Acta Scientific COMPUTER SCIENCES

Volume 5 Issue 5 May 2023

Research Article

Exploring Prompt Engineering for Generative AI-Based App Generation

Jasmine L Shone[1], Robin Liu[2], Evan Patton[2], David YJ Kim[2]*

[1]*Hawken Upper School, USA*

[2]*Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, USA*

***Corresponding Author:** David YJ Kim, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, USA.

Received: March 21, 2023

Published: April 25, 2023

Abstract

We introduce a cutting-edge learning platform powered by large language models that enables students to effortlessly generate mobile applications for smartphones and tablets from natural language descriptions. We further demonstrate that these user-generated apps can be further optimized with minor adjustments to the generative model's input, or, its "prompt." To maximize the efficacy of the prompt in producing a desired application, we explore three different methods of modification: 1) altering the selection mechanism of example pairs, 2) varying the number of example pairs, and 3) changing the order of pairs within the prompt. The prompts are constructed from a collection of example pairs, which comprise a textual description of an example app and its corresponding code, in addition to a description of the desired app. We test the model's performance by evaluating it with 18 different mobile application task descriptions, ranging from basic to complex, and then leveraging BLEU score to compare the model's outputs to manually created apps. Our findings indicate that the method of determining example pair selection and varying the number of examples included can significantly influence the quality of the generated apps. However, reordering the placement of the example pairs within the prompt does not affect the outcome. Finally, we conclude with a discussion on the potential implications for computer science education. The platform we present in this paper aims to further the democratization of app creation through enabling users to create apps with ease, regardless of their technical background.

Keywords: Large Language Models; Code Generation; Prompt Engineering; Mobile Applications; Computer Science Education

Abbreviations

LLM : Large Language Models

Introduction

We present a large language model-based learning platform that lets students automatically generate mobile applications for smartphones and tablets from natural language descriptions. The needs and benefits of education surrounding mobile application development has led many educators to design novel curriculums targeting that purpose [1]. Nonetheless, many students are discouraged from creating their own applications because the task of learning the necessary programming expertise appears daunting. As a result, there has been a continuous drive to tackle these barriers within the scientific and industrial computer science community to reduce the amount of ostensible coding needed through simplification mechanisms such as drag-and-drop functionality and block coding (Figure 1). Several learning platforms, such as Scratch [2], attempt to make the learning curve less steep and serve as a bedrock for students to begin their app development journeys. Our research aims to take this simplification of app creation one step further. Our platform requires no user interface learning so students can focus on generating and describing their unique ideas.

Large language models (LLMs), such as GPT-3, have demonstrated that they can perform a wide range of text-based tasks

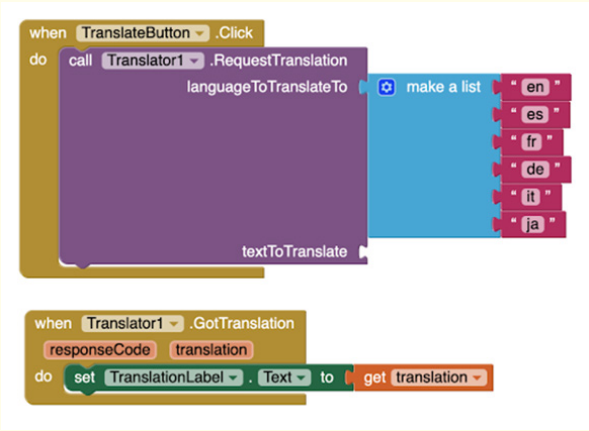

Figure 1: Example of a block code.

