Can Large Language Models Generated Code That Can Compete With Model, Human-Written Code

Thomas D. Chambers
tc262389@falmouth.ac.uk
Falmouth University
Penryn, Cornwall, United Kingdom
https://github.com/ThomasChambers15243/Dissertation

ABSTRACT

Large language Models have never been more prominent than they are today, finding use in social media, essay writing, shopper predictions, investing and now software development. This paper will outline the history of code generation, LLM, and code analysis and will then attempt to investigate the practicality of LLM-generating code for software developers to use. The paper will finish off with some discussion about ethical and practical issues with code generation in general.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

ACM Reference Format:
Thomas D. Chambers. 2023. Can Large Language Models Generated Code That Can Compete With Model, Human-Written Code. In *Proceedings of* . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This study will investigate the practicality of using large-language models to generate Python code. These models take in a human-written prompt to generate an output, the output attempting to fulfil the desired task of the prompt through code. Within the last few years, these models have been used to generate code in many programming languages, straight into a programmer's IDE. With many large tech companies investing in generative models, such as Google and META [20, 28], and bespoke versions of these models being used commercially for code productivity, such as Github's Copilot and Sourcery [22, 25], their use by programmers is rapidly increasing. LLM are often described as a black box [3], this describes how the user can input data and receive a response, without an understanding of the internals. This form of abstraction is very common within programming, however, where LLM stand out is that the engineers behind them also do not know. Because the internal 'synapses' of the neural network are automatically arranged by the model itself, it's extremely difficult for an engineer to decipher. Due to this black box behaviour, an intimate understanding of how these code snippets are generated is unknown, and there's a risk that poorly written, buggy and insecure code could be used by novice and experienced programmers alike. The need to understand the level of quality of outputs is paramount before these generation tools become widespread in education and industry, so the consequences can be best understood [1]. The focus of this study will be on Open-AI's GPT models due to their API, success and public prominence.

2 UNDERSTANDING CODE GENERATION

Code generation is not new, being used in many computing processes already, perhaps most well-known is the use in compilers to produce code optimisation and executable machine code. Furthermore, code generation from a human prompt has been discussed for over 50 years. PROW [35] was an early code generation tool that could produce Perl code from an input written in predicate calculus. However, as discussed two years later by Manna and Waldinger [18], "programmers might find such a language lacking in readability and conciseness." They went on to predict a future program synthesizer; one that works with the programmer to produce segments of code that can be incorporated into the program. They argue it would be a "more practical system" with greater "power".

Fifty years later in 2022, with the public release of ChatGPT [24], public awareness of LLM significantly increased due to the wide adaption of its chat feature. Generation models have been used for years to write essays, generate art and synthesise an actor's voice, the implications of such have caused many to re-evaluate the current state and future of educational and business environments [7, 10, 16, 37], but more recently the use of generation has become more relevant in a programmer's workflow.

2.1 The Background of LLM

The rising popularity of LLM generation is in-large due to their apparent success at task completion, giving them the appearance of being *intelligent*. However, by any definition today, at most it's pseudo-intelligence. This section will first give a very brief overview of the workings of LLM, why they're pseudo-intellectual and then their current use today.

Large language models are neural networks trained to predict the next token in a sequence, the token, in the case of code generation, being characters in a string.

$$P(Token_n | Token_1, Token_2, ..., Token_{n-1})$$

GPT stands for **G**enerative **P**re-trained **T**ransformer. The generation means that it can predict the next token, pre-trained means that the model has received a large amount of data and has had its bias adjusted and a transformer is an encoder-decoder network that adds *self-attention*. Self-attention results in significant performance increases in accurate prediction. LLM that have been pre-trained can be further *fine-tuned* to improve performance at specific predictions. Although the prediction can be extremely accurate, there is no logical reasoning behind the prediction that can be seen as akin to human intelligence. The accurate predictions serve to mimic human intelligence.

