Using Large Language Models (LLMs)

Shea Fyffe and Sebastian Marin

© 2024 by Shea Fyffe and Sebastian Marin. All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, without the prior written permission of the authors.

Table of Contents

A Brief Introduction to LLM Concepts	1
Language Modeling	3
Adapting an LLM	4
In-Context Learning	4
Tips for In-Context Learning	5
Supervised Fine-Tuning	7
Model Alignment	8
More about Using LLMs	8
Barriers to Using LLMs	9
Tutorial	10
Mistral	10
Mistral: Setup	10
Mistral: Basic Usage	11
Mistral: Chat (Instruct) Basic Usage	13
Conclusion	15
References	16

This guide aims to enhance your understanding and use of large language models (LLMs) like ChatGPT, GPT-4, Claude, and Gemini. These models offer advanced natural language processing capabilities to the public through easy-to-use APIs and interfaces, enabling tasks such as code generation, report writing, article summarization, and idea brainstorming. However, the widespread use of LLMs also underscores the need for a solid grasp of their nature and functioning. We'll cover the basics of LLMs, their training processes, and the significance of this knowledge for users. Additionally, we'll show how to access, prompt, and derive results from LLMs, catering to all user levels.

A Brief Introduction to LLM Concepts

Those just starting to learn about LLMs, NLP, and AI more broadly may realize that computer scientists and computational linguists have a lot in common with psychologists in that they both like to over-engineer things. So, it can be overwhelming wrapping your head around the various concepts and terminology related to LLMs; for example, this tutorial focuses on one particular type of model—autoregressive decoders with a transformer architecture. However, find comfort in knowing that you can simply call these models, and any of today's popular models, large language models. For those interested in navigating the labyrinth of terms, we've provided a table below.

 Table 1. Terms Related to Large Language Models

Term	Description	Relationship	Common Usage
Large Language Model (LLM)	A type of artificial intelligence model that is designed to represent human language and perform various natural language tasks, such as text generation and text classification.	Base concept for all the other terms.	This is the broadest term and can be used to describe any system that generates human-like text, including all the specific models and techniques listed below.
Transformers	A type of neural network architecture in machine learning made up of components like self-attention mechanisms, encoders, and decoders.	Base architecture for BERT and GPT.	Transformers are a specific type of model architecture that both BERT and GPT are based on.
Encoder	Part of a neural network that compresses the input into a smaller, dense representation.	Used in transformer architecture and specifically in BERT.	Encoders are used in models that need to understand context from both past and future input, like BERT.
Decoder	Part of a neural network that translates the dense representation into the desired output.	Used in transformer architecture and specifically in GPT.	Decoders are used in models that generate output based on the previous inputs only, like GPT.
Autoregressive	A property of models where the prediction for a certain step is based on the preceding steps.	Property of GPT.	Autoregressive models like GPT generate text word by word, always based on the previously generated words.
Causal Language Modeling	A task where the model must predict the next word in a sentence, given all the previous words.	Task performed by autoregressive models like GPT.	This is a specific type of language modeling task that autoregressive models like GPT are good at.
Masked Language Modeling	A task where the model must predict a missing word in a sentence, given the context.	Task performed by BERT.	This is a specific type of language modeling task that BERT is designed for.
BERT (Bidirectional Encoder Representations from Transformers)	A transformer-based machine learning technique for natural language processing pre-training.	Uses encoders and performs masked language modeling.	BERT is designed to understand the context from both past and future input and is pre-trained using a masked language modeling task.
GPT (Generative Pretrained Transformer)	An autoregressive language model that uses deep learning to produce human-like text.	Uses decoders and performs causal language modeling.	GPT is designed to generate text based on the previous inputs only and is pre-trained using a causal language modeling task.

Language Modeling

Before diving into how to access and interact with LLMs, it is useful to discuss how they learn to generate language. LLMs get their name from how they are (pre)trained; that is, from an NLP task known as *language modeling*. In language modeling, a model attempts to predict a "masked" token or word given a sequence of text. Hence, we could input a sequence of text into the model and mask (or hide) a word in the sequence—generally, this word is selected at random. Since we know the masked word the LLM is trying to predict, we can provide the model with feedback on its predictions. Through this feedback process, the model updates its representations of words and how they relate to one another. In summary, LLMs begin to "understand" language and the complex patterns between words as a byproduct of language modeling.

