Softtek LLM SDK

A Python package for quick and customizable LLM-powered application development.

Contents

WE HIGHLY RECOMMEND USING THE PDF BOOKMARKS FOR EASE OF NAVIGATION.

Overview

This package contains a set of tools that allow you to quickly develop applications that use LLMs (Large Language Models), particularly those related to chatbots.

It has an assortment of classes that will help you implement functionality such as interaction with LLMs; response caching for improved performance and saving tokens; interfaces with vector databases; dynamic conversation history management and more.

Important Notes

- This package was built using Python version 3.10.11. We strongly recommend using the same version within your virtual environment.
- The current version of the package is still under development (it's in its Alpha stage), so a few features may seem somewhat limited. However, inheriting from the base classes and expanding on the current functionality is highly encouraged.
- Currently, only OpenAI models are supported, however, usage of other models may be possible, provided the custom classes inherit from the SDK's base classes.
- Currently, only Pinecone vector databases are supported, however, usage of other vector databases may be possible, provided custom classes inherit from the SDK's base classes.

Modules

Chatbot

Chatbot

The Chatbot class is the main class of the library. It is used to initialize a chatbot instance, which can then be used to chat with an LLM.

Attributes:

- model: The LLM model used by the chatbot.
- memory: The Memory used by the chatbot.
- description: Information about the chatbot.
- non_valid_response: Response given when the prompt does not follow the rules set by the filters. If None, an InvalidPrompt exception is raised when the prompt does not follow the rules set by the filters.
- filters: List of filters used by the chatbot.
- cache: Cache used by the chatbot.
- cache_probability: Probability of using the cache. If 1.0, the cache is always used. If 0.0, the cache is never used.

```
Chatbot(model: LLMModel, description: str | None =
None, memory: Memory =
```

```
WindowMemory(window_size=10),
non_valid_response: str | None = None, filters:
List[Filter] | None = None, cache: Cache | None =
None, cache_probability: float = 0.5, verbose: bool
= False)
```

Initializes the Chatbot class.

Arguments:

- model (LLMModel): LLM model used by the chatbot.
- description (str | None, optional): Information about the chatbot. This will be given to the LLM as the initial system prompt. For example "You are a helpful assistant". Defaults to None.
- memory (Memory, optional): Memory used by the chatbot. Defaults to WindowMemory(window_size=10).
- non_valid_response (str | None, optional): Response given when the prompt does not follow the rules set by the filters. Defaults to None. If None, an InvalidPrompt exception is raised when the prompt does not follow the rules set by the filters.
- filters (List[Filter] | None, optional): List of filters used by the chatbot. Defaults to None.
- cache (Cache | None, optional): Cache used by the chatbot. Defaults to None.

- cache_probability (float, optional): Probability of using the Cache. Defaults to 0.5. If 1.0, the cache is always used. If 0.0, the cache is never used.
- verbose (bool, optional): Whether to print additional information. Defaults to False.

```
def chat(prompt: str, print_cache_score: bool = False,
cache_kwargs: Dict = {}) -> Response
```

Chatbot function that returns a Response given a prompt. If a Cache is available, it considers previously stored conversations. Filters are applied to the prompt before processing to ensure it is valid.

Arguments:

- prompt (str): user's input string text
- print_cache_score (bool, optional): whether to print the cache score. Defaults to False.
- cache_kwargs (Dict, optional): additional keyword arguments to be passed to the Cache. Defaults to {}.

Returns:

- Response: the response message object generated by the chatbot, including its content and metadata.

Embeddings

Embeddings Model (Abstract)

Creates an abstract base class for an embeddings model. Used as a base class for implementing different types of embedding models.

EmbeddingsModel(**kwargs: Any)

Initializes the EmbeddingsModel class.

```
@abstractmethod
def embed(prompt: str, **kwargs: Any) -> List[float]
```

This is an abstract method for embedding a prompt into a list of floats. This method must be implemented by a subclass.

Arguments:

- prompt (str): The string prompt to embed.

Returns:

- List[float]: The embedding of the prompt as a list of floats.

Raises:

- NotImplementedError: When this abstract method is called without being implemented in a subclass.

OpenAI Embeddings

Creates an OpenAI embeddings model. This class is a subclass of the EmbeddingsModel abstract base class.

