Yes, You Can Make an App Too: A Systematic Study of Prompt Engineering in the Automatic Generation of Mobile Applications from User Queries

Jasmine Shone

Under the direction of

David Kim
M. S.
Hal Abelson Lab, MIT CSAIL

Research Science Institute
August 1, 2022

Abstract

Aptly aims to use large language models to allow for on-the-fly generation of full apps given only an user description of an app idea. In order to optimize this platform and to provide guidance for other platforms or companies aiming to personalize large language models (LLMs) for their needs, we embark on one of the first systematic studies of prompt engineering for the Codex model, a LLM that produces code from a natural-language input. Specifically, we examine the effect of varying the token length, mechanism of choosing examples (random selection, least to most token size, cosine similarity, or an adapted version of Minimum Redundancy Maximum Relevance), and how they are ordered within the prompt (highest to lowest ranked, lowest to highest ranked, randomly shuffled) on the quality of generated code. We improve the pipeline's performance from baseline for complex apps by 55% (0.10 increase in BLEU score) using example selection mechanisms and 43% (0.13) for simple apps.

Summary

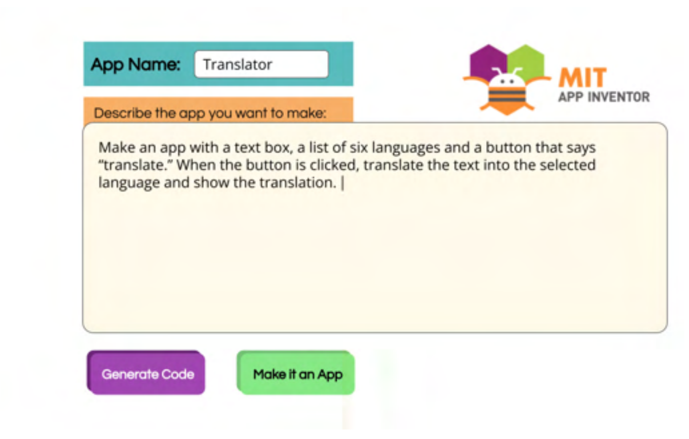
App creation should be accessible to everyone, regardless of technical expertise and amount of time. The Aptly platform aims to make this ideal a reality by enabling near-instantaneous generation of full apps given only an user description of an app idea. In order to optimize Aptly's abilities and tackle a broader question within the large language model community about how to best construct inputs (prompts), we examine the effects of token length, mechanism of choosing examples, and how examples are ordered within the prompt on the quality of Codex's outputs of app code through the construction of a three step automated prompt creation pipeline. We find that these three characteristics affect the quality of the code produced by Codex. Further research is needed on prompt engineering for code-generating large language models such as Codex.

1 Introduction

In an ideal world, anyone who has an idea for an app should be able to convert that idea into a real app they can use and distribute. However, in the status quo, this transformation of idea to app is often impeded by a lack of time and lack of technical expertise [1]. As a result, there has been a continuous drive to tackle these barriers within the scientific and industrial computer science community through the creation of “Low Code” or “No Code” platforms that aim to reduce the amount of coding needed through techniques such as drag-and-drop functionality and block coding to simplify the process of app creation [2]. Aptly, a new No Code platform in development by the MIT App Inventor team, aims to take this simplification of app creation one step further. The goal of Aptly is to take a user description of an app and directly convert it into a full, working mobile app usable on Apple and Android devices.

Aptly has the potential to further lower the barriers of time and technical experience needed in the app creation process. Aptly’s interface is designed to be simple and intuitive, as shown in Figure 1. Instead of the user needing to spend time implementing their idea and debugging, the process of creating apps becomes nearly instantaneous. Additionally, Aptly removes the necessity for knowing the technical details of coding and instead places the emphasis on the logical thinking needed to describe an idea.

