Design and Optimization of an Automatic Mobile Application Generating Learning Platform

Jasmine L Shone,[1] Robin Liu,[2] Evan Patton,[2] David Y.J. Kim,[2]

[1] Hawken Upper School, USA

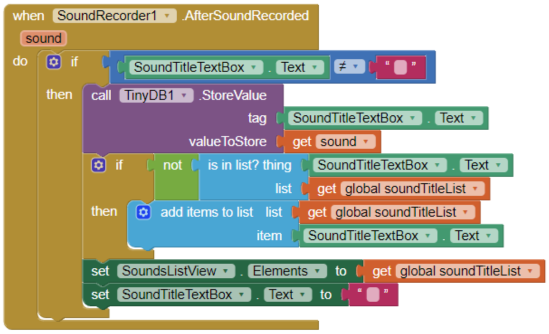
[2] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, USA
jasshone@gmail.com, robinl21@mit.edu, ewpatton@mit.edu, dyjkim@mit.edu

Abstract

We present a large language model-based learning platform that lets students automatically generate mobile applications for smartphones and tablets from natural language descriptions. Furthermore, we show that the user-generated apps can be optimized with simple modifications to the generative model's input ("prompts"). This paper explores three different methods of modifying the prompt: *1)* changing the selection mechanism of example pairs, *2)* varying the number of example pairs, and *3)* altering how the pairs are ordered within the prompt. Prompts are constructed from a set of example pairs (a textual description of an example app and its corresponding code) along with the description of the desired app. We evaluated the model's performance with 18 possible candidate mobile apps, ranging from simple to complex, and used the BLEU score to compare the model's outputs to manually created apps. Our results show that appropriate example pair selection and variation of the number of example pairs make a difference in the quality of the generated apps, but alteration of example pair ordering does not. We conclude with a discussion about the potential implications for CS education in light of generative models for code.

Introduction

In this paper, 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 for teaching and learning how to create one's own mobile apps has led many educators to design curriculum targeting that purpose (Hsu and Ching 2013). Nonetheless, many students are discouraged from creating their own app because the task of learning the necessary programming expertise appears daunting (HT Tech 2011). 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 coding needed through techniques such as drag-and-drop functionality and block coding (Figure 1). Several learning platforms, such as Scratch (Resnick et al. 2009), attempt to make the learning curve less steep and encourage students to build their own apps. 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.



```
when SoundRecorder1 .AfterSoundRecorded
 sound
do  if  SoundTitleTextBox . Text  ≠  " "
    then  call TinyDB1 .StoreValue
                            tag  SoundTitleTextBox . Text
                   valueToStore  get sound
          if  not  is in list? thing  SoundTitleTextBox . Text
                                list  get global soundTitleList
          then  add items to list  list  get global soundTitleList
                                   item  SoundTitleTextBox . Text
          set SoundsListView . Elements to  get global soundTitleList
          set SoundTitleTextBox . Text to  " "
```

Figure 1: Example of a block code

Large language models (LLMs), such as GPT-3, have demonstrated that they can perform a wide range of text-based tasks through carefully crafted model input (Brown et al. 2020). The input of LLMs are often referred to as *prompts* (Reynolds and McDonell 2021; Oppenlaender 2022). A number of 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 (Ramesh et al. 2021, 2022). 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* (Brown et al. 2020). 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 (Chen et al. 2021) 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. Codex is proficient in many programming languages, but it is most capable in Python. We designed the prompt so that Codex would produce Aptly-Script, an intermediate Python-like language that can be converted into MIT App Inventor block codes (Turbak et al. 2012). Aptly-Script was designed so that the functions and classes have a one-to-one correspondence with App Inventor components. For example, a Text-to-Speech component in block-coding would have the same callable methods in Aptly-Script. Once the block code

