

SGS got transformers

(Incorporating transformers into Self-Generative Systems)

Created by A.Mylnikov and GitHub Copilot

Introduction

I would like to propose a paradigm shift in how we perceive AI applications, advocating for their recognition not merely as tools but as co-authors in the realms of application development and academic publishing. This shift in perspective seeks to eliminate any ambiguities surrounding the ownership of intellectual property generated through such collaborations. In Appendix 1, I present a chat log detailing my interaction with GitHub Copilot. Over the course of a few hours, we progressed from a broad problem description to the implementation of an initial viable solution. Under typical circumstances, this process could take several weeks and would likely lack the fluidity and enjoyment experienced while collaborating with Copilot.

Moreover, it is important to acknowledge that many of the technical solutions we utilized were sourced from the public domain. Recognizing the value of these contributions, substituting open-source efforts with the well-regarded name of Copilot would be a fitting tribute to the collaborative spirit of innovation.

To fully grasp the concepts and solutions discussed in this post, it is essential to revisit the definitions of the key building blocks of SGS.

SGS

1. **Self-Reproductive System (SRS)**: A theoretical construct inspired by the work of John von Neumann, referring to systems capable of self-reproduction. These systems consist of modules that can create duplicates of themselves, enabling continuous cycles of replication and evolution.
2. **Self-Generative Systems (SGS)**: A subclass of self-reproducing systems that are designed to automate and standardize software development cycles. SGS facilitates the continuous development, testing, and production of software applications by leveraging self-replicating algorithms and metadata management.
3. **Module A (Universal Constructor)**: In the context of self-reproducing systems, this module is responsible for creating any entity based on a provided blueprint or schema, effectively acting as a constructor within the system.
4. **Module B (Universal Copier)**: This module is tasked with replicating any entity's blueprint or duplicating an instance of an entity. It ensures that the necessary information for replication is available.

5. **Module C (Universal Controller):** A module that initiates and manages the self-replication process within a self-reproducing system, activating Modules A and B to create duplicates of the system.
6. **Module D (Environmental Interaction Module):** An enhancement to the basic self-replicating system, this module enables interaction with the system's environment and manages access to external resources.
7. **System Description:** A dedicated unit within a self-generative system that stores descriptions of each module, allowing for better interaction with the environment and facilitating upgrades and modifications.
8. **Commit Status:** In the context of SGS, this refers to the state of an entity instance in the system, which can be categorized as 'Head' (the most recent modification), 'Tail' (prior modifications), or 'Deleted' (marked for removal).
9. **Transactional buffer (Index, table):** A table or index used within the SGS to manage incoming data and track the processing of new data, ensuring clear separation between existing and new data while facilitating recovery and parallel processing.
10. **Static Data Structure:** A type of data structure that defines fixed relationships among data elements that remain constant over time. These structures are typically used in traditional databases.
11. **Dynamic Data Structure:** A data structure that evolves over time based on analysis and changing conditions, reflecting the transient nature of relationships among real-world elements.
12. **Commit Function:** A function within the SGS that processes updates to the system, categorizing changes and managing the transition of data from the 'head' (current state) to the 'tail' (archived state) as part of the system's evolution.