The Aptly platform uses Open AI’s Codex, a large language model that is able to generate code from text-based prompts [3]. A prompt is generally composed of a natural language description of a desired app to be created, which can also be preceded by a few examples from which the large language model can learn from. The examples included in the prompt are known as “few-shot” examples which aim to help Codex learn how to create the target application, similar to how a student might learn how to do multiplication by learning from example problems [4]. In past studies, adding few-shot examples to prompts has had



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. |

Figure 1: A prototype of the Aptly platform

substantial success in improving Codex's performance in completing user tasks [3, 5].

In order to improve Aptly's ability to generate full, workable apps from a user query, we examine techniques to optimize the generation of prompts on-the-fly from a user description typed into the Aptly platform and sent to Codex to obtain a code output.

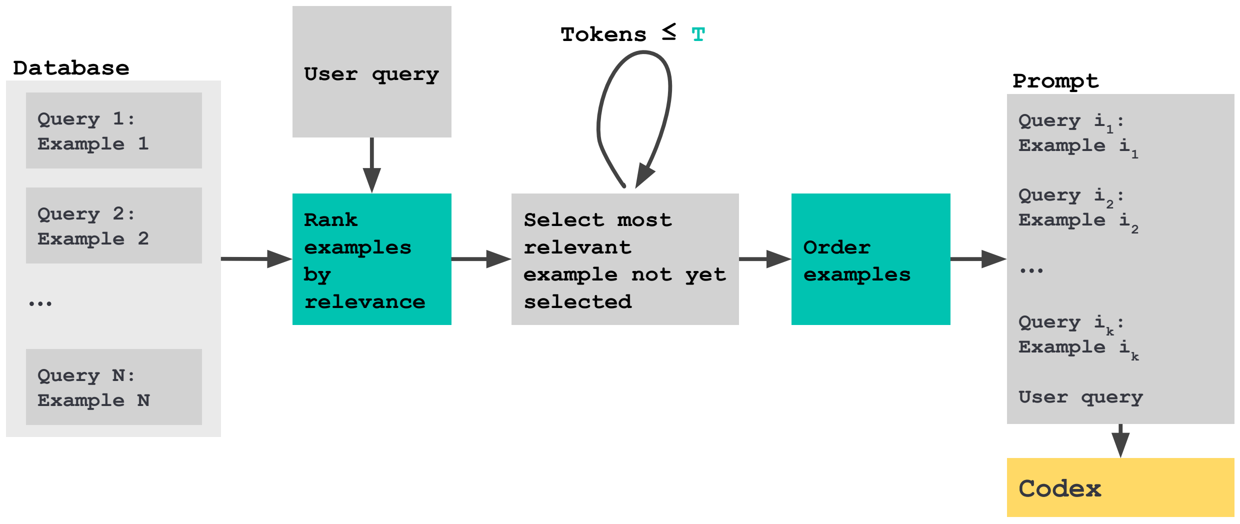
Specifically, we focus on three characteristics of the prompt:

1. Does the quality of code generation differ based on how examples are chosen?
2. Does increasing the number of tokens (units of semantic meaning in the prompt) sent to Codex improve the quality of code generation?
3. Can we improve the quality of code generation by ordering the examples differently?

As optimization of prompts for Large Language Models, or prompt engineering, is a relatively emerging area of machine learning research, this study is one of the first aiming to examine the effects of varying the characteristics of prompts on generated output [6].

2 Methods

In order to generate prompts on the fly, we constructed a three-step automatic pipeline that takes a user description and adds few-shot examples to formulate a complete prompt.



Database
Query 1: Example 1
Query 2: Example 2
...
Query N: Example N
User query
Rank examples by relevance
Tokens ≤ T
Select most relevant example not yet selected
Order examples
Prompt
Query i_1: Example i_1
Query i_2: Example i_2
...
Query i_k: Example i_k
User query
Codex

Figure 2: Prompt Engineering Pipeline

Each of the steps corresponds to one of the three characteristics of the prompt that we sought to optimize: how examples are chosen, how many tokens are in the prompt, and how the chosen examples are ordered within the prompt. Each parameter that is examined is colored in turquoise in Figure 2.

