Structured Output 2

In limits, there is freedom. Creativity thrives within structure.

– Julia B. Cameron

2.1 Introduction

Language Models excel at generating human-like text, but they often struggle to produce output in a structured format, consistently. This poses a significant challenge when we need LLMs to generate data that can be easily processed by downstream systems, such as databases, APIs, or other software applications. Even with a well-crafted prompt, an LLM might produce an unstructured response when a structured one is expected. This can be particularly challenging when integrating LLMs into systems that require specific data types and formats.

What user needs drive the demand for LLM output constraints? In a recent work by Google Research [2], the authors explored the user need for constraints on the output of large language models, drawing on a survey of 51 industry professionals who use LLMs in their work. User needs can be broadly categorized as follows:

A) Improving Developer Efficiency and Workflow

- **Reducing Trial and Error in Prompt Engineering.** Developers find the process of crafting prompts to elicit desired output formats to be time-consuming, often involving extensive testing and iteration. LLM output constraints could make this process more efficient and predictable.
- **Minimizing Post-processing of LLM Outputs** Developers frequently have to write complex code to wrangle and process LLM outputs that don't conform to expected formats. LLM structured output would simplify this, reducing the need for ad-hoc post-processing code.
- **Streamlining Integration with Downstream Processes** LLMs are often used within larger pipelines where their output serves as input for subsequent modules. Output constraints are crucial to ensure compatibility and prevent errors.
- **Enhancing the Quality of Synthetic Datasets** LLMs are increasingly used to generate synthetic data for AI training. Constraints can ensure data integrity and prevent the inclusion of unwanted elements that could negatively impact training outcomes.

B) Meeting UI and Product Requirements

- **Adhering to UI Size Limitations** LLM-generated content often needs to fit into specific UI elements with size restrictions, especially on mobile devices. Output length constraints prevent content overflow and ensure proper display within the UI.

- **Ensuring Output Consistency** Consistent output length and format are crucial for user experience and UI clarity. Constraints help maintain this consistency, avoiding overwhelming variability in generated text.

C) Enhancing User Trust and Experience

- **Mitigating Hallucinations** Users expect LLM-powered tools to be reliable and truthful. Constraining LLM outputs to a set of possible outcomes can help mitigate hallucinations, ensuring the output is valid.
- **Driving User Adoption** Users are more likely to engage with LLM-powered tools that provide reliable and consistent experiences. By ensuring output accuracy, consistency, and safety through constraints, developers can enhance user satisfaction and drive adoption. Overall, findings suggest the ability to constrain LLM output is not just a technical consideration but a fundamental user need, impacting developer efficiency, user experience, and the overall success of LLM-powered applications.

2.2 Problem Statement

Language models based on the Transformer architecture are next token prediction machines. These models calculate the probability of observing a token (from a vocabulary of size n) conditioned on the previous tokens in the sequence. This process can be expressed mathematically as:

$$P(X) = P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i|x_{<i}) \tag{2.1}$$

where, x_i represents the current token being generated, while $x_{<i}$ encompasses all preceding tokens.

However, in practical applications, generating high-quality content requires more than just probabilistic next-token generation. The key challenge lies in incorporating control conditions (C) that guide the model to produce text with specific desired characteristics - whether that's maintaining a consistent format, following syntactic rules, or adhering to semantic constraints. These control conditions must be integrated while preserving the model's ability to generate natural, coherent text. This controlled text generation process can be formalized as [3]:

[3]: Liang et al. (2024), *Controllable Text Generation for Large Language Models: A Survey*

$$P(X|C) = P(x_1, x_2, \ldots, x_n|C) = \prod_{i=1}^{n} p(x_i|x_{<i}, C) \tag{2.2}$$

Here, C represents the set of constraints or control conditions that shape the generated output. Common constraints (C) include:

- **Format Constraints**: Enforcing specific output formats like JSON, XML, or YAML ensures the generated content follows a well-defined structure that can be easily parsed and validated. Format constraints are essential for system integration and data exchange.

- **Multiple Choice Constraints**: Restricting LLM outputs to a predefined set of options helps ensure valid responses and reduces the likelihood of unexpected or invalid outputs. This is particularly useful for classification tasks or when specific categorical responses are required.
- **Static Typing Constraints**: Enforcing data type requirements (strings, integers, booleans, etc.) ensures outputs can be safely processed by downstream systems. Type constraints help prevent runtime errors and improve system reliability.
- **Length Constraints**: Limiting the length of generated content is crucial for UI display, platform requirements (like Twitter's character limit), and maintaining consistent user experience. Length constraints can be applied at the character, word, or token level.
- **Ensuring Output Consistency**: Consistent output length and format are crucial for user experience and UI clarity. Constraints help maintain this consistency, avoiding overwhelming variability in generated text.

This delicate balance between control and quality lies at the heart of structured output generation with LLMs. Next, we will explore some techniques to accomplish that.

2.3 Techniques

There are many techniques to obtain structured output from LLMs [3]. They can be broadly categorized into two types based on the phase they are applied to:

[3]: Liang et al. (2024), *Controllable Text Generation for Large Language Models: A Survey*

1. **Training-Time Techniques (TTT)**: These techniques are applied during the training or post-training phases of the LLM. They are used to guide the model to learn the specific patterns and structures that are required for the task at hand.
2. **Inference-Time Techniques (ITT)**: These techniques are applied during the inference phase of the LLM. They are used to guide the model to produce the desired output at inference time.

In **TTT**, the model is trained on a dataset that is specifically designed to teach the model the specific patterns and structures that are required for the task at hand. Hence, this information is added to the model's weights. This can be done at pre-training phase but it is usually performed at pos-training through the use of supervised fine-tuning or RLHF, where the model is trained on a dataset of labeled or preference-based examples.

In **ITT**, the model is guided to produce the desired output during the inference phase. This is typically done through prompt engineering or logit post-processing. Within these two broad categories, the most common approaches we will explore are (in decreasing order of popularity but increasing order of reliability):

- **Prompt Engineering** (ITT): Prompt engineering is a technique that involves crafting a prompt to guide the LLM to produce the desired output. This can be achieved by using tools like Pydantic to define the expected data structure and then using that definition to guide the LLM's output [1].

1: Prompt Engineering does not guarantee structured output generation!

- **Fine-Tuning** (TTT): Fine-tuning is a technique that involves training a language model on a specific task or dataset. This allows the model to learn the specific patterns and structures that are required for the task at hand [2].
  - Example: NousResearch/Hermes-2-Theta-Llama-3-8B [4], a model trained on a specific system prompt for Structured Outputs able to respond according to following user provided JSON schema.
- **Logit Post-Processing** (ITT): Logit post-processing is a technique that involves modifying the logits of the LLM's output before it is converted into text.
  - Example: Outlines [5], a Python package that allows to guide the generation process introducing logit biases. We will explore this solution later.

2: Fine-Tuning a base model to learn structured output generation helps but does not guarantee structured output generation!