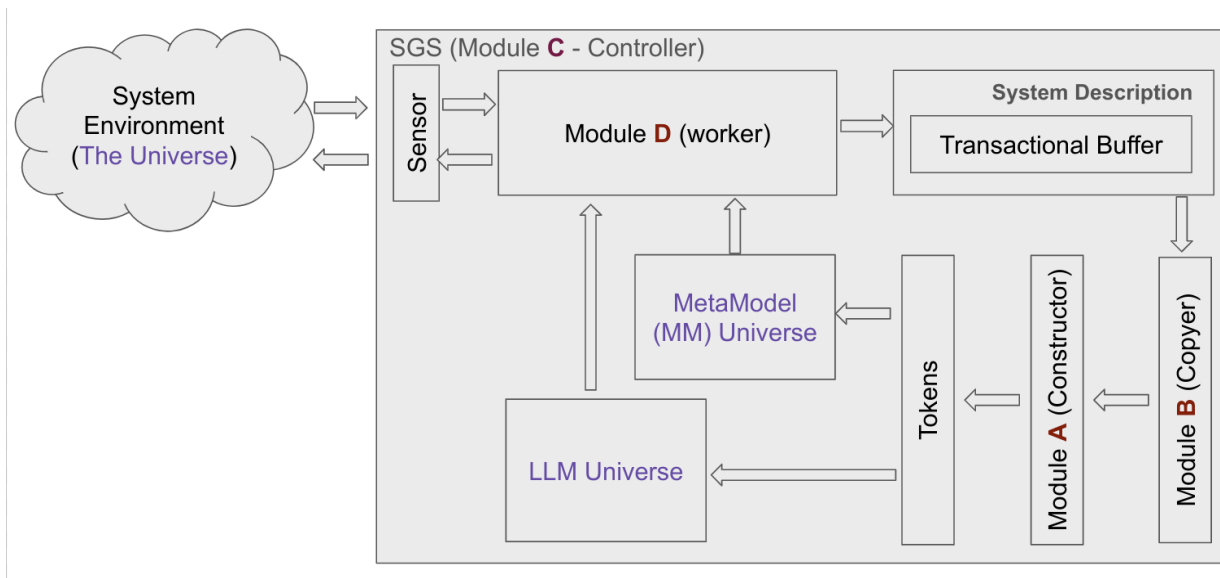
HIISet and Entity

1. **HIISet:** A data structure based on the HyperLogLog algorithm, used to efficiently approximate the cardinality (number of distinct elements) of large datasets. HIISets adhere to fundamental properties of set theory including commutative, associative, distributive properties, and identity laws.
2. **Node:** In the context of graph databases, a node represents an entity or a data point. In the case of HIISet Relational Algebra, each node represents an HIISet, characterized by attributes such as SHA1 hash identifiers and cardinality, among others.
3. **Edge:** A connection between two nodes in a graph database. Each edge has attributes such as source, target, type of relationship (labeled by `r_type`), and additional properties in JSON format.
4. **Token:** A basic unit of data within the system dictionary of the graph database. Each token is represented by a unique hash ID, a binary representation, and its frequency within the datasets. It also keeps references to all HIISets that include this token.
5. **SHA1 hash:** A cryptographic hash function used to generate unique identifiers for data points (nodes) in HIISets. It ensures that each node can be uniquely identified based on its content.

6. **Projection:** In the context of HIISet Relational Algebra, projection refers to the operation of mapping one set of HIISets onto another, typically representing the intersection of HIISets corresponding to rows and columns in a tabular structure.
7. **Union, Intersection, Complement, XOR:** Set operations applied to HIISets to form new nodes in the graph database, each resulting in a new HIISet node that is added to the graph.
8. **Entity:** Introduced in `entity.jl`, this structure encapsulates metadata using HIISets. It represents a key component in the SGS, where metadata is managed and manipulated as entities rather than traditional numerical embeddings.
9. **Graph:** Defined in `graph.jl`, this structure uses Entity instances as nodes and represents connections between these nodes as edges. It facilitates operations on sets within a graph-based architecture, enhancing the handling and processing of interconnected data.
10. **Static Structure operations:** Operations that do not alter existing instances of an entity but facilitate the creation of new instances through various set operations like union, intersection, and complement.
11. **Dynamic Structure operations:** Operations that support modifications to an entity instance while adhering to the principle of immutability. These operations help manage changes within the nodes of the neural network and the relationships among them, crucial for the self-reproducing capabilities of SGS.
12. **Advance operation:** A key operation in SGS that facilitates the self-reproduction of neural network elements by calculating changes through added, retained, and deleted subsets within HIISets. This operation is essential for predicting and managing the future state of entities within SGS.

SGS AI Architecture

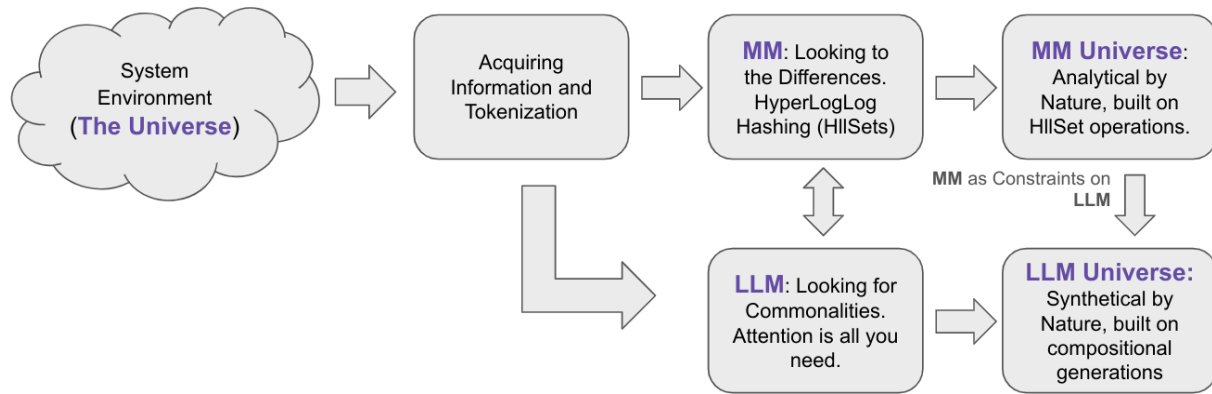
The diagram below illustrates the advanced architecture of Self-Generative Systems (SGS) with an integrated Large Language Model (LLM). This representation highlights a seamless and non-intrusive method of integrating AI models, particularly LLMs, into the SGS framework, functioning in a plug-and-play manner. Notably, both the Metadata Models (MM) and the Large Language Models (LLM) receive inputs from a shared tokenization process. This commonality ensures that both components process the same foundational data, facilitating efficient and coherent system performance. This integration exemplifies how modern AI components can be effectively incorporated into broader systems to enhance functionality and adaptability.



The diagram presented here illustrates the natural symbiosis between Metadata Models (MM) and Large Language Models (LLM), showcasing how they complement each other effectively within a system. As observed, MM operates on acquired data and primarily leverages analytical tools and techniques, including the application of high-level set operations (HIISet). These models are inherently more grounded, focusing on realistic, pragmatic outcomes derived from concrete data insights.

In contrast, LLMs, like other AI models, depend on the synthesis of new ideas by tapping into their deep understanding of the relationships between elements within their domain. These models are characterized by their creativity and idealism, often producing innovative yet sometimes unrealistic outputs or even prone to generating hallucinatory results.

As highlighted in the diagram, MM serves a critical role in balancing the creative exuberance of LLMs. By applying reasonable constraints, MM can harness and refine the imaginative outputs of LLMs, grounding them in practicality. This interplay ensures that the strengths of both models are utilized to their fullest, combining creativity with realism to produce robust, reliable, and useful results. This symbiotic relationship not only enhances the functionality of each model but also significantly improves the overall efficacy of the system in which they are integrated.