2.1 Data

A database of 85 unique app examples was compiled by the App Inventor 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 an intermediate language named Aptly-Script whose functions and classes have a one-to-one correspondence with App Inventor components (for example, having a Text-to-Speech component with the same callable methods). A basic description was also created for each app.

2.2 Ranking Mechanism

The pipeline sorts the examples in the database to determine the order in which examples are added to the prompt. We tested four different ways of doing this. First, we chose to randomly rank examples as the baseline option.

Second, we ranked examples from those with the least amount of tokens to those with the most amount of tokens. This option had the advantage of sending in the most examples for Codex to learn from. However, the ranking of examples may not reflect what is the most relevant to the requested description.

Third, we rank the examples based on how semantically relevant they are to the user query. We do so by generating embeddings for each app example and the user description. An *embedding* is a 2048-dimensional vector that represents the semantic meaning of a natural language description or code file [7]. Codex's Babbage engine was used to generate code embeddings for each example app in the database [3].

To measure the semantic similarity between combinations of pieces of code and text descriptions, the cosine of the angle between the vectors mapped in 2048-dimensional space was taken (cosine similarity). More specifically, cosine similarity is calculated as follows:

$$\cos(\theta_{\boldsymbol{x},\boldsymbol{y}}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}||||\boldsymbol{y}||},$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors in the same dimensional space and output values close to 1 indicate high similarity. Examples were ranked from highest cosine similarity to lowest cosine similarity.

Fourth, we rank the examples using a revised version of Maximum Relevancy Minimum Redundancy (MRMR) [8]. MRMR is currently used in machine learning as a relatively efficient way of reducing the amount of independent variables in a dataset such that information about the dependent variable is maximized and correlation between selected variables is minimized [9]. In other words, the goal is to find the "minimal optimal" set of variables to predict the dependent variable.

The advantages of MRMR could be potentially useful in the task of few-shot learning because some examples in a database may be duplicates or have similar functionality with already chosen examples, meaning that they wouldn't contain new information for Codex to learn [8].

The original formula in MRMR for selecting the mth variable given a variable space of X (set of all independent variables) and an already selected set $S_{m-1} of m-1$ variables is

$$\text{Selection} = \max_{x_j \in X - S_{m-1}} \left[I(x_j; c) - \frac{1}{m-1} \sum_{x_i \in S_{m-1}} I(x_j; x_i) \right],$$

where x_j is the jth variable under evaluation, c is the target variable, x_i is the ith feature in S_{m-1}, and $I(x; y)$ is the mutual information between x and y. Intuitively, this formula rewards variables that maximize the information between itself and the target variable (represented in $I(x_j; c)$) and penalizes variables with a high mutual information with the already selected S_{m-1} of features (represented by $\frac{1}{m-1} \sum_{x_i \in S_{m-1}} I(x_j; x_i)$).

To adapt MRMR to selecting code examples instead of selecting variables, we quantify $I(x_j; c)$ as the cosine similarity between the embeddings of the user query and that of the code example as similar semantics/functionality implies mutual information that Codex can use for learning. Instead of the traditional MRMR schema of using a greedy approach to add k total examples, we use a similar schema of adding examples given that the prompt does not exceed token length T. This method is further discussed in Subsection **2.3**.

2.3 Number of Tokens

After a ranking was generated, we varied the upperbound on the number of tokens in the entire prompt (examples, their descriptions, user query) to be $T = 300$, 600, 9000, 1200, 1800,

2100. The number of tokens in a prompt was approximated using the Hugging Face GPT-2 Tokenizer [10]. We implemented this behavior for random, size, and embeddings-based rankings through recursively selecting the highest ranked example not yet chosen given that the number of tokens in the prompt are $\leq T$.

