Prototyping AI 1: Intro to LLMs

Michael Frantz | michaelgfrantz@gmail.com

About the instructor

- Background in molecular biology (undergrad) and neuroscience (PhD)
- Became invested in statistical methods in ~2013
- Taught General Assembly's Data Science Immersive course (2016)
- GoGuardian: Applications of NLP in education (2017-2024)
- meMR Health: Applications of NLP in medical record analysis (2024-present)

Course overview

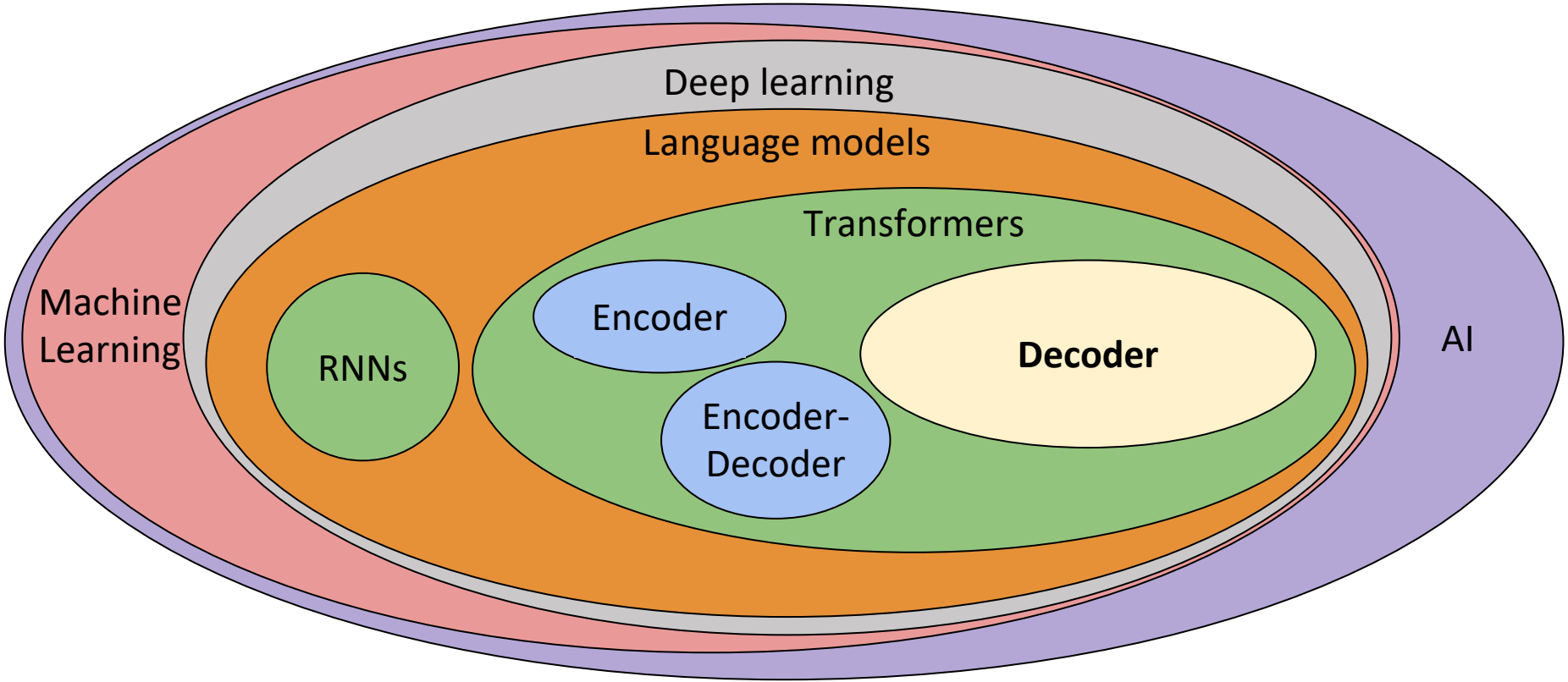
Day 1: Using LLMs in Python

- Setup OpenAI and Colab
- Build a basic chatbot
- Build a classifier using dynamic few-shot learning

Day 2: Retrieval augmented generation

- Build a basic “chat with PDF”
- Build an advanced “chat with PDF”
- Stretch goal: Agentic frameworks

What are large language models?



How large are large language models?

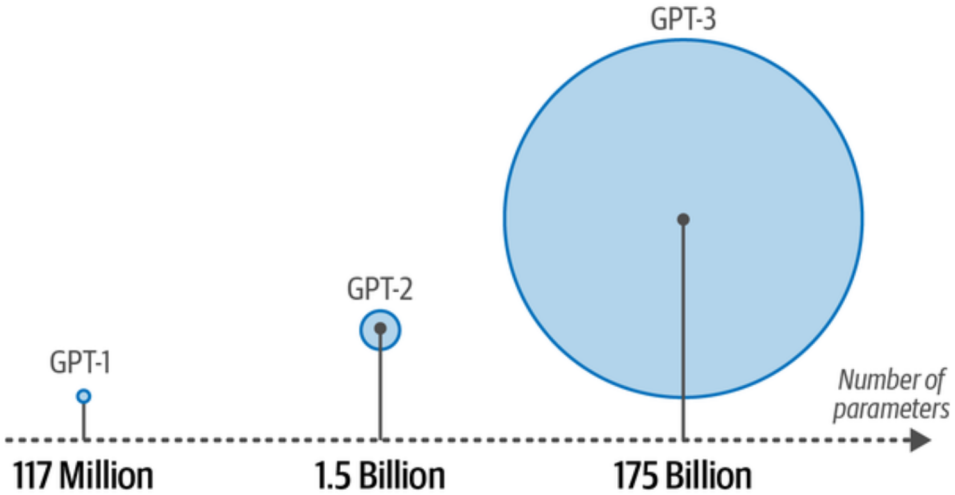
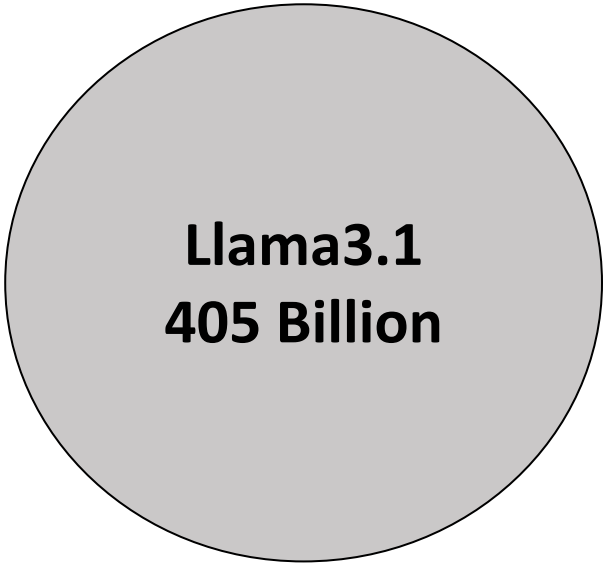
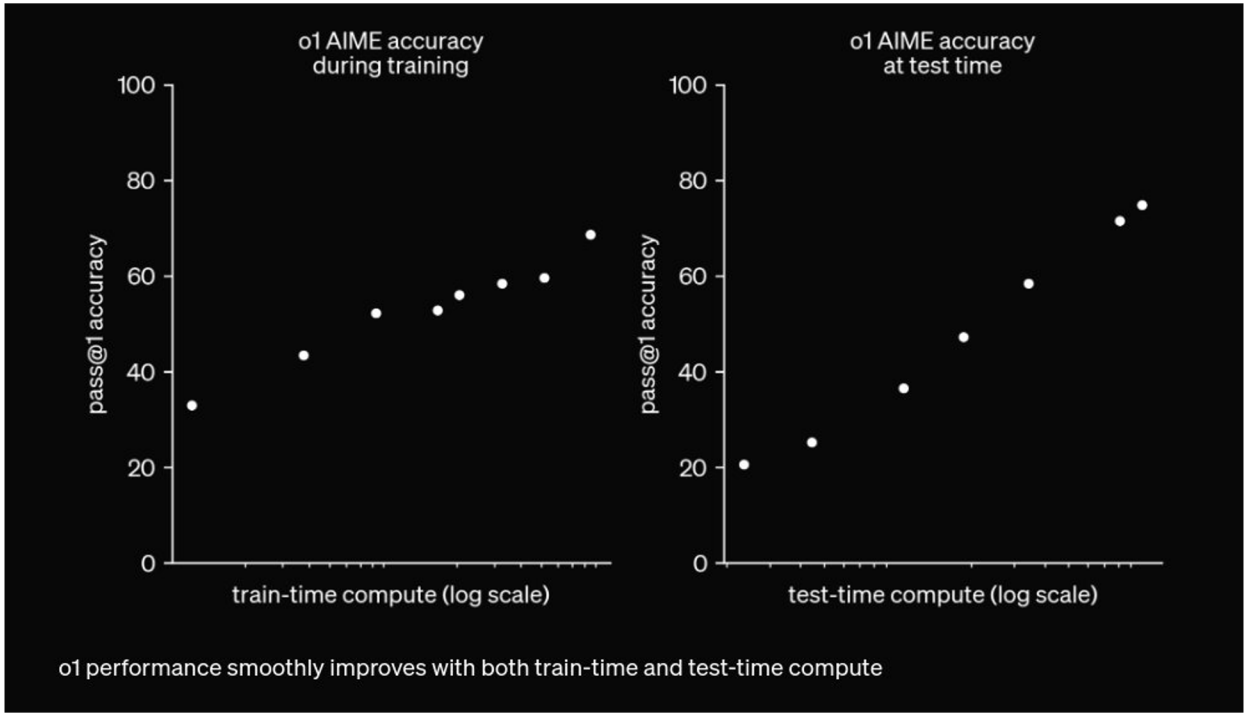


Figure 1-25. GPT models quickly grew in size with each iteration.



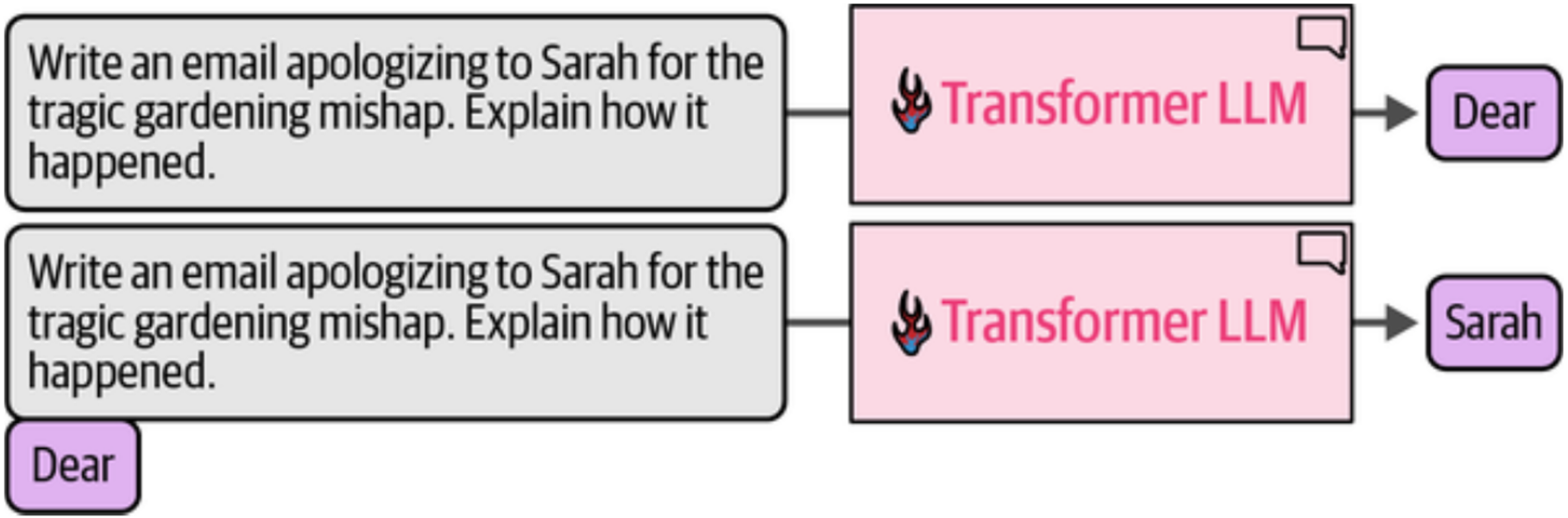
What makes them large language models?



