Getting started with ellmer

Source: vignettes/ellmer.Rmd

```
library(ellmer)
```

ellmer makes it easy to access the wealth of large language models (LLMs) from R. But what can you do with those models once you have access to them? This vignette will give you the basic vocabulary you need to use an LLM effectively and will show you some examples to ignite your creativity.

In this vignette we'll mostly ignore how LLMs work, using them as convenient black boxes. If you want to get a sense of how they actually work, we recommend watching Jeremy Howard's posit::conf(2023) keynote: A hacker's guide to open source LLMs.

Vocabulary

We'll start by laying out the key vocab that you'll need to understand LLMs. Unfortunately the vocab is all a little entangled: to understand one term you'll often have to know a little about some of the others. So we'll start with some simple definitions of the most important terms then iteratively go a little deeper.

It all starts with a **prompt**, which is the text (typically a question or a request) that you send to the LLM. This starts a **con-**

kens, which represent either individual words or subcomponents of a word. The tokens are used to compute the cost of using a model and to measure the size of the **context**, the combination of the current prompt and any previous prompts and responses used to generate the next response.

It's useful to make the distinction between providers and models. A **provider** is a web API that gives access to one or more **models**. The distinction is a bit subtle because providers are often synonymous with a model, like OpenAI and GPT, Anthropic and Claude, and Google and Gemini. But other providers, like Ollama, can host many different models, typically open source models like LLaMa and Mistral. Still other providers support both open and closed models, typically by partnering with a company that provides a popular closed model. For example, Azure OpenAI offers both open source models and OpenAI's GPT, while AWS Bedrock offers both open source models and Anthropic's Claude.

What is a token?

An LLM is a *model*, and like all models needs some way to represent its inputs numerically. For LLMs, that means we need some way to convert words to numbers. This is the goal of the **tokenizer**. For example, using the GPT 4o tokenizer, the string "When was R created?" is converted to 5 tokens: 5958 ("When"), 673 (" was"), 460 (" R"), 5371 (" created"), 30 ("?"). As you can see, many simple strings can be represented by a single token. But more complex strings require multiple tokens. For example, the string "counterrevolutionary" requires 4 tokens: 32128 ("counter"), 264 ("re"), 9477 ("volution"), 815

It's important to have a rough sense of how text is converted to tokens because tokens are used to determine the cost of a model and how much context can be used to predict the next response. On average an English word needs ~1.5 tokens so a page might require 375-400 tokens and a complete book might require 75,000 to 150,000 tokens. Other languages will typically require more tokens, because (in brief) LLMs are trained on data from the internet, which is primarily in English.

LLMs are priced per million tokens. State of the art models (like GPT-4.1 or Claude 3.5 sonnet) cost $2-3 per million input tokens, and $10-15 per million output tokens. Cheaper models can cost much less, e.g. GPT-4.1 nano costs $0.10 per million input tokens and $0.40 per million output tokens. Even $10 of API credit will give you a lot of room for experimentation, particularly with cheaper models, and prices are likely to decline as model performance improves.

Tokens also used to measure the context window, which is how much text the LLM can use to generate the next response. As we'll discuss shortly, the context length includes the full state of your conversation so far (both your prompts and the model's responses), which means that cost grow rapidly with the number of conversational turns.

In ellmer, you can see how many tokens a conversations has used by printing it, and you can see total usage for a session with `token_usage()`.

```
chat <- chat_openai(model = "gpt-4.1")
. <- chat$chat("Who created R?", echo = FALSE)
chat
#> <Chat OpenAI/gpt-4.1 turns=2 input=11 output=110
```

```
#> **R** is a programming language and software env
token_usage()
#>   provider   model input output cached_input pric
#> 1   OpenAI gpt-4.1    11    110            0 $0.0
```

If you want to learn more about tokens and tokenizers, I'd recommend watching the first 20-30 minutes of Let's build the GPT Tokenizer by Andrej Karpathy. You certainly don't need to learn how to build your own tokenizer, but the intro will give you a bunch of useful background knowledge that will help improve your understanding of how LLM's work.

What is a conversation?

A conversation with an LLM takes place through a series of HTTP requests and responses: you send your question to the LLM as an HTTP request, and it sends back its reply as an HTTP response. In other words, a conversation consists of a sequence of a paired turns: a sent prompt and a returned response.

It's important to note that a request includes not only the current user prompt, but every previous user prompt and model response. This means that:

- The cost of a conversation grows quadratically with the number of turns: if you want to save money, keep your conversations short.
- Each response is affected by all previous prompts and responses. This can make a conversation get stuck in a local optimum, so it's generally better to iterate by starting a new conversation with a better prompt rather than having a long back-and-forth.

tion with one model and finish it with another.

What is a prompt?

The user prompt is the question that you send to the model. There are two other important prompts that underlie the user prompt:

- The **platform prompt**, which is unchangeable, set by the model provider, and affects every conversation. You can see what these look like from Anthropic, who publishes their core system prompts.

- The **system prompt** (aka developer prompt), which is set when you create a new conversation, and affects every response. It's used to provide additional instructions to the model, shaping its responses to your needs. For example, you might use the system prompt to ask the model to always respond in Spanish or to write dependency-free base R code. You can also use the system prompt to provide the model with information it wouldn't otherwise know, like the details of your database schema, or your preferred ggplot2 theme and color palette.

