Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2)

Márk Lajkó
mlajko@inf.u-szeged.hu

Viktor Csuvik
csuvikv@inf.u-szeged.hu

László Vidács
lac@inf.u-szeged.hu

Department of Software Engineering
MTA-SZTE Research Group on
Artificial Intelligence
University of Szeged
Szeged, Hungary

ABSTRACT

The goal of Automated Program Repair (APR) is to find a fix to software bugs, without human intervention. The so-called Generate and Validate (G&V) approach deemed to be the most popular method in the last few years, where the APR tool creates a patch and it is validated against an oracle. Recent years for Natural Language Processing (NLP) were of great interest, with new pre-trained models shattering records on tasks ranging from sentiment analysis to question answering. Usually these deep learning models inspire the APR community as well. These approaches usually require a large dataset on which the model can be trained (or fine-tuned) and evaluated. The criterion to accept a patch depends on the underlying dataset, but usually the generated patch should be exactly the same as the one created by a human developer. As NLP models are more and more capable to form sentences, and the sentences will form coherent paragraphs, the APR tools are also better and better at generating syntactically and semantically correct source code. As the Generative Pre-trained Transformer (GPT) model is now available to everyone thanks to the NLP and AI research community, it can be fine-tuned to specific tasks (not necessarily on natural language). In this work we use the GPT-2 model to generate source code, to the best of our knowledge, the GPT-2 model was not used for Automated Program Repair so far. The model is fine-tuned for a specific task: it has been taught to fix JavaScript bugs automatically. To do so, we trained the model on 16863 JS code snippets, where it could learn the nature of the observed programming language. In our experiments we observed that the GPT-2 model was able to learn how to write syntactically correct source code almost on every attempt, although it failed to learn good bug-fixes in some cases. Nonetheless it was able to generate the correct fixes in most of the cases, resulting in an overall accuracy up to 17.25%.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APR'22, May 19, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527350

CCS CONCEPTS

• **Computing methodologies** → Natural language generation; Neural networks; • **Hardware** → **Failure recovery, maintenance and self-repair**; • **Software and its engineering** → Software testing and debugging.

KEYWORDS

Automated Program Repair, Machine learning, JavaScript, Code Refinement, GPT

ACM Reference Format:
Márk Lajkó, Viktor Csuvik, and László Vidács. 2022. Towards JavaScript program repair with Generative Pre-trained Transformer (GPT-2). In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524459.3527350

1 INTRODUCTION

The existence of large datasets and the availability of cheap computing capacities have facilitated the great development of artificial intelligence and machine learning in recent years [48]. The technologies underpinning AI have made huge leaps in the past decade, bringing exciting applications such as language understanding, vision recognition, and intelligent digital assistants. Recent years also changed the field of Automated program Repair, where several data-driven approach has been published thus forming a separate branch of research [34]. These techniques usually create a large train-test-validate database and evaluate the APR tool on that, just like a traditional ML model. The criterion to accept a patch as a correct one is rather strict in these cases: the produced patch should be exactly the same as the one which were created by the developer who fixed the bug. By comparison tools that follow the Generate and Validate approach, validation is usually done against an oracle, which is usually the test suite. A program is marked as a possible fix, if it passes all the available test cases. This latter condition gives no assurance that the program is *correct*, since over- and underfitting [27] often occurs, resulting in inadequate patches. Although there are some approaches that tried to tackle with this problem [8, 10, 14], the question of patch correctness is considered to be still open [15].

As more and more data-driven APR approaches are being published, the recorded results are promising [9, 10, 22, 31, 46]. The architecture of the underlying model and the dataset varies from paper to paper, thus the comparison is rather challenging. A first

attempt was made in 2019 by Tufano *et al.* in [42] to create a public dataset 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. At the time of writing this article the top three approaches are CoTexT [36], PLBART [1] and DeepDebug [10] with accuracy values ranging from roughly 18% to 23%. However either the aforementioned approaches propose a new architecture or use state-of-the models, to the best of our knowledge, the APR community has not yet used the GPT-2 [2] model to automatically fix bugs at the time of writing.

The original Generative Pre-trained Transformer, or in short GPT, model was published by OpenAI [38], but didn't gain such an immense popularity since other NLP models seemed to overperform it. However with the introduction of GPT-2 the table have turned since the model could do something that other models never could (or at least was not designed for): write stories about talking unicorns [2]. There were no fundamental algorithmicbreakthroughs concerning GPT-2, the original model was essentially scaled-up, resulting a model with 10x more parameters than the original. However GPT was designed to generate coherent sentences and paragraphs (since it was originally trained on the text from 8 million websites), fine-tuning it made possibly for us to generate source code.

