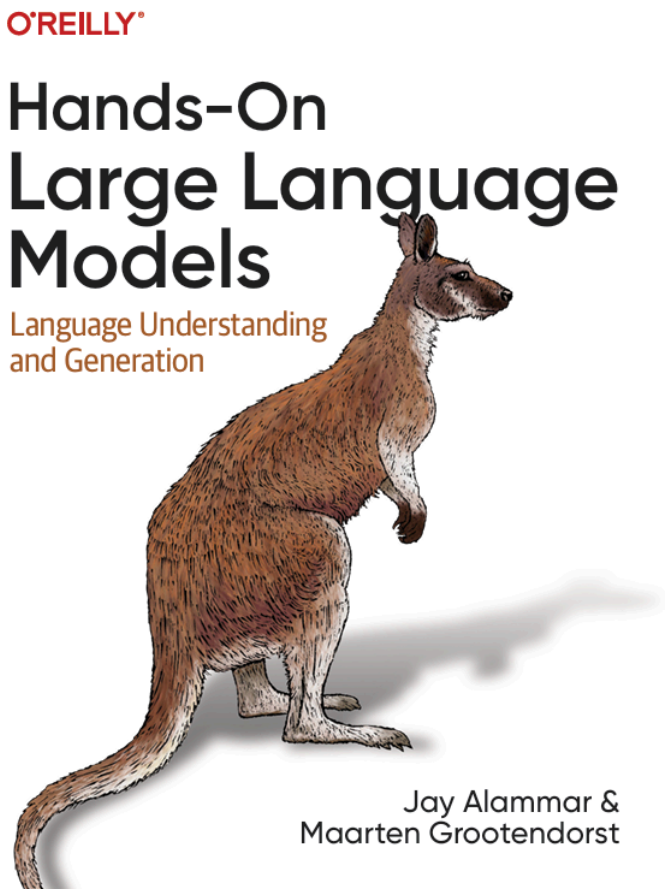
Chapter 6 - Prompt Engineering

Methods for improving the output through prompt engineering.

Buy the Book! O'Reilly GitHub Repository Open in Colab

This notebook is for Chapter 6 of the Hands-On Large Language Models book by Jay Alammar and Maarten Grootendorst.



[OPTIONAL] - Installing Packages on Google colab

If you are viewing this notebook on Google Colab (or any other cloud vendor), you need to **uncomment and run** the following codeblock to install the dependencies for this chapter:

💡 **NOTE**: We will want to use a GPU to run the examples in this notebook. In Google Colab, go to **Runtime > Change runtime type > Hardware accelerator > GPU > GPU type > T4**.

```
# %%capture
# !pip install langchain>=0.1.17 openai>=1.13.3 langchain_openai>=0.1.6 transfc
# !CMAKE_ARGS="-DLLAMA_CUBLAS=on" pip install llama-cpp-python
```

Loading our model

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=False,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

# Create a pipeline
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False,
)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserW
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access publ
  warnings.warn(
`torch_dtype` is deprecated! Use `dtype` instead!
Device set to use cuda
The following generation flags are not valid and may be ignored: ['temperature']
```

```
# Prompt
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
]

# Generate the output
output = pipe(messages)
print(output[0]["generated_text"])
```

```
WARNING:transformers_modules.microsoft.Phi-3-mini-4k-instruct.920b6cf52a79ecff57
 Why don't chickens like to go to the gym? Because they can't crack the egg-sist
```

```
# Apply prompt template
prompt = pipe.tokenizer.apply_chat_template(messages, tokenize=False)
print(prompt)
```

```
<s><|user|>
Create a funny joke about chickens.<|end|>
<|assistant|>
```

```
# Using a high temperature
output = pipe(messages, do_sample=True, temperature=1)
print(output[0]["generated_text"])
```

```
 Why don't chickens ever play hide and seek? Because good luck hiding when every
```

```
# Using a high top_p
output = pipe(messages, do_sample=True, top_p=1)
print(output[0]["generated_text"])
```

```
 Why don't chickens like math class? Because they can't solve for "x" in their e
```

Intro to Prompt Engineering

The Basic Ingredients of a Prompt

Advanced Prompt Engineering

Complex Prompt

```
# Text to summarize which we stole from https://jalammar.github.io/illustrated-
text = """In the previous post, we looked at Attention – a ubiquitous method ir
The Transformer was proposed in the paper Attention is All You Need. A TensorFl
Let’s begin by looking at the model as a single black box. In a machine transla
Popping open that Optimus Prime goodness, we see an encoding component, a decoc
The encoding component is a stack of encoders (the paper stacks six of them on
The encoders are all identical in structure (yet they do not share weights). Ea
The encoder’s inputs first flow through a self-attention layer – a layer that h
```