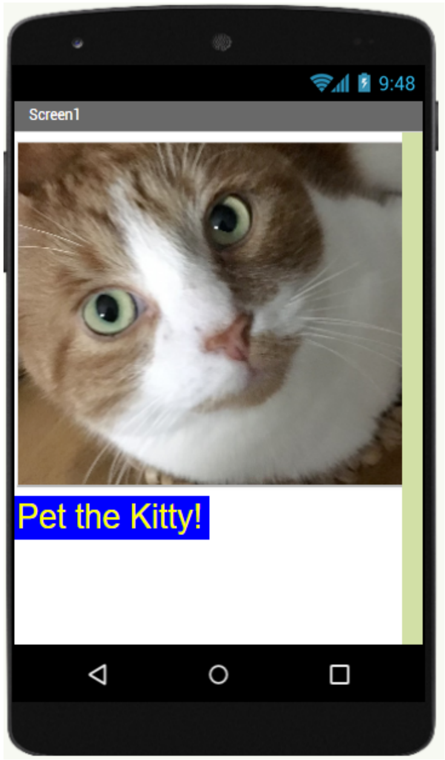


Figure 2: A fully functional application with a picture of a kitty that, when clicked, plays a meow sound.

is created, we can use the MIT App Inventor platform 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"*,

We will then automatically synthesize a prompt by first concatenating a number of example pairs, where each example pair 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. Then we follow the example pairs with the requested textual description of the translator app to complete the prompt. We feed this prompt into Codex which will generate the Aptly-Script for the Hello Kitty application. We can then convert the generated Aptly-Script into App Inventor blocks to generate a fully functional application (Figure 2).

We also show we can further improve the performance of our platform with some prompt engineering tricks. For instance, we can intuitively expect that the content of example pairs would affect the performance of app creation. When a user asks for an application that translates English to Spanish, the provision of example pairs that showcase how to use the "translation component" could help the model complete the user request. Another intuitive expectation we can have is that the more example pairs we provide, the more the model can learn. Thus, the amount of example pairs will affect the app creation process. Finally, past studies have observed that the placement of an example in a prompt has been found to potentially change the emphasis Codex places on the example within its few-shot learning process; a study about GPT-3, found that examples placed closer to the end had a greater impact on the generated results (Zhao et al. 2021). With all these possibilities 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*?

Methods

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:

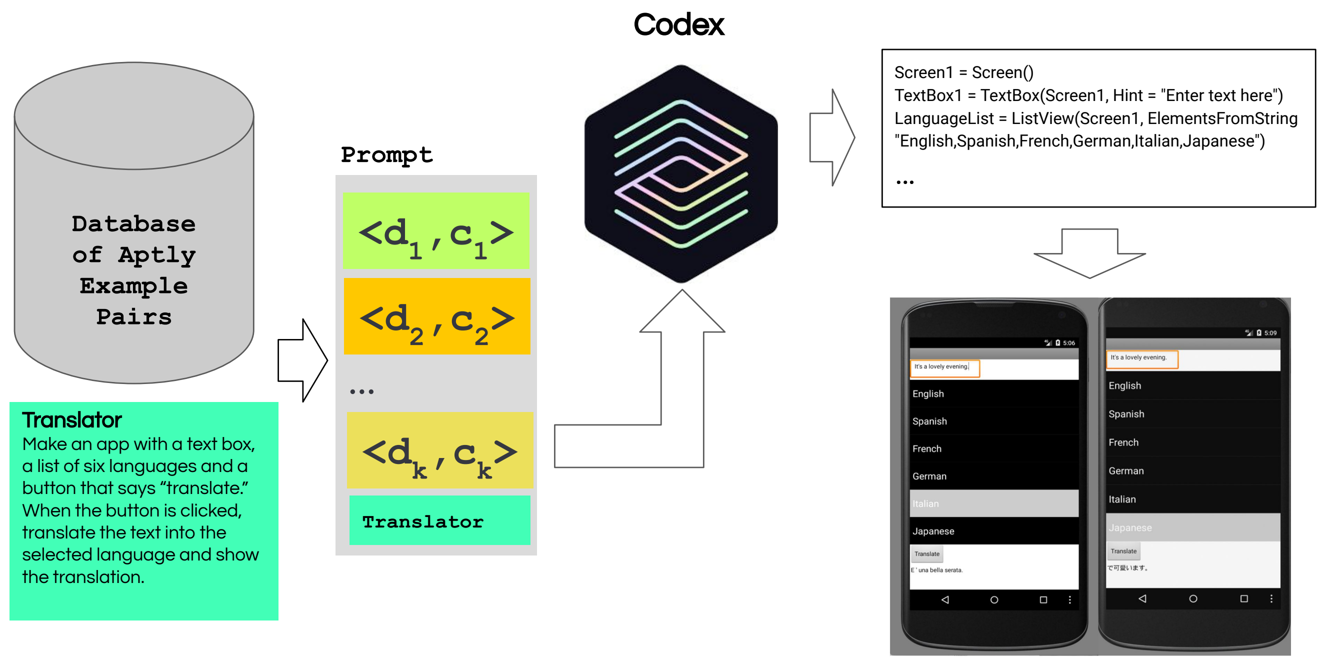
```
Create an app called HelloPurr with a
picture of a kitty that, when clicked,
plays a meow sound.

START
Screen1 = Screen(AppName="HelloPurr")
Cat = Button(Screen1, Image="kitty.png")
Meow = Sound(Screen1, Source="meow.mp3")

when Cat.Click():
    call Meow.Play()
STOP
```

