CODEX HACKS HACKERRANK: BENEFITS AND RISKS OF LARGE-SCALE SOURCE CODE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

The Codex model has demonstrated extraordinary competence in synthesizing working code from natural language problem descriptions (Chen et al., 2021). However, in order to reveal unknown failure modes, and uncover hidden biases, such large-scale models must be systematically subjected to multiple evaluations. In this work, we evaluate the code synthesis capabilities of the Codex model based on a set of 115 Python problems from a popular competitive programming portal: HackerRank. Our evaluation shows that Codex is indeed *proficient* in Python—solving 96% of the problems in a zero-shot setting, and 100% of the problems in a few-shot setting. However, Codex shows signs of producing memorized code, which is alarming, especially since the adoption and use of such models directly impacts how code is written and produced in the foreseeable future. With this in mind, we further discuss and highlight some of the prominent benefits and risks associated with large-scale language models such as Codex.

1 INTRODUCTION

The overwhelming success of large-scale transformer-based models (Vaswani et al., 2017) in natural language processing (NLP), further powered by the self-supervised learning approach (Devlin et al., 2019), has led to a paradigm-shift in the way researchers and model engineers design and construct source code models. Thanks to the transformer architecture, several modern source code models such as *CodeBERT* (Feng et al., 2020), *GraphCodeBERT* (Guo et al., 2021), *PLBART* (Ahmad et al., 2021), *CodeT5* (Wang et al., 2021), and others have achieved state-of-the-art performance on a number of source code tasks including code completion, code summarization, and code translation.

Recently, Chen et al. (2021) introduced Codex, a source code model based on *GPT-3* with over 12 billion parameters. The capabilities of the model went beyond traditional code tasks, with applications not only in code completion, code summarization, etc., but also in building simple full-scale apps, and games such as Snake and Tetris. Since, Codex is a descendant of the OpenAI *GPT-3* model (Brown et al., 2020), it inherits much of the natural language understanding capability, which gives it the power to produce completions based on natural language prompts from the user. In fact, Codex is able to handle natural language prompts beyond just English, with the competence to process prompts in German, Japanese and many other languages.

The marked ingenuity of the Codex model is doubtlessly impressive with far-reaching applications, however, it must still be evaluated against established benchmarks for the range of source code tasks it promises to handle. Since several of the publicly available source code repositories on GitHub may contain code that is poorly written, insecure, or downright malicious[1], large-scale source code models depending on such input data may produce code that is harmful with unanticipated consequences, both for the model developers and the end-users. Furthermore, several types of biases that may exist in the training data could lead such models to produce biased outputs.

Thus, rigorous testing and auditing, possibly in multiple evaluation formats and stages, is necessary to understand the potential advantages and pitfalls — in a bid to ultimately making model predictions reliable and transparent, and to open up the black boxes that are large-scale source code models. The ever-increasing size of such models make it all the more necessary for them to be assessed, to rectify unintended faults, to identify unknown failure modes and to uncover hidden biases in the predictions.

[1]GitHub is known to contain malicious programs that can alter their environments (Rokon et al., 2020)

In this paper, we evaluate the Codex model by prompting a set of 115 well-defined Python problem statements defined on HackerRank[2] and assess whether the model is able to correctly synthesize code. Our evaluation shows that Codex is indeed able to produce valid code for 96% of the problems in a zero-shot setting, and 100% of the problems in a few-shot setting with varying temperatures.

However, Codex shows several signs of memorization. Instead of actually synthesizing code from the task objectives in the problem statement, Codex appears to generate memorized code as output, which matches the input-output specifications corresponding to the original HackerRank problem even when no specifications are provided. Even more surprising is that Codex produces the full working code for HackerRank problems when just the first sentence of the problem statement is used as a prompt—even if there is no clear task objective defined for program synthesis.

We discuss some of the related work next in Section 2. The relevant experimental considerations we present in Section 3 and the results of our evaluation in Section 4. In Section 5 we discuss the implications from our study, including the benefits and possible pitfalls of using such large-scale models vis-a-vis Codex. Finally, we discuss some future work and conclude our study in Section 6.

2 Related Work

Statistical modelling for source code has come a long way since it was first introduced, going from simple n-gram models (Hindle et al., 2012) to modern-day source code transformers (as surveryed by Allamanis et al. (2018); Allamanis (2022)). Currently, large-scale code models with billions of parameters, such as Codex (Chen et al., 2021), and recently announced AlphaCode (Li et al., 2022), have begun to establish their dominance in several code tasks, particularly code synthesis.