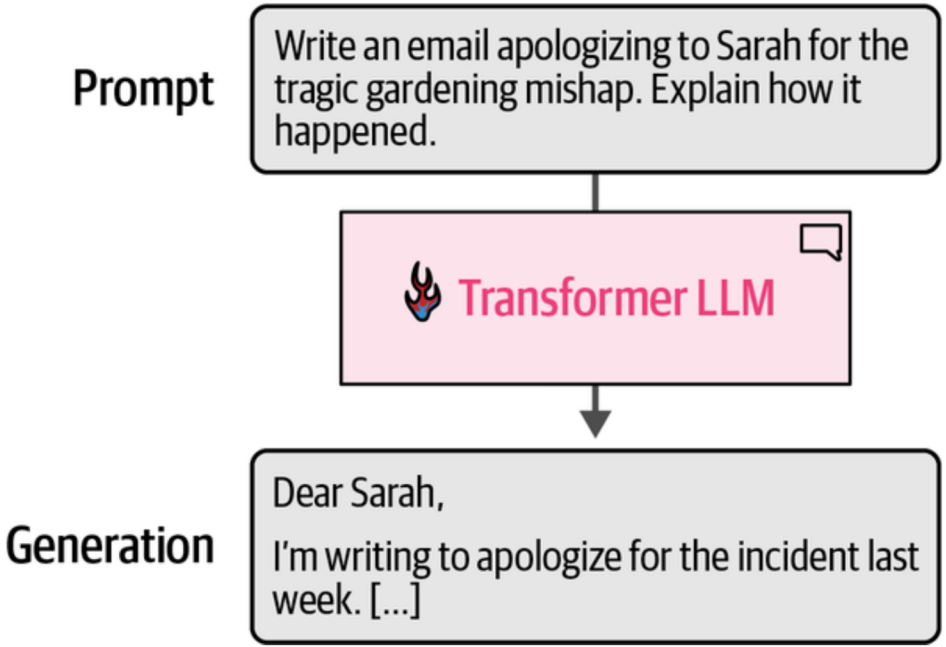
https://openai.com/index/learning-to-reason-with-llms

How do Transformer LLMs work?

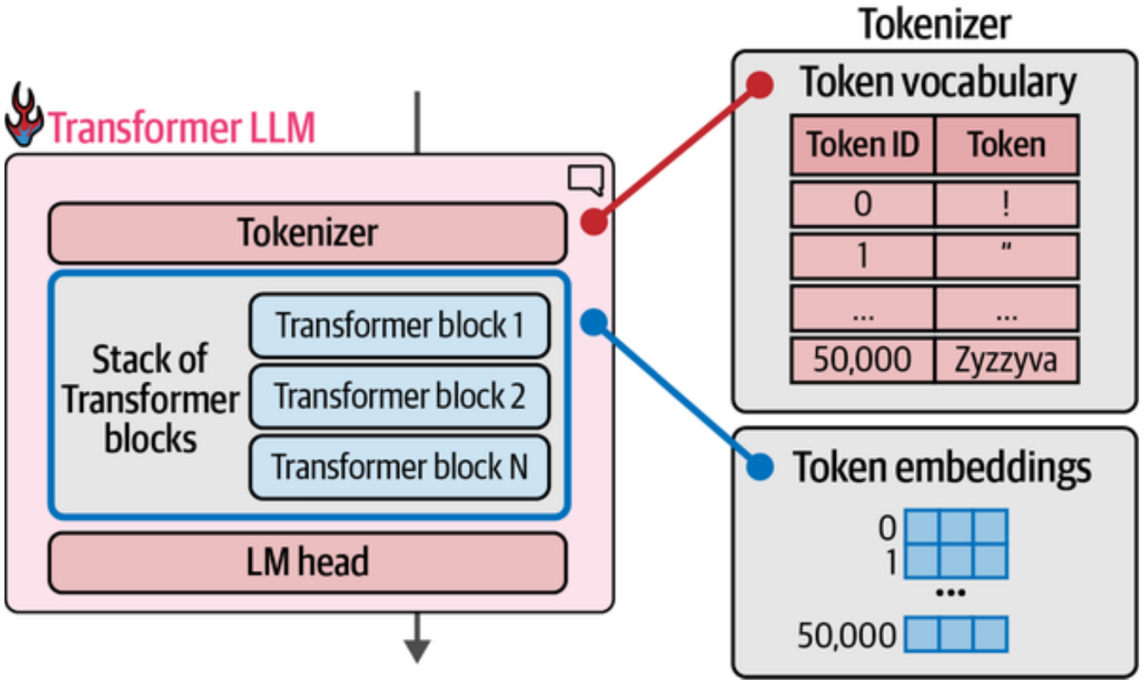
LLMs generate words 1 at a time



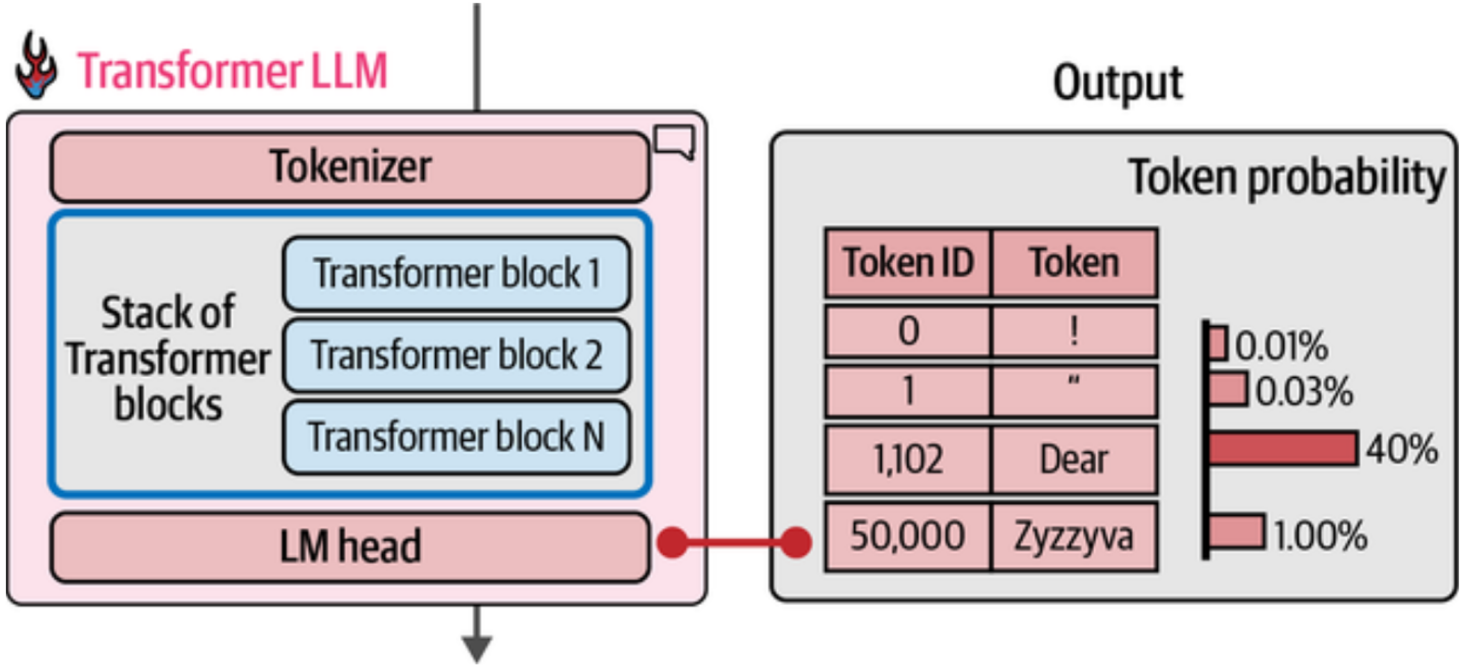
LLMs generate text based on a prompt



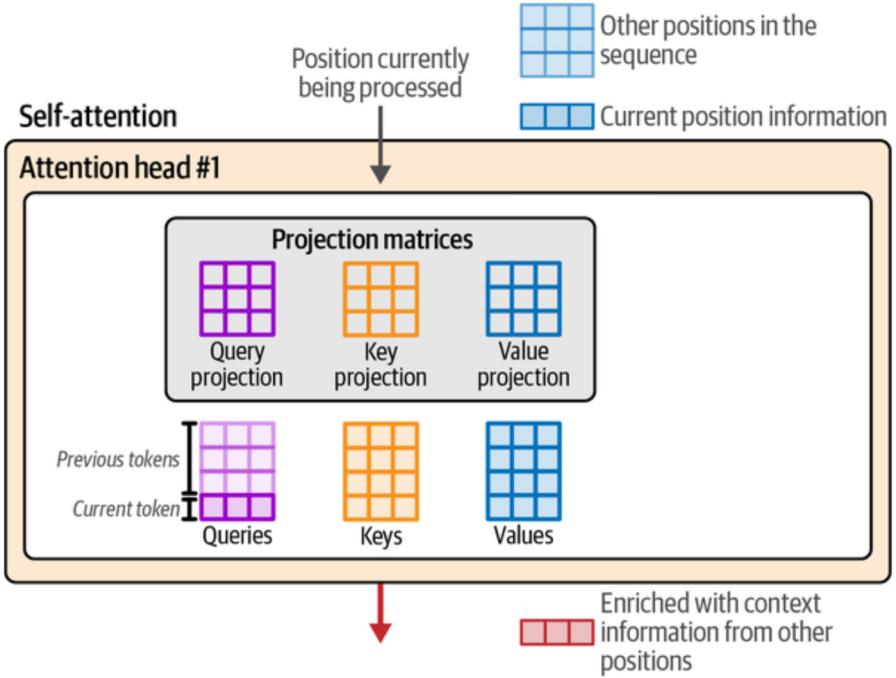
LLM inputs are tokens converted to vectors



