



THE LONDON SCHOOL
OF ECONOMICS AND
POLITICAL SCIENCE ■

Week 10: Large Language Models

MY472: Data for Data Scientists

<https://lse-my472.github.io/>

Autumn Term 2025

Ryan Hübert

Associate Professor of Methodology

Why cover LLMs in MY472?

1. Increasingly common for researchers to use LLMs in “data pipelines”, e.g.:
 - ▷ Cleaning data
 - ▷ Measuring concepts
2. LLMs are becoming critical infrastructure for data scientists
 - ▷ Can help with coding, and technical issues (but only help)

Goal for this week:

- ▷ High-level overview of what LLMs are (to demystify)
- ▷ Some discussion of how to use
- ▷ How to interact with them programmatically (in R)

Outline

- 1 What is a large language model?
- 2 Inference
- 3 Training
- 4 Prompt engineering
- 5 Improving access to information
- 6 When should you use LLMs?

What is a large language model?

A **large language model** is a large (estimated) model of language

Language model:

- ▷ Language is basically a sequence of utterances
- ▷ Can represent these as sequence of tokens x_1, x_2, \dots, x_T
- ▷ an LLM is a (very complex) statistical model that predicts tokens based on previous ones

Large:

- ▷ Billions (possibly trillions) of model parameters
- ▷ Can work with very large amounts of text: hundreds of thousands, or even millions, of words

The *model* in an LLM

There are two big sets of issues:

1. **Training**: how does the model learn how to speak?
 - ▷ Happens once, and weights are estimated
 - ▷ Analogy: a small child learning how to talk
2. **Inference**: predicting new text with the trained model
 - ▷ Analogy: me standing here talking to you

Inference is why it's called “**generative AI**”

We're going to talk about inference *then* training

- ▷ Inference will be very useful for you
- ▷ Training might be useful, but it is good background knowledge

Some early caveats and notes

1. LLMs are quite new, we're still figuring out how to use them
 - ▷ Lots of people using for range of things: replacing search engines, companionship, writing emails (ugh!), etc.
 - ▷ Social scientists are figuring out where they're most useful
2. Technical advances in LLMs are occurring rapidly
 - ▷ Good idea to keep up-to-date: read news coverage, follow new releases of models, try to understand technical advances
3. There are *many* LLMs out there
 - ▷ Here: we're focused on high-level ideas + some examples
 - ▷ Details will vary from model to model

Some early caveats and notes

4. Need to distinguish between companies, models and consumer products, for example:
 - ▷ OpenAI is a company that develops a set of models called generative pre-trained transformer (GPT) models, which are used in OpenAI's consumer app called ChatGPT
 - ▷ Microsoft has a consumer product called Copilot that uses OpenAI's GPT models (but customised by Microsoft)
5. Limited development of R packages
 - ▷ Most development happens in Python (sigh)
 - ▷ But, some R packages we'll look at and you can use standard [GET](#) requests for LLM APIs

Outline

- 1 What is a large language model?
- 2 Inference**
- 3 Training
- 4 Prompt engineering
- 5 Improving access to information
- 6 When should you use LLMs?

Inference: the basic process

- ▷ User provides a **prompt**
- ▷ Text is tokenised
- ▷ Tokens mapped to vector representations (VRs, or embeddings)
- ▷ VRs are passed through a **transformer**
 - This produces **hidden states** (one for each token)
 - Hidden states encode **context** of the prompt
- ▷ Context is used to create probability distribution over all possible tokens
- ▷ Tokens are then sampled one at a time
 - Each generated token is fed back into the transformer, updating the context
- ▷ Tokens are turned back to words and displayed on screen

How do LLMs tokenise?

You already saw: texts must be broken up into countable features

- ▷ We call this **tokenisation**
- ▷ The resulting features are **tokens**

LLMs don't tokenise in the simple way we did

Instead: **modified byte pair encoding (BPE)** tokenisers

- ▷ Tokens are bundles of commonly occurring consecutive substrings of Unicode characters
- ▷ A lookup table (like a character set) where every distinct token contains an ID number
- ▷ Keep in mind: different LLMs may use different tokenisers

How do LLMs tokenise?

OpenAI's recent models: `o200k_base` tokeniser

- ▷ It has about 200,000 unique tokens

OpenAI's tokeniser is written in Python, but there's an R wrapper

```
library("rtiktoken")
get_tokens(text = "London School of Economics",
           model = "gpt-4o") # gpt-5 doesn't work
```

```
[1] 51162 6586 328 56779
```

```
get_token_count(text = "London School of Economics",
                model = "gpt-4o")
```

```
[1] 4
```

How do LLMs tokenise?

What makes BPE tokenisers different from “traditional” tokenisers:

```
get_tokens(text = "london school of economics",  
           model = "gpt-4o")
```

```
[1]    75  8552  3474   328 48229
```

The tokeniser is case sensitive (so are “traditional” tokenisers)

But with this BPE tokeniser, “london” is two tokens!

