Reportix

A novel AI-powered report assistant

Candidate 27 and Candidate 21

Written using Reportix - AI Report Assistant

Spring 2024

Introduction

Writing reports can be time consuming, experience the ease and efficiency of Reportix!

In this project we have implemented an AI powered report writing assistant named Reportix. The implementation is a WebApp, with a Python backend, designed to run entirely on-device even on laptops.

Reportix is an AI-powered report writing assistant designed to help students streamline their reporting process. By automating the creation of high-quality reports, Reportix enables students to focus on the content and insights of their reports, rather than getting bogged down in the writing process. With its intuitive interface and powerful algorithms, Reportix can help students save time and produce reports that are both accurate and engaging. In this project, we will explore how Reportix can help students streamline their reporting process and produce high-quality reports. By leveraging Ollama's model management capabilities, Reportix provides users with a seamless experience and access to a wide range of models - all without requiring any additional storage space. And by allowing multiple users to connect to a single running backend, Reportix also offers the potential for collaborative reporting and knowledge-sharing.

Model management is handled thru Ollama to provide a smoother user experience and to consolidate models with other applications to conserve storage space, therefore Reportix requires no additional space for models outside what Ollama uses. This feature also allows the user to select between the many models available in Ollama.

Collaboration features have been explored and is working in basic regard by having multiple users connect to a single running backend. This could relatively easily be expanded to a full feature, even though it is not the goal of this current project.

In our search we have found this use case to be unique, but with the recent explosion of AI-powered tools we can't disregard the chance that a similar application exists. However, in this report we hope to clearly explain our approach, possible improvements, and conclude with our key findings from the project.

Methods

This section will explain the process of how Reportix work. Here we explain how the models work, and what they are used for in the web interface

Models

Supervised Fine-Tuning (SFT) was utilized to enhance the model's performance in creating accurate and consistent project structures. This process involves preprompting the models for certain tasks. We are using a model for each of the following tasks: Code summarization and understanding, suggestions for report, file ranking. These models are necessary for reading the project folder and files, understand these, summarize and suggest necessary parts to note in the sections. In the webapp the user is supposed to provide the site with the following: Title, Subtitle, Author, Date and Project Directory involving all of the code used in the project. Besides that the webapp includes two dropdown menus which change model type and latex-template style. Meaning, the user can choose a model he/she wishes to use for suggestions, and a pre-made templates for easy startup.

File Ranking

File ranking is done for compression of the project incase it is extremely large, and consists of many files. We have not caped the size of the project that can be uploaded. If the project consists of lets say 10000 file, it will take a long time to upload, but it is possible. This is where the File Ranking comes into play. After we read the project structure using some directory traversing, we create a string value which holds the whole structure of the project and pass this string as input into the File Ranking model. This model simply decides which files are necessary to keep and which files are not as important for the project analysis. This is based on files that are readable and end with for example .py or .js, etc. . This then effectively compress the size of the project, and creates a new structure with only the most important files deemed by the model.

Code Summarization

After the File Ranking model has created a compressed project structure and chosen the necessary files, we throw this new structure into the Code Summarization model. Code Summarization then goes into each and every file and reads the contents of the files. This is then used to create a summary the project based on the contents of the project in hand. When this is done, we have everything we need to generate solid suggestions for the user writing the report.

Report Suggestions

Report Suggestions generate suggestions based on the summary of the code, and the text already written by the user in a certain section, based on the IMRaD report structure. The title and subtitle is automatically given to the model as part of its preprompt. We explored the ability to provide the model with other sections as input in addition to the current section, but found no noticeable improvement in the suggestions, and at least not enough to warrant the use of precious context window space.

Preprompts

Prepromting of the models is crucial mainly because that is where the models task designed. In the prepromt text of each and every model, model parameters are defined as follow: PARAMETER temperature 0.5 , PARAMETER num_ctx 8192. The parameter num_ctx reffers to context size of the model. The value 8192 is maximum tokens which llama-3 model manages to intake. We use the maximum value mainly because we wish to speed up the process of reading the files. Since a model takes some time to create an output, we wish to make as few calls to the model as possible.
Temperature in a model reffers to parameter which defines the language models output, where the higher value for temperature generates responses with more randomness and creativity, while lower values make the model more predictable and consistent in its outputs. The temperature used for File Ranking is 0.5, where llama 3 default value is 0.8. This 0.5 value is used mainly because we wish to have a consistent folder structure, while for the rest of the models we use a higher temperature of 0.7. This temperature is higher because we wish more creativity when creating text summaries.

Tokenisation problem

Tokenization/Chunking problem. When reading file contents a problem occurs. This problem is based on context size. When reading a file the idea is to add the file contents to the chunk which will be processed by model, and check if the chunk size is not exceeding the limit of models context window. Here the problem becomes the fact that length of the read file is the number of characters in it. This can be adjusted to read words only by splitting the string into words based on whitespaces between these. Then the last problem occurs because of tokenization happening in a certain way. The tokenized text by the model will have more string values than the file split by whitespace. Example tokenization: ["Hust", "on", ",", "we", "have", "a", "prob", "lem"], Example simply whitespace split: ["Huston,", "we", "have", "a", "problem"]. This effectivelly shows the missmatch in the lengths of the lists. Solution: Tokenize the words when reading the files using same tokenizer as the model to match these chunks.

