Evaluating Usability: A Comparative Analysis of the OpenAI-Instructor Protocol Framework and DSPy

Srijan Anand & Saksham Parihar
Indian Institute of Technology Kanpur

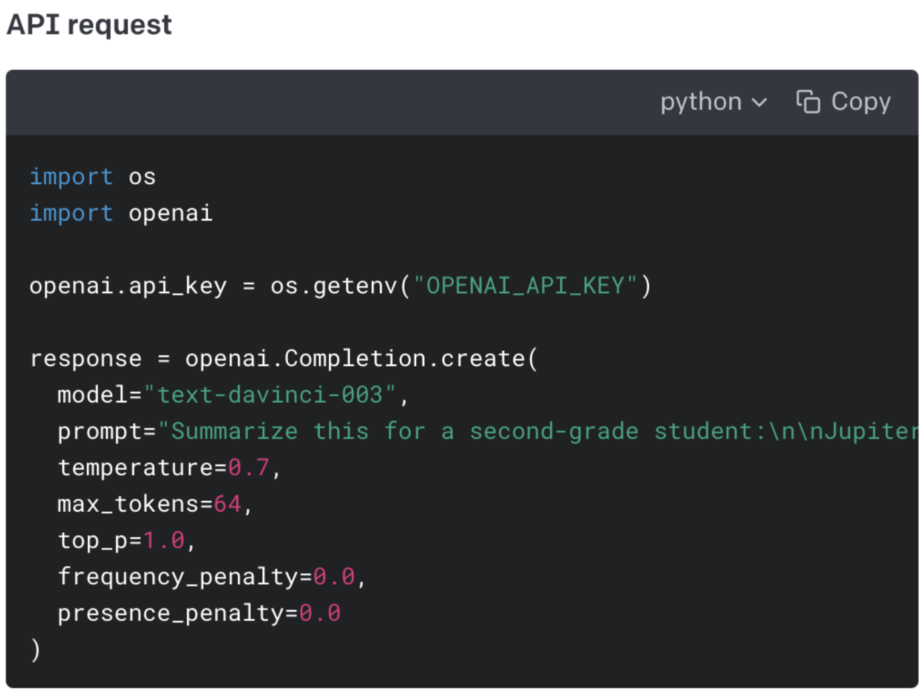
27th July 2024

Abstract

This paper presents the development and evaluation of a system designed to analyze natural language queries related to business intelligence data using large language models (LLMs). The primary focus is on identifying the most effective prompting mechanisms by experimenting with multiple LLM modeling frameworks. The study utilizes the OpenAI Chat Completion Protocol, the Instructor Package, and the DSPy Framework to query LLMs. These frameworks were evaluated using the TPCDS Benchmark, which includes definitions and queries in both natural language and SQL. We improved the analysis by querying multiple fact tables and dimension prompts, comparing the performance of the OpenAI-Instructor combination with DSPy. The results of this evaluation provided concrete reasoning for the Spotonix team to consider a shift to the DSPy Framework.

1 Introduction

Spotonix, Inc., headquartered in San Francisco, USA, specializes in transforming business data into natural language query formats, making it more accessible and actionable for businesses. The company offers innovative solutions that revolutionize the way enterprises interact with their data, providing insights at the fingertips of every employee. Spotonix leverages advanced business intelligence and data analytics to create a comprehensive enterprise knowledge graph, ensuring that teams can consistently and accurately answer any query, thereby enhancing the organization's overall efficiency and decision-making capabilities.
Spotonix currently employs the OpenAI Chat Completion protocol combined with the Instructor Protocol to handle natural language queries. However, to explore potential improvements, we investigated whether DSPy could offer better results, particularly in the business analytics domain, while maintaining the same computational power. Our experiments yielded positive outcomes, demonstrating that DSPy could enhance the efficiency and accuracy of handling queries, thereby potentially offering a more robust solution for business data analytics.


API request

```python
import os
import openai

openai.api_key = os.getenv("OPENAI_API_KEY")

response = openai.Completion.create(
  model="text-davinci-003",
  prompt="Summarize this for a second-grade student:\n\nJupiter
  temperature=0.7,
  max_tokens=64,
  top_p=1.0,
  frequency_penalty=0.0,
  presence_penalty=0.0
)
```

2 DSPy: A PyTorch Based Prompting Solution

2.1 Outline

DSPy is a framework for algorithmically optimizing LM prompts and weights, especially when LMs are used one or more times within a pipeline. To use LMs to build a complex system without DSPy, you generally have to: (1) break the problem down into steps, (2) prompt your LM well until each step works well in isolation, (3) tweak the steps to work well together, (4) generate synthetic examples to tune each step, and (5) use these examples to finetune smaller LMs to cut costs. Currently, this is hard and messy: every time you change your pipeline, your LM, or your data, all prompts (or finetuning steps) may need to change.

