Automated Program Repair with the GPT Family, including GPT-2, GPT-3 and CodeX

Márk Lajkó
mlajko@inf.u-szeged.hu
Department of Software Engineering
University of Szeged
Szeged, Hungary

Viktor Csuvik
csuvikv@inf.u-szeged.hu
Department of Software Engineering
University of Szeged
Szeged, Hungary

Tibor Gyimothy
gyimi@inf.u-szeged.hu
Department of Software Engineering
University of Szeged
Szeged, Hungary

László Vidács
lac@inf.u-szeged.hu
Department of Software Engineering
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary

ABSTRACT

Automated Program Repair (APR) is a promising approach for addressing software defects and improving software reliability. There are various approaches to APR, including using Machine Learning (ML) techniques such as neural networks and evolutionary algorithms, as well as more traditional methods such as static analysis and symbolic execution. In recent years, there has been growing interest in using ML techniques for APR, including the use of large language models such as GPT-2 and GPT-3. These models have the ability to generate human-like text and code, making them well-suited for tasks such as generating repair patches for defective programs. In this paper, we explore the use of the GPT family (including GPT-2, GPT-J-6B, GPT-3 and Codex) for APR of JavaScript programs and evaluate their performance in terms of the number and quality of repair patches generated. Our results show that these state-of-the-art language models are able to generate repair patches that successfully fix the defects in the JavaScript programs, with Codex performing slightly better overall. To be precise, in our self-assembled dataset, Codex was able to generate 108 repair patches that are exactly the same as the developer fix for the first try. If we consider multiple patch generations, up to 201 buggy programs are being repaired automatically from the 1559 evaluation dataset (12.89%).

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; *Software testing and debugging*; *Maintaining software.*

APR '24 , April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0577-9/24/04…$15.00
https://doi.org/10.1145/3643788.3648021

KEYWORDS

Automated Program Repair, Transformers, GPT-3, Codex, JavaScript

ACM Reference Format:
Márk Lajkó, Viktor Csuvik, Tibor Gyimothy, and László Vidács. 2024. Automated Program Repair with the GPT Family, including GPT-2, GPT-3 and CodeX. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3643788.3648021

1 INTRODUCTION

Automated program repair (APR) is a field of software engineering that aims to automatically fix defects in computer programs. APR has the potential to significantly improve software reliability and reduce the cost and time associated with manual debugging and repair [42]. In recent years, there has been growing interest in using machine learning techniques for APR [13, 29]. These techniques have shown promise in generating high-quality repair patches for a variety of programming languages and domains [35]. However, APR is a challenging task due to the complexity and variability of software systems with many open challanges [14, 25]. APR approaches may be limited by the quality and coverage of the training data, as well as the ability of the model to generalize to new programs and defects. Additionally, APR approaches must be able to scale to large codebases and handle a wide range of defects. As a result, there is a need for further research to improve the effectiveness and efficiency of APR approaches [10, 37, 46].

Data-driven automated program repair (APR) approaches utilize machine learning techniques to learn from a dataset of programs and their corresponding repair patches. These approaches can be effective in generating high-quality repair patches for a wide range of defects and programming languages [31, 43]. Data-driven APR approaches often involve training a machine learning model on a large dataset of programs and their corresponding repair patches, and then using the trained model to generate repair patches for new programs with defects. One such dataset was made in 2019 by Tufano *et al.* [41], on which these models can be trained and evaluated. Their seminal work has been encased in the CodeXGLUE benchmark [30], featuring a platform for future publications including diverse programming language tasks. The dataset is highly

successful among researchers, new model architectures are proposed rapidly (usually published on arXiv.org first, thus bypassing the traditionally slow publication process) and new records are booked in a monthly basis.

The Generative Pre-trained Transformers (GPT-*) are large language models developed by OpenAI. These models are trained on a massive amount of text data and are able to generate human-like text in a variety of languages and styles. GPT-2 was the first model in the GPT series, and was released in 2019. It was trained on a dataset of 8 million web pages and is able to generate text in a wide range of topics [39]. GPT-3, released in 2020, is an even larger model, trained on a dataset of billions of web pages. It is able to generate text in a variety of styles and languages, and has been used for tasks such as translation, summarization, and question answering [5]. GPT-3.5 is a further improved version of GPT-3, released in 2021. It is trained on an even larger dataset and is able to generate text with improved quality and fluency [5]. CodeX is a large language model developed by OpenAI that is specifically designed for generating code in a variety of programming languages [6]. It is trained on a dataset of millions of lines of code and is able to generate high-quality code that is syntactically and semantically correct. It also powers Github Copilot [16] which turns natural language prompts into coding suggestions across dozens of languages.