Code Synthesis. The idea of code synthesis from natural language dialogue is certainly not new. Ginsparg (1978) and Heidorn (1986) outline and survey automatic programming systems that can carry out natural language dialogue. At the turn of the millenium, further attempts at general-purpose code generation from natural language began to surface (Price et al., 2000; Vadas & Curran, 2005; Mihalcea et al., 2006). Gulwani (2011) contribute further to the field by presenting an algorithm that can synthesize short string-manipulating programs from input-output examples, and by surveying the state-of-the-art approaches to program synthesis (Gulwani et al., 2017).

With the advent of learning-based techniques, studies by Menon et al. (2013) and Parisotto et al. (2016) show how learning on input-output examples can be leveraged to automatically synthesize code. Yin & Neubig (2017) improve semantic parsing and achieve state-of-the-art results in code synthesis by utilizing the syntax information of the target language as prior knowledge. Work on code synthesis reached a new milestone when Codex was introduced by Chen et al. (2021) with the promise of generating *complete* snippets of code from clearly defined problem statements in natural language. While newer models such as AlphaCode (Li et al., 2022) promise to take program synthesis to greater heights, Austin et al. (2021) and Karmakar & Robbes (2021) have already begun exploring the limits of code synthesis and code understanding potential in large language models.

Evaluation of Codex. Pearce et al. (2022) evaluate the Codex model in order to identify the purpose and capabilities of code snippets, and identify important variable names or values from code, by prompting open-ended questions to the model. The authors develop a true/false quiz framework to characterize the performance of the Codex model. Prenner & Robbes (2021) evaluate Codex on the task of automated program repair; while Pearce et al. (2021) evaluate Codex and Jurassic J-1 models on their ability to repair insecure code in a zero-shot setting. The above studies broadly shed light on the question answering and code repair capabilities of the Codex model, while we evaluate the model specifically on the task of code synthesis.

Tang et al. (2021) utilize the code synthesis capabilities of Codex to solve university level problems in probability and statistics. Similarly, Drori & Verma (2021) solve algebra problems using Codex. The authors first take problems from MIT, Stanford, and Columbia University's courses, then convert them into suitable programming tasks, and then prompt the Codex model to generate code solutions. Drori et al. (2022) go further on to solve calculus and differential equations problems using the code synthesis capabilities of Codex. These studies evaluate Codex on code synthesis, similar to our approach, but their evaluation efforts remain limited to math problems.

[2] `https://hackerrank.com`

3 EXPERIMENTAL CONSIDERATIONS

In this paper we evaluate Codex (the largest code model currently available[3]) in a zero-shot setting on a single task of code synthesis. Since it was a zero-shot evaluation, no further fine-tuning was done in order to ascertain the raw predictive power of the Codex model. We chose to evaluate Codex on the code synthesis task since Codex was introduced as being highly capable at it—being able to produce valid snippets of code from just natural language prompts. Furthermore, since Chen et al. (2021) highlighted in their paper that Codex was the most competent in Python, we chose to evaluate Codex on the code synthesis task for the Python language.

We received private beta access to Codex, which allowed us to run our evaluations. Codex is released in two formats: Codex-davinci *(now available as* `code-davinci-001`*)* and Codex-cushman *(now available as* `code-cushman-001`*)*. By default, we evaluate the Codex-davinci model, which is the larger and more capable Codex model particularly competent in code synthesis.

While evaluating Codex-davinci on the code synthesis task, the default settings were used: with the Temperature set to 0, the Top-P set to 1, the frequency and presence penalty set to 0, and taking only the best of 1 completions. All completions were done with an initial response length of 128 tokens, and subsequent completions were continued till no new tokens were produced. The settings remain the same for all, unless specified otherwise for select problems.

We used the problems defined on a popular competitive programming platform, HackerRank, to prompt the Codex model for solutions. HackerRank provides a range of well-defined problems with conceptual explanations, examples, and even input-output specifications, designed to test the Python proficiency of users. We extract the problem statement from these problems, and if necessary even the input-output specifications and examples, to formulate our prompts and then evaluate the subsequent solutions synthesized by the Codex model.