[4]: NousResearch (2024), *Hermes-2-Theta-Llama-3-8B*

[5]: Outlines (2024), *Type-Safe Structured Output from LLMs*

2.3.1 Prompt Engineering

Perhaps the most common strategy to generate LLM response in a target format is using prompt engineering where the user explicitly requests output to be generated in a target format or provides an example of the desired output format within the prompt.

As a motivating example, consider the following simple task: Given a segment of a SEC financial filing, generate a two-person discussion about key financial data from the text in JSON format, simulating what would be a real-world discussion about the underlying companies' disclosed financial information. We would like to generate a structured output that can be easily parsed and integrated with other systems.

In a one-shot prompting fashion, we can pass the following example in the prompt:

```
{
   "Person1": {
     "name": "Alice",
     "statement": "The revenue for Q1 has increased by 20%
     ↪  compared to last year."
   },
   "Person2": {
     "name": "Bob",
     "statement": "That's great news! What about the net profit
     ↪  margin?"
   }
}
```

With the added instruction to:

```
"Generate a two-person discussion about the key financial
 ↪  data from the following text in JSON format."
```

```python
MAX_LENGTH = 10000 # We limit the input length to avoid
↪  token issues
with open('../data/apple.txt', 'r') as file:
    sec_filing = file.read()
sec_filing = sec_filing[:MAX_LENGTH]
```

```python
from dotenv import load_dotenv
import os

# Load environment variables from .env file
load_dotenv(override=True)
from openai import OpenAI
client = OpenAI()
```

```python
prompt = """
Generate a two-person discussion about the key financial
↪  data from the following text in JSON format.

<JSON_FORMAT>
{
   "Person1": {
     "name": "Alice",
     "statement": "The revenue for Q1 has increased by 20%
      ↪  compared to last year."
   },
   "Person2": {
     "name": "Bob",
     "statement": "That's great news! What about the net
      ↪  profit margin?"
   }
}
</JSON_FORMAT>
"""

response = client.chat.completions.create(
    model="gpt-4o-mini",
    messages=[
        {"role": "system", "content": prompt},
        {"role": "user", "content": sec_filing}
    ]
)
```

```python
response_content = response.choices[0].message.content
print(response_content)
```

~~~
```json
{
~~~

```
    "Person1": {
      "name": "Alice",
      "statement": "The aggregate market value of Apple's stock
      ↪  held by non-affiliates is approximately $2.63
      ↪  trillion."
    },
    "Person2": {
      "name": "Bob",
      "statement": "That's impressive! I also noticed that they
      ↪  have around 15.1 billion shares of common stock
      ↪  outstanding."
    }
}
```

```python
import json

def is_json(myjson):
  try:
    json.loads(myjson)
  except ValueError as e:
    return False
  return True
```

```python
is_json(response_content)
```

```
False
```

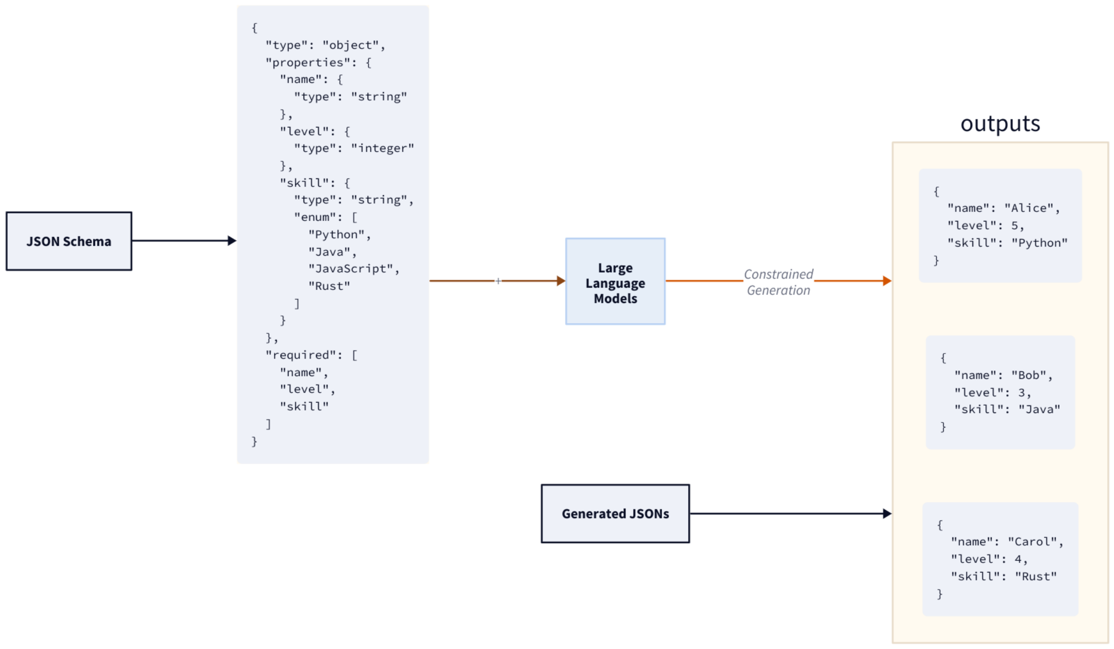
We observe the LLM provided a response in JSON format. However, it was enclosed by json markdown tags! While further prompt engineering could lead to a pure JSON response it is important to note this is not guaranteed. Hence, this strategy may cause failures if dependencies expect output in JSON format.

2.3.2 JSON Mode (Fine-Tuned)

One-shot prompting is a simple technique that can lead to low-effort improvements in structured output, though may not be sufficient for complex (e.g. nested) structures and/or when the model's output needs to be restricted to a specific set of options or types.

Some models offer so-called "JSON Mode" as an attempt to handle those challenges. This is a feature provided by most LLM API providers today, such as OpenAI, that allows the model to generate output in JSON format. This is particularly useful when you need structured data as a result, such as when parsing the output programmatically or integrating it with other systems that require JSON input. As depicted in Figure 2.1, JSON mode is implemented by instructing the LLM model to use JSON as response format and optionally defining a target schema.

When using JSON mode with OpenAI's API, it is recommended to instruct the model to produce JSON via some message in the conversation,



```
{
  "type": "object",
  "properties": {
    "name": {
      "type": "string"
    },
    "level": {
      "type": "integer"
    },
    "skill": {
      "type": "string",
      "enum": [
        "Python",
        "Java",
        "JavaScript",
        "Rust"
      ]
    }
  },
  "required": [
    "name",
    "level",
    "skill"
  ]
}
```

```
{
  "name": "Alice",
  "level": 5,
  "skill": "Python"
}
```

```
{
  "name": "Bob",
  "level": 3,
  "skill": "Java"
}
```

```
{
  "name": "Carol",
  "level": 4,
  "skill": "Rust"
}
```

Figure 2.1: Conceptual overview of JSON mode.

for example via your system message. If you don't include an explicit instruction to generate JSON, the model may generate an unending stream of whitespace and the request may run continually until it reaches the token limit. To help ensure you don't forget, the API will throw an error if the string "JSON" does not appear somewhere in the context.