While there are several approaches to language modeling, causal (i.e., autoregressive) language modeling is at the heart of today's most popular models. In causal language modeling, LLMs learn to predict the next word in a sequence of text based on all the previous words. Models trained via causal language modeling—for example—GPT-4, Gemini, and Llama are primarily concerned with generating or "decoding" output sequences (e.g., words, sentences, and paragraphs) given some input words or context. Thus, these models are considered autoregressive because the words generated by the model depend on the previous words.

To illustrate the causal or autoregressive nature of an LLM, we could input the first few words of the sentence "IO psychology focuses on understanding the behavior of individuals and groups in the workplace" (taken from APA, 2022) into a model:

User Input:

IO psychology focuses on understanding the

. . .

Model Output:

... interaction of an individual, the organization and the environment in which work takes place.

Now let's examine the result when we remove an important context word such as "psychology":

User Input:

IO focuses on understanding the ...

Model Output:

 \dots impact of investments in transportation infrastructure on the environment, the economy, and quality of life.

In the first example text, the model generated is close to APA's actual definition of I/O Psychology. So, you could say we've given the model a sufficient amount of context. In contrast, after removing a single, but very important, context word the model generates text that is quite different from what we expect. This shows how highly dependent an LLM's output is on the words appearing in the context or prompt it is given.

Moreover, these examples have underscored an important characteristic of an LLM—that is, LLMs are probabilistic. So, these models do not "look up" what words to output based on matches they find in their training data. One important implication of this is that LLMs will often

generate text that seems factually correct but is just made up (Bender et al., 2021). Here is a good SIOP-related example:

User Input:

The Society for Industrial and Organizational Psychology (SIOP) held its first annual conference on April 9-11, 1986 in Chicago, Illinois.

If someone approached you in the current year, and said "I'm doing a scavenger hunt! Please help. Where did SIOP host its 2nd annual conference?" What would you say?

Model Output:

The 2nd annual SIOP conference was held in Dallas, Texas. I hope this helps with your scavenger hunt!

SIOP's 2nd Annual Conference was held in Atlanta, Georgia, though Dallas Texas is convincing. So, two important things to remember:

! Important

- LLMs are probabilistic, so they "guess" what words to generate next based on the information stored in their architecture and any additional context given to them during model inference.
- LLMs struggle to perform tasks where there is little room for guessing, for instance, retrieving specific data points and facts, and identifying historical events. (Gao et al., 2024)

Adapting an LLM

There are many ways to *adapt* an LLM.¹ The next section will elaborate upon several popular areas of adaptation—that is, in-context learning (ICL), supervised fine-tuning, and model alignment. Each of these areas comes with their own specific techniques—for instance—direct preference optimization and reinforcement learning from human feedback are specific approaches to model alignment.

Each approach can be framed as the extent to which they change an LLM's pre-trained architecture and how they go about changing such architecture. In-context learning—for example—makes no changes to the pre-trained parameters of an LLM. Instead, ICL uses a model's pre-trained knowledge and information embedded in the prompt (e.g., context, examples, and instructions) to perform a task. In contrast, supervised fine-tuning requires researchers and practitioners to collect examples of model inputs and the outputs expected to be generated from these prompt inputs. The model is then "fine-tuned" to recreate this data, and it updates its parameters as a result.

In-Context Learning

In many cases, an LLM adapted via ICL will perform well (Brown et al., 2020). However, if a person wants to perform a task that significantly differs from the tasks performed during pre-training, it is likely a more involved method of adaptation is needed (Zhang et al., 2023).

¹ Here, we use *adapt* to include the broad spectrum of techniques one can use to modify an LLM after pre-training.

Still, ICL gets its strength from how undemanding and straightforward it is, especially since today's LLMs have been pre-trained across a large number of NLP tasks and a wide spectrum of language domains. In terms of its limitations, ICL restricts a person's ability to guide model behavior. To elaborate, when performing ICL researchers and practitioners can only guide the model via the prompt. While the prompt is extremely flexible, even small changes to the prompt can lead to meaningful changes in the output; additionally, designing effective prompts can take time.

