Choosing An LLM

For this section of the project, I suggest you do some research (using ChatGPT, Gemini, etc.) on different models provided by Hugging Face. The starter code that you've been given uses Qwen2.5, but you can easily change which model is used by changing the model name.

```
# Define the name of the model we want
model_name = "Qwen/Qwen2.5-1.5B-Instruct"

# Load the model weights (if you use a bigger model, add quantization_config=bnb_config)
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto")

# Get the specific tokenizer for this model
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

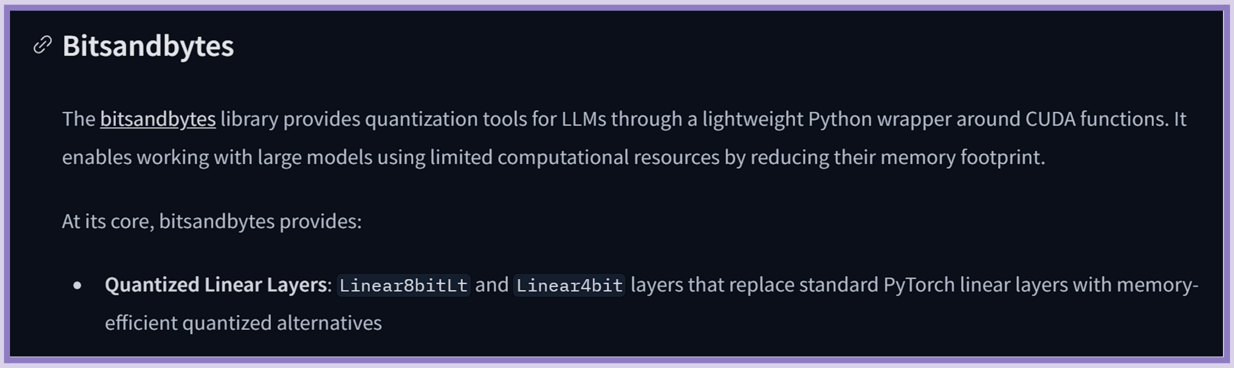
I suggest exploring different options provided by Hugging Face in order to gain a richer understanding of how models work. When asked about this project in the future, you can talk about why you chose a particular model and specific challenges you ran into with that model.

Here are some reasons why I choose Qwen2.5. When researching models to use for this mini-project, you might want to look for models that share some of these traits.

If you look on Hugging Face, you may notice that there are different variations of Qwen2.5. There's versions with 5 billion, 7 billion, 14 billion, and even 72 billion parameters. While larger models are generally more capable than smaller models, they are also much more expensive to run. Since we are using Colab, I'd generally recommend using models that have around 3 billion parameters if you don't want to wait very long for the model to run.

Sweet Spot ≤ 3 Billion Parameters

However, you can still use a model with over 3 billion parameters. If you plan on using a larger model, I suggest you look at using a bitsandbytes configuration to make the model easier to run. This allows you to load the model weights in 4-bit instead of 32-bit, which reduces memory usage and speeds up inference. This should only take a few lines of code.

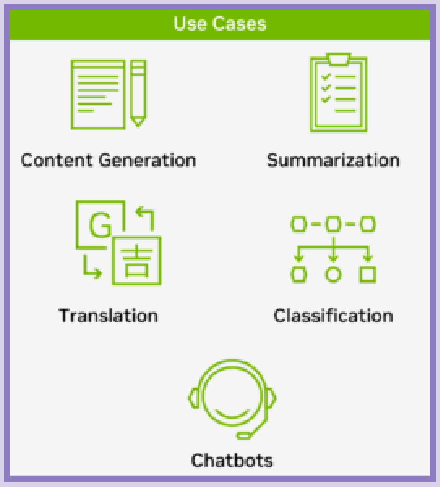

Bitsandbytes

The bitsandbytes library provides quantization tools for LLMs through a lightweight Python wrapper around CUDA functions. It enables working with large models using limited computational resources by reducing their memory footprint.

At its core, bitsandbytes provides:

- **Quantized Linear Layers:** `Linear8bitLt` and `Linear4bit` layers that replace standard PyTorch linear layers with memory-efficient quantized alternatives

Another reason why I chose Qwen2.5 is because it's designed for human dialogue. One mistake I made when I first started using models on Hugging Face was thinking that every model would act like ChatGPT and would be optimized for interactive conversations. However, not all models are designed with the goal of human dialogue in mind.



Some models might be designed to understand text for tasks like classification or translation instead of generating natural dialogue.

Some models might be more domain specific and could be optimized for topics like math, coding, etc.

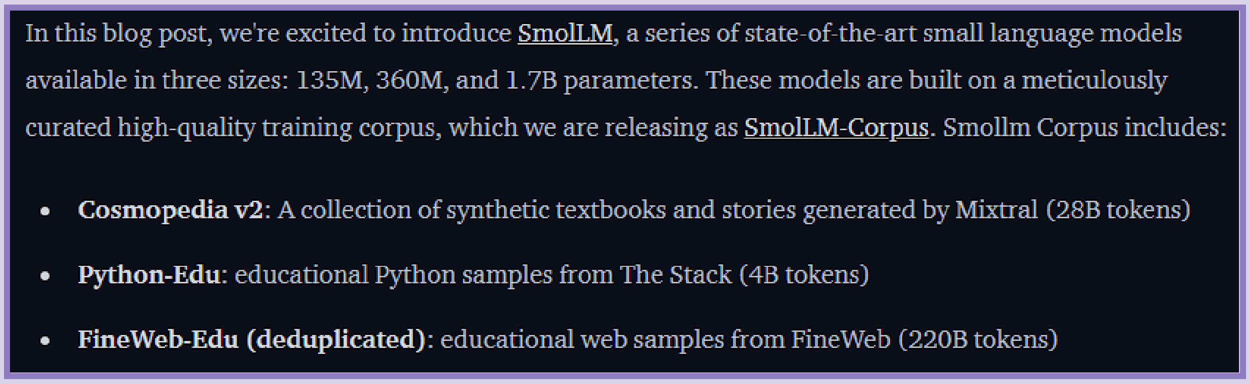
When I first did this mini-project, I tried using Deepseek R1 and SmolLM, but those models weren't producing good quality responses. While it is difficult to know exactly why a model isn't behaving correctly, I still think it's valuable to think of potential reasons for why a model is behaving a certain way. Even if your hypotheses aren't always correct, being aware of these factors can guide better model selection in future projects.

I tried using Deepseek R1 because of its strong reasoning skills and ability to understand long passages. However, one reason why it might've performed poorly for me could be because of my topic and the nature of the questions asked.

While it is true that Deepseek R1 performs very well on reasoning benchmarks, that doesn't mean it'll be good at answering questions about a movie. It might be optimized to solve structured word problems that require stringing together a lot of facts to produce a solution rather than interpreting narrative content and understanding story nuances.

I also tried using SmolLM because of how lightweight it is and its ability to process long passages.

According to a blog post for SmolLM, it seems like it does have some exposure to narrative-like content through Cosmopedia v2, so what could be the issue?



In this blog post, we're excited to introduce SmolLM, a series of state-of-the-art small language models available in three sizes: 135M, 360M, and 1.7B parameters. These models are built on a meticulously curated high-quality training corpus, which we are releasing as SmolLM-Corpus. Smollm Corpus includes:

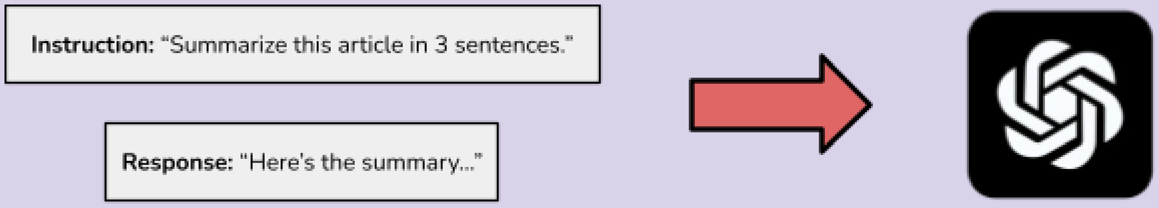
- **Cosmopedia v2**: A collection of synthetic textbooks and stories generated by Mixtral (28B tokens)
- **Python-Edu**: educational Python samples from The Stack (4B tokens)
- **FineWeb-Edu (deduplicated)**: educational web samples from FineWeb (220B tokens)

This is where the difference between base models and instruction tuned models could come in.

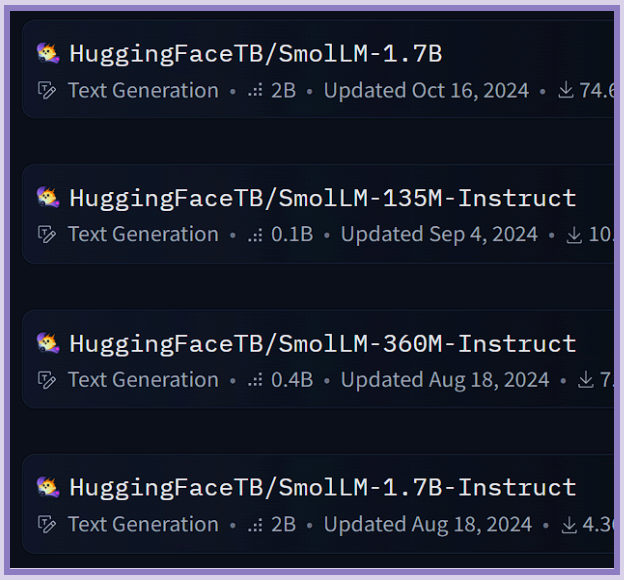
* **Note**: Just because I didn't get good quality responses with these models, that doesn't mean that you can't. Perhaps these models weren't a good fit for the task or I could've structured my prompt differently. Try testing different things and seeing for yourself.

Instruction-Tuning Models

Instruction tuning is a process where a model is fine-tuned on datasets that look like instructions + responses, rather than just raw text (like a wikipedia page or textbook).


Instruction: "Summarize this article in 3 sentences."
Response: "Here's the summary..."

By letting the model train on these examples, we are showing it how to follow user instructions and helping it generate more useful responses. This is the difference between a model that just writes text and a model that acts like a helpful assistant.



HuggingFaceTB/SmolLM-1.7B
Text Generation • 2B • Updated Oct 16, 2024 • 74.

HuggingFaceTB/SmolLM-135M-Instruct
Text Generation • 0.1B • Updated Sep 4, 2024 • 10

HuggingFaceTB/SmolLM-360M-Instruct
Text Generation • 0.4B • Updated Aug 18, 2024 • 7

HuggingFaceTB/SmolLM-1.7B-Instruct
Text Generation • 2B • Updated Aug 18, 2024 • 4.3

On Hugging Face, some models have a base version as well as an instruction tuned version. This might be useful when determining which model you want to use.

* *I didn't try using the instruction tuned version of SmolLM-1.7B. If you use it, let me know how it is :)*

Designing Prompt

Now that you've selected a model to use, we can begin designing the prompt we'll use to feed in the information. There isn't a single 'right' way to create a prompt, but I suggest using a LLM like ChatGPT or Gemini for inspiration to experiment with phrasing and structure.

My code for this mini-project uses the structure shown below:

```
Using only the provided text, answer this question about the Your Name anime. Give
only the answer in at most 5 sentences. Do not add extra explanation.

Question: {QUESTION}

Text:
{PARAGRAPH 1}
{PARAGRAPH 2}
{PARAGRAPH 3}
{PARAGRAPH 4}
{PARAGRAPH 5}

Answer:
```

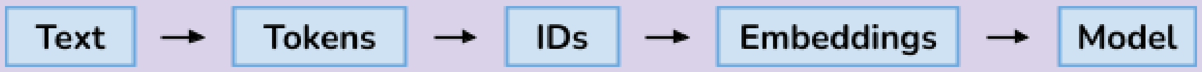
This isn't the only or best way for you to structure your prompt. There are valid reasons to put the text before the question or directions. I recommend exploring different options.

Purpose Of Tokenizer

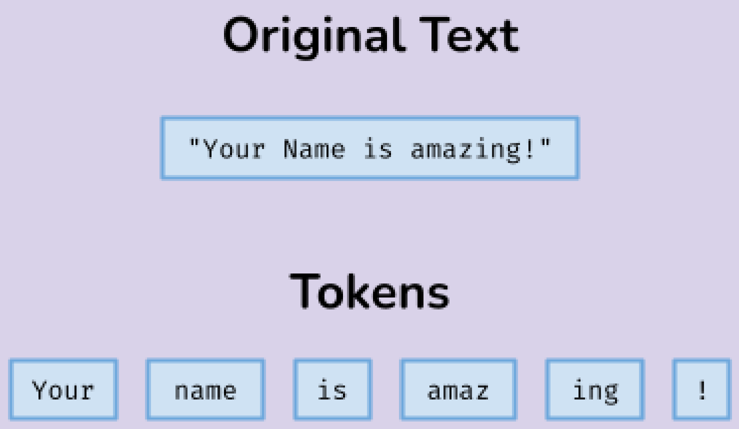
When loading up the model, you may have noticed this code in the same cell:

```
# Get the specific tokenizer for this model
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

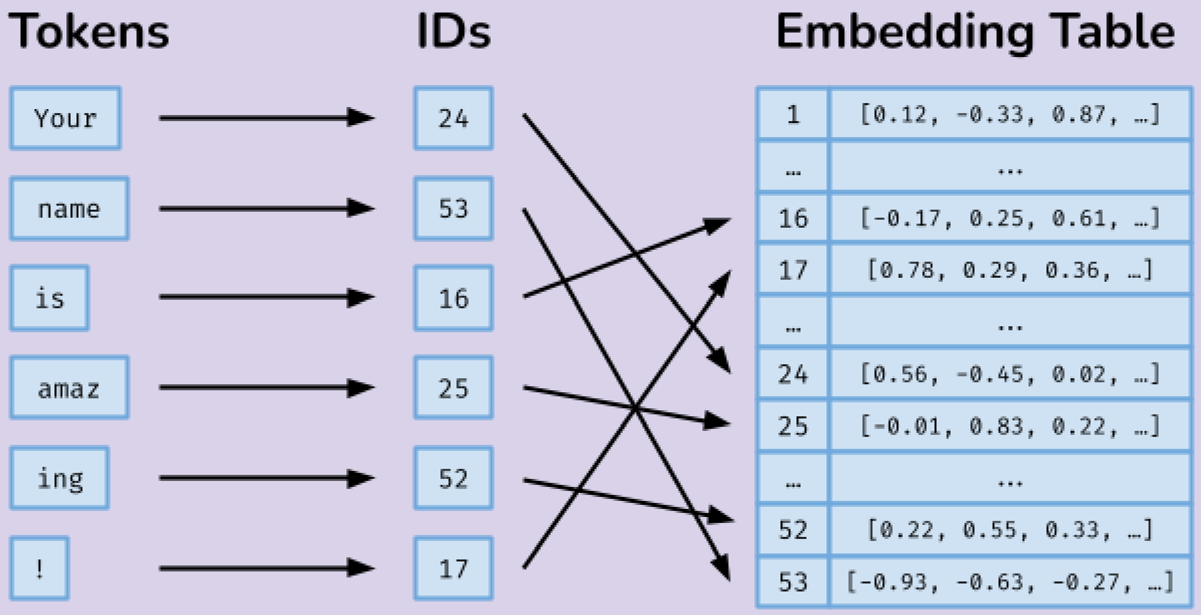
A tokenizer is like a translator between human language and what the model understands.