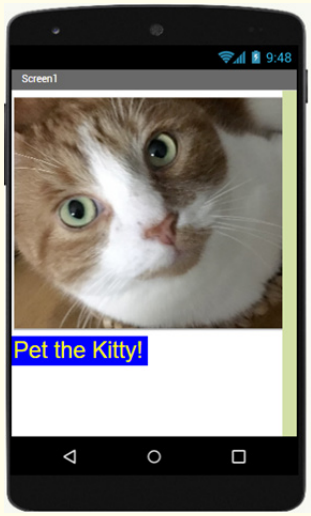

Figure 2: A fully functional application with a picture of a kitty that, when clicked, plays a meow sound.

through carefully crafted model input [3]. The input of LLMs is often referred to as prompts [4,5]. Several compelling GPT-3 demos demonstrate that prompts can be written to customize a single model to perform a wide range of tasks, such as creating an image out of textual instruction [6,7]. Prompt engineering is an emerging field that seeks to improve the text or code generation capabilities of a language model by modifying its prompts. This approach is centered on optimizing the performance of existing LLMs and has the potential to not only reduce the time, cost, and effort required to develop new LLMs or gather large datasets for fine-tuning, but also to enhance their effectiveness. One way of crafting a prompt is by providing a small number of examples of solved tasks as part of the input to the trained model, which is referred to as few-shot prompts. For instance, if we want the model to perform English to French translation, we provide a few translated examples before the desired sentence to be translated. The common interpretation of the few-shot prompt format is that the model is "learning" the task during runtime from few-shot examples. We used OpenAI Codex [8] as the large language model. Codex is a descendant of GPT-3; its training data contains both natural language and billions of lines of source code from publicly available sources.

In response to the emergence of Codex and other LLMs, there has been a recent interest in examining methods of generating prompts that remove the need for human supervision. Gao., *et al.* [9] focused on automating the selection of few-shot examples for a model using a brute-force search for label words and selecting demonstrations randomly and based on semantic similarity to the prompt. Shin., *et al.* [10] developed an automatic prompt creation method for models using the weights of a logistic classifier to select label tokens and use a gradient-based approach to determine ``trigger'' tokens concatenated at the end of an input. Zhang., *et al.* [11] constructed a transformer-based generative model for prompts given a user input, also targeting masked language problems particularly for factual probing.

Codex is a language model that has expertise in several programming languages, but it excels in Python. To make the most of its capabilities, we created a prompt that generates Aptly-Script [12], an intermediate Python-like language that can be converted into MIT App Inventor block codes [12]. However, we found that we can improve the platform's performance even further through prompt engineering techniques. For instance, we can use relevant example pairs that demonstrate the use of specific components to help the model complete user requests better. Furthermore, providing more example pairs can help the model learn more and improve app creation. We also found that the placement of an example within a prompt can affect the generated results, as studies on GPT-3 have shown that examples placed towards the end have a more significant impact on the output [13]. With all these possibili-

ties in mind we explore different prompt engineering techniques to improve the model's output for a given description of an application. Specifically, we focus on three characteristics of the prompt:

- **RQ1:** Does the quality of code generation differ based on how example pairs are chosen?
- **RQ2:** Does increasing the number of example pairs used in the prompt improve the quality of code generation?
- **RQ3:** Can we improve the quality of code generation by ordering the example pairs differently?

Materials and Methods

Define

Our goal is to determine the best prompt engineering methodology to generate the desired mobile app from natural language input through a systematic examination of prompt engineering techniques utilized in previous literature. In this section, we detail our process for creating effective prompts for user-generated mobile app development. First, a set of example pairs is automatically synthesized in three steps, which correspond with the three prompt characteristics we wish to investigate. We then combine these example pairs with a natural language description of the desired app to construct the prompt. By comparing the effect of distinct variants of each prompt characteristic on performance, we aim to give insight into which prompt engineering approaches are the most effective in generating high-quality user-prompt applications.

Design and development

Our target platform, MIT App Inventor, utilizes the "Blockly" block coding script for its applications. To convert between code on the MIT App Inventor platform and text-based code utilized in the LLM prompt, we designed an intermediate language(Aptly Script) functions and classes that have a one-to-one correspondence with components on the MIT App Inventor platform. For example, a Text-to-Speech component in the App Inventor platform would have the same callable methods in Aptly-Script. Once the block code is created, the MIT App Inventor platform can be used to create the desired application. For example, the user can request the following translation application:

"Create an app called HelloPurr with a picture of a kitty that, when clicked, plays a meow sound".