LLM outputs are token probabilities

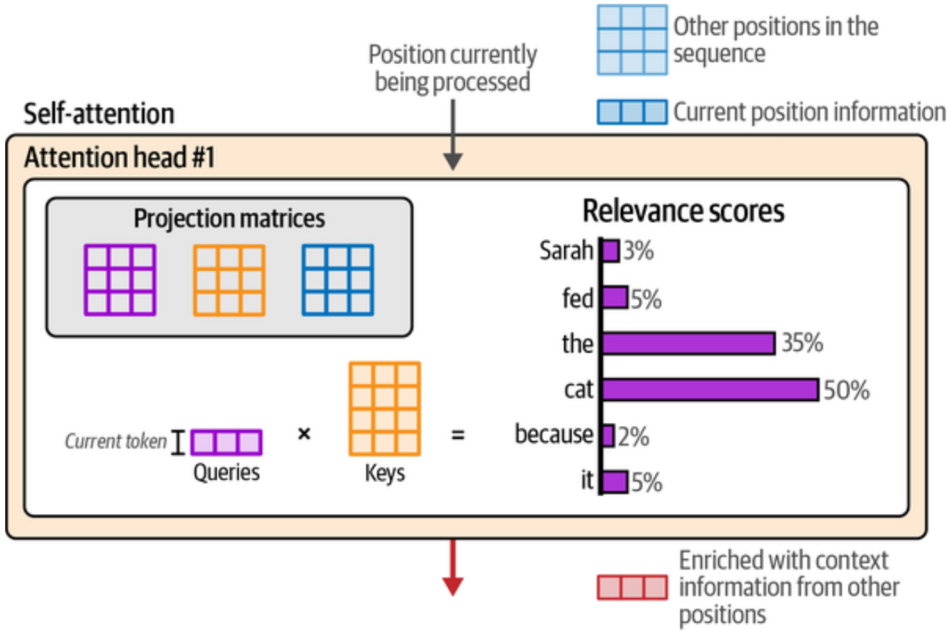


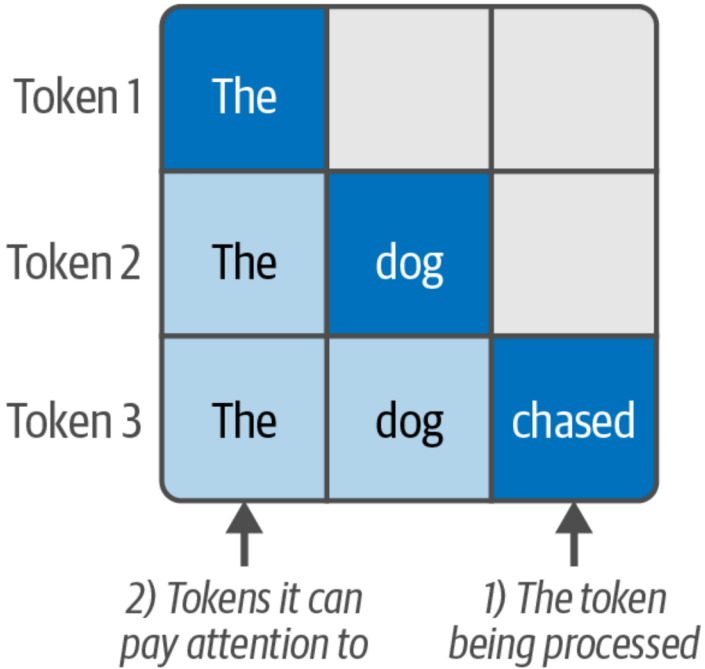
The transformer block



Self-attention: The attention matrix

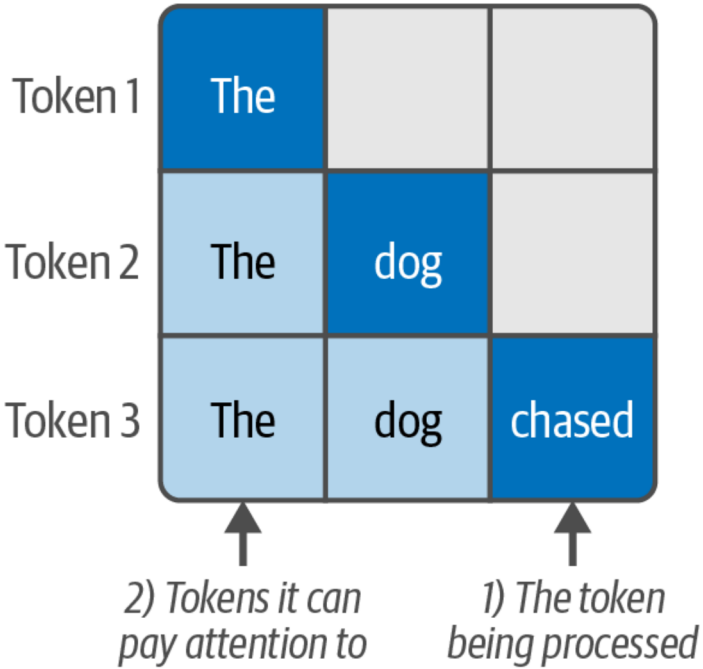
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

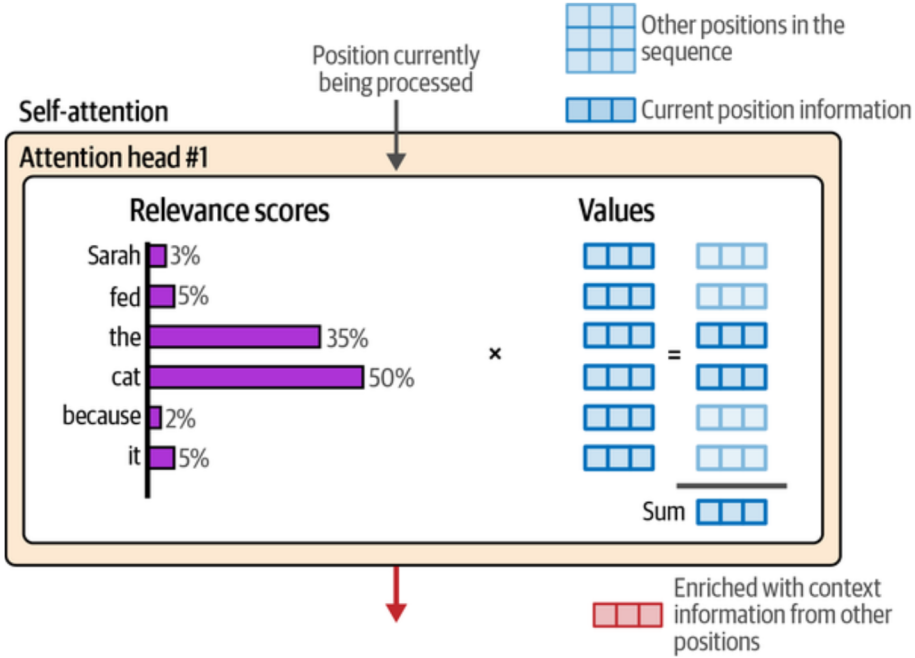




Self-attention: updating representations

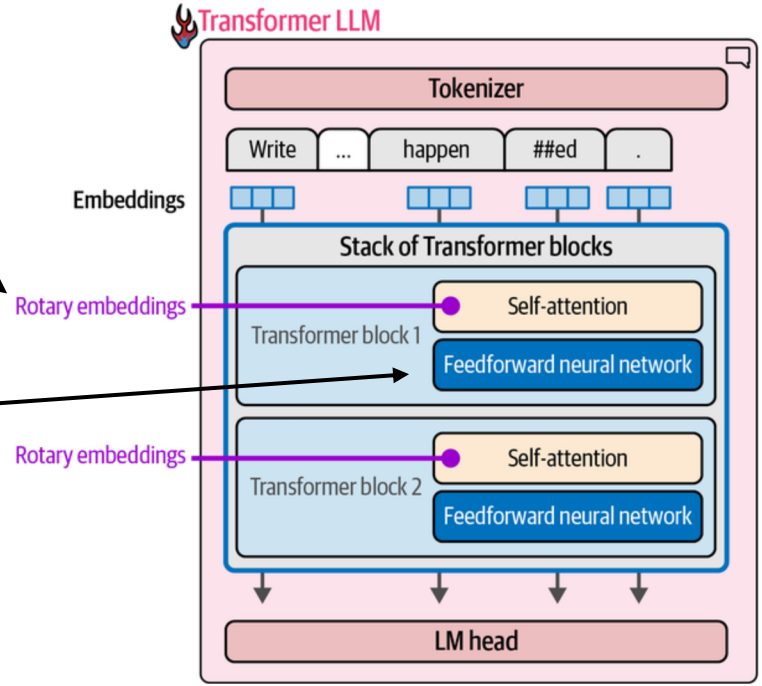
$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$





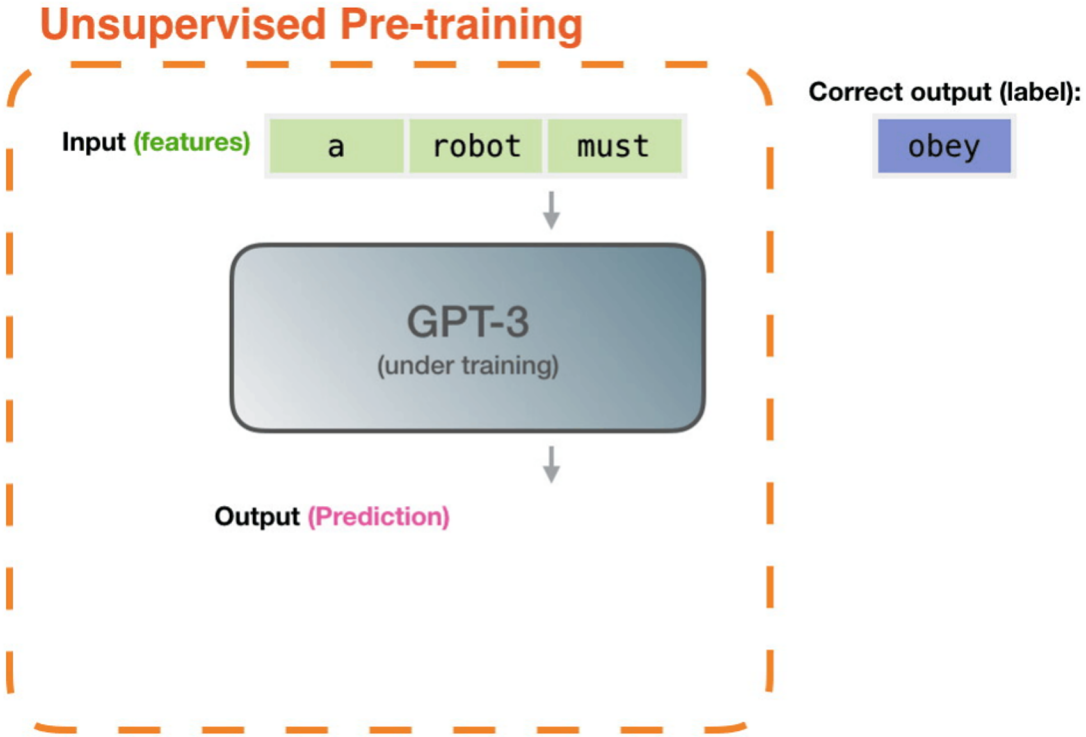
The modern transformer

- Rotary embeddings encode where each token is in the context of the sequence by rotating the vector
- Feedforward neural networks project (compress) the outputs of many attention heads back to the dimensionality of the model



LLM Training

Training process



https://jalammar.github.io/how-gpt3-works-visualizations-animations/

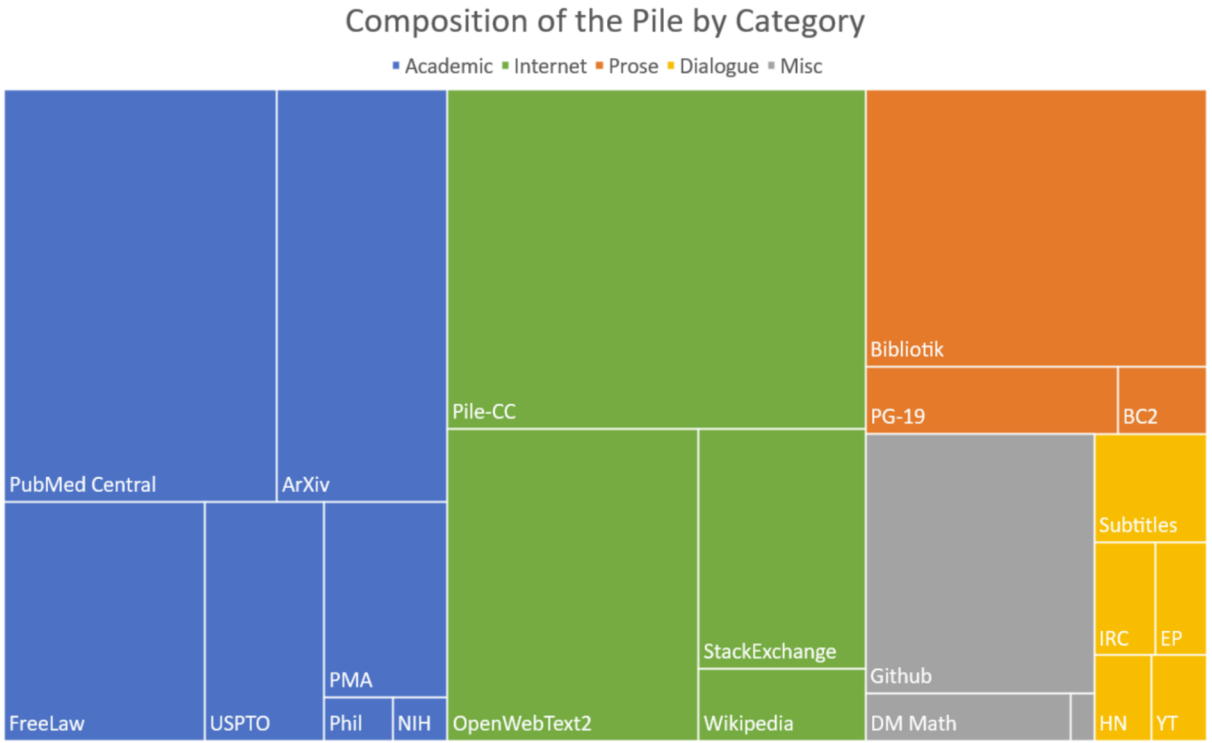
The training loop for LLMs is pretty standard

LLAMA3.1-405b pre-training:

- Task: next word prediction
- Optimizer: AdamW
- Peak learning rate of 8×10^{-5}
- Linear warm up of 8,000 steps
- Cosine learning rate schedule decaying to 8×10^{-7} over 1,200,000 steps.
- Batch size:
  - 4M tokens, 4096 seq length for 262M tokens
  - 8M tokens, 8,192 seq length until 2.87T tokens
  - 16M tokens after 2.87T tokens