In this paper we investigate the effectiveness of several GPT variants in the Automated Program Repair domain. From a recent work [23, 24] we know that GPT-2 is able to generate fixes (although it is not superior compared to state-of-the-art) approaches, and when it is fine-tuned its performance is even better. The current research focuses on more recent GPT models, precisely on GPT-J-6B (HuggingFace), GPT-3 (OpenAI text-davinci-003) and Codex (OpenAI code-davinci-002). We show that without fine-tuning any of these models they are surprisingly effective at generating repair patches as a software developer would. It is also apparent that Codex (which is essentially the same as GPT-3, but trained on a lot of source code) slightly outperforms all other GPT variants. Our experiments contribute to the research community by showing our method, how to prompt these models and what are the limitations of the current approach. Our experiment data is available in a GitHub repository [1].

The structure of the paper is the following. We first describe the details of our APR approach, which involves using GPT-2 GPT-J-6B, GPT-3 and Codex to generate potential repair patches for JavaScript programs with defects. We evaluate the effectiveness of our approach using a dataset of real-world JavaScript programs and compare the performance of the aforementioned GPT models in terms of the number and quality of repair patches generated. We then present an overview of APR and its applications in the field of software engineering in the Section 5 and discuss limitations and future work in Section 6.

Our contributions are the following:

- A Large Language model-based approach to generate patches for 1559 JavaScript bugs.
- We generate patches with both Hugging-face and OpenAI GPT models then evaluate and compare the results.
- Open Repository: All codes and generated patches for each bug are available in our repository

[1]https://github.com/AAI-USZ/APR23-GPT

2 METHOD

In this research, we proposed the use of the GPT family of language models, specifically GPT-2, GPT-J-6B, GPT-3 (text-davinci-003) and Codex (code-davinci-002), for automated program repair. We depicted our approach on Figure 1. The following is a detailed description of the method used in our study: (1) We pre-processed the dataset that was prepared for the 3rd International Workshop on Automated Program Repair [24] by reformatting the code snippets to make them suitable for input to the GPT models, (2) repair generation: we used the fine-tuned GPT models to generate repair suggestions for the buggy code snippets in the dataset, (3) the repair suggestions generated by the GPT models and the CodeX tool are then evaluated against the developer fix (the generated patch should match the fix generated by a human developer). In the following sections we are going to describe these steps in details.

2.1 Models

In this paper, we compare the following pre-trained models: GPT-2 (Hugging Face), GPT-J-6B (Hugging Face), GPT-3 (OpenAI text-davinci-003) and Codex (OpenAI code-davinci-002). In the following, we will describe how we used these models.

GPT-3: GPT-3 models were designed to generate natural language. Text-davinci-003 is the most capable GPT-3 model. 175B parameters.

Codex: The Codex models are descendants of GPT-3 models that can understand and generate code. Their training data consist of both natural language and code. Code-davinci-002 is most capable Codex model. Particularly good at translating natural language to code. 175B parameters.

The davinci models have 175B parameters, and they are also known as GPT-3.5 series. GPT-3.5 series is a series of models trained on a blend of text and code from before Q4 2021 and was trained on an Azure AI supercomputing infrastructure.

We used "text-davinci-003"; according to the official OpenAI documentation, this is the most capable model of GPT-3.

2.2 API Call

To generate patches, we used API calls. An example is shown how we used the API to generate patches automatically using Codex.

```
response = openai.Completion.create(
model="text-davinci-003",
prompt=input,
temperature=0.6,
max_tokens=512,
top_p=0.8,
frequency_penalty=0.0,
presence_penalty=0.0
)
```

The input used is the same for all models. This refers to the code before the bug that we try to fix. Another important parameter is top_p=0.8. This parameter influences the chance of generating the best output. With 1.0 value we would always get the same output from the model and with value 0 it would be completely random.

For GPT-2 and GPT-J-6B Hugging Face API call was used to generate new examples with similar parameters. These two hugging face models can fit on our own GPU contrarily to Codex and GPT-3

