Connecting to a Feature Store

Feature stores are a concept from traditional machine learning that make sure data fed into models is up-to-date and relevant. For more on this, see here.

This concept is extremely relevant when considering putting LLM applications in production. In order to personalize LLM applications, you may want to combine LLMs with up-to-date information about particular users. Feature stores can be a great way to keep that data fresh, and LangChain provides an easy way to combine that data with LLMs.

In this notebook we will show how to connect prompt templates to feature stores. The basic idea is to call a feature store from inside a prompt template to retrieve values that are then formatted into the prompt.

Feast

To start, we will use the popular open source feature store framework Feast.

This assumes you have already run the steps in the README around getting started. We will build off of that example in getting started, and create and LLMChain to write a note to a specific driver regarding their up-to-date statistics.

Load Feast Store

Again, this should be set up according to the instructions in the Feast README

```
from feast import FeatureStore

# You may need to update the path depending on where you stored it
feast_repo_path = "../../../../../my_feature_repo/feature_repo/"
store = FeatureStore(repo_path=feast_repo_path)
```

Prompts

Here we will set up a custom FeastPromptTemplate. This prompt template will take in a driver id, look up their stats, and format those stats into a prompt.

Note that the input to this prompt template is just `driver_id`, since that is the only user defined piece (all other variables are looked up inside the prompt template).

```
from langchain.prompts import PromptTemplate, StringPromptTemplate
```

API Reference:

- PromptTemplate from `langchain.prompts`
- StringPromptTemplate from `langchain.prompts`

```
template = """Given the driver's up to date stats, write them note
relaying those stats to them.
If they have a conversation rate above .5, give them a compliment.
Otherwise, make a silly joke about chickens at the end to make them feel
better

Here are the drivers stats:
Conversation rate: {conv_rate}
Acceptance rate: {acc_rate}
Average Daily Trips: {avg_daily_trips}

Your response:"""
prompt = PromptTemplate.from_template(template)
```

```
class FeastPromptTemplate(StringPromptTemplate):
    def format(self, **kwargs) -> str:
        driver_id = kwargs.pop("driver_id")
        feature_vector = store.get_online_features(
            features=[
                "driver_hourly_stats:conv_rate",
                "driver_hourly_stats:acc_rate",
                "driver_hourly_stats:avg_daily_trips",
            ],
            entity_rows=[{"driver_id": driver_id}],
        ).to_dict()
        kwargs["conv_rate"] = feature_vector["conv_rate"][0]
        kwargs["acc_rate"] = feature_vector["acc_rate"][0]
```

```
        kwargs["avg_daily_trips"] = feature_vector["avg_daily_trips"][0]
        return prompt.format(**kwargs)
```

```
prompt_template = FeastPromptTemplate(input_variables=["driver_id"])
```

```
print(prompt_template.format(driver_id=1001))
```

```
    Given the driver's up to date stats, write them note relaying those
stats to them.
    If they have a conversation rate above .5, give them a compliment.
Otherwise, make a silly joke about chickens at the end to make them feel
better

    Here are the drivers stats:
    Conversation rate: 0.4745151400566101
    Acceptance rate: 0.055561766028404236
    Average Daily Trips: 936

    Your response:
```

Use in a chain

We can now use this in a chain, successfully creating a chain that achieves personalization backed by a feature store

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- LLMChain from `langchain.chains`

```
chain = LLMChain(llm=ChatOpenAI(), prompt=prompt_template)
```

```
chain.run(1001)
```

```
    "Hi there! I wanted to update you on your current stats. Your
acceptance rate is 0.055561766028404236 and your average daily trips are
936. While your conversation rate is currently 0.4745151400566101, I have
no doubt that with a little extra effort, you'll be able to exceed that .5
mark! Keep up the great work! And remember, even chickens can't always
cross the road, but they still give it their best shot."
```

Tecton

Above, we showed how you could use Feast, a popular open source and self-managed feature store, with LangChain. Our examples below will show a similar integration using Tecton. Tecton is a fully managed feature platform built to orchestrate the complete ML feature lifecycle, from transformation to online serving, with enterprise-grade SLAs.

Prerequisites

- Tecton Deployment (sign up at https://tecton.ai)
- `TECTON_API_KEY` environment variable set to a valid Service Account key

Define and Load Features

We will use the user_transaction_counts Feature View from the Tecton tutorial as part of a Feature Service. For simplicity, we are only using a single Feature View; however, more sophisticated applications may require more feature views to retrieve the features needed for its prompt.

```
user_transaction_metrics = FeatureService(
    name = "user_transaction_metrics",
    features = [user_transaction_counts]
)
```

The above Feature Service is expected to be applied to a live workspace. For this example, we will be using the "prod" workspace.

```
import tecton

workspace = tecton.get_workspace("prod")
feature_service =
workspace.get_feature_service("user_transaction_metrics")
```

Prompts

Here we will set up a custom TectonPromptTemplate. This prompt template will take in a user_id , look up their stats, and format those stats into a prompt.