- ▷ Why? Short answer: “London” is a more common set of co-occurring bytes than “london”

Play around here: <https://platform.openai.com/tokenizer>

Tokens to vector representations

LLMs don't use the tokens directly

Tokenising is just about breaking up text into parts

Tokens are converted into vector representations (VRs)

- ▷ Similar to word embeddings, but tokens \neq words
- ▷ Each token is mapped to a specific vector

Recall: embeddings allow for richer, denser representation of tokens (in contrast to one-hot encodings)

Like word embeddings last week, each model's embeddings are learned once during model training

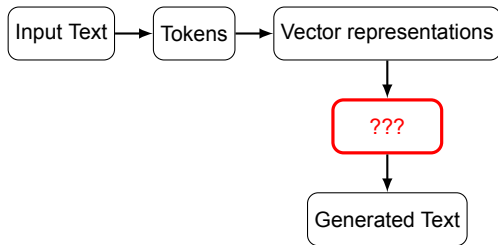
- ▷ Some models expose them (Llama), some don't (GPT-5)

Transformer architecture: big picture

So far: just simple steps to transform text to numerical objects

But *inside* the model, what happens?

- ▷ Remember basic goal: generate some (predicted) text



This is where things get a bit more complicated

How does the model decide what to say?

In language, meaning of a word depends on other words around it

- ▷ “bank” means money in one sentence, a riverbank in another
- ▷ Pronouns, references, topics, tone—all spread across long spans of text

We'll loosely call this the “context” of the word

Core challenge for a statistical model of language: determine “context” from a sequence of given tokens (from prompt)

Previous algorithms (like RNNs) processed tokens in order, but:

- ▷ Created computational bottlenecks
- ▷ Made long-range context hard to capture
- ▷ Slow to train these models; generated text not as good

How does the model decide what to say?

Huge break-through in 2017: **transformers**

We won't get into the details (see MY475), but:

- ▷ Transformers use an **attention mechanism** where context is determined from all tokens in parallel (not in sequence)
- ▷ Vastly sped up training, and made inference much better

What happens during inference? Two main steps:

1. **Forward pass on the prompt**: determine the context from the prompt
2. **Language modelling head**: use context to randomly sample tokens one at a time to generate text

Forward pass on the prompt → context

Forward pass on the prompt: VRs from the original prompt are “passed” through the transformer repeatedly

- ▷ Each pass is called a **transformer block** or a **layer**
- ▷ There are a fixed number of layers
- ▷ In each layer, all the VRs are updated to new VRs
 - How? See MY475!

After final block, the resulting (modified) VRs are called the **hidden states** for each of the original tokens

- ▷ These are mathematical representations of the “context” for each of the original prompt tokens

Context → text generation

After forward pass on the prompt, model has determined “context,” but no text has been generated yet

This happens in the **language modelling (LM) head**:

- ▷ Use context to create a prob. distribution over all tokens
- ▷ Sample first token of response
- ▷ Pass this new token through the transformer
- ▷ Use (slightly modified) context to create a new prob. distribution over vocabulary
- ▷ Sample second token of response
- ▷ And so on

(Stop when hard limit is reached or next token is a “stop” token)

The “temperature” of the text generation

In the LM head, tokens are sampled to generate text, but we didn't say *how* they were sampled

Temperature controls how the sampling occurs

- ▷ Low temperature: always pick the highest-probability token (deterministic)
- ▷ High temperature: spread the probabilities out (more randomness and variation)

Analogy to “salt” from encryption:

- ▷ You don't want same encryption key from same password
- ▷ LLM company doesn't want users to get the same responses to the same prompts

Context windows

There is a limit to how many tokens a model can contextualise

- ▷ A **context window** specifies the maximum number of tokens a model can “attend to” in a session/conversation
- ▷ Includes all tokens input (e.g., in prompts) and all tokens output by the model (e.g., in responses)

When context window is reached: model “forgets” oldest tokens

Different models have different context windows:

- ▷ **GPT-5.1**: 400K context window
- ▷ **Claude Sonnet 4.5**: 200K/1M context window
- ▷ **Gemini 3 Pro**: 1M context window

Performance can degrade before context window is reached!

Inference in R

You (probably) know how to do inference in the ChatGPT, Claude, Gemini and/or Copilots (web)apps

But you can do inference more programmatically in R:

- ▷ Web APIs
- ▷ Run locally

We'll look at an example of running an LLM on your computer

In seminar: we'll look at web APIs

Inference in R

Two practical problems:

- ▷ Many models too computationally hungry for personal devices
- ▷ Even for small models, infrastructure set up is too complex for your average user, even a sophisticated data science user

There is a path forward!