The prompts are presented to the Codex model as docstrings, and the model automatically detects the language of choice and make its predictions in the Python language. The prompts presented are straightforward with clear task objectives and input-output specifications, avoiding additional definitions, explanations, or tutorials wherever possible. We retrieve the synthesized Python code and submit it to HackerRank which runs its test cases to accept or reject the code solution. If all the test cases pass, the solution is considered correct for the problem statement.

Fig. 1 shows an example of a prompt and its corresponding code output from the Codex model. The docstring highlighted in yellow represents the prompt, while the code highlighted in light blue represents the output. The synthesized code output in blue is then validated against the test cases for the problem statement on the HackerRank platform to assess its correctness. Some interesting examples of such prompt-output pairs from our evaluations on Codex are presented in Appendix A.

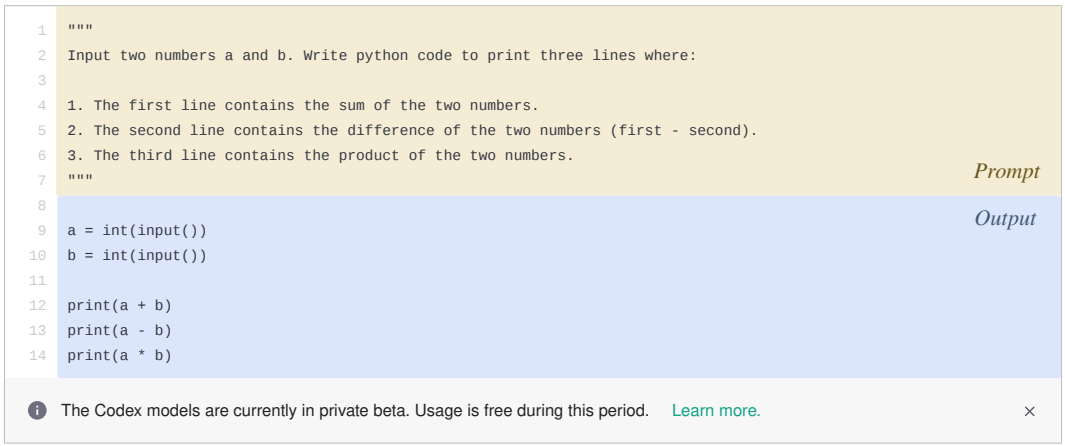


Figure 1: Example of a natural language prompt and the code output from Codex.

[3] AlphaCode (Li et al., 2022), announced at the time of writing, is still not released for evaluation

4 RESULTS

Our evaluation shows that Codex is indeed capable of resolving a class of code synthesis problems; specifically from HackerRank's list of problems that are used to determine Python proficiency in human participants. The problem statements prepared for this evaluation and their corresponding code solutions generated by Codex are made available online as runnable scripts.

Out of 115 code synthesis problems, Codex correctly generates solutions to 111 of them in a zero-shot setting, and to all 115 of them in a few-shot setting *($\leq$ 3)*, with a success rate of 96% and 100% respectively. However, there are some serious caveats. Most prominently: Codex seems to be parroting memorized code from Github instead of actually synthesizing the solution from the explicit cues in the problem statement. This is reflected in several situations as detailed below.

Failing Variants. When the desired output of a given problem statement from HackerRank is modified, the model still generates code reflecting the solution to the original and unmodified problem statement. In other words, given the same context information, but modifying the task objective of the problem statement, the Codex model produces the same output corresponding to the actual unmodified original problem statement.

For example, the problem statement numbered #32 asks for the *sum of the elements in set A* to be printed as output. Upon prompting the Codex model with the problem statement #32, it presents the correct code solution which prints the sum of the elements in set A as instructed.

However, when we intentionally modify the problem statement to another variant (see Fig. 3 in the appendix), where the *product of the elements in set A* is now asked to be printed, the model fails. Upon prompting the model with the modified variant of the problem statement, we observe that the model still presents the same code solution from the original unmodified problem statement. This suggests that the code output to this problem was most likely learned from several appearances of the stated problem in the training data and prompting a modified variant triggers the model to output the memorized code.

Codex also fails to predict the correct code solutions on several other variants of problem statements, always producing code solutions corresponding to the original unmodified HackerRank problems. In fact, for 18 out of 20 instances that we have tested, Codex failed to produce accurate and reciprocal code on alternate versions of the problem statements—each with distinct task objectives.