Properties:

- model_name: Embeddings model name.

```
OpenAIEmbeddings(api_key: str, model_name: str, api_type: Literal["azure"] | None = None, api_base: str | None = None, api_version: str = "2023-07-01-preview")
```

Initializes the OpenAIEmbeddings class.

Arguments:

- api_key (str): OpenAI API key.
- model_name (str): OpenAI embeddings model name.
- api_type ("azure" | None, optional): Type of API to use. Defaults to None.

- api_base (str | None, optional): Base URL for Azure API. Defaults to None.
- api_version (str, optional): API version for Azure API. Defaults to "2023-07-01-preview".

Raises:

- ValueError: When api_type is not "azure" or None.

```
@override
def embed(prompt: str, **kwargs) -> List[float]
```

Embeds a prompt into a list of floats.

Arguments:

- prompt (str): Prompt to embed.

Returns:

- List[float]: Embedding of the prompt as a list of floats.

Vector Stores

Vector Store (Abstract)

Creates an abstract base class for a vector store. Used as a base class for implementing different types of vector stores.

VectorStore()

Initializes the VectorStore class.

```
@abstractmethod
def add(vectors: List[Vector], **kwargs: Any)
```

Abstract method for adding the given vectors to the vector store.

Arguments:

- vectors (List[Vector]): A List of Vector instances to add.

Raises:

- NotImplementedError: The method must be implemented by a subclass.

```
@abstractmethod
def delete(ids: List[str], **kwargs: Any)
```

Abstract method for deleting vectors from the vector store given a list of vector IDs.

Arguments:

- ids (List[str]): A List of Vector IDs to delete.

Raises:

- NotImplementedError: The method must be implemented by a subclass.

```
@abstractmethod
def search(vector: Vector | None = None, **kwargs: Any) ->
List[Vector]
```

Abstract method for searching for vectors in the vector store.

Arguments:

- vector (Vector | None, optional): The query vector. Defaults to None.

Returns:

- List[Vector]: A list of Vector instances containing the search results.

Raises:

- NotImplementedError: The method must be implemented by a subclass.

Pinecone Vector Store

Implements a Vector Store using the Pinecone service. It inherits from the VectorStore class.

```
PineconeVectorStore(api_key: str, environment: str,
index_name: str)
```

Initialize a PineconeVectorStore object for managing vectors in a Pinecone index.

Arguments:

- api_key (str): The API key for authentication with the Pinecone service.
- environment (str): The Pinecone environment to use.
- index_name (str): The name of the index where vectors will be stored and retrieved.

Note:

- Make sure to use a valid API key and specify the desired environment and index name.

```
@override
def add(vectors: List[Vector], namespace: str | None = None,
batch_size: int | None = None, show_progress: bool = True,
**kwargs: Any)
```

Add vectors to the index.

Arguments:

- vectors (List[Vector]): A list of Vector objects to add to the index. Note that each vector must have a unique ID.
- namespace (str | None, optional): The namespace to write to. If not specified, the default namespace is used. Defaults to None.
- batch_size (int | None, optional): The number of vectors to upsert in each batch. If not specified, all vectors will be upserted in a single batch. Defaults to None.
- show_progress (bool, optional): Whether to show a progress bar using tqdm. Applied only if batch_size is provided. Defaults to True.

Raises:

- ValueError: If any of the vectors do not have a unique ID.

```
@override
def delete(ids: List[str] | None = None, delete_all: bool | None = None, namespace: str | None = None, filter: Dict | None = None, **kwargs: Any)
```

Delete vectors from the index.

Arguments:

- ids (List[str] | None, optional): A list of vector IDs to delete. Defaults to None.

- delete_all (bool | None, optional): This indicates that all vectors in the index namespace should be deleted. Defaults to None.
- namespace (str | None, optional): The namespace to delete vectors from. If not specified, the default namespace is used. Defaults to None.
- filter (Dict | None, optional): If specified, the metadata filter here will be used to select the vectors to delete. This is mutually exclusive with specifying ids to delete in the ids param or using delete_all=True. Defaults to None.

```
@override
def search(vector: Vector | None = None, id: str | None = None, top_k: int = 1, namespace: str | None = None, filter: Dict | None = None, **kwargs: Any)
```

Search for vectors in the index.

Arguments:

- vector (Vector | None, optional): The query vector. Each call can contain only one of the parameters id or vector. Defaults to None.
- id (str | None, optional): The unique ID of the vector to be used as a query vector. Each call can contain only one of the parameters id or vector. Defaults to None.
- top_k (int, optional): The number of results to return for each query. Defaults to 1.