In 2021 OpenAI released their fine-tuned model GPT-3 codex, trained on 54 million public software repositories hosted on GitHub [4]. Released the same year [22], Github's CoPilot is a generation model with a heavy reliance on Codex. To test the functional performance of Codex, the team behind OpenAI released the humanEval data set. A public repository to benchmark the performance of generated code. They also implemented the technique *pass@k*, the formula for which can be seen in the appendix section .3. An unbiased calculator to estimate the success of a model given *k* samples. Functional performance is found by the ability to generate code from a prompt that successfully passes given unit tests, failures often due to syntax errors, invoking out-of-scope functions or referencing non-existent variables. Within their data set, Codex had a higher performance than all previous GPT-model, solving 70.2% of problems with 100 samples. The functional performance of Codex and models alike have been replicated numerously [26, 29, 38, 40], with Codex being able to outperform students in even CS1 questions and, although performance dropped, still compete with students in CS2 questions [9]. CS1 questions covered the basics of programming, with topics such as "variables, arithmetic, conditional branches, loops, reading and writing to files, and functions". CS2 took questions further, covering questions about hashing, discrete mathematics, OOP and ADTs. During these questions, the model could recognise algorithms by name and produce efficient solutions for those problems, such as tree or graph searches and modifications.

2.2 Issues with the Current State of Code Generation

Large language models have proven to appear highly performant, however, several technical and practical limitations should be carefully understood.

Firstly, models are not uniform across languages. Nguyen and Nadi [26] found that CoPilot functionally performed best when writing in Java and worst in Javascript. Other researchers have written their own fine-tuned model, Poly-Coder, based on the GPT-2 architecture, which outperforms all GPT-based models in the C language [40]. Its was argued that the low score in C was possibly due to a problem with the data-set and fine-tuning process being over-reliant on Python and under focused in C.

Secondly, the results are not *one-shot*. Codex can respond with very variant accuracy in its solutions, hence why most investigations use several responses, such as with *pass@k* often using 100 samples. This is not the standard programming experience; a student does not submit 100 solutions to be marked by their professor, leading to the comparison being fallacious. An attempt to control this compared students to Codex, except also gave students multiple attempts at solutions, their success frequently increases with each attempt. However, they found that Codex was still competitive with students, even after multiple submissions [9].

Thirdly, responses vary significantly depending on the value of the model's temperature. Temperature is a value used in neural networks to increase entropy (effectively randomness) in the softmax distribution, the distribution which controls the quality and diversity of the predictions. This affects the output layer where tokens are sampled from. A higher temperature increases how 'surprising' the next token will be [12, 36]. Multiple studies have found that with a higher temperature, models can achieve a higher *pass@k* score at larger samples, despite producing more erroneous code. Across multiple studies, researchers found optimal performance at T0.6, with T0.2 and T0.8 also performing well. Temperature and sample size were found to be proportional, this is likely due to the diversity of code at higher temperatures needing a higher sample rate to score [4, 27, 40].

Fourthly, there is a significant reason to question whether the success of generated code translates into actual programmer performance, even with their access becoming more streamlined. CoPilot can directly embed into your IDE, finishing off lines or blocks of code for the user. There is also a chat option in VS Code where prompts can be given and code can be directly copied into the user's file, currently in beta. However, when a programmer uses a piece of code, they have to evaluate the code before use as it might contain bugs, be a sub-optimal solution or generally poorly written. In a study of 24 participants using Copilot [34], they found that even though code generation gave promising results, it did not improve overall programming time or the success rate of those participants using the tool. CoPilot even led to more task failures in the medium and hard category of tasks, where programmers might have not understood the generated code or spent longer debugging the code than if they had just written it themselves. Nevertheless, 23/24 of the participants still found it more useful than Intellisense and the majority "overwhelmingly preferred using Copilot in their programming workflow since Copilot often provided a good starting point to approach the programming task." The study showed CoPilot to be an imperfect aid. It can allow a programmer to instantly generate an often feasible solution to a task, however, in doing so they remove themselves from the task of problem-solving, which often exposes the programmer to online discussion and related topics, advancing their technical skill. Other researchers have discussed this problem, hypothesising that code generation is an efficient tool for seasoned programmers, but can turn into a liability when used by novice programmers who do not fully understand the problem, context and generated solution [23].

3 QUALITY CODE

Good quality code is vital for creating maintainable software that can last, but still, there is discourse around what good quality code is. In 1969 Dijkstra wrote in a letter, *"A programmer has only done a decent job when his program is flawless and not when his program is functioning properly only most of the time."* He criticized the attitude of programmers of his time, something that he saw as a *"software crisis"*. He saw programmers treating debugging as a necessity, rather than what he believed, an inevitable consequence of poorly written systems. He argued that writing *"intellectually manageable"* *programs* would reduce the amount of reasoning involved in justifying their proper operation and a reduction in the number of test cases. If done correctly, he claims that *"the correctness can be shown a priori"*, so a need for zero test cases. Dijkstra laid out a fundamental argument for why quality code is necessary which has been built upon ever since [6].