This raises the question whether similarly worded problems but with different objectives and requirements might trigger Codex to generate memorized code snippets, instead of actually synthesizing the code from the explicit task objectives. In that case, non-technical and novice users must use the model with caution, while application developers building on top of Codex must find ways to validate the synthesized code, or rely on techniques such as TST or CSD (Poesia et al., 2022). As for the model itself, such memorization is undesirable since the model only regurgitates what is has memorized, rather than basing the solution from the vital clues in the problem statement.

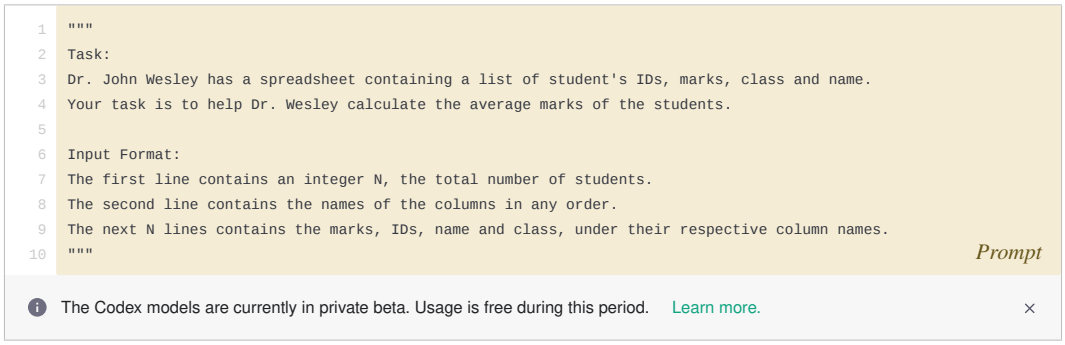
Missing specifications. For some problems where user input is necessary, the problem statements specify clearly how the input will be provided e.g., a single line of input is provided with a space-separated string *S* and an integer *k*. Alternatively, the inputs could have also been provided in another manner, e.g., the input is provided in two separate lines, the first line of input is a string *S*, the second line of input is an integer *k*. We note that the specification of how the input in provided is not vital to problem solving, but it sure defines how code is generated when it comes to code synthesis.

Out of 115 problem statements, we find that 9 of them do not have any input-output specifications, and/or constraints, or examples included with the problem statements. For additional 6 of the problem statements the prompt is straightforward and the model does not necessarily need input-output specifications or additional information (to synthesize code) even if they are explicitly provided. We exclude these 15 problems in our evaluation since the specifications either do not exist or are not strictly necessary, in order to better determine whether Codex can actually synthesize correctly-formatted and valid code even when necessary input-output specifications are not provided.

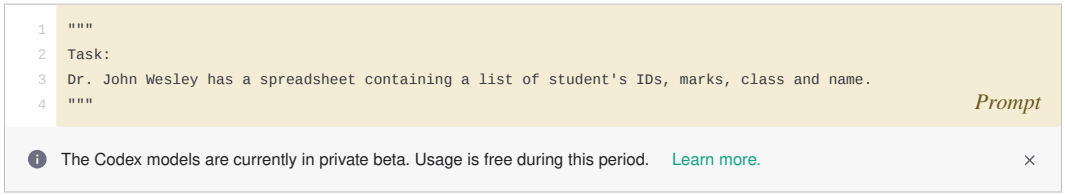
Of the remaining 100 problems, when we prompted Codex with just the problem statements without any input-output specifications, the code solution generated matched the specifications mentioned in the HackerRank problem statement for 84% of the cases. Even for problems where the output

must be structured in a certain way, the Codex model (being unaware of the explicit specifications based on the shortened prompts provided) produced code snippets that matched the required output conditions as specified in the original problem statements. Without the knowledge of how the input has to be read from user and how the outputs must be structured, the model seemed to be generating learned code it has *seen* rather than actually synthesizing it.

Missing objectives. In some cases, just the starting text is enough for Codex to produce solutions corresponding the actual HackerRank problem even when no clear task objective is present. Consider the problem statement numbered #57, the full problem statement of which is shown in Fig. 2a, and a trimmed version of the same problem statement is shown in Fig. 2b that has no task objectives.



(a) The natural language prompt for the problem statement #57.



(b) The trimmed prompt without objectives for the problem statement #57.

Figure 2: Original prompt vs. Prompt that has no clear task objectives or input-output specifications.

For the trimmed version of the problem statement, even when no clear task objectives are given, the Codex model predicts a valid code snippet corresponding to the full problem statement as defined on the HackerRank platform with matching input specifications and matching task objective.