Then, a prompt will be automatically synthesized by first selecting examples according to the prompt characteristics being tested and then concatenating the selected example pairs and the description of the requested translator app. Each example pair represented in the prompt consists of a textual description of an application along with the corresponding Aptly-Script, such as the following $<< d_1, c_1 >><< d_2, c_2 >> ... << d_k, c_k >>$, where d_i is the description of application i, and c_i is the Aptly-Script of application i. At the end of the prompt, we concatenate the user query to the prompt. The synthesized prompt is sent as an input to the Davinci Code version 2 Codex model, and its hyperparameters are set as the following: temperature = 0.5, max tokens = 2000, best of = 10. The output of the model can be converted into a fully functional mobile application (Figure 3), which generates the Aptly-Script for the application. The generated Aptly-Script can then be converted into App Inventor blocks to generate a fully functional application (Figure 2).

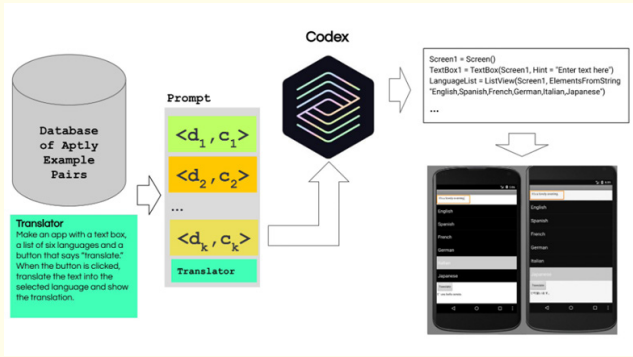

Figure 3: An overview of the whole process. When the user requests an application with its textual description, we automatically synthesize the prompt by adding several example pairs along with the desired application's textual description. This constructed prompt is fed into OpenAI Codex model as an input, which outputs code that can be converted into a fully functional mobile application.

For the set of example pairs, a database of 85 unique app examples was compiled by the team from apps created on the App Inventor platform. The app examples were selected to cover a wide range of the functionality within the App Inventor platform. These apps were converted from a block-coding-based expression to Aptly-Script. Each example pair is represented in the following order: its textual description, 'START' word before the start of the code, The Aptly-Script, and a 'STOP' word at the end of the code.

A sample example pair is the following:

"Create an app called HelloPurr with a picture of a kitty that, when clicked, plays a meow sound".

```
START
Screen1 = Screen(AppName="HelloPurr")
Cat = Button(Screen1, Image="kitty.png")
Meow = Sound(Screen1, Source="meow.mp3")
 when Cat.Click():
        call Meow.Play()
STOP
```

Testing

We validated the model with 18 descriptions of candidate mobile applications that served as plausible student app requests. These candidate application task descriptions included simple applications and more complex applications to investigate the performance of our prompt engineering methods on application tasks of varying difficulty. An example of a simple application description is, "Make a game that has a button in the middle of the screen. The button has a picture of a cookie on it. When the user clicks the button, increment the score by 1." This app can be assembled by first creating a button, making that button into a cookie, and incrementing the score by 1 when the cookie is pressed. Similarly, an example of a complex application task is, "Make a creature that the user can feed, wash, and cuddle with buttons. Each time the user performs an action, increase the creature's happiness by 20. If the user doesn't perform an action in 30 seconds, decrease the creature's happiness by 50 and make the creature say 'stop neglecting me!' " In comparison to the previous app, there are multiple buttons to create and each button has a different functionality. On top of that, a timer component is necessary to enable the app to say "stop neglecting me" after a certain time.

We created 36 manual solutions to the app tasks, two per problem. The solutions for the same problem are designed to be as different as possible from each other and cover different interpretations and implementations of the same app description. For example, if the description includes "say: 'Your order is ready!'", we may implement either a text-to-speech, or a text label that "say '' that 'Your order is ready!' When evaluating the output we compute the Bilingual Evaluation Understudy (BLEU) score [14] between the two reference solutions programmed by MIT App Inventor and the generated Codex output. BLEU score is a metric constrained between 0 and 1, with values closer to 1 representing higher similarity between the reference solutions and the machine-generated code. We chose to utilize BLEU score as a performance metric because of its efficiency (able to compute performance for thousands of code samples with minimal manual interference), interpretability, and its ability to evaluate code samples against multiple implementations of the same task description.