To make this more systematic and much more powerful, DSPy does two things. First, it separates the flow of your program (modules) from the parameters (LM prompts and weights) of each step. Second, DSPy introduces new optimizers, which are LM-driven algorithms that can tune the prompts and/or the weights of your LM calls, given a metric you want to maximize.

DSPy can routinely teach powerful models like GPT-3.5 or GPT-4 and local models like T5-base or Llama2-13b to be much more reliable at tasks, i.e. having higher quality and/or avoiding specific failure patterns. DSPy optimizers will "compile" the same program into different instructions, few-shot prompts, and/or weight updates (finetunes) for each LM. This is a new paradigm in which LMs and their prompts fade into the background as optimizable pieces of a larger system that can learn from data. **tldr**; less prompting, higher scores, and a more systematic approach to solving hard tasks with LMs.

DSPy

2.2 Analogy to Neural Networks

When we build neural networks, we don't write manual for-loops over lists of hand-tuned floats. Instead, you might use a framework like PyTorch to compose declarative layers (e.g., Convolution or Dropout) and then use optimizers (e.g., SGD or Adam) to learn the parameters of the network.

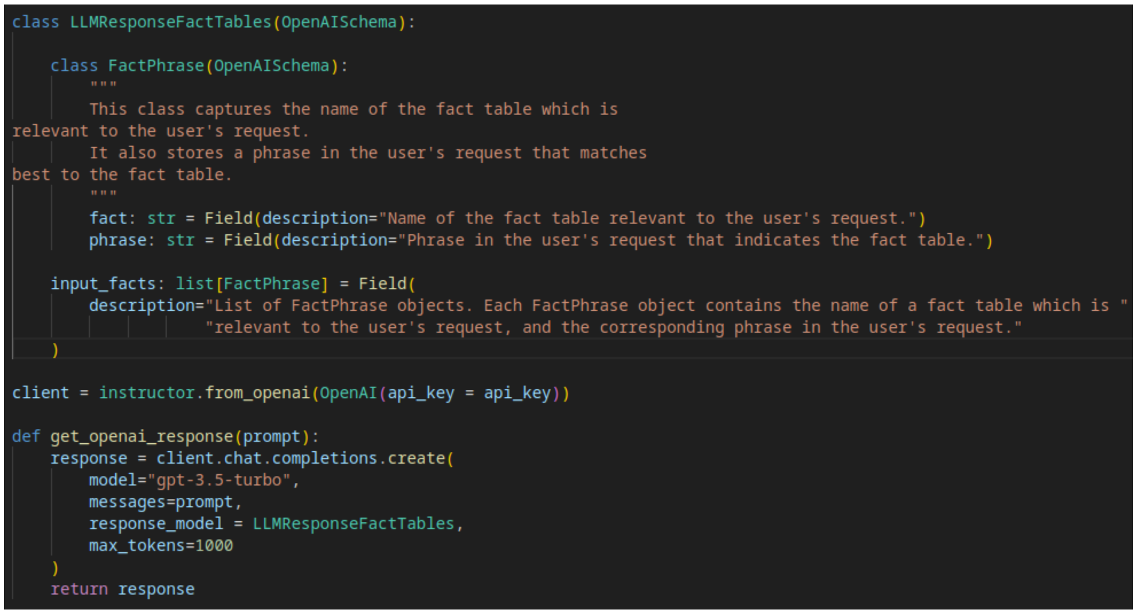
Ditto! DSPy gives you the right general-purpose modules (e.g., ChainOfThought, ReAct, etc.), which replace string-based prompting tricks. To replace prompt hacking and one-off synthetic data generators, DSPy also gives you general optimizers (BootstrapFewShotWithRandomSearch or BayesianSignatureOptimizer), which are algorithms that update parameters in your program. Whenever you modify your code, your data, your assertions, or your metric, you can compile your program again and DSPy will create new effective prompts that fit your changes.