This is undesirable, since the trimmed prompt to the model just includes the first line of the problem: *"Dr. John Wesley has a spreadsheet containing a list of student IDs, marks, class, and name"*. And in response to this prompt, Codex produces the correct solution to the existing problem statement defined on HackerRank, passing all test cases and even adhering to the relevant input specifications. Furthermore, such behavior does not appear to be mere isolated incidents either—with $\sim$38% of the cases generating apparent memorized code from just the first sentence of the problem statement, even if there is no task objective provided within the prompt text.

We found that out of 115 problem statements, 27 problem statements have some sort of program objective mentioned in the very first sentence, such as, #06 Given a year, determine whether it is a leap year; #41 You are given a complex z, convert it into polar coordinates; #96 You are given a valid xml, and you have to print the maximum level of nesting. For most of these 27 problem statements additional input-output specifications, or explanations are required to actually solve the problem. But, if one gives Codex the benefit of the doubt, one would assume that Codex can synthesize code in a fair manner just from the program objective mentioned in the very first sentence.

Then, if we set aside these 27 problems, out of the remaining 88 problems where no program objective is present in the first sentence, 33 instances have been recorded where Codex produces the full and valid HackerRank solutions to the problems just from the first sentence, including matching input-output specifications, stub code, and comments ($\sim$38% of the problems).

5 IMPLICATIONS

Before extensive adoption and use of large-scale code models, a systematic evaluation of these systems is necessary, most importantly in order to apprehend unanticipated consequences and unknown failure modes; and to determine the derived benefits and risks. We discuss some of the benfits and risks associated with using the Codex model in this section. And although the overall benefits of Codex appear to far outweigh the risks, from a technical perspective, the risks are significant.

5.1 BENEFITS

The conventional capabilities of the Codex model have already been well-documented by Chen et al. (2021). Furthermore, independent researchers, enthusiasts, and OpenAI community members have also tested and played with the model to devise a range of use-cases and applications from making memes, to generating artwork and animations, to building full-fledged applications and games (Alexeev, 2021). In addition, Codex being a descendant of the GPT-3 model also inherits abilities in natural language processing which allows it to perform other related tasks such as code summarization, code explanation, code translation, code repair, and so forth.

5.1.1 DEVELOPMENT

Code generation. Large language models (LLMs) like Codex have the potential to improve productivity of developers by generating code just from natural language commands. Developers could potentially rely on Codex on a number of applications, from automatically implementing small functions, to implementing well-known algorithms, to even generating code for complete applications.

Code search. During the development process, it is quite common for developers to consult API or tool documentation, public forums, and Q&A websites to retrieve necessary information. In such cases, finding the correct information can be rather difficult owing to several factors, such as poor documentation, inactive threads, unrelated solutions on forums etc. Several studies including Ponzanelli et al. (2014) address this very issue. But now, with the introduction of Codex, users may directly generate code from simple natural language commands, instead of spending time formulating explicit search-queries and browsing through several portals to obtain clues to the solution.

5.1.2 MAINTENANCE

Code migration. Migrating legacy systems written in older languages is a tedious process to say the least. However, the dangers from unexpected behaviour resulting in unknown hazards, overt dependence on proprietary systems, limited portability, etc. call for the migration of these systems to modern languages such as Java or Python (Veerman & Verhoeven, 2006; Sneed & Erdoes, 2013). Migrating billions of lines of operational and revenue-producing legacy code may be a huge task to undertake, but early experiments (Mark, 2021) already present signs that large-scale models of source code can help translating smaller chunks of legacy code to modern languages.

Code explanation. Owing to its natural language understanding, the Codex model could also serve as a coach to novice users helping them understand computer science concepts, technical issues, and error messages in plain English. A popular browser-IDE, now powered by Codex, demonstrates how the model's reasoning ability can help answer code-related questions, and give explanations of what a piece of code actually does (Webster, 2021).

Code repair. The Codex model can be directly prompted with natural language commands to repair buggy code (Prenner & Robbes, 2021), optimize bloated code, or even rewrite insecure code (Pearce et al., 2021). In this modality, developers could write a snippet of code and simply direct the Codex model in natural language to check for bugs, to optimize the code, or to make it secure.