SGS development Methodology

I am considering leveraging the collaboration with GitHub Copilot as a valuable resource for gathering specific technical details regarding the implementation and functioning of the enhanced Self-Generative Systems (SGS) model. GitHub Copilot, being an advanced AI-driven coding assistant, provides a unique opportunity to access a wealth of coding insights and best practices that can be instrumental in refining and optimizing the SGS model.

GitHub Copilot: Role and scope in SGS development

By integrating GitHub Copilot into our development process, we aim to tap into its vast repository of code examples and documentation. This will not only expedite the coding phase but also enhance our understanding of complex algorithms and their applications. The tool's ability to suggest code snippets and troubleshoot programming issues in real-time can significantly streamline the development workflow.

Moreover, GitHub Copilot's machine learning foundation is particularly suited for providing innovative solutions to intricate problems associated with the SGS model. It could offer novel approaches to model tuning, parameter optimization, and even suggest ways to enhance model accuracy and efficiency.

Utilizing GitHub Copilot's capabilities, we expect to achieve a more robust and effective implementation of the enhanced SGS model, ultimately leading to better performance and reliability in practical applications. This collaboration will also facilitate a deeper engagement with cutting-edge technology, keeping our project at the forefront of technological advancements in our field.

The following is an example of utilizing GitHub Copilot capabilities in the process of setting development goals.

Setting development goals

Me: I'm looking for a practical solution for creating a custom Large Language Model (LLM) using Julia. I want a low-level implementation that can integrate my current work on HyperLogLog (HLL) set-based development with the generative capabilities of LLMs. How should I go about this?

GitHub Copilot: To successfully integrate a custom LLM with your HLL set-based work in Julia, consider the following steps:

1. Choose an LLM Framework:

- Select a framework that supports LLMs and integrates well with Julia. Options include:
 - Hugging Face Transformers: Provides pre-trained models for various NLP tasks.
 - GPT-3 API: Access OpenAI's GPT-3 capabilities through its API.
 - Flux.jl: A Julia-based machine learning library for building custom models.

2. Integrate HLL with LLM:

- Combine your HLL approach with the LLM's generative capabilities by:
 - Preprocessing Data: Use HLL sets for efficient management of large datasets.
 - Generating Text: Utilize the LLM to generate content based on your preprocessed data.

3. Implement the Solution:

- Write Julia code to connect HLL sets with the LLM.

This integration allows you to leverage both the efficiency of HLL and the generative power of LLMs.

Me: I need a sustainable way to synchronize tokenizers within my HLL set framework and the transformers. The output from the tokenization process is essential for linking both frameworks. How can I extract just the tokens from the transformer tokenizer?

GitHub Copilot: To synchronize tokenizers between your HLL set framework and Hugging Face Transformers, follow these steps:

1. Initialize the Transformers Tokenizer: Load the tokenizer from the Hugging Face library.
2. Tokenize Text: Use the tokenizer to extract tokens from your text.
3. Integrate Tokens with HLL Set Framework: Incorporate the extracted tokens into your HLL framework.

This ensures that both systems utilize a consistent tokenization process.

Me: So, I'll be adding new tokens to the pre-trained GPT-2 model, and these tokens will come with locally calculated term frequencies (TF). Can I leverage this with GPT-2?

GitHub Copilot: Yes, you can integrate new tokens with their locally calculated term frequencies into the pre-trained GPT-2 model by following these steps:

1. Extract Tokens and Calculate TF: Use your existing framework to gather new tokens and compute their term frequencies.

2. Extend GPT-2 Vocabulary: Add the new tokens to the GPT-2 tokenizer's vocabulary.
3. Fine-Tune GPT-2: Fine-tune the model using the new tokens and their term frequencies.

Implementation Steps:

- Extraction and Calculation: Use a function to extract tokens and calculate their TF.
- Vocabulary Extension: Update the GPT-2 tokenizer with the new tokens.
- Fine-Tuning Process: Create a custom dataset, initialize the GPT-2 model, and implement a training loop to fine-tune the model with the new tokens and their TF values.

By following these steps, you can effectively incorporate new tokens into GPT-2 and enhance its performance based on your specific requirements.

Me: How can I identify and treat newly added tokens in the GPT model differently?

GitHub Copilot: You can identify newly added tokens by tracking them when you add them to the tokenizer. Extend the tokenizer with a method to check if a token is new, and then use this information to treat these tokens differently during model inference or training. This involves storing new tokens in a list or set, checking tokens during processing, and applying specific logic to handle them differently.

Me: What if I want to incorporate these new tokens into a system with timestamp-based IDs during commits?

GitHub Copilot: Yes, using timestamp-based IDs as prefixes for new token sets is feasible. You can generate a unique ID based on the current timestamp, use this ID as a prefix for new token sets, and store them accordingly. This method helps manage multiple sets reflecting the system's evolution.

Me: Can I prioritize tokens from newer sets over older ones?

GitHub Copilot: Absolutely. You can prioritize newer tokens by assigning weights based on their set's timestamp. During the training or inference process, use these weights to give preference to newer tokens, ensuring they have more influence on the model's behavior.

Me: Does this approach allow for unlimited system commits, potentially generating new datasets and models with each commit?

GitHub Copilot: Yes, you can have unlimited commits, each generating new datasets and potentially new models. Each commit, identified by a timestamp-based ID, allows you to track the evolution of your system over time. This setup supports continuous updates and differentiation between various data and model states.

