Chat models

INFO

Head to Integrations for documentation on built-in integrations with chat model providers.

Chat models are a variation on language models. While chat models use language models under the hood, the interface they expose is a bit different. Rather than expose a "text in, text out" API, they expose an interface where "chat messages" are the inputs and outputs.

Chat model APIs are fairly new, so we are still figuring out the correct abstractions.

Get started

Setup

To start we'll need to install the OpenAI Python package:

```
pip install openai
```

Accessing the API requires an API key, which you can get by creating an account and heading here. Once we have a key we'll want to set it as an environment variable by running:

```
export OPENAI_API_KEY="..."
```

If you'd prefer not to set an environment variable you can pass the key in directly via the `openai_api_key` named parameter when initiating the OpenAI LLM class:

```
from langchain.chat_models import ChatOpenAI

chat = ChatOpenAI(openai_api_key="...")
```

otherwise you can initialize without any params:

```
from langchain.chat_models import ChatOpenAI

chat = ChatOpenAI()
```

Messages

The chat model interface is based around messages rather than raw text. The types of messages currently supported in LangChain are `AIMessage`, `HumanMessage`, `SystemMessage`, and `ChatMessage` -- `ChatMessage` takes in an arbitrary role parameter. Most of the time, you'll just be dealing with `HumanMessage`, `AIMessage`, and `SystemMessage`

`__call__`

Messages in -> message out

You can get chat completions by passing one or more messages to the chat model. The response will be a message.

```
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)

chat([HumanMessage(content="Translate this sentence from English to
French: I love programming.")])
```

```
    AIMessage(content="J'aime programmer.", additional_kwargs={})
```

OpenAI's chat model supports multiple messages as input. See here for more information. Here is an example of sending a system and user message to the chat model:

```
messages = [
    SystemMessage(content="You are a helpful assistant that translates
English to French."),
    HumanMessage(content="I love programming.")
]
chat(messages)
```

```
    AIMessage(content="J'aime programmer.", additional_kwargs={})
```

`generate`

Batch calls, richer outputs

You can go one step further and generate completions for multiple sets of messages using `generate`. This returns an `LLMResult` with an additional `message` parameter.

```
batch_messages = [
    [
        SystemMessage(content="You are a helpful assistant that translates English to French."),
        HumanMessage(content="I love programming.")
    ],
    [
        SystemMessage(content="You are a helpful assistant that translates English to French."),
        HumanMessage(content="I love artificial intelligence.")
    ],
]
result = chat.generate(batch_messages)
result
```

```
    LLMResult(generations=[[ChatGeneration(text="J'aime programmer.", generation_info=None, message=AIMessage(content="J'aime programmer.", additional_kwargs={}))], [ChatGeneration(text="J'aime l'intelligence artificielle.", generation_info=None, message=AIMessage(content="J'aime l'intelligence artificielle.", additional_kwargs={}))]], llm_output={'token_usage': {'prompt_tokens': 57, 'completion_tokens': 20, 'total_tokens': 77}})
```

You can recover things like token usage from this LLMResult

```
result.llm_output
```

```
{'token_usage': {'prompt_tokens': 57,
  'completion_tokens': 20,
  'total_tokens': 77}}
```