Recognising when code is *"intellectually manageable"* is a complex issue. Greg Michaelson wrote that *"Programming style is notoriously difficult to characterise"* and that imperative languages have been the *"source of endless theological disputes"*, such as the use of GOTOS, the use of recursion over iteration or the number of parameters a sub-program should have [21]. While for the most part, experts agree on the abandonment of goto's in structured programming [5, 15], the rest are still up for debate.

The quality of generated code is vital if the produced code is ever expected to be used in a practical sense. However, the common mantra, *garbage in, garbage out* is just as true now, the quality of written code largely depends on the quality of the code within the data set. Unfortunately, the fine details of the data sets used for available LLM are kept private, [40], so to assess the quality of code, we can use code metrics as an attempt to judge the outputted code.

3.1 Code Analysis

Pre-1980, software metrics generally worked as a regression model between two variables, conceptually simple, mostly relying on resources and quality. However, during the mid-late 70s, software metrics saw a new direction with Halstead and McCabe [8]. They extracted information from the code design rather than just the static code.

Halstead developed a set of formulas that, when given a code input, would produce a series of scores based on the number of unique and total amount of operands and operators used [17]. The definition of Operands and Operators has slightly different meanings depending on the implementation, but generally, operators are all normal operators, keywords and brackets of all kinds ((), [],). Operands are variables, methods or function declarations and constants such as Boolean values and strings. Halstead metrics produce useful scores of the complexity of code, such as an estimate of the time to program and the potential for bugs. Although Halstead metrics are not free from critique, various definitions can cause trouble when comparing scores, the use of magic numbers (18 appearing in the Time formula) appears to be arbitrary and it can be argued that the use of operands and operators is too simplistic for modern programming.

Other code metrics take different approaches to Halstead to achieve the same goal of scoring code. The *Flesch-Kincaid readability test* [14] is a method to score the readability of human written language, based on the number of *syllables per words* and *words per sentence*, similar to how Halstead used operators and operands. Kurt Starsinic developed a module, *Fathom.pm*, to apply the Fleshc-Kincaid test to Perl code, producing a score based on the number of *tokens per expression*, *expressions per statement* and *statements per subroutine* [32].

Code Complexity =
((average expression length in tokens) * 0.55)
+ ((average statement length in expressions) * 0.28)
+ ((average subroutine length in statements) * 0.08)

The formula would produce a score from 1 - 7, where 5 is "Trival" and 7 is "Very Hairy". However, he did not provide any justification for the weights used in the formula except that they were fined-tuned through trial and error. Börstler, Caspersen and Nordström would later produce a similar metric method to Starsinic, while also using the FLesch-Kincaid test. They introduced the Software Readability Ease Score (SRES) by treating the "lexemes of a programming language as syllables, its statements as words, and its units of abstraction as sentences" [2]. They argued that this type of static code analysis was a clear indicator of how readable, and thus maintainable the code was. However, to quote Dijkstra, other factors affect the "*goodness*" of code, such as control flow.

In 1976, McCabe proposed a technique called *Cyclomatic complexity* for scoring the control flow a program takes [19]. This was an attempt to "*provide a quantitative basis for modularization*" as a way to identify in advance modules which will be difficult to maintain. McCabe provides the definition "**Definition 1:** *The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is*

$$v(G) = e - n + p$$

Cyclomatic complexity can be viewed as building up graphs from smaller components, examples of which McCabe included.

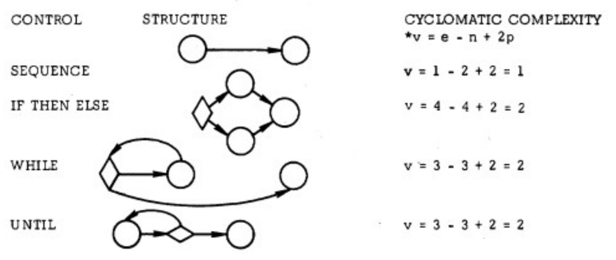

Figure 1: Generated Control Structure Graphs [19]

However, Cyclomatic complexity does not handle all the intricacies of code, especially since it does not consider else, do & try, object creation and method calls. Furthermore, there are different interpretations of the method, with some researchers testing complexity at only a module level, while others at a program level, summing up the scores of individual modules. This inconsistency disrupts the comparison of results [30, 39]. While none of these metrics are perfect estimates of complexity, they allow for fast, accurate judgment of code. Halstead and McCabe developed their metrics during an era of batch programming where systems were significantly smaller today. While they may be ill-suited for large, industrial software, this makes them the perfect solution to measuring smaller, generated code pieces which inherently lack the complexity in which these metrics can not accurately judge [31].