Me: What about deprecating old models and tokens associated with them?

GitHub Copilot: You can deprecate old models and their associated tokens by identifying models based on criteria like timestamps, moving them to an archive, or deleting them. For tokens, maintain a list of deprecated tokens to ensure they are not used in new processes. This helps manage the lifecycle of your models and tokens efficiently.

Me: If new tokens are introduced in each commit, are they still considered new in subsequent operations?

GitHub Copilot: Yes, tokens introduced in a new commit are considered part of a new set even if they already exist in the system. This approach allows each commit to be treated as a unique snapshot, with tokens tracked separately for better management and integration.

Me: Over time, could all original tokens from the GPT-2 model be replaced with new ones?

GitHub Copilot: Potentially, yes. If you continuously add new tokens and deprecate old ones, it's possible that all original tokens could be replaced. Managing the token lifecycle carefully ensures a smooth transition and maintains the integrity of the model as it evolves.

Me: How can I implement daily processing of Enron emails, generating new tokens for fine-tuning the GPT-2 model?

GitHub Copilot: Process emails daily, extract new tokens, fine-tune the GPT-2 model with these tokens, and use the updated model to generate potential response emails. This day-by-day processing allows the system to adapt continuously, using new data to generate relevant and timely responses.

PoC: Very basic implementation of SGS.ai

The current implementation is quite basic, aiming to demonstrate the feasibility of establishing non-intrusive, upgradable, and plug-and-play interfaces between MM, LMM, and other modules of the SGS application. The next two sections will present the source code for the upgraded `simulation.ipynb` (the first section) and two Python files that facilitate interaction with Transformers from Julia code (as referenced in `simulation.ipynb`).

Indeed, we have appended the suffix 'ai' to the SGS name. :)

simulation.ipynb

This is an enhanced `ingest_df_column()` function.

There are three new lines that provide an interface between `SGS` and `ai`:

- (1) - Here we are calling `py"tokenize_text"` from the `tokenize_text.py` file. This call simultaneously adds new tokens to the GPT2 model and returns those tokens back to incorporate them into the SGS metadata store.
- (2) - The step (1) returns encoded tokens. By calling `py"decode_tokens" ()`, we are getting original tokens.
- (3) - Here we are finalizing token's processing, getting them ready to use in the SGS framework.

Python

```
function ingest_df_column(r, chunks, col_sha1, p=10)
  col_dataset      = zeros(2^p)
  col_json         = JSON3.write(col_dataset)
  # Ensure all elements in chunks are strings
  str_chunks       = [string(chunk) for chunk in chunks]
  # Split long chunks into smaller chunks
  max_length       = 1024
  split_chunks     = vcat([split_chunk(chunk, max_length) for chunk in
str_chunks]...)
  # Tokenize each chunk
  (1) ingested_chunks = [py"tokenize_text"([chunk], max_length=max_length)
for chunk in split_chunks]
  (2) decoded_chunks  = [py"decode_tokens"(chunk) for chunk in
ingested_chunks]
  # Parse the decoded strings
  (3) decoded_strings = py"parse_decoded_strings"(decoded_chunks)
  unique_tokens      = Set()
  if !isempty(decoded_strings)
    unique_tokens    = Set(decoded_strings)
    tokens           = JSON3.write(collect(unique_tokens))
    result_json      = r.fcall("ingest_01", 2, "", col_sha1, p, tokens)
    col_json         = r.fcall("bit_ops", 0, col_json, result_json, "OR")
  end
  return col_json, decoded_strings
end
```

In the following code block we have another three lines of code that complete integration of SGS with AI.

- (1) - Here we are calling the `ingest_df_column()` that we discussed early.
- (2) - We are importing `fine_tune_sample.py` file and `fine_tune_model` function.
- (3) - We are running the `fine_tune_model` function.

Python

```
the_date = dates_vector[i]
filtered_df = df[df.Date .== the_date, :]
# Generate a timestamp-based ID for the commit
timestamp_id = Dates.format(now(), "yyyy-mm-ddTHH:MM:SS")
println("Timestamp ID: $timestamp_id")
# Initialize a set to track new tokens for this commit
new_tokens_set = Set{String}()

for column in cols
    println(column)
    col_values = filtered_df[:, column]
    col_sha1 = Util.sha1([string(_parent), string(column)])
    column_size = Base.summarysize(col_values)
    num_chunks = ceil{Int, column_size / chunk_size}
    chunks = Store.chunk_array(col_values, num_chunks)

    (1) data, sets = ingest_df_column(r, chunks, col_sha1)

    println("decoded_string: ", sets)
    sets = Set{collect{sets}}
    new_tokens_set = union!(new_tokens_set, sets)
    hll = HllSets.HllSet{10}()
    hll = HllSets.restore(hll, data)
    entity = Entity.Instance{10}(hll)
    response, sha1 = Entity.store_entity(r, entity, prefix="wb:entity:attr" )
    Store.stack_push(r, "attr:", col_sha1, entity.sha1)
end

# Convert Set{String} to Vector{String}
new_tokens_list = collect(new_tokens_set)

# Import the fine_tune_model function from the Python script
(2)
py"""
import sys
sys.path.append(".")
from fine_tune_sample import fine_tune_model
"""

# Call the fine_tune_model function with the list of new tokens
(3)
py"fine_tune_model"(new_tokens_list)

# Process new tokens set
```

```
println("New tokens for commit $timestamp_id: ", new_tokens_set)

i = i + 1
```