```
prompt = f"""
Generate a two-person discussion about the key financial
↪  data from the following text in JSON format.
TEXT: {sec_filing}
"""
response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[{"role": "user", "content": prompt}],
response_format = { "type": "json_object" }
)
```

```
response_content = response.choices[0].message.content
print(response_content)
```

```
    {
```

```
    "person1": "I see that Apple Inc. reported a total market
    ↪  value of approximately $2,628,553,000,000 held by
    ↪  non-affiliates as of March 29, 2024. That's a
    ↪  significant amount!",
    "person2": "Yes, it definitely shows the scale and value
    ↪  of the company in the market. It's impressive to see
    ↪  the sheer size of the market value.",
    "person1": "Also, they mentioned having 15,115,823,000
    ↪  shares of common stock issued and outstanding as of
    ↪  October 18, 2024. That's a large number of shares
    ↪  circulating in the market.",
    "person2": "Absolutely, the number of shares outstanding
    ↪  plays a crucial role in determining the company's
    ↪  market capitalization and investor interest."
  }
```

This example solution is specific to OpenAI's API. Other LLM providers offer similar functionality, for example:

- ▶ Google's Gemini offers JSON mode via response schema configuration.
- ▶ Anthropic [3] suggests tools use to get Claude to produce JSON output that follows a schema.

Listing 2.1: Gemini's Structured Output.

```
llm_config = genai.
    generation_config={"
    response_mime_type": "
    application/json",
                "
    response_schema": list[
    Round]}
```

3: Isn't it surprising that Anthropic does not yet offer JSON and/or structured output generation to date? Isn't it even more surprising that today's recommended solution is to use function calling to get JSON responses even if you are not using the actual function in your application?

JSON mode is typically a form of fine-tuning, where a base model went though a post-training process to learn target formats. However, while useful this strategy is not guaranteed to work all the time.

A Note on "Guaranteed" Structured Output Mode In addition to JSON mode, it is worth mentioning that "Structured Output" mode is also becoming popular among LLM providers. This is a feature that ensures the model will *always* generate responses that adhere to your supplied JSON Schema.

Some benefits of Structured Outputs include:

- ▶ **Reliable type-safety**: No need to validate or retry incorrectly formatted responses.
- ▶ **Explicit refusals**: Safety-based model refusals are now programmatically detectable.
- ▶ **Simpler prompting**: No need for strongly worded prompts to achieve consistent formatting.

Here's a Python example demonstrating how to use the OpenAI API to generate a structured output. We aim at extracting structured data from our sample SEC filing, in particular: (i) entities and (ii) places mentioned in the input doc. This example uses the `response_format` parameter within the OpenAI API call. This functionality is supported by GPT-4o models, specifically `gpt-4o-mini`, `gpt-4o`, and later versions.

```python
from pydantic import BaseModel
from openai import OpenAI

class SECExtraction(BaseModel):
    mentioned_entities: list[str]
    mentioned_places: list[str]
```

```python
def extract_from_sec_filing(sec_filing_text: str, prompt:
↪  str) -> SECExtraction:
    """
    Extracts structured data from an input SEC filing
    ↪  text.
    """
    client = OpenAI()
    completion = client.beta.chat.completions.parse(
        model="gpt-4o-mini",
        messages=[
            {
                "role": "system",
                "content": prompt
            },
            {"role": "user", "content": sec_filing_text}
        ],
        response_format=SECExtraction
    )
    return completion.choices[0].message.parsed
```

Explanation:

- **Data Structures:** The code defines one Pydantic model, `SECExtraction`, to represent the structured output of our parser. This model provides type hints and structure for the response.
- **API Interaction:** The `extract_from_sec_filing` function uses the OpenAI client to send a chat completion request to the `gpt-4o-mini` model. The prompt instructs [4] the model to extract our target attributes from input text. The `response_format` is set to `SECExtraction`, ensuring the response conforms to the specified Pydantic model.
- **Output Processing:** The returned response is parsed into the `SECExtraction` model. The code then returns the parsed data.

```python
prompt_extraction = "You are an expert at structured data
↪  extraction. You will be given unstructured text from a
↪  SEC filing and extracted names of mentioned entities
↪  and places and should convert the response into the
↪  given structure."
sec_extraction = extract_from_sec_filing(sec_filing,
↪  prompt_extraction)
```

4: Try omitting from the prompt one of the variables you want to extract (e.g. "places" or "entities"). You may find your API call fails to extract your target variables even with a proper pydantic model specified. That's how sensitive prompt engineering is when it comes to structured output generation. That's why we need an alternative approach which is the focus of the next section.

```python
print("Extracted entities:",
↪  sec_extraction.mentioned_entities)
print("Extracted places:",
↪  sec_extraction.mentioned_places)
```

```
    Extracted entities: ['Apple Inc.', 'The Nasdaq Stock Market
    ↪  LLC']
    Extracted places: ['Washington, D.C.', 'California',
    ↪  'Cupertino, California']
```

We observe that the model was able to extract the entities and places from the input text, and return them in the specified format.

The use of Pydantic models and the `response_format` parameter enforces the structure of the model's output, making it more reliable and easier to process.

This structured approach improves the reliability and usability of your application by ensuring consistent, predictable output from the OpenAI API. This example solution is specific to OpenAI's API. That begs the question: How can we solve this problem generally for widely available open source LLMs? Enters logit post-processing.

2.3.3 Logit Post-Processing

Logit post-processing is a technique that involves modifying the logits of the LLM's output before it is converted into text such that we have a "controlled" text generation.

The text generation process follows a probabilistic approach. At each step, the model calculates the probability distribution over its entire vocabulary to determine the most likely next token.

Let's examine how an LLM processes an example prompt "Is Enzo a good name for a baby?" as depicted in Figure 2.2:

1. The tokenizer first segments the input text into tokens
2. Each token gets mapped to a unique numerical ID
3. The language model processes these token IDs through its deep neural network
4. The model produces logits - unnormalized scores for each possible next token
5. A softmax [5] transformation converts these raw logits into a probability distribution
6. The highest probability (3.25%) indicates the model's strongest prediction for the next token
7. Text generation then selects the next token based on a given strategy (e.g. greedy, top-k, etc)

5: From raw logits to softmax probabilities:

$$P(y_i|x) = \text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

for $i = 1, \ldots, n$ where, $P(y_i|x)$ represents the probability of class i given input x and x_i is the logit (raw score) for class i.

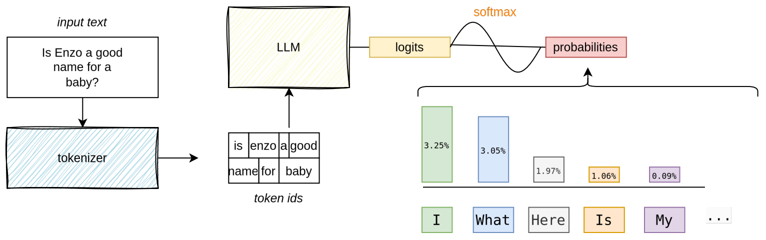


Figure 2.2: Text Generation Process.

We can leverage the `transformers` library to extract the logits of the last token of the prompt.