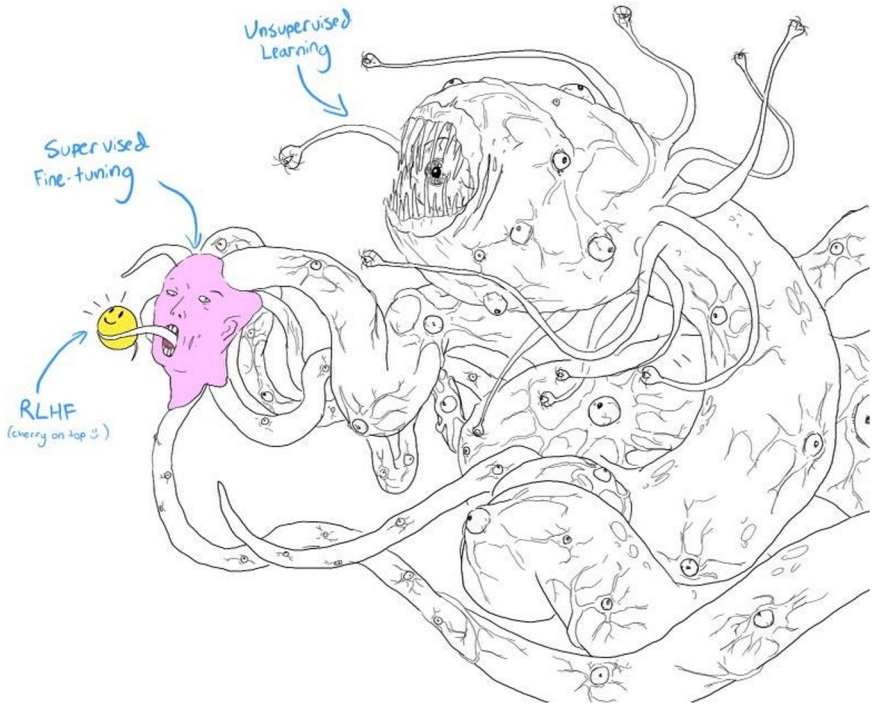
https://arxiv.org/pdf/2407.21783

LLMs are pre-trained on diverse text datasets



https://arxiv.org/pdf/2101.00027

“AI Shoggoth” - LLM training as a meme



https://knowyourmeme.com/photos/2546575-shoggoth-with-smiley-face-artificial-intelligence

Fine Tuning Paradigms

Instruction tuning

Follow instructions

```
{
    “instruction” : “...”,
    “output” : “...”,
}
```

Preference optimization

Follow preferences

```
{
    “instruction” : “...”,
    “preferred” : “...”,
    “rejected” : “...”,
}
```

Leonie
@helloiamleonie

https://x.com/helloiamleonie/status/1823305416635277536

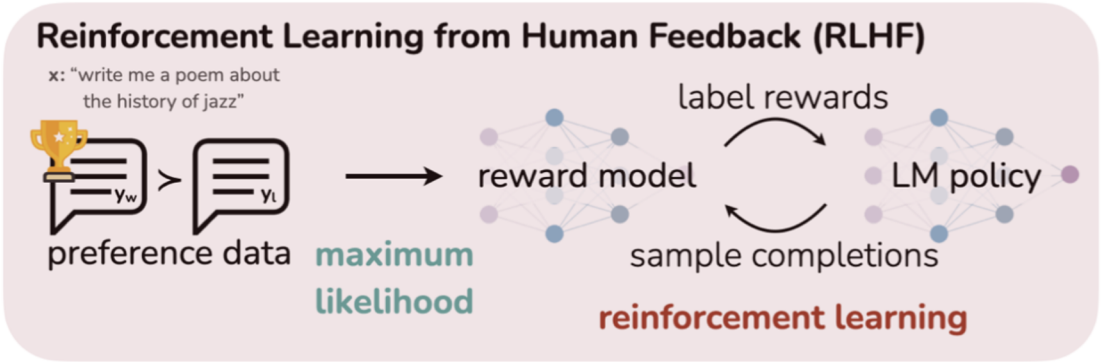
Llama 3.1 SFT data

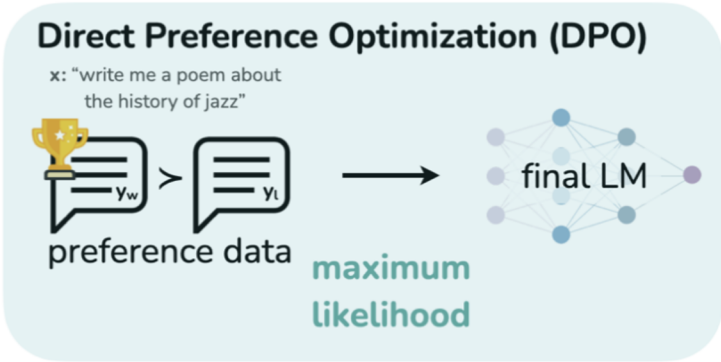
Dataset	% of examples	Avg. # turns	Avg. # tokens	Avg. # tokens in context	Avg. # tokens in final response
General English	52.66%	6.3	974.0	656.7	317.1
Code	14.89%	2.7	753.3	378.8	374.5
Multilingual	3.01%	2.7	520.5	230.8	289.7
Exam-like	8.14%	2.3	297.8	124.4	173.4
Reasoning and tools	21.19%	3.1	661.6	359.8	301.9
Long context	0.11%	6.7	38,135.6	37,395.2	740.5
Total	100%	4.7	846.1	535.7	310.4

Table 7 Statistics of SFT data. We list internally collected SFT data used for Llama 3 alignment. Each SFT example consists of a context (i.e., all conversation turns except the last one) and a final response.

https://arxiv.org/pdf/2407.21783

Preference optimization

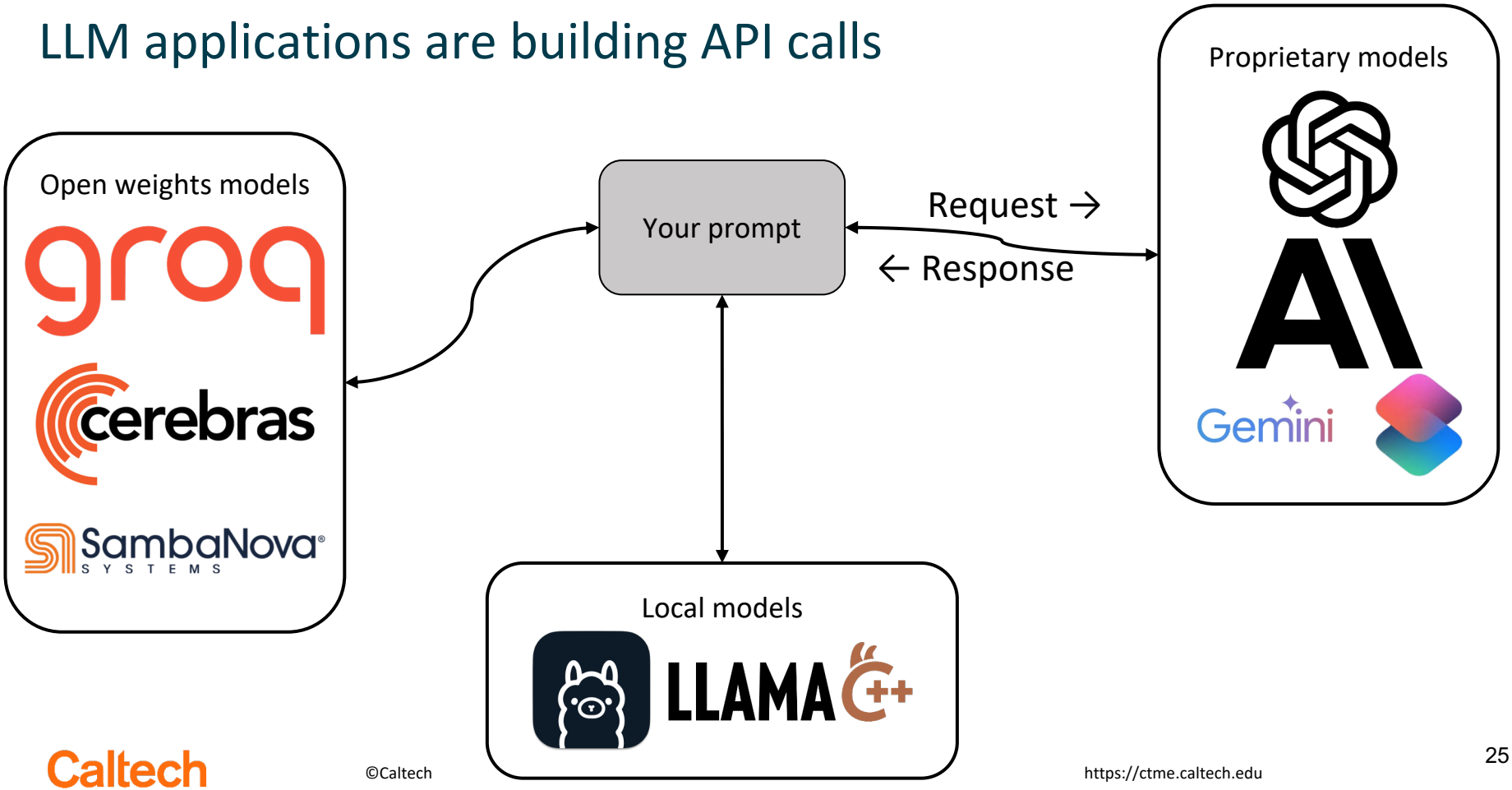




https://huggingface.co/blog/rlhf

Using LLMs

LLM applications are building API calls



Getting today's Colab notebook

- Navigate to github.com/mgfrantz/CTME-llm-lecture-resources
  - > prototyping_ai
  - > 01_getting_started_with_llms.ipynb
  - > 

  - > File > Save a copy in drive

Setting up our OpenAI account

Raise your hand in Zoom when you've finished the following steps:

- Create an OpenAI account (platform.openai.com)
- Add a balance of $5 (settings > billing > add to credit balance)
- Create an API key (settings > API keys > create new secret key)
  - **MAKE SURE TO COPY THIS**
- Paste the API key to Google Colab secrets as OPENAI_API_KEY
  - URL: colab.research.google.com

Components of the API call

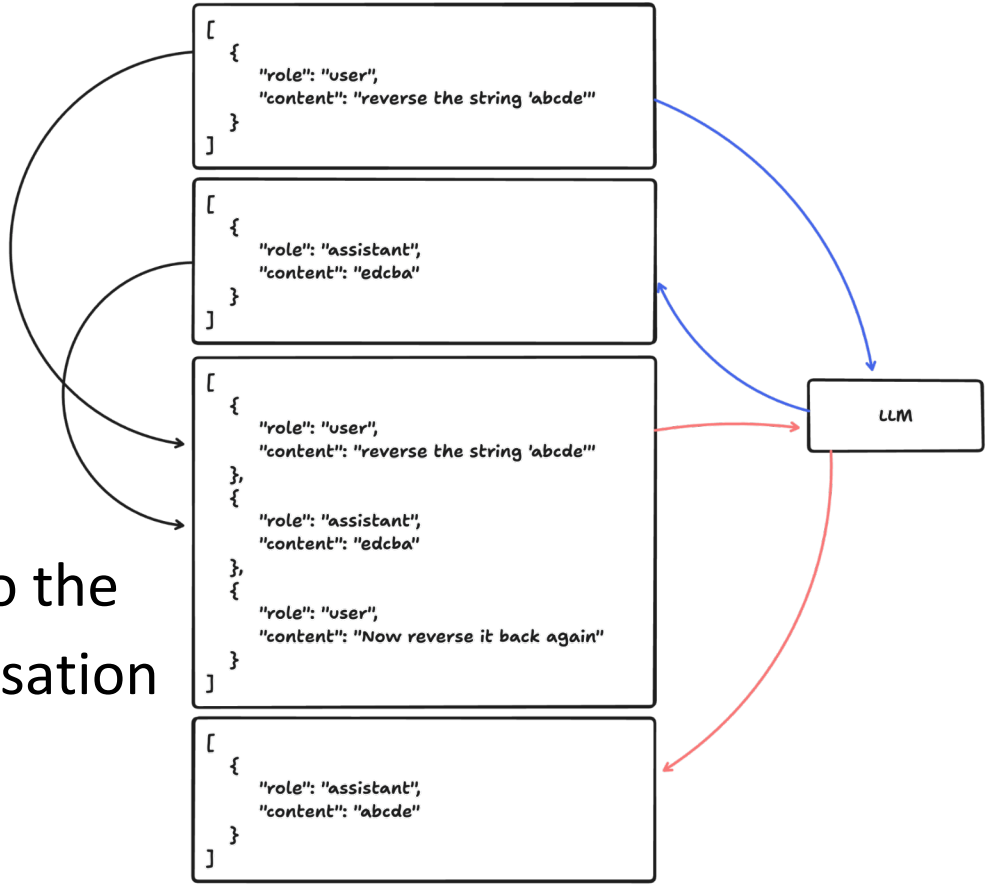
- `model`: Which model to call?
- `messages`: A list of dictionaries with ‘role’ and ‘content’ fields
  - `[{“role”: “user”, “content”: “Hello!”}]`
- Generation hyperparameters:
  - `temperature`
  - `top_p`
  - `n`
  - …