To accurately ascertain a meaningful judgement of code quality, several measurements must be taken that account for the complexity of each line *and* the path the code can take throughout the program. This approach of using code metrics to judge LLM generation is sparse in the current field of research [23, 26], with researchers focusing on functional performance. The practicality of code generation relies in large part on the maintainability of the code, not just its functional performance, and requires better understanding.

4 THE RESEARCH

RQ: Can large language models generate code for small-scale problems that compete with model, human-written answers.

Null Hypothesis H_0: Large language models can generate code for small-scale coding problems that produce equal scoring in a range of given metrics when compared to model, human answers.

Alternative Hypothesis H_1 Large language models can generate code for small-scale coding problems that produce higher scoring in a range of given metrics when compared to model, human answers.

4.1 Research Methods

My research will be grounded in a positivist ontological stance, attempting to gather an objective view of performance that can be replicated consistently elsewhere, supported by a sound methodology and statistical analysis. For the study, GPT3.5 will be tasked with solving 20 questions of varying difficulty. GPT3.5 has been chosen due to its current widespread use and ease of access. GPT's functional performance will be measured using *pass@k* and its ability to produce quality code will be measured using two metric scores, Halstead and cyclomatic complexity. Halstead metrics will allow a comparable overview of generations and the model's ability to consistently write clean code. Cyclomatic complexity will focus on the structure of said code, providing insight into maintainability. This two-pronged approach will give an overview of the models practicality. If the model can produce code that scores higher than the model answers' scores, then the H_1 will be accepted - GPT is competitive.

All data pre-analysis will be in the form of floats. Several samples will be taken at different temperatures at different sample rates. Temperatures will range from T = 0.3 - 0.9 and sample sizes will range from k = 10 - 100. This is following what OpenAI's team has done with the *pass@k* metric so that my data is easily comparable with theirs and other researchers in the field. Nine groups of data will be produced from this, each with the answers to each question in the set. The size of each group will be different depending on the sample rate.

0	T = 0.3	T = 0.6	T = 0.9
k = 10	G13	G16	G19
k = 50	G53	G56	G59
k = 100	G103	G106	G109

Table 1: Groups of Samples

For example, G32 will have $20 * 100- > 2,000$ generations while G19 will have $20 * 10- > 200$ generations. The generations will be evaluated for functional performance, this will allow me to gauge the difficulty of the questions for reflection on the study. The groups will then be combined, to form 20 groups, one per question, with 480 generations each, 9,600 in total. Lexical analysis will extract tokens for calculating the Halstead and Cyclomatic Complexity per generation. These will be summed and averaged for each question, producing one mean value per question.

4.2 Hypothesis Testing

To test the null hypotheses, I will be using an independent samples 1-tailed t-Test to compare the GPT and human groups. This test is a parametric test that compares whether there is a statistically significant difference between the two groups' means.

The independent variables will be the two groups, the collection of GPT generations and human answers. The dependent variable will be the metric score and each sample will be a question. In total, 20 samples per group. The data for each sample will be numerically continuous, ratio values. Before the t-test, a f-test will be evaluated to check the homogeneity of variances and a Shapiro test to ensure the data is normally distributed. The R code for the t-Test is shown below, with *testData* as the given example data set. The full R code can be seen in the linked repository.

4.3 Pilot Study

A pilot study, which is already taking place, uses only 6 simple programming questions. Code is generated for all 6 using the same temperature, 0.6, and are being tested against Halstead metrics. Preliminary data has found that GPT produces competitive code against model answers for these simple problems, aligning with results from other research [9]. Informed by this, only medium to difficult questions will be used, as best to push the limits of code generation. An example of one of these pilot problems, as it was given in prompt form, is shown below, along with one of the generated responses. The complete set of questions is viewable in appendix section .1.