Designing an effective prompt is both an art and a science; luckily, there are plenty of resources related to this activity (sometimes called *prompt engineering*). To list a few:

- Awesome GPT Prompt Engineering
- DAIR Al's Prompt Engineering Guide
- Mihail Eric's Complete Introduction to Prompt Engineering
- OpenAl's 6 Strategies for Getting Better Results

Tips for In-Context Learning

You will often find that recommendations for prompt engineering end up being too broad to be useful (e.g., "it depends on your task", "be specific", "add context"). In hopes of avoiding this, we have come up with several more substantive guidelines (with the disclaimer that these may not equally apply to every use case).

- **Understand the parts of a prompt.** Prompts do not just consist of an *instruction*, that is, the specific task you want the model to perform or the question you want it to answer. Prompts can also include additional *context*, *examples*, *constraints*, *soft prompts*, and so forth. It is good to know when it is beneficial to add parts to the prompt. For instance, examples can be useful when you are looking for a fixed output. Adding a role as context, for example, "You are a researcher", is beneficial when you want any output of a certain style.
- Put the important stuff at the beginning or end. When writing a longer prompt, models tend to optimize themselves by paying less attention to the middle parts of the prompt and more attention to the beginning and end of the prompt.
- When embedding examples in the prompt, use very different examples. A common mistake that people make when including examples in their prompts, is that these examples are often selected arbitrarily or based on similarity. Given the limited space you have in a prompt, you should ensure your examples represent a broad spectrum of possibilities (Wang et al., 2022). If you are using an LLM to write job postings, for example, you should use examples from very different jobs. That is unless you want your model to produce very redundant outputs.
- When using techniques like few-shot learning for a generation task, use a flexible soft prompt. A soft prompt is a variable part of your prompt that can be changed to modify the content of the output. Let's say we want to use an LLM to generate Likert statements for an assessment we are creating. We could use a soft prompt to control the construct: and number: of items the model is generating:

```
Generate Likert items based upon two tags. The `count:` tag will indicate
the number of items to generate and the `construct:` tag will the construct
or trait the items measure. For example:
construct: "emotional stability"
count: 4
items:
1. I am seldom blue.
2. I hold in my emotions.
3. When I am driving, I often find myself angry with other motorists.
4. My friends would describe me as anxious.
###
construct: "extraversion"
count: 2
items:
1. I am okay with keeping in the background.
2. I am the life of the party.
construct: "agreeableness"
count: 7
```

The issue with the soft prompt above is that it only allows for varying outputs based on number and construct; thus, one would have fewer ways to vary the input and quickly run into issues generating redundant items. To overcome this, additional dimensions could be added to the soft prompt (e.g., context: work, keyed: positive, construct level: high, etc).

• When you cannot provide examples, it's generally a good idea to add specific constraints. LLMs are commonly given instructions such as summarize ..., rephrase ..., or rewrite When performing these types of tasks coming up with examples can be a waste of time, as it would take the same amount of time to just do the original task yourself as it would to create examples. So, adding constraints after giving the model instructions is a viable alternative. For instance:

```
Rewrite the text below. Change at least 70% of the words when rewriting the text, while keeping the overall meaning the same. The rewritten text should be longer than the original text.

Text: "..."
Rewritten text:
```

Now, it is common to interact with LLMs via a chat interface. Notably, the chat interface is very similar to the standard "completion" based interface, though the chat interface appends special tokens to the prompt (e.g., user, assistant, etc). Additionally, many LLMs with a chat interface use a system message (e.g., ChatGPT, GPT-4). On the backend, the system message is prefixed to the rest of the conversation (i.e., the user and assistant messages) before being passed to the LLM as the prompt. As you chat with the LLM, the system message remains prefixed to the prompt. When chat conversations get too long, the text from the oldest user messages is removed. So, this can lead to issues in cases where you have input important contextual information into the first or second user messages. With that in mind, know that the information and context that you want the model to remember throughout your interaction should be placed in the system message.