LaTeX templates

Reportix uses a simple templating system for LaTeX code where the user is given the option to select a template from the directory "latex-templates". Then Reportix will use markers of the style "$Reportix : Title$" for placing the details and sections of the document. This seems to be a good midpoint between making it extendable in the future, as well as making it very easy to add more templates to the application. Writing a new template requires no change in the code for the application and can be done using pure LaTeX code.

Results

Reportix works well with a simple installation process consisting of two prerequisites (LaTeX and Ollama) and a pair of commands to run the user interface and the

backend.

Issues and Bugs

In our testing we have not found any major issues or bugs. The web interface works well across computers, mobile devices may require adjustments due to their smaller screen size. Minor issues such as issues with some special characters have been found and should be easily fixable if they are found to impact the user experience, but we have not yet found that to be the case.

Using different models

We have tested the application with multiple models: Llama3, Llama2, Phi, Phi3, Llava. It seems all models that do well with understanding text does a reasonable job aiding in writing reports, though our best results have been using Llama3. It is difficult to quantify the quality of suggestions as written text, but we've attempted to use it heavily and based our analysis on this use.

Discussion

Future Work

There are many features that could be added to extend and improve the functionality of Reportix. Additionally the user interface and experience could definitely be refined.

The current editor assumes the user is always intending to write in the IMRaD style, but the backend has been coded to be available for more general styling. With some work the user interface could be extended to allow the user to select between report styles, and even customise them. This could be implemented as another document setting with a selection between report styles, and further with a separate menu to create, modify, and delete report styles from the users library.

Editing support for features such as text styling, subsections, math, and the like is available thru the input being LaTeX code; however it would be an improvement to have a visual editor that supports these features directly in the interface.

The current interface allows the user to view and download their document as a PDF, extending this support to include more file types, as well as downloading the raw LaTeX code would be a useful feature.

References should be a part of the user interface as they currently have to be added manually. The AI shouldn't necessarily suggest references, but it could definitely be expanded to search the internet for similar documents and provide the user with an extended context for their report. A simple user interface to manage references and an automatically inserted list at the end of the document would be a useful and good start to such an implementation.

Bundling a LaTeX compiler, as opposed to the current approach relying on the compiler installed on the system, could improve the user experience. Especially if the service is to be packaged into a single application binary this would make the installation procedure one-click. Ollama would still be necessary, but in this case the web interface could be expanded with options to install Ollama, pull models into Ollama, and manage local models. his has not been done as it would specialise the application and re-

quire significant time.

Alternatively the application could be served from a central server, which would probably not want to include model management and LaTeX compiler bundling as features, but rather expand in the direction of adding user accounts, improving collaboration, and managing multiple projects stored in a cloud.

Completion is not currently in the AI models vocabulary, meaning they will keep suggesting improvements forever, this may be desired behaviour, but if not a system could be implemented to hav the AI models score the reports for doneness and when a certain threshold is reached stop providing suggestions.

Model Bias

As with any AI project there is a chance that the model outputs biased responses based on its training set. Reportix allows the user to easily rerun the suggestion generation with other models, which should help the user in accounting for these biases if they can identify them. Unidentified biases remain a risk, with no clear way to avoid them. Model bias in text output is particularly hard to detect as the only automated way of analysing text for bias is AI models.

Any effects of model bias in our application should be reasonably limited in their potential to cause problems as the suggestions are only given to the user writing the report and not automatically inserted, this way the user will proof-read all content of the exported report. It would also be possible to use models tuned to detecting bias to further ensure and potentially make the user aware of both the models and their own biases.

Conclusion

Our initial results with Reportix show promise for AI-powered report assistance tools. There are many existing tools for more general writing assistance, but we've found that directing the AI more gives more relevant suggestions. Additionally including this in a purpose-built interface simplifies the process and avoids the user having to constantly ask the AI models manually.

We've found Reportix to provide decent suggestions that improve the clarity and accuracy of our reports, as well as aiding us when we encounter a writers block.

We believe our key finding is that with current models an approach such as Reportix can yield a valuable product that improves clarity and accuracy of reports, while reducing the time investment needed to produce them and simplifies the process to the user.

Finally it's worth mentioning that this report has been written in Reportix, and it's suggestions has shaped the report.

References

This project would be impossible, or significantly more complex, without the many great projects we rely on for building the user interface, enabling communication, running an API, using AI models.

1. Ollama [https://ollama.com]
2. FastAPI [https://fastapi.tiangolo.com]
3. TailwindCSS [https://tailwindcss.com]

4. daisyUI [https://daisyUI.com]
5. LLaMA: Open and Efficient Foundation Language Models [https://arxiv.org/abs/2302.13971]
6. [https://github.com/meta-llama/llama3/blob/main/llama/tokenizer.py]

All the code for this project is available for download on GitHub in this repository: https://github.com/victorzimmer/Reportix