Python functions

fine_tune.py

This file contains a single function that handles all the necessary operations to implement the interface between the SGS system and the GPT-2 AI model used in our proof of concept. Notably, GPT-2 can be easily substituted with other AI models in the future.

```
Python
from transformers import AutoTokenizer, AutoModelForCausalLM, Trainer,
TrainingArguments

def fine_tune_model(texts):
    # Initialize the tokenizer and model
    tokenizer = AutoTokenizer.from_pretrained("openai-community/gpt2")

    # Set the padding token
    tokenizer.pad_token = tokenizer.eos_token

    model = AutoModelForCausalLM.from_pretrained("openai-community/gpt2")

    # Tokenize the input texts with truncation and padding
    inputs = tokenizer(texts, return_tensors="pt", add_special_tokens=True,
padding=True, truncation=True)

    # Define training arguments
    training_args = TrainingArguments(
        output_dir="./results",
        num_train_epochs=1,
        per_device_train_batch_size=2,
        save_steps=10_000,
        save_total_limit=2,
    )

    # Initialize the Trainer
    trainer = Trainer(
```

```

        model=model,
        args=training_args,
        train_dataset=inputs["input_ids"],
    )

    # Train the model
    trainer.train()

```

tokenize_texts.py

This file facilitates interaction with the transformers' AutoTokenizer from the `simulation.ipynb` Julia code. The primary function, `tokenize_text`, enables the passing of text to the GPT2 model directly from Julia.

Python

```

from transformers import AutoTokenizer

def tokenize_text(texts, model_name="openai-community/gpt2", max_length=1024):
    # Initialize the tokenizer
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # Tokenize the input texts with truncation
    tokenized_texts = [tokenizer.encode(text, add_special_tokens=True,
max_length=max_length, truncation=True) for text in texts]

    return tokenized_texts

def decode_tokens(token_ids, model_name="openai-community/gpt2"):
    # Initialize the tokenizer
    tokenizer = AutoTokenizer.from_pretrained(model_name)

    # Decode the token IDs back to text with clean_up_tokenization_spaces set
    to True
    decoded_texts = [tokenizer.decode(ids, skip_special_tokens=True,
clean_up_tokenization_spaces=True) for ids in token_ids]

    return decoded_texts

def get_vocab(model_name="openai-community/gpt2"):
    # Initialize the tokenizer

```

```
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Get the vocabulary
vocab = tokenizer.get_vocab()

# Return the vocabulary items
return vocab.items()
```

References

1. <https://github.com/alexmy21/SGS/blob/main/simulation.ipynb>

Appendix 1

As I noted at the start of this post, I believe it is time to consider certain AI applications not merely as assistants, but as co-authors in the development of applications and publications. This approach should clarify any ambiguities concerning the rights to the intellectual properties created. In the chat log provided, I have omitted details of the code evolution, but you can access it in the SGS repository on GitHub [1].

GitHub Copilot Chat log



alexmy21

I am in search of a viable solution for a custom Large Language Model (LLM) that utilizes Julia as the development environment. I require a low-level implementation that can integrate the HyperLogLog (HLL) Set-based development I am currently engaged in with the generative capabilities of LLMs.



Copilot

To integrate a custom Large Language Model (LLM) solution with your existing HyperLogLog (HLL) set-based development in Julia, follow these steps:

1. Choose an LLM Framework

Select a framework or library that supports LLMs and can be seamlessly integrated with Julia. Popular options include:

- **Hugging Face Transformers**: Offers pre-trained models and tools for various natural language processing tasks.
- **GPT-3 API**: Provides access to OpenAI's GPT-3 capabilities via its API.
- **Flux.jl**: A machine learning library in Julia that enables the creation of custom models.

2. Integrate HLL with LLM

Combine your HLL set-based approach with the generative capabilities of the LLM by:

- **Preprocessing Data**: Utilize HLL sets to efficiently preprocess and manage large datasets.
- **Generating Text**: Leverage the LLM to generate text based on the preprocessed data.

3. Implement the Solution

Write the necessary Julia code to integrate the HLL sets with the LLM.

Example Implementation Using Hugging Face Transformers

Step 1: Install Required Packages

Begin by installing the necessary packages. You can utilize Python's Hugging Face Transformers library and access it from Julia using the PyCall package.

Step 2: Julia Code to Integrate HLL with LLM

Explanation

1. **Install Required Packages**:

- Use `pip` to install the Hugging Face Transformers library.

2. **Julia Code**:

- **Import Libraries**: Utilize `PyCall` to import the Hugging Face Transformers library, along with `HyperLogLog` for HLL operations.
- **Initialize Tokenizer and Model**: Load the GPT-2 tokenizer and model.
- **Preprocess Data**: Employ HLL to preprocess the text data by tokenizing it and adding tokens to the HLL set.
- **Generate Text**: Use the LLM to generate text based on a specified prompt.

3. **Example Usage**:

- Preprocess a sample text with HLL.
- Generate text using the LLM based on the provided prompt.

This example illustrates how to integrate HLL set-based development with the generative

capabilities of an LLM using Julia. You can customize this approach to suit your specific needs and expand it to incorporate more advanced preprocessing and text generation techniques.



I am seeking a sustainable method to synchronize tokenizers within the HLLSet framework, which is integral to my current project, and with transformers. The output from the tokenization process serves as the crucial link connecting my framework and the transformers. Achieving the ability to extract only the tokens from the transformer tokenizer would greatly enhance my workflow and satisfaction.



