Tagging and Extraction Using OpenAI functions

In [1]:
```
import os
import openai

from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
openai.api_key = os.environ['OPENAI_API_KEY']
```

In [2]:
```
from typing import List
from pydantic import BaseModel, Field
from langchain.utils.openai_functions import convert_pydantic_to_openai_function
```

Tagging is basically Classifying the Text, like here we will be tagging the text either into poitive, negative, neutral sentiment. And also we will be classifying the langugage used in the text.

Making the Pydantic Class:

In [3]:
```
class Tagging(BaseModel):
    """Tag the piece of text with particular info."""
    sentiment: str = Field(description="sentiment of text, should be `pos`, `neg`, or `neutr
    language: str = Field(description="language of text (should be ISO 639-1 code)")
```

In [4]:
```
convert_pydantic_to_openai_function(Tagging)
```

```
{'name': 'Tagging',
 'description': 'Tag the piece of text with particular info.',
 'parameters': {'title': 'Tagging',
  'description': 'Tag the piece of text with particular info.',
  'type': 'object',
  'properties': {'sentiment': {'title': 'Sentiment',
    'description': 'sentiment of text, should be `pos`, `neg`, or `neutral`',
    'type': 'string'},
   'language': {'title': 'Language',
    'description': 'language of text (should be ISO 639-1 code)',
    'type': 'string'}},
  'required': ['sentiment', 'language']}}
```

In [5]:
```
from langchain.prompts import ChatPromptTemplate
from langchain.chat_models import ChatOpenAI
```

In [6]:
```
model = ChatOpenAI(temperature=0)
```

In [7]:
```
tagging_functions = [convert_pydantic_to_openai_function(Tagging)]
```

In [8]:
```
tagging_functions
```

```
[{'name': 'Tagging',
  'description': 'Tag the piece of text with particular info.',
  'parameters': {'title': 'Tagging',
   'description': 'Tag the piece of text with particular info.',
   'type': 'object',
   'properties': {'sentiment': {'title': 'Sentiment',
     'description': 'sentiment of text, should be `pos`, `neg`, or `neutral`',
     'type': 'string'},
    'language': {'title': 'Language',
     'description': 'language of text (should be ISO 639-1 code)',
     'type': 'string'}},
   'required': ['sentiment', 'language']}}]
```

In [9]:
```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Think carefully, and then tag the text as instructed"),
    ("user", "{input}")
])
```

In [10]:
```
model_with_functions = model.bind(
    functions=tagging_functions,
    function_call={"name": "Tagging"}
)
```

In [11]:
```
tagging_chain = prompt | model_with_functions
```

In [12]:
```
tagging_chain.invoke({"input": "I love langchain"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Tagging', 'arguments':
'{\n  "sentiment": "pos",\n  "language": "en"\n}'}})
```

In [13]:
```
tagging_chain.invoke({"input": "non mi piace questo cibo"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Tagging', 'arguments':
'{\n  "sentiment": "neg",\n  "language": "it"\n}'}})
```

In [14]:
```
from langchain.output_parsers.openai_functions import JsonOutputFunctionsParser
```

In [15]:
```
tagging_chain = prompt | model_with_functions | JsonOutputFunctionsParser()
```

In [16]:
```
tagging_chain.invoke({"input": "non mi piace questo cibo"})
```

```
{'sentiment': 'neg', 'language': 'it'}
```

Extraction

Extraction is similar to tagging, but used for extracting multiple pieces of information.

We will try to extract name and age from the text:

In [29]:
```
from typing import Optional
class Person(BaseModel):
    """Extract the following Information about a person."""
    name: str = Field(description="person's name")
    age: Optional[int] = Field(description="person's age")
```

We want to extract list of name, age. So we will define another Pydantic Class:

In [30]:
```
class Information(BaseModel):
    """Information to extract of a Person."""
    #list-->that will be list of Person's class information
    people: List[Person] = Field(description="List of info about people")
```