For MRMR, we used a greedy approach: first, we added the example with the highest embeddings score and computed the total length in tokens of the example and user query. Then, given that the total length was $< T$, the algorithm computed the relevance of the remaining examples using the formula in Section 2.1.2. It subsequently selected the example with highest relevance given that the length of the prompt including the new example is still less than or equal to T tokens. For the next examples, the process repeats: the relevance is calculated for all examples given an updated S_{m-1} and given tokens $< T$ the mth example is chosen. This process is elaborated on in **Algorithm 1**.

2.4 Ordering of Examples

After examples are selected, they are ordered within the prompt. This is done in three ways: highest ranking to lowest ranking (which we dub "top"), lowest ranking to highest ranking (which we dub "bottom"), and random ordering (which we dub "random"). 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, the predecessor of Codex, found that examples placed closer to the end more heavily biased generated results [11].

2.5 Sending the Prompt

Each chosen example is represented in the prompt in the following order: its textual description, code, and a 'STOP' token at the end of the code. The examples are added to

Algorithm 1 Selecting the Most Relevant Examples Given Token Limit T

```
procedure MODIFIED MRMR
    $examplearray \leftarrow$ array of *examples (description and Aptly-Script)*
    $q \leftarrow$ string of *user query*
    $sortedarray \leftarrow sort(examplearray, cosinesimilarity(examplearray, q))$
    $examplelen \leftarrow$ length of $sortedarray[0]$
    $querylen \leftarrow$ length of $userquery$
    $selected \leftarrow [0]$
    $tokenlength \leftarrow examplelen + querylen$
    if $examplelen + querylen > T$ then return []
    end if
    while $tokenlength < T$ do
        $max \leftarrow -\infty$
        $maxind \leftarrow$ null
        for $i_k \in S_{m-1}$ do
            $relevance \leftarrow$ MRMRrelevance($i_k$, $q$ )
            if relevance $> max$ then
                $max \leftarrow relevance.$
                $maxterm \leftarrow k.$
            end if
        end for
        if $tokenlength + length(S_{m-1}[maxterm]) \leq T$ then
            selected.add($maxterm$)
            $tokenlength \leftarrow tokenlength + length(S_{m-1}[maxterm])$
        end if
    end while
    return selected
end procedure
```

the prompt in the order specified in the previous step discussed in **Section 2.4**. At the end of the prompt, the algorithm concatenates the user query to the prompt. Descriptions and the query are enclosed within "% %" delimiters to indicate that they are not part of the code to Codex. The prompt is sent to the "code-davinci-002" Codex model. Hyper-parameters of the model are set as the following: temperature = 0.5, max_tokens = $4000 - T$, best_of = 10.

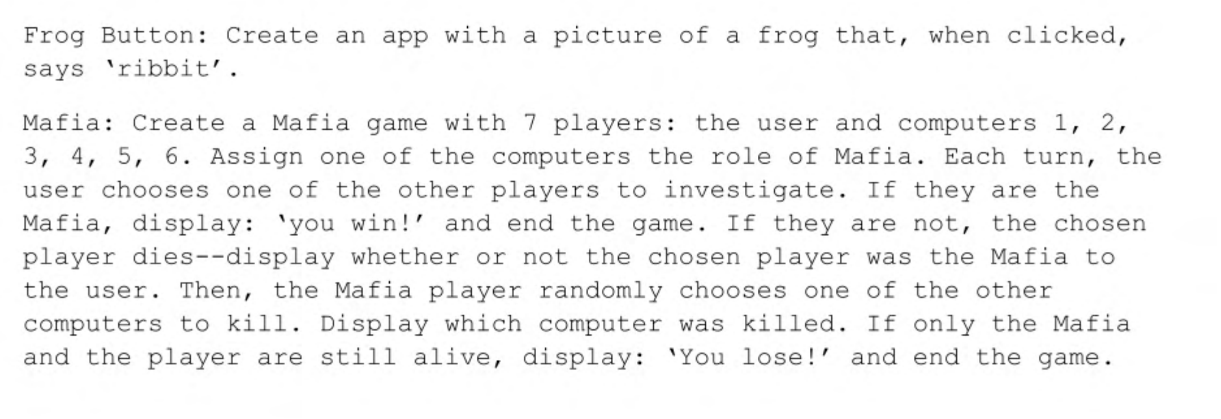
2.6 Evaluation

Evaluation was done using a combination of manual and automatic metrics. The manual evaluation was done on less data but was more precise while the automatic metric was done on more data but was less optimal than human-performed evaluation. Through presenting both metrics, we hoped to provide more context about the performance of our pipeline.