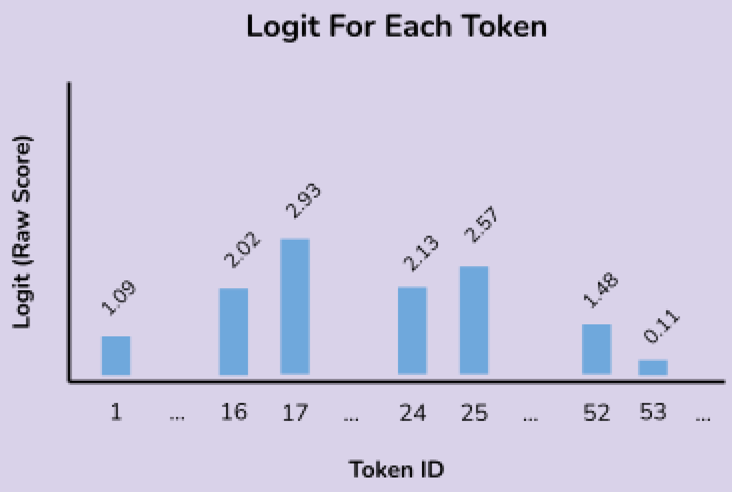
First, the tokenizer breaks up the text into tokens. These tokens can be a word, part of a word, or even a character, depending on the tokenizer. Below is an example of this.



Next, the tokenizer looks up the unique integer ids of these tokens. This helps with locating the token's embedding in the embedding table. The token's embedding is just a vector that captures the meaning of that specific token.



These embedding vectors will then be fed into the model's Transformer layers. At the very end, the model will output logits for all tokens in its vocabulary. You can think of these logits as raw scores that reflect how likely the model believes each token is to come next in the sequence.



After the model has assigned a score for each token, we can use these scores to assign probabilities for each token.

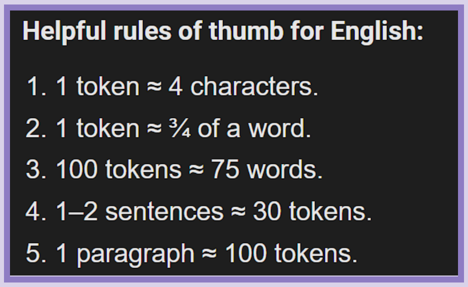
We do this by using the softmax function.

You don't need to know the softmax function right now, but it will most likely be referenced in the main project or in future classwork / projects.

LLM Parameters

We'll skip the stopping conditions for now because we want to get the model running first in order to observe its behavior. This section will be relatively light to prevent this module from getting too long, but I suggest you look into anything that interests you.

When you scroll to the bottom of the Jupyter notebook, you may see the variable `max_input_tokens`. As the name suggests, this is the maximum number of tokens that your prompt can be (including the paragraphs). On top of considering the max context length your model is capable of, remember that the longer the prompt, the longer it'll take to run.



Helpful rules of thumb for English:

1. 1 token ≈ 4 characters.
2. 1 token ≈ ¾ of a word.
3. 100 tokens ≈ 75 words.
4. 1–2 sentences ≈ 30 tokens.
5. 1 paragraph ≈ 100 tokens.

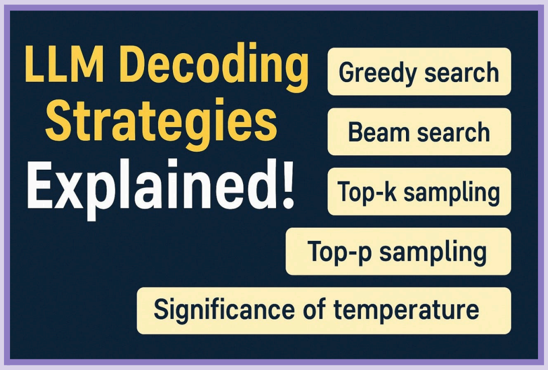
Don't worry too much about getting this exactly correct. If you set `max_num_paragraphs` to a reasonable number, you should be fine. This is just an extra safeguard in case scraping goes wrong.

Before you run the code, make sure you set `max_new_tokens` to a reasonable number. I set mine to 400 when I did this mini-project. If you make this number too large, the model may not know when to stop and you might end up waiting over 20 minutes for a single response.

If your code is taking a long time to run (mine runs in under a minute), it could be because:

1. Your prompt is very large. You might want to experiment with reducing the max number of paragraphs you're using.
2. You're letting the model generate too much. If this is the case, you might want to experiment with reducing the max number of tokens the model can generate.
3. You're using a very large model. If you choose to use a bitsandbytes configuration to load a larger model, it will run slower than smaller models. I don't have much experience using large models, but I did use a 8 billion parameter model for a project (it took about 5 minutes to generate a single response for a short prompt).