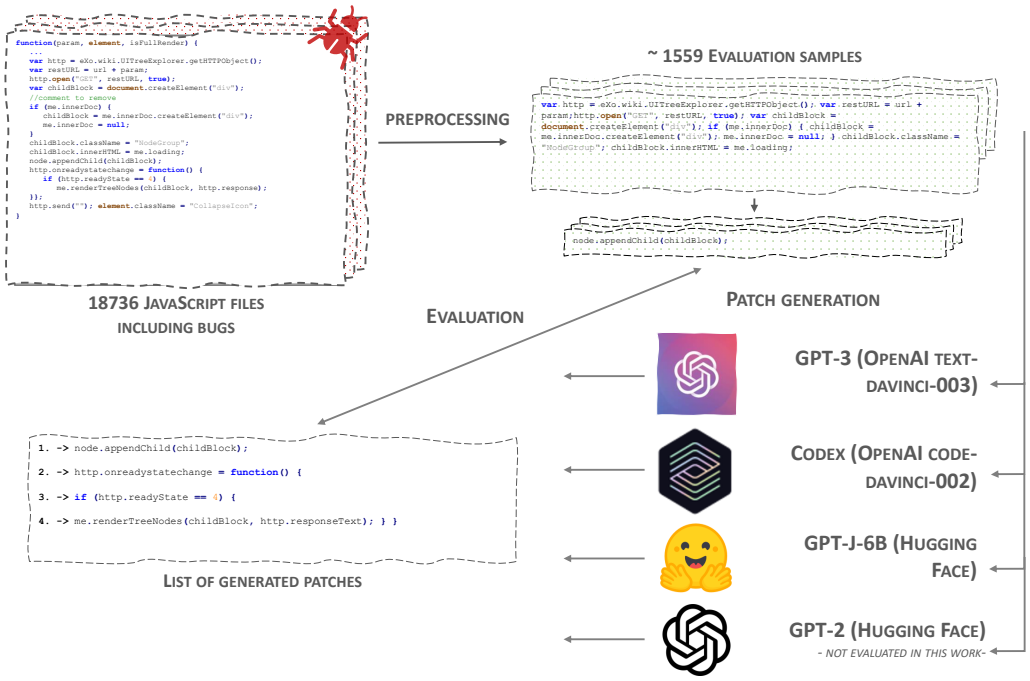

Figure 1: High-level approach of this paper

Model	Available from	Dataset	# parameters
GPT-2	Hugging-face	text_orientated	124M
GPT-J-6B	Hugging-face	text_orientated	6B
GPT-3	OpenAI API	text_orientated	175B
Codex	OpenAI API	code_orientated	175B

where we just use an API Call and let OpenAI generate the output for us.

2.3 Dataset

In our experiment we used a dataset that was prepared for the 3rd International Workshop on Automated Program Repair [24], which consists of 18736 files. This JavaScript dataset was created from the BugsJS [18] dataset, which contains reproducible JavaScript bugs from 10 open-source Github projects. The dataset contains both single- and multi-line bugs as well. The dataset was chosen to be representative of real-world code, with a variety of bugs and complexity levels. Commits were retrieved using the GitHub REST API [17] and GH Archive [15]. The interested reader is encouraged to read the original paper, where the dataset creation is described in more detail [23, 24].

2.4 Preprocessing

After we removed the comments from this dataset we split it up to 16863 training JS files and 1873 test JS files. The training part of the dataset was used to fine-tune GPT-2 in recent work [23, 24]. In this paper we compare only the pre-trained versions of each GPT-models so we are going to focus on the 1873 test files. Out of these 1873 test files we only generated candidates for 1559 files because the bug environment was not always adequate. One example is if the bug is in the first line, we cannot give any prompt to GPT, because unlike BERT like models, GPT uses only the tokens before generated text instead of both sides. We executed some preprocessing steps to save all developer fixes which we can later compare to the fixes the GPT models generate. We further pre-processed the dataset and created a directory consisting of the test files in the proper form to feed it to our GPT models. This means that in these files we saved the 2040 tokens before the bug location for each bug. So in the end we created from the test_split 2 directories. In the first one we have the developer fixes for each bug in the second one we have the prompt of the GPT models for each bug, which consist of 2040 tokens, but in the patch generation step we further reduced this number. This retrieved dataset and the preprocessed version are available on our GitHub.

2.5 Patch generation

As we described in previous sections, from the test split we created 2 directories one of them containing the prompts for the GPT models and the other one containing the developer fixes for each bug. The purpose of patch generation is to create another directory containing all candidates for each bug. We called the GPT models generation function 5 times for each bug, we are going to refer these as generation numbers. For each generation we consider each generated line as a candidate patch. We make sure that we do not generate the same candidate line more than once, and we save the generation number and the line number for each candidate patch.

In summary, after the patch generation step, we have one txt file for each bug, each line of these files is a possible patch and the first line of this file is the first line of the code that was generated in the

first call of the GPT_generate function. Since we try to fix one-liner bugs the comparison between the developer fix and a candidate is pretty straightforward. In the next section, we are going to discuss the details.