Note that the input to this prompt template is just `user_id`, since that is the only user defined piece (all other variables are looked up inside the prompt template).

```
from langchain.prompts import PromptTemplate, StringPromptTemplate
```

API Reference:

- PromptTemplate from `langchain.prompts`
- StringPromptTemplate from `langchain.prompts`

```
template = """Given the vendor's up to date transaction stats, write them
a note based on the following rules:

1. If they had a transaction in the last day, write a short
congratulations message on their recent sales
2. If no transaction in the last day, but they had a transaction in the
last 30 days, playfully encourage them to sell more.
3. Always add a silly joke about chickens at the end

Here are the vendor's stats:
Number of Transactions Last Day: {transaction_count_1d}
Number of Transactions Last 30 Days: {transaction_count_30d}

Your response:"""
prompt = PromptTemplate.from_template(template)
```

```
class TectonPromptTemplate(StringPromptTemplate):
    def format(self, **kwargs) -> str:
        user_id = kwargs.pop("user_id")
        feature_vector = feature_service.get_online_features(
            join_keys={"user_id": user_id}
        ).to_dict()
        kwargs["transaction_count_1d"] = feature_vector[
            "user_transaction_counts.transaction_count_1d_1d"
        ]
        kwargs["transaction_count_30d"] = feature_vector[
            "user_transaction_counts.transaction_count_30d_1d"
        ]
        return prompt.format(**kwargs)
```

```
prompt_template = TectonPromptTemplate(input_variables=["user_id"])
```

```
print(prompt_template.format(user_id="user_469998441571"))
```

```
    Given the vendor's up to date transaction stats, write them a note
based on the following rules:

    1. If they had a transaction in the last day, write a short
congratulations message on their recent sales
    2. If no transaction in the last day, but they had a transaction in
the last 30 days, playfully encourage them to sell more.
    3. Always add a silly joke about chickens at the end

    Here are the vendor's stats:
    Number of Transactions Last Day: 657
    Number of Transactions Last 30 Days: 20326

    Your response:
```

Use in a chain

We can now use this in a chain, successfully creating a chain that achieves personalization backed by the Tecton Feature Platform

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- LLMChain from `langchain.chains`

```
chain = LLMChain(llm=ChatOpenAI(), prompt=prompt_template)
```

```
chain.run("user_469998441571")
```

```
    'Wow, congratulations on your recent sales! Your business is really
soaring like a chicken on a hot air balloon! Keep up the great work!'
```

Featureform

Finally, we will use Featureform an open-source and enterprise-grade feature store to run the same example. Featureform allows you to work with your infrastructure like Spark or locally to define your feature transformations.

Initialize Featureform

You can follow in the instructions in the README to initialize your transformations and features in Featureform.

```
import featureform as ff

client = ff.Client(host="demo.featureform.com")
```

Prompts

Here we will set up a custom FeatureformPromptTemplate. This prompt template will take in the average amount a user pays per transactions.

Note that the input to this prompt template is just avg_transaction, since that is the only user defined piece (all other variables are looked up inside the prompt template).

```
from langchain.prompts import PromptTemplate, StringPromptTemplate
```

API Reference:

- PromptTemplate from `langchain.prompts`
- StringPromptTemplate from `langchain.prompts`

```
template = """Given the amount a user spends on average per transaction,
let them know if they are a high roller. Otherwise, make a silly joke
about chickens at the end to make them feel better

Here are the user's stats:
Average Amount per Transaction: ${avg_transcation}

Your response:"""
prompt = PromptTemplate.from_template(template)
```

```
class FeatureformPromptTemplate(StringPromptTemplate):
    def format(self, **kwargs) -> str:
        user_id = kwargs.pop("user_id")
        fpf = client.features([("avg_transactions", "quickstart")],
{"user": user_id})
        return prompt.format(**kwargs)
```

```
prompt_template = FeatureformPrompTemplate(input_variables=["user_id"])
```

```
print(prompt_template.format(user_id="C1410926"))
```

Use in a chain

We can now use this in a chain, successfully creating a chain that achieves personalization backed by the Featureform Feature Platform

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import LLMChain
```

API Reference:

- ChatOpenAI from `langchain.chat_models`
- LLMChain from `langchain.chains`