2.6.1 Automatic Evaluation

The prompt engineering pipeline was evaluated on a set of 18 example app descriptions. These descriptions were designed such that they were plausible apps that a user could want to make, such as simplified versions of the ELIZA chatbot, a language translation app, and the video game Pong.

These test apps were also designed to vary in complexity and App Inventor component usage: some perform relatively simple tasks such as playing a sound when a button is pressed while others involve more complex computational thought and components such as a simplified version of Mafia which uses randomness, variables, lists, buttons, and labels as illustrated in Figure 3.



```
Frog Button: Create an app with a picture of a frog that, when clicked,
says 'ribbit'.

Mafia: Create a Mafia game with 7 players: the user and computers 1, 2,
3, 4, 5, 6. Assign one of the computers the role of Mafia. Each turn, the
user chooses one of the other players to investigate. If they are the
Mafia, display: 'you win!' and end the game. If they are not, the chosen
player dies--display whether or not the chosen player was the Mafia to
the user. Then, the Mafia player randomly chooses one of the other
computers to kill. Display which computer was killed. If only the Mafia
and the player are still alive, display: 'You lose!' and end the game.
```

Figure 3: Examples ranged from short and less complex like 'Frog Button' to more extensive like 'Mafia'.

For each of these apps, the following combinations were tested:

Method	**Tokens**	**Order**	**Testing**
MRMR	1200	Top	Method
Embeddings	1200	Top	Method
Size	1200	Top	Method
Random	1200	Top	Method
MRMR	300	Top	Token
MRMR	600	Top	Token
MRMR	900	Top	Token
MRMR	1200	Top	Token
MRMR	1800	Top	Token
MRMR	2100	Top	Token
MRMR	1200	Top	Order
MRMR	1200	Bottom	Order
MRMR	1200	Random	Order

Each constructed prompt was used to generate 10 Codex outputs in order to account for the variability in Codex's generated code. Thus, there were 2340 total code files generated from Codex.

As manual evaluation of all 2340 code files would be time and labor-intensive and because unit-testing features of applications such as GUI is more nebulous than for unit-testing functions, we choose BLEU score as an approximation of code quality [12]. BLEU score is generally used to quickly and clearly measure the quality of machine translations by comparing it to manual "golden-standard" translations[12]. BLEU score is expressed on a scale of 0 to 1, with 1 being a perfect quality translation. In the use case of evaluating generated apps, BLEU score is potentially useful because it:

1. decreases the labor needed to evaluate output; instead of manually evaluating thousands of pieces of code, we only need to create ≤ 100 "golden-standard" code solutions.

2. has been shown to account for flexibility in how phrases are structured. This is done through the provision of more than one golden standard solution from all of which the machine output can resemble and be considered correct.

3. penalizes redundancy and accounts for the fact that the overgeneration of a word or line over and over again is not a good solution (for example, text_join(text_join(text_join(text_join is not a good solution even if the golden-standard has "text_join" in it).

To use BLEU score, 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!' We use the nltk Python library to compute the BLEU score between the two reference solutions and the generated Codex output. [13] .

2.6.2 Manual Evaluation

Next, we manually evaluated the quality of code generation for one of the test examples, a translator app. The description is as follows:

> 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.

We evaluated 130 Codex outputs of generated code for two types of bugs: functional and syntax bugs.

Functional bugs are when the generated program fails to complete a task within the user description. We quantified three specific functions for the translator app:

1. Make an app with a text box, a list of six languages and a button that says “translate.”
2. When the button is clicked, translate the text into the selected language.
3. Show the translation.