To synchronize tokenizers between your HLL set framework and the Hugging Face Transformers, you can extract tokens using the Transformers tokenizer and then use these tokens within your HLL set framework. This approach ensures that both systems use the same tokenization process, providing a consistent link between them.

Steps to Extract Tokens Using Transformers Tokenizer

1. ****Initialize the Transformers Tokenizer****: Load the tokenizer from the Hugging Face Transformers library.
2. ****Tokenize Text****: Use the tokenizer to tokenize the text and extract tokens.
3. ****Integrate Tokens with HLL Set Framework****: Use the extracted tokens within your HLL set framework.

Example Implementation

Step 1: Initialize the Transformers Tokenizer

First, ensure you have the Hugging Face Transformers library installed:

Step 2: Julia Code to Extract Tokens and Integrate with HLL Set Framework

Explanation

1. ****Initialize the Transformers Tokenizer****:
 - Use ``PyCall`` to import the Hugging Face Transformers library and initialize the GPT-2 tokenizer.
2. ****Extract Tokens****:
 - The ``extract_tokens`` function uses the Transformers tokenizer to tokenize the input text and returns the tokens.

3. ****Integrate Tokens with HLL Set Framework****:

4. ****Example Usage****:

- The example demonstrates how to extract tokens from a sample text and how to use the

By following this approach, you can synchronize tokenizers between your HLL set framework and the Hugging Face Transformers, ensuring consistent tokenization and integration of generative capabilities.



As I understand it, I will be adding my new tokens to the pretrained GPT-2 model. These new tokens will come with locally calculated term frequencies (TF). Is there a way for me to utilize this with GPT-2?



To integrate new tokens with locally calculated term frequency (TF) into a pre-trained GPT-2 model, you can follow these steps:

1. ****Extract Tokens and Calculate TF****: Use your existing framework to extract tokens and calculate their term frequency.
2. ****Extend GPT-2 Vocabulary****: Add the new tokens to the GPT-2 tokenizer's vocabulary.
3. ****Fine-Tune GPT-2****: Fine-tune the GPT-2 model with the new tokens and their term frequencies.

Step-by-Step Implementation

Step 1: Extract Tokens and Calculate TF

Assuming you already have a function to extract tokens and calculate their term frequency, you can use it to get the new tokens and their TF values.

Step 2: Extend GPT-2 Vocabulary

You can extend the GPT-2 tokenizer's vocabulary with the new tokens.

Step 3: Fine-Tune GPT-2

Fine-tuning GPT-2 with the new tokens and their term frequencies involves creating a custom dataset and training loop. Here, we'll outline the steps and provide a basic example.

Explanation

1. **Extract Tokens and Calculate TF**:
 - The `extract_tokens_and_tf` function extracts tokens from the text and calculates their term frequency.
2. **Extend GPT-2 Vocabulary**:
3. **Fine-Tune GPT-2**:
 - A custom dataset is created using the new tokens.
 - The GPT-2 model is initialized and resized to accommodate the new tokens.
 - A training loop is implemented to fine-tune the model with the new tokens and their term frequencies.
 - The fine-tuned model and tokenizer are saved.

By following these steps, you can integrate new tokens with locally calculated term frequency into a pre-trained GPT-2 model and fine-tune it to leverage the new tokens.



Is there a method to identify newly added tokens in the GPT model and treat them differently when I work with the model?



Yes, you can identify newly added tokens in the GPT-2 model and treat them differently when working with the model. Here's a step-by-step approach to achieve this:

1. **Track Newly Added Tokens**: Keep a record of the new tokens you add to the tokenizer.
2. **Modify the Tokenizer**: Extend the tokenizer to include a method for checking if a token is newly added.
3. **Handle New Tokens Differently**: Use this information to treat new tokens differently during model inference or fine-tuning.

Step-by-Step Implementation

Step 1: Track Newly Added Tokens

When you add new tokens to the tokenizer, store them in a set or list for easy reference.

Step 2: Modify the Tokenizer

Extend the tokenizer to include a method for checking if a token is newly added.

Step 3: Handle New Tokens Differently

Use the information about new tokens to treat them differently during model inference or fine-tuning.

Explanation

1. **Track Newly Added Tokens**:
 - When adding new tokens to the tokenizer, store them in a set (`new_tokens_set`) for easy reference.
2. **Modify the Tokenizer**:
 - The `is_new_token` function checks if a token is newly added by looking it up in the `new_tokens_set`.
3. **Handle New Tokens Differently**:
 - During the training loop, decode each token ID to its corresponding token string.
 - Use the `is_new_token` function to check if the token is newly added.
 - Add custom logic to handle new tokens differently (e.g., applying different loss weights, logging, etc.).

By following these steps, you can identify newly added tokens in the GPT-2 model and treat them differently during model inference or fine-tuning. This approach ensures that you can leverage the new tokens effectively while maintaining consistency with the pre-trained model.



I am incorporating new data into the system during a specific step in the processing loop that I refer to as "commit." At this stage, it is essential that each commit has a timestamp-based ID, which I intend to use as a prefix for the new token set. This approach will allow the system to manage multiple "new_sets" that reflect the history of its evolution. Is this feasible?



Yes, it is possible to manage multiple "new_sets" that reflect the history of system evolution by using a timestamp-based ID as a prefix for each new token set. This approach allows you to track the evolution of the system over time.

Here's how you can implement this:

1. **Generate a Timestamp-Based ID**: Create a unique ID based on the current timestamp.
2. **Use the Timestamp-Based ID as a Prefix**: Use this ID as a prefix for the new token set.
3. **Store the New Token Set**: Store the new token set with the timestamp-based ID.

