamaIndex further improve retrieval efficiency, making this method highly effective.

Key elements of Agentic chunking

1. Adaptive chunking strategy: Dynamically choose the best chunking method based on the type of content, the intent behind the query and the needs for retrieval to ensure effective segmentation.

2. Dynamic chunk sizing: Modifying chunk sizes in real time by considering the semantic structure and context, instead of sticking to fixed token limits.

3. Context-preserving overlap: Smartly assessing the overlap between chunks to keep coherence intact and avoid losing essential information, thereby enhancing retrieval efficiency.

Advantages of Agentic chunking over traditional methods

Agentic chunking offers advantages over traditional chunking:

a. Retains context: Maintains crucial information without unnecessary breaks.

b. Smart sizing: Adjusts chunk boundaries according to meaning and significance.

c. Query-optimized: Continuously refines chunks to match specific queries.

d. Efficient retrieval: Improves search and RAG systems output by minimizing unnecessary fragmentation.

In this tutorial, you will experiment with agentic chunking strategy by using the IBM Granite-3.0-8B-Instruct model now available on watsonx.ai®. The overall goal is to perform efficient chunking to effectively implement RAG.

Your subscription will be delivered in English. You will find an unsubscribe link in every newsletter. You can manage your subscriptions or unsubscribe here. Refer to our IBM Privacy Statement for more information.

Prerequisite

You need an IBM Cloud account® to create a watsonx.ai project.

Steps

Step 1. Set up your environment

While you can choose from several tools, this tutorial walks you through how to set up an IBM account to use a Jupyter Notebook.

1. Log in to watsonx.ai by using your IBM Cloud account.

2. Create a watsonx.ai project.
   You can get your project ID from within your project. Click the Manage tab. Then, copy the project ID from the Details section of the General page. You need this ID for this tutorial.

3. Create a Jupyter Notebook.

This step opens a notebook environment where you can copy the code from this tutorial. Alternatively, you can download this notebook to your local system and upload it to your watsonx.ai project as an asset. To view more Granite® tutorials, check out the IBM Granite Community. This Jupyter Notebook along with the datasets used can be found on GitHub.

Step 2. Set up a watsonx.ai Runtime instance and API key

1. Create a watsonx.ai Runtime service instance (select your appropriate region and choose the Lite plan, which is a free instance).

2. Generate an API Key.

3. Associate the watsonx.ai Runtime service instance to the project that you created in watsonx.ai.

Step 3. Install and import relevant libraries and set up your credentials

You will need few libraries and modules for this tutorial. Make sure to import the following ones and if they're not installed, a quick pip installation resolves the problem.

Note, this tutorial was built using Python 3.12.7

```
!pip install -q langchain langchain-ibm langchain_experimental langchain-text-splitters
langchain_chroma transformers bs4 langchain_huggingface sentence-transformers
import getpass import requests from bs4 import BeautifulSoup from langchain_ibm import
WatsonxLLM from langchain_huggingface import HuggingFaceEmbeddings from
langchain_community.document_loaders import WebBaseLoader from langchain.schema import
SystemMessage, HumanMessage from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder from
langchain.prompts import ChatPromptTemplate from langchain.vectorstores import Chroma from
langchain.tools import tool from langchain.agents import AgentExecutor from langchain.memory
import ConversationBufferMemory from transformers import AutoTokenizer from
ibm_watsonx_ai.foundation_models.utils.enums import EmbeddingTypes from
ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams from
langchain.output_parsers import CommaSeparatedListOutputParser from langchain.embeddings
import HuggingFaceEmbeddings from langchain.schema import Document from
langchain.chains.combine_documents import create_stuff_documents_chain
```

To set our credentials, we need the **"WATSONX_APIKEY"** and **"WATSONX_PROJECT_ID"** . We will also set the URL serving as the API endpoint.

```
load_dotenv(os.getcwd()+"/.env", override=True) credentials = {     "url": "https://us-
south.ml.cloud.ibm.com",     "apikey": os.getenv("WATSONX_APIKEY", ""), } project_id =
os.getenv("PROJECT_ID", "")
```

Step 4. Initialize your language model.