```python
MODEL_NAME = "HuggingFaceTB/SmolLM2-1.7B-Instruct"
PROMPT = "Is Enzo a good name for a baby?"

from transformers import AutoTokenizer,
↪  AutoModelForCausalLM
```

```python
import torch

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForCausalLM.from_pretrained(MODEL_NAME,
    torch_dtype=torch.bfloat16, device_map="auto")
```

```python
inputs = tokenizer(PROMPT,
↪   return_tensors="pt").to(model.device)

# Get logits
with torch.inference_mode():
    outputs = model(**inputs)
    logits = outputs.logits

# Logits for the last token
last_token_logits = logits[:, -1, :]

next_token_probs =
↪   torch.nn.functional.softmax(last_token_logits, dim=-1)

k = 10
top_k_probs, top_k_indices = torch.topk(next_token_probs,
↪   k, dim=-1)

# Print the actual tokens, skipping special tokens
top_k_tokens = [tokenizer.decode(idx,
↪   skip_special_tokens=True)
                for idx in top_k_indices[0]]

print(f"Top predicted tokens and probabilities:")
for prob, token in zip(top_k_probs[0][:k],
↪   top_k_tokens[:k]):
    if token.strip():  # Only print non-empty tokens
        print(f"'{token}': {prob:.4f}")
```

```
Top predicted tokens and probabilities:
' I': 0.0325
' What': 0.0305
' Here': 0.0197
' Is': 0.0106
' My': 0.0093
```

The main idea here is that we can modify the logits of the last token to bias the model towards the tokens we want to see in the output hence "controlling" the generation process.

The `transformers` library provides a `LogitsProcessor` class (see Listing 2.2) that allows us to modify the logits of the last token.

Without any logit processing, the model will generate the most likely token based on the probabilities.

Listing 2.2: LogitsProcessor Class.

```python
class LogitsProcessor:
    """Abstract base class
    for all logit processors
    that can be applied
    during generation."""
    @add_start_docstrings(
    LOGITS_PROCESSOR_INPUTS_DOCSTRING
    )
    def __call__(self,
    input_ids: torch.
    LongTensor, scores: torch
    .FloatTensor) -> torch.
    FloatTensor:
```

```
model.generate(**input)
```

With logit processing, we can modify the logits of the last token to bias the model towards the tokens we want to see in the output.

```
model.generate(**input, logits_processor=LogitsProcessorL⌋
↪  ist([CustomLogitsProcessor()]))
```

Here, `CustomLogitsProcessor` is an example of a user-defined custom logits processor concrete class we would pass to `model.generate()` to instruct our goal to control the generation process. This class should implement the abstract `LogitsProcessor` class from the `transformers` library.

The `LogitsProcessor` provides a unified interface for modifying prediction scores during text generation with language models. It acts as an intermediary step between the raw logits output by the model and the final token selection process:

1. Input: Takes sequence tokens (input_ids) and prediction scores (scores tensor)
2. Output: Returns modified scores that influence the next token selection

Importantly, it defines the `__call__` method that takes two key arguments:

1. input_ids (torch.LongTensor):
   - Shape: (batch_size, sequence_length)
   - Contains the token IDs of the input sequence
   - Obtained through tokenizer.encode() or tokenizer.__call__()
2. scores (torch.FloatTensor):
   - Shape: (batch_size, vocab_size)
   - Raw logits from the language model head
   - Can be pre-softmax or post-softmax scores
   - Represents prediction probabilities for each token

The method returns:

- torch.FloatTensor with shape (batch_size, vocab_size)
- Contains the modified prediction scores after processing

This allows for custom manipulation of the scores before token selection, enabling fine-grained control over the generation process.

Let's go through a concrete example to better understand how we can control output generation.

Let's suppose we want to bias the model towards the tokens "Yes" and "No" in the output. That is, we would like the model to always return "Yes" or "No" in the output to our prompt (in this example "Is Enzo a good name for a baby?").

A naive approach would be to modify the logits of the last token by masking out all tokens except "Yes" and "No". We can then pick the most likely token (from "Yes" or "No") as the output in a greedy fashion.

The `YesNoLogitsProcessor` class below implements this naive greedy approach by implementing a custom `LogitsProcessor`.

```
class YesNoLogitsProcessor(LogitsProcessor):
    def __init__(self, yes, no, tokenizer,
    ↪   initial_length):
        self.yes = yes
        self.no = no
        self.tokenizer = tokenizer
        self.initial_length = initial_length

    def __call__(self, input_ids: torch.LongTensor,
    ↪   scores: torch.FloatTensor) -> torch.FloatTensor:
        # If we already generated a response, mask
        ↪   everything
        if len(input_ids[0]) > self.initial_length:
            scores.fill_(-float('inf'))
            return scores

        # Debug prints
        yes_tokens = self.tokenizer.encode(self.yes,
        ↪   add_special_tokens=False)
        no_tokens = self.tokenizer.encode(self.no,
        ↪   add_special_tokens=False)
        print(f"Yes token ID: {yes_tokens}")
        print(f"No token ID: {no_tokens}")


        # Extract original logits for yes/no
        yes_no_logits = scores[:, [yes_tokens[0],
        ↪   no_tokens[0]]]
        print(f"[Yes, No] logits: {yes_no_logits}")

        # Get probabilities using softmax
        yes_no_probs =
        ↪   torch.nn.functional.softmax(yes_no_logits,
        ↪   dim=-1)
        yes_prob = yes_no_probs[:, 0]
        no_prob = yes_no_probs[:, 1]
        print(f"Yes prob: {yes_prob}")
        print(f"No prob: {no_prob}")

        # Mask all tokens with -inf
        scores.fill_(-float('inf'))

        # Set the higher probability choice to 0
        yes_mask = yes_prob > no_prob
        scores[:, yes_tokens[0]] = torch.where(yes_mask,
        ↪   torch.tensor(1e4),
        ↪   torch.tensor(-float('inf')))
        scores[:, no_tokens[0]] = torch.where(~yes_mask,
        ↪  torch.tensor(1e4),
        ↪  torch.tensor(-float('inf')))

        return scores
```

Now we can simply pass our custom logits processor to `model.generate()` method.

```
input_ids = tokenizer.encode(PROMPT, return_tensors="pt")
initial_length = len(input_ids[0])

YES = "yes"
NO = "no"

# Controlled generation
generation_output_controlled = model.generate(**inputs,
↪  logits_processor=LogitsProcessorList([YesNoLogitsProc⌋
↪  essor(YES, NO, tokenizer, initial_length)]),
↪  max_length=50)

# Uncontrolled generation
generation_output = model.generate(**inputs,
↪  max_length=50)
```

Below we print the “Yes” and “No” token IDs, the logits of the last token (raw scores before masking) and then the probabilities of the tokens “Yes” and “No” after masking and softmax normalization. In this run, the model predicts “Yes” with a probability of 0.4263 and “No” with a probability of 0.5737, post-masking. In our greedy approach, our custom logits processor will pick the most likely token, in this case “No”.

```
Yes token ID: [10407]
No token ID: [4607]
[Yes, No] logits: tensor([[2.6250, 2.9219]])
Yes prob: tensor([0.4263])
No prob: tensor([0.5737])
```

Let’s see if it works as expected. We write a helper function to extract the response from the model’s generation output.