```
chain = LLMChain(llm=ChatOpenAI(), prompt=prompt_template)
```

```
chain.run("C1410926")
```

AzureML Managed Feature Store

We will use AzureML Managed Feature Store to run the below example.

Prerequisites

- Create feature store with online materialization using instructions here Enable online materialization and run online inference.
- A successfully created feature store by following the instructions should have an `account` featureset with version as `1`. It will have `accountID` as index column with features `accountAge`, `accountCountry`, `numPaymentRejects1dPerUser`.

Prompts

- Here we will set up a custom AzureMLFeatureStorePromptTemplate. This prompt template will take in an `account_id` and optional `query`. It then fetches feature values from feature store and format those features into the output prompt. Note that the required input to this prompt template is just `account_id`, since that is the only user defined piece (all other variables are looked up inside the prompt template).
- Also note that this is a bootstrap example to showcase how LLM applications can leverage AzureML managed feature store. Developers are welcome to improve the prompt template

further to suit their needs.

```
import os
os.environ['AZURE_ML_CLI_PRIVATE_FEATURES_ENABLED'] = 'True'
```

```
import pandas

from pydantic import Extra
from langchain.prompts import PromptTemplate, StringPromptTemplate
from azure.identity import AzureCliCredential
from azureml.featurestore import FeatureStoreClient, init_online_lookup,
get_online_features

class AzureMLFeatureStorePromptTemplate(StringPromptTemplate,
extra=Extra.allow):

    def __init__(self, subscription_id: str, resource_group: str,
feature_store_name: str, **kwargs):
        # this is an example template for proof of concept and can be
changed to suit the developer needs
        template = """
            {query}
            ###
            account id = {account_id}
            account age = {account_age}
            account country = {account_country}
            payment rejects 1d per user = {payment_rejects_1d_per_user}
            ###
            """
        prompt_template=PromptTemplate.from_template(template)
        super().__init__(prompt=prompt_template, input_variables=
["account_id", "query"])

        # use AzureMLOnBehalfOfCredential() in spark context
        credential = AzureCliCredential()

        self._fs_client = FeatureStoreClient(
            credential=credential,
            subscription_id=subscription_id,
            resource_group_name=resource_group,
            name=feature_store_name)
```

```
        self._feature_set =
self._fs_client.feature_sets.get(name="accounts", version=1)

        init_online_lookup(self._feature_set.features, credential,
force=True)


    def format(self, **kwargs) -> str:
        if "account_id" not in kwargs:
            raise "account_id needed to fetch details from feature store"
        account_id = kwargs.pop("account_id")

        query=""
        if "query" in kwargs:
            query = kwargs.pop("query")

        # feature set is registered with accountID as entity index column.
        obs = pandas.DataFrame({'accountID': [account_id]})

        # get the feature details for the input entity from feature store.
        df = get_online_features(self._feature_set.features, obs)

        # populate prompt template output using the fetched feature
values.
        kwargs["query"] = query
        kwargs["account_id"] = account_id
        kwargs["account_age"] = df["accountAge"][0]
        kwargs["account_country"] = df["accountCountry"][0]
        kwargs["payment_rejects_1d_per_user"] =
df["numPaymentRejects1dPerUser"][0]

        return self.prompt.format(**kwargs)
```

API Reference:

- PromptTemplate from `langchain.prompts`
- StringPromptTemplate from `langchain.prompts`

Test

```
# Replace the place holders below with actual details of feature store
that was created in previous steps
```

```
prompt_template = AzureMLFeatureStorePromptTemplate(
            subscription_id="",
            resource_group="",
            feature_store_name="")
```

```
print(prompt_template.format(account_id="A1829581630230790"))
```

```
###
account id = A1829581630230790
account age = 563.0
account country = GB
payment rejects 1d per user = 15.0
###
```

Use in a chain

We can now use this in a chain, successfully creating a chain that achieves personalization backed by the AzureML Managed Feature Store

```
os.environ["OPENAI_API_KEY"]="" # Fill the open ai key here

from langchain.chat_models import ChatOpenAI
from langchain import LLMChain

chain = LLMChain(llm=ChatOpenAI(), prompt=prompt_template)
```

API Reference:

- ChatOpenAI from `langchain.chat_models`

```
# NOTE: developer's can further fine tune
AzureMLFeatureStorePromptTemplate
# for getting even more accurate results for the input query
```

```
chain.predict(account_id="A1829581630230790", query ="write a small thank you note within 20 words if account age > 10 using the account stats")
```

```
    'Thank you for being a valued member for over 10 years! We appreciate your continued support.'
```