Most of today's Automated Program Repair (APR) tools are implemented in such a way to repair programs written in C, Java or even Python. However for the eighth year in a row, JavaScript (JS) is the most commonly used programming language [41]. It is the de-facto web programming language globally and the most adopted language on GitHub [17]. JavaScript is massively used in the client-side of web applications to achieve high responsiveness and user friendliness. In recent years, due to its flexibility and effectiveness, it has been increasingly adopted also for server-side development, leading to full-stack web applications [21]. For these very reasons we designed our experiments to operate on JavaScript, thus we trained and evaluated the GPT-2 model on JavaScript.

To the best of our knowledge, the GPT-2 model was not used for Automated Program Repair so far. In addition we wanted to gain ground for JavaScript in the APR field, since it is mostly dominated by approaches for Java or C. To be able to train the GPT model, we mined 18736 bug-fixing commits from GitHub and preprocessed them before fed to the model. These samples are divided in the classic train-test-validation sets and the model was evaluated on these samples. Based on our experiments GPT-2 was able to repair 126 programs on first try, while it generated correct patches in 269 cases if it had multiple chances to do so.

Although GPT-2 is not the latest GPT model to generate source code, we found it interesting to use it and repair programs with it. We hypothesize that the results achieved can be further improved with newer and larger model variants. During the experiments our resources were limited, thus we couldn't experiment with models of more parameters. Although it limits our work to some degree, the training data we assembled and the experiments are reproducible with larger models as well, for any future researchers in the field.

The results indicate that indeed, GPT-2 can successfully predict fixed JavaScript code for most of the cases. We publish the constructed dataset, the source code of the model and the trained models on GitHub [1].

The paper is organized as follows. After a high-level overview of our research, the dataset and the model is described in Section 2. Thereafter Section 3.1 and Section 3.2 describe the preprocessing and training steps. After that the process of patch generation is depicted in Section 3.3 and we present the settings with which the experiments were carried out. Evaluation and analysis are presented in Section 4, followed by the discussion of this experiment. Related work is discussed in Section 5, and we conclude the paper in the last section.

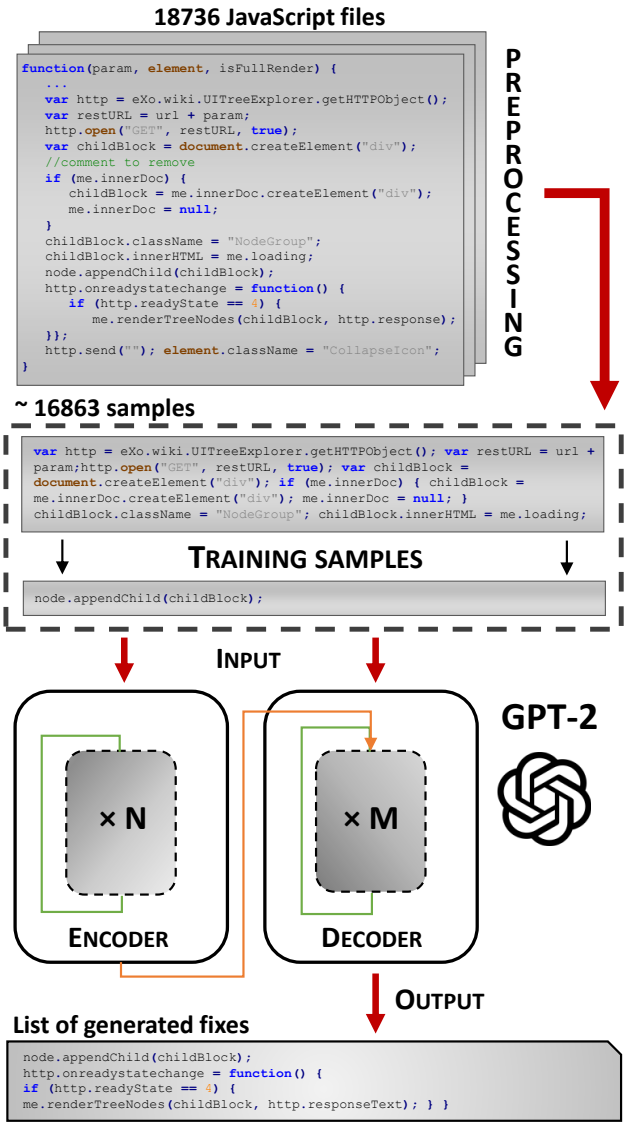

Figure 1: The high-level approach of patch generation.