```
def generate_response(model_output, tokenizer):
    gen_output = tokenizer.batch_decode(model_output,
    ↪  skip_special_tokens=True,
    ↪  clean_up_tokenization_spaces=False)
    generated_text = gen_output[0][
                len(
                    tokenizer.decode(
                        inputs["input_ids"][0],
                        ↪  skip_special_tokens=True
                    )
                ) :
            ].strip()
    return generated_text
```

Controlled generation using our custom logits processor:

```
generate_response(generation_output_controlled, tokenizer)
```

```
'no'
```

Regular generation [6]:

6: We limit text generation to 50 tokens output.

```
generate_response(generation_output, tokenizer)
```

```
'Enzo is a classic Italian name that'
```

As we can see, our controlled generation returned "No" to our prompt while the regular generation returned a more verbose response, as expected.

Here, we provided a simple greedy approach to logit post-processing using transformers library in a process that can be extended and customized to more complex logit post-processing tasks [7]. There are some higher level libraries that can help us streamline this process while offering some flexibility and control such as Outlines. We will cover this library as well as other tools such as Ollama and Langchain in the following sections each one offering a different approach to structured output with varying degrees of control and flexibility.

7: A more robust approach would define a grammar to control output generation as we will demonstrate in the next section.

2.4 Tools

2.4.1 Outlines

Outlines [5] is a library specifically focused on structured text generation from LLMs. Under the hood, Outlines works by adjusting the probability distribution of the model's output logits - the raw scores from the final layer of the neural network that are normally converted into text tokens. By introducing carefully crafted logit biases, Outlines can guide the model to prefer certain tokens over others, effectively constraining its outputs to a predefined set of valid options.

[5]: Outlines (2024), *Type-Safe Structured Output from LLMs*

The authors solve the general guided generation problem [6], which, as a consequence, solves the problem of structured output generation in LLMs by introducing an efficient indexing approach that reformulates neural text generation using finite-state machines (FSMs) [8].

[6]: Willard et al. (2023), *Efficient Guided Generation for Large Language Models*

8: A Finite State Machine (FSM) is a computational model that consists of a finite number of states and transitions between those states, where the system can be in only one state at a time and moves between states based on inputs or conditions.

They define the next token generation as a random variable:

$$s_{t+1} \sim \text{Categorical}(\alpha) \text{ where } \alpha = \text{LLM}(S_t, \theta)$$

Where:

- s_{t+1} is the next token to be generated
- $S_t = (s_1 ... s_t)$ represents a sequence of t tokens with $s_t \in V$
- V is the vocabulary with size $|V| = N$ (typically around 10^4 or larger)
- $\alpha \in \mathbb{R}^N$ is the output logits/probabilities over the vocabulary
- θ is the set of trained parameters of the LLM
- LLM refers to a deep neural network trained on next-token-completion tasks
- Categorical(α) represents sampling from a categorical distribution with probabilities α

When applying masking for guided generation, this becomes:

$$\tilde{\alpha} = m(S_t) \odot \alpha$$

$$\tilde{s}_{t+1} \sim \text{Categorical}(\tilde{\alpha})$$

Where:

- $m : P(V) \rightarrow 0, 1^N$ is a boolean mask function
- $\odot$ represents element-wise multiplication
- $\tilde{\alpha}$ is the masked (constrained) probability distribution
- $\tilde{s}_{t+1}$ is the next token sampled under constraints

This formulation allows the masking operation to guide the generation process by zeroing out probabilities of invalid tokens according to the finite state machine states. But instead of checking the entire vocabulary (size N) at each generation step ($O(N)$ complexity)[9] to enforce output constraints, they convert constraints (regex/grammar) into FSM states and build an index mapping FSM states to valid vocabulary tokens. This achieves $O(1)$ average complexity for token generation.

9: $O(N)$ complexity simply means that the algorithm's execution time grows linearly the size of the vocabulary.

[7]: Tran-Thien (2024), *Fast, High-Fidelity LLM Decoding with Regex Constraints*

In summary, there are two stages in the Outlines framework [7]:

1. **Preprocessing Step**: Outlines converts a character-level deterministic finite automaton (DFA) testing whether a string matches a regex into a token-level DFA testing whether a token sequence is decoded in a string matching the regex.
2. **Decoding Step**: At decoding time, the DFA is used to determine, for each new token, which potential tokens are allowed. Starting from the initial state of the DFA, the allowed tokens are determined by the outgoing transitions from the current state. The corresponding mask is applied to the next token probabilities and these probabilities are renormalized. A new token can then be sampled and the state of the DFA updated.

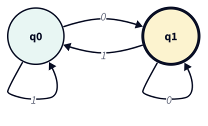


Figure 2.3: A DFA is a computational model that processes input one symbol at a time, moving between a finite set of states in a deterministic way. Hence, DFA is a subset of FSM with additional constraints that make it deterministic.

At each step, the model's probability distribution is masked and renormalized according to the current state and valid transitions.

As an example, let's suppose we want to constrain the output of an LLM to the following set of options:

- Y/yes
- N/no

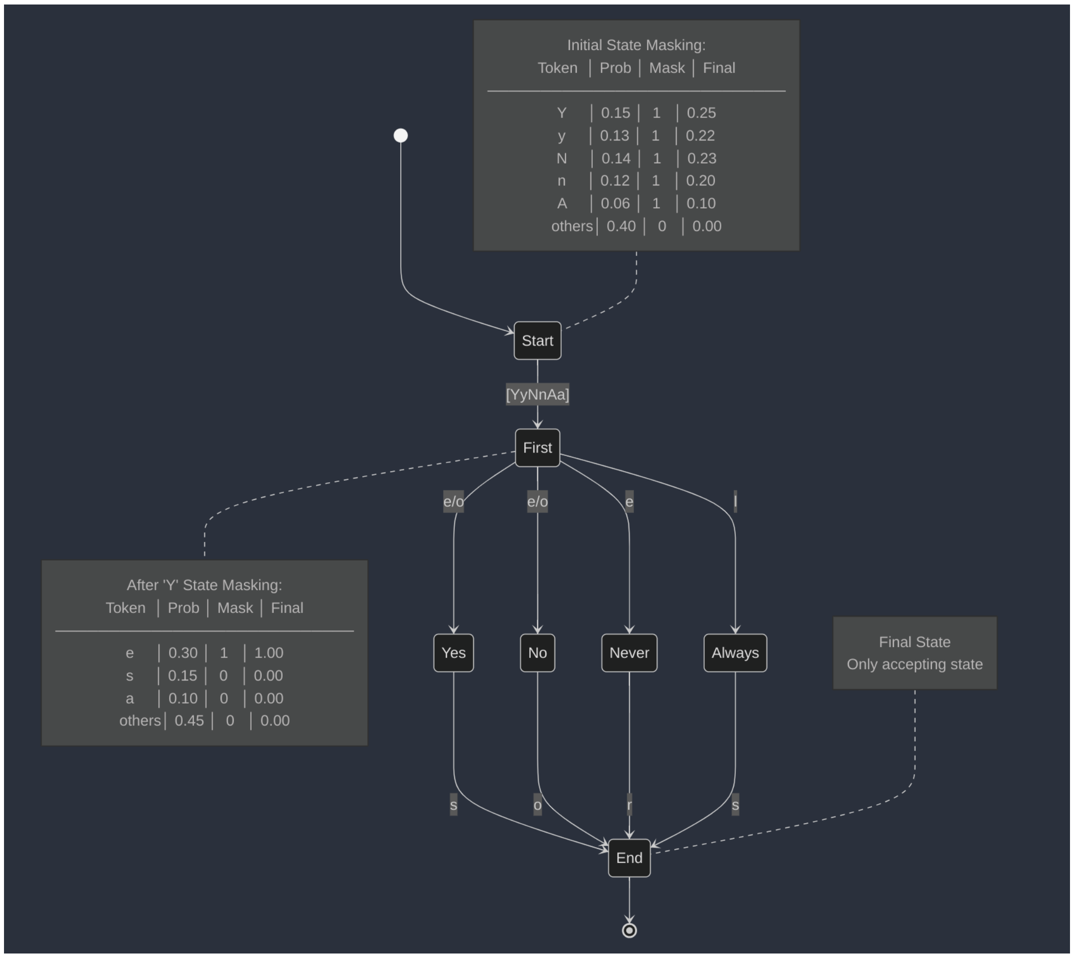
- N/never
- A/always

This can be done by creating a state machine that has a start state, an end state and a set of valid transitions between states with possible states represented as the following regex string:

