ARI3333: Generative AI Practical Project

Developing a Content Generation System for Creative Storytelling

Gianluca Aquilina
ID Card Number: 348904L

Github Repository where the code is located:
https://github.com/GianUOM/ARI3333AssignmentCode

Contents

1 Introduction

The advancement of Generative AI has allowed us to create more creative AI-driven story writing. This project will explore the design and implementation of an AI-powered story generation system, which allows the users to create unique narratives customizable through an user interface.

The main objective of this task was to create a storytelling system using an AI model that assures coherence and is well-structured while allowing users to define their story parameters, such as genre, tone, characters, and setting, to provide more engaging and diverse stories. Moreover, it considers ethical concerns related to AI bias and improper content filtering to improve usability for all audiences.

The report will detail the system's architecture, the code implementation, and the functionality while mentioning the challenges encountered and solutions used. As a quick overview, this system is built using Streamlit for the user interface, with the generation of the text powered by LLaMA 3.2-3B-Instruct using the Hugging Face API. To improve the efficiency of storytelling, the system combines user input processing, text refinement features, and export functionalities. Throughout this implementation, the project will demonstrate the potential of AI in creative writing while noting some future improvements that can make the system better.

2 Design Choice for the Text Generation Module

When it came to picking the model to use for text generation, I decided to use the LLaMA 3.2-3B-Instruct, which was accessed via the Hugging Face API. The reason why this model was chosen was that a free Inference API can be used to get the whole model rather than dowloading the model locally or paying for use of API. For instance, the OpenAI models were tested, but, like GPT-2, even with prompt engineering, the output was not as good as expected, and for more advanced models (GPT-3 and 4), payment of API credits was required to start using it. Overall, the LLaMA model is an excellent choice, as with a few tweaks of prompt engineering, it is capable of generating some high-quality and relevent text based on the user preferences.

To ensure that the narrative is coherent, I constructed some more detailed prompts that tell the model to generate the text following a clear beginning, middle, and end. In order to shape our outputs using user-defined story parameters of genre, tone, character, setting, and word range, the concept of prompt engineering played a crucial role in this aspect.

Additionally, this system allows for refinement of the generated stories, which enables the user to change the input parameters and regenerate sections. This makes the user experience more satisfying as it gives it more control of the story he/she wants to read.

3 System Architecture and Code Explanation

In this chapter, I will discuss the system components used and integrated to provide a smooth user interaction. In addition, I will also discuss in detail the technical workflow of the system through some coding explanations.

3.1 System Components

1. User Interface: I went for the Streamlit library, as it is simple to code and and it provides a neat dashboard for users to input story parameters and change or modify the generated text
2. Text Generation Model: I used LLaMA 3.2-3B-Instruct model through the communication with the Inference API of Hugging Face
3. Refinement Section: Allows the user to, after viewing the generated text, regenerate or modify the story by adjusting the parameters, such as a tone and character change or any other custom changes
4. Export PDF: Allows the user to convert the generated text into a PDF file to save the story, in case the story is lost

3.2 Code Explanation

3.2.1 User Input and Initializing Session State Variables

Firstly, the session state variables are set to store the current story and the selected story parameters by the user which will be stored in the *st.session_state.story_params*.

```
if "current_story" not in st.session_state:
    st.session_state["current_story"] = ""

if "story_params" not in st.session_state:
    st.session_state.story_params = {
        "genre": "",
        "tone": "",
        "word_limit": "",
        "character": "",
        "setting": ""
    }
```

Code 1: the story parameters stored inputted by the user

3.2.2 Generating the Initial Story

Once the user clicks *Generate Story* button, the system fixes the prompts from the *create_prompt* method that specify the genre, tone, setting, word range, and character(s), along with more instructions.

```python
def create_prompt(genre, tone, character, setting, word_limit):
    word_limits = {
        "Really short (150 - 300 words)": (150, 300),
        "Short (400 - 600 words)": (400, 600),
        "Medium (700 - 900 words)": (700, 900),
        "Long (1000 - 1200 words)": (1000, 1200),
        "Very long (1300 - 1500 words)": (1300, 1500)
    }
    min_words, max_words = word_limits[word_limit]
    return (
        f"Write a complete {genre} story with a {tone} tone about {character} in {setting}. "
        f"The story must be between {min_words} and {max_words} words. "
        "Include a clear beginning, middle, and end with proper character development and plot progression. "
        "Write the story directly without any explanations or meta-commentary. "
    )
```

