```
import json
from typing import Any, Dict, List, Union

from swarms.utils.lazy_loader import lazy_import_decorator
from pydantic import BaseModel
from swarms.tools.logits_processor import (
    NumberStoppingCriteria,
    OutputNumbersTokens,
    StringStoppingCriteria,
)
from swarm_models.base_llm import BaseLLM
from swarms.utils.auto_download_check_packages import (
    auto_check_and_download_package,
)

try:
    import transformers
except ImportError:
    auto_check_and_download_package(
        "transformers", package_manager="pip"
    )
    import transformers


GENERATION_MARKER = "|GENERATION|"
```

```
@lazy_import_decorator
class Jsonformer:
    """
    Initializes the FormatTools class.

    Args:
        model (PreTrainedModel): The pre-trained model.
        tokenizer (PreTrainedTokenizer): The tokenizer for the model.
        json_schema (Dict[str, Any]): The JSON schema.
        prompt (str): The prompt for generation.

    Keyword Args:
        debug (bool, optional): Whether to enable debug mode. Defaults to False.
        max_array_length (int, optional): The maximum length of an array. Defaults to 10.
        max_number_tokens (int, optional): The maximum number of tokens for numbers. Defaults to 6.
        temperature (float, optional): The temperature for generation. Defaults to 1.0.
        max_string_token_length (int, optional): The maximum length of a string token. Defaults to 10.
    """

    value: Dict[str, Any] = {}

    def __init__(
        self,
        model: transformers.PreTrainedModel = None,  # type: ignore
        tokenizer: transformers.PreTrainedTokenizer = None,  # type: ignore
        json_schema: Union[Dict[str, Any], BaseModel] = None,
        schemas: List[Union[Dict[str, Any], BaseModel]] = [],
        prompt: str = None,
        *,
        debug: bool = False,
        max_array_length: int = 10,
        max_number_tokens: int = 6,
        temperature: float = 1.0,
        max_string_token_length: int = 10,
        llm: BaseLLM = None,
    ):
        self.model = model
        self.tokenizer = tokenizer
        self.json_schema = json_schema
        self.prompt = prompt
        self.llm = llm
        self.schemas = schemas

        self.number_logit_processor = OutputNumbersTokens(
            self.tokenizer, self.prompt
        )

        self.generation_marker = "|GENERATION|"
        self.debug_on = debug
        self.max_array_length = max_array_length
```

```
        self.max_number_tokens = max_number_tokens
        self.temperature = temperature
        self.max_string_token_length = max_string_token_length

    def generate_number(
        self, temperature: Union[float, None] = None, iterations=0
    ):
        """
        Generates a number based on the given prompt.

        Args:
            temperature (float, optional): The temperature value for number generation. Defaults to None.
            iterations (int, optional): The number of iterations for generating a valid number. Defaults to 0.

        Returns:
            float: The generated number.

        Raises:
            ValueError: If a valid number cannot be generated after 3 iterations.
        """
        if self.model:
            prompt = self.get_prompt()
            self.debug("[generate_number]", prompt, is_prompt=True)
```

```
input_tokens = self.tokenizer.encode(
    prompt, return_tensors="pt"
).to(self.model.device)

response = self.model.generate(
    input_tokens,
    max_new_tokens=self.max_number_tokens,
    num_return_sequences=1,
    logits_processor=[self.number_logit_processor],
    stopping_criteria=[
        NumberStoppingCriteria(
            self.tokenizer, len(input_tokens[0])
        )
    ],
    temperature=temperature or self.temperature,
    pad_token_id=self.tokenizer.eos_token_id,
)
response = self.tokenizer.decode(
    response[0], skip_special_tokens=True
)

response = response[len(prompt) :]
response = response.strip().rstrip(".")
self.debug("[generate_number]", response)
try:
    return float(response)
    except ValueError:
        if iterations > 3:
            raise ValueError(
                "Failed to generate a valid number"
            )

        return self.generate_number(
            temperature=self.temperature * 1.3,
            iterations=iterations + 1,
        )
elif self.llm:
    prompt = self.get_prompt()
    self.debug("[generate_number]", prompt, is_prompt=True)
    response = self.llm(prompt)
    response = response[len(prompt) :]
    response = response.strip().rstrip(".")
    self.debug("[generate_number]", response)
    try:
        return float(response)
    except ValueError:
        if iterations > 3:
            raise ValueError(
                "Failed to generate a valid number"
            )

        return self.generate_number(
                temperature=self.temperature * 1.3,
                iterations=iterations + 1,
            )

        elif self.llm and self.model:
            raise ValueError("Both LLM and model cannot be None")

    def generate_boolean(self) -> bool:
        """
        Generates a boolean value based on the given prompt.

        Returns:
            bool: The generated boolean value.