3 RESULTS

In this result section, we are going to compare GPT-2, GPT-J-6B, GPT-3 and Codex performance. Not surprisingly overall Codex was the most efficient, GPT-3 was the second, then followed GPT-J-6B and GPT-2 as we can see in Table 1. As we described in Section 2, we generated candidate fixes for each bug and the comparison between the candidate line and the developer fix was carried out with edit distance. As we can see in the above mentioned table there are 5 generation which means that we used each GPT model's generate function 5 times for each bug. The results are aggregated for each generation so in the next generation there can only be greater or equal fixes. The model is less likely to find a new candidate in later generations due to our parameter settings.

EM stands for exact match, this means we accepted a candidate only if the candidate line and the developer fix was the same. There were several cases when the candidate and the developer fix were the same except some white spaces, we tried to be as strict as possible so for EM we did not count these candidates, although we are aware of that these fixes would be acceptable in a real life scenario. Top_x means that in each generation (each time we used GPT Generate function), we tried to match only the first x lines that the GPT model generated. For example top_1 means that we only tried to compare the first line the model generated with the developer fix and Top_10 means that in a given generation, we considered the first 10 lines (of what the GPT generated) as a candidate. ED_5 and ED_10 means that instead of exact match we compared the developer fix with a candidate and if the edit distance was lower than 5, 10 respectively, then we counted the given candidate.

As we can see in the table GPT-J-6B performed the worst when we only considered the first candidate line(First generation, Top1, EM) the model generated for us but as we had more and more candidates the model managed to beat the performance of GPT-2. For example if we considered 5 generations and from each generation the first 10 lines as a candidate then GPT-J-6B generated 34 exact matches to the developer fix, while GPT-2 only generated 27. As we mentioned earlier the parameter settings were the same for each GPT model so we expected GPT-J-6B to outperform GPT-2 in all cases.

As expected OpenAI's GPT-3 and Codex outperformed both hugging face models (GPT-2, GPT-J-6B). Codex was the best performing model. Codex was designed specifically for source code generation while the rest mostly focuses on Natural Language, albeit all models were trained on source code as well. As we can see in the table GPT-3 generated 104 perfect fixes on the first try while Codex generated 108, and GPT-3 generated 162 exact matches in 5 generations and 10 lines while Codex generated 201. An interesting observation is that although the stronger model we use the more EM we get, but the less ED5, ED10 we get. We suppose that this is because the better models are more and more accurate.

4 DISCUSSION

We compared a few generated patches of each model, these can be observed in the Appendix. The patches of the GPT-2 and the GPT-J-6B models tend to be similar and we can say the same about GPT-3 and Codex. It looks like the training data is more likely to affect these model performances than the actual parameter number. We can see both of the models of Hugging Face and the models of OpenAI are capable of generating complex patches, the difference is rather in the accuracy than in the complexity. In example 37 one can see that the OpenAI models are capable of fixing complex regular expressions but Hugging Face models could generate a nearly identical regular expression to the developer fix. The interested reader can observe all of the best generated patches for each model in our GitHub repository [2].

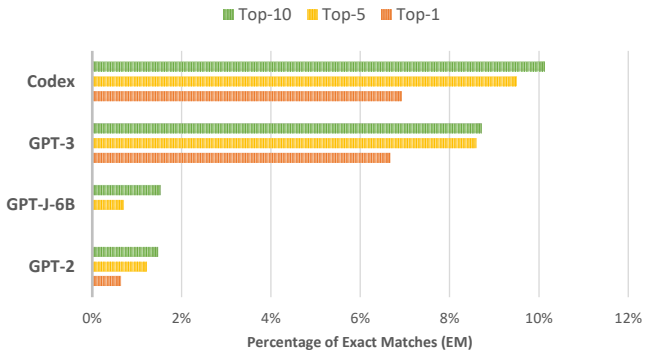

Figure 2: Exact Match (EM) results of the observed models

On Figure 2 we highlighted the Exact Match (EM) results of each model. Here we can see, that the performance of the models retrieved from Hugging Face are way lower compared to the OpenAI models. The top-1, top-5 and top-10 results behave quite similarly in each model and their ranking remains the same. Codex and GPT-3 was able to generate the correct fix on their first try in around 7% of cases, and they were successful around 11% of the cases when they had 10 tries (top-10 result). Overall Codex outperformed all other model variants to a small extent, while GPT-2 and GPT-J-6B achieved a significantly worse result.