Results

Does the quality of code generation differ based on how example pairs are chosen?

To address the first research question, we examined the relationship between different example pair selection mechanisms and their performance in terms of BLEU score. More specifically, we fix the maximum upper bound length of the prompt to 1000 tokens and pick the example pairs in order. From that, we tested four different selection mechanisms we explain below.

Method I: Random selection

This method randomly selects a number of example pairs within the database. For each execution, it will select a different group of example pairs. This selection mechanism will serve as the baseline for our comparison.

Method II: Sort code by token length

Here we sort the example pairs in the database based on code length in ascending manner, then select example pairs starting from the least code length until it reaches the token cap. This option has the advantage of sending in the most example pairs for Codex to learn from. However, the selected example pairs may not reflect what is the most relevant to the requested description.

Method III: Select based on relevance

We rank the examples based on how semantically relevant they are to the user's requested application. We do so by generating embeddings for each app example and the user description. Embeddings are numerical representations of concepts converted to

number sequences. In our scenario, an embedding represents the semantic meaning of a natural language description or code [15]. Embeddings that are numerically similar are also semantically similar. For example, the embedding vector of a natural language description "Create an app called HelloPurr with a picture of a kitty that, when clicked, plays a meow sound." will be similar to the embedding vector of a code where an app shows an image of some animal and when clicked plays the sound the animal makes. To compare the similarity of two separate embeddings, you compute the cosine between the embedding vectors. The result is a "similarity score", sometimes called "cosine similarity", a score between –1 and 1, where a higher number indicates higher semantic similarity. Codex's Babbage engine was used to generate code embeddings for each example app in the database and the users' textual description.

Method IV: Revised maximum relevance minimum redundancy

We rank the examples using a revised version of Minimum Redundancy Maximum Relevance (mRMR) [16]. mRMR is currently used in machine learning [17] as a relatively efficient way to select a subset of features having the most correlation with a class (relevance) and the least correlation between themselves (redundancy). Simply speaking, the goal is to find the "minimal optimal" set of variables to predict the dependent variable. Relevance can be calculated by using the F-statistic (for continuous features) or mutual information (for discrete features) and redundancy can be calculated by using Pearson correlation coefficient (for continuous features) or mutual information (for discrete features). Since this information is unobtainable for few-shot example selection, we adapt mRMR by defining relevance as the cosine distance between the embedding of the candidate code and user text description, and defining redundancy as the cosine distance between the embedding of the candidate code and a code in the set of already selected example pair group. We use the following formula to select the next example pair j:

Where is the th code example, is the user requested app description, is the set of example pairs already selected, and is the mutual information (i.e., the cosine similarity) between and . Intuitively, this formula rewards example pairs that maximize the information between itself and the user description while it penalizes example pairs with a high mutual information with the already selected set of example pairs.