[1] https://github.com/RGAI-USZ/APR22-JS-GPT

2 APPROACH

In Figure 1 one can observe the high-level approach of this paper. First, JavaScript files are being preprocessed so they can be used as training samples to our GPT-2 model. These samples form a (p_{buggy}, p_{fixed}) tuple, where p_{buggy} is the state of the program before it has been fixed, while p_{fixed} is the applied patch. Note that we focused on bugs which affect only one line, thus p_{fixed} is always a single line, while p_{buggy} is the 900 tokens before that. After training the model is able to predict patches for a given input. To evaluate the model we compared these outputs to developer fixes. Note that the GPT-2 model is able to generate more than a single line, taking advantage of this, we handled the output as an ordered list and made experiments that investigate not only the first line (candidate patch) but the the ones that follow as well. In the next sections we describe the used dataset and briefly introduce the GPT-2 model.

2.1 Dataset

For the experiments we created our own dataset from BugsJS [19] which contains reproducible JavaScript bugs from 10 open-source Github projects. The dataset contains both single- and multi-line bugs as well. The detailed description of these bugs are beyond the scope of the current research, the interested reader is encouraged to take a look at the original paper for further details. We retrieved commits from it using GitHub REST API [18] and GH Archive [16] to get detailed information about a commit. Since GH Archive stores the commit hash and the commit message as well, we could filter on bug-fixing commits in this step. All commit messages containing one of the following keywords are identified as a bug-fix: `["fix", "solve", "bug", "issue", "problem", "error"]`. The same patterns are used in the work of Tufano *et al.* [42] and a similar approach in [13]. Next, files are being fetched that are affected by the commit. Using GitHub API, we were able to filter out non-JavaScript files (files with not ".js" extension) and download the before- (i.e. p_{buggy}) and after (i.e. p_{fixed}) state of it. At the end of this phase we identified 18736 files. These files served as the basis of our preprocessing step descibed in Section 3.1.

Figure 2: Loss curve

2.2 Model

Although not in details, in this section we describe the GPT-2 model [2]. The original Generative Pre-trained Transformer, or in short GPT, model was published in 2018, a descendant and improved version of this is GPT-2. It's architecture is based on the Transformer, which is an attention model - it learns to focus attention on the previous words that are the most relevant to the task at hand: predicting the next word in the sentence. Since it was designed to generate sentences, it has fixed input and output dimensions. Since it is a statistical architecture, no linguistic information is hardcoded into it, by fine-tuning it, the model can learn to write source code as well.

3 EXPERIMENT SETUP

3.1 Preprocessing

We formed the dataset so it is suitable to be the input of our model. First every comment is being removed since they do not affect the execution. Then we split the 18736 mined JS files into 16863 training files and 1873 test (to predict candidates) files. For training we used code snippets from all 16863 training files (interval of tokens around the bug location from the fixed file) and from the 1873 test files we generated candidates for 1559 files only because the bug environment is not always adequate. The training data source code is preprocessed from the start of the file until we reach the bug location and additional 10 lines (so the fix is also included). Note that for training we picked the fixed version of the files, so the model only learns correct code and not buggy ones. For evaluation purposes from the evaluation split we created 1 file for generation which contains all tokens from the start of the file till the bug location (from the original buggy file but this part of the original buggy and fixed files are essentially the same) and 1 for evaluation for each file, the latter described file for evaluation consists 3 lines after the bug location so we can use this file for comparing generated patches to the target. Since the model takes input sequences of fixed length (see Section 2.2) of text, we had to add a post-processing step before the input can be fed. For training purposes as described previously, the code chunks are extracted from the beginning of the file until the bug location + 10 lines. This data is of course not of equal length, so the last 2040 tokens are taken from this chunk. The input is then saved to a file where every line consists of 2040 tokens and it will be fed to the model line-by-line. Keep in mind that the preprocessing steps are different for training and prediction and in later sections we are going to further reduce the number of tokens fed to GPT-2 and the number of these tokens are going to depend on whether we are using the model for training or inference.

3.2 Training

The GPT model's input is a simple text file in natural language processing (there is no target like in classic machine learning, the model itself can learn on plain text to generate additional text). In our paper the model's input is a simple text file (later train.txt) as in NLP but instead of text we train on code. In our train.txt in each line we have a part of a code (interval of tokens around the bug location from the fixed file) belonging to one of our training sample. As we described earlier in each line of our train.txt there are 2040

tokens (most of them before the bug and the last part of the interval is the fix itself and the following tokens). We further reduce these 2040 tokens to 768 tokens in a way that we delete the first tokens only so we still have the tokens after the bug location (developer fix) and the tokens right before the bug.

