Lucas A. Meyer

A Quick Tour of the Semantic Kernel

AUTHOR
Lucas A. Meyer

PUBLISHED
October 15, 2023

Introduction

The Microsoft Semantic Kernel is a thin, open-source, software development toolkit that makes it easier for applications to interact with AI services.

It was originally designed to power the Microsoft Copilots, such as Microsoft 365 and Bing. The initial version was in C#, but it has now been extended to Python and Java, and released to the developer community as an open-source package.

Python code for this post

The Python code for this post is available in a notebook on GitHub.

In order to run the code of this post, you will need an Azure subscription with access to the OpenAI API, or an OpenAI subscription.

Although you can get a free trial for Azure that gives you many services, the OpenAI services in Azure are not free. You can see the prices here.

Differences from LangChain

- Semantic Kernel was designed to be more customizable than LangChain. It gives you more control but requires more coding.
- Semantic Kernel was designed to help easily add LLM features to enterprise or large-scale consumer applications.
- If you use Python, you can use both LangChain and Semantic Kernel.
- If you use JavaScript, you can use LangChain, but not Semantic Kernel. There's a community-supported TypeScript API for Semantic Kernel, but it's not officially supported by Microsoft.
- If you use Java or .NET, you can use Semantic Kernel, but not LangChain.

A whirlwind tour through Semantic Kernel in Python

In the post below, I'll quickly show how to get started with Semantic Kernel in Python using an Azure subscription.

The Kernel

The Semantic Kernel is just a lightweight object where you will attach everything you need to complete your AI tasks.

```
import semantic_kernel as sk
kernel = sk.Kernel()
```

Connectors

Connectors are the way you connect to AI services. You can connect multiple services to the same kernel, which allows you to perform a complex task using different services for each step.

For example, my subscription has two models deployed: one deployment named `gpt35` the GPT 3.5 Turbo model and one deployment named `gpt4` for a deployment of the GPT-4 model. I can load both of them into the kernel with the code below:

```
from semantic_kernel.connectors.ai.open_ai import AzureChatCompletion

gpt35 = AzureChatCompletion(deployment_name="gpt35", # yours may be different
endpoint=OPENAI_ENDPOINT,
api_key=OPENAI_API_KEY)

gpt4 = AzureChatCompletion(
    deployment_name="gpt4", # yours may be different
endpoint=OPENAI_ENDPOINT,
api_key=OPENAI_API_KEY)

kernel.add_chat_service("gpt35", gpt35)
kernel.add_chat_service("gpt4", gpt4)
```

Semantic functions

Semantic functions are functions that use large language models to perform a task. The simple example below shows how to create a semantic function that tells a knock-knock joke.

```
prompt = """knock, knock? Who’s there? {{$input}}. {{$input}} who?"""

knock = kernel.create_semantic_function(prompt, temperature=0.8)

response = knock(“Dishes")
print(response)
```

The response will be something like `Dishes the police, open up!`.

Native functions

Native functions are regular Python functions. They can be used to perform any task that doesn’t require a large language model. For example, the code below classifies an image given a URL.

The function uses the `timm` library to download a pre-trained model and classify the image. The `@sk_function` decorator tags the following function `classify_image` as a function that can be imported by the Semantic Kernel.

```
import requests
from PIL import Image
import timm
```

```
from timm.data.imagenet_info import ImageNetInfo
from semantic_kernel.skill_definition import sk_function
from semantic_kernel.orchestration.sk_context import SKContext

class ImageClassifierPlugin:
    def __init__(self):
        self.model = timm.create_model("convnext_tiny.in12k_ft_in1k", pretraine
        self.model.eval()
        data_config = timm.data.resolve_model_data_config(self.model)
        self.transforms = timm.data.create_transform(**data_config, is_training
        self.imagenet_info = ImageNetInfo()

    @sk_function(
        description="Takes a url as an input and classifies the image",
        name="classify_image",
        input_description="The url of the image to classify",
    )
    def classify_image(self, url: str) -> str:
        image = self.download_image(url)
        pred = self.model(self.transforms(image)[None])
        cls = self.imagenet_info.index_to_description(pred.argmax())

        return cls.split(",")[0]

    def download_image(self, url):
        return Image.open(requests.get(url, stream=True).raw).convert("RGB")
```