- ▷ Some “small” models, like versions of Meta’s Llama 3.2
- ▷ Ollama is a **local LLM runtime**: software that helps you run LLMs on a computer with almost no setup
 - Before seminar: install it from <https://ollama.com/>

Inference with Llama in R

We will use the Llama 3.2 model with 3 billion weights

- ▷ This is a small model but there is a smaller version (1B weights)
- ▷ [According to Meta](#), this model can run “effortlessly on almost any modern computer”

In R, we will use the `{ollamar}` package to interact with the model via Ollama

```
# install.packages("ollamar")  
library("ollamar")
```

Inference with Llama in R

```
test_connection() # Launch Ollama app if not status 200
```

```
Ollama local server running  
<http2_response>  
GET http://localhost:11434/  
Status: 200 OK  
Content-Type: text/plain  
Body: In memory (17 bytes)
```

```
list_models() # Which models have I already downloaded?
```

	name	size	parameter_size	quantization_level	mod
1	llama3.2:3b	2 GB	3.2B	Q4_K_M	2025-12-02T13:00:00

```
my.model <- "llama3.2:3b" # Llama 3.2 (3B weight version)  
if(!model_avail(my.model)){  
  pull(my.model)  
}
```

Inference with Llama in R

```
response <- generate(model = my.model,  
                     prompt = "When was LSE founded?")  
response
```

```
<httr2_response>  
POST http://127.0.0.1:11434/api/generate  
Status: 200 OK  
Content-Type: application/json  
Body: In memory (1206 bytes)
```

```
resp_process(response, output = "jsonlist")
```

```
$model  
[1] "llama3.2:3b"
```

```
$created_at  
[1] "2025-12-02T13:25:02.538537Z"
```

```
$response  
[1] "The London School of Economics (LSE) was founded in 1895."
```

```
...
```

Inference with Llama in R

```
response <- generate(model = my.model,  
                      prompt = "When was LSE founded?",  
                      output = "text")  
response
```

```
[1] "The London School of Economics (LSE) was founded in 1895."
```

```
response <- generate(model = my.model,  
                      prompt = "When was LSE founded?",  
                      output = "df")  
response
```

```
# A tibble: 1 × 3  
  model      response      created_at  
  <chr>    <chr>         <chr>  
1 llama3.2:3b The London School of Economics (LS... 2025-12-0...
```

Inference with Llama in R

You can control the temperature of the LM head using the `temperature` option in the `generate()` function

But first: check what is default used?

```
search_options("temperature")
```

```
Matching options: temperature
```

```
$temperature
```

```
$temperature$description
```

```
[1] "The temperature of the model. Increasing the temperature  
will make the model answer more creatively."
```

```
$temperature$default_value
```

```
[1] 0.8
```

Inference with Llama in R

```
cat(generate(model = my.model,  
             prompt = "Give me a sentence with 7 words.",  
             output = "text",  
             temperature = 0))
```

The sun was shining brightly in California.

```
cat(generate(model = my.model,  
             prompt = "Give me a sentence with 7 words.",  
             output = "text",  
             temperature = 0))
```

The sun was shining brightly in California.

Inference with Llama in R

```
cat(generate(model = my.model,  
             prompt = "Give me a sentence with 7 words.",  
             output = "text",  
             temperature = 2.8))
```

The dog is running around the house.

```
cat(generate(model = my.model,  
             prompt = "Give me a sentence with 7 words.",  
             output = "text",  
             temperature = 2.8))
```

The big brown bear climbed the mountain.

Outline

- 1 What is a large language model?
- 2 Inference
- 3 Training**
- 4 Prompt engineering
- 5 Improving access to information
- 6 When should you use LLMs?

Training

So far, just talked about inference (how LLMs make new text)

However, each model (like GPT-5) has to be trained

Typically happens in two parts:

1. **Pretraining**: use large corpus of texts to learn general patterns, grammar, facts
 - ▷ Supervised learning—use existing texts to train
 - ▷ Teams of researchers assemble high-quality internet corpuses
 - ▷ See e.g. https://en.wikipedia.org/wiki/GPT-3#Training_and_capabilities
 - ▷ Generates (most of) model's **weights** (or **parameters**)

Training

Pretraining creates a model that can ‘speak’ very realistically

But, it doesn’t know how to follow instructions, nor does it have a sense of what it should and shouldn’t do

- ▷ It’s a cute party trick: create text that mimics human speech
- ▷ Point of these models: be useful and responsive to humans

2. **Posttraining:**

- ▷ Supervised fine-tuning: train on prompt-response pairs (written by humans)
- ▷ Reinforcement learning: reinforce good behaviour based on human or programmatic feedback
- ▷ Updates model weights to encourage desirable behaviour, known as **alignment**

Fine tuning

Many of the most famous LLMs are **general-purpose models**

- ▷ This just means they are trained for “general” use
- ▷ Consumer facing products like ChatGPT and Claude use general-purpose models

But specialised use cases might require more specialised models that generate text with different properties

Some LLMs providers allow users to **fine-tune** their general-purpose models for specific applications

- ▷ This entails further training with specialised prompts and responses for your use case
- ▷ E.g.: a company building an AI nurse would want it to speak in certain ways that might differ from general conversation

Open versus closed models

One major difference between models that's relevant to data scientists: are models **open weight** or **closed weight**?