5 RELATED WORK

In this paper we focused on the GPT-family which is a Transformer architecture. Most of today's deep-learning approaches that tackle with text-text or code-code problems use similarly a Transformer. For example in [34] authors fine-tune a T5 model for several tasks including fixing bugs, injecting code mutants, generating assert statements, and generating code comments. Their results show that the performance of the model is comparable to the state-of-the-art tools in each downstream task. Variants of the Transformer model are also used for code-related tasks, like in [1] where authors propose a grammar-based rule-to-rule model which leverages two encoders modeling both the original token sequence and the grammar rules, enhanced with a new tree-based self-attention. Their proposed approach outperformed the state-of-the-art baselines in

[2] https://github.com/AAI-USZ/APR23-GPT

Table 1: **Results of the GPT model variants to generate patches automatically. The observed dataset consists of 1559 evaluation samples. We considered the generations in an accumulative fashion: if we consider the first generation and the Top_1 result, only one patch is examined, in contrast in the fifth generation there are five candidate patches (one patch per generation). In this sense, the Top_1 results in the fifth generation includes 5 candidate patches. The abbreviations used are the following: EM - Exact Match, ED_N - Edit Distance within the range N (candidates with character differences less than N).**

	GPT-2								
	Top_1			Top_5			Top_{10}		
Generation	# EM	# ED_5	ED_{10}	# EM	# ED_5	ED_{10}	# EM	# ED_5	ED_{10}
#1	10	18	39	19	29	86	23	35	111
#2	11	22	48	20	37	107	24	44	139
#3	12	23	59	21	42	128	26	52	168
#4	13	26	65	22	46	143	27	57	186
#5	13	26	65	22	46	143	27	57	186
	GPT-J-6B								
#1	0	0	0	11	19	36	24	42	74
#2	0	0	0	13	23	43	31	53	89
#3	0	0	0	13	23	45	32	55	95
#4	0	0	0	13	26	50	33	59	102
#5	0	0	0	14	28	55	34	63	110
	GPT-3								
#1	104	127	167	131	168	231	136	175	250
#2	111	136	186	142	184	265	149	194	287
#3	115	141	198	147	191	281	154	201	306
#4	121	153	215	154	208	313	162	220	341
#5	121	154	217	154	210	319	162	223	348
	Codex								
#1	108	126	159	148	181	237	158	194	266
#2	113	134	171	162	199	272	173	215	309
#3	120	144	186	172	219	300	183	236	341
#4	125	150	193	183	233	319	195	253	367
#5	128	154	202	188	239	333	201	261	384

terms of generated code accuracy. Another seminal work is DeepDebug [10], where the authors used pretrained Transformers to fix bugs automatically. Here Drain *et al.* use the standard transformer architecture with copy-attention. They conducted several experiments including training from scratch, pretraining on either Java or English and using different embeddings. They achieved their best results when the model was pretrained on both English and Java with an additional syntax token prediction. This model is evaluated on the dataset by Tufano *et al.* [41], which was later included into the CodeXGLUE (General Language Understanding Evaluation benchmark for CODE) benchmark [30]. There are other well-known APR datasets as well. ManyBugs [26] contains bugs written in C - it were used to evaluate many well-known APR tools (Genprog [42], Prophet [29], etc.), Bugs.jar [40] is comprised of 1,158 Java bugs and their patches. A few datasets of larger-scale is also available publicly, but the format of these are not suitable for our experiments [8, 22]

Since the previously mentioned datasets did not fit our needs, we created our own data to evaluate the observed models. This is not unique in the field, there are several approaches that operate on their own train-test-validate dataset, like SequenceR [7], Hoppity [9] or CoCoNuT [31]. It is also a common practice to evaluate deep-learning based APR tools directly on Defects4J [21], where such tools recently even outperformed traditional approaches [27]. Despite this standard Generate and Validate (G&V) tools for Automated Program Repair is still widely used to this day. GenProg [42] was one of the first to perform a fully automatic fix with relatively good results. It was originally written for the C programming language, but has since been implemented for Java [32]. In a 2014 initiative, a framework was created that can automatically repair Java programs [33]. It also includes the implementation of several repair strategies, such as Genprog, Kali or Cardumen. There are some web-based APR tools already with the aim to fix JavaScript bugs, but they are usually implemented to solve special problems. Vejovis [36] suggests fixes for errors related to DOM interactions. A tool called BikiniProxy [11] is an HTTP proxy that makes fixes on HTML and JavaScript based on five strategies. A similar approach is followed in the SAFEWAPI tool [4] and in AndroEvolve [19], where authors focus on fixing errors in API calls.