```
The outputs of the self-attention layer are fed to a feed-forward neural networ
The decoder has both those layers, but between them is an attention layer that
Now that we've seen the major components of the model, let's start to look at t
As is the case in NLP applications in general, we begin by turning each input w
Each word is embedded into a vector of size 512. We'll represent those vectors
The embedding only happens in the bottom-most encoder. The abstraction that is
After embedding the words in our input sequence, each of them flows through eac
Here we begin to see one key property of the Transformer, which is that the wor
Next, we'll switch up the example to a shorter sentence and we'll look at what
Now We're Encoding!
As we've mentioned already, an encoder receives a list of vectors as input. It
"""

# Prompt components
persona = "You are an expert in Large Language models. You excel at breaking do
instruction = "Summarize the key findings of the paper provided.\n"
context = "Your summary should extract the most crucial points that can help re
data_format = "Create a bullet-point summary that outlines the method. Follow t
audience = "The summary is designed for busy researchers that quickly need to g
tone = "The tone should be professional and clear.\n"
text = "MY TEXT TO SUMMARIZE"  # Replace with your own text to summarize
data = f"Text to summarize: {text}"

# The full prompt - remove and add pieces to view its impact on the generated c
query = persona + instruction + context + data_format + audience + tone + data
```

```
messages = [
    {"role": "user", "content": query}
]
print(tokenizer.apply_chat_template(messages, tokenize=False))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-130493804.py in <cell line: 0>()
      2     {"role": "user", "content": query}
      3 ]
----> 4 print(tokenizer.apply_chat_template(messages, tokenize=False))

NameError: name 'tokenizer' is not defined
```

```
# Generate the output
outputs = pipe(messages)
print(outputs[0]["generated_text"])
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-1617680404.py in <cell line: 0>()
      1 # Generate the output
----> 2 outputs = pipe(messages)
      3 print(outputs[0]["generated_text"])

NameError: name 'pipe' is not defined
```

In-Context Learning: Providing Examples

```
# Use a single example of using the made-up word in a sentence
one_shot_prompt = [
    {
        "role": "user",
        "content": "A 'Gigamuru' is a type of Japanese musical instrument. An e
    },
    {
        "role": "assistant",
        "content": "I have a Gigamuru that my uncle gave me as a gift. I love t
    },
    {
        "role": "user",
        "content": "To 'screeg' something is to swing a sword at it. An example
    }
]
print(tokenizer.apply_chat_template(one_shot_prompt, tokenize=False))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-2794044972.py in <cell line: 0>()
     14     }
     15 ]
---> 16 print(tokenizer.apply_chat_template(one_shot_prompt, tokenize=False))

NameError: name 'tokenizer' is not defined
```

```
# Generate the output
outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
```

```
 During the intense duel, the knight skillfully screeged his opponent's shield,
```

Chain Prompting: Breaking up the Problem

```
# Create name and slogan for a product
product_prompt = [
    {"role": "user", "content": "Create a name and slogan for a chatbot that le
]
outputs = pipe(product_prompt)
product_description = outputs[0]["generated_text"]
print(product_description)
```

```
 Name: "MindMeld Messenger"

Slogan: "Unleashing Intelligent Conversations, One Response at a Time"
```

```
# Based on a name and slogan for a product, generate a sales pitch
sales_prompt = [
    {"role": "user", "content": f"Generate a very short sales pitch for the fol
]
outputs = pipe(sales_prompt)
sales_pitch = outputs[0]["generated_text"]
print(sales_pitch)
```

```
 Introducing MindMeld Messenger - your ultimate communication partner! Unleash i
```

Reasoning with Generative Models

Chain-of-Thought: Think Before Answering

```
# # Answering without explicit reasoning
# standard_prompt = [
#     {"role": "user", "content": "Roger has 5 tennis balls. He buys 2 more car
#     {"role": "assistant", "content": "11"},
#     {"role": "user", "content": "The cafeteria had 25 apples. If they used 20
# ]

# # Run generative model
# outputs = pipe(standard_prompt)
# print(outputs[0]["generated_text"])
```

```
 The cafeteria started with 25 apples. They used 20 apples to make lunch, so the

25 - 20 = 5 apples left.

Then they bought 6 more apples, so they now have:

5 + 6 = 11 apples.
```

```
# Answering with chain-of-thought
cot_prompt = [
    {"role": "user", "content": "Roger has 5 tennis balls. He buys 2 more cans
    {"role": "assistant", "content": "Roger started with 5 balls. 2 cans of 3 t
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20 t
]

# Generate the output
outputs = pipe(cot_prompt)
print(outputs[0]["generated_text"])
```

```
You seem to be using the pipelines sequentially on GPU. In order to maximize eff
 The cafeteria started with 23 apples. They used 20 apples, so they had 23 - 20
```

Zero-shot Chain-of-Thought

```
# Zero-shot Chain-of-Thought
zeroshot_cot_prompt = [
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20 t
]

# Generate the output
outputs = pipe(zeroshot_cot_prompt)
print(outputs[0]["generated_text"])
```

```
 Step 1: Start with the initial number of apples, which is 23.
Step 2: Subtract the number of apples used to make lunch, which is 20. So, 23 -
Step 3: Add the number of apples bought, which is 6. So, 3 + 6 = 9 apples.