OpenAI calls this the chain of command: if there are conflicts or inconsistencies in the prompts, the platform prompt overrides the system prompt, which in turn overrides the user prompt.

When you use a chat app like ChatGPT or Claude.ai you can only iterate on the user prompt. But when you're programming with LLMs, you'll primarily iterate on the system prompt. For example, if you're developing an app that helps a

Writing a good prompt, which is called **prompt design**, is key to effective use of LLMs. It is discussed in more detail in `vignette("prompt-design")`.

Example uses

Now that you've got the basic vocab under your belt, I'm going to fire a bunch of interesting potential use cases at you. While there are special purpose tools that might solve these cases faster and/or cheaper, an LLM allows you to rapidly prototype a solution. This can be extremely valuable even if you end up using those more specialised tools in your final product.

In general, we recommend avoiding LLMs where accuracy is critical. That said, there are still many cases for their use. For example, even though they always require some manual fiddling, you might save a bunch of time ever with an 80% correct solution. In fact, even a not-so-good solution can still be useful because it makes it easier to get started: it's easier to react to something rather than to have to start from scratch with a blank page.

Chatbots

A great place to start with ellmer and LLMs is to build a chatbot with a custom prompt. Chatbots are a familiar interface to LLMs and are easy to create in R with shinychat. And there's a surprising amount of value to creating a custom chatbot that has a prompt stuffed with useful knowledge. For example:

- Help people use your new package. To do so, you need a custom prompt because LLMs were trained on data prior

works.

- Build language specific prompts for R and/or Python. Shiny Assistant helps you build shiny apps (either in R or Python) by combining a prompt that gives general advice on building apps with a prompt for R or python. The Python prompt is very detailed because there's much less information about Shiny for Python in the existing LLM knowledgebases.

- Help people find the answers to their questions. Even if you've written a bunch of documentation for something, you might find that you still get questions because folks can't easily find exactly what they're looking for. You can reduce the need to answer these questions by creating a chatbot with a prompt that contains your documentation. For example, if you're a teacher, you could create a chatbot that includes your syllabus in the prompt. This eliminates a common class of question where the data necessary to answer the question is available, but hard to find.

Another direction is to give the chatbot additional context about your current environment. For example, aidea allows the user to interactively explore a dataset with the help of the LLM. It adds summary statistics about the dataset to the prompt so that the LLM knows something about your data. Along these lines, imagine writing a chatbot to help with data import that has a prompt which include all the files in the current directory along with their first few lines.

Structured data extraction

LLMs are often very good at extracting structured data from unstructured text. This can give you traction to analyse data

quick and dirty sentiment analysis by extracting any specifically mentioned products and summarising the discussion as a few bullet points.

- Geocoding: LLMs do a surprisingly good job at geocoding, especially extracting addresses or finding the latitute/longitude of cities. There are specialised tools that do this better, but using an LLM makes it easy to get started.

- Recipes: I've extracted structured data from baking and cocktail recipes. Once you have the data in a structured form you can use your R skills to better understand how recipes vary within a cookbook or to look for recipes that use the ingredients currently in your kitchen. You could even use shiny assistant to help make those techniques available to anyone, not just R users.

Structured data extraction also works well with images. It's not the fastest or cheapest way to extract data but it makes it really easy to prototype ideas. For example, maybe you have a bunch of scanned documents that you want to index. You can convert PDFs to images (e.g. using {imagemagick}) then use structured data extraction to pull out key details.

Learn more about structured data extraction in `vignette("structured-data")`.

Programming

LLMs can also be useful to solve general programming problems. For example:

- Write a detailed prompt that explains how to update code to use a new version of a package. You could combine this

which includes prompts for automatically generating roxygen documentation blocks, updating testthat code to the 3rd edition, and converting `stop()` and `abort()` to use `cli::cli_abort()`.

- You could automatically look up the documentation for an R function, and include it in the prompt to make it easier to figure out how to use a specific function.
- You can use LLMs to explain code, or even ask them to generate a diagram.
- You can ask an LLM to analyse your code for potential code smells or security issues. You can do this a function at a time, or explore the entire source code of your package or script in the prompt.
- You could use gh to find unlabelled issues, extract the text, and ask the LLM to figure out what labels might be most appropriate. Or maybe an LLM might be able to help people create better reprexes, or simplify reprexes that are too complicated?
- I find it useful to have an LLM document a function for me, even knowing that it's likely to be mostly incorrect. Having something to react to make it much easier for me to get started.
- If you're working with code or data from another programming language, you can ask an LLM to convert it to R code for you. Even if it's not perfect, it's still typically much faster than doing everything yourself.

Miscellaneous

- Automatically generate alt text for plots, using `content_image_plot()`.
- Analyse the text of your statistical report to look for flaws in your statistical reasoning (e.g. misinterpreting p-values or assuming causation where only correlation exists).
- Use your existing company style guide to generate a brand.yaml specification to automatically style your reports, apps, dashboards and plots to match your corporate style guide.

ON THIS PAGE