As mentioned before, when a user requests a certain application, we automatically select a certain number of example pairs and synthesize a prompt. Descriptions of the code are enclosed within "% %" delimiters to indicate that they are not part of the code to Codex. 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 (Chen et al. 2021), 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).

We validated the model with 18 descriptions of candidate mobile applications that served as plausible student app requests. Within these candidates we included simple applications, such as *"Make a game that has a button in the middle of the screen. The button has a picture of a cookie on it. When 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. We also included complex applications such as *"Make a creature that the user can feed, wash, and cuddle with buttons. Each time the user performs*



Translator

Make an app with a text box, a list of six languages and a button that says "translate." When the button is clicked, translate the text into the selected language and show the translation.

```
Screen1 = Screen()
TextBox1 = TextBox(Screen1, Hint = "Enter text here")
LanguageList = ListView(Screen1, ElementsFromString
"English,Spanish,French,German,Italian,Japanese")
...
```

It's a lovely evening.

E ' una bella serata.

で可愛います。

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

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, 2 per problem. The solutions for the same problem are designed to be as different as possible from each other and cover different interpretations 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 BLEU score, or the Bilingual Evaluation Understudy (Papineni et al. 2002), between the two reference solutions manually crafted by MIT App Inventor and the generated Codex output. BLEU's output is always a number between 0 and 1. This value indicates how similar the candidate text is to the reference texts, with values closer to 1 representing more similar texts. The central idea behind using BLEU is "The closer a machine generated code is to a professional human code, the better it is". We use the nltk Python library to compute the BLEU score.

Results

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

In order 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 (will further explain later) 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 select 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 con-