Code 2: the construction of prompt using user-defined story parameters

Afterwards, the *generate_story()* method creates an HTTP POST request to the Hugging Face API and passes parameters such as *max_new_tokens*, *temperature*, and *top_p* to create more random outputs. It uses the **try-except** blocks to handle API errors and some unexpected response formats and, using the Streamlit features of spinner and block, will provide real-time feedback.

```python
def generate_story(api_key, prompt, max_tokens):
    try:
        API_URL = "https://api-inference.huggingface.co/models/meta-llama/Llama-3.2-3B-Instruct"
        headers = {
            "Authorization": f"Bearer {api_key}",
            "Content-Type": "application/json",
            "x-wait-for-model": "true"
        }
        data = {
            "inputs": prompt,
            "parameters": {
                "max_new_tokens": max_tokens,
                "temperature": 0.8,
                "top_p": 0.9,
                "do_sample": True
            }
        }

        with st.spinner("Generating story..."):
            st.info("Sending␣request␣to␣API...")
            response = requests.post(API_URL, headers=headers, json=data)

            if response.status_code != 200:
                st.error(f"API␣Error:␣Status␣Code␣{response.status_code}")
                st.error(f"Error␣Details:␣{response.text}")
                return None

            response_data = response.json()
            st.info("Received␣response␣from␣API")

            if isinstance(response_data, list) and len(response_data) > 0:
                generated_text = response_data[0].get("generated_text", "")
                if not generated_text:
                    st.error("No␣text␣was␣generated")
                    return None
                cleaned_text = clean_generated_text(generated_text, prompt)
                if not cleaned_text:
                    st.error("Text␣was␣cleaned␣but␣resulted␣in␣empty␣content")
                    return None
                return cleaned_text
            else:
                st.error(f"Unexpected␣API␣response␣format:␣{response_data}")
                return None
    except requests.exceptions.RequestException as e:
        st.error(f"API␣Request␣Error:␣{str(e)}")
        return None
    except Exception as e:
        st.error(f"Error␣generating␣story:␣{str(e)}")
        st.error(f"Error␣type:␣{type(e)}")
        return None
```

Code 3: the generate_story() method

Once the text is generated, the text-cleaning method, *clean_generated_text*, will remove any unwated AI-generated explanations by filtering specific substrings irrelevant from the story before outputting the generated story.

```
def clean_generated_text(response_text, prompt):
    if response_text.startswith(prompt):
        response_text = response_text[len(prompt):].strip()

    patterns_to_remove = [
        "In␣this␣rewritten␣version",
```

```python
        "The␣emotional␣tone␣of␣the␣rewritten␣story",
        "While␣keeping␣the␣main␣plot␣and␣setting",
        "Here's␣the␣story␣with␣a",
        "I'd␣like␣to␣rewrite␣this␣story␣with␣a",
        "Here's␣your␣story␣with␣a",
        "The␣changes␣made␣include:",
        "Overall,␣the␣rewritten␣story",
        "**Act␣I:",
        "**Act␣II:",
        "**Act␣III:",
        "**Act␣IV:",
        "---",
        "**Characters:**",
        "**Setting:**"
    ]

    end_positions = []
    for pattern in patterns_to_remove:
        pos = response_text.find(pattern)
        if pos != -1:
            end_positions.append(pos)

    if end_positions:
        response_text = response_text[:min(end_positions)].strip()

    if "**" in response_text:
        response_text = response_text.split("**")[0].strip()

    response_text = response_text.replace("**", "")

    if "tone:" in response_text.lower():
        response_text = response_text.split("tone:")[-1].strip()

    return response_text.strip()
```

Code 4: the clean_generatde_text() method

3.2.3 Refinement Original Text

The user has the option to refine and modify the story by selecting a refinement option from a menu of either changing the tone or changing the characters or applying a custom change by allowing the user to specify the instruction, and then the system constructs a prompt to keep all core elements while adjusting the story.