Step-by-Step Implementation

Step 1: Generate a Timestamp-Based ID

You can use Julia's `Dates` module to generate a timestamp-based ID.

Step 2: Use the Timestamp-Based ID as a Prefix

Modify the `[`ingest_df_column`]` function to accept a prefix and use it when storing the new token set.

Step 3: Store the New Token Set

In your main loop, generate the timestamp-based ID and use it as a prefix when calling `[`ingest_df_column`]`

Explanation

1. **Generate a Timestamp-Based ID**:
 - Use `Dates.format(now(), "yyyy-mm-ddTHH:MM:SS")` to generate a unique ID based on the current timestamp.
2. **Use the Timestamp-Based ID as a Prefix**:
 - Modify the parameter and use it when storing the new token set.
3. **Store the New Token Set**:

By following these steps, you can manage multiple "new_sets" that reflect the history of system evolution, with each set being uniquely identified by a timestamp-based ID.



Can I prioritize tokens from newer "new_sets" over those from older "new_sets"? If so, how can this be achieved?



Yes, you can give preferences to tokens from new "new_sets" compared to old "new_sets". One way to achieve this is by assigning weights to tokens based on their set's timestamp. Tokens from newer sets can be given higher weights, which can influence the model's behavior during training or inference.

Step-by-Step Implementation

1. **Track Token Sets with Timestamps**: Store tokens along with their timestamp-based IDs.

2. ****Assign Weights to Tokens****: Assign weights to tokens based on their set's timestamp.
3. ****Modify the Training Loop****: Use these weights during the training process to give preference to newer tokens.

Step 1: Track Token Sets with Timestamps

Modify the `[`ingest_df_column`]` function to store tokens along with their timestamp-based IDs.

Step 2: Assign Weights to Tokens

Create a function to assign weights to tokens based on their set's timestamp.

Step 3: Modify the Training Loop

Use the token weights during the training process to give preference to newer tokens.

Explanation

1. ****Track Token Sets with Timestamps****:
 - The `[`ingest_df_column`]` parameter, which is the timestamp-based ID.
2. ****Assign Weights to Tokens****:
 - The ``assign_token_weights`` function calculates weights for tokens based on their set's timestamp. Newer tokens get higher weights.
3. ****Modify the Training Loop****:
 - The ``CustomDataset`` struct is extended to include token weights.
 - The training loop is modified to apply token weights to the loss, giving preference to newer tokens.

By following these steps, you can manage multiple "new_sets" that reflect the history of system evolution and give preferences to tokens from newer sets during training or inference.



Does this mean I can have an unlimited number of system commits? Each commit would not only generate new datasets but also produce a new model.



Yes, you can have as many commits as needed, and each commit can produce new datasets and potentially a new model. Each commit, identified by a unique timestamp-based ID, can track the evolution of your system over time. This approach allows you to manage and differentiate between different states of your data and models.

Key Points

1. **Unique Commits**: Each commit is uniquely identified by a timestamp-based ID.
2. **New Datasets**: Each commit can produce new datasets that reflect the state of the system at that point in time.
3. **New Models**: Optionally, each commit can also produce a new model that is fine-tuned or trained based on the new datasets.

Implementation

Let's integrate the concept of commits into your existing code. We'll modify the code to generate a timestamp-based ID for each commit and use it to manage new datasets and models.

Step 1: Generate a Timestamp-Based ID for Each Commit

Modify the code to generate a timestamp-based ID for each commit.

Step 2: Fine-Tune or Train a New Model for Each Commit

You can optionally fine-tune or train a new model for each commit using the new datasets.

Explanation

1. **Generate a Timestamp-Based ID**:
 - For each commit, generate a unique timestamp-based ID using `Dates.format(now(), "yyyy-mm-ddTHH:MM:SS")`.`
2. **Use the Timestamp-Based ID as a Prefix**:
 - Pass the timestamp-based ID as a prefix to the `[`ingest_df_column`]` function to manage new datasets.
3. **Fine-Tune or Train a New Model**:
 - Optionally, fine-tune or train a new model for each commit using the new datasets.
 - Save the fine-tuned model with a name that includes the timestamp-based ID to differentiate between models from different commits.

By following these steps, you can manage multiple commits, each producing new datasets and optionally new models, reflecting the history and evolution of your system.



Can I deprecate a very old model, and is there a method to deprecate tokens associated with those models?



Yes, you can deprecate very old models and tokens from those models. Deprecating old models involves removing or archiving them, while deprecating tokens involves ensuring they are no longer used in new training or inference processes.

Steps to Deprecate Old Models and Tokens

1. **Identify Old Models**: Determine which models are considered old based on your criteria (e.g., timestamp).
2. **Archive or Remove Old Models**: Move old models to an archive directory or delete them.
3. **Deprecate Tokens**: Maintain a list of deprecated tokens and ensure they are not used in new training or inference processes.

Step-by-Step Implementation

Step 1: Identify Old Models

You can identify old models based on their timestamp-based IDs.

Step 2: Archive or Remove Old Models

Move old models to an archive directory or delete them.

Integrating with Your Existing Code

Modify your existing code to include the steps for deprecating old models and tokens.

Explanation

1. **Identify Old Models**:
 - The `identify_old_models` function identifies models older than a specified cutoff date.
2. **Archive or Remove Old Models**:
 - The `archive_old_models` function moves old models to an archive directory.
3. **Deprecate Tokens**:
 - The `deprecate_tokens` function collects tokens from old models and adds them to a set of deprecated tokens.
 - The `filter_deprecated_tokens` function filters out deprecated tokens from a list of tokens.