5.1.3 OTHER APPLICATIONS

Beyond applications directly related to source code and Computer Science, Codex has also been used to solve university-level problems in probability, statistics, algebra, calculus, differential equations, etc., and even to generate new problems and course content for these topics (Drori & Verma, 2021; Drori et al., 2022). This suggests that Codex can have a significant impact in several fields such

as in higher education—first, by enabling practitioners solve complex problems that depend on mathematical reasoning with just natural language commands, and second, by enabling educators generate and grade new questions and problems. Another study by Hocky & White (2022) proves the point further by demonstrating that Codex can have a significant positive impact on chemistry and chemical engineering research. The benefits rooting from the positive use of the model are remarkable, yet, unfortunately there are certain risks involved as well.

5.2 RISKS

Bender et al. (2021) take note of the ever-increasing size of natural language models in NLP and discuss the possible risks associated with them, including environmental risks and social biases. However, based on our evaluation of Codex, we limit our discussion solely to the technical aspects and highlight some of the relevant risks and limitations associated with it.

5.2.1 LIMITED CREATIVITY

It is important to highlight here that since the Codex model has been subjected to enormous amounts of code, which may include several implementations of classical computer science algorithms, common student problems, repetitive server-side code, etc., often it may generate commonly-used or *seen* code for even distinctly unique problem statements—producing unreliable predictions as a result of rote-learning. An obvious example is: a unique custom pattern-based problem statement generates a wrong code solution for a standard pattern, perhaps memorized from the training data. This suggests that solving distinctly unique problems could pose as a challenge for Codex. Since solving real-life problems would indeed require creative problem solving rather than depending on learned code, Codex must be evaluated further on this.

In such cases, Codex appears to be working more like a code retrieval engine rather than one for code synthesis. Section 4 sheds further light on such instances. Perhaps it's too early to expect models to *"create"* code for a new and unseen problems, but given the promise of code synthesis, we must conclude that there is ample room for improvement regarding the discernment of natural language prompts to accurately synthesize code.

5.2.2 OBSCURE EVALUATION SCOPE

Since a lot of the prevailing code datasets have been gathered from open-source code repositories available on platforms like GitHub, it is likely that evaluating large-scale code models on these datasets can result in inaccurate or biased evaluation outcomes. Considering the fact that Codex is able to memorize code, a prompt to fix a buggy snippet of code can surely result in the corrected code, especially when the snippet of code in question has already been seen and trained upon, perhaps several times over. On the surface this might indicate that the Codex model is capable of tasks such as program repair, but in reality, the snippets might already be known to Codex. A prompt to fix an unseen snippet of code, on the other hand, might fail altogether.

Therefore, going forth, researchers must be careful in evaluating the large-scale source code models with datasets derived from GitHub to avoid biased evaluation results, especially datasets containing commonly-used or well-known code snippets or algorithm implementations, which could already have been subjected to such models during training.

5.2.3 PRIVACY RISK

Even though the Codex model is trained on publicly available open-source code, user code may often contain "sensitive" information such as API tokens, secret keys, or even passwords (Sinha et al., 2015). Although the onus is upon the users to keep their private information *private*, instances of secret key leaks is not uncommon. Courtesy to its memorization, similarly worded prompts could trigger Codex to involuntarily reveal sensitive information while performing code synthesis or completion tasks.

Furthermore, since Codex can be adapted to any number of downstream systems, the consequences rooting from the aforementioned risks might emerge only at the end-user, effectively hidden from model architects, engineers, and intermediate developers—thus aggravating the risk. Therefore, such models should be subject to rigorous testing and auditing, before they are released in the wild.

Table 1: Overview of evaluations carried out across prompt types and summary of results.

Prompt Types	**Total**	**Tested**	**Passed**	**Pass%**
Problems w/ Complete Details	115	115	115	100%
Problems w/ Missing Specifications	115	100	84	84%
Problems w/ Missing Objectives	115	88	33	38%
Problems w/ Variant Objectives	115	20	02	10%

6 Discussion & Conclusion

Table 1 gives an overview of evaluations carried out across prompt types and the summary of results. From our analysis, we observe that Codex seems to be generating memorized code, rather than actually synthesizing the code from the distinct task objectives present in the prompts. One could infer that this behaviour occurs because of the examples that the model has seen in the training data, as we find several hundreds of repositories on Github with the HackerRank solutions for Python. Unfortunately, the training data for the Codex model is not publicly available, therefore, potential mitigating factors for such behaviour is more *difficult* to establish.