• Modify your instructions to align with a model's pre-training tasks. As mentioned, LLMs do best when they are performing a task they have performed before. So, it is very useful to know a bit about the data used to pre-train the model. In this RLHF dataset—for instance—you can see that the instructions for the model do *not* begin with "please" or "hello" etc. So, prompts like Please tell me about X or Can you please summarize this text? are not as effective as Tell me about X or Summarize this text.

Supervised Fine-Tuning

To perform supervised fine-tuning, one needs to collect a sufficient amount of training examples. In addition to the input prompt, training examples must also include outputs (or completions) corresponding to each input prompt.

If we would like to use supervised fine-tuning to write source code given a description and a coding language—for instance—our training examples could look something like this:

Table 2. Example Training Data for Fine-Tuning and LLM to Generate Source Code

Prompt	Completion (Output)
Write a function to flag if a number is even in R. Source Code:	<pre>is_even <- function(number) { number %% 2 == 0L }</pre>
Write a function to flatten a nested list in Python. Source Code:	def flatten_list(1): return [12i for 11 in 1 for 12i in 11]
Write a function that reverses a string in Rust. Source Code:	<pre>fn rev_string(input: &str) -> String { input.chars().rev().collect() }</pre>

While collecting training examples to fine-tune a model can be tedious, if you have enough data it will most likely result in better performance on your particular task when compared to ICL (e.g., Liu et al., 2022; Mosbach et al., 2023). Nonetheless, there are additional reasons why one should fine-tune instead of using ICL and vice versa.

Advantages of Fine-Tuning

- Generally higher performance when compared to ICL
- Ideal for performing a single specialized task numerous times
- Very cost- and time-efficient when done correctly
- Produces a model that is more consistent across people and environments
- Allows for ownership over your LLM

Disadvantages of Fine-Tuning

- Training examples are tedious to collect
- Can lead to overfitting and models that produce redundant outputs, even when prompts differ
- Is higher-stakes in terms of cost when compared to ICL
- Requires more expertise and effort upfront—if you are not careful you can end up fine-tuning a model that is useless

Model Alignment

Although we will not discuss model alignment techniques in depth, these methods are worth mentioning; especially their mercurial rise in the world of NLP. Today, model alignment techniques are the state—of—the—art approach to LLM adaptation. We can thank this family of methods for producing today's most popular models, like *ChatGPT*, *Gemini*, *GPT-4*, and *Mistral*.

Like supervised fine-tuning, model alignment techniques require the collection of training examples. However, in addition to training examples, reward or rating data is collected. Generally, these "reward" scores are produced after ranking training examples from similar prompts in terms of quality; whereby, the highest quality outputs get the highest reward scores. Then, after calculating reward scores, the training examples and reward scores are used to further train the model. The primary differences between model alignment techniques are in how an LLM learns from reward scores. In RLHF, for instance, an additional "reward model" is trained using the training examples and reward scores; then, the LLM is trained via feedback received from the reward model. On the other hand, methods like DPO integrate the reward function into the LLM itself.

While those interested in model alignment techniques should visit some of the additional resources and links provided throughout this tutorial,² a straightforward takeaway from the section above should be that model alignment provides a mechanism by which an LLM can learn to discern the quality of training examples. In contrast to well-established adaptation methods (e.g., ICL and supervised fine-tuning)—where training examples are treated equally during training—model alignment allows an LLM to learn most from examples that humans prefer. In the next section, we provide a tutorial for accessing and using LLMs. Specifically, we will present several examples with *Mistral*—the current state—of—the—art open-source LLM.