GPT-2 [3] was introduced in 2018, while GPT-3 [5] in 2020 by OpenAI. Since then they have received a large amount of citations, using the model for diverse tasks [44] and several works have investigated the capabilities and limits of the model [44]. Thanks to it's availability the internet is full of examples of the amazing generative capabilities of the model, from poetry, news or essay writing [12]. In the previously mentioned CodeXGLUE benchmark [30] the capabilities of GPT was also utilized. They used their CodeGPT model for several tasks, including code completion. In fact, CodeGPT achieved an overall score of 71.28 in this task. In a more recent work [2], CodeGPT was used as a baseline model for text-to-code and code generation tasks. The model is pretrained on the n CodeSearchNet [30] corpora. Their newly introduced model (PLBART) overperformed the GPT model in the code generation task in every aspect, while in text-to-code generation GPT achieves the best Exact Match (EM) score.

In recent work [23, 24] the authors used the GPT-2 architecture to repair bugs automatically. That approach did not achieve state-of-the-art results, although they used that model to repair programs automatically first. Since then others also incorporated these models for software engineering tasks and also in the APR domain. In a recent work authors introduce Text2App [20], that allows users to create functional Android applications from natural language specifications. In a more recent work, Prenner *et al.* [38] used Codex to repair programs automatically. Although their work is similar to ours, there are key differences. Prenner *et al.* evaluate Codex on the QuixBugs [28] benchmark, which contains artificial samples of limited size (40 bugs in Python and Java). In comparison we created a dataset consisting of 1873 JavaScript samples and evaluated multiple GPT variants on it. The used prompts are also differ: in this paper the prompt for the models are always the same (the buggy code) and the expected output is the developer fix. On the other hand in [38], authors experimented with different prompts, even giving Codex a hint where the buggy line is. We also experimented with multiple completion for each problem and examined edit distances as well, while Prenner *et al.* only evaluated their approach using a single prompt.

As mentioned in [45] Codex has an edit/insert mode now which is very promising since with this tool, developers can use both sides of a bug location to generate a patch.

6 LIMITATIONS AND FUTURE WORK

In this paper we used large language models which were trained on a variety of text information, including source code. Despite the fact that these models were not fine-tuned to repair programs, they are quite effective in this task with some clear limitations.

6.1 Data leakage

Since the used language models were also trained on source code (especially Codex), we cannot guarantee that the used data was not in their training set. To address this issue one should know exactly which repositories were included by OpenAI, and that information is not widely available. However, there are mitigating factors: (1) since the models were trained until a certain period of time, the data used by us is of a different version compared to the data OpenAI might have used; (2) although our dataset suffices our evaluation criteria, it constitutes only a tiny fraction of the training data used by OpenAI.

6.2 Fine-tuning

From a previous study [23] we know that the fine-tuned version of GPT-2 overperformed the pre-trained version, thus we speculate that it also holds for GPT-3 and Codex. Since their training is very hardware intensive, it is out of scope of the current research paper, however fine-tuning is available through API calls that might improve performance.

6.3 Different prompts

GPT-3 operates quite well in the few-show setting, where the task description is given to the model with a few samples [5]. It also holds for Codex, so experimenting with prompts might also improve their performance. From [38] we know that Codex behaves differently for different input prompts, but from their research it is not clear which are the most useful - a more thorough analysis is needed in the field.

Codex edit/insert tool is also a promising tool to use in this research field since it allows to use the context of the bug from both sides(tokens both before and after the bug location) instead of just the tokens before the bug location. In the future, we are planning to investigate how effective this tool is in Automatic Program Repair.

7 CONCLUSIONS

In this paper we used GPT-J-6B, GPT-3 and Codex to generate patches for buggy programs automatically. Compared to previous works on GPT-2 we showed that these models outperform GPT-2 in orders of magnitude in terms of the number and quality of repair patches generated. By evaluating these GPT variants on our dataset it is apparent that Codex generated the most correct patches, which is not that surprising since it was designed to produce source code. On the other hand, these pre-trained models were used without fine-tuning them, which can presumably improve the performance of these models even more. In conclusion, our results demonstrate the potential of GPT-3 and Codex for APR of JavaScript programs and suggest that these models may be useful tools for improving software reliability in practice. Although the comparison between different APR approaches are not straightforward due to several factors (different databases, different types of bugs etc.), we can conclude that while pre-trained GPT-2 was able to fix the bugs in only 1.73%, the best-performing pre-trained language model, Codex was able to fix bugs in 12.89% which is a great improvement.

ACKNOWLEDGMENTS

This work was supported in part by the ÚNKP-23-3-SZTE-435 New National Excellence Program of the Ministry for Culture and Innovation, by the national project TKP2021-NVA-09 implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the source of the National Research, Development and Innovation Fund and by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory.