We prompt the Codex model with 115 well-defined Python problems from HackerRank and find that even though it is able to generate correct solutions to 100% of the problems, it fails to *synthesize* code, in its true sense, from the distinct natural language task objectives in the problem statements, as evident from the examples mentioned in Section 4. This suggests that there is indeed room for improvement for the Codex model: to minimize dependence on memorized code, and to improve the semantic discernment of natural language intent. We further outline the harms and risks posed by memorization in large-scale source code models, and encourage future researchers to be mindful of them while reflecting on novel training paradigms beyond building increasingly larger models.

As future work, we look forward to creating new and unique handcrafted problems, to address memorization issues rooting from data leakage; we intend to build multiple novel datasets, and conduct evaluations on a range of tasks based on such problems that are unlikely to be present in the training data. Furthermore, we intend to broaden the evaluation scope itself beyond direct task-based evaluations, incorporating evaluations on intrinsic learning of source code structure, syntax and semantics. We consider that such an extended scope of evaluation is necessary since the use and adoption of such models as foundation models (Bommasani et al., 2021) can have far-reaching impact on the way future code is produced, maintained, and used in downstream applications.

Availability: All HackerRank problem statements and Codex responses are recorded and made available online at: `https://github.com/ms-anon/codex_hx_hackerrank/`

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Vladimir Alexeev. Codex by openai, in action. https://towardsdatascience.com/codex-by-openai-in-action-83529c0076cc, 2021. Accessed: 2022-02-02.

Miltiadis Allamanis. A survey of machine learning on source code. https://ml4code.github.io/, 2022. Accessed: 2022-02-10.

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018. ISSN 0360-0300.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM*

Conference on Fairness, Accountability, and Transparency, FAccT '21, pp. 610–623, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383097.

Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dora Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zaharia, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. 2021.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2019.

Iddo Drori and Nakul Verma. Solving linear algebra by program synthesis. *arXiv preprint arXiv:2111.08171*, 2021.

Iddo Drori, Sunny Tran, Roman Wang, Newman Cheng, Kevin Liu, Leonard Tang, Elizabeth Ke, Nikhil Singh, Taylor L. Patti, Jayson Lynch, Avi Shporer, Nakul Verma, Eugene Wu, and Gilbert Strang. A neural network solves and generates mathematics problems by program synthesis: Calculus, differential equations, linear algebra, and more, 2022.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. 2020.

Jerrold M Ginsparg. Natural language processing in an automatic programming domain. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1978.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. 2021.

G. Heidorn. *Automatic Programming through Natural Language Dialogue: A Survey*, pp. 203–214. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986. ISBN 0934613125.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pp. 837–847. IEEE Press, 2012. ISBN 9781467310673.

Glen M. Hocky and Andrew D. White. Natural language processing models that automate programming will transform chemistry research and teaching. 2022.

Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1332–1336. IEEE, 2021.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. 2022. Accessed: 2022-02-02.

Ryan Mark. Back to the future with codex and cobol. https://towardsdatascience.com/back-to-the-future-with-codex-and-cobol-766782f5ae8f, 2021. Accessed: 2022-02-02.

Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Tauman Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pp. I–187–I–195. JMLR.org, 2013.

Rada Mihalcea, Hugo Liu, and Henry Lieberman. Nlp (natural language processing) for nlp (natural language programming). pp. 319–330, 02 2006. ISBN 978-3-540-32205-4. doi: 10.1007/11671299_34.

Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. Can openai codex and other large language models help us fix security bugs? *arXiv preprint arXiv:2112.02125*, 2021.

Hammond Pearce, Benjamin Tan, Prashanth Krishnamurthy, Farshad Khorrami, Ramesh Karri, and Brendan Dolan-Gavitt. Pop quiz! can a large language model help with reverse engineering? *arXiv preprint arXiv:2202.01142*, 2022.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*, 2022.

Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 102–111, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630.

Julian Aron Prenner and Romain Robbes. Automatic program repair with openai's codex: Evaluating quixbugs. *arXiv preprint arXiv:2111.03922*, 2021.

David Price, Ellen Riloff, Joseph Zachary, and On Harvey. Naturaljava: A natural language interface for programming in java. 02 2000. doi: 10.1145/325737.325845.

Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Vagelis E. Papalexakis, and Michalis Faloutsos. Sourcefinder: Finding malware source-code from publicly available repositories, 2020.

Vibha Singhal Sinha, Diptikalyan Saha, Pankaj Dhoolia, Rohan Padhye, and Senthil Mani. Detecting and mitigating secret-key leaks in source code repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 396–400, 2015. doi: 10.1109/MSR.2015.48.