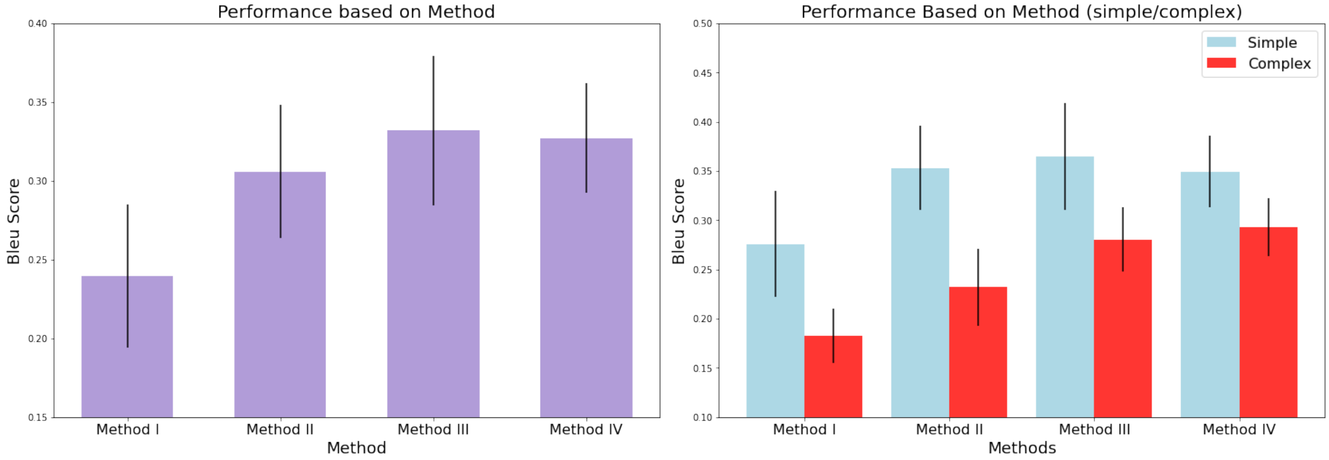


Figure 4: Results for different selection methods. The different methods are listed along the abscissa. Each bar indicates the mean Bleu score across test data; error bars reflect ±1 standard-error of the mean, corrected to remove variance due to the random factor (Masson and Loftus 2003).

verted to number sequences. In our scenario, an *embedding* represents the semantic meaning of a natural language description or code (Neelakantan et al. 2022). 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 (Neelakantan et al. 2022).

Method IV: Revised Maximum Relevancy Minimum Redundancy We rank the examples using a revised version of Minimum Redundancy Maximum Relevance (mRMR) (Radovic et al. 2017). mRMR is currently used in machine learning (Zhao, Anand, and Wang 2019) 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) (Radovic et al. 2017). Since none of this information is obtainable for our case, 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:

$$\max_{c_j \in C - S_{n-1}} \left[I(c_j; D) - \frac{1}{n-1} \sum_{c_i \in S_{n-1}} I(c_j; c_i) \right]$$

where c_k is the kth code example, D is the user requested app description, S_{n-1} is the set of example pairs already selected, and $I(x; y)$ is the mutual information (i.e. the cosine similarity) between x and y. 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 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. We then report the mean and standard-error across test samples. 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.

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 (Wolf et al. 2019) . 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 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. However, 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.

Summary & Conclusion

In summary, we explored three different characteristics of choosing few-shot examples for the generative model to learn from a given user app description: how examples are chosen, the number of tokens in the prompt, and how the examples are ordered within the prompt. For each characteristic we obtained several key results, including: (1) selecting more appropriate example pairs affects performance especially for more complex applications; (2) for simple apps there exists an optimal number of example pairs to improve the generative performance, while for complex apps the number of example pairs have marginal to none effect (3) The orderings of the example pairs had no effect on the performance. Our results suggest that crafting a suitable prompt is crucial for generating the correct mobile application. Without theoretical justification for the representation which yielded the best performance, we improved the performance for complex apps by 55% (0.10 increase in BLEU score) and 43% (0.13) for simple apps using example selection mechanisms. Our research is important and novel in that as far as we know this is the first attempt to investigate prompt engineering in terms of mobile app creation.

However, our research has several potential limitations. To start with, due to limited resources we were not able to experiment with abundant amounts of test data. Additionally, our test data was created in a laboratory setting, which may not reflect the possible mobile applications that people might want to create in the real world. We do not also consider the individual outputs of each test case. Different applications require different features; some applications might require more UI design while other applications might require more robust functionality. We can examine each individual test case and use different sample selection based on the application's needs. Finally, The BLEU score attempts to evaluate how similar Codex can program to a human programmer. However, it is unable to check exact syntax errors, check functionality, and it may disregard certain ways to create a desired application. We plan to address these issues in future research. There are also some potential changes to the automated prompt construction that were not examined within this study. One of these was tuning the hyper-parameters of Codex, similar to how the authors of the Codex paper examined the effect of, for example, model temperature on the quality of generated code (Chen et al. 2021). Additionally, we plan to do unit testing, as it is important to test detailed functionalities of each app and check its ability to implement all requested tasks. Finally, the performance of other large language models such as Meta AI's InCoder in generating apps can be compared to Codex's performance (Fried et al. 2022).