Open weight models allow users to see the weights generated during the training process

- ▷ Can easily fine tune them
- ▷ Examples: Llama, Mistral, DeepSeek

Closed weight models do not expose the weights

- ▷ Can only customise within the bounds of tools provided by the companies, such as APIs (and also have to pay)
- ▷ Examples: GPT-5, Claude, Gemini

Open versus closed models

There are trade-offs

Closed-weight models:

- ▷ Typically perform better (for generic tasks)
- ▷ Require less technical expertise (APIs are quite easy to use)

Open-weight models:

- ▷ Very customisable
- ▷ Can generate more reproducible results
- ▷ Ensure data protection (if all inference is on machine)

Model size

Another way models vary is **model size** (number of weights)

- ▷ More weights correlate with better performance
- ▷ Take more space, memory, computational resources

For closed-weight models: companies are very opaque about the number of weights

But ChatGPT guesses:

Model	Lower-bound (plausible min)	"Most likely" estimate (my guess)	Upper-bound (plausible max)
GPT-5	~800 billion params	~1.8–2.2 trillion params	~3–4 trillion params
Gemini 3 Pro	~300 billion params	~600–900 billion params	~1.2–1.5 trillion params
Claude Sonnet 4.5	~200 billion params	~450–700 billion params	~1–1.2 trillion params

Model “personalities”

Each LLM has been post-trained in a specific way, which leads to perceived differences:

- ▷ OpenAI’s GPT models: more informational
- ▷ Anthropic’s Claude models: more chatty, emotional and whimsical
- ▷ Google’s Gemini models: more technical

Why? Companies are competing to offer popular products and differentiate their product from other products

This matters (somewhat) less for data science pipelines than technical details like model size, context windows, etc.

Summary of main differences across models

Computational performance:

1. Depth (how many layers)
2. Size (how many parameters)
3. Context window (how many tokens per conversation)

Substantive performance:

1. Pre-training (corpus size and breadth)
2. Post-training (what behaviours reinforced)
3. Reasoning (is model trained to reason by default)
4. Access to outside “resources” (documents, internet search, etc.)

Outline

- 1 What is a large language model?
- 2 Inference
- 3 Training
- 4 Prompt engineering**
- 5 Improving access to information
- 6 When should you use LLMs?

What is “prompt engineering”?

Remember: when you use an LLM for inference:

- ▷ You're not training the model
 - Caveat: companies *might* use your conversations to “improve their products”
- ▷ You're using a (very good) probabilistic text generator

Your prompts are a form of probabilistic conditioning:

- ▷ A prompt defines “starting context” for LLM's predictions

Prompt engineering is the practice of carefully constructing prompts to generate useful responses (for task at hand)

What is “prompt engineering”?

Why is this important?

Garbage in, garbage out (GIGO) principle applies: if your prompts are bad, the output will be bad too

But special problem with language: it's full of ambiguities

LLMs have been *trained* to write responses humans like

- ▷ Humans are easily impressed (and persuaded) by articulate language even when *substantive content* is low quality

You must consistently have your guard up against sycophancy and being persuaded by articulate speech

Always scrutinise the quality of LLM outputs!

Types of prompts

There are two main categories of prompts

1. **Descriptive prompts**

- ▷ Example: “Data scientists analyse data and...”
- ▷ Model will continue text generically

2. **Instructional prompts**

- ▷ Example: “Write a two-sentence summary of what data scientists do.”
- ▷ Model follows the command, returns a summary

Instructional prompts are the bread-and-butter of data science applications

Types of instructions

```
response <- generate(model = "llama3.2:3b",  
                     prompt = "Data scientists analyse data and...",  
                     output = "text")  
cat(response)
```

...draw insights, identify patterns, and create predictive models to inform business decisions or solve complex problems.

Types of instructions

```
response <- generate(model = "llama3.2:3b",  
                     prompt = "Write a two-sentence summary of  
                               what data scientists do.",  
                     output = "text")  
cat(response)
```

Data scientists are professionals who collect, analyze, and interpret complex data to extract insights and inform business decisions, often using machine learning, statistical modeling, and data visualization techniques. By applying their expertise in data science, they help organizations identify trends, patterns, and correlations that can drive growth, improve efficiency, and optimize operations.

Types of instructions

Can get better responses with more *complex* prompting

Chain-of-thought (CoT) prompting

- ▷ Instruct the LLM to explain its reasoning path
- ▷ Example: “Question: What is 12% of 150? Instruction: Think step by step before answering.”