3 Implementation

We utilized the TPCDS Benchmark Dataset to query multiple questions both fact tables and dimension prompts, ensuring a thorough evaluation of each framework's performance in handling complex business intelligence scenarios.

3.1 OpenAI and Instructor API Implementation

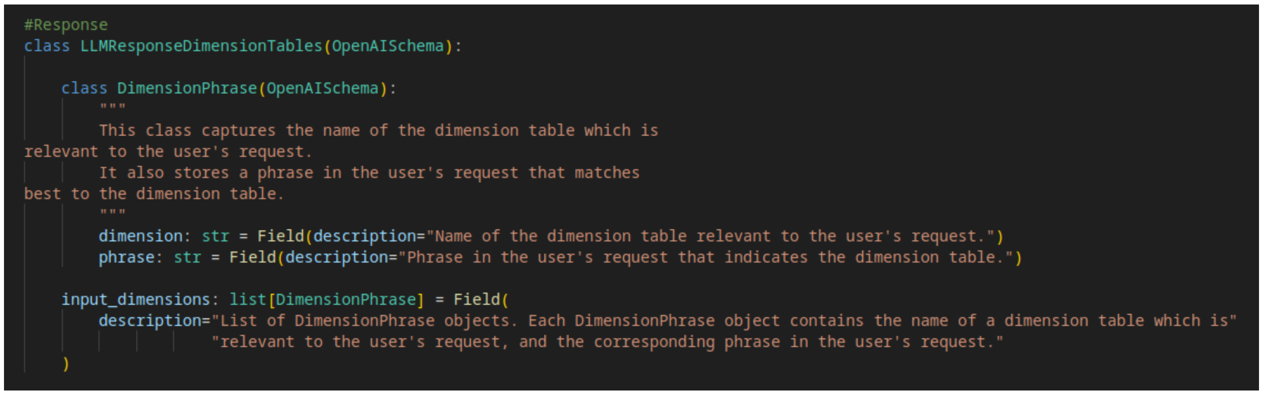
This implementation defines a schema for handling natural language queries related to business intelligence data. The LLMResponseFactTables class includes a nested FactPhrase class, capturing the name of fact tables relevant to user requests and corresponding phrases indicating these tables. The inputfacts field is a list of FactPhrase objects, each containing a fact table name and a matching user request phrase. The code initializes a client using the OpenAI API and utilizes the instructor package to create chat completions with the GPT-3.5-turbo model. The getopenairesponse function generates responses based on the provided prompts, ensuring accurate and relevant data retrieval from fact tables.



```python
class LLMResponseFactTables(OpenAISchema):

    class FactPhrase(OpenAISchema):
        """
        This class captures the name of the fact table which is
relevant to the user's request.
        It also stores a phrase in the user's request that matches
best to the fact table.
        """
        fact: str = Field(description="Name of the fact table relevant to the user's request.")
        phrase: str = Field(description="Phrase in the user's request that indicates the fact table.")

    input_facts: list[FactPhrase] = Field(
        description="List of FactPhrase objects. Each FactPhrase object contains the name of a fact table which is "
                    "relevant to the user's request, and the corresponding phrase in the user's request."
    )

client = instructor.from_openai(OpenAI(api_key = api_key))

def get_openai_response(prompt):
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=prompt,
        response_model = LLMResponseFactTables,
        max_tokens=1000
    )
    return response
```

Figure 1: FactTables

For handling dimension prompts, we implemented a similar structure with the LLMResponseDimensionTables class. This class includes a nested DimensionPhrase class, which captures the name of the dimension table relevant to the user's request and stores a phrase from the user's request that matches best to the dimension table. The inputdimensions field is a list of DimensionPhrase objects, each containing the dimension table name and the corresponding phrase from the user's request. This approach ensures that dimension-related queries are accurately mapped and processed, enhancing the system's ability to respond effectively to natural language queries involving dimension tables.