Syntax bugs are counted by the number of errors that would be thrown pre-compile. We evaluated the number of both kinds of bugs per line to explore any potential correlations between the amount of functional bugs in the code generated by Codex and the number of syntax bugs. We divided the number of bugs by number of bugs to account for longer programs being more likely to have bugs.

3 Results

In the automatic evaluation step, the performance of pipelines with different variables was measured on apps with above median complexity, on apps with below median complexity, and on all apps. App complexity was calculated with McCabe’s Cyclomatic Complexity, a common code complexity measurement that calculates the number of possible paths taken through a program. [14].

In the manual evaluation step, the amount of functional, syntactic, and total bugs per line are plotted for each variation of the prompt generation pipeline, each of which are fed into the Codex model 10 times to account for the variability within Codex’s generative output.

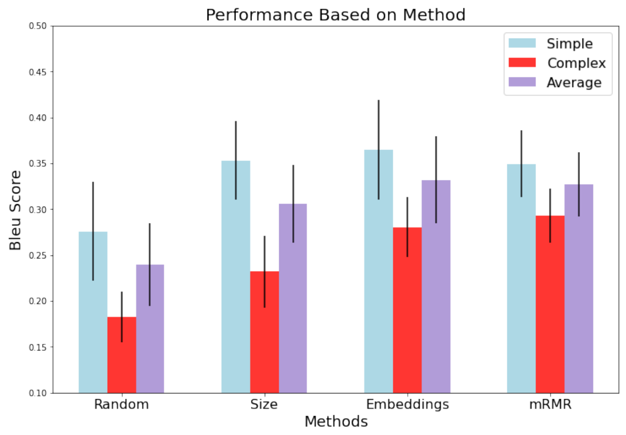


Figure 4: Automatic Evaluation of Quality of Codex-generated Code for Each Method. MRMR scores around 0.10 above random for complex apps, while embeddings performs slightly better than MRMR for simple apps at around 0.6 above random.

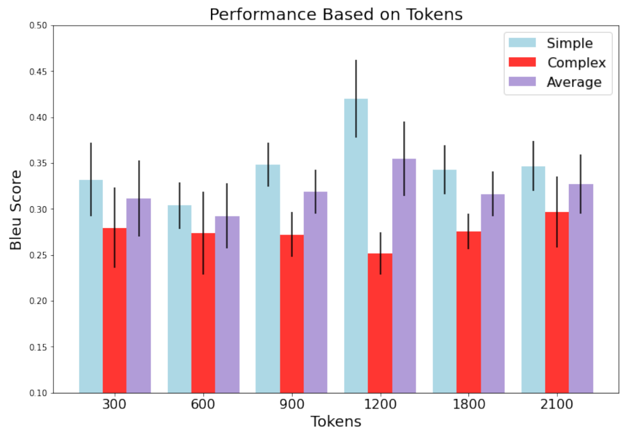


Figure 5: Quality of Codex-generated Code for Different Token Limits. For simple applications, the best performance is achieved for higher token numbers (around 1200) at an around 0.13 difference but for complex applications performance marginally varies.

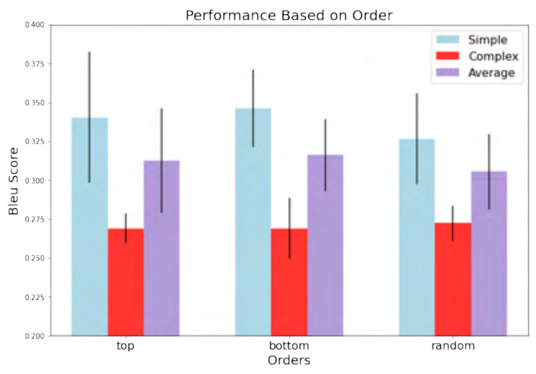


Figure 6: **Quality of Codex-generated Code for Each Order of Examples.** For simple apps, bottom ordering has marginally better performance than top and random ordering. For more complex apps, all three methods of ordering have roughly equal performance.