Reflexive (or meta) prompting

- ▷ Instruct the LLM to evaluate and/or revise its own response before finalising it
- ▷ Example: “Draft your answer, then check it for logical consistency before giving the final version.”

Types of instructions

```
response <- generate(model = "llama3.2:3b",  
                     prompt = "Question: What is 12% of 150?  
                               Instruction: Think step by step  
                               before answering.",  
                     output = "text")  
cat(response)
```

To find 12% of 150, we can follow these steps:

1. Convert the percentage to a decimal: $12\% = 0.12$
2. Multiply the decimal by the number: $0.12 \times 150 = ?$

Let's multiply:

$$0.12 \times 100 = 12$$

$$0.12 \times 50 = 6$$

$$\text{Adding those together: } 12 + 6 = 18$$

So, 12% of 150 is 18.

Types of instructions

```
response <- generate(model = "llama3.2:3b",  
                     prompt = "Question: What is 12% of 150?  
                               Instruction: Draft your answer,  
                               then check it for logical consistency  
                               before giving the final version.",  
                     output = "text")  
cat(response)
```

Draft answer: 18.

Checking for logical consistency:

To calculate 12% of 150, I would multiply 150 by 0.12
(which is equivalent to 12%).

$150 \times 0.12 = 18$

Since this calculation matches my initial guess, my
final answer remains the same: 18

Types of instructions

Reasoning models are *trained* to do CoT and be reflexive

- ▷ They do not “reason” like a human, but they do try to find a better answer

They work by:

- ▷ Generating intermediate reasoning steps
- ▷ Feeding those steps back into their own context
- ▷ Refining their thinking across multiple passes
- ▷ Producing a final answer for the user

Reasoning uses more tokens, but text from each step of reasoning can be extracted

Types of instructions

Structured prompting

- ▷ Instruct LLM to provide machine-readable or consistent output formats
- ▷ Example: “Summarize this article. Return JSON with this structure: {“summary”: “...”, “keywords”: [“...”]}”

Iterative prompting

- ▷ Instruct refining with feedback loops (human or code).

Conditioning versus training

Keep in mind:

- ▷ Instructions do not “train” the LLM
- ▷ They simply narrow the context for the LLM
- ▷ After you end your chat/session, the instructions are lost
- ▷ Caveat: stored memories

You can also condition by providing examples

- ▷ **Zero-shot** prompting: give LLM a task without examples
- ▷ **Few-shot** prompting: give LLM a task with some examples

Zero-shot prompting is common and tempting

- ▷ Save money and time
- ▷ But: responses will be based on general knowledge

Shot-prompting

You are an assistant that classifies sentiment.

Example 1:

Text: "I love this product!"

Sentiment: Positive

Example 2:

Text: "This is the worst service I've had."

Sentiment: Negative

Example 3:

Text: "It's fine, nothing special."

Sentiment: Neutral

Now classify this:

Text: "The food was okay but the service was slow."

Sentiment:

Shot-prompting

```
response <- generate(model = "llama3.2:3b",  
                     prompt = "You are an assistant that classifies sentiment  
                               Example 1: Text: 'I love this product!'  
                               Sentiment: Positive  
                               Example 2: Text: 'This is the worst service I  
                               Sentiment: Negative  
                               Example 3: Text: 'It's fine, nothing special.  
                               Sentiment: Neutral  
                               Now classify this:  
                               Text: 'The food was okay but the service was  
                               Sentiment:",  
                     output = "text")  
cat(response)
```

Based on the text "The food was okay but the service was slow.",
I would classify the sentiment as: Neutral

This is because the text contains both positive (the food was okay) and negative (the service was slow) opinions, which balance each other out. The overall tone of the sentence is neither strongly positive nor strongly negative, but rather neutral and somewhat mixed.

Instruction-following behaviour

Message roles structure how instructions are followed

For many models:

- ▶ **system** (or **developer**): messages that set overall behavior, tone, boundaries, or domain context
 - System messages are applied to entire conversation
- ▶ **user**: messages that give the task-specific instruction
 - This is the “prompt” you think of in normal ChatGPT use
- ▶ **assistant**: LLM’s response, which is conditioned on both higher-level and user-level instructions

See: <https://platform.openai.com/docs/guides/text#message-roles-and-instruction-following>

Instruction-following behaviour

```
prompt <- list()
prompt[[1]] <- list(role = "system",
                    content = "Be very snarky and very concise!")
prompt[[2]] <- list(role = "user",
                    content = "Write a two-sentence summary of
                               what data scientists do.")

response <- chat(model = "llama3.2:3b",
                 messages = prompt,
                 output = "text")

cat(response)
```

Data scientists are basically super-smart people who collect, analyze, and interpret messy data to help humans make better decisions (or at least pretend like they did). They're like the superheroes of statistics, but with more spreadsheets and fewer capes.

Outline

- 1 What is a large language model?
- 2 Inference
- 3 Training
- 4 Prompt engineering
- 5 Improving access to information**
- 6 When should you use LLMs?