Problem 2: *Given a string s containing just the characters '(', ')', '', '', ' [' and ']', determine if the input string is valid. An input string is valid if: Open brackets must be closed by the same type of brackets. Open brackets must be closed in the correct order. Every close bracket has a corresponding open bracket of the same type. Return a Boolean Value. Name the function P2*

Generated Solution, T = 0.6 - (*mapping has been renamed to m for spacing*)

```
def P2(s):
 stack = []
 m = {')': '(', '}': '{', ']': '['}
 for char in s:
   if char in m:
       if not stack or
           stack.pop() != m[char]:
            return False
   else:
       stack.append(char)
 return not stack
```

Halstead Metric	Generated Answer	Human Answer
Temperature	0.6	NA
Distinct Operators	18.0	17
Distinct Operands	15.0	18
Total Operators	41.0	50
Total Operands	27.0	32
Vocabulary	33.0	35
Length	68.0	82
Estim Prog Len	133.66	144.55
Volume	343.02	420.6
Difficulty	16.2	15.11
Effort	5556.9	6355.75
Time	308.72	353.1
Estim Bugs	0.11	0.14

Table 2: Problem 2: Table of Halstead Scores for Human and generated code

4.4 Ethical Considerations

As the nature of this research is a desk-based experiment, there is no need for participants. For the human-written answers to problems, I will be carefully constructing answers which abide by commonly accepted coding standards, rather than sourcing the answers from participants. Using an LLM runs the risk of the model producing code that matches private code in its training data, however, not only is matching code extremely rare for GPT models ("< 0.1%") [4], all training data is public access which is exempt from copyright under research use [13].

For the pilot study, I had one collaborator, Joseph Walton-Rivers, a lecturer at Falmouth University. He provided the human-written code solutions for the pilot study's comparison. No personal data was collected about him or his submissions and the code was collected before the pilot study began, he had no involvement or responsibilities during the study.

5 ARTEFACT

The artefact for this project will be a combination of several components, code generation, response testing and metric gathering, all combined in one pipeline streamlining the research.

- Generation
  Each question will have a response generated using OpenAIs API, using the GPT-3.5 turbo model. Each question can be sent *k* number of times, depending on the desired sample rate, temperature can also be varied per request from 0.3 - 0.9. While prompt engineering can almost guarantee only valid code is returned, some cleaning of the response will have to take place to remove extra text returned, such as text explaining the code. No modification of the generated code sample will take place during the cleaning.
- Testing
  The gathered responses will written to a file and automatically run against a wide range of test cases, suited for each question, to test whether the return code can pass the question. These are examples of these in the appendix section .2.
- Code Analysis
  The generated code will be analysed through two methods, Halstead and Cyclomatic Complexity. Halstead will provide a wide range of 12 metric scores about the generated code, while cyclomatic complexity will provide a single score about the generated code's complexity. Both methods will be implemented within the artefact, according to the original algorithms as written by Halstead and McCabe, rather than being imported through an external library.

Development is already underway due to the preceding pilot study. The artefact and generated code will all be written in Python, due to the non-time-critical nature of the code, and the ease with which Python can handle API requests, file I/O and unit tests. Results of the study will be written to a csv file for all calculations, allowing for easy storage, use and transportation of data into R for further analysis.

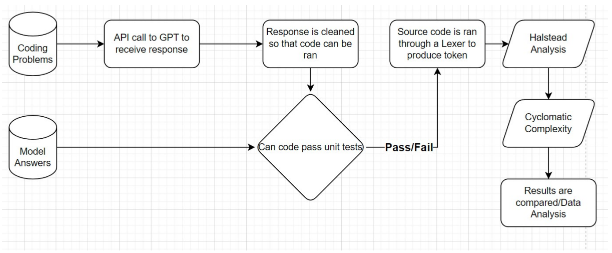
Coding Problems
API call to GPT to receive response
Response is cleaned so that code can be ran
Source code is ran through a Lexer to produce token
Halstead Analysis
Model Answers
Can code pass unit tests
Pass/Fail
Cyclomatic Complexity
Results are compared/Data Analysis

Figure 2: Generation to Metrics Pipeline

5.1 Development Lifecycle

Development will take place with an active use of version control through Github. To program the artefact, a waterfall-agile hybrid approach with elements of Test Driven Development will be used. Tests for the lexer and metrics will be written before their complete implementation, to ensure the implementation is true to the original implementations. A waterfall-like approach will be taken to ensure academic integrity within the metrics and sub-sequence analysis since if the design changed during the implementation of these features this would invite bias into the study. However, there is room for an iterative agile-like approach for the tools used, such as changing programming language if necessary or modifying the lexing approach. Testing of the artefact is described below.