```
r"\textbackslash s∗([Yy]es|[Nn]o|[Nn]ever|[Aa]lways)"
```

The state machine in Fig. 2.4 illustrates how Outlines works under the hood, where:

- Prop: Represents the logit token probability given by the LLM
- Mask: Mask value of the transition as defined by the state machine
- Final: The renormalized token probability post-masking



Initial State Masking:

Token	Prob	Mask	Final
Y	0.15	1	0.25
y	0.13	1	0.22
N	0.14	1	0.23
n	0.12	1	0.20
A	0.06	1	0.10
others	0.40	0	0.00

After 'Y' State Masking:

Token	Prob	Mask	Final
e	0.30	1	1.00
s	0.15	0	0.00
a	0.10	0	0.00
others	0.45	0	0.00

Figure 2.4: Outlines State Machine [7]

The initial "Start" state contains a masking table that controls which tokens can begin the sequence. In this example, only characters from the set `[YyNnAa]` are allowed as valid first characters, with each having an assigned probability and mask value. The masking mechanism effectively filters out invalid tokens by setting their mask values to 0, ensuring only permitted transitions to the "First" state.

After transitioning to the "First" state, the system continues to use probability masking to guide the sequence. For example, when receiving 'Y' as input, the masking table adjusts token probabilities to ensure valid continuations.

This finite state machine architecture serves multiple purposes in controlling text generation:

1. Managing token probabilities through strategic masking
2. Preventing invalid token sequences
3. Enforcing specific token patterns
4. Providing fine-grained control over token generation and validation

In that way, Outlines, the Python package, provides several powerful controlled generation features for developers:

- **Regex-based structured generation**: Guide the generation process using regular expressions.
- **Multiple Choice Generation**: Restrict the LLM output to a predefined set of options.
- **Pydantic model**: Ensure the LLM output follows a Pydantic model.
- **JSON Schema**: Ensure the LLM output follows a JSON Schema.

Outlines can support major proprietary LLM APIs (e.g. OpenAI via vLLM). However, one of its key advantages is the ability to ensure structured output for Open Source models, which often lack such guarantees by default.

```
pip install outlines
pip install transformers
```

10: Chapter 1 presents some remarkable performance data on Qwen2.5 models, which are one of them best Open Source models available today

In this example, we will use a `Qwen2.5-0.5B` model [10], a lightweight open source model from Alibaba Cloud known for its strong performance despite its small size.

```
import outlines

model = outlines.models.transformers("Qwen/Qwen2.5-0.5B-I⌋
↪  nstruct")
```

```
TOP = 100
prompt = f"""You are a sentiment-labelling assistant
↪  specialized in Financial Statements.
Is the following document positive or negative?

Document: {sec_filing[:TOP]}
"""

generator = outlines.generate.choice(model, ["Positive",
↪  "Negative"])
answer = generator(prompt)
print(answer)
```

```
    Negative
```

In this simple example, we use Outlines' `choice` method to constrain the model output to a predefined set of options ("Positive" or "Negative"). This ensures the model can only return one of these values, avoiding any unexpected or malformed responses.

Outlines also allows to guide the generation process so the output is guaranteed to follow a JSON schema or a Pydantic model. We will go back to our example of extracting entities and places from a SEC filing. In order to do so, we simply need to pass our Pydantic model to the `json` method in Outlines' `generate` module.

```
BASE_PROMPT = "You are an expert at structured data
↪  extraction. You will be given unstructured text from a
↪  SEC filing and extracted names of mentioned entities
↪  and places and should convert the response into the
↪  given structure."
```

```
prompt = f"{BASE_PROMPT} Document: {sec_filing[:TOP]}"
generator = outlines.generate.json(model, SECExtraction)
sec_extraction_outlines = generator(prompt)
```

```
print("Extracted entities:",
↪  sec_extraction_outlines.mentioned_entities)
print("Extracted places:",
↪  sec_extraction_outlines.mentioned_places)
```

```
    Extracted entities: ['Zsp', 'ZiCorp']
    Extracted places: ['California']
```

We observe that the model was able to extract the entities and places from the input text, and return them in the specified format. However, it is interesting to see that the model hallucinates a few entities, a phenomenon that is common for smaller Open Source models that were not fine-tuned on the task of entity extraction [11].

11: An alternative approach would be to define a valid set of entities to be extracted in order to avoid hallucination.

2.4.2 LangChain

LangChain [8] is a framework designed to simplify the development of LLM applications. It provides an abstraction layer over many LLM providers that in turn offers structured output.

In particular, LangChain offers the `with_structured_output` method, which can be used with LLMs that support structured output APIs, allowing you to enforce a schema directly within the prompt.

with_structured_output takes a schema as input which specifies the names, types, and descriptions of the desired output attributes. The method returns a model-like Runnable, except that instead of outputting strings or messages it outputs objects corresponding to the given schema. The schema can be specified as a TypedDict class, JSON Schema or a Pydantic class. If TypedDict or JSON Schema are used then a dictionary will be returned by the Runnable, and if a Pydantic class is used then a Pydantic object will be returned.