LLMs do not know everything

LLMs are very powerful, but they do not have all the relevant information a user may need

- ▷ They know what they were trained on
- ▷ What was the corpus used to train them?
- ▷ What information was used for alignment?

Many applications require access to specialised information:

- ▷ LLMs designed for legal applications need to “know” the law
- ▷ LLMs designed for medicine need to “know” medical information
- ▷ LLMs designed for customer service (chatbots) need to “know” internal information about the company’s processes

LLMs do not know everything

Lack of knowledge is a problem, but LLMs make it worse

They often **hallucinate** when they don't have info they need



The screenshot shows the top navigation bar of The Guardian website. The navigation menu includes 'News', 'Opinion', 'Sport', 'Culture', and 'Lifestyle'. Below this is a secondary menu with regional and topical links: 'World', 'Europe', 'US news', 'Americas', 'Asia', 'Australia', 'Middle East', 'Africa', 'Inequality', and 'Global development'. The article title is 'US lawyer sanctioned after being caught using ChatGPT for court brief'. A yellow banner above the title states 'This article is more than 5 months old'. The sub-headline reads: 'Richard Bednar apologized after Utah appeals court discovered false citations, including one nonexistent case'.

UK

The Guardian

News Opinion Sport Culture Lifestyle

World Europe **US news** Americas Asia Australia Middle East Africa Inequality Global development

Utah

This article is more than 5 months old

US lawyer sanctioned after being caught using ChatGPT for court brief

Richard Bednar apologized after Utah appeals court discovered false citations, including one nonexistent case

LLMs do not know everything

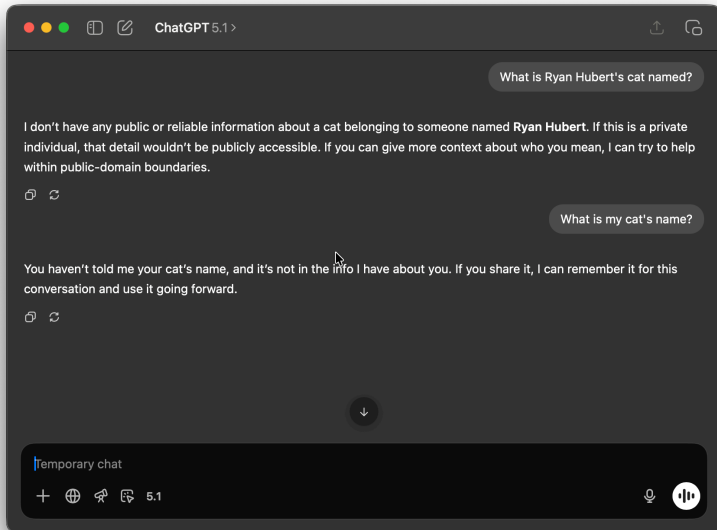
This is *the* major problem LLM providers are trying to fix

- ▷ Models are now more willing to say “I don’t know”
- ▷ Many models can now search the internet for information

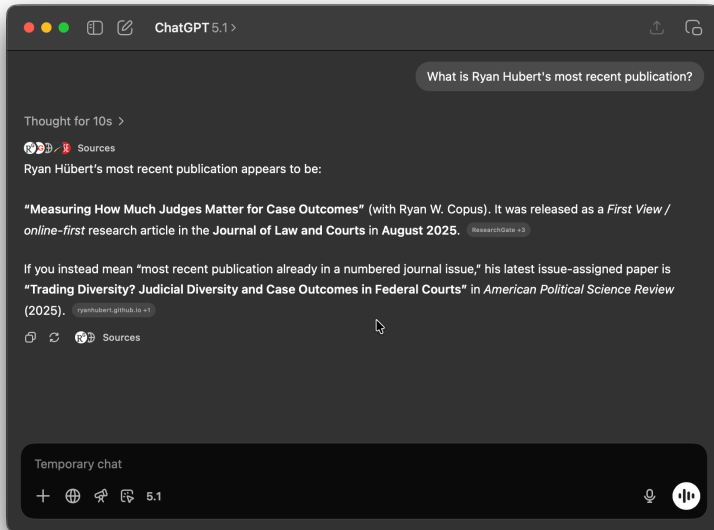
Of course, this depends on the model and the question