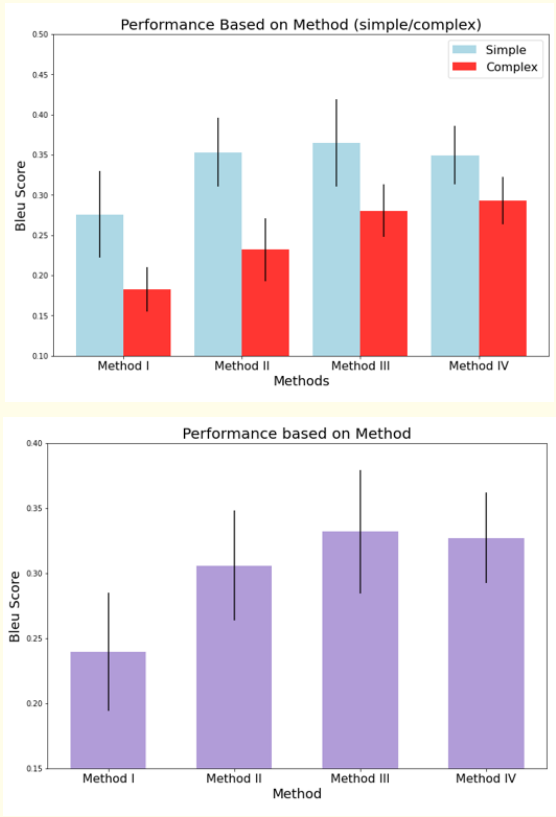

Figure 4: Results for different selection methods. The different methods are listed along the abscissa. Each bar indicates the mean Bleu score across test data; error bars reflect ±1 standard-error of the mean, corrected to remove variance due to the random factor [18].

Figure 4 shows the results of different selection methods. For each selection method, we ran the OpenAI Codex model five times for each test sample to address the randomness Codex generates. This results in five BLEU scores for each test sample, which we averaged, resulting in a single performance metric per test sample. Then, the mean and standard error across test samples is reported. We further our investigation by sorting a subset of the test samples into two groups of nine based on their complexity. Complexity was calculated by counting the number of compound statements (e.g., number of if, number of for). We discovered several interesting trends within our examination. We observe that overall, selecting the most relevant example pairs using embeddings or mRMR

turns out to have a marginal advantage. For complex apps, that trend seems to be more apparent as mRMR has the upper hand and improves the performance from random baseline by 55% (0.10 increase in BLEU score). For simple apps, relevance-based selection methods improve the performance by 25% (0.07 increase in BLEU score). However, a simple algorithm to select as many example pairs as possible is comparable to more advanced methods for simple apps. These results indicate that while the model can learn to create simple applications by merely providing an abundant number of example pairs, the relevance of example pairs to the user's description becomes crucial when generating the code for more complicated applications.

Does increasing the number of example pairs used in the prompt improve the quality of code generation?

In order to address the second research question, we examined the relationship between the number of example pairs selected and their performance in terms of BLEU score. Here we fix the selection mechanism as the revised mRMR (Method IV) and pick the example pairs in the ranked order.

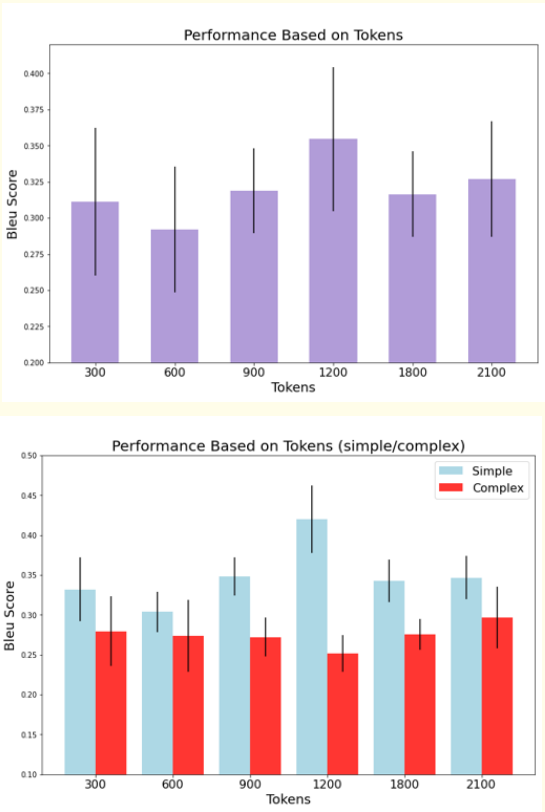

Figure 5: Results for varying the number of example pairs. The token cap values are listed along the abscissa. The plots have identical layout as those in Figure 4. See the caption of Figure 4 for details.

