Tagging

The tagging chain uses the OpenAI `functions` parameter to specify a schema to tag a document with. This helps us make sure that the model outputs exactly tags that we want, with their appropriate types.

The tagging chain is to be used when we want to tag a passage with a specific attribute (i.e. what is the sentiment of this message?)

```
from langchain.chat_models import ChatOpenAI
from langchain.chains import create_tagging_chain, create_tagging_chain_pydantic
from langchain.prompts import ChatPromptTemplate
```

```
    /Users/harrisonchase/.pyenv/versions/3.9.1/envs/langchain/lib/python3.9/site-
packages/deeplake/util/check_latest_version.py:32: UserWarning: A newer version of deeplake (3.6.4) is
available. It's recommended that you update to the latest version using `pip install -U deeplake`.
      warnings.warn(
```

```
llm = ChatOpenAI(temperature=0, model="gpt-3.5-turbo-0613")
```

Simplest approach, only specifying type

We can start by specifying a few properties with their expected type in our schema

```
schema = {
    "properties": {
        "sentiment": {"type": "string"},
        "aggressiveness": {"type": "integer"},
        "language": {"type": "string"},
    }
}
```

```
chain = create_tagging_chain(schema, llm)
```

As we can see in the examples, it correctly interprets what we want but the results vary so that we get, for example, sentiments in different languages ('positive', 'enojado' etc.).

We will see how to control these results in the next section.

```
inp = "Estoy increiblemente contento de haberte conocido! Creo que seremos muy buenos amigos!"
chain.run(inp)
```

```
    {'sentiment': 'positive', 'language': 'Spanish'}
```

```
inp = "Estoy muy enojado con vos! Te voy a dar tu merecido!"
chain.run(inp)
```

```
    {'sentiment': 'enojado', 'aggressiveness': 1, 'language': 'Spanish'}
```

```
inp = "Weather is ok here, I can go outside without much more than a coat"
chain.run(inp)
```

```
    {'sentiment': 'positive', 'aggressiveness': 0, 'language': 'English'}
```

More control

By being smart about how we define our schema we can have more control over the model's output. Specifically we can define:

- possible values for each property
- description to make sure that the model understands the property
- required properties to be returned

Following is an example of how we can use *enum*, *description* and *required* to control for each of the previously mentioned aspects:

```
schema = {
    "properties": {
        "sentiment": {"type": "string", "enum": ["happy", "neutral", "sad"]},
        "aggressiveness": {
            "type": "integer",
            "enum": [1, 2, 3, 4, 5],
            "description": "describes how aggressive the statement is, the higher the number the more
aggressive",
        },
        "language": {
            "type": "string",
```

```
            "enum": ["spanish", "english", "french", "german", "italian"],
        },
    },
    "required": ["language", "sentiment", "aggressiveness"],
}
```

```
chain = create_tagging_chain(schema, llm)
```

Now the answers are much better!

```
inp = "Estoy increiblemente contento de haberte conocido! Creo que seremos muy buenos amigos!"
chain.run(inp)
```

```
    {'sentiment': 'happy', 'aggressiveness': 0, 'language': 'spanish'}
```

```
inp = "Estoy muy enojado con vos! Te voy a dar tu merecido!"
chain.run(inp)
```

```
    {'sentiment': 'sad', 'aggressiveness': 10, 'language': 'spanish'}
```

```
inp = "Weather is ok here, I can go outside without much more than a coat"
chain.run(inp)
```

```
{'sentiment': 'neutral', 'aggressiveness': 0, 'language': 'english'}
```

Specifying schema with Pydantic

We can also use a Pydantic schema to specify the required properties and types. We can also send other arguments, such as 'enum' or 'description' as can be seen in the example below.

By using the `create_tagging_chain_pydantic` function, we can send a Pydantic schema as input and the output will be an instantiated object that respects our desired schema.

In this way, we can specify our schema in the same manner that we would a new class or function in Python - with purely Pythonic types.

```
from enum import Enum
from pydantic import BaseModel, Field
```

```
class Tags(BaseModel):
    sentiment: str = Field(..., enum=["happy", "neutral", "sad"])
    aggressiveness: int = Field(
        ...,
        description="describes how aggressive the statement is, the higher the number the more aggressive",
        enum=[1, 2, 3, 4, 5],
    )
    language: str = Field(
        ..., enum=["spanish", "english", "french", "german", "italian"]
    )
```

```
chain = create_tagging_chain_pydantic(Tags, llm)
```

```
inp = "Estoy muy enojado con vos! Te voy a dar tu merecido!"
res = chain.run(inp)
```

```
res
```

```
    Tags(sentiment='sad', aggressiveness=10, language='spanish')
```