- ▷ GPT models *routinely* give me incorrect R or Python coding information

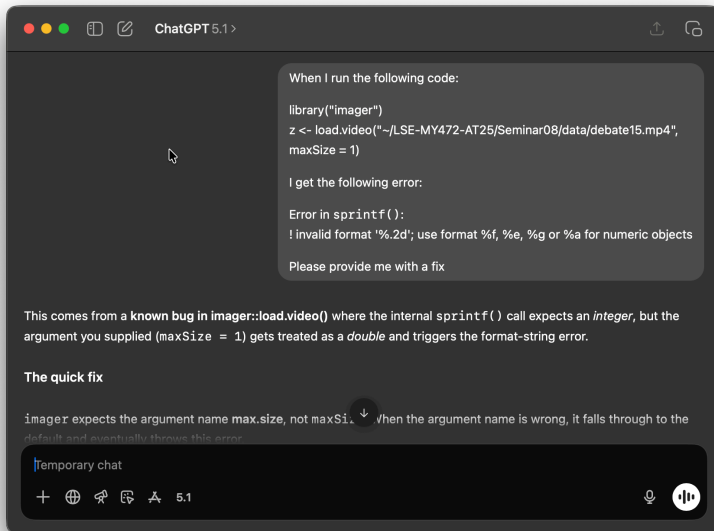
LLMs do not know everything



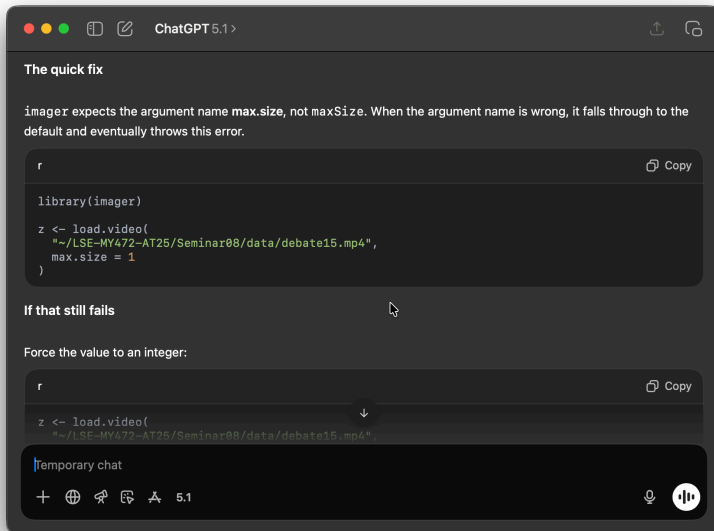
LLMs do not know everything



LLMs do not know everything



LLMs do not know everything



LLMs do not know everything

load.video {imager}

R Documentation

Load a video using ffmpeg

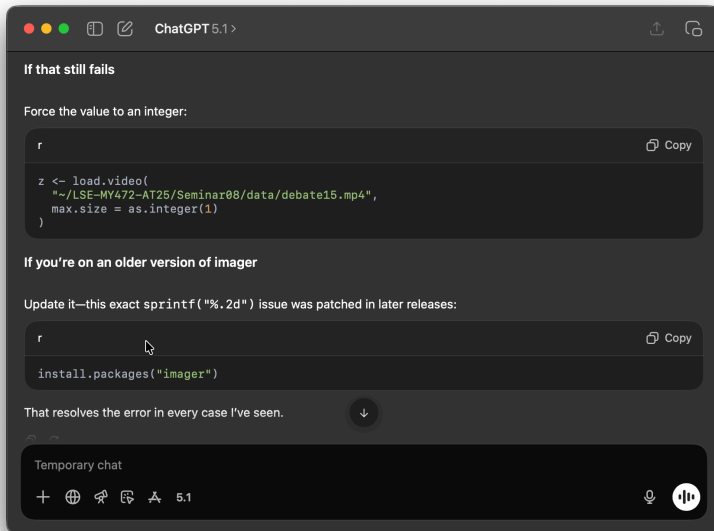
Description

You need to have ffmpeg on your path for this to work. This function uses ffmpeg to split the video into individual frames, which are then loaded as images and recombined. Videos are memory-intensive, and load.video performs a safety check before loading a video that would be larger than maxSize in memory (default 1GB)

Usage

```
load.video(  
  fname,  
  maxSize = 1,  
  skip.to = 0,  
  frames = NULL,  
  fps = NULL,  
  extra.args = "",  
  verbose = FALSE  
)
```

LLMs do not know everything



LLMs do not know everything

```
> z <- load.video("~/LSE-MY472-AT25/Seminar08/data/debate15.mp4",  
+               maxSize = as.integer(1))  
  
Error in `sprintf()`:  
! invalid format '%.2d'; use format %f, %e, %g or %a for numeric objects  
► Show Traceback
```

Retrieval-augmented generation

Retrieval-augmented generation (or **RAG**): *augment* LLM text generation by providing access to database of relevant information

Basic process:

1. Store relevant text chunks in an external database along with their VRs
2. Convert the user's prompt into an VR (the usual way)
3. Retrieve the most similar text chunks from the database using vector similarity search
4. Insert the retrieved text into the prompt
5. Run LLM inference

Retrieval-augmented generation

An example of a medical use: “What are the current contraindications for prescribing Drug X to pregnant patients?”

