Vector store-augmented text generation

This notebook walks through how to use LangChain for text generation over a vector index. This is useful if we want to generate text that is able to draw from a large body of custom text, for example, generating blog posts that have an understanding of previous blog posts written, or product tutorials that can refer to product documentation.

Prepare Data

First, we prepare the data. For this example, we fetch a documentation site that consists of markdown files hosted on Github and split them into small enough Documents.

```
from langchain.llms import OpenAI
from langchain.docstore.document import Document
import requests
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.vectorstores import Chroma
from langchain.text_splitter import CharacterTextSplitter
from langchain.prompts import PromptTemplate
import pathlib
import subprocess
import tempfile
```

```
def get_github_docs(repo_owner, repo_name):
    with tempfile.TemporaryDirectory() as d:
```

```
        subprocess.check_call(
            f"git clone --depth 1 https://github.com/{repo_owner}/{repo_name}.git .",
            cwd=d,
            shell=True,
        )
        git_sha = (
            subprocess.check_output("git rev-parse HEAD", shell=True, cwd=d)
            .decode("utf-8")
            .strip()
        )
        repo_path = pathlib.Path(d)
        markdown_files = list(repo_path.glob("*/*.md")) + list(
            repo_path.glob("*/*.mdx")
        )
        for markdown_file in markdown_files:
            with open(markdown_file, "r") as f:
                relative_path = markdown_file.relative_to(repo_path)
                github_url = f"https://github.com/{repo_owner}/{repo_name}/blob/{git_sha}/{relative_path}"
                yield Document(page_content=f.read(), metadata={"source": github_url})


sources = get_github_docs("yirenlu92", "deno-manual-forked")

source_chunks = []
splitter = CharacterTextSplitter(separator=" ", chunk_size=1024, chunk_overlap=0)
for source in sources:
    for chunk in splitter.split_text(source.page_content):
        source_chunks.append(Document(page_content=chunk, metadata=source.metadata))
```

```
    Cloning into '.'...
```

Set Up Vector DB

Now that we have the documentation content in chunks, let's put all this information in a vector index for easy retrieval.

```
search_index = Chroma.from_documents(source_chunks, OpenAIEmbeddings())
```

Set Up LLM Chain with Custom Prompt

Next, let's set up a simple LLM chain but give it a custom prompt for blog post generation. Note that the custom prompt is parameterized and takes two inputs: `context`, which will be the documents fetched from the vector search, and `topic`, which is given by the user.

```
from langchain.chains import LLMChain

prompt_template = """Use the context below to write a 400 word blog post about the topic below:
    Context: {context}
    Topic: {topic}
    Blog post:"""

PROMPT = PromptTemplate(template=prompt_template, input_variables=["context", "topic"])

llm = OpenAI(temperature=0)

chain = LLMChain(llm=llm, prompt=PROMPT)
```

Generate Text

Finally, we write a function to apply our inputs to the chain. The function takes an input parameter `topic`. We find the documents in the vector index that correspond to that `topic`, and use them as additional context in our simple LLM chain.

```
def generate_blog_post(topic):
    docs = search_index.similarity_search(topic, k=4)
    inputs = [{"context": doc.page_content, "topic": topic} for doc in docs]
    print(chain.apply(inputs))
```

```
generate_blog_post("environment variables")
```

```
    [{'text': '\n\nEnvironment variables are a great way to store and access sensitive information in your Deno applications. Deno offers built-in support for environment variables with `Deno.env`, and you can also use a `.env` file to store and access environment variables.\n\nUsing `Deno.env` is simple. It has getter and setter methods, so you can easily set and retrieve environment variables. For example, you can set the `FIREBASE_API_KEY` and `FIREBASE_AUTH_DOMAIN` environment variables like this:\n\n```ts\nDeno.env.set("FIREBASE_API_KEY", "examplekey123");\nDeno.env.set("FIREBASE_AUTH_DOMAIN", "firebasedomain.com");\n\nconsole.log(Deno.env.get("FIREBASE_API_KEY")); // examplekey123\nconsole.log(Deno.env.get("FIREBASE_AUTH_DOMAIN")); // firebasedomain.com\n```\n\nYou can also store environment variables in a `.env` file. This is a great'}, {'text': '\n\nEnvironment variables are a powerful tool for managing configuration settings in a program. They allow us to set values that can be used by the program, without having to hard-code them into the code. This makes it easier to change settings without having to modify the code.\n\nIn Deno, environment variables can be set in a few different ways. The most common way is to use the `VAR=value` syntax. This will set the environment variable `VAR` to the value `value`. This can be used to set any number of environment variables before running a command. For example, if we wanted to set the environment variable `VAR` to `hello` before running a Deno command, we could do so like this:\n\n```\nVAR=hello deno run main.ts\n```\n\nThis will set the environment variable `VAR` to `hello`
```

before running the command. We can then access this variable in our code using the `Deno.env.get()` function. For example, if we ran the following command:\n\n```\nVAR=hello && deno eval "console.log(\'Deno: \' + Deno.env.get(\'VAR'}, {'text': '\n\nEnvironment variables are a powerful tool for developers, allowing them to store and access data without having to hard-code it into their applications. In Deno, you can access environment variables using the `Deno.env.get()` function.\n\nFor example, if you wanted to access the `HOME` environment variable, you could do so like this:\n\n```js\n// env.js\nDeno.env.get("HOME");\n```\n\nWhen running this code, you\'ll need to grant the Deno process access to environment variables. This can be done by passing the `--allow-env` flag to the `deno run` command. You can also specify which environment variables you want to grant access to, like this:\n\n```shell\n# Allow access to only the HOME env var\ndeno run --allow-env=HOME env.js\n```\n\nIt\'s important to note that environment variables are case insensitive on Windows, so Deno also matches them case insensitively (on Windows only).\n\nAnother thing to be aware of when using environment variables is subprocess permissions. Subprocesses are powerful and can access system resources regardless of the permissions you granted to the Den'}, {'text': '\n\nEnvironment variables are an important part of any programming language, and Deno is no exception. Deno is a secure JavaScript and TypeScript runtime built on the V8 JavaScript engine, and it recently added support for environment variables. This feature was added in Deno version 1.6.0, and it is now available for use in Deno applications.\n\nEnvironment variables are used to store information that can be used by programs. They are typically used to store configuration information, such as the location of a database or the name of a user. In Deno, environment variables are stored in the `Deno.env` object. This object is similar to the `process.env` object in Node.js, and it allows you to access and set environment variables.\n\nThe `Deno.env` object is a read-only object, meaning that you cannot directly modify the environment variables. Instead, you must use the `Deno.env.set()` function to set environment variables. This function takes two arguments: the name of the environment variable and the value to set it to. For example, if you wanted to set the `FOO` environment variable to `bar`, you would use the following code:\n\n```'}]