The cafeteria now has 9 apples.
```

Tree-of-Thought: Exploring Intermediate Steps

```
# Zero-shot Chain-of-Thought
zeroshot_tot_prompt = [
    {"role": "user", "content": "Imagine three different experts are answering
]
```

```
# Generate the output
outputs = pipe(zeroshot_tot_prompt)
print(outputs[0]["generated_text"])
```

```
 Expert 1: Step 1 - Start with the initial number of apples: 23 apples.

Expert 2: Step 1 - Subtract the apples used for lunch: 23 - 20 = 3 apples remain
```

```
Expert 3: Step 1 - Add the newly bought apples: 3 + 6 = 9 apples.


Expert 1: Step 2 - Confirm the final count: The cafeteria has 9 apples.

Expert 2: Step 2 - Review the calculations: 23 - 20 = 3, then 3 + 6 = 9. The cal

Expert 3: Step 2 - Agree with the result: The cafeteria indeed has 9 apples.


All experts agree on the final count: The cafeteria has 9 apples.
```

Output Verification

Providing Examples

```
# Zero-shot learning: Providing no examples
zeroshot_prompt = [
    {"role": "user", "content": "Create a character profile for an RPG game in
]

# Generate the output
outputs = pipe(zeroshot_prompt)
print(outputs[0]["generated_text"])
```


```
 {
  "character_profile": {
    "name": "Eldrin Stormbringer",
    "race": "Human",
    "class": "Warlock",
    "level": 5,
    "alignment": "Chaotic Good",
    "attributes": {
      "strength": 8,
      "dexterity": 14,
      "constitution": 10,
      "intelligence": 12,
      "wisdom": 10,
      "charisma": 16
    },
    "skills": [
      {
        "name": "Fireball",
        "proficiency": 18
      },
      {
        "name": "Shadowstep",
        "proficiency": 16
      },
```

```
        {
          "name": "Charm Person",
          "proficiency": 14
        }
      ],
      "equipment": [
        {
          "name": "Stormbringer Staff",
          "type": "Magic Weapon",
          "damage": 15
        },
        {
          "name": "Leather Armor",
          "type": "Light Armor",
          "defense": 10
        },
        {
          "name": "Ring of Protection",
          "type": "Magic Armor",
          "defense": 5
        }
      ],
      "background": "Eldrin grew up in a small village, where he was known for his
    }
}
```

```
# One-shot learning: Providing an example of the output structure
one_shot_template = """Create a short character profile for an RPG game. Make s

{
  "description": "A SHORT DESCRIPTION",
  "name": "THE CHARACTER'S NAME",
  "armor": "ONE PIECE OF ARMOR",
  "weapon": "ONE OR MORE WEAPONS"
}
"""
one_shot_prompt = [
    {"role": "user", "content": one_shot_template}
]

# Generate the output
outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
```

```
 {
  "description": "A cunning rogue with a mysterious past, skilled in stealth and
  "name": "Lysandra Shadowstep",
  "armor": "Leather Cloak of the Night",
  "weapon": "Dagger of Whispers, Throwing Knives"
}
```

Grammar: Constrained Sampling

```
import gc
import torch
del model, tokenizer, pipe

# Flush memory
gc.collect()
torch.cuda.empty_cache()
```

```
from llama_cpp.llama import Llama

# Load Phi-3
llm = Llama.from_pretrained(
    repo_id="microsoft/Phi-3-mini-4k-instruct-gguf",
    filename="*fp16.gguf",
    n_gpu_layers=-1,
    n_ctx=2048,
    verbose=False
)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: User
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access publ
  warnings.warn(
```

```
# Generate output
output = llm.create_chat_completion(
    messages=[
        {"role": "user", "content": "Create a warrior for an RPG in JSON format
    ],
    response_format={"type": "json_object"},
    temperature=0,
)['choices'][0]['message']["content"]
```

```
import json

# Format as json
json_output = json.dumps(json.loads(output), indent=4)
print(json_output)
```

```
{
    "name": "Eldrin Stormbringer",
    "class": "Ranger",
    "level": 5,
```

```
        "attributes": {
            "strength": 14,
            "dexterity": 18,
            "constitution": 12,
            "intelligence": 10,
            "wisdom": 13,
            "charisma": 9
        },
        "skills": {
            "archery": {
                "proficiency": 20,
                "critical_hit_chance": 5,
                "damage_range": [
                    8,
                    14
                ]
            },
            "stealth": {
                "proficiency": 17,
                "critical_hit_chance": 3,
                "damage_range": [
                    2,
                    6
                ]
            },
            "nature_magic": {
                "proficiency": 15,
                "critical_hit_chance": 4,
                "healing_range": [
                    3,
                    7
                ],
                "damage_range": [
                    -2,
                    2
                ]
            }
        },
        "equipment": {
            "weapons": [
                "Longbow",
                "Dagger"
            ],
            "armor": "Leather Armor",
            "accessories": [
                "Boots of Speed",
                "Ring of Protection"
            ]
        },
        "background": "Eldrin grew up in the wilderness, learning to hunt and track
    }
```

```
Start coding or generate with AI.
```