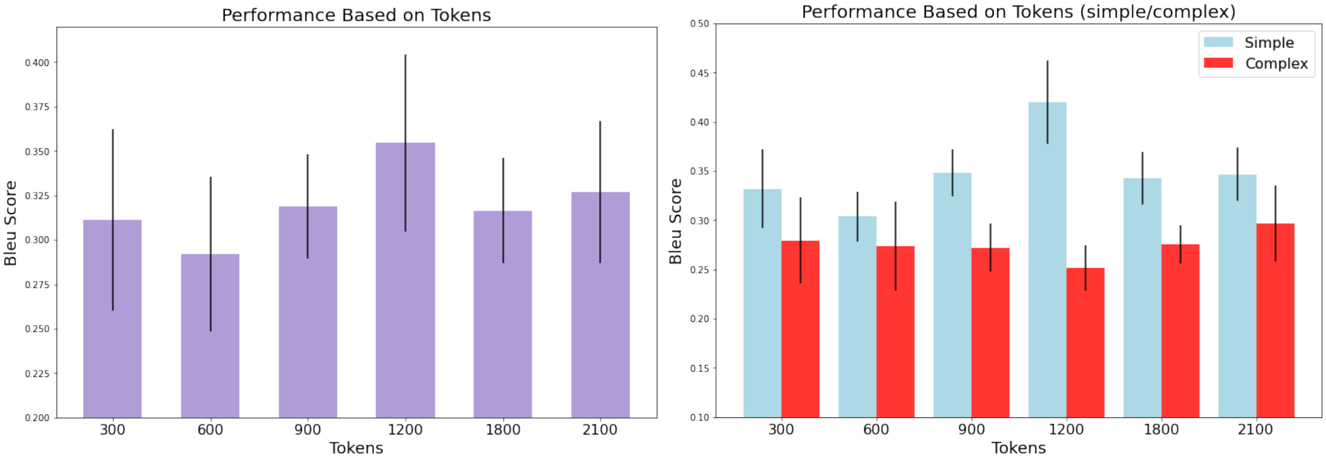


Figure 5: Results for varying the number of example pairs. The token cap values are listed along the abscissa. The plots have identical layout as those in Figure 4. See the caption of Figure 4 for details.

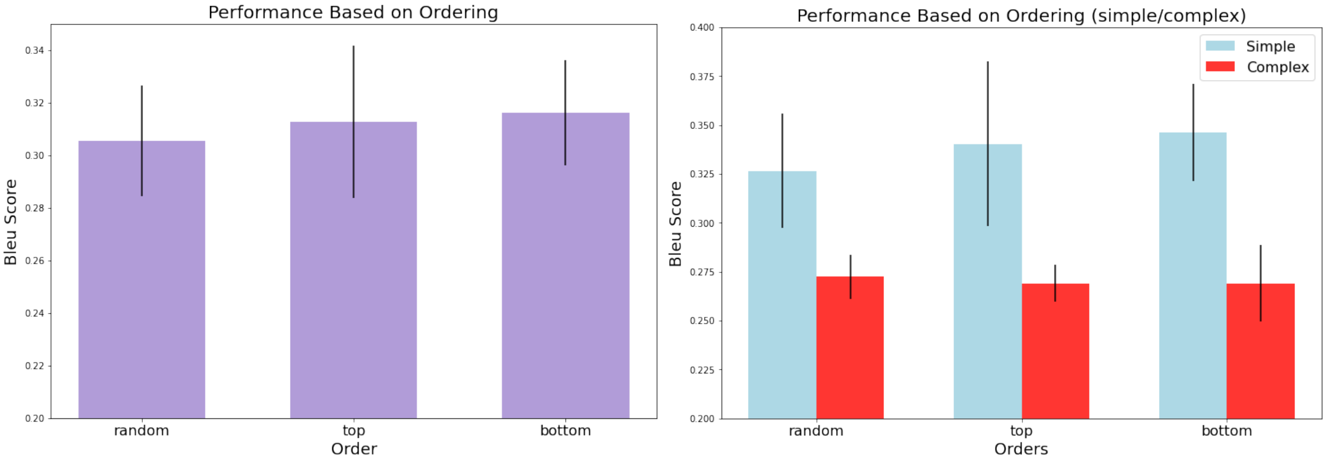


Figure 6: Results for altering the order of example pairs. The orderings are listed along the abscissa. The plots have identical layout as those in Figure 4. See the caption of Figure 4 for details.