Demo: Make your first API call

Building our first chatbot

Messages

- Messages are a list of dictionaries
- Alternating `user` and `assistant` messages
- Messages are appended to the list to continue the conversation



Roles

- System [optional]
  - Must be the first message
  - Contains relevant context, task definitions, tone/style instructions
- User: Your prompt or chat message
- Assistant: LLM’s response

Streaming responses [bonus]

- Set `stream=True` in your LLM call
- Returns a python generator instead of a response object
- Useful if you want to display tokens as they are generated

Exercise: Building a basic chatbot with gradio

How text is generated

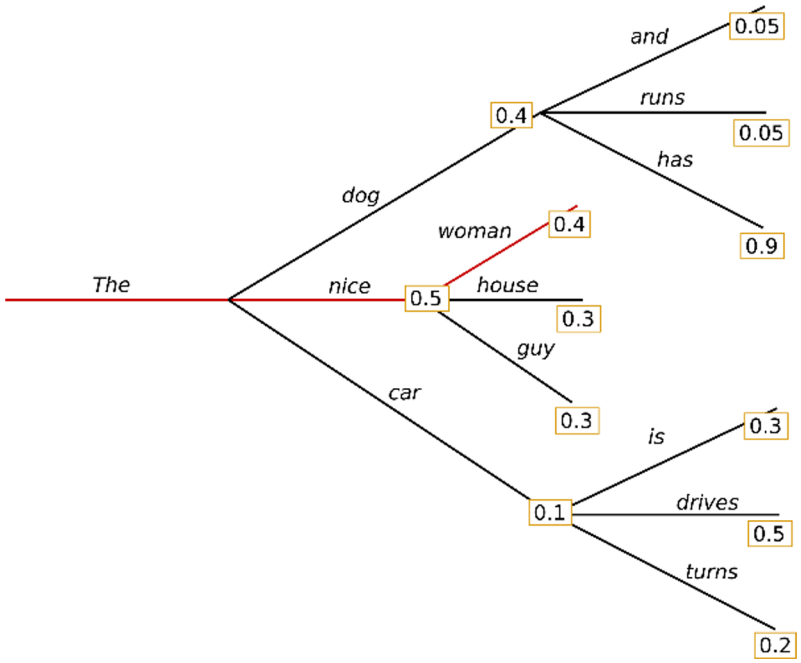
Greedy decoding

Definition: $w_t = argmax_w P(w|w_{1:t-1})$

Explanation: At each time t, we choose the word w with the highest probability given all the previous words $w_{1:t-1}$

https://huggingface.co/blog/how-to-generate

Greedy decoding



Input: “The”

$argmax_w P(w|\text{“The”}) = \text{“nice”}$

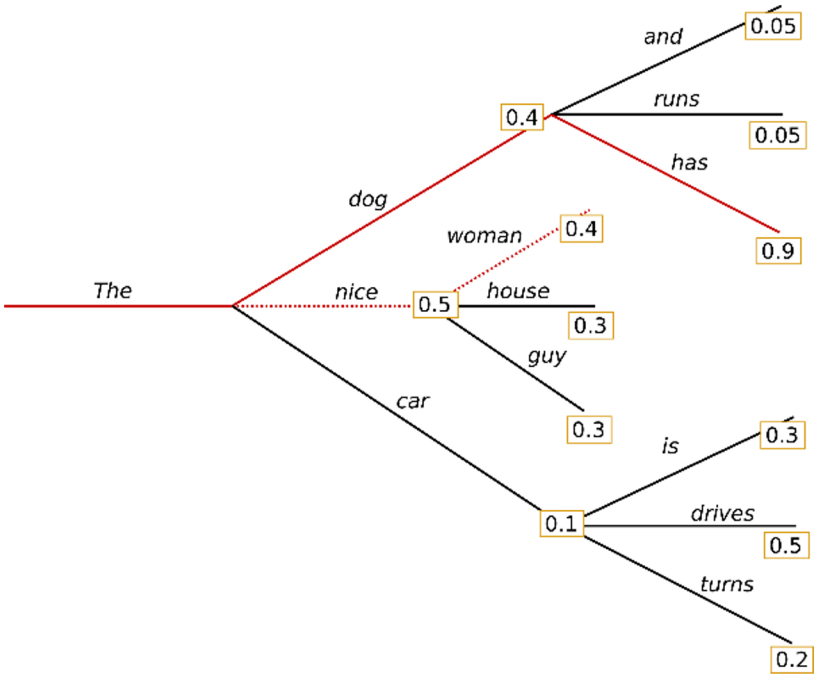
$argmax_w P(w|\text{“The nice”}) = \text{“woman”}$

Output: “The nice woman”

$$P(W) = p(\text{“nice”}) * p(\text{“woman”}|\text{”The nice”})$$
$$= 0.5 * 0.4$$
$$= 0.2$$

https://huggingface.co/blog/how-to-generate

Beam search



n_beams = 2

“the nice” = 0.5

“the dog” = 0.4

“the car” = 0.1

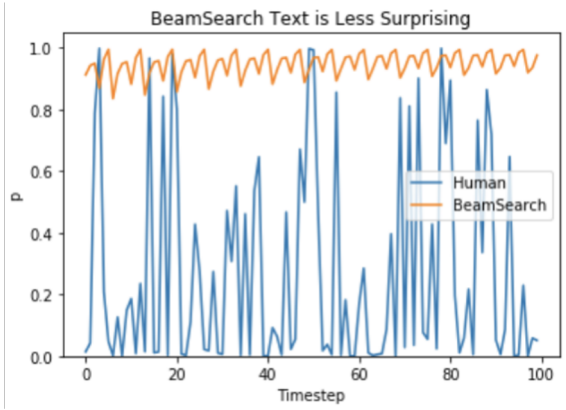
“the nice woman” = 0.5 * 0.4 = 0.2

“the dog has” = 0.4 * 0.9 = 0.36

https://huggingface.co/blog/how-to-generate

Issues with beam search

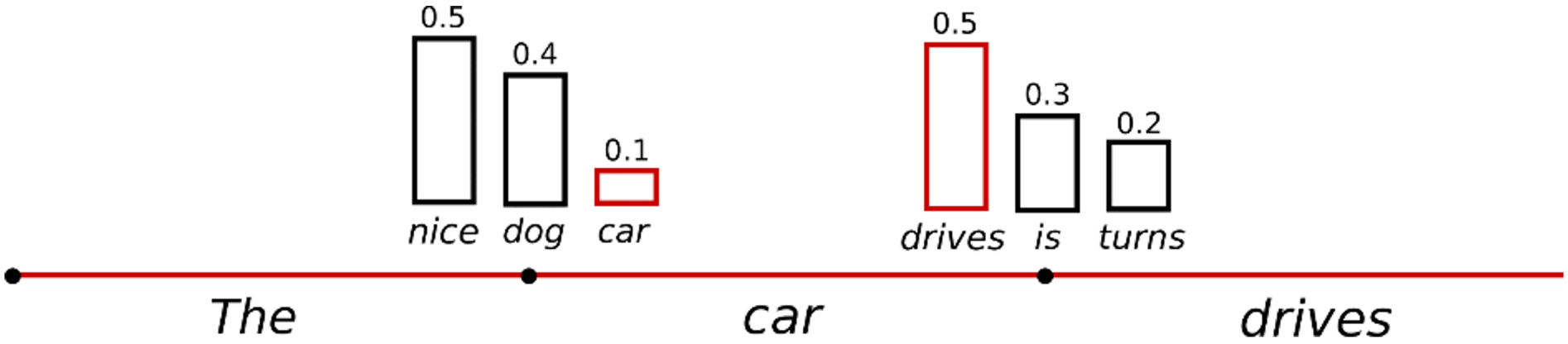
- Repetition can be addressed with no_repeat_ngram_size hyperparameter.
- Beam search can result in awkward text.



https://huggingface.co/blog/how-to-generate, https://arxiv.org/abs/1904.09751

Sampling

- Model as $w_t \sim P(w|w_{1:t-1})$
- Each decoding step is a probability distribution over all tokens



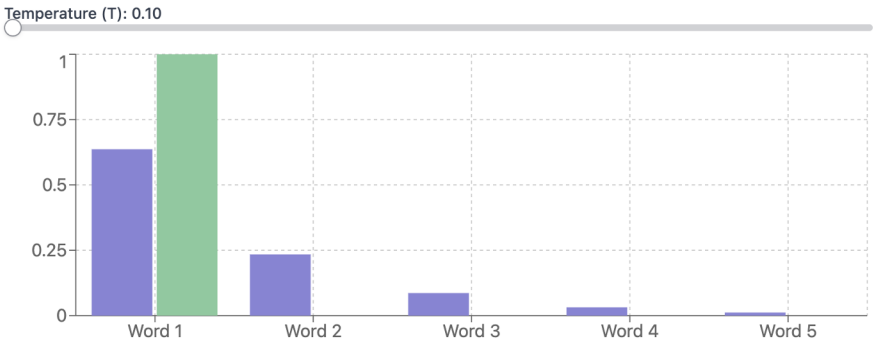
https://huggingface.co/blog/how-to-generate

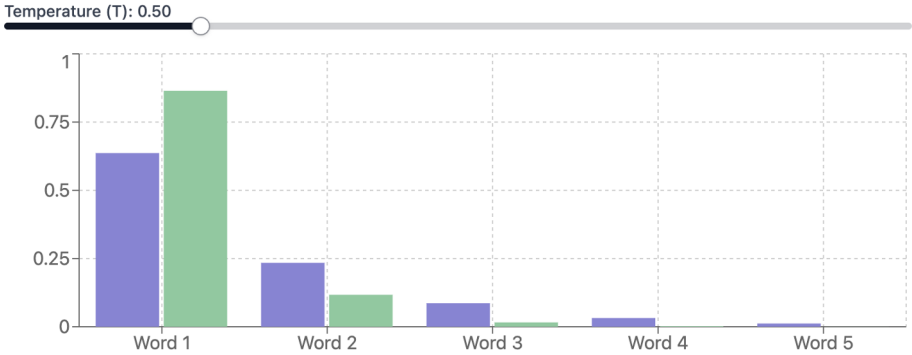
Sampling issues

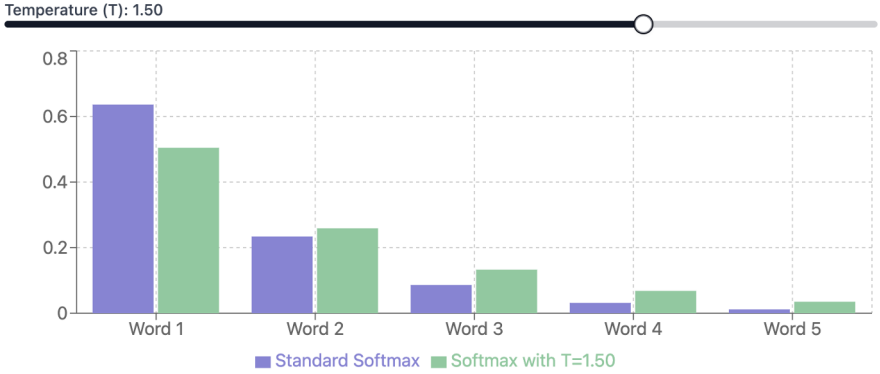
- Models output gibberish due to low probabilities over even the most likely token.
- Solution: Hyperparameter `temperature` T
- Standard Softmax: $P(w_i) = \exp(z_i) / \Sigma \exp(z_j)$
- Softmax with Temperature: $P(w_i) = \exp(z_i/T) / \Sigma \exp(z_j/T)$

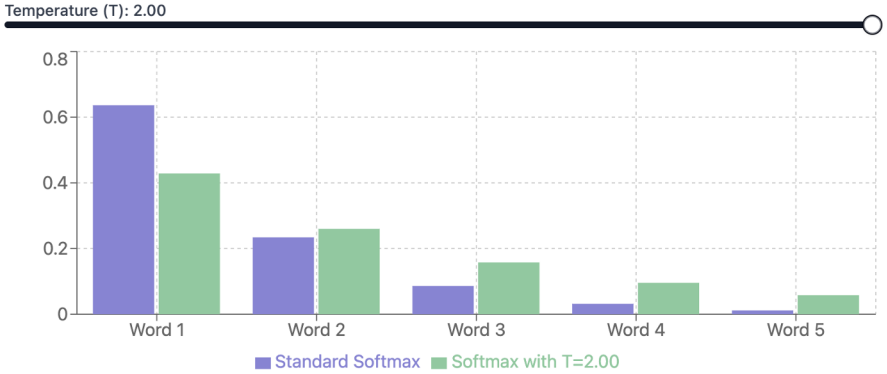
https://huggingface.co/blog/how-to-generate