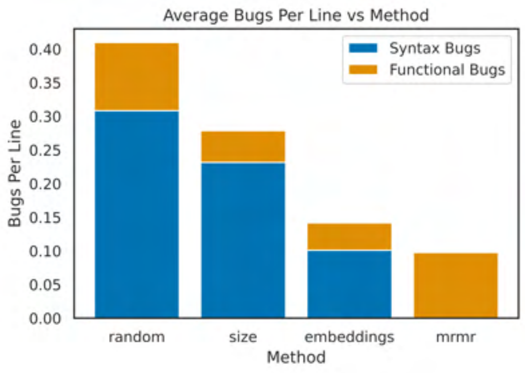


Figure 7: **Bugs in Codex-generated Code for Different Methods.** The embeddings-based pipeline generated code with the least amount of functional bugs. The mrmr-based pipeline generated code with the least amount of syntax bugs.

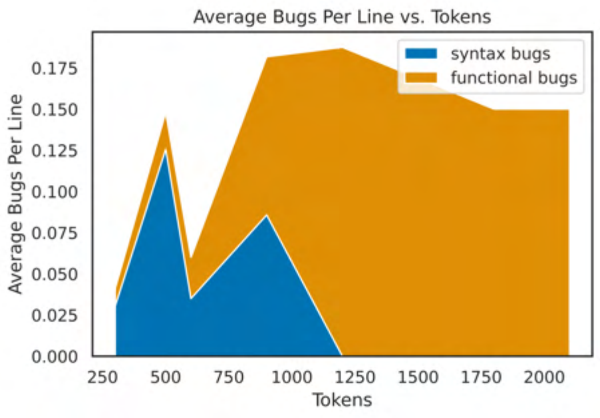


Figure 8: **Bugs in Codex-generated Code for Different Token Limits.** As the token length of the prompt increases, the amount of functional bugs tend to increase while the number of syntax bugs tend to decrease. The token number at which this tradeoff is optimized seems to be around 600.

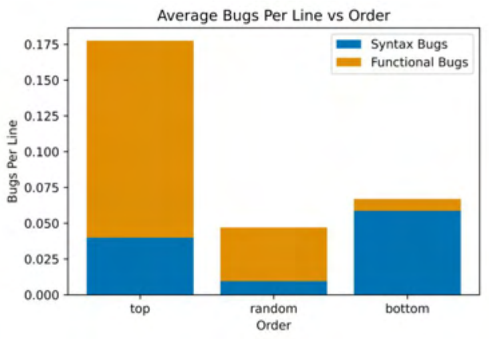


Figure 9: **Bugs in Codex-generated Code for Each Order of Examples.** The pipeline with the bottom ordering has the least amount of functional bugs. The pipeline with the random ordering has the least syntax bugs.

4 Discussion

We discovered several interesting trends within our examination of prompt engineering. First, we find that for complex apps, MRMR and Embeddings perform the best, indicating that the relevance of examples to the user task is important when Codex must execute a more complicated task. The reverse is true; for simple apps, simply sending in as many examples as possible performs well compared to MRMR and Embeddings. Similarly, quality of examples rather than quantity appears to be more important for complex apps: performance marginally decreases then increases with token size as shown in **Figure 5**. Bottom ordering appears to perform marginally better than Top and Random for simple apps but not complicated ones.

The manual evaluation reveals an interesting trend: while the mrmr solutions have no Syntax bugs, they have more functional bugs than embeddings. Moreover, while increasing the tokens decreases the syntax bugs, the functional bugs increase. Both of these results indicate a tradeoff between sending new information (more tokens or more lesser-correlated examples from MRMR) and making sure that Codex stays on task.

5 Future Work

There are some potential changes to the automated prompt contruction pipeline that were not examined within this study. One of these was tuning the hyperparameters 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 [3]. Additionally, the performance of other large language models such as Meta AI's InCoder in generating apps can be compared to Codex's performance [15]. Finally, to increase the robustness of the pipeline and the evaluation of it, the dataset of App Inventor examples and test App tasks can be further expanded.