Another variable you might've noticed is `temperature`. To understand what temperature does, it might be useful to know that the next token generated by the model IS NOT always the token that the model assigned the largest score to (the token with the largest logit).

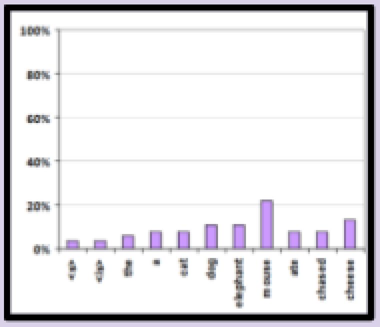

LLM Decoding Strategies Explained!
Greedy search
Beam search
Top-k sampling
Top-p sampling
Significance of temperature

There are many different methods for determining which token should be next based on the logits.

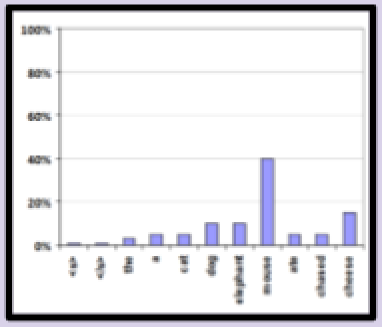
For this mini-project, we'll use a sampling-based method.

This means that when determining the next token, we roll a dice using the probability distribution. A high-probability token is still more likely, but lower-probability ones can sometimes appear. This trades perfect consistency for more variety and creativity.

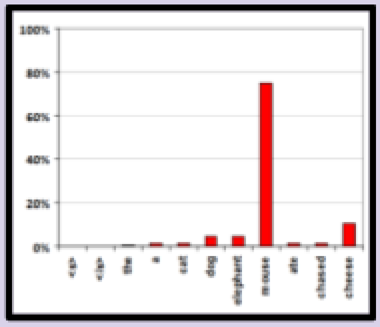
Temperature affects how weighted that dice roll is by altering the probability distribution.



Softer (T=2)



Normal (T=1)



Sharper (T=0.5)

We can see how different values for temperature affect the probability distribution:

- When T=1, the probability distribution is unchanged.
- When T < 1, the outputs become more deterministic because high-probability tokens become even more dominant.
- When T > 1, the outputs become more diverse because the probabilities are flattened out and more uniformly distributed.

Try experimenting with different values for temperature. From my testing, you can feel the difference immediately when you make drastic jumps.

Stopping Conditions

After tweaking some of the model parameters, you may notice that your model still isn't outputting perfect responses. Perhaps it's constantly repeating the same thing over or rambling about unrelated topics after it's answered the question.

* *If your model is working as intended, great job! You can either move onto the main project or switch to another model to get experience with implementing stopping conditions.*

When I ran my code without implementing any stopping conditions, the model would usually ramble after answering the question. It sometimes talks about random topic as shown below:

```
Question: "What happened to the comet? What happened to Mitsuha and Itomori?"

Text:
Your Name explained that Mitsuha ...
The destruction of Itomori by the comet ...
...

Answer:
In the Your Name anime, the comet destroyed Itomori in 2013 ... Human-computer
interaction is a field that involves designing and developing systems that enable
humans to interact with computers ...
```

There's no set protocol for observing LLM behavior, but I'll provide what I did in case that helps. I think it's a good idea to go into the Jupyter Notebook and spend a while by yourself first (with the help of ChatGPT, Gemini, etc.) because there's multiple ways to tackle this problem. If you ever get stuck, you can come back and take a look at what I did for inspiration.

Good luck :D

SPOILER BLOCK

Before looking at what I did, try editing the code and coming up with your own ideas of how to address this issue.