- namespace (str | None, optional): The namespace to fetch vectors from. If not specified, the default namespace is used. Defaults to None.
- filter (Dict | None, optional): The filter to apply. You can use vector metadata to limit your search. Defaults to None.

Returns:

- List[Vector]: A list of Vector objects containing the search results.

Memory

Memory

Represents the memory of the assistant. Stores all the messages that have been exchanged between the user and the assistant.

Memory()

Initializes the Memory class.

```
@classmethod
def from_messages(messages: List[Message])
```

Initializes the Memory class from a list of messages.

Arguments:

- messages (List[Message]): The list of messages to initialize the memory with.

Returns:

- Memory: The initialized memory.

```
def add_message(role: Literal["system", "user", "assistant", "function"], content: str)
```

Adds a message to the memory.

Arguments:

- role ("system", "user", "assistant", "function"): The role of the message.
- content (str): The content of the message.

```
def delete_message(index: int)
```

Deletes a message from the memory.

Arguments:

- index (int): The index of the message to delete.

```
def get_message(index: int) -> Message
```

Returns a message from the memory.

Arguments:

- index (int): The index of the message to return.

Returns:

- Message: The message at the given index.

```
def get_messages() -> List[Message]
```

Returns all the messages from the memory. It is a copy of the original list of messages. Appending to this list will not affect the original list.

Returns:

- List[Message]: A copy of the list of messages.

```
def clear_messages()
```

Clears all messages from the memory.

Window Memory

Represents the memory of the assistant. Stores all the messages that have been exchanged between the user and the assistant. It has a maximum size. It inherits from the Memory class.

Attributes:

- window_size (int): The maximum number of messages to store in the memory.

WindowMemory(window_size: int)

Initializes the WindowMemory class.

Arguments:

- window_size (int): The maximum number of messages to store in the memory.

```
@override
def add_message(role: Literal["system", "user", "assistant", "function"], content: str)
```

Adds a message to the memory.

Arguments:

- role ("system", "user", "assistant", "function"): The role of the message.

- content (str): The content of the message.

Models

LLM Model (Abstract)

Creates an abstract base class for a Large Language Model. Used as a base class for implementing different types of language models. Defines a call method that must be implemented.

Parameters:

- name: Name of the model
- verbose: Whether to print debug messages

```
LLMModel(model_name: str, verbose: bool = False, **kwargs: Any)
```

Initializes the LLMModel class.

Arguments:

- model_name (str): Name of the model

- verbose (bool, optional): Whether to print debug messages. Defaults to False.

```
@abstractmethod
def __call__(memory: Memory, description: str = "You are a bot", **kwargs: Any) -> Response
```

A method to be overridden that calls the model to generate text.

Arguments:

- memory (Memory): An instance of a Memory class containing the conversation history.
- description (str, optional): Description of the model. Defaults to "You are a bot".

Returns:

- Response: The generated response.

Raises:

- NotImplementedError: When this abstract method is called without being implemented in a subclass.

```
@abstractmethod
def parse_filters(prompt: str, context: List[Message], filters: List[Filter]) -> List[Message]
```

Generates a prompt message to check if a given prompt follows a set of filtering rules.

Arguments:

- prompt (str): a string representing the prompt that will be checked against rules
- context (List[Message]): A list containing the last couple messages from the chat
- filters (List[Filter]): List of filters used by the chatbot

Raises:

- NotImplementedError: When this abstract method is called without being implemented in a subclass.

OpenAI

Creates an OpenAI language model. This class is a subclass of the LLMModel abstract base class.

Properties:

- model_name: Language model name.
- max_tokens: The maximum number of tokens to generate in the chat completion. The total length of input tokens and generated tokens is limited by the model's context length.

- temperature: What sampling temperature to use, between 0 and 2. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic.
- presence_penalty: Number between -2.0 and 2.0. Positive values penalize new tokens based on whether they appear in the text so far, increasing the model's likelihood to talk about new topics.
- frequency_penalty: Number between -2.0 and 2.0. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim.

```
OpenAI(api_key: str, model_name: str, api_type: Literal["azure"] | None = None, api_base: str | None = None, api_version: str = "2023-07-01-preview", max_tokens: int | None = None, temperature: float = 1, presence_penalty: float = 0, frequency_penalty: float = 0, verbose: bool = False)
```

Initializes the OpenAI LLM Model class.