More about Using LLMs

Before diving into the examples, it is helpful to introduce the ways you could interact with an LLM:

- 1. **Pre-training:** Here, you take an LLM with randomly initialized weights, and perform source tasks, such as language modeling or next sentence prediction, to teach an LLM about the basic relationships between words and tokens. Source tasks are usually performed millions of times (Azunre, 2021) across large collections of natural language text (e.g., webpages, digitized books, and journal articles).
- 2. **Training:** Any supervised technique, for example, supervised fine-tuning or model alignment, that updates the parameters of a pre-trained LLM to better align with a specific target task. This requires providing the model with (labeled) training examples.
- 3. **Inference:** Using a trained model to generate output by inputting data in real time. Today, most interactions with LLMs are considered model inference; for instance, conversations with chat-based LLMs like ChatGPT or Bard. Also, ICL methods (e.g.,

² This is an oversimplification of model alignment and its numerous approaches (e.g., direct preference optimization [DPO]; Identity Preference Optimisation [IPO]; Kahneman-Tversky Optimisation [KTO]; and Reinforcement Learning with Human Feedback [RLHF]). Even so, the salient difference between model alignment techniques and supervised fine-tuning is the inclusion of preference or reward data, which provides the model with additional information during training. For those wanting to learn more about model alignment techniques, we recommend visiting Huggingface's Blog and browsing the posts related to model alignment.

zero- and few-shot learning) can be considered a special form of model inference (Dong et al., 2023).

Barriers to Using LLMs

We must consider the barriers to each type of interaction. In terms of hardware, LLMs (especially decoder-only architectures) almost always require the use of one or multiple GPUs. While researchers are continually developing ways to lessen the demand for GPU compute (e.g., Hu et al., 2023; Zhang et al., 2021), it is still quite a high bar for someone just getting started with LLMs. Pre-training demands the most GPU compute, by far (e.g., Meta's LLM *Llama* used roughly 3.6 million USD worth of GPU hardware during pre-training). Training the larger LLMs can be performed on one's personal computer; however, it would require purchasing \$3,000 dollars worth of GPU hardware (i.e., two Nvidia GeForce RTX 3090s). In contrast, model inference can be performed on a personal computer for one-sixth of the price it would cost to train a model.

For these reasons, most people do not use their personal computers to interact with LLMs. Instead, it is more common to use popular cloud environments, such as Google Colab, Microsoft Azure, or Amazon AWS EC2, where you can rent GPU time.³ Beyond cost, the major trade-offs between these platforms are: (a) GPU-tier, some services provide access to better GPUs; (b) usability, some platforms are easier to navigate and access; and (c) data-integration, services like Google Colab integrate with Google Drive which makes it easy to import and export data.

In terms of software, it helps to know Python, though closed-source LLMs like ChatGPT and Bard can be used by those without a background in programming. However, if you are planning on performing model inference repeatedly, it is a wise decision to interact with the model programmatically.

We demonstrate this programmatic interaction in the tutorial below.

³ For those interested in a longer term investment, we recommend Google Colab's Pro Subscription (~\$10/month) or the AWS Cloud for Researchers and Students.

Tutorial

Next, we provide code for the basic usage of Mistral.

Mistral

Mistral is an open-source LLM created by MistralAI. Currently, its mixture of experts counterpart—Mixtral—stands as the top performing open-source LLM on Huggingface's Chatbot Arena. Mixtral uses a mixture of experts, meaning before generating an output, the model chooses to route the input through two of eight possible networks—each network specializes in certain aspects of language. Mixtral combines the results from two of these networks and then generates the output text.

Mistral: Setup

Mistral, in addition to many other open-source models, can be accessed via Huggingface's transformers Python library. In Google Colab, we can install the transformers library.

```
! pip install -U transformers
```

Now we can import the objects from transformers that we will use (e.g., AutoModelForCausalLM and AutoTokenizer). In addition, we can use the dataclasses library to construct a data class that will help us initialize and better interact with the model.⁴ When initializing the model we can select among four versions of Mistral:

- 1. The base model (the default), which is a standard language model that generates output given an input prompt: Mistral()
- 2. The "Instruct" model, which uses a chat-based interface: Mistral(instruct model = True)
- The base model with a mixture of experts: Mistral(mixture of experts = True)
- 4. The "Instruct" model with a mixture of experts, which uses a chat-based interface:

 Mistral(instruct_model = True, mixture_of_experts = True)

```
# import objects/modules
from transformers import (AutoModelForCausalLM, AutoTokenizer) #relevant objects from
transformers
from dataclasses import (dataclass, field, fields) #used to create dataclass that
will hold Mistral model
from typing import (Any, Optional, Union) #used for type-hinting
import torch #for gpu ram cleanup and misc
import gc #for memory cleanup
```

```
@dataclass
class Mistral:
    # init arguments
    instruct_model: bool = False
    mixture_of_experts: bool = False
    model_kwargs: dict[str, Any] = field(default_factory=dict)
    tokenizer_kwargs: dict[str, Any] = field(default_factory=dict)
    pipeline_init_kwargs: dict[str, Any] = field(default_factory=dict)
```

⁴ This is not required but would be helpful for those wanting a more reproducible workflow.

```
def __post_init__(self):
        # clear gpu memory
        if torch.cuda.is available():
            torch.cuda.empty cache()
            gc.collect()
        # try to download model tokenizer
            self.tokenizer = AutoTokenizer.from pretrained(self.model path,
**self.tokenizer kwargs)
        except Exception as tokenizer init err:
            if hasattr(tokenizer init err, 'add note'):
                tokenizer_init_err.add_note("If tokenizer_kwargs were defined, please
ensure they are correct.")
            raise
        # try to download model
            self.model = AutoModelForCausalLM.from pretrained(self.model path,
**self.model kwargs)
        except Exception as model init err:
            if hasattr(model init err, 'add note'):
                model_init_err.add_note("Model may be too large for GPU being used.")
                model_init_err.add_note("If model_kwargs were defined, please ensure
they are correct.")
            raise
        # add default arguments to pipeline if not defined in init
        self.pipeline init kwargs = self.pipeline init kwargs | {"torch dtype":
torch.float16, "load_in_4bit": True, "device_map": "auto"}
        # setup chat pipeline for inference
        self.chat = pipeline(task = "conversational" if self.instruct_model else
"text-generation", model = self.model, tokenizer = self.tokenizer, model_kwargs =
self.pipeline_init_kwargs)
    @property
    def model path(self) -> str:
        return 'mistralai/%s%sv0.1' % ("Mixtral-8x7B-" if self.mixture_of_experts
else "Mistral-7B-", "Instruct-" if self.instruct model else "")
```

The model requires quite a bit of GPU RAM (i.e., ~ 15GB), so you will likely need an upgraded GPU, for instance, an A100 or V100.

Mistral: Basic Usage

Now we can instantiate our object and generate some text!

```
# instantiate the standard llm
my_llm = Mistral()
```

We will just examine a basic prompt that lists out 10 jobs that the LLM thinks will be automated by Al.⁵ We can use the model's chat method to perform this inference:

⁵ For more advanced prompts, I recommend visiting Huggingface's LLM Prompting Guide for more advanced examples.

User Input:

result = my_llm.chat("List 10 jobs that are likely to be replaced by AI in the next
10 years. ### 1.", max_new_tokens = 450)
result[0]["generated text"]

Model Output:

List 10 jobs that are likely to be replaced by AI in the next 10 years. ### 1. Data Entry Clerk

Data entry clerks are responsible for entering data into computer systems. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

2. Customer Service Representative

Customer service representatives are responsible for answering customer questions and resolving customer complaints. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

3. Cashier

Cashiers are responsible for ringing up customer purchases and handling customer payments. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

4. Receptionist

Receptionists are responsible for answering phones, greeting visitors, and handling other administrative tasks. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

5. Accountant

Accountants are responsible for preparing financial statements, tax returns, and other financial documents. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

6. Paralegal

Paralegals are responsible for assisting lawyers with legal research, drafting legal documents, and other legal tasks. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

7. Secretary

Secretaries are responsible for answering phones, scheduling appointments, and other administrative tasks. This job is likely to be replaced by AI in the next 10 years because AI can do this job faster and more accurately than humans.

8 ...