After running the model multiple times, I noticed that whenever the model began rambling about random topics, there was never a space after the last sentence of the actual answer.

```
Answer:
In the Your Name anime, the comet destroyed Itomori in 2013, killing Mitsuha.
However, from Taki's perspective ... Additionally, the comet's presence is linked to
mysterious occurrences over the centuries, including the formation of Lake Itomori
and potential future impacts.Human-computer interaction is a field that involves
designing and developing systems that enable humans to interact with computers ...
```

This detail is quite odd since the rest of the sentences all have spaces after their periods. To help see what was going on, I displayed any special tokens that might've been hidden when decoding. You can do this too by editing the `generate_responses` function.

```
def generate_response(tokenized_prompt, model, tokenizer, ... ):

  # Set up inputs by putting tensors onto the GPU
  inputs = {key: tensor_value.to(model.device) for ... }
  prompt_length = inputs['input_ids'].shape[-1]

  # Attach custom stopping criteria
  stopping_criteria = StoppingCriteriaList([Custom_Stop_Conditions( ... )])

  outputs = model.generate(**inputs,
                           temperature=temperature,
                           max_new_tokens=max_new_tokens,
                           do_sample=True,
                           pad_token_id=tokenizer.eos_token_id,
                           eos_token_id=tokenizer.eos_token_id,
                           stopping_criteria=stopping_criteria)

  generated_ids = outputs[0][prompt_length:]
  generated_text = tokenizer.decode(generated_ids, skip_special_tokens=False)

  return generated_text
```

When I did this, I noticed that there was a specific token that always appeared right before the model started rambling about unrelated topics.

```
Answer:
In the Your Name anime, the comet destroyed Itomori in 2013, killing Mitsuha.
However, from Taki's perspective ... Additionally, the comet's presence is linked to
mysterious occurrences over the centuries, including the formation of Lake Itomori
and potential future impacts.◁endoftext▷Human-computer interaction is a field that
involves designing and developing systems that enable humans to interact ...
```

From the looks of it, this seems to be a token that signals to the model that it is at the end of the answer. However, this is confusing because in the generation parameters, the model is told to stop when it generates the tokenizer's end-of-sequence token.

```
def generate_response(tokenized_prompt, model, tokenizer, ... ):

  # Set up inputs by putting tensors onto the GPU
  inputs = {key: tensor_value.to(model.device) for ... }
  prompt_length = inputs['input_ids'].shape[-1]

  # Attach custom stopping criteria
  stopping_criteria = StoppingCriteriaList([Custom_Stop_Conditions( ... )])

  outputs = model.generate(**inputs,
                           temperature=temperature,
                           max_new_tokens=max_new_tokens,
                           do_sample=True,
                           pad_token_id=tokenizer.eos_token_id,
                           eos_token_id=tokenizer.eos_token_id,
                           stopping_criteria=stopping_criteria)

  generated_ids = outputs[0][prompt_length:]
  generated_text = tokenizer.decode(generated_ids, skip_special_tokens=False)

  return generated_text
```

Next, I checked the tokenizer's end-of-sequence token using `print(tokenizer.eos_token)`. After running this code is a separate cell, it outputted `◁im_end▷`.

From this, we can see that the model seems to be using a token to signal the end of the answer that isn't the tokenizer's dedicated end-of-sequence token. Why is this happening?

Remember models produce what they were trained on and the tokenizer is just a tool for translation. During training, models are exposed to examples from a mix of public and proprietary datasets. These datasets may use different tokens or formatting to represent the end of a sequence. As a result, the model may learn to use another token to signal the end rather than the tokenizer's dedicated end-of-sequence token.

On top of checking for that specific token, I also added another condition that stopped generation whenever a newline character was used. This is because the model sometimes started a new paragraph whenever it began rambling off topic or repeating itself.

One idea that could be interesting to try is telling the model to generate a specific sequence after it's done answering. Maybe you can tell it to generate "###" after it's done and detecting that for your stop condition. I didn't try this method, but it could be a good idea.

* **Note**: Hopefully this section was helpful or insightful for you. It's hard to debug LLM behavior because there's no set protocol and different models can behave differently. If you're still having trouble getting your model to stop at the correct point, please feel free to ask for help during our next meeting - I'm always happy to help :)

With all that being said, congratulations on finishing this mini-project 🥳