In [31]:
```
convert_pydantic_to_openai_function(Information)
```

```
{'name': 'Information',
 'description': 'Information to extract of a Person.',
 'parameters': {'title': 'Information',
  'description': 'Information to extract of a Person.',
  'type': 'object',
  'properties': {'people': {'title': 'People',
    'description': 'List of info about people',
    'type': 'array',
    'items': {'title': 'Person',
     'description': 'Extract the following Information about a person.',
     'type': 'object',
     'properties': {'name': {'title': 'Name',
       'description': "person's name",
       'type': 'string'},
      'age': {'title': 'Age',
       'description': "person's age",
       'type': 'integer'}},
     'required': ['name']}}},
  'required': ['people']}}
```

In [32]:
```
extraction_functions = [convert_pydantic_to_openai_function(Information)]

#we will force it use the function
extraction_model = model.bind(functions=extraction_functions, function_call={"name": "Inform
```

In [33]:
```
extraction_model.invoke("Joe is 30, his mom is Martha")
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Information', 'argument
s': '{\n  "people": [\n    {\n      "name": "Joe",\n      "age": 30\n    },\n    {\n      "na
me": "Martha",\n      "age": null\n    }\n  ]\n}'}})
```

As it is giving 0 as age where it is not given, so we want it to not show age or any info when it is not given:

In [34]:
```
prompt = ChatPromptTemplate.from_messages([
    ("system", "Extract the relevant information, if not explicitly provided do not guess. E
    ("human", "{input}")
])
```

In [35]:
```
extraction_chain = prompt | extraction_model
```

```
In [36]: extraction_chain.invoke({"input": "Joe is 30, his mom is Martha"})
```

```
AIMessage(content='', additional_kwargs={'function_call': {'name': 'Information', 'argument
s': '{\n  "people": [\n    {\n      "name": "Joe",\n      "age": 30\n    },\n    {\n      "na
me": "Martha"\n    }\n  ]\n}'}})
```

Parsing the Output to get required info only:

```
In [37]: extraction_chain = prompt | extraction_model | JsonOutputFunctionsParser()
```

```
In [38]: extraction_chain.invoke({"input": "Joe is 30, his mom is Martha"})
```

```
{'people': [{'name': 'Joe', 'age': 30}, {'name': 'Martha'}]}
```

We can extract on the base of the Key of dictionary:

```
In [39]: from langchain.output_parsers.openai_functions import JsonKeyOutputFunctionsParser
```

```
In [40]: extraction_chain = prompt | extraction_model | JsonKeyOutputFunctionsParser(key_name="people
```

```
In [41]: extraction_chain.invoke({"input": "Joe is 30, his mom is Martha"})
```

```
[{'name': 'Joe', 'age': 30}, {'name': 'Martha'}]
```

Doing it for real

We can apply tagging to a larger body of text.

For example, let's load this blog post and extract tag information from a sub-set of the text.

```
In [42]: from langchain.document_loaders import WebBaseLoader
         loader = WebBaseLoader("https://lilianweng.github.io/posts/2023-06-23-agent/")
         documents = loader.load()
```

```
In [43]: doc = documents[0]
```

```
In [44]: page_content = doc.page_content[:10000]
```

In [45]:
```
print(page_content[:1000])
```

```




LLM Powered Autonomous Agents | Lil'Log
```

We will do Tagging to get summary, classify text to language, and to get the keywords:

In [46]:
```
class Overview(BaseModel):
    """Overview of a section of text."""
    summary: str = Field(description="Provide a concise summary of the content.")
    language: str = Field(description="Provide the language that the content is written in."
    keywords: str = Field(description="Provide keywords related to the content.")
```

In [47]:
```
overview_tagging_function = [
    convert_pydantic_to_openai_function(Overview)
]
tagging_model = model.bind(
    functions=overview_tagging_function,
    function_call={"name":"Overview"}
)
tagging_chain = prompt | tagging_model | JsonOutputFunctionsParser()
```

In [48]:
```
tagging_chain.invoke({"input": page_content})
```

```
{'summary': 'This article discusses the concept of building autonomous agents powered by LLM
(large language model) as their core controller. It explores the key components of such agent
systems, including planning, memory, and tool use. It also covers various techniques for task
decomposition and self-reflection in autonomous agents. The article provides examples of case
studies and challenges in implementing LLM-powered autonomous agents.',
 'language': 'English',
 'keywords': 'LLM, autonomous agents, planning, memory, tool use, task decomposition, self-re
flection, case studies, challenges'}
```