REFERENCES

[1] 2021. Grammar-Based Patches Generation for Automated Program Repair. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021* (2021), 1300–1305. https://doi.org/10.18653/v1/2021.findings-acl.111

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. (mar 2021), 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211 arXiv:2103.06333

[3] Ilya Sutskever Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei. 2020. [GPT-2] Language Models are Unsupervised Multitask Learners. *OpenAI Blog* 1, May (2020), 1–7.

[4] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. *SAFEWAPI: Web API Misuse Detector for Web Applications.* Association for Computing Machinery, New York, NY, USA. 507–517 pages. https://doi.org/10.1145/2635868.2635916

[5] 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, T. J. Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeff 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. 2020. Language Models are Few-Shot Learners. *ArXiv* abs/2005.14165 (2020).

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray,

Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code.

[7] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 01 (sep 2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179 arXiv:1901.01808

[8] Zimin Chen and Martin Monperrus. 2018. The CodRep Machine Learning on Source Code Competition. (2018). arXiv:1807.03200

[9] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. 2020. *Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs.* Technical Report. 1–17 pages.

[10] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. 2021. Generating bug-fixes using pretrained transformers. *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021* (jun 2021), 1–8. https://doi.org/10.1145/3460945.3464951 arXiv:2104.07896

[11] T. Durieux, Y. Hamadi, and M. Monperrus. 2018. Fully Automated HTML and Javascript Rewriting for Constructing a Self-Healing Web Proxy. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE).* 1–12. https://doi.org/10.1109/ISSRE.2018.00012

[12] Katherine Elkins and Jon Chun. 2020. Can GPT-3 Pass a Writer's Turing Test? *Journal of Cultural Analytics* (sep 2020). https://doi.org/10.22148/001c.17212

[13] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[14] Mariani Leonardo Gazzola Luca, Micucci Daniela. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (jan 2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[15] GHArchive 2023. GH Archive Official Website. https://www.gharchive.org.

[16] GitHub Copilit 2023. GitHub Copilot. https://github.com/features/copilot.

[17] GitHub REST API 2023. GitHub REST API Official Website. https://docs.github.com/en/rest.

[18] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: A benchmark of javascript bugs. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019.* 90–101. https://doi.org/10.1109/ICST.2019.00019

[19] Stefanus A. Haryono, Ferdian Thung, David Lo, Lingxiao Jiang, Julia Lawall, Hong Jin Kang, Lucas Serrano, and Gilles Muller. 2020. AndroEvolve: Automated Android API Update with Data Flow Analysis and Variable Denormalization. https://doi.org/10.48550/ARXIV.2011.05020

[20] Masum Hasan, Kazi Sajeed Mehrab, Wasi Uddin Ahmad, and Rifat Shahriyar. 2021. Text2App: A Framework for Creating Android Apps from Text Descriptions. (2021). arXiv:2104.08301

[21] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings.* Association for Computing Machinery, Inc, 437–440.

[22] Rafael Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020* (may 2020), 573–577. https://doi.org/10.1145/3379597.3387491 arXiv:1905.13334

[23] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. 2022. Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?. In *Computational Science and Its Applications – ICCSA 2022 Workshops*, Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, and Chiara Garau (Eds.). Springer International Publishing, Cham, 79–91.

[24] Márk Lajkó, Viktor Csuvik, and László Vidács. 2022. Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2). In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR).* 61–68. https://doi.org/10.1145/3524459.3527350

[25] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (oct 2018), 3007–3033. https://doi.org/10.1007/s10664-017-9577-2

[26] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (dec 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[27] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. *Proceedings - International Conference on Software Engineering* 2022-May (2022), 511–523. https://doi.org/10.1145/3510003.3510177 arXiv:2205.01859

[28] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017).* Association for Computing Machinery, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941

[29] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016* (2016), 298–312. https://doi.org/10.1145/2837614.2837617

[30] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined* (2021). arXiv:2102.04664

[31] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* 20 (2020), 101–114.

[32] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (aug 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[33] Matias Martinez and Martin Monperrus. 2016. ASTOR: A program repair library for Java (Demo). In *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis.* Association for Computing Machinery, Inc, New York, New York, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[34] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. *Proceedings - International Conference on Software Engineering* (may 2021), 336–347. https://doi.org/10.1109/ICSE43902.2021.00041 arXiv:2102.02017

[35] Martin Monperrus. 2020. *The Living Review on Automated Program Repair.* Technical Report.

[36] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. 2014. Vejovis: suggesting fixes for JavaScript faults. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014.* ACM Press, New York, New York, USA, 837–847. https://doi.org/10.1145/2568225.2568257

[37] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. 2021. CoTexT: Multi-task Learning with Code-Text Transformer. (may 2021), 40–47. https://doi.org/10.18653/v1/2021.nlp4prog-1.5 arXiv:2105.08645

[38] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. *Proceedings - International Workshop on Automated Program Repair, APR 2022* (2022), 69–75. https://doi.org/10.1145/3524459.3527351

[39] Alec Radford, Tim Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. [GPT-1] Improving Language Understanding by Generative Pre-Training. *Preprint* (2018), 1–12.