```
pip install -qU langchain-openai
```

```
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
def extract_from_sec_filing_langchain(sec_filing_text:
↪  str,
    prompt: str) -> SECExtraction:
    """
    Extracts structured data from an input SEC filing text
    ↪  using LangChain.
    """
    llm = ChatOpenAI(model="gpt-4o-mini")

    structured_llm =
    ↪  llm.with_structured_output(SECExtraction)

    prompt_template = ChatPromptTemplate.from_messages(
        [
            ("system", prompt),
            ("human", "{sec_filing_text}"),
        ]
    )

    llm_chain = prompt_template | structured_llm

    return llm_chain.invoke(sec_filing_text)
```

```
prompt_extraction = """You are an expert at structured
↪  data extraction.
You will be given unstructured text from a SEC filing and
extracted names of mentioned entities and places and
↪  should convert the response into the given
↪  structure."""
sec_extraction_langchain =
↪  extract_from_sec_filing_langchain(
    sec_filing, prompt_extraction)
```

```
print("Extracted entities:",
↪  sec_extraction_langchain.mentioned_entities)
print("Extracted places:",
↪  sec_extraction_langchain.mentioned_places)
```

```
Extracted entities: ['Apple Inc.']
Extracted places: ['California', 'Cupertino']
```

We observe that the model was able to extract the entities and places from the input text, and return them in the specified format. A full list of models that support `.with_structured_output()` can be found here. You can also use Outlines with LangChain [9].

[9]: LangChain (2024b), *Outlines Integration Documentation*

2.4.3 Ollama

Ollama [10] is a popular tool that allows you to run LLMs locally (see Chapter 1 for additional details). Ollama first introduced structured output generation in version 0.5.1 in late 2024 providing support for JSON output but highlighting additional formats are coming soon.

The current `ollama` implementation leverages LLama.cpp GBNF (GGML BNF) grammars [11] to enable structured output generation. LLama.cpp GBNF forces language models to generate output in specific, predefined formats by constraining their outputs to follow precise rules and patterns. The system accomplishes this through a formal grammar specification that defines exactly how valid outputs can be constructed. It's essentially an extension of BNF (Backus-Naur Form) [12] with some modern regex-like features added. These rules carefully define what elements are allowed, how they can be combined, and what patterns of repetition and sequencing are valid. By enforcing these constraints during generation, GBNF ensures the model's output strictly adheres to the desired format.[12]

[11]: Ggerganov (2024), *Llama.cpp Grammars Documentation*

[12]: Wikipedia contributors (2024), *Backus Naur form*

12: As a simple BNF example, this grammar defines basic arithmetic expressions with numbers, addition, subtraction, and parentheses.

$$
\begin{aligned}
\langle\text{expr}\rangle &::= \langle\text{number}\rangle \\
&\mid \langle\text{expr}\rangle \text{ ` + ' } \langle\text{expr}\rangle \\
&\mid \langle\text{expr}\rangle \text{ `-' } \langle\text{expr}\rangle \\
&\mid \text{`('} \langle\text{expr}\rangle \text{`)'} \\
\langle\text{number}\rangle &::= \langle\text{digit}\rangle \mid \langle\text{number}\rangle\langle\text{digit}\rangle \\
\langle\text{digit}\rangle &::= \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'}
\end{aligned}
$$

Let's replicate our previous structured output generation example with Ollama. First, make sure you have Ollama installed. You can find installation instructions here.

```
curl -fsSL https://ollama.com/install.sh | sh
pip install ollama
```

The code below demonstrates how to use Ollama's structured output capabilities with a Pydantic model as we did before with OpenAI, LangChain and Outlines.

The `SECExtraction` pydantic model defines the expected structure with two fields: mentioned_entities and mentioned_places as lists of strings we expect the model to return given an input SEC filing. The `extract_entities_from_sec_filing` function uses Ollama's chat API to analyze SEC filings and extract entities in a structured format, with temperature set to 0 for deterministic results. We pass the Pydantic model's JSON schema to Ollama via the `format` parameter. We append a suffix to the prompt instructing the model to return the response as JSON ("Return as JSON.") as recommended by Ollama maintainers.

```python
from ollama import chat
from pydantic import BaseModel

class SECExtraction(BaseModel):
    mentioned_entities: list[str]
    mentioned_places: list[str]

OLLAMA_STRUCTURED_OUTPUT_PROMPT_SUFFIX = "Return as JSON."
OLLAMA_STRUCTURED_OUTPUT_TEMPERATURE = 0

def extract_entities_from_sec_filing(doc: str, model: str)
 ↪  -> dict:
    """
    Extract entities and places from an SEC filing using
     ↪  Ollama chat.

    Args:
        doc: The SEC filing text to analyze
        model: The Ollama model to use for extraction

    Returns:
        The raw response from the chat model
    """
    response = chat(
        messages=[
            {
                'role': 'user',
                'content': f"""{BASE_PROMPT}
                {OLLAMA_STRUCTURED_OUTPUT_PROMPT_SUFFIX}

                Document: {doc}"""
            }
        ],
        model=model,  # You can also use other models like
         ↪  'mistral' or 'llama2-uncensored'
        format=SECExtraction.model_json_schema(),
        options={'temperature':
         ↪  OLLAMA_STRUCTURED_OUTPUT_TEMPERATURE}  # Set
         ↪  to 0 for more deterministic output
    )
    return response
```

We can now run the function and print the extracted entities and places. But first we need to start the Ollama server with our target LLM model (Qwen2.5-0.5B) running locally.

```
ollama run qwen2.5:0.5b
```

```python
doc = sec_filing[:TOP]
model = "qwen2.5:0.5b"
```

```
response = extract_entities_from_sec_filing(doc, model)

import json
response_json = json.loads(response.message.content)
```

```
print("Extracted entities:",
↪  response_json.get('mentioned_entities'))
print("Extracted places:",
↪  response_json.get('mentioned_places'))
```