Comparing softmax with and without temperature



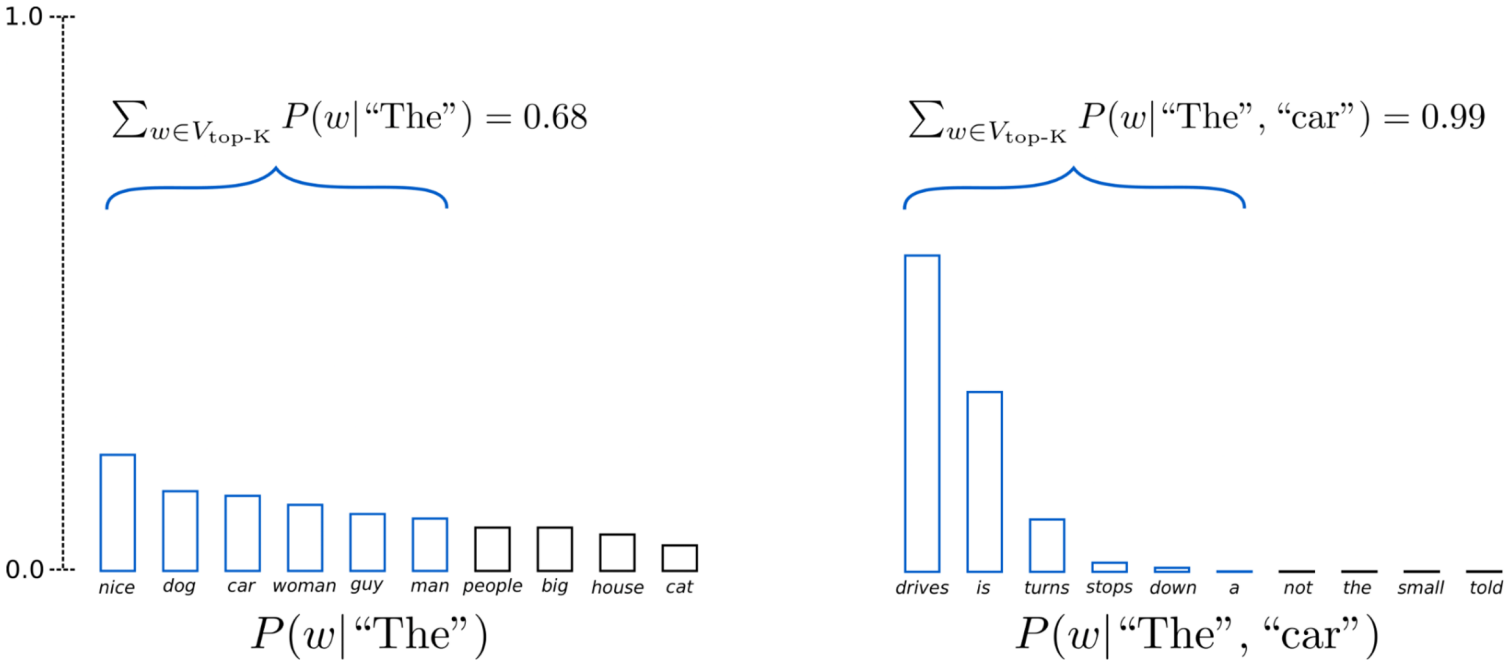






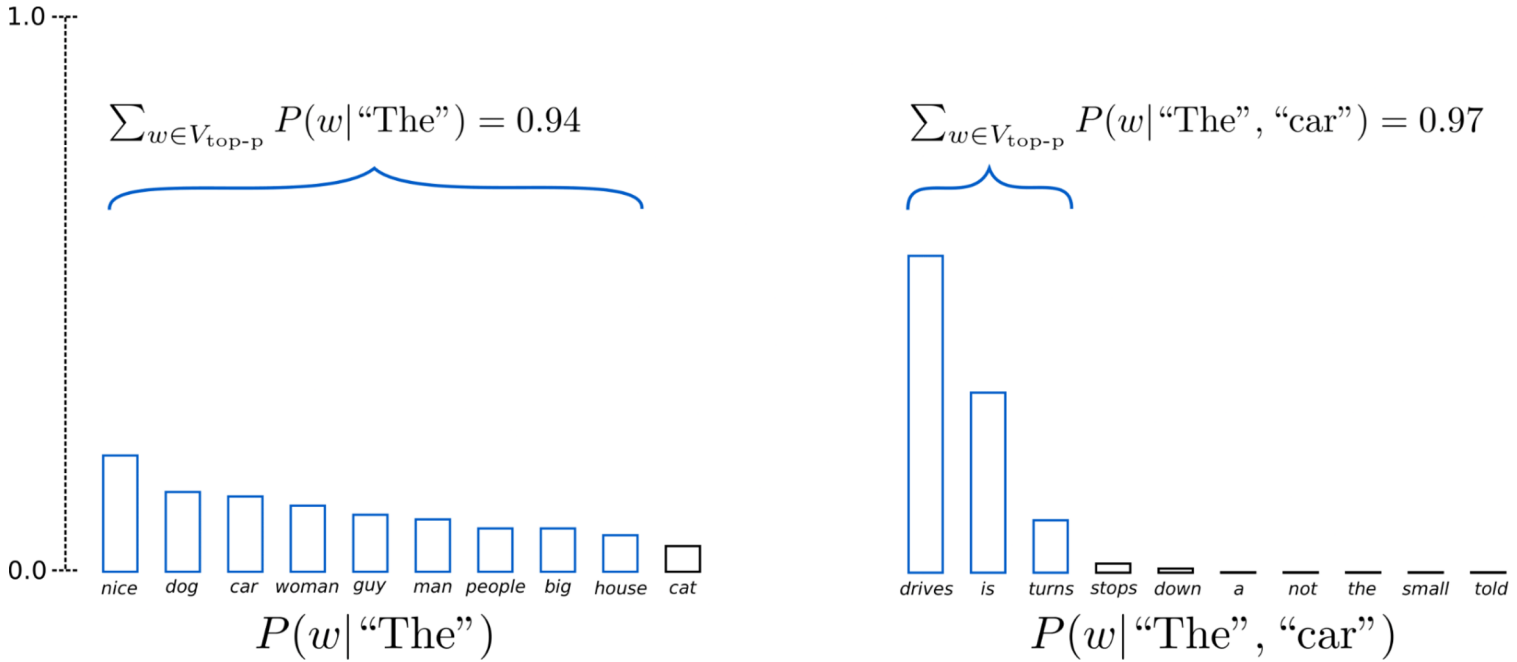
Made with claude.ai (artifacts)

Top-K Sampling



https://huggingface.co/blog/how-to-generate

Top-P (nucleus) Sampling



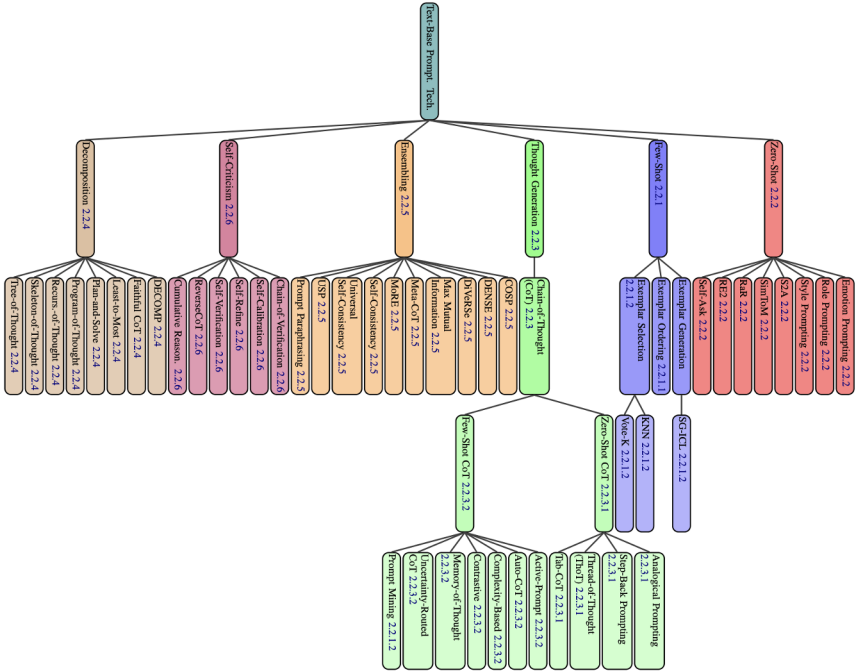
https://huggingface.co/blog/how-to-generate

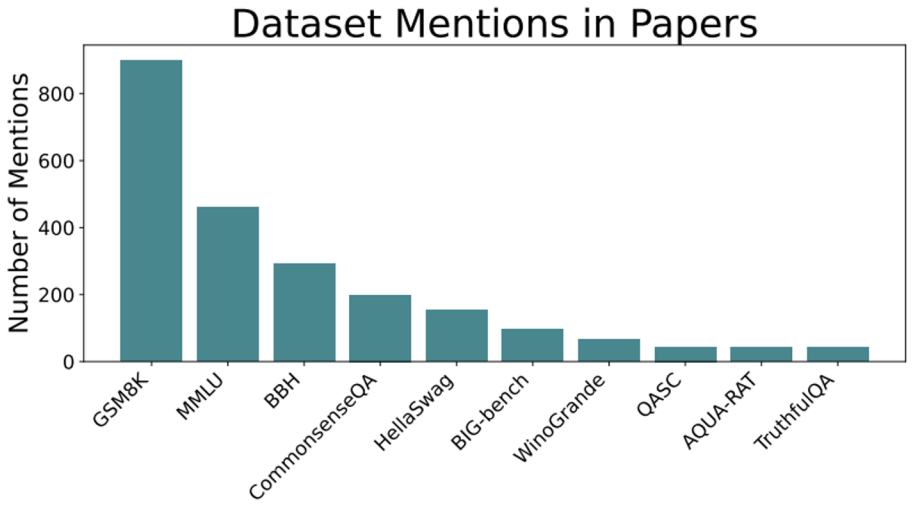
Demos: Text generation hyperparameters

Exercise: Chatbot with generation hyperparameters

Prompt engineering

Prompt engineering is a HUGE field





Agenda:

- Zero-shot prompting (with structured outputs)
- Few-shot prompting
- Embeddings and vector stores
- Dynamic few-shot prompting

High-level libraries

LangChain

Basic example: calling LLMs

```python
# Create the messages (same as above)
messages = [
    {"role": "user", "content": "Say: This is a test!"},
]

# Make the API call
chat_completion = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=messages,
    # stream=True
)
```

```python
# Create the model object
llm = OpenAI(model="gpt-4o-mini")
# Create the messages
messages = [
    ChatMessage(role='user', content="Say: This is a test!")
]
# Call the model
llm.chat(messages)
```

Starting simple: zero-shot

Example: sentiment analysis -> [positive, negative, neutral]

Prompt:

```
prompt_template = "Classify the sentiment of the
following text as positive, negative, or neutral. \
Text: \
{text} \
Answer: "
prompt = prompt_template.format(text="I loved this
movie!")
sentiment = call_llm(prompt)
```

Note: Pydantic and structured outputs

- Most LLMs have been fine-tuned to generate structured data
- Bad way: beg the model to output JSON
- Hard way: make a very nested dictionary with lots of fields describing each attribute
- Best way: Create a Pydantic class and pass it to the model

```python
from pydanic import BaseModel, Field

class MyClass(BaseModel):
    my_attribute: type = Field(..., description="Description of my attribute")
```

