Google Hackathon Report
Team 10: Prompt to Slides
November 4, 2023
Aniruddh Ingle, Sylvanne Braganza, Sherry Li, Hritik Bhansali, Cece Zhang

Project Overview

Our project aimed to transform text-based content into engaging presentations with the help of Google's powerful PaLM2 API. By harnessing the capabilities of this cutting-edge natural language processing tool, we streamlined the process of creating dynamic presentations from textual information. Whether it was converting lengthy reports into informative slides or transforming written content into compelling spoken presentations, our team enhanced the efficiency and effectiveness of communication. With the PaLM2 API, we automated the structuring and summarizing of information, allowing users to effortlessly transform their ideas and data into engaging spoken presentations. This project is a game-changer in simplifying the presentation creation process, making it an invaluable tool for professionals, educators, and anyone looking to communicate their ideas effectively.

MakerSuite for Prompt Engineering

In our project, we leveraged Google's MakerSuite for prompt engineering. Initially, we explored different prompt types, including text prompts, data prompts, and chat prompts. After experimenting with these options, we found that the data prompt was the most suitable choice for our project aims due to its ability to tailor the input and output structure according to our project's specific requirements (text to Markdown-formatted script).

Data prompts work by taking custom instructions provided by users. In our case, we asked it to generate a presentation in Markdown script format tailored to a specific age group. Then we give the model specific input and output examples to work with. These prompts give the model examples to replicate. Google calls these "few-shot prompts." They served as training data for the model. Over time, we fine-tuned these examples based on the specific content structure and style we wanted for our presentations. This iterative process allowed the model to learn and adapt to our requirements. We proceeded step by step.

Our initial goal was to use this data prompt to efficiently break blocks of text into bullet points, simplifying the content structuring process. Once we achieved this, we took it a step further by instructing the data prompt to format the extracted bullet points as a Markdown script for a two-slide presentation, consisting of one title slide and one content slide. Finally, we tailored our examples to generate multi-slide presentation Markdown outputs, providing even greater flexibility and automation in the presentation creation process.

Our input texts were initially taken from Wikipedia articles; our outputs were initially written based on a template HTML Markdown code (Appendix A) that we manually filled in. In MakerSuite, we were then able to have the model use our examples (Appendix B) to generate more examples.

We also experimented with the underlying LLM at this stage. LLM responses exhibit both deterministic and random characteristics. When you input a prompt to an LLM, it generates a probability distribution over potential tokens (words) likely to appear next. This initial stage is entirely deterministic; the LLM consistently produces the same distribution when given the same prompt. However, in the subsequent stage, the LLM transforms these distributions into actual text responses using various decoding methods. A straightforward decoding approach might select the most probable token at each step, ensuring determinism. Alternatively, you can opt for a response generation method that involves random sampling from the model's output distribution, introducing a degree of randomness. This stochastic behavior can be controlled by adjusting the temperature setting. A temperature of 0 results in deterministic token selection, while higher temperatures introduce more randomness, yielding unexpected and surprising model responses. (LLM Concepts Guide, 2023)

In our prompt generation stage, we changed this temperature setting to achieve a broad range of example prompts that still formatted the output as our desired Markdown code script. MakerSuite then allowed us to add the newly generated examples that we approved as more examples to train the model.

Once we were happy with the model's responses to our test inputs, we easily exported it to Python code (Appendix C) in Google Colab, and called the same model using the PaLM API.

Document Search with Embeddings

This technique allows for searching documents by comparing their embedded representations, or vectors of numbers that capture the semantic meaning of text.

The steps are as follows. First, we must generate embedded representations for all of the text in the document(s). This can be done using a pre-trained embedding model, a few of which are provided by Google. Next, we must generate an embedded representation of the query text. This is done using the same embedding model as used in the first step. Now, we can compare the embedded representation of the query text to the embedded representations of the input document(s). This is done using similarity metrics, such as distance determinations. Finally, we must return the results with the highest scores.

We based our code on the example provided by Google here:
https://developers.generativeai.google/examples/doc_search_emb.

This code allowed us to use the PaLM API to create embeddings so that we could perform document search. In our code, we can generate embeddings, build an embeddings database, and then document search with a Q&A system. The embedded representations are compared using the dot product of the vectors, ranging from -1 to 1.

One challenge encountered in our project was that the embedding system we used had a limitation where it only worked with a maximum of 10,000 bytes of data. Consequently, this meant that we could not include more than approximately 5,000 words in the input document. To address this limitation, we implemented a line in our code which processed text stored in the variable cleaned_text. Initially, it encodes this text into bytes using the UTF-8 encoding, which is a common practice to ensure that text can be represented as a sequence of bytes. Following the encoding step, the code then extracts the first 9900 bytes from the encoded byte sequence using slicing. This ensured that our text remains within the acceptable size range. Finally, the sliced byte sequence is decoded back into a text string using the UTF-8 encoding, with the 'ignore' option specified for error handling. If any decoding issues arise due to text truncation, this option instructs Python to skip those problematic characters and continue decoding the rest of the text, thus preserving as much of the original content as possible.

We then incorporated our engineered prompts into our document search with embeddings code.

There are several advantages to using document search with embedded text over keyword-based, traditional search methods for our project. The main reason is that it is more robust to synonyms, misspellings, which might crop up in our users' queries or in the conversion from the input PDF document(s) to string(s). Another advantage is that it can be used to search for documents in a variety of languages, since it does just depend on semantic meanings.

Google API

Our final step was connecting our input and output PDF files to our Google Drive account for ease of use and seamless access.

We were able to do this using Google APIs Explorer. The Google Drive API allows our model to access resources from Google Drive.

How to Use

We have included a ReadMe file in our submission.

Briefly, a user can upload a PDF document into a folder in their Google Drive. They can then adjust the query to specify age or education level. Finally, they run our python script to generate a PDF presentation of slides, which will be saved back into their Google Drive.

Future Directions

We plan to continue to build on this project for the remainder of the mini.

As previously discussed, we are presently utilizing the PaLM API to generate embeddings that enable us to perform searches within PDF documents stored in a Google Drive using our queries. So far, our testing has predominantly focused on prompts in the English language. However, as document search with embedded text is adaptable and capable of searching in various languages, we intend to extend our testing to encompass languages beyond English, including Chinese and Hindi, to build on this project.

Furthermore, we intend to broaden our scope by exploring additional types and lengths of documents that can be searched using our queries. Currently, our testing is limited to searching through PDF documents. However, we are planning to expand our search capabilities by including Word documents and websites in our testing, aiming to enhance the versatility and adaptability of our query.