Each refinement option follows this process:

1. The system regenerates the story with the updated parameters

2. The new version is temporarily stored in the *st.session_state['temp_redifined_story]* variable

3. The user will then choose to either keep the version by overwriting the *st.session_state["current_story"]* variable or keep the original version

```
if refine_option == "Change Tone":
        new_tone = st.selectbox(
            "Select new tone:",
            ["Adventurous", "Emotional", "Humorous", "Dark", "Mysterious", "Romantic", "Philosophical"],
            key="tone_select"
        )

        if st.button("Apply Tone Change", key="apply_tone"):
            try:
                current_word_limit = st.session_state.story_params['word_limit']
                min_words, max_words = word_limits[current_word_limit]
                max_tokens = max_words * 2

                refine_prompt = create_prompt(
                    st.session_state.story_params['genre'],
                    new_tone,
                    st.session_state.story_params['character'],
                    st.session_state.story_params['setting'],
                    current_word_limit
                )

                with st.spinner("Refining story..."):
                    refined_story = generate_story(API_KEY, refine_prompt, max_tokens)

                    if refined_story:

                        st.session_state['temp_refined_story'] = refined_story


                        st.markdown("### Refined Story:")
                        st.text_area(
                            "Preview",
                            st.session_state['temp_refined_story'],
                            height=300,
                            key=f"refined_story_preview_{int(time.time())}"
                        )


                        col1, col2 = st.columns([1, 4])
                        with col1:
                            if st.button("Keep This Version", key=f"keep_version_{int(time.time())}"):
                                if handle_keep_version(st.session_state['temp_refined_story']):

                                    st.session_state.story_params['tone'] = new_tone
                                    st.success("New version saved!")
                                    time.sleep(0.5)
                                    st.rerun()
                    else:
                        st.error("Failed to generate refined story. Please try again.")

            except Exception as e:
                st.error(f"Error during tone refinement: {str(e)}")
                st.error(traceback.format_exc())
```

Code 5: if the refine option is to change tone (very similar implementations with the 2 other refinement options)

3.2.4 Story to PDF Export

Once happy with the story, the users can export the story via the *export_to_pdf()* method by creating a formatted PDF with the genre and tone as headers. Using the *st.download_button()* method, it allows the user to download the PDF in their machine

```
def export_to_pdf(story, genre, tone):
    try:
        pdf = FPDF()
        pdf.add_page()

        pdf.set_left_margin(20)
        pdf.set_right_margin(20)

        pdf.set_font("Arial", 'B', 16)
        pdf.cell(0, 10, "Generated Story", ln=True, align='C')
        pdf.ln(10)

        pdf.set_font("Arial", 'B', 12)
        pdf.cell(0, 10, f"Genre:␣{genre}", ln=True)
        pdf.cell(0, 10, f"Tone:␣{tone}", ln=True)
        pdf.ln(5)

        pdf.set_font("Arial", size=12)
        paragraphs = story.split('\n')
        for paragraph in paragraphs:
            if paragraph.strip():
                pdf.multi_cell(0, 10, paragraph.encode('
                    latin-1', 'replace').decode('latin-1'))
                pdf.ln(5)

        file_name = f"story_{datetime.now().strftime('%Y%m%
            d_%H%M%S')}.pdf"
        pdf.output(file_name)
        return file_name
    except Exception as e:
        st.error(f"Failed␣to␣export␣story␣to␣PDF:␣{str(e)}")
        return None
```

Code 6: Exporting the story to PDF

4 Challenges Faced and Solutions

1. In the early stages of using the model, content was being produced that really lacked the logical flow of the story. To address this, prompt structuring was crucial to enforcing a beginning, middle, and end format.

2. I had some API request failures and latency but were managed by utilizing exception handling to make sure that the user gets the precise error messages and retry attempts

3. There was some bias in the early stages of the development but was mitigated through the structured prompt engineering and content filtering before displaying the final output

4. The 'Keep New Version' button unfortunately didn't work properly, as when the button was being pressed, the text wasn't being updated with the new version. This was attempted to be solved by making the state variable of the current story set to the new refined text, but this wasn't successful

5 Ethical Considerations and Mitigation Strategies

5.1 Content Moderation and User Safety

I made sure that inappropriate content in the character and setting is not allowed to have a generated story. Moreover, The system provides the chance for users to refine their text and ajust any content in the refined text

5.2 Addressing Bias in AI-Generated Content

Like mentioned in the Challenges Section, bias was solved by structuring the prompt that encourages inclusive storytelling. the refinement process can also allow the users to edit content that may accidentally be stereotypical

5.3 Transparency and User Control

Users can iteratively refine generated stories and can manually review output before exporting. This ability gives the user control and can be flexible with the storytelling.

6 Future Improvements

Some future improvements I would add to this system would be to maybe fine-tune the model to have more emotional depth in the story. Moreover, I could have also integrated video generation, as it would have been a cool feature to have apart from the text generation module. Lastly, an improvement would have been to highlight a part of the story and ask the AI to edit that part specifically to increase usability for the user.