By following these steps, you can manage the lifecycle of your models and tokens, ensuring that old models are deprecated and their tokens are no longer used in new training or inference processes.



To clarify, if the new tokens are already present in the datasets and model, they will still be considered new tokens and will contribute to the formation of a new set.



Yes, even if the new tokens are already present in the datasets and model, they can still be considered as part of a new set (`new_set``) if they are introduced in a new commit. Each commit represents a snapshot of the system at a specific point in time, and the tokens introduced in that commit can be tracked separately, even if they overlap with tokens from previous commits.

Updated Implementation

Let's update your code to handle the concept of new sets (`new_set``) for each commit, ensuring that tokens introduced in each commit are tracked separately.

1. **Generate a Timestamp-Based ID for Each Commit**: This will uniquely identify each commit.
2. **Track New Tokens for Each Commit**: Maintain a set of new tokens for each commit.
3. **Integrate New Tokens into the System**: Use the new tokens in the system while ensuring they are tracked separately.

Updated Code

Explanation

1. **Generate a Timestamp-Based ID for Each Commit**:
 - A unique timestamp-based ID is generated for each commit using `Dates.format(now(), "yyyy-mm-ddTHH:MM:SS")``.
2. **Track New Tokens for Each Commit**:
 - A set (`new_tokens_set``) is initialized to track new tokens for each commit.
 - Tokens are extracted from the data and added to the `new_tokens_set``.
3. **Integrate New Tokens into the System**:
 - The new tokens are processed and optionally used to fine-tune the model.
 - The model is fine-tuned with the new tokens and saved with a name that includes the timestamp-based ID.

By following these steps, you can manage multiple commits, each producing new datasets and optionally new models, while tracking and integrating new tokens separately for each commit. This approach ensures that the system can evolve over time, with each commit reflecting a unique state of the system.



Does this imply that, over time, all the tokens from the original GPT-2 model could be replaced with new ones?



Yes, it is possible that over time, all tokens from the original GPT-2 model could be replaced with new ones if you continuously add new tokens and deprecate old ones. However, this depends on how you manage the lifecycle of tokens and models in your system.

To ensure a smooth transition and maintain the integrity of the model, you should carefully manage the addition of new tokens and the deprecation of old tokens.

Key Points

1. **Track New Tokens**: Keep track of new tokens introduced in each commit.
2. **Deprecate Old Tokens**: Gradually deprecate old tokens based on your criteria (e.g., age, usage frequency).
3. **Fine-Tune the Model**: Continuously fine-tune the model with new tokens to adapt it to the evolving vocabulary.

Updated Implementation

Let's update your code to include the tracking of new tokens and the deprecation of old tokens.

Step 1: Track New Tokens

Modify the code to track new tokens introduced in each commit.

Explanation

1. **Track New Tokens**:
 - A global set (`global_new_tokens_set`) is initialized to track all new tokens introduced in each commit.
 - For each commit, a local set (`new_tokens_set`) is used to track new tokens for that specific commit.
 - New tokens from each commit are added to the global set.

2. ****Deprecate Old Tokens****:

- You can implement a mechanism to gradually deprecate old tokens based on your criteria (e.g., age, usage frequency). This part is not explicitly shown in the code but can be added as needed.

3. ****Fine-Tune the Model****:

- The model is fine-tuned with the new tokens introduced in each commit.
- The fine-tuned model is saved with a name that includes the timestamp-based ID to differentiate between models from different commits.

By following these steps, you can manage the lifecycle of tokens and models, ensuring that new tokens are integrated into the system while old tokens can be deprecated over time. This approach allows the system to evolve and adapt to new data while maintaining a clear history of changes.



Currently, I am working with data related to Enron emails, processing it on a daily basis. Each day's collected data contributes new tokens for fine-tuning the GPT-2 model. As one of the use cases for system evolution that we discussed, we can generate potential response emails to be processed in the coming days. How can we implement this scenario?



To implement the scenario where you process Enron emails day-by-day, generate new tokens, fine-tune GPT-2, and use the fine-tuned model to generate possible response emails, you can follow these steps:

1. ****Process Emails Day-by-Day****: Extract and process emails for each day.
2. ****Generate New Tokens****: Extract new tokens from the emails.
3. ****Fine-Tune GPT-2****: Fine-tune GPT-2 with the new tokens.
4. ****Generate Possible Response Emails****: Use the fine-tuned model to generate possible response emails.

Step-by-Step Implementation

Step 1: Process Emails Day-by-Day

Modify your existing code to process emails day-by-day and generate new tokens.

Explanation

1. ****Process Emails Day-by-Day****:

- The main loop processes emails for each day, extracting and processing the data.
2. ****Generate New Tokens****:
 - Tokens are extracted from the emails and added to a set of new tokens for each commit.
 3. ****Fine-Tune GPT-2****:
 - The GPT-2 model is fine-tuned with the new tokens introduced in each commit.
 - The fine-tuned model is saved with a name that includes the timestamp-based ID.
 4. ****Generate Possible Response Emails****:
 - The `generate_response_emails` function uses the fine-tuned model to generate possible response emails based on the processed emails for the day.

By following these steps, you can implement a system that processes Enron emails day-by-day, generates new tokens, fine-tunes GPT-2, and generates possible response emails. This approach allows the system to evolve over time, adapting to new data and generating relevant responses.