[40] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. *Proceedings - International Conference on Software Engineering* (2018), 10–13. https://doi.org/10.1145/3196398.3196473

[41] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19).* IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[42] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09).* IEEE Computer Society, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[43] Li Yi, Shaohua Wang, and Tien N. Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering.* IEEE Computer Society, 602–614. https://doi.org/10.1145/3377811.3380345

[44] Tony Z. Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate Before Use: Improving Few-Shot Performance of Language Models. (2021). arXiv:2102.09690

[45] Martin Mirchev Abhik Roychoudhury Shin Hwei Tan Zhiyu Fan, Xiang Gao. 2023. Automated Repair of Programs from Large Language Models. (jan 2023). arXiv:2205.10583

[46] Yueting Zhuang, Ming Cai, Xuelong Li, Xiangang Luo, Qiang Yang, and Fei Wu. 2020. The Next Breakthroughs of Artificial Intelligence: The Interdisciplinary Nature of AI. *Engineering* 6, 3 (mar 2020), 245–247.

Appendix

Examples of best fixes. GPT-2 upper left corner, GPT-J-6B upper right corner, GPT-3 lower left corner, Codex lower right corner

```
//6
callbacks.forEach(function(fn, i){
callback = fn(method);

//12
if (2 == arguments.length) {
if (2 == arguments.length) {

//14
var chalk = require('chalk');
var hl = require('hl');

//17
expect(console.log.calledWith(sinon.match('Date'
    ))).be.true;
console.log
  .restore();

//24
app.match('/user/123').should.have.length(3);
app.match.get('/user/123/:op?').should.have.
    length(3);

//37
var rSwigComment = /\{#[\s\S]*?#\}/g;
var rSwigVar = /\{[\s\S]*?\}\}/g;

//1022
if ('OPTIONS' == req.method) return res.send
    (200);
api.all('*', function(req, res, next){
```

```
//6
callbacks.forEach(function(fn, i){
callbacks.route = path;

//12
if (2 == arguments.length) {
if (2 == arguments.length) {

//14
var chalk = require('chalk');
var path = require('path')

//17
expect(console.log.calledWith(sinon.match('Date'
    ))).be.true;
expect(console.log).to.be.calledWith(hexo.data.
    post);

//24
app.match('/user/123').should.have.length(3);
app.match.all('/user/123').should.have.length(3)
    ;

//37
var rSwigComment = /\{#[\s\S]*?#\}/g;
var rSwigVar = /\{\{[\s\S]*?\}\}/g;

//1022
if ('OPTIONS' == req.method) return res.send
    (200);
if(!req.query.method) {
```

```
//6
callbacks.forEach(function(fn, i){
callbacks.forEach(function(fn){

//12
if (2 == arguments.length) {
if (2 == arguments.length) {

//14
var chalk = require('chalk');
var file = new File(baseDir);

//17
expect(console.log.calledWith(sinon.match('Date'
    ))).be.true;
expect(console.log.calledWith(sinon.match('Date'
    ))).to.be.true;

//24
app.match('/user/123').should.have.length(3);
app.lookup('/user/123').should.have.length(3);

//37
var rSwigComment = /\{#[\s\S]*?#\}/g;
var rSwigComment = /\{#[\s\S]*?#\}/g;

//1022
if ('OPTIONS' == req.method) return res.send
    (200);
if ('OPTIONS' == req.method) return res.send
    (200);
```

```
//6
callbacks.forEach(function(fn, i){
callbacks.forEach(function(fn){

//12
if (2 == arguments.length) {
if (2 == arguments.length) {

//14
var chalk = require('chalk');
var chalk = require('chalk');

//17
expect(console.log.calledWith(sinon.match('Date'
    ))).be.true;
expect(console.log.calledWith(sinon.match('baz')
    )).to.be.true;

//24
app.match('/user/123').should.have.length(3);
app.match('/user/123').should.have.length(3);

//37
var rSwigComment = /\{#[\s\S]*?#\}/g;
var rSwigComment = /\{#[\s\S]*?#\}/g;

/1022
if ('OPTIONS' == req.method) return res.send
    (200);
if ('OPTIONS' == req.method) return res.send
    (200);
```

The numbers indicate the bug number. For each bug we have two lines, the first line is the developer fix (target) the second line is the best candidate fix of our model.