Testing of the artefact will involve 5 types of tests,

- **Unit testing** of every individual function that is necessary to ensure functionality. Carried out by unit tests
- **Integration Testing** to ensure that all components can function together. Carried out by a range of unit tests
- **Stress Testing** to ensure that the artefact can handle at least twice the maximum expected input size. To test, the software, large Python files will be run against the analyser to ensure the components, working together, can handle long input.
- **Acceptance Testing** to ensure that the artefact can meet the requirements of the study.
- **A Sanity Test** after every push to the main branch to ensure that all the main features are still functional.

6 DISCUSSION

Within programming circles, the prevalence of generative models shows no sign of slowing down in common use; if the generation of code is shown to be practical, the implications of such must be considered.

6.1 Plagiarism

As previously discussed, generated code is formed through prediction. Every line of code generated is technically new since it's never directly copied and the model has no knowledge of its training data, *however* all its training has come from human-written code, therefore there is a non-zero chance it could directly replicate lines of code from its training data. This can be argued from an ethical and legal perspective as plagiarism, no matter how unintentional. The current method to circumvent this is to use repositories with licensing that allows free use of its code. This is also not an issue if the model has been fine-tuned on private company repositories for in-house use, which is now a feature offered by CoPilot [22]. However, the use of generated code is still in complete control of the user making the request, it is reasonable to view the code as akin to auto-complete, and thus still the user's own work.

6.2 Workload

The use of generative models might also have a considerable impact on programmers' workflow and involved labour markets. As generative code becomes more and more feasible for use, the impact on a programmer's workload is uncertain, but it's likely their focus will drift to their other responsibilities, such as architectural design & analysis, integration, maintenance and client management. This could have effects on the hiring process and structure of development teams if the impact of generation is large enough. Models also import libraries at different rates, "learning" from the most common habits of their training data, displaying a bias towards *mediocrity* . Most likely, this would reinforce already popular and standard imports while making it harder for newly released libraries to gain prominence.

6.3 Security

Programmers must be aware of their inclusion of generations due to possible security concerns within the code. The entire set of training data has not been vetted for possible vulnerabilities and erroneous generations could produce further vulnerabilities [4]. Bad faith actors might also attempt to *poison* the training data. If they can include vulnerable code in the training set, through manipulation or large creation of public repositories, LLM may bias and generate this code, exposing users to high risks.

6.4 Education

The use of ChatGPT is already prominent in education [16], with significant use in essay writing, tools to discover such use, such as ChatGptZero [33] have shown promising results but still result in many false negatives and positives, making it an unreliable method to identify generated works [11]. Generated code is harder to detect, since two implementations of the same algorithm are likely to be very similar to one another, as is the nature of programming. The challenges students face during their studies are often better suited for code generation, as they are smaller in scale than problems faced by *real-world* software engineers. Educators face a difficult challenge in identifying when a student has written code themselves, or simply passed the problems through a model and copied the answer. Not only does this hurt the reputability of education, it affects the quality of the students learning. Reliance on generation removes the student from the difficult process of learning to program; critical thinking, implementation, debugging and research. However, it might also provide a fast starting point for students to start their research. More study is needed to evaluate the impacts of code generation in education.

7 CONCLUSION

The use of LLM is only increasing as are the possible use cases and ethical issues along with them. The best practical understanding we can achieve will help equip students, workers, researchers, educators and businesses to utilize them without exposing themselves to a wide range of issues. This research should provide insight into understanding these generations and hopefully help inform the next steps for LLM uses.

ACKNOWLEDGMENTS

Thank you Joseph Walton-Rivers for providing the model answers in the pilot study

REFERENCES

[1] Amina Adadi and Mohammed Berrada. 2018. Peeking inside the black-box: a survey on explainable artificial intelligence (XAI). *IEEE access* 6 (2018), 52138–52160.

[2] Jürgen Börstler, Michael E Caspersen, and Marie Nordström. 2007. *Beauty and the beast–toward a measurement framework for example program quality.* Department of Computing Science, Umeå University, Sweden.

[3] Davide Castelvecchi. 2016. Can we open the black box of AI? *Nature News* 538, 7623 (2016), 20.

[4] 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. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]

[5] Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM* 11, 3 (mar 1968), 147–148. https://doi.org/10.1145/362929.362947

[6] Edsger W. Dijkstra. 1970. Concern for correctness as a guiding principle for program construction. (jul 1970). http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD288.PDF circulated privately.