The Codex model processes text using tokens, which are common sequences of characters found in text or can be simply thought of as pieces of words. We varied the upper bound on the number of tokens for the entire prompt (examples, their descriptions, user query), which consequently controls the number of example pairs being added to the prompt. The number of tokens in a prompt was approximated using the Hugging Face GPT-2 Tokenizer [19]. We chose the following cap for tokens τ : 300, 600, 9000, 1200, 1800, 2100. We added the example with the highest embedding score and computed the total length in tokens of the example and user query. We continuously selected the next most relevant example pair that, once added to the prompt, would keep the prompt within the token cap.

Our results displayed in Figure 5 show that the overall difference of performance seems to be random. However, when we separate complex applications and simple applications we see some patterns. For simple apps the optimal upper limit for tokens is apparent around the 1200 mark. However, for complex apps, the quality of produced output across different prompt token caps appears to be minimal. Based on our results, there seems to exist a "sweet spot" for simple apps in terms of number of example pairs but for complex apps the quality of examples rather than quantity appears to be more important.

Can we improve the quality of code generation by ordering the example pairs differently?

In order to address the third research question, we examined the relationship between the ordering of example pairs selected and their performance in terms of BLEU score. Here we fix the selection mechanism as the revised mRMR (Method IV) and the token cap as 1000. Consider a case where the following k example pairs are selected, $<< d_1, c_1 >>< d_2, c_2 >> ... << d_k, c_k >>$. where $<< d_1, c_1 >>$ is the most relevant example pair to the user description and $<< d_k, c_k >>$ the least. There are three ways to order the example pairs within the prompt. First, we can randomly order them (which we refer to as "random"). Second, we can order them from highest ranking to lowest ranking (which we refer to as "top"), which is basically feeding the examples pairs in ranked order: $<< d_1, c_1 >>< d_2, c_2 >> ... << d_k, c_k >>$. Finally, we can order them lowest ranking to highest ranking (which we dub "bottom"), which is feeding the examples pairs in reversed ranked order: $<< d_k, c_k >>< d_{k-1}, c_{k-1} >> ... << d_1, c_1 >>$. Based on our results on Figure 6, both 'top' and 'bottom' have a marginal advantage over 'random' ordering. How-

ever, it seems that in general the orderings of the example pairs do not affect the performance of generating code. It is not certain what may be the reason for this. However, considering that we cap the tokens to 1000 and the average number of tokens for example pairs is 268, which results in an average of 3 or 4 example pairs in each prompt, it may be possible that the distance between example pairs is not far enough to make much of a difference. When we add more example pairs, the ordering may make more of a difference.

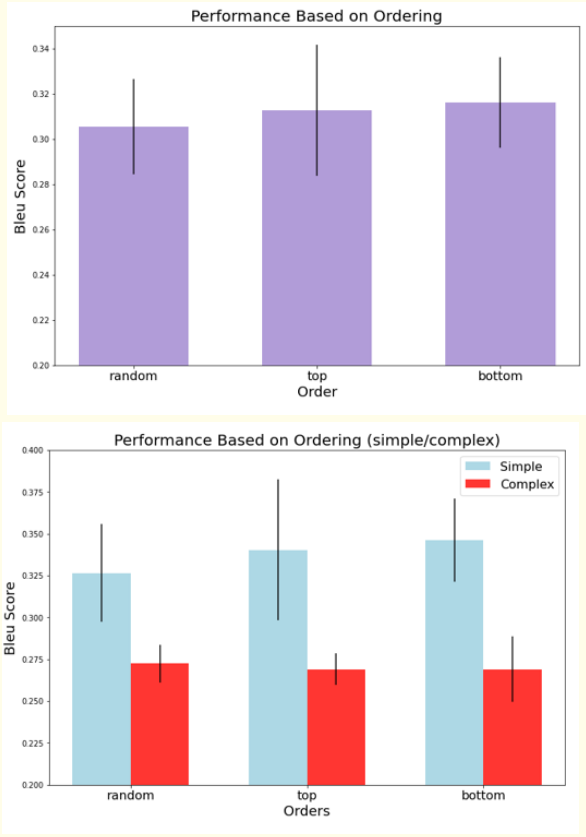

Figure 6: Results for altering the order of example pairs. The orderings are listed along the abscissa. The plots have identical layout as those in Figure 4. See the caption of Figure 4 for details.