For this tutorial, we suggest using IBM's Granite-3.0-8B-Instruct model as the LLM to achieve similar results. You are free to use any AI model of your choice. The foundation models available through watsonx can be found here.

```
llm = WatsonxLLM(     model_id="ibm/granite-3-8b-instruct",     url=credentials.get("url"),
   apikey=credentials.get("apikey"),     project_id=project_id,     params={
      GenParams.DECODING_METHOD: "greedy",         GenParams.TEMPERATURE: 0,
      GenParams.MIN_NEW_TOKENS: 5,         GenParams.MAX_NEW_TOKENS: 250,
      GenParams.STOP_SEQUENCES: ["Human:", "Observation"],     }, )
```

Step 5. Load your document

This function extracts the text content from IBM's explainer page on machine learning. This function removes unwanted HTML elements (scripts, styles), and returns clean, readable text.

```
def get_text_from_url(url):     response = requests.get(url)     if response.status_code != 200:         raise ValueError(f"Failed to fetch the page, status code: {response.status_code}")     soup = BeautifulSoup(response.text, "html.parser")     for script in soup(["script", "style"]):  script.decompose()     return soup.get_text(separator="\n", strip=True) url = "https://www.ibm.com/think/topics/machine-learning" web_text = get_text_from_url(url) web_text
```

Instead of using a fixed-length chunking method, we used an LLM to split the text based on meaning. This function leverages an LLM to intelligently split text into semantically meaningful chunks based on topics.

```
def agentic_chunking(text):     """     Dynamically splits text into meaningful chunks using LLM.     """    system_message = SystemMessage(content="You are an AI assistant helping to split text into meaningful chunks based on topics.")         human_message = HumanMessage(content=f"Please divide the following text into semantically different, separate and meaningful chunks:\n\n{text}")  response = llm.invoke([system_message, human_message]) # LLM returns a string     return response.split("\n\n") # Split based on meaningful sections
chunks = agentic_chunking(web_text) chunks
```

Let's print the chunks for better understanding of their output structure.

```
for i, chunk in enumerate(chunks,1):     print(f"Chunk {i}:\n{chunk}\n{'-'*40}")
```

Great! The chunks were successfully created by the agents in the output.

Step 6: Create a vector store

Now that we have experimented with agentic chunking on the text, let's move along with our RAG implementation.

For this tutorial, we choose the chunks produced by the agents and convert them to vector embeddings. An open source vector store that we can use is Chroma DB. We can easily access Chroma functionality through the langchain_chroma package. Let's initialize our Chroma vector database, provide it with our embeddings model and add our documents produced by agentic chunking.

```
embeddings_model = HuggingFaceEmbeddings(model_name="ibm-granite/granite-embedding-30m-english")
```

Create a Chroma vector database

```
vector_db = Chroma(     collection_name="example_collection",  embedding_function=embeddings_model )
```

Convert each text chunk into a document object

```
documents = [Document(page_content=chunk) for chunk in chunks]
```

Add the documents to the vector database.

```
vector_db.add_documents(documents)
```

Step 7: Structure the prompt template

Now, we can create a prompt template for our LLM. This template ensures that we can ask multiple questions while maintaining a consistent prompt structure. Additionally, we can integrate our vector store as the retriever, finalizing the RAG framework.

```
prompt_template = """<|start_of_role|>user<|end_of_role|>Use the following pieces of context to answer the question at the end. If you don't know the answer, just say that you don't know, don't try to make up an answer. {context} Question: {input}<|end_of_text|>
<|start_of_role|>assistant<|end_of_role|>""" qa_chain_prompt = PromptTemplate.from_template(prompt_template) combine_docs_chain = create_stuff_documents_chain(llm, qa_chain_prompt) rag_chain = create_retrieval_chain(vector_db.as_retriever(), combine_docs_chain)
```

Step 8: Prompt the RAG chain

Using these agentic chunks in the RAG workflow, let's start a user query. First, we can strategically prompt the model without any additional context from the vector store we built to test whether the model is using its built-in knowledge or truly by using the RAG context. Using the machine learning explainer from IBM, let's ask the question now.

```
output = llm.invoke("What is Model optimization process") output
```

Clearly, the model was not trained on information about the model optimization process and without outside tools or information, it cannot provide us with the correct information. The model hallucinates. Now, let's try providing the same query to the RAG chain with the agentic chunks we built.

```
rag_output = rag_chain.invoke({"input": "What is Model optimization process?"})
rag_output['answer']
```

Great! The Granite model correctly used the agentic RAG chunks as context to provide us with correct information about the model optimization process while preserving semantic coherence.