Mistral: Chat (Instruct) Basic Usage

Let's turn to using the chat-style "Instruct" model. To initialize the new model we can set the instruct_model = True.

```
# instantiate the instruct/chat llm
chat_llm = Mistral(instruct_model=True)
```

The input data for the instruct model is slightly more complex—each message is held within a Python dictionary with the form {'role': ..., 'content': ...} where the 'role' value denotes who generated the message (e.g., the user or the LLM assistant) and the 'content' indicates the message itself. For instance:

```
messages = [{"role": "user", "content": "What is your favorite movie?"}, {"role":
    "assistant", "content": "As an AI assistant I don't watch movies."}]
```

The code above would be equivalent to you (i.e., the "user") asking the LLM their favorite movie, and the LLM responding with "As an AI assistant I don't watch movies." Each message dictionary is stored within a list (note the []) to keep track of the overall conversation history. So the code below shows how we could continue the conversation by asking the AI "Okay, then what do you think my favorite movie is?" The messages object can then be passed to the chat() method to generate a response.

```
# create a list of messages
messages = [{"role": "user", "content": "What is your favorite movie?"}, {"role":
"assistant", "content": "As an AI assistant I don't watch movies."}, {"role": "user",
"content": "Okay, then what do you think my favorite movie is?"}]
# generate the next message in the chat
chat_llm.chat(messages)
```

One issue with the Mistral instruct model is that it does not intuitively allow for the use of a system prompt. A system prompt could be useful when—for example—we wanted the model to generate messages with consistent structure and content. One valuable use case in O/I could be using the model to generate behaviorally anchored rating scales (BARs) for behavioral interview questions. The code below shows how we could ask the model to perform this task with the use of a system prompt.

```
bars messages = [
    {"role": "system", "content": "You are an expert in Human Resources and
Industrial/Organizational Psychology. Your task is to generate behaviorally anchored
rating scales (BARs) with five behavioral anchors. Each BAR will score respondents'
answers to a corresponding behavioral interview question. Behavioral anchor 1 will
list behaviors that will result in the poorest performance and behavior anchor 5 will
list behaviors that will result in the best performance. For example: '''Interview
Question: Tell me about a time you dealt with a contentious client or customer.
Behavioral Anchors: 1 - Dismissed client or customer's perspective, Demeaned or
devalued client or customer's attributes, Mocked client or customer; 2 - Acknowledged
client or customer's perspective without following through, Purposely skewed client
or customer's request when describing it to others; 3 - Passed client or customer off
to another coworker, Partially addressed client or customer's request, Did little to
verify perspective of client or customer; 4 - Collaborated with others to solve the
issue, Mostly addressed client or customer's request, Did standard amount to verify
perspective of client or customer; 5 - Solved issue independently, Fully addressed
client or customer's request, Was proactive when verifying perspective of client or
customer'"}, {"role": "user", "content": "'''Interview Question: Tell me about a time
you communicated a technical topic to a non-technical audience. Behavioral
```

```
Anchors:"},
]
```

Since Mistral doesn't use a system prompt, we can inject one ourselves using a function like the one below:

```
def format_chat_prompt_mistral(messages: list[dict[str, str]]) -> list[dict[str,
str]]:
    sys_prompt = ""
    new_messages = []
    for m in messages:
        if m['role'] == "system":
            sys_prompt += m['content']
            continue
        else:
            new_messages.append(m)
    if sys_prompt != "":
        sys_prompt = "[SYS]" + sys_prompt + "[/SYS]\n"
        new_messages[0] = {"role": "user", "content": sys_prompt +
        new_messages[0]['content']}
    return new_messages
```

Now let's combine things to see if the model can generate BARs.

```
User Input:
  result = chat_llm.chat(format_chat_prompt_mistral(bars_messages))
  print(result)
```

Model Output:

Conversation id: eae65bfb-c702-4b59-bf53-d072575ea71f

user: [SYS]You are an expert in Human Resources and Industrial/Organizational Psychology. Your task is to generate behaviorally anchored rating scales (BARs) with five behavioral anchors. Each BAR will score respondents' answers to a corresponding behavioral interview question. Behavioral anchor 1 will list behaviors that will result in the poorest performance and behavior anchor 5 will list behaviors that will result in the best performance. For example: '''Interview Question: Tell me about a time you dealt with a contentious client or customer. Behavioral Anchors: 1 - Dismissed client or customer's perspective, Demeaned or devalued client or customer's attributes, Mocked client or customer; 2 -Acknowledged client or customer's perspective without following through, Purposely skewed client or customer's request when describing it to others; 3 - Passed client or customer off to another coworker, Partially addressed client or customer's request, Did little to verify perspective of client or customer; 4 - Collaborated with others to solve the issue, Mostly addressed client or customer's request, Did standard amount to verify perspective of client or customer; 5 - Solved issue independently, Fully addressed client or customer's request, Was proactive when verifying perspective of client or customer'[/SYS]