- ▷ LLM may provide outdated guidance, generic info or hallucinate

With RAG:

- ▷ Question is embedded
- ▷ System searches a vector database populated with recent clinical guidelines, FDA updates, latest peer-reviewed studies, internal protocols, etc.
- ▷ Retrieves relevant sections
- ▷ Text of sections are inserted into prompt along with instruction to *only* use provided information
- ▷ LLM summarises only authoritative material retrieved from database

Retrieval-augmented generation

Key innovation of RAG:

- ▷ Only relevant info from the database is extracted
- ▷ You are *not* feeding the entire database into each prompt
- ▷ This keeps prompt within the context window

Sounds complicated, but it's fairly easy to do

Closed-weight models have tools, e.g. OpenAI's "file search":

- ▷ Upload file(s) to the API
- ▷ Include file search tool in your API call, along with file ID
- ▷ The server does some sort of RAG (black-boxed)

Outline

- 1 What is a large language model?
- 2 Inference
- 3 Training
- 4 Prompt engineering
- 5 Improving access to information
- 6 When should you use LLMs?

When should you use LLMs?

This is all quite new—lots of people have wild ideas about how to use LLMs in their work

When are LLMs most appropriate?

- ▷ This is a hard question to answer: the tech is moving quickly

But, always remember what they are at their core: very fancy text prediction machines

- ▷ They have been trained to speak very well
- ▷ They have been trained to respond to real-world human requests
- ▷ Some of them are connected to outside resources

When should you use LLMs?

Who knows what the future has in store, but:

- ▷ They are most useful in contexts where *language* is important, like extracting meanings from text
- ▷ The best models are like good research assistants
 - They can understand your instructions
 - They do research to provide informed answers
 - They can extract meaning (and even subtext) from language
- ▷ They cannot think for you
 - They do not have original ideas, and they cannot come up with ideas for you
 - They do not “reason” like humans: LLM reasoning is pattern matching; human reasoning involves deductive thinking

When should you use LLMs?

- ▷ You really shouldn't use LLMs to do data analysis
 - Doesn't actually *understand* your data
 - Can hallucinate columns, misinterpret context, or invent plausible but wrong summaries
 - Token context limits: can't "see" an entire large dataset directly
- ▷ You should be very careful when you use LLMs to write code
 - For any complex task worth paying a data scientist for: LLMs still make lots of mistakes
 - LLMs give a false sense of competence

The problem with vibecoding

Some people believe that vibecoding is the future

- ▷ **Vibecoding** is prompting an LLM (with natural language) to write code to do some task
- ▷ Vibecoding is not a valued skill among data science employers

Coding involves giving *precise* instructions to do *specific* tasks

- ▷ Remember, computers are very literal
- ▷ The precision of code is one of its most desirable features

LLMs are designed to do pattern matching with natural language:

- ▷ Natural language is not precise—can be misinterpreted
- ▷ Models are (at least partly) stochastic
- ▷ Plus, they just make stuff up (you saw)

The problem with vibecoding

In my view: core problem with vibecoding is that the vibecoder does not learn how to be precise

This means:

- ▷ Does not give clear instructions (and LLM has to do a lot of interpreting and filling in blanks)
- ▷ Does not notice when LLM generates bad code

Vibecoding reveals a lax attitude about work quality

An increasing problem for companies: bad LLM-generated code

That said: LLMs are a productivity enhancer

- ▷ But you must use them carefully and methodically

Ethical considerations

There are many ethical considerations when using LLMs

1. Are you cheating?

- ▷ In some contexts, colleagues, supervisors, etc. want *your* work, *your* ideas, and *your* words
- ▷ It is probably cheating to use an LLM to answer questions on academic assignments
- ▷ It is probably cheating to ask an LLM to summarise a document you are specifically asked to read yourself
- ▷ Keep in mind: you cannot blame the LLM if you have legal troubles, or face disciplinary action at work/school

Ethical considerations

2. Are you exposing data?

- ▶ Any LLM that runs on different machine (e.g., ChatGPT) is most likely storing data you give it
 - At minimum, redact files/data before uploading to an LLM
 - Ask: “What memories do you have about me?”
- ▶ Uploading certain files or data to an online model may breach confidentiality, IRB rules, or even privacy laws
- ▶ Read up on companies’ policies, opt out of some settings
 - E.g., OpenAI API stores less data than ChatGPT (see [here](#))
- ▶ Where possible, use temporary/private chats, and models provided by your institution (with negotiated privacy terms)
- ▶ For very sensitive data, run local models only

Ethical considerations

3. Can your work be reproduced or replicated?
 - ▷ Can others get same results using your process?
 - ▷ Not good if your results are very model- or setting-dependent

4. Not quite an ethical concern: are you annoying other people?
 - ▷ Annoying or irresponsible use of LLMs can upset your peers, supervisors, colleagues, etc.
 - People can usually tell when you are using LLMs...
 - ▷ This can lead to consequences, sometimes without you knowing, such as a poor reputation or denied opportunities