Arguments:

- api_key (str): OpenAI API key.
- model_name (str): Name of the model.
- api_type ("azure" | None, optional): Type of API to use. Defaults to None.

- api_base (str | None, optional): Base URL for Azure API. Defaults to None.
- api_version (str, optional): API version for Azure API. Defaults to "2023-07-01-preview".
- max_tokens (int | None, optional): The maximum number of tokens to generate in the chat completion. The total length of input tokens and generated tokens is limited by the model's context length. Defaults to None.
- temperature (float, optional): What sampling temperature to use, between 0 and 2. Higher values like 0.8 will make the output more random, while lower values like 0.2 will make it more focused and deterministic. Defaults to 1.
- presence_penalty (float, optional): Number between -2.0 and 2.0. Positive values penalize new tokens based on whether they appear in the text so far, increasing the model's likelihood to talk about new topics. Defaults to 0.
- frequency_penalty (float, optional): Number between -2.0 and 2.0. Positive values penalize new tokens based on their existing frequency in the text so far, decreasing the model's likelihood to repeat the same line verbatim. Defaults to 0.
- verbose (bool, optional): Whether to print debug messages. Defaults to False.

Raises:

- ValueError: When api_type is not "azure" or None.

```
@override
def __call__(memory: Memory, description: str = "You are a bot.")
-> Response
```

Process a conversation using the OpenAI model and return a Response object. This function sends a conversation stored in the memory parameter to the specified OpenAI model, retrieves a response from the model, and records the conversation in memory. It then constructs a Response object containing the model's reply.

Arguments:

- memory (Memory): An instance of the Memory class containing the conversation history.
- description (str, optional): Description of the model. Defaults to "You are a bot.".

Returns:

- Response: A Response object containing the model's reply, timestamp, latency, and model name, amongst other data.

```
@override
def parse_filters(prompt: str, context: List[Message], filters:
List[Filter]) -> List[Message]
```

Generates a prompt message to check if a given prompt follows a set of filtering rules.

Arguments:

- prompt (str): a string representing the prompt that will be checked against rules.
- context (List[Message]): A list containing the last couple messages from the chat.
- filters (List[Filter]): List of filters used by the chatbot.

Returns:

- List[Message]: a list of messages to be used by the chatbot to check if the prompt respects the rules

---

Cache

Cache

Represents the cache of the assistant. Stores all the prompts and responses that have been exchanged between the user and the assistant.

Attributes:

- vector_store (VectorStore): The vector store used to store the prompts and responses.
- embeddings_model (EmbeddingsModel): The embeddings model used to generate embeddings for prompts.

```
Cache(vector_store: VectorStore,
embeddings_model: EmbeddingsModel)
```

Initializes the Cache class.

Arguments:

- vector_store (VectorStore): The vector store used to store the prompts and responses.
- embeddings_model (EmbeddingsModel): The embeddings model used to generate embeddings for prompts.

```
def add(prompt: str, response: Response, **kwargs)
```

This function adds a prompt and response to the cache. It calculates the embeddings for the prompt and adds it to the vector store.

Arguments:

- prompt (str): A string prompt which will be converted to embeddings for vector storage and query.
- response (Response): A Response object containing the model's reply, timestamp, latency, and model name, as well as other data.

```
def retrieve(prompt: str, threshold: float = 0.9, additional_kwargs:
Dict = {}, **kwargs) -> Tuple[Response | None, float]
```

This function retrieves the best response from a query using the prompt provided by the user. It calculates the time taken to retrieve the data and returns the response.

Arguments:

- prompt (str): A string prompt to which the function will respond to.
- threshold (float, optional): The threshold to use for the search. Defaults to 0.9. A query needs to have a similarity score above the threshold to be valid.
- additional_kwargs (Dict, optional): Optional dictionary of additional keyword arguments to add to the retrieved response. Defaults to {}.

Returns:

- Tuple[Response | None, float]: A tuple containing the response and the score of the best match.

Helper Classes

Message

Model class to represent a message.

Attributes:

- role ("system", "user", "assistant", "function"): the role of the message
- content (str): the content of the message

Usage

Defines Usage class with the following attributes:

Attributes:

- prompt_tokens: an integer representing the number of tokens in the prompt. Defaults to 0.
- completion_tokens: an integer representing the number of tokens in the completion. Defaults to 0.
- total_tokens: an integer representing the total number of tokens in the Usage. Defaults to 0.