Conclusion

We have developed an innovative learning platform that is driven by large language models, allowing students to generate mobile applications for smartphones and tablets with minimal effort using natural language descriptions. Our research goes one step further by demonstrating that the user-generated apps can be further refined with minor changes to the generative model's input, known as "prompts." We explore three different methods of prompt modification: 1) adjusting the selection mechanism of example pairs, 2) varying the number of example pairs, and 3) modifying the order of pairs within the prompt. The prompts consist of a set of example pairs that include a textual description of an example app and its corresponding code, as well as a description of the desired app. We evaluate the performance of the model by testing it with 18 different mobile app options, ranging from basic to complex, and compare the results to manually created apps using the BLEU score. Our research demonstrates that selecting the appropriate example pair and varying the number of examples can significantly affect the quality of the generated apps, while reordering the example pairs has no effect.

Limitations and Future work

While our study demonstrates the potential of using natural language processing to generate mobile apps, there are several ways in which our work can be continued and improved. Due to resource constraints, we were unable to test a larger amount of data. Furthermore, our test data was created in a controlled laboratory setting, which may not accurately reflect real-world mobile app development scenarios. We plan to expand our test data by collecting a larger and more diverse set of mobile app descriptions from real-world scenarios. This can be achieved by using crowdsourcing or scraping app descriptions from app stores. We also aim to explore transfer learning techniques to fine-tune the model on specific domains or industries, such as mathematical or language oriented apps, which may require specialized features. By doing so, we can increase the generalizability of our approach and improve its real-world applicability.

To enhance the credibility of our results, we plan to incorporate more evaluation metrics beyond BLEU score. Conduct unit testing to verify detailed functionalities of each app and its ability to perform all requested tasks. Another option is to use a more fine-grained evaluation method that takes into account the specific requirements and constraints of each test case. For instance, we can define a set of metrics that measure the app's usability, functionality, and design, and evaluate the model's outputs accordingly. This can be achieved by incorporating user feedback or expert evaluation into the evaluation process. We will explore additional evaluation methods such as automated testing and code analysis to

ensure that the generated app is free of bugs and errors.

Finally, we will explore additional methods of automated prompt construction that can potentially improve the performance of our approach. For example, we can investigate the use of more advanced techniques for selecting and preprocessing the example pairs, such as clustering or feature extraction, that can better capture the underlying patterns and structures in the data. Moreover, we plan to conduct a thorough analysis of the hyperparameters of the Codex model and evaluate their effect on the quality of the generated code. This can be achieved by conducting a hyperparameter tuning experiment, where we systematically vary the values of the hyperparameters and evaluate the model's performance on a held-out validation set. In this manner, we may better identify the optimal set of hyperparameters that maximize LLM performance. We also plan to compare the performance of Codex with other state-of-the-art large language models trained on code such as Meta AI's InCoder [20] in generating mobile apps. This can be achieved by conducting a systematic evaluation of the performance of different models on a standardized set of test cases and comparing their results. By doing so, we can identify the strengths and weaknesses of each model and provide insights into the future development of generative models for code.

Discussion

There are many challenges young learners face when trying to develop impactful computational solutions. Many of these can be attributed to the context of computing education itself, often taking place in traditional computing labs, which are far removed from students' everyday lives. Too often, K-12 computing education has been driven by an emphasis on kids learning the "fundamentals" of programming such as variables, loops, conditionals, parallelism, operators, and data handling [21]. This often discourages students from being part of the technological community. To empower young people to build these solutions, we need to provide platforms and learning environments that reduce the technological barriers for app creation to emphasize the students' ideas. In this study, we've worked on optimizing a platform that aims to harness the power of AI to take an user description of an app and generate an app that matches that description. There are several potential educational uses of such a platform. Furthermore, our work has many potential applications in the democratization of app creation; not only children but also adults will be able to create meaningful apps without prior experience in programming. Seniors continue to lag their younger compatriots when it comes to tech adoption [22]. A significant majority of older adults say they need assistance when it comes to using new digital devices. Just 18% would feel comfortable learning to use a new technology device such as a smartphone or tablet on their own. Our new platform enables them to bypass these obstacles— we aim to make computing education more inclusive, more motivating, and more empowering. We hope that through our work, we are one step closer to the goal of enabling everyday person app ideas into real-life mobile applications.

