intro_prompt_design

October 9, 2023

[ ]:
```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

1 Prompt Design - Best Practices

Run in Colab

View on GitHub

Open in Vertex AI Workbench

1.1 Overview

This notebook covers the essentials of prompt engineering, including some best practices.

Learn more about prompt design in the official documentation.

1.1.1 Objective

In this notebook, you learn best practices around prompt engineering – how to design prompts to improve the quality of your responses.

This notebook covers the following best practices for prompt engineering:

- Be concise
- Be specific and well-defined
- Ask one task at a time
- Turn generative tasks into classification tasks
- Improve response quality by including examples

1.1.2 Costs

This tutorial uses billable components of Google Cloud:

- Vertex AI Generative AI Studio

Learn about Vertex AI pricing, and use the Pricing Calculator to generate a cost estimate based on your projected usage.

1.1.3 Install Vertex AI SDK

[ ]:
```
!pip install "shapely<2.0.0"
!pip install google-cloud-aiplatform --upgrade --user
```

Colab only: Uncomment the following cell to restart the kernel or use the button to restart the kernel. For Vertex AI Workbench you can restart the terminal using the button on top.

[ ]:
```
# # Automatically restart kernel after installs so that your environment can␣
 ↪access the new packages
# import IPython

# app = IPython.Application.instance()
# app.kernel.do_shutdown(True)
```

1.1.4 Authenticating your notebook environment

- If you are using **Colab** to run this notebook, uncomment the cell below and continue.
- If you are using **Vertex AI Workbench**, check out the setup instructions here.

[ ]:
```
# from google.colab import auth
# auth.authenticate_user()
```

1.1.5 Import libraries

Colab only: Uncomment the following cell to initialize the Vertex AI SDK. For Vertex AI Workbench, you don't need to run this.

[ ]:
```
# import vertexai

# PROJECT_ID = "[your-project-id]"  # @param {type:"string"}
# vertexai.init(project=PROJECT_ID, location="us-central1")
```

[ ]:
```
from vertexai.language_models import TextGenerationModel
```

1.1.6 Load model

[ ]:
```
generation_model = TextGenerationModel.from_pretrained("text-bison@001")
```

1.2 Prompt engineering best practices

Prompt engineering is all about how to design your prompts so that the response is what you were indeed hoping to see.

The idea of using "unfancy" prompts is to minimize the noise in your prompt to reduce the possibility of the LLM misinterpreting the intent of the prompt. Below are a few guidelines on how to engineer "unfancy" prompts.

In this section, you'll cover the following best practices when engineering prompts:

- Be concise
- Be specific, and well-defined
- Ask one task at a time
- Improve response quality by including examples
- Turn generative tasks to classification tasks to improve safety

1.2.1 Be concise

Not recommended. The prompt below is unnecessarily verbose.

```
[ ]: prompt = "What do you think could be a good name for a flower shop that␣
      ↪specializes in selling bouquets of dried flowers more than fresh flowers?␣
      ↪Thank you!"

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Recommended. The prompt below is to the point and concise.

```
[ ]: prompt = "Suggest a name for a flower shop that sells bouquets of dried flowers"

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

1.2.2 Be specific, and well-defined

Suppose that you want to brainstorm creative ways to describe Earth.

Not recommended. The prompt below is too generic.

```
[ ]: prompt = "Tell me about Earth"

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Recommended. The prompt below is specific and well-defined.

```
[ ]: prompt = "Generate a list of ways that makes Earth unique compared to other␣
      ↪planets"

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

1.2.3 Ask one task at a time

Not recommended. The prompt below has two parts to the question that could be asked separately.

[ ]:
```
prompt = "What's the best method of boiling water and why is the sky blue?"

print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Recommended. The prompts below asks one task a time.

[ ]:
```
prompt = "What's the best method of boiling water?"

print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

[ ]:
```
prompt = "Why is the sky blue?"

print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

1.2.4 Watch out for hallucinations

Although LLMs have been trained on a large amount of data, they can generate text containing statements not grounded in truth or reality; these responses from the LLM are often referred to as "hallucinations" due to their limited memorization capabilities. Note that simply prompting the LLM to provide a citation isn't a fix to this problem, as there are instances of LLMs providing false or inaccurate citations. Dealing with hallucinations is a fundamental challenge of LLMs and an ongoing research area, so it is important to be cognizant that LLMs may seem to give you confident, correct-sounding statements that are in fact incorrect.

Note that if you intend to use LLMs for the creative use cases, hallucinating could actually be quite useful.

Try the prompt like the one below repeatedly. You may notice that sometimes it will confidently, but inaccurately, say "The first elephant to visit the moon was Luna".

[ ]:
```
prompt = "Who was the first elephant to visit the moon?"

print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

1.2.5 Turn generative tasks into classification tasks to reduce output variability

Generative tasks lead to higher output variability The prompt below results in an open-ended response, useful for brainstorming, but response is highly variable.

[ ]:
```
prompt = "I'm a high school student. Recommend me a programming activity to␣
↪improve my skills."

print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Classification tasks reduces output variability The prompt below results in a choice and may be useful if you want the output to be easier to control.

```
[ ]: prompt = """I'm a high school student. Which of these activities do you suggest␣
      ↪and why:
     a) learn Python
     b) learn Javascript
     c) learn Fortran
     """

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

1.2.6 Improve response quality by including examples

Another way to improve response quality is to add examples in your prompt. The LLM learns in-context from the examples on how to respond. Typically, one to five examples (shots) are enough to improve the quality of responses. Including too many examples can cause the model to over-fit the data and reduce the quality of responses.

Similar to classical model training, the quality and distribution of the examples is very important. Pick examples that are representative of the scenarios that you need the model to learn, and keep the distribution of the examples (e.g. number of examples per class in the case of classification) aligned with your actual distribution.

Zero-shot prompt Below is an example of zero-shot prompting, where you don't provide any examples to the LLM within the prompt itself.

```
[ ]: prompt = """Decide whether a Tweet's sentiment is positive, neutral, or␣
      ↪negative.

     Tweet: I loved the new YouTube video you made!
     Sentiment:
     """

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

One-shot prompt Below is an example of one-shot prompting, where you provide one example to the LLM within the prompt to give some guidance on what type of response you want.

```
[ ]: prompt = """Decide whether a Tweet's sentiment is positive, neutral, or␣
      ↪negative.

     Tweet: I loved the new YouTube video you made!
     Sentiment: positive

     Tweet: That was awful. Super boring
     Sentiment:
     """

     print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Few-shot prompt Below is an example of few-shot prompting, where you provide one example to the LLM within the prompt to give some guidance on what type of response you want.

```
[ ]: prompt = """Decide whether a Tweet's sentiment is positive, neutral, or␣
      ↪negative.

    Tweet: I loved the new YouTube video you made!
    Sentiment: positive

    Tweet: That was awful. Super boring
    Sentiment: negative

    Tweet: Something surprised me about this video - it was actually original. It␣
      ↪was not the same old recycled stuff that I always see. Watch it - you will␣
      ↪not regret it.
    Sentiment:
    """

    print(generation_model.predict(prompt=prompt, max_output_tokens=256).text)
```

Choosing between zero-shot, one-shot, few-shot prompting methods Which prompt technique to use will solely depends on your goal. The zero-shot prompts are more open-ended and can give you creative answers, while one-shot and few-shot prompts teach the model how to behave so you can get more predictable answers that are consistent with the examples provided.