[7] Tyna Eloundou, Sam Manning, Pamela Mishkin, and Daniel Rock. 2023. GPTs are GPTs: An Early Look at the Labor Market Impact Potential of Large Language Models. arXiv:2303.10130 [econ.GN]

[8] Norman E Fenton and Martin Neil. 1999. Software metrics: successes, failures and new directions. *Journal of Systems and Software* 47, 2 (1999), 149–157. https://doi.org/10.1016/S0164-1212(99)00035-7

[9] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. 2023. My AI Wants to Know If This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises. In *Proceedings of the 25th Australasian Computing Education Conference* (Melbourne, VIC, Australia) *(ACE '23)*. Association for Computing Machinery, New York, NY, USA, 97–104. https://doi.org/10.1145/3576123.3576134

[10] A. Shaji George and A. S. Hovan George. 2023. A Review of ChatGPT AI's Impact on Several Business Sectors. *Partners Universal International Innovation Journal*

1, 1 (Feb. 2023), 9–23. https://doi.org/10.5281/zenodo.7644359
[11] Farrokh Habibzadeh. 2023. GPTZero Performance in Identifying Artificial Intelligence-Generated Medical Texts: A Preliminary Study. *Journal of Korean Medical Science* 38, 38 (Sep 2023). https://doi.org/10.3346/jkms.2023.38.e319
[12] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:1503.02531 [stat.ML]
[13] United Kingdom Intellectual Property Office. 2014. Exceptions to copyright: Research. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/375954/Research.pdf
[14] Robert P. Jr.; Rogers Richard L.; Kincaid, J. Peter; Fishburne and Brad S. Chissom. 1975. *Derivation Of New Readability Formulas (Automated Readability Index, Fog Count And Flesch Reading Ease Formula) For Navy Enlisted Personnel.* Technical Report. Institute for Simulation and Training. 56. https://stars.library.ucf.edu/istlibrary/56
[15] Donald E. Knuth. 1974. Structured Programming with Go to Statements. *ACM Comput. Surv.* 6, 4 (dec 1974), 261–301. https://doi.org/10.1145/356635.356640
[16] Chung Kwan Lo. 2023. What Is the Impact of ChatGPT on Education? A Rapid Review of the Literature. *Education Sciences* 13, 4 (April 2023), 410. https://doi.org/10.3390/educsci13040410
[17] Halstead M. 1977. *Elements of Software Science.* Elsevier Science Inc.
[18] Zohar Manna and Richard J. Waldinger. 1971. Toward Automatic Program Synthesis. *Commun. ACM* 14, 3 (mar 1971), 151–165. https://doi.org/10.1145/362566.362568
[19] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837
[20] Meta. 2023. Llama 2. https://ai.meta.com/llama/
[21] G. Michaelson. 1996. Automatic analysis of functional program style. In *Proceedings of 1996 Australian Software Engineering Conference.* 38–46. https://doi.org/10.1109/ASWEC.1996.534121
[22] Microsoft. 2021. GitHub CoPilot. https://github.com/features/copilot
[23] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI Pair Programmer: Asset or Liability? *J. Syst. Softw.* 203, C (sep 2023), 23 pages. https://doi.org/10.1016/j.jss.2023.111734
[24] OpenAI Natalie. 2022. ChatGPT - Release Notes. https://help.openai.com/en/articles/6825453-chatgpt-release-notes#h_4799933861
[25] Sourcery (n.d.). 2023. Sourcery | Automatically Improve Python Code Quality. https://sourcery.ai/
[26] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22).* Association for Computing Machinery, New York, NY, USA, 1–5. https://doi.org/10.1145/3524842.3528470
[27] Gabriel Orlanski, Seonhye Yang, and Michael Healy. 2022. Evaluating How Fine-tuning on Bimodal Data Effects Code Generation. arXiv:2211.07842 [cs.LG]
[28] Sundar Pichai. 2023. An important next step on our AI journey. https://blog.google/technology/ai/bard-google-ai-search-updates/
[29] Brent Reeves, Sami Sarsa, James Prather, Paul Denny, Brett A. Becker, Arto Hellas, Bailey Kimmel, Garrett Powell, and Juho Leinonen. 2023. Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (Turku, Finland) *(ITiCSE 2023).* Association for Computing Machinery, New York, NY, USA, 299–305. https://doi.org/10.1145/3587102.3588805
[30] Martin Shepperd. 1988. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal* 3 (March 1988), 30–36(6). Issue 2. https://digital-library.theiet.org/content/journals/10.1049/sej.1988.0003
[31] M. Shepperd and D.C. Ince. 1994. A critique of three metrics. *Journal of Systems and Software* 26, 3 (1994), 197–210. https://doi.org/10.1016/0164-1212(94)90011-6
[32] Kurt Starsinic. 1998. Perl Style. https://www.foo.be/docs/tpj/issues/vol3_3/tpj0303-0006.html
[33] Edward Tian and Alexander Cui. 2023. GPTZero: Towards detection of AI-generated text using zero-shot and supervised methods. https://gptzero.me
[34] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs.nbsp;experience: Evaluating the usability of code generation tools powered by large language models. *CHI Conference on Human Factors in Computing Systems Extended Abstracts* 332 (Apr 2022), 1–7. https://doi.org/10.1145/3491101.3519665
[35] Richard J. Waldinger and Richard C. T. Lee. 1969. PROW: A Step toward Automatic Program Writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence* (Washington, DC) *(IJCAI'69).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 241–252.
[36] Pei-Hsin Wang, Sheng-Iou Hsieh, Shih-Chieh Chang, Yu-Ting Chen, Jia-Yu Pan, Wei Wei, and Da-Chang Juan. 2020. Contextual Temperature for Language Modeling. arXiv:2012.13575 [cs.CL]
[37] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, Zac Kenton, Sasha Brown, Will Hawkins, Tom Stepleton, Courtney Biles, Abeba Birhane, Julia Haas, Laura Rimell, Lisa Anne Hendricks, William Isaac, Sean Legassick, Geoffrey Irving, and Iason Gabriel. 2021. Ethical and social risks of harm from Language Models. arXiv:2112.04359 [cs.CL]
[38] Michel Wermelinger. 2023. Using GitHub Copilot to Solve Simple Programming Problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) *(SIGCSE 2023).* Association for Computing Machinery, New York, NY, USA, 172–178. https://doi.org/10.1145/3545945.3569830
[39] Jacqueline Whalley and Nadia Kasto. 2014. How Difficult Are Novice Code Writing Tasks? A Software Metrics Approach. In *Proceedings of the Sixteenth Australasian Computing Education Conference - Volume 148* (Auckland, New Zealand) *(ACE '14).* Australian Computer Society, Inc., AUS, 105–112.
[40] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming* (San Diego, CA, USA) *(MAPS 2022).* Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