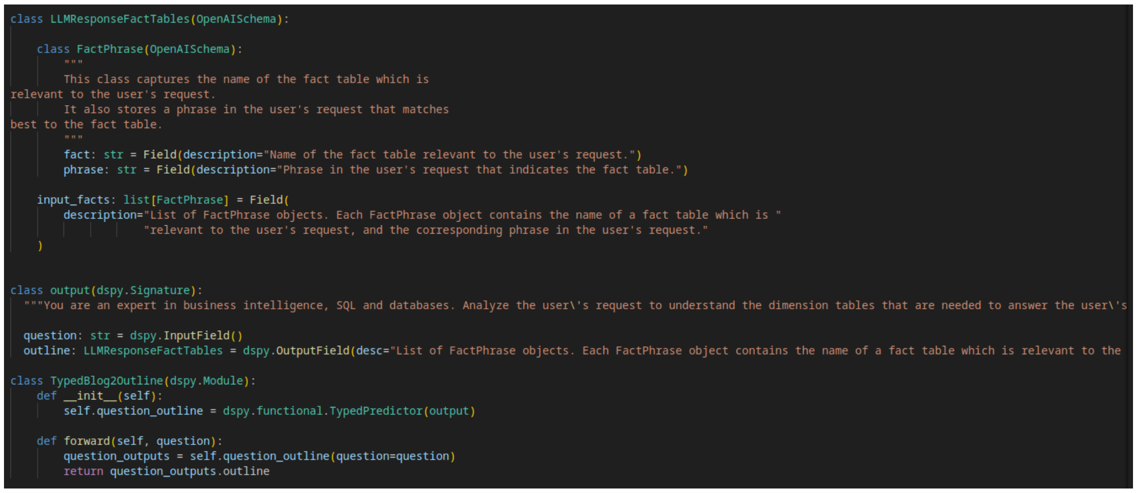


```python
#Response
class LLMResponseDimensionTables(OpenAISchema):

    class DimensionPhrase(OpenAISchema):
        """
        This class captures the name of the dimension table which is
relevant to the user's request.
        It also stores a phrase in the user's request that matches
best to the dimension table.
        """
        dimension: str = Field(description="Name of the dimension table relevant to the user's request.")
        phrase: str = Field(description="Phrase in the user's request that indicates the dimension table.")

    input_dimensions: list[DimensionPhrase] = Field(
        description="List of DimensionPhrase objects. Each DimensionPhrase object contains the name of a dimension table which is"
                    "relevant to the user's request, and the corresponding phrase in the user's request."
    )
```

Figure 2: Dimensions

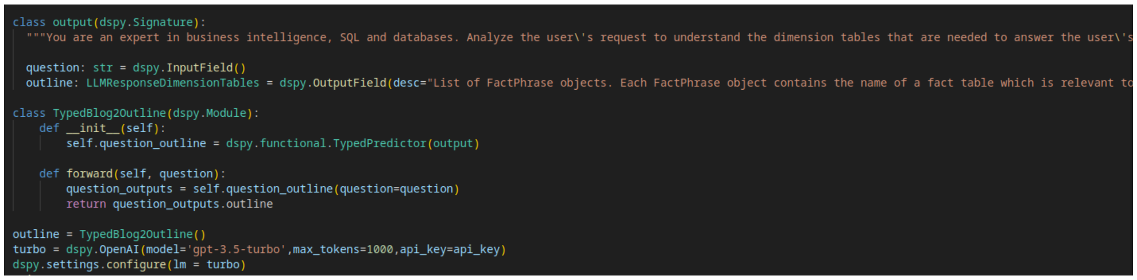
3.2 DSPy Implementation

In DSPy, we implemented a similar structure to handle natural language queries for fact tables & dimensions. The classes defines the schema for capturing the fact table name and the corresponding phrase & for dimensions, the dimension and corresponding phrase from the user's request. The output class uses DSPy's Signature to specify the input question and the expected output format, leveraging LLMResponseFactTables & LLMResponseDimensionTables. The TypedBlog2Outline module uses DSPy's functional API to process the question and generate the appropriate outline based on the user's request. This implementation enables efficient and accurate analysis of business intelligence queries within the DSPy framework.



```python
class LLMResponseFactTables(OpenAISchema):

    class FactPhrase(OpenAISchema):
        """
        This class captures the name of the fact table which is
relevant to the user's request.
        It also stores a phrase in the user's request that matches
best to the fact table.
        """
        fact: str = Field(description="Name of the fact table relevant to the user's request.")
        phrase: str = Field(description="Phrase in the user's request that indicates the fact table.")

    input_facts: list[FactPhrase] = Field(
        description="List of FactPhrase objects. Each FactPhrase object contains the name of a fact table which is "
                    "relevant to the user's request, and the corresponding phrase in the user's request."
    )


class output(dspy.Signature):
  """You are an expert in business intelligence, SQL and databases. Analyze the user\'s request to understand the dimension tables that are needed to answer the user\'s

  question: str = dspy.InputField()
  outline: LLMResponseFactTables = dspy.OutputField(desc="List of FactPhrase objects. Each FactPhrase object contains the name of a fact table which is relevant to the

class TypedBlog2Outline(dspy.Module):
    def __init__(self):
        self.question_outline = dspy.functional.TypedPredictor(output)

    def forward(self, question):
        question_outputs = self.question_outline(question=question)
        return question_outputs.outline
```