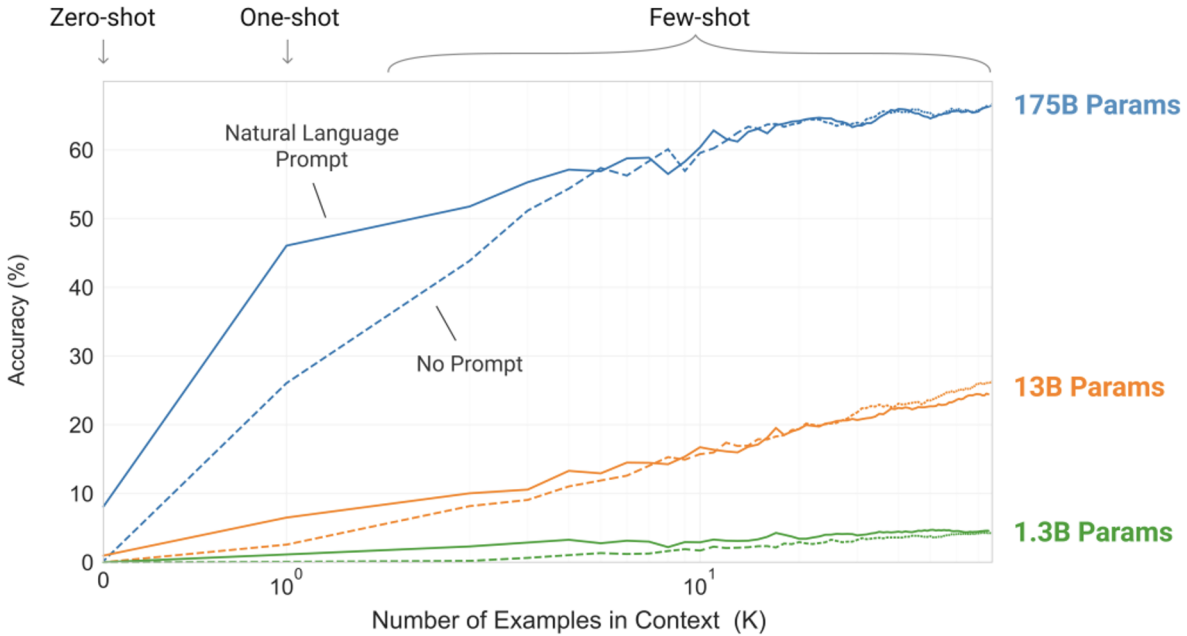
Demo: Zero-shot prompting

Few-shot prompting

Few-shot learning

```
prompt_template = “Classify the sentiment of the following text as positive,
negative, or neutral. \
Examples:

Text: The food was delicious
Answer: positive

Text: I did not enjoy the scenery
Answer: negative

Text: Go straight then turn left at the stop sign
Answer: neutral

Text: {text}
Answer: ”
prompt = prompt_template.format(text=”I loved this movie!”)
sentiment = call_llm(prompt)
```

Language models are few-shot learners (2020)



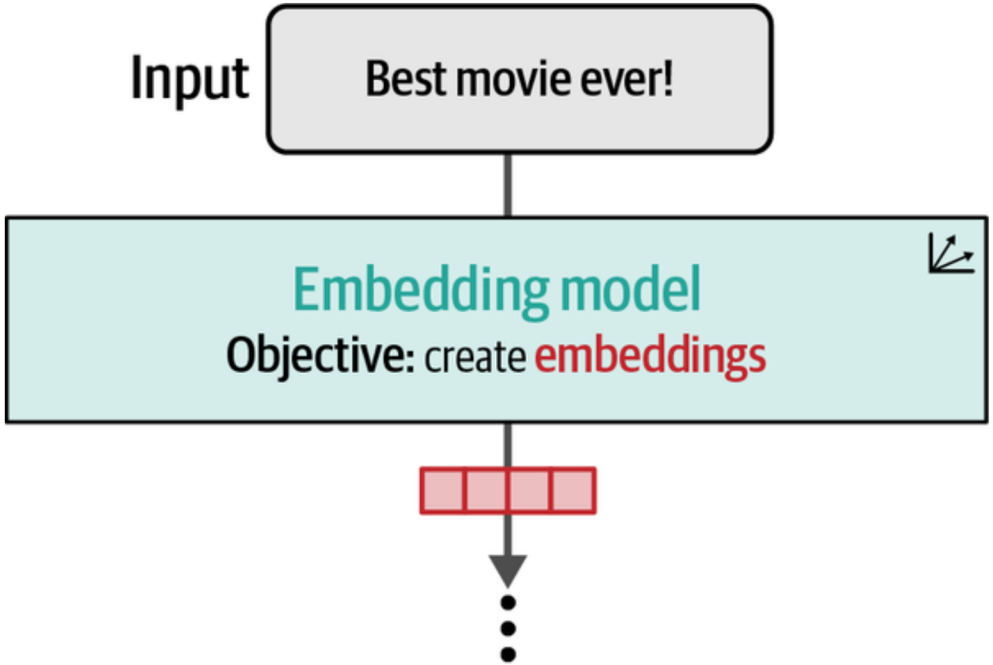
Model Name	n_{params}
GPT-3 Small	125M
GPT-3 Medium	350M
GPT-3 Large	760M
GPT-3 XL	1.3B
GPT-3 2.7B	2.7B
GPT-3 6.7B	6.7B
GPT-3 13B	13.0B
GPT-3 175B or "GPT-3"	175.0B

Language models are few-shot learners

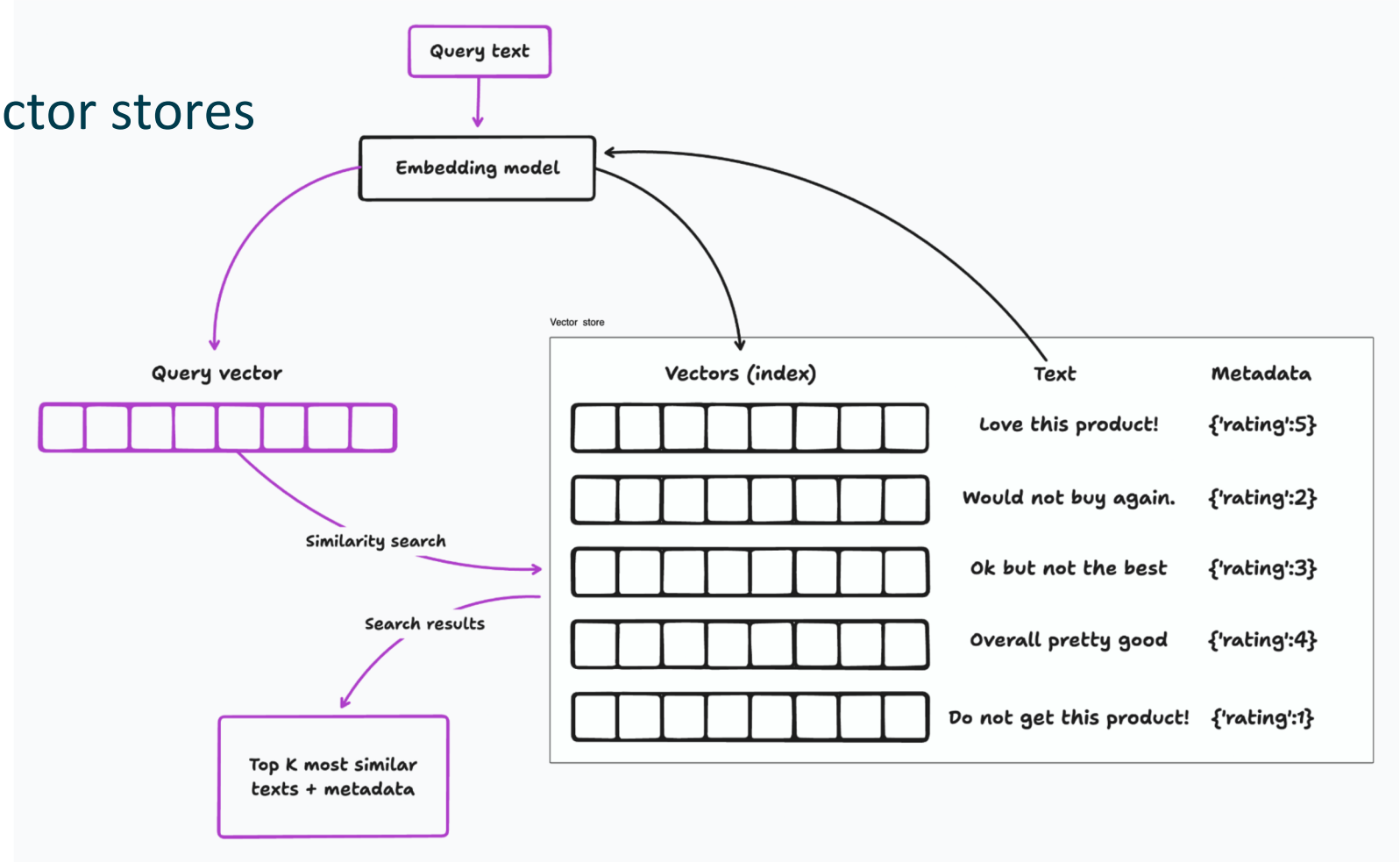
Demo: Few-shot prompting

Embeddings and vector stores

Embeddings


Input
Best movie ever!
Embedding model
Objective: create embeddings

Vector stores



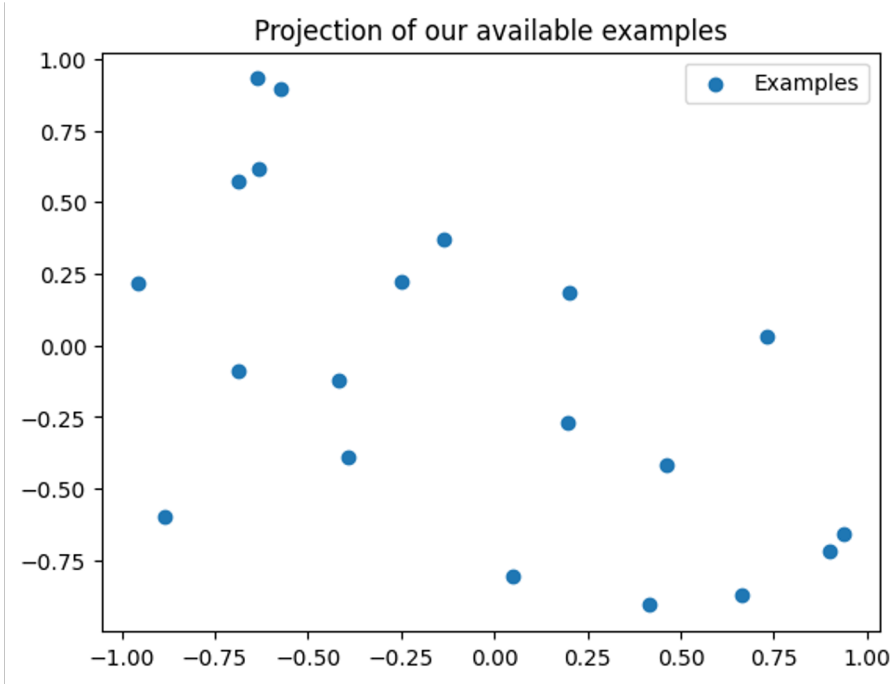
Cosine similarity

Demo: Embeddings and vector stores

Dynamic few-shot prompting

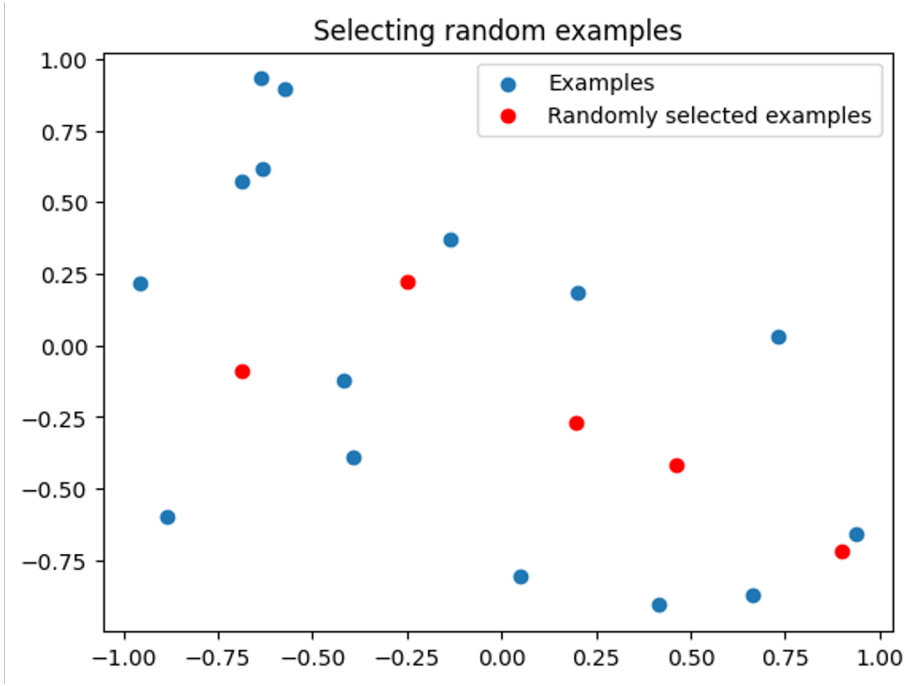
Sampling examples

- The figure 👉 is a 2-d projection of “embedded” (text -> vector) examples.