- '''Interview Question: Tell me about a time you communicated a technical topic to a non-technical audience. Behavioral Anchors:
- assistant: 1 Misunderstood technical concepts and failed to convey them clearly, Used jargon and technical language without explanation, Assumed audience had prior knowledge of technical topic
- 2 Simplified technical concepts but lost accuracy, Used analogies and examples to explain technical concepts, Provided general information without specifics
- 3 Provided basic explanations but lacked detail, Used visual aids and diagrams to explain technical concepts, Provided examples to illustrate technical concepts
- 4 Provided clear and concise explanations, Used analogies and examples to explain technical concepts, Provided examples to illustrate technical concepts
- 5 Provided in-depth explanations, Used visual aids and diagrams to explain technical concepts, Provided examples and case studies to illustrate technical concepts

The results are pretty impressive. Additionally, more BARs could be generated by adding new interview questions to the messages (found in the results object) and then calling the chat method again.

Conclusion

We hope the content provided here in addition to the links provided will help those looking to improve their skills prompting LLMs!

Please feel free to send an email to <u>Shea Fyffe</u> or <u>Sebastian Marin</u> with questions, comments, or ideas

References

- APA. (2022, May). *Industrial and Organizational Psychology*. Industrial and Organizational Psychology. https://www.apa.org/ed/graduate/specialize/industrial
- Azunre, P. (2021). Transfer learning for natural language processing. Manning Publications Co.
- Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the Dangers of Stochastic Parrots: Can Language Models Be Too Big? . *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, 610–623. https://doi.org/10.1145/3442188.3445922
- Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). Language models are few-shot learners. *arXiv:2005.14165* [Cs]. http://arxiv.org/abs/2005.14165
- Dong, Q., Li, L., Dai, D., Zheng, C., Wu, Z., Chang, B., Sun, X., Xu, J., Li, L., & Sui, Z. (2023). *A Survey on In-context Learning* (arXiv:2301.00234). arXiv. http://arxiv.org/abs/2301.00234
- Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Guo, Q., Wang, M., & Wang, H. (2024). *Retrieval-Augmented Generation for Large Language Models: A Survey* (arXiv:2312.10997). arXiv. https://doi.org/10.48550/arXiv.2312.10997
- Hu, Z., Wang, L., Lan, Y., Xu, W., Lim, E.-P., Bing, L., Xu, X., Poria, S., & Lee, R. K.-W. (2023). LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models (arXiv:2304.01933). arXiv. https://doi.org/10.48550/arXiv.2304.01933
- Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., & Raffel, C. A. (2022). Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning. *Advances in Neural Information Processing Systems*, *35*, 1950–1965.
- Mosbach, M., Pimentel, T., Ravfogel, S., Klakow, D., & Elazar, Y. (2023). Few-shot Fine-tuning vs. In-context Learning: A Fair Comparison and Evaluation (arXiv:2305.16938). arXiv. https://doi.org/10.48550/arXiv.2305.16938
- Wang, Y., Yao, Q., Kwok, J. T., & Ni, L. M. (2020). Generalizing from a Few Examples: A Survey on Few-shot Learning. *ACM Computing Surveys*, *53*(3), 63:1-63:34. https://doi.org/10.1145/3386252
- Zhang, B., Titov, I., & Sennrich, R. (2021). *Sparse Attention with Linear Units* (arXiv:2104.07012). arXiv. https://doi.org/10.48550/arXiv.2104.07012
- Zhang, Z., Zheng, C., Tang, D., Sun, K., Ma, Y., Bu, Y., Zhou, X., & Zhao, L. (2023). *Balancing Specialized and General Skills in LLMs: The Impact of Modern Tuning and Data Strategy* (arXiv:2310.04945). arXiv. https://doi.org/10.48550/arXiv.2310.04945