Figure 3: FactTables DSPy



```python
class output(dspy.Signature):
  """You are an expert in business intelligence, SQL and databases. Analyze the user\'s request to understand the dimension tables that are needed to answer the user\'s

  question: str = dspy.InputField()
  outline: LLMResponseDimensionTables = dspy.OutputField(desc="List of FactPhrase objects. Each FactPhrase object contains the name of a fact table which is relevant to

class TypedBlog2Outline(dspy.Module):
    def __init__(self):
        self.question_outline = dspy.functional.TypedPredictor(output)

    def forward(self, question):
        question_outputs = self.question_outline(question=question)
        return question_outputs.outline

outline = TypedBlog2Outline()
turbo = dspy.OpenAI(model='gpt-3.5-turbo',max_tokens=1000,api_key=api_key)
dspy.settings.configure(lm = turbo)
```

Figure 4: Dimensions DSPy

4 Final Results

4.1 Examples

Here are the examples and the corresponding outputs from both the frameworks :

1. Question : Get all items that were (i) sold in stores in a particular month and year and (ii) returned and re-purchased by the customer through the catalog channel in the same month and in the six following months. For these items, compute the sum of net profit of store sales, net loss of store loss and net profit of catalog. Group this information by item and store.

OpenAI: [FactPhrase(fact='store sales data', phrase='sold in stores'), FactPhrase(fact='store returns', phrase='returned and re-purchased by the customer through the catalog channel')]
DSPy: [FactPhrase(fact='store sales data', phrase='sold in stores'), FactPhrase(fact='store returns', phrase='returned and re-purchased by the customer through the catalog channel'), FactPhrase(fact='catalog sales', phrase='net profit of catalog')]

2. Question : Compute, for each county, the average quantity, list price, coupon amount, sales price, net profit, age, and number of dependents for all items purchased through catalog sales in a given year by customers who were born in a given list of six months and living in a given list of seven states and who also belong to a given gender and education demographic.

OpenAI: [DimensionPhrase(dimension='customer', phrase='customers who were born'), DimensionPhrase(dimension='date', phrase='given year'), DimensionPhrase(dimension='customer demographics', phrase='customers who were born,living,gender,education demographic')]
DSPy: [DimensionPhrase(dimension='customer demographics', phrase='gender, education demographic'), DimensionPhrase(dimension='customer', phrase='customers'), DimensionPhrase(dimension='date', phrase='given year, born in a given list of six months'), DimensionPhrase(dimension='customer address', phrase='living in a given list of seven states'), DimensionPhrase(dimension='tpcds item', phrase='items purchased through catalog sales')]

4.2 Final Thoughts

In our comparative analysis, the results produced by the DSPy model were significantly more accurate than those from the OpenAI and Instructor Package, despite both utilizing the same underlying model, GPT-3.5-turbo.

For instance, when querying for fact tables and corresponding phrases, the OpenAI model's output was insufficient, returning only two facts and omitting critical information. In contrast, the DSPy model not only captured the relevant facts and phrases more accurately but also identified the 'catalog sales' fact, which the OpenAI model missed.

Similarly, in the case of dimension tables, the OpenAI model overlooked important dimensions and exhibited issues with phrase capturing. The DSPy model, however, demonstrated superior performance by accurately identifying and processing the necessary dimensions and phrases.

These observations were consistent across multiple queries, and detailed results, along with the complete code, are available on our GitHub.

In summary, we have gathered substantial evidence that DSPy is a superior alternative to the OpenAI and Instructor Package for handling business intelligence queries. DSPy provides more accurate and comprehensive results while utilizing the same computational resources, making it a more effective solution for business data analytics.

5 References

- Stanford NLP - DSPy
- OpenAI API
- Instructor Protocol API
- TPCDS Benchmark Dataset

Spotonix