Now for the Extraction, lets extract Research Papers Mentioned in the Citation Part:

In [55]:
```
class Paper(BaseModel):
    """Information about research papers mentioned."""
    title: str
    author: Optional[str]

#we want to extract list of papers so making another pydantic class
class Info(BaseModel):
    """Information to extract"""
    papers: List[Paper]
```

In [56]:
```
paper_extraction_function = [
    convert_pydantic_to_openai_function(Info)
]
extraction_model = model.bind(
    functions=paper_extraction_function,
    #forcing it to use info function
    function_call={"name":"Info"}
)
extraction_chain = prompt | extraction_model | JsonKeyOutputFunctionsParser(key_name="papers
```

In [57]:
```
extraction_chain.invoke({"input": page_content})
```

```
[{'title': 'LLM Powered Autonomous Agents', 'author': 'Lilian Weng'}]
```

So above result are not the papers that are mentioned in Citation Section, but it is the Title and author name of the main paper.

So lets try with the appropriate System Message:

In [58]:
```
system_message = """A article will be passed to you. Extract from it all papers that are men

Do not extract the name of the article itself. If no papers are mentioned that's fine -
you don't need to extract any! Just return an empty list.

Do not make up or guess ANY extra information. Only extract what exactly is in the text."""

prompt = ChatPromptTemplate.from_messages([
    ("system", system_message),
    ("human", "{input}")
])
```

In [59]:
```
extraction_chain = prompt | extraction_model | JsonKeyOutputFunctionsParser(key_name="papers
```

In [60]:
```
extraction_chain.invoke({"input": page_content})
```

```
[{'title': 'Chain of thought (CoT; Wei et al. 2022)', 'author': 'Wei et al.'},
 {'title': 'Tree of Thoughts (Yao et al. 2023)', 'author': 'Yao et al.'},
 {'title': 'LLM+P (Liu et al. 2023)', 'author': 'Liu et al.'},
 {'title': 'ReAct (Yao et al. 2023)', 'author': 'Yao et al.'},
 {'title': 'Reflexion (Shinn & Labash 2023)', 'author': 'Shinn & Labash'},
 {'title': 'Chain of Hindsight (CoH; Liu et al. 2023)',
  'author': 'Liu et al.'},
 {'title': 'Algorithm Distillation (AD; Laskin et al. 2023)',
  'author': 'Laskin et al.'}]
```

If we give it wrong input:

In [61]:
```
extraction_chain.invoke({"input": "hi"})
```

```
[]
```

So till now Page content we have been passing only have first 10,000 characters of the paper, if we want to pass whole paper. We can not do that directly because of the token limit, so we can split it to small pieces of chunks and then we can pass them individually and combine the result:

In [62]:
```
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_overlap=0)
```

In [63]:
```
splits = text_splitter.split_text(doc.page_content)
```

In [64]:
```
len(splits)
```

```
14
```

To merge the lists/or getting a list from matrix/list of lists, we will write a function that will be usefull when we will get a list from each chunk result and then combining the lists, we can convert them to single list:

In [65]:
```
def flatten(matrix):
    flat_list = []
    for row in matrix:
        flat_list += row
    return flat_list
```

In [66]:
```
flatten([[1, 2], [3, 4]])
```

```
[1, 2, 3, 4]
```

In [67]:
```
print(splits[0])
```

```
Task Decomposition

Self-Reflection


Component Two: Memory

Types of Memory

Maximum Inner Product Search (MIPS)


Component Three: Tool Use

Case Studies

Scientific Discovery Agent

Generative Agents Simulation
```

In [*]:
```
from langchain.schema.runnable import RunnableLambda
```

In [*]:
```
prep = RunnableLambda(
    lambda x: [{"input": doc} for doc in text_splitter.split_text(x)]
)
```

In [ ]:
```
prep.invoke("hi")
```

In [ ]:
```
chain = prep | extraction_chain.map() | flatten
```

In [ ]:
```
chain.invoke(doc.page_content)
```

In [ ]:
```
```