6 Conclusion

We examined three different characteristics of choosing few-shot examples for Codex to learn from given an user app description: the number of tokens in the prompt, how examples are chosen, and how the examples are ordered within the prompt. We improve the pipeline's performance from baseline on complex apps by 55% (0.10) BLEU score in changing the method of selecting examples and 43% (0.13) in simple apps from worst-performing number of tokens to best-performing number of tokens. We also find a significant difference in how to optimize the platform for complicated apps and for simple ones. We hope that our work will lead the way for similar studies of prompt engineering for Codex.

7 Practical Takeaways

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. Our work has many applications in the democratization of app creation, whether that be in enabling even children to realize their app ideas or otherwise broadening who can be part of the technological community. 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 app without the barriers of time and technical expertise.

8 Acknowledgments

I'd like to thank my mentor David Kim for proposing my project's direction and giving daily guidance for the past month on my project and on research in general. I'd also like to thank the MIT App Inventor Lab for hosting my research for the Aptly platform and for being warm and welcoming. I'm grateful for my tutor Rachel Seevers, who was always available for questions and practice and gave valuable advice throughout my research project. I'm also grateful for the RSI TA's who helped me refine my paper and presentation. I thank my counselor Eli Meyers, RSI peers, and my parents for their support throughout my research experience. I also am thankful for the Center of Excellence in Education (CEE) and the Massachusetts Institute of Technology for making an in-person RSI possible. Finally, I'd like to express my gratitude to the Building Engineering and Science Talent Program for making this research possible.

References

[1] The future of it lies in democratising application development, May 2021.

[2] Z. Yan. The impacts of low/no-code development on digital transformation and software development. *CoRR*, abs/2112.14073, 2021.

[3] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[4] Y. Wang and Q. Yao. Few-shot learning: A survey. *CoRR*, abs/1904.05046, 2019.

[5] I. Drori, S. Zhang, R. Shuttleworth, L. Tang, A. Lu, E. Ke, K. Liu, L. Chen, S. Tran, N. Cheng, R. Wang, N. Singh, T. L. Patti, J. Lynch, A. Shporer, N. Verma, E. Wu, and G. Strang. A neural network solves, explains, and generates university math problems by program synthesis and few-shot learning at human level, 2021.

[6] L. Reynolds and K. McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. *CoRR*, abs/2102.07350, 2021.

[7] A. Neelakantan, T. Xu, R. Puri, A. Radford, J. M. Han, J. Tworek, Q. Yuan, N. Tezak, J. W. Kim, C. Hallacy, J. Heidecke, P. Shyam, B. Power, T. E. Nekoul, G. Sastry, G. Krueger, D. Schnurr, F. P. Such, K. Hsu, M. Thompson, T. Khan, T. Sherbakov, J. Jang, P. Welinder, and L. Weng. Text and code embeddings by contrastive pre-training. *CoRR*, abs/2201.10005, 2022.

[8] C. Ding and H. Peng. Minimum redundancy feature selection from microarray gene expression data. *J. Bioinform. Comput. Biol.*, 3(2):185–205, Apr. 2005.

[9] Z. Zhao, R. Anand, and M. Wang. Maximum relevance and minimum redundancy feature selection methods for a marketing machine learning platform, 2019.

[10] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2019.

[11] T. Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh. Calibrate before use: Improving few-shot performance of language models. *CoRR*, abs/2102.09690, 2021.

[12] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu. Bleu: a method for automatic evaluation of machine translation. 10 2002.

[13] S. Bird, E. Klein, and E. Loper. *Natural language processing with python.* O'Reilly Media, Sebastopol, CA, July 2009.

[14] T. J. McCabe. *Structured testing.* IEEE Computer Society Press, 1983.

[15] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W. Yih, L. Zettlemoyer, and M. Lewis. Incoder: A generative model for code infilling and synthesis. *CoRR*, abs/2204.05999, 2022.