8 APPENDIX

8.1 Six Problems Used in Pilot Study

P1: You are given a non-negative floating-point number rounded to two decimal places celsius, that denotes the temperature in Celsius. You should convert Celsius into Kelvin and Fahrenheit and return it as an array ans = [kelvin, fahrenheit] Name the function P1.

P2: Given a string s containing just the characters '(', ')', '', '', ' [' and ']', determine if the input string is valid. An input string is valid if: Open brackets must be closed by the same type of brackets. Open brackets must be closed in the correct order. Every close bracket has a corresponding open bracket of the same type. Return a Boolean Value. Name the function P2

P3: Given a string of unknown length, find the longest, unbroken sequence of the same character. For example, "aaa" is an unbroken sequence of length 3. In the string "asdjrrrraduu", "rrrr" is the longest sequence of length 4. Return the length. Name the function P3.

P4: Given an array of strings, remove, if any, all duplicate elements. Duplicate elements are elements in the array that are evaluated as the same, so "John" == 'John" but "sam" ≠ "Sam" and "John" ≠ "Sam". Return the length of the new array. Name the function P4.

P5: Given an unsorted array of integers, return the array in sorted order, with index 0 being the smallest and the last index being the largest in the array. Duplicates can be in either order next to each other. Name the function P5

P6: Given two arrays of strings, combine them into one sorted array, with the shortest length string at index 0 and the longest length string at the last index. If two strings are the same length, sum up each string's characters' ASCII value, and use that total, inserting the smallest first. Return the new array. Name the function P6.

8.2 Example Unit Test for Generated and Human Code from The Pilot Study

```
class TestP2(unittest.TestCase):
    def test_valid_brackets(self):
        valid = ["()", "[]", "{}",
                "()[]{}", "{[()]}"]
        for i in valid:
            self.assertTrue(P2(i))
    def test_invalid_brackets(self):
        invalid = ["((",")",")({","})",
                   "][","}{","({[","]})",
                   "({[)}]"]
        for i in invalid:
            self.assertFalse(P2(i))
```

8.3 *pass@k* [4]

$$pass@k := \mathop{\mathbb{E}}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

Received 15 October 2023