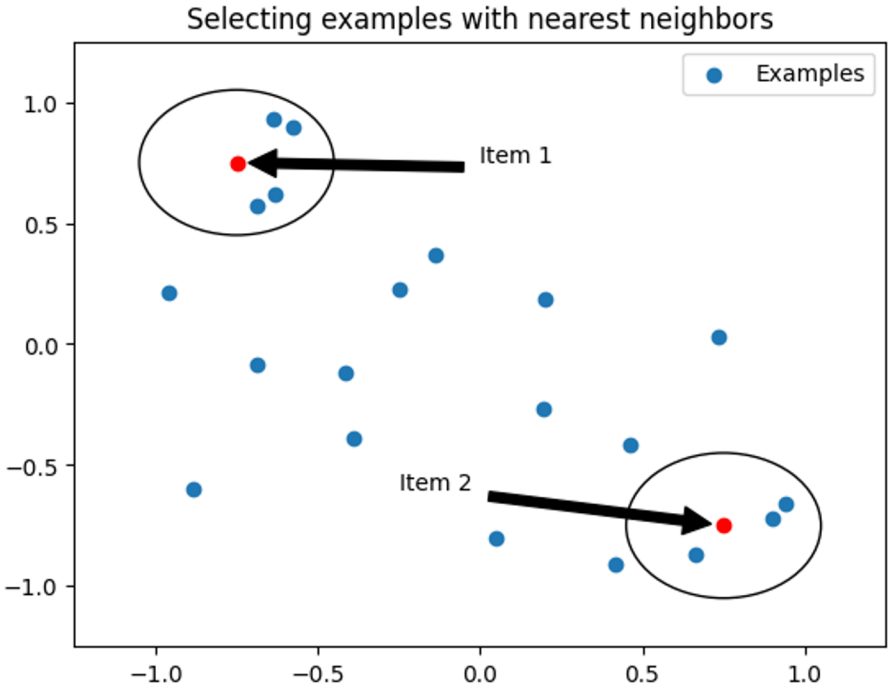
Sampling examples

- The figure 👉 is a 2-d projection of “embedded” (text -> vector) examples.
- We can **randomly select examples** from this distribution to populate our prompts



Sampling examples

- The figure 👉 is a 2-d projection of “embedded” (text -> vector) examples.
- We can randomly select examples from this distribution to populate our prompts
- We can be even smarter by also embedding our current example and finding the most similar examples to best illustrate the task using **nearest neighbors** (dynamic few-shot learning)



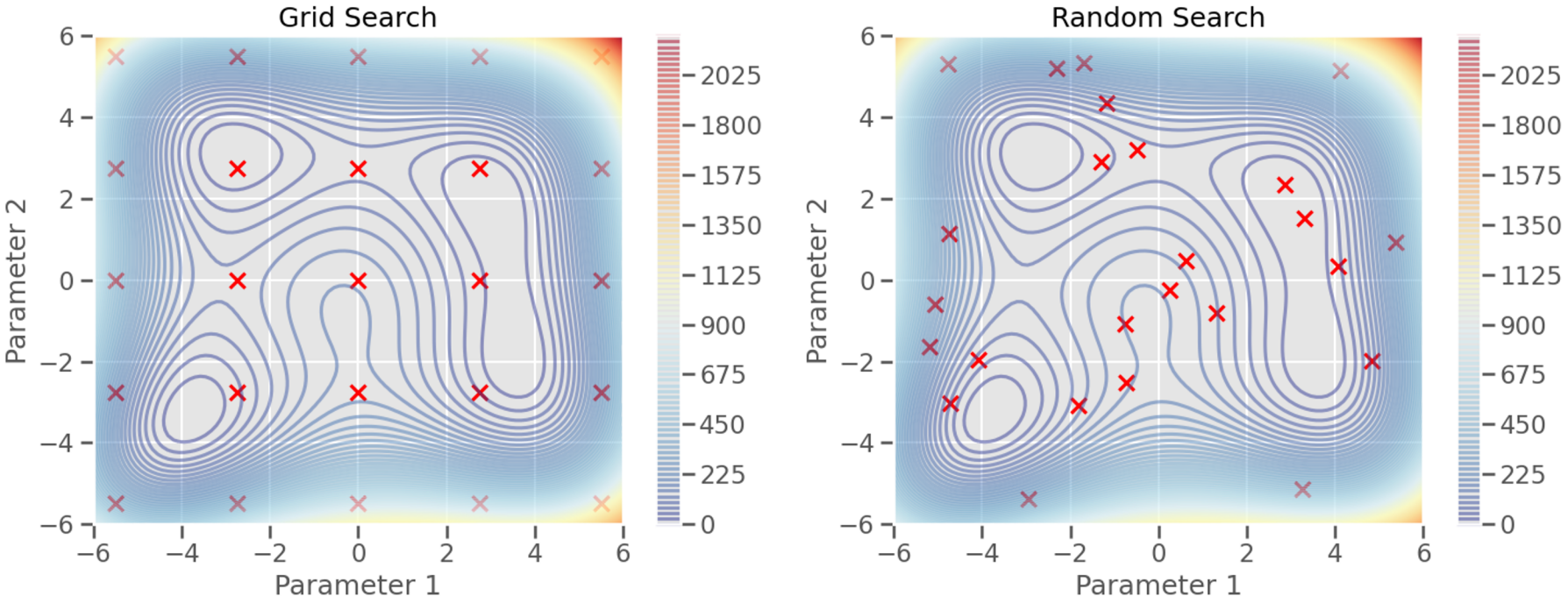
Dynamic few-shot learning

```
relevant_examples = vectorstore.retrieve(text)

prompt_template = “Classify the sentiment of the following text
as positive, negative, or neutral. \
Examples:

{relevant_examples}

Text: {text}
Answer: ”
prompt = prompt_template.format(
    text=”I loved this movie!”,
    relevant_examples=relevant_examples
)
sentiment = call_llm(prompt)
```

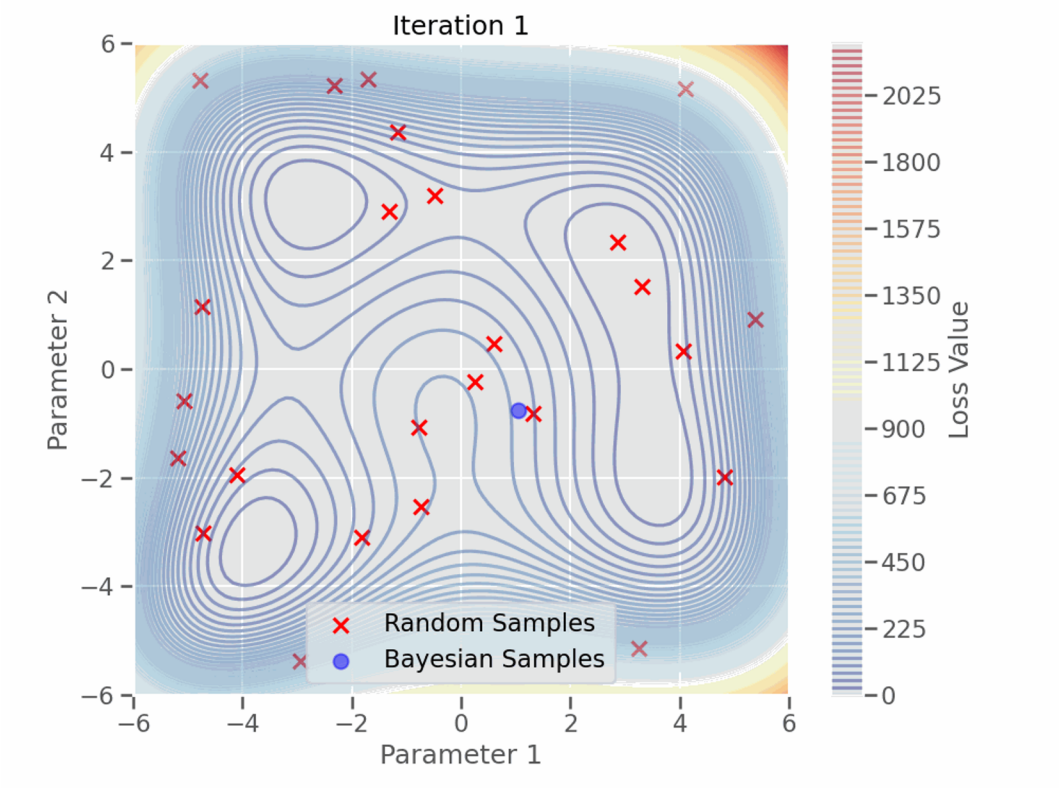
Exercise: Dynamic few-shot prompting

Bonus: Prompt optimization

Traditional hyperparameter search



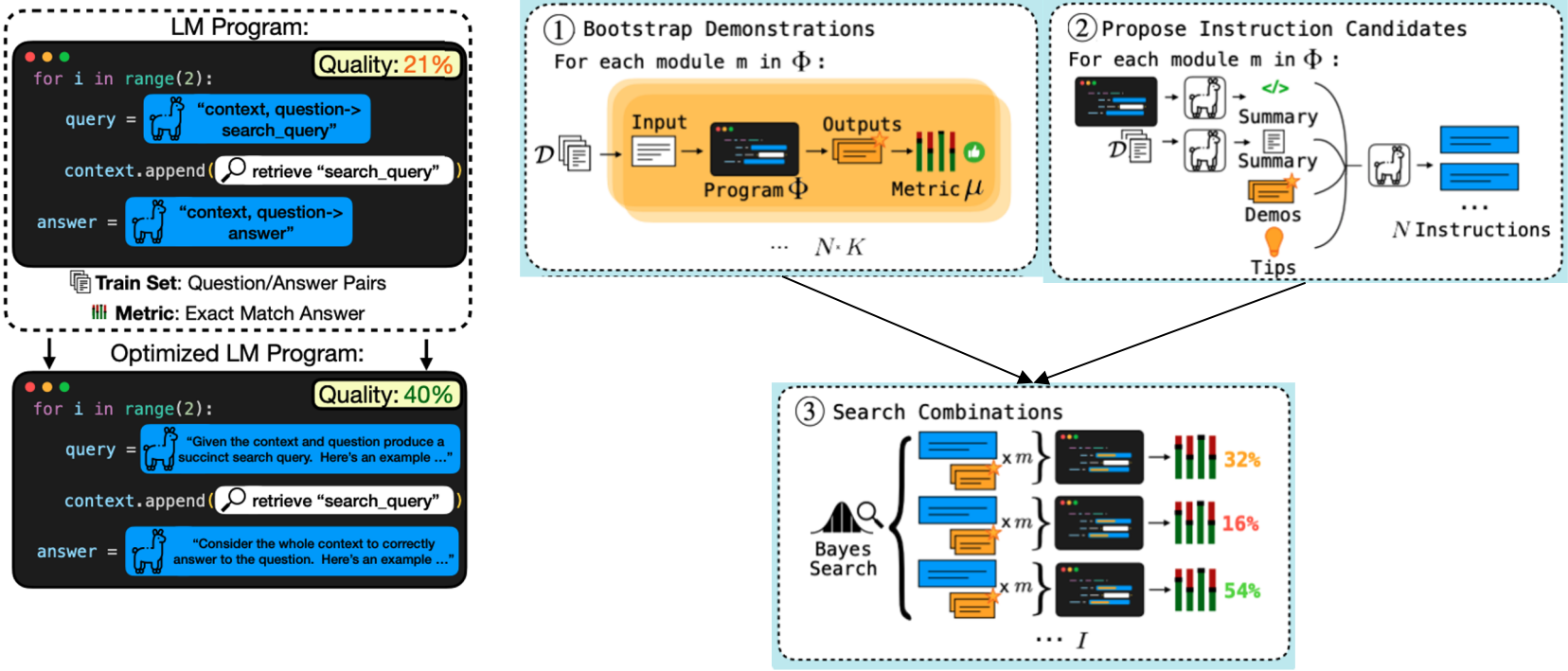
Bayesian Optimization



DSPy: Demonstrate, Search, Predict

- Create LLM programs in a torch-like API
- Create evals for your programs
- Generate and/or find optimal samples from an LLM or your training data
- Generate and search over different instructions

Prompts are hyperparameters: MIPRO



https://arxiv.org/pdf/2406.11695

Bonus exercise: LLM optimization