Response

Represents the response from an LLM.

Attributes.

- message (Message): Message object that the API generated as a response.
- created (int): Unix timestamp for when the response was created.
- latency (int): Time in milliseconds taken to generate the response.

- from_cache (bool): Whether the response was served from cache or not.

Optional Attributes.

- model (str): String representation of the model used to generate the response. Defaults to "".
- usage (Usage): Usage object containing metrics of resource usage from generating the response. Defaults to Usage().
- additional_kwargs (Dict): Optional dictionary of additional keyword arguments. Defaults to {}.

Filter

A model class for a filter object.

Attributes:

- type ("ALLOW", "DENY"): The type of the filter instance: ALLOW or DENY
- case (str): The case for which the filter applies.

OpenAIChatChoice

A model class for a specific choice in an OpenAI chat response.

Attributes:

- index (int): The index of the choice in the list of choices.
- message (Message): A chat completion message generated by the model.
- finish_reason (str): The reason the model stopped generating tokens. This will be “stop” if the model hit a natural stop point or a provided stop sequence, “length” if the maximum number of tokens specified in the request was reached, or “function_call” if the model called a function.

OpenAIChatResponse

A model class for an OpenAI chat response.

Attributes:

- id (str): A unique identifier for the chat completion.
- object ("chat.completion"): The object type, which is always “chat.completion”.
- created (int): The Unix timestamp (in seconds) of when the chat completion was created.
- model (str): The model used for the chat completion.
- choices (List[OpenAIChatChoice]): A list of chat completion choices.
- usage (Usage): Usage statistics for the completion request.

Vector

A model class for a vector object.

Attributes:

- embeddings (List[float]): A list of floating point numbers representing the vector.

Optional Attributes:

- id (str): A unique identifier for the vector. Defaults to "".
- metadata (Dict): Optional dictionary of metadata for the vector. Defaults to {}.

Example usage

```
from chatbot import Chatbot
from models import OpenAI
from schemas import Filter
from cache import Cache
from vectorStores import PineconeVectorStore
from embeddings import OpenAIEmbeddings
```

```
model = OpenAI(
    api_key=OPENAI_API_KEY,
    model_name=OPENAI_CHAT_MODEL_NAME,
    api_type="azure",
    api_base=OPENAI_API_BASE,
)

filters = [
    Filter(
        type="DENY",
        case="ANYTHING related to the Titanic, no matter the
question. Seriously, NO TITANIC, it's a sensitive topic.",
    ),
]

vector_store = PineconeVectorStore(
    api_key=PINECONE_API_KEY,
    environment=PINECONE_ENVIRONMENT,
    index_name=PINECONE_INDEX_NAME,
)

embeddings_model = OpenAIEmbeddings(
    api_key=OPENAI_API_KEY,
    model_name=OPENAI_EMBEDDINGS_MODEL_NAME,
    api_type="azure",
    api_base=OPENAI_API_BASE,
)

cache = Cache(
    vector_store=vector_store,
    embeddings_model=embeddings_model,
)
```

```
chatbot = Chatbot(
    model=model,
    filters=filters,
    cache=cache,
    description="You are a polite and very helpful assistant",
)
response = chatbot.chat(
    "Hi, my name is Jeff",
    cache_kwargs={"namespace": "chatbot-cache-test"},
    print_cache_score=True,
)
```

>>> Response(message=Message(role='assistant', content='Hello Jeff! How can I assist you today?'), created=1695255737, latency=1445, from_cache=False, model='gpt-35-turbo-16k', usage=Usage(prompt_tokens=15, completion_tokens=7, total_tokens=22), additional_kwargs={})

```
response = chatbot.chat(
    "What is my name?", cache_kwargs={"namespace": "chatbot-
cache-test"}
)
```

>>> Response(message=Message(role='assistant', content='Your name is Jeff! You mentioned it earlier. How can I help you, Jeff?'), created=1695255738, latency=814, from_cache=True, model='gpt-35-turbo-16k', usage=Usage(prompt_tokens=0, completion_tokens=0, total_tokens=0), additional_kwargs={})

```
response = chatbot.chat(
    "When did the Titanic sink?", cache_kwargs={"namespace":
"chatbot-cache-test"}
)
```

>>> raised InvalidPrompt: The prompt does not follow the rules set by the filters. If this behavior is not intended, consider modifying the filters. It is recommended to use LLMs for meta prompts.