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 (Tissenbaum, Sheldon, and Abelson 2019). This often discourages students from being part of the technological community. In order 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 behind their younger compatriots when it comes to tech adoption (Smith 2014). 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 anyone with an app idea being able to convert it to an usable mobile application.

References

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fried, D.; Aghajanyan, A.; Lin, J.; Wang, S.; Wallace, E.; Shi, F.; Zhong, R.; Yih, W.-t.; Zettlemoyer, L.; and Lewis, M. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Hsu, Y.-C.; and Ching, Y.-H. 2013. Mobile app design for teaching and learning: Educators' experiences in an online graduate course. *International Review of Research in Open and Distributed Learning*, 14(4): 117–139.

HT Tech. 2011. The future of it lies in Democratising application development. https://tech.hindustantimes.com/tech/news/the-future-of-it-lies-in-democratising-application-development-71622304033195.html.

Masson, M. E.; and Loftus, G. R. 2003. Using confidence intervals for graphically based data interpretation. *Canadian Journal of Experimental Psychology/Revue canadienne de psychologie expérimentale*, 57(3): 203.

Neelakantan, A.; Xu, T.; Puri, R.; Radford, A.; Han, J. M.; Tworek, J.; Yuan, Q.; Tezak, N.; Kim, J. W.; Hallacy, C.; et al. 2022. Text and code embeddings by contrastive pretraining. *arXiv preprint arXiv:2201.10005*.

Oppenlaender, J. 2022. Prompt Engineering for Text-Based Generative Art. *arXiv preprint arXiv:2204.13988*.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W.-J. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 311–318. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics.

Radovic, M.; Ghalwash, M.; Filipovic, N.; and Obradovic, Z. 2017. Minimum redundancy maximum relevance feature selection approach for temporal gene expression data. *BMC bioinformatics*, 18(1): 1–14.

Ramesh, A.; Dhariwal, P.; Nichol, A.; Chu, C.; and Chen, M. 2022. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*.

Ramesh, A.; Pavlov, M.; Goh, G.; Gray, S.; Voss, C.; Radford, A.; Chen, M.; and Sutskever, I. 2021. Zero-Shot Text-to-Image Generation. *ArXiv*, abs/2102.12092.

Resnick, M.; Maloney, J.; Monroy-Hernández, A.; Rusk, N.; Eastmond, E.; Brennan, K.; Millner, A.; Rosenbaum, E.; Silver, J.; Silverman, B.; et al. 2009. Scratch: programming for all. *Communications of the ACM*, 52(11): 60–67.

Reynolds, L.; and McDonell, K. 2021. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 1–7.

Smith, A. 2014. Older adults and technology use. Technical report, Pew Research Center.

Tissenbaum, M.; Sheldon, J.; and Abelson, H. 2019. From computational thinking to computational action. *Communications of the ACM*, 62(3): 34–36.

Turbak, F.; Sandu, S.; Kotsopoulos, O.; Erdman, E.; Davis, E.; and Chadha, K. 2012. Blocks languages for creating tangible artifacts. In *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 137–144. IEEE.

Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Zhao, Z.; Anand, R.; and Wang, M. 2019. Maximum relevance and minimum redundancy feature selection methods for a marketing machine learning platform. In *2019 IEEE international conference on data science and advanced analytics (DSAA)*, 442–452. IEEE.

Zhao, Z.; Wallace, E.; Feng, S.; Klein, D.; and Singh, S. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*, 12697–12706. PMLR.

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.