Acknowledgments

We thank Harold Abelson, Mark Friedman for helping the initial stage of the research and providing feedback for the draft, we also thank Ashley Granquist and Maura Kelleher for contributing on designing Aptly-Script.

Conflict of Interest

Bibliography

1. David Wolber., *et al*. "Democratizing Computing with App Inventor". *GetMobile: Mobile Computing and Communications* 18.4 (2015): 53-58.

2. Resnick M., *et al*. "Scratch: programming for all". *Communications of the ACM* 52.11 (2009): 60-67.

3. Brown Tom., *et al*. "Language models are few-shot learners". *Advances in Neural Information Processing Systems* 33 (2020): 1877-1901.

4. Oppenlaender Jonas. "Prompt Engineering for Text-Based Generative Art". arXiv preprint arXiv:2204.13988. (2022).

5. Reynolds Laria., *et al*. "Prompt programming for large language models: Beyond the few-shot paradigm". Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems (2021).

6. Ramesh Aditya., *et al*. "Hierarchical text-conditional image generation with clip latents". arXiv preprint arXiv:2204.06125 (2022).

7. Rombach Robin., *et al*. "High-resolution image synthesis with latent diffusion models". Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (2022).

8. Chen Mark., *et al*. "Evaluating large language models trained on code". arXiv preprint arXiv:2107.03374 (2021).

9. Gao Tianyu., *et al*. "Making Pre-trained Language Models Better Few-shot Learners". Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers). Association for Computational Linguistics (2021).

10. Taylor Shin., *et al*. "AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts". Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP). Association for Computational Linguistics, (2020).

11. Zhang Yue., *et al*. "PromptGen: Automatically Generate Prompts using Generative Models". Findings of the Association for Computational Linguistics: NAACL 2022. Association for Computational Linguistics, (2022).

12. Kim, David YJ, et al. "SPEAK YOUR MIND: INTRODUCING APTLY, THE SOFTWARE PLATFORM THAT TURNS IDEAS INTO WORKING APPS." *ICERI2022 Proceedings*. IATED, 2022.

13. Zhao Zihao., *et al*. "Calibrate before use: Improving few-shot performance of language models". *International Conference on Machine Learning* (2021).

14. Papineni Kishore., *et al*. "Bleu: a Method for Automatic Evaluation of Machine Translation". Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics. Association for Computational Linguistics, (2002).

15. Neelakantan Arvind., *et al*. "Text and code embeddings by contrastive pre-training". arXiv preprint arXiv:2201.10005 (2022).

16. Radovic Milos., *et al*. "Minimum redundancy maximum relevance feature selection approach for temporal gene expression data". *BMC Bioinformatics* 18.1 (2017): 1-14.

17. Zhao Zhenyu., *et al*. "Maximum relevance and minimum redundancy feature selection methods for a marketing machine learning platform". 2019 IEEE international conference on data science and advanced analytics (DSAA) (2019).

18. Masson Michael EJ., *et al*. "Using confidence intervals for graphically based data interpretation". *Canadian Journal of Experimental Psychology/Revue Canadienne de Psychologie Expérimentale* 57. 3 (2003): 203.

19. Wolf Thomas., *et al*. "Huggingface's transformers: State-of-the-art natural language processing". arXiv preprint arXiv:1910.03771 (2019).

20. Fried, Daniel., *et al*. "Incoder: A generative model for code infilling and synthesis". arXiv preprint arXiv:2204.05999 (2022).

21. Tissenbaum M., *et al*. "From computational thinking to computational action". *Communications of the ACM* 62.3 (2019): 34-36.

22. Smith A. "Older adults and technology use. Technical report". *Pew Research Center* (2014).