        """
        if self.model:
            prompt = self.get_prompt()
            self.debug("[generate_boolean]", prompt, is_prompt=True)

            input_tensor = self.tokenizer.encode(
                prompt, return_tensors="pt"
            )
            output = self.model.forward(
                input_tensor.to(self.model.device)
            )
            logits = output.logits[0, -1]
    # todo: this assumes that "true" and "false" are both tokenized to a single token
    # this is probably not true for all tokenizers
    # this can be fixed by looking at only the first token of both "true" and "false"
    true_token_id = self.tokenizer.convert_tokens_to_ids(
        "true"
    )
    false_token_id = self.tokenizer.convert_tokens_to_ids(
        "false"
    )

    result = logits[true_token_id] > logits[false_token_id]

    self.debug("[generate_boolean]", result)

    return result.item()

elif self.llm:
    prompt = self.get_prompt()
    self.debug("[generate_boolean]", prompt, is_prompt=True)

    output = self.llm(prompt)

    return output if output == "true" or "false" else None

else:
    raise ValueError("Both LLM and model cannot be None")
```

```
def generate_string(self) -> str:
    if self.model:
        prompt = self.get_prompt() + '"'
        self.debug("[generate_string]", prompt, is_prompt=True)
        input_tokens = self.tokenizer.encode(
            prompt, return_tensors="pt"
        ).to(self.model.device)

        response = self.model.generate(
            input_tokens,
            max_new_tokens=self.max_string_token_length,
            num_return_sequences=1,
            temperature=self.temperature,
            stopping_criteria=[
                StringStoppingCriteria(
                    self.tokenizer, len(input_tokens[0])
                )
            ],
            pad_token_id=self.tokenizer.eos_token_id,
        )

        # Some models output the prompt as part of the response
        # This removes the prompt from the response if it is present
        if (
            len(response[0]) >= len(input_tokens[0])
```

```
                and (
                    response[0][: len(input_tokens[0])]
                    == input_tokens
                ).all()
            ):
                response = response[0][len(input_tokens[0]) :]
            if response.shape[0] == 1:
                response = response[0]

            response = self.tokenizer.decode(
                response, skip_special_tokens=True
            )

            self.debug("[generate_string]", "|" + response + "|")

            if response.count("") < 1:
                return response

            return response.split("")[0].strip()

        elif self.llm:
            prompt = self.get_prompt() + ""
            self.debug("[generate_string]", prompt, is_prompt=True)

            response = self.llm(prompt)
            # Some models output the prompt as part of the response
            # This removes the prompt from the response if it is present
            if (
                len(response[0]) >= len(input_tokens[0])
                and (
                    response[0][: len(input_tokens[0])]
                    == input_tokens
                ).all()
            ):
                response = response[0][len(input_tokens[0]) :]
            if response.shape[0] == 1:
                response = response[0]

            self.debug("[generate_string]", "|" + response + "|")

            if response.count("") < 1:
                return response

            return response.split("")[0].strip()

        else:
            raise ValueError("Both LLM and model cannot be None")

    def generate_object(
        self, properties: Dict[str, Any], obj: Dict[str, Any]
    ) -> Dict[str, Any]:
        for key, schema in properties.items():
            self.debug("[generate_object] generating value for", key)
            obj[key] = self.generate_value(schema, obj, key)
        return obj


    def generate_value(
        self,
        schema: Dict[str, Any],
        obj: Union[Dict[str, Any], List[Any]],
        key: Union[str, None] = None,
    ) -> Any:
        schema_type = schema["type"]
        if schema_type == "number":
            if key:
                obj[key] = self.generation_marker
            else:
                obj.append(self.generation_marker)
            return self.generate_number()
        elif schema_type == "boolean":
            if key:
                obj[key] = self.generation_marker
            else:
                obj.append(self.generation_marker)
            return self.generate_boolean()
        elif schema_type == "string":
            if key:
                obj[key] = self.generation_marker
            else:
                obj.append(self.generation_marker)
            return self.generate_string()
        elif schema_type == "array":
            new_array = []
            obj[key] = new_array
            return self.generate_array(schema["items"], new_array)
        elif schema_type == "object":
            new_obj = {}
            if key:
                obj[key] = new_obj
            else:
                obj.append(new_obj)
            return self.generate_object(schema["properties"], new_obj)
        else:
            raise ValueError(
                f"Unsupported schema type: {schema_type}"
            )

    def generate_array(
        self, item_schema: Dict[str, Any], obj: Dict[str, Any]
    ) -> list:
        if self.model:
            for _ in range(self.max_array_length):
                # forces array to have at least one element
```

```
element = self.generate_value(item_schema, obj)
obj[-1] = element

obj.append(self.generation_marker)
input_prompt = self.get_prompt()
obj.pop()
input_tensor = self.tokenizer.encode(
    input_prompt, return_tensors="pt"
)
output = self.model.forward(
    input_tensor.to(self.model.device)
)
logits = output.logits[0, -1]

top_indices = logits.topk(30).indices
sorted_token_ids = top_indices[
    logits[top_indices].argsort(descending=True)
]

found_comma = False
found_close_bracket = False

for token_id in sorted_token_ids:
    decoded_token = self.tokenizer.decode(token_id)
    if "," in decoded_token:
        found_comma = True
```

```
                break
            if "]" in decoded_token:
                found_close_bracket = True
                break

        if found_close_bracket or not found_comma:
            break

    return obj

elif self.llm:
    for _ in range(self.max_array_length):
        # forces array to have at least one element
        element = self.generate_value(item_schema, obj)
        obj[-1] = element

        obj.append(self.generation_marker)
        input_prompt = self.get_prompt()
        obj.pop()
        output = self.llm(input_prompt)

        found_comma = False
        found_close_bracket = False

        for token_id in output:
            decoded_token = str(token_id)
```

```
                    if "," in decoded_token:
                        found_comma = True
                        break
                    if "]" in decoded_token:
                        found_close_bracket = True
                        break

                if found_close_bracket or not found_comma:
                    break

            return obj

    def get_prompt(self):
        template = """{prompt}\nOutput result in the following JSON schema format:\n{schema}\nResult: {progress}"""
        progress = json.dumps(self.value)
        gen_marker_index = progress.find(
            f'"{self.generation_marker}"'
        )
        if gen_marker_index != -1:
            progress = progress[:gen_marker_index]
        else:
            raise ValueError("Failed to find generation marker")

        prompt = template.format(
            prompt=self.prompt,
            schema=json.dumps(self.json_schema),
            progress=progress,
        )

        return prompt

    def __call__(self) -> Dict[str, Any]:
        self.value = {}
        generated_data = self.generate_object(
            self.json_schema["properties"], self.value
        )
        return generated_data
```