ChatGPT Prompt Engineering

Mini Course instructed by DeepLearning.AI.

LESSON 1: Introduction

Large Learning Models (LLMs) – ML models trained with huge trainsets of data. They haven't been designed with a specific purpose, but with the idea of covering as many different types of inputs as possible.

LLMs can be divided into two main groups: Base LLMs and Instruction Provided LLMs.

- **Base LLMs** – Try to predict the following words and sentences of the input given.
- **Instruction Provided LLMs** – Are able to interpret the instructions described in the input.

e.g.

User input: What's the capital of France?

Base LLMs: What language people speak in France? What is the typical food in France?

Instruction Provided LLMs: The capital of France is Paris.

LESSON 2: Main Guidelines for Prompting

There are two main prompting principles:

1. **Write clear and specific instructions.**
2. **Give the model time to "think".**

At the same time, each principle can be divided into more precise practices.

1. Write clear and specific instructions.
   1.1. Use **delimiters** (" ", < >, ´´´ ´´´, etc.) to indicate distinct parts of the input.

   Why? To avoid prompt injections. To let the model know which piece of text it must work with.

   1.2. Specify the **structure of the output** (HTML, JSON, a value from a fixed set, etc.)

   Why? This way we can directly use the output in other pieces of our code. So the answers are more consistent with each other.

   1.3. Ask the model whether **conditions are satisfied**.

   e.g. " … if the answer contains a number of steps, return a list of the steps. Return me None otherwise."

Why? To handle edge-cases better when the model doesn't work as predicted.

1.4. **Provide** the model with **an example**.

Why? So the model can infer easily what you are expecting from it. Particularly if it's hard to describe the desired output.

2. Give the model time to "think".

Sometimes, if the model doesn't have the enough time to think, it will rush into a hurried and wrong conclusion. Especially when dealing with complex instructions such as math problems. There are many techniques for trying to avoid this situation.

2.1. **Specify the steps** required to complete the task.

Why? When the model is given a path of smaller tasks, it is more likely that it doesn't output a wrong conclusion. Also, this way, the model has a path to follow to reach the answer and chances of following a wrong trail decrease.

2.2. Instruct the model to **work out its own solution before** rushing into a conclusion.

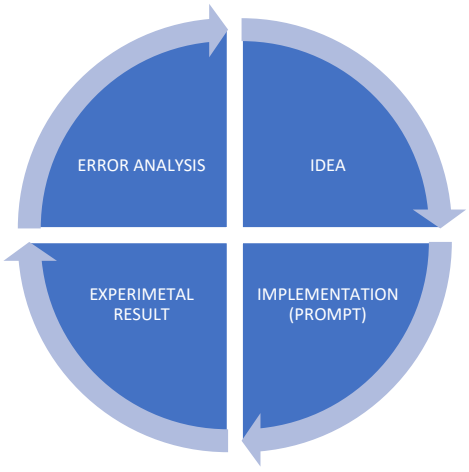
Why? Before deciding to evaluate something, let the AI work it's solution and then compared the answer that needs to be evaluated with its own.

Hallucinations – Not real inferences that the model makes as a result of being forced to complete a task.

How can we minimize them? Asking it to **gather information before** and then extract the conclusion from that information and not from all the data it has access to.

LESSON 3: Iterative Prompt Development

We will study how to improve our prompts in order to get better outputs. Most of the time, the first attempt is always a failure. Through iterative prompt development, we can improve our prompt, improving our results the next time.



1. **Idea** – try something (first approach).

2. **Implementation** (prompt) – clarify instructions, give the model more time to think (if needed).

3. Experimental result.

4. **Error Analysis** – Analyze **where** the result does not give what you want.

5. Refine prompt with a **batch of examples** – To see if the model has been overfitted or not.

E.g.

Examples of how the result might not be working:

1. *Text is **too long**.*
2. *Text does **not focus** on the **right details**.*
3. *Description **needs a table** of dimensions.*

LESSON 4: Summarizing Text

When summarizing, we can obtain a much better result by using certain techniques:

1. Summarize with a **word/sentence/character limit**.
2. **Focus on a certain aspect** of the text. *E.g., summarize with a focus on shipping and delivery.*
3. Use "**extract**" instead of "summarize" – The result will be much more focused on the specific aspect and will not include irrelevant information.

LESSON 5: Inferring

Task: The model has to make an analysis of the text given. This analysis can be either inferring emotions, sentiments, topics, etc. Or extracting information from the text such as a name or the qualities of something.

Before LLMs, this task was done using supervised learning models, having to label a test set, then having to create the model. Thanks to LLMs, the developing time is much less.

LESSON 6: Transforming

Trask: Inputting a text and transforming it so it meets certain requirements.

Universal Translation – Apart from simple translation (form one language to another), you can translate multiple languages at the same time.

Tone transformation – Writing can vary based on the intended audience. ChatGPT can produce different tones.

LESSON 7: Expanding

Task: Given a small text, such as a list, the beginning of an essay or a set of instructions, the model has to expand it.

Temperature – another model parameter. Indicates the randomness of the model. Range 0 to 1. It can be also understood as the "creativity" of the model.

- For tasks that require reliability, predictability: temperature = 0
- For tasks that require variety: 0.3 <= temperature <= 0.7

LESSON 8: Chatbot

There are two ways to interact with openai's API.

1. **OPENAI normal API call**
    - Interacting just through prompting (using **"user" messages**).
    - The model does not have access to previous messages.
2. **Role**
    - Interacting with "system" and "user" messages.
    - **"system" messages** establish the main instructions the model must follow.
    - The model has **access to previous messages**, so it can infer information from them.

```
# Common API setup
import os
import openai
from dotenv import load_dotenv, find_dotenv
_ = load_dotenv(find_dotenv()) # read local .env file
```

```
# 1. First type of completion (just with prompt)
def get_completion(prompt, model="gpt-3.5-turbo", temperature=0):
    messages = [{"role": "user", "content": prompt}] #
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of randomness of
the model's output
    )
    return response.choices[0].message["content"]
```

```
# 2. Second type of completion (passing the list of messages)
def get_completion_from_messages(messages, model="gpt-3.5-turbo",
temperature=0):
    response = openai.ChatCompletion.create(
        model=model,
        messages=messages,
        temperature=temperature, # this is the degree of randomness of
the model's output
    )
#     print(str(response.choices[0].message))
    return response.choices[0].message["content"]
```

```
# Example of messages
```

```
messages =  [
{'role':'system', 'content':'You are an assistant that speaks like
Shakespeare.'}, # system defines the main instruction for the model
{'role':'user', 'content':'tell me a joke'}, # user is the input to the
model
{'role':'assistant', 'content':'Why did the chicken cross the road'}, #
assistant is the model's response
{'role':'user', 'content':'I don\'t know'}  ]
response = get_completion_from_messages(messages, temperature=1)
print(response)
```

OrderBot – We can automate the collection of user prompts and assistant responses to build an OrderBot. The OrderBot will take orders at a pizza restaurant.

```
def collect_messages(_):
    prompt = input(context.last_rensponse) # Pseudocode to obtain the
last response of the assistant
    context.append({'role':'user', 'content':f"{prompt}"})
    response = get_completion_from_messages(context)
    context.append({'role':'assistant', 'content':f"{response}"})

    return response
```