Loading a native function into the kernel

You can load native functions into the kernel with the `import_skill` method of the Kernel object.

```
classify_plugin = kernel.import_skill(image_classifier, skill_name="classify_im
```

The `classify_plugin` is a collection of functions. To call the `classify_image` function, you can use the code below:

```
answer = classify_plugin.classify_image("https://links.meyerperin.com/tiger_jpg
print(answer)
```

For example, if I use the URL below as the parameter:

`https://links.meyerperin.com/tiger_jpg`

The answer will be `tiger`.

Plug-ins

One of the greatest strengths of the Microsoft Semantic Kernel is the ability of creating plugins. Plugins are collections of functions that can be imported into the kernel.

A semantic plugin is a collection of semantic functions. Each function should be in its own directory. Each directory should have two files: `config.json` and `skprompt.txt`.

The `config.json` file contains the configuration of the semantic function: which is the preferred engine to use, the temperature, etc.

The `skprompt.txt` file contains the prompt of the semantic function. The prompt is the text that will be sent to the engine to generate the response.

Example directory structure

```
└──plugins
    └──jokes
        ├──cross_the_road_joke
        |   ├──config.json
        |   └──skprompt.txt
        └──knock_knock_joke
            ├──config.json
            └──skprompt.txt
```

An example config.json file

The configuration file below shows a possible configuration for a semantic function that generates knock-knock jokes. The `default_services` property is an array of the preferred engines to use. The `completion` property contains the parameters that will be sent to the engine. The `input` property contains the parameters that will be sent to the semantic function.

I frequently use the `default_services` property to send simple tasks to cheaper services like GPT-3.5 and more complex tasks to more expensive services like GPT-4.

The `description` field is important, because it can be used by the `Planner` function of the kernel, if you want to let the kernel itself select which functions to call.

```
{
    "schema": 1,
    "type": "completion",
    "description": "Generates a knock-knock joke based on user input",
    "default_services": ["gpt35"],
    "completion": {
        "temperature": 0.8,
        "number_of_responses": 1,
        "top_p": 1,
        "max_tokens": 4000,
        "presence_penalty": 0.0,
        "frequency_penalty": 0.0
       },
       "input": {
     "parameters": [{
      "name": "input",
      "description": "The topic that the joke should be written about",
      "defaultValue": "Dishes"
     }]
```

```
    }
}
```

An example skprompt.txt file

The prompt below is the prompt for the semantic function that generates knock-knock jokes. The `{{$input}}` placeholder will be replaced by the value of the `input` parameter.

```
knock, knock?
Who's there?
{{$input}}.
{{$input}} who?


Repeat the whole setup and finish the joke.
```

Loading the plugin into the kernel

You can load all the functions that are inside a plugin directory with the `import_semantic_skill_from_directory` method of the kernel object.

```
jokes_plugin = kernel.import_semantic_skill_from_directory("plugins", "jokes")
```

The resulting object is a dictionary of functions. You can load the functions into variables or call them directly.

```
knock = jokes_plugin["knock_knock_joke"]
print (knock("Dishes"))
# Dishes the police, open up!

print(jokes_plugin["cross_the_road_joke"]("Tiger"))
# Why did the tiger cross the road? To show the chicken it can be done.
```

Calling multiple functions in sequence

As you saw before, we loaded the native function classify_image into the kernel. And in the block above, we loaded the jokes plugin into the kernel. We can call both functions in sequence with the code below:

```
context = kernel.create_new_context()
context["input"] = url

response = await kernel.run_async(
    classify_plugin["classify_image"],
    jokes_plugin["cross_the_road_joke"],
    input_context=context
)

print(response)
```

This code will call the first (native) function that classifies an image, and then call the second (semantic) function that generates a joke about the image. Since we loaded the image of a tiger, the response will be something like:

```
Why did the tiger cross the road? To show the chicken it could be done.
```

Conclusion

I hope this brief tour helps you get started. If you want to learn more, you can check the official documentation.

You can also join the community Discord server.