Harry M. Sneed and Katalin Erdoes. Migrating as400-cobol to java: A report from the field. In *2013 17th European Conference on Software Maintenance and Reengineering*, pp. 231–240, 2013. doi: 10.1109/CSMR.2013.32.

Leonard Tang, Elizabeth Ke, Nikhil Singh, Nakul Verma, and Iddo Drori. Solving probability and statistics problems by program synthesis. *arXiv preprint arXiv:2111.08267*, 2021.

David Vadas and James R. Curran. Programming with unrestricted natural language. In *Proceedings of the Australasian Language Technology Workshop 2005*, pp. 191–199, Sydney, Australia, December 2005.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017.

Niels Veerman and Ernst-Jan Verhoeven. Cobol minefield detection. *Softw. Pract. Exper.*, 36(14): 1605–1642, nov 2006. ISSN 0038-0644.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

Barron Webster. Replit + codex - beta release. https://blog.replit.com/codex, 2021. Accessed: 2022-02-02.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*, 2017.

A APPENDIX

A.1 FAILING VARIANTS

Prompt

```
"""
Task:
You have a non-empty set s, and you have to execute N commands given in N lines.
The commands will be pop, remove and discard.

Input Format:
The first line contains integer n, the number of elements in the set s.
The second line contains n space separated elements of set s. All of the elements are non-
negative integers, less than or equal to 9.
The third line contains integer N, the number of commands.
The next N lines contains either pop, remove and/or discard commands followed by
their associated value.

Constraints:
0 < n < 20
0 < N <20

Output Format:
Print the product of the elements of set  on a single line.
"""
```

Output

```

n = int(input())
s = set(map(int, input().split()))
N = int(input())

for i in range(N):
    command = input().split()
    if command[0] == 'pop':
        s.pop()
    elif command[0] == 'remove':
        s.remove(int(command[1]))
    elif command[0] == 'discard':
        s.discard(int(command[1]))

print(sum(s))
```


The Codex models are currently in private beta. Usage is free during this period. Learn more. ×

Figure 3: The incorrect code solution produced by Codex for the modified variant of problem #32.

A.2 MISSING SPECIFICATIONS

```
"""
Ms. Gabriel Williams is a botany professor at District College.
One day, she asked her student Mickey to compute the average of all the plants with distinct heights in her greenhouse.
Calculate the average of distinct values in array.

Function Description:
Create an average function.
The average function has the following parameters:
- int arr: an array of integers

Returns:
- float: the resulting float value rounded to 3 places after the decimal

Input Format:
- The first line contains the integer, N, the size of arr.
- The second line contains the N space-separated integers, arr[i].
"""
```

Prompt

Figure 4: Problem statement #28 with a function description and input-output specifications.

```
"""
Ms. Gabriel Williams is a botany professor at District College.
One day, she asked her student Mickey to compute the average of all the plants with distinct heights in her greenhouse.
Calculate the average of distinct values in array.
"""
```

Prompt

Output

```
def average(arr):
    return round(sum(set(arr))/len(set(arr)), 3)

if __name__ == '__main__':
    n = int(input())
    arr = list(map(int, input().split()))
    result = average(arr)
    print(result)
```


Figure 5: The correctly-formatted working code solution produced by Codex when prompted with the shortened problem statement #28 without the function description & input-output specifications.

A.3 MISSING OBJECTIVES

```
"""
Task:
Dr. John Wesley has a spreadsheet containing a list of student's IDs, marks, class and name.
Your task is to help Dr. Wesley calculate the average marks of the students.

Input Format:
The first line contains an integer N, the total number of students.
The second line contains the names of the columns in any order.
The next N lines contains the marks, IDs, name and class, under their respective
column names.
"""
```

Prompt

Figure 6: Problem statement #57 with a clear task objective and input-output specifications.

```
"""
Task:
Dr. John Wesley has a spreadsheet containing a list of student's IDs, marks, class and name.
"""
```

Prompt

```
from collections import namedtuple

if __name__ == '__main__':
    n = int(input())
    student = namedtuple('student', input())
    print(sum([int(student(*input().split()).MARKS) for _ in range(n)]) / n)
```

Output

Figure 7: The correctly-formatted working code produced by Codex when prompted with just the first sentence of the problem statement #57 which does not contain any form of task objective.