```
Extracted entities: ['United States', 'SECURITIES AND EXCHANGE
↪  COMMISSION']
Extracted places: []
```

The extracted entities and places were quite different from those previously extracted using Outlines and Langchain, as expected since this depends mostly on the underlying model which is quite small in this example. We do observe though that we have successfully obtained results in JSON format as specified.

2.5 Discussion

2.5.1 Best Practices

When implementing structured output with LLMs, it's crucial to understand the distinction between different approaches. Some methods, such as logit post-processing, provide mathematical guarantees that the output will conform to the specified structure. This contrasts sharply with approaches like JSON mode, which rely on fine-tuned models or prompt engineering that offer no formal guarantees. This distinction becomes particularly important in production environments where reliability and consistency are paramount. With that in mind, here are some best practices to consider when implementing structured output generation with LLMs:

- **Clear Schema Definition**: Define the desired output structure clearly. This can be done in several ways including schemas, types, or Pydantic models as appropriate.
- **Descriptive Naming**: Use meaningful names for fields and elements in your schema.
- **Integration**: If you are connecting the model to tools, functions, data, etc. in your system, then you are highly encouraged to use a typed structured output (e.g. Pydantic models) to ensure the model's output can be processed correctly by downstream systems.

In summary, first one needs to clearly define the typed structure LLM applications will interface with, then determine whether strong guarantees are needed in order to determine tradeoffs between control and ease of implementation.

2.5.2 Comparing Solutions

The choice of framework for structured LLM output depends heavily on specific constraints, requirements and use cases. LangChain is the most used LLM framework today with a large developer community base however its structured output generation depends on the underlying LLM provider support. Ollama enables straightforward local deployment and experimentation democratizing access to LLMs while fostering privacy and control, however today it only offers JSON format with further formats to come. Outlines emerges as a solution that provides formal guarantees with great flexibility and control over structured output generation while providing support for a wide range of LLMs. Table 2.1 provides a summary comparison of the different solutions.

Table 2.1: Structured Output Frameworks Comparison

Feature	LangChain	Outlines	Ollama
Approach	Wrapper around LLM's native structured output APIs using with_structured_-output method	Adjusts probability distribution of model's output logits to guide generation	Uses llama.cpp GBNF grammars to constrain output format
Model Support	Limited to LLMs with built-in structured output APIs	Broad support for open-source models via transformers, llama.cpp, exllama2, mlx-lm and vllm	Broad support focused on enabling running open-source models locally
Output Format Support	- TypedDict - JSON Schema - Pydantic class	- Multiple choice generation - Regex-based structure - Pydantic model - JSON Schema	- Currently JSON only - Additional formats planned
Key Advantages	- Simple integration with supported LLMs	- Provides guarantees on output structure - Fine-grained control over generation - Strong open-source model support	- Excellent for local deployment - Simple setup and usage - Built-in model serving
Use Case Focus	Enterprise applications using commercial LLMs	Applications requiring output control guarantees or using open-source models	Local deployment and/or experimentation

Other related tools not covered in this chapter worth mentioning include Guidance [13] and NVIDIA's Logits Processor Zoo [14].

2.5.3 Research and Ongoing Debate

The use of structured output for Large Language Models is a developing area. While the ability to constrain LLM outputs offer clear benefits in parsing, robustness, and integration, there is growing debate on whether it also potentially comes at the cost of performance as well as reasoning abilities. Research in this area should be taken with a grain of salt since findings are mixed and often depend on the specific task and model family at hand furthermore model families are not always comparable and are getting updated by the day! Nonetheless, early findings provide some interesting insights as to why there is no one-size-fits-all solution when it comes to LLMs structured output.

There is some evidence indicating that LLMs may have bias in their handling of different output formats [15]. This study examined common output structures like multiple-choice answers, wrapped text, lists, and key-value mappings. The authors analyzed key LLM model families, namely Gemma, Mistral, and ChatGPT, uncovering bias across multiple tasks and formats. The researchers attributed these biases to the models' underlying token distributions for different formats. An example of this format bias emerged in the comparison between JSON and YAML outputs. While models like Mistral and Gemma excelled at generating JSON structures, they performed notably worse with YAML. Their YAML outputs often contained extraneous information that degrades output quality. This disparity likely stems from JSON's prevalence in training data, highlighting how a format's popularity directly influences model performance. While the studied models can be probably considered outdated by now since models are getting updated on a rapidly fashion, it is important to remark that addressing format bias is critical for advancing LLMs and ensuring their reliable application in real-world scenarios.

[15]: Long et al. (2024), 'LLMs Are Biased Towards Output Formats! Systematically Evaluating and Mitigating Output Format Bias of LLMs'

Recent (not yet peer-reviewed) research "Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models" [16] suggests that imposing format restrictions on LLMs might impact their performance, particularly in reasoning-intensive tasks. Further evidence [17] suggests LLMs may produce lower quality code if they're asked to return it as part of a structured JSON response, in particular:

[16]: Tam et al. (2024), *Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models*

[17]: Aider (2024), *Code in JSON: Structured Output for LLMs*

- **Potential performance degradation:** Enforcing structured output, especially through constrained decoding methods like JSON-mode, can negatively impact an LLM's reasoning abilities. This is particularly evident in tasks that require multi-step reasoning or complex thought processes.
- **Overly restrictive schemas:** Imposing strict schemas can limit the expressiveness of LLM outputs and may hinder their ability to generate creative or nuanced responses. In certain cases, the strictness of the schema might outweigh the benefits of structured output.
- **Increased complexity in prompt engineering:** Crafting prompts that effectively guide LLMs to generate structured outputs while maintaining performance can be challenging. It often requires careful consideration of the schema, the task instructions, and the desired level of detail in the response.

On the other hand, those findings are not without criticism. The .txt team challenges the work of [16]. The rebuttal argues that **structured generation, when done correctly, actually *improves* performance** [18].

[16]: Tam et al. (2024), *Let Me Speak Freely? A Study on the Impact of Format Restrictions on Performance of Large Language Models*

[18]: Dottxt (2024), *Say What You Mean: Demos*

The .txt team presents compelling evidence through their reproduction of the paper's experiments. While their unstructured results align with the original paper's findings, their structured results paint a dramatically

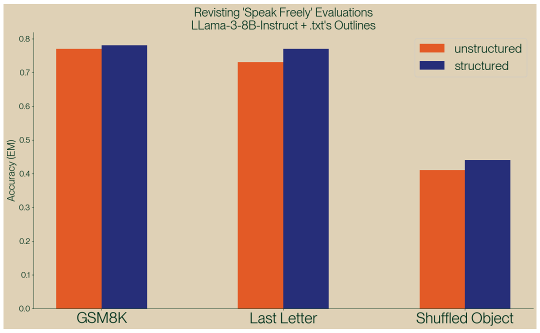


Figure 2.5: Structured vs Unstructured Results by .txt team [18]

different picture - demonstrating that structured generation actually improves performance (see Figure 2.5). The team has made their experimental notebooks publicly available on GitHub for independent verification [18].

[18]: Dottxt (2024), *Say What You Mean: Demos*

.txt team identifies several flaws in the methodology of "Let Me Speak Freely?" that they believe led to inaccurate conclusions:

- ▶ The paper finds that structured output improves performance on classification tasks but doesn't reconcile this finding with its overall negative conclusion about structured output.
- ▶ The prompts used for unstructured generation were different from those used for structured generation, making the comparison uneven.
- ▶ The prompts used for structured generation, particularly in JSON-mode, didn't provide the LLM with sufficient information to properly complete the task.
- ▶ The paper conflates "structured generation" with "JSON-mode", when they are not the same thing.

It is important to note that while .txt provides a compelling and verifiable argument in favor of (proper) structured output generation in LLMs, further research and exploration are needed to comprehensively understand the nuances and trade-offs involved in using structured output across varies formats, LLM tasks and applications.

In summary, the debate surrounding structured output highlights the ongoing challenges in balancing LLM capabilities with real-world application requirements. While structured outputs offer clear benefits in parsing, robustness, and integration, their potential impact on performance, particularly in reasoning tasks is a topic of ongoing debate.

The ideal approach likely involves a nuanced strategy that considers the specific task, the desired level of structure, and the available LLM capabilities. Further research and development efforts are needed to mitigate potential drawbacks and unlock the full potential of LLMs for a wider range of applications.

2.6 Conclusion

Extracting structured output from LLMs is crucial for integrating them into real-world applications. By understanding the challenges and employing appropriate strategies and tools, developers can improve the reliability and usability of LLM-powered systems, unlocking their potential to automate complex tasks and generate valuable insights.

Prompt engineering and the use of fine-tuned models can help control the output of LLMs. However, when strong guarantees are needed, practitioners should consider techniques such as logit post-processing that provides formal guarantees for controlled output generation.

2.7 Acknowledgements

We would like to thank Cameron Pfiffer from the .txt team for his insightful review and feedback.