For our experiment we used pretrained GPT-2 to generate patches. We trained our model on Nvidia GeForce RTX 3090 and the batch size was 7 due to the limited GPU memory. The training took 3 hours 13 minutes. As tokenizer we used GPT-2 pretrained tokenizer with additional tokens: bos_token= '<|startoftext|>, eos_token= '<|endoftext|>', pad_token= '<|pad|>'. For the training we built a custom pytorch dataset and used it for our custom data loader. As optimizer we used AdamW optimizer and used liner learning rate scheduler with warmup (warmup_steps = 1e2, total_steps = len(train_dataloader) * epochs). As early stopping parameter we used patience 3. We set 100 as maximum number of epochs. In figure 2 we can see how quick our model learned and we can also see how the early stopping helped us to reduce training time. Additional parameters of the GPT-2 model: top_k=50, top_p=0.8, do_sample=True, max_length=1024, num_return_sequences=1.

As we described earlier the dataset was split into 2 categories: train and evaluation (we use the fine-tuned GPT model to generate patches). The train dataset was then transformed into the aforementioned train.txt which contains line-by-line the code chunks around the bug location of the fixed original file. We further split the lines of train.txt (each line can be interpreted as a training sample) into 90% train and 10% validation set. The early stopping was based on the results of the validation set. Note that despite the data leakage is still possible we made sure to reduce the chances of it. To do so in train.txt we didn't train on whole files but just on token intervals of the original fixed files around the bug location.

3.3 Patch Generation

First we expanded the GPT-2 model's generate function so that the function only returns a list of lines of the generated text (patch) itself without the input given to the model. For every bug we called our generate function 10 times which means we generated 10 patches for every bug. Since we focused on one line fixes, we set the generated token length to 124 (the GPT-2 model's original generate function was set to 1024 token length and 900 tokens were used as input for our model (1024-900=124)). The expanded generate function returns 124 tokens in a list of lines so the number of generated lines vary by bug and generation. In every generation we compared each generated line to our target text (one-liner patch to be generated), which means for every bug we have 10*x candidate one-liner patches, where x corresponds to the generated 124 tokens divided by the number of line separators in our generated text, and 10 comes from the number of generations. For every bug we saved all of the above mentioned candidate lines with the corresponding generation number and line number(the line number of the generated one-liner candidate). In these files we also saved the closest candidate by edit distance. Finally we created one txt file with all target_txts with the adequate closest candidate one-liner patch by edit distance.

```
//2
rendererSync=require('./extend').rendererSync.list()
//113
queryParams=util.parseQueryParams(location.search)
//115
it('should invoke the callback 404',function(done)
//180
expect(console.log.calledWith(sinon.match('Name'))).
    be.true;
//261
fs.readdirSync(__dirname + '/../controllers').
    forEach(function(name){
//354
for (let i = 0, len = args.length; i < len; i++) {
```

Listing 1: Examples of correct fixes.

4 RESULTS

In the previous section we described how we created candidate files for each bug. The evaluation of the results was based on these candidate text files where all candidates can be found for each bug. We compared each of these candidate patches to the target text by edit distance. In this section we analyze the results in two subsections: Quantitative Evaluation & Qualitative Evaluation. As you can see later the model is able to correctly infer variable names, this is because we did not train in a cross-project way. The pre-trained GPT-2 model was pre-trained on many natural next and code including javascript, so the pre-trained model without fine-tuning can also be used to generate patches but it will not be able to predict variable names accurately. The model is expected to be retrained on each project before generating patches to accurately predict variable names. In addition we also have to mention that although it is not simple, it is possible to extract pre-training data from language models, this is a distinct research topic. Some language models were trained on private datasets and some researchers managed to extract sensitive personal data from them (names, phone numbers, email addresses, IRC conversations etc.) [5]. For this reason data leakage is possible in all research where authors use pre-trained language models like GPT, BERT etc. (except if they do not use the pre-training + fine-tuning way of training but retrain the whole model from scratch which requires extremely efficient hardware and a lot of time. Or the pre-trained data is publicly available and checked by the authors). Because of the aforementioned statement some level of data leakage is possible in most APR research. We didn't filter our dataset by time so there is an additional possibility of data leakage but as we described in Section 3.2 we made sure to reduce the possibility by training only on a small interval of tokens around the bug location.

4.1 Quantitative evaluation

During our quantitative evaluation we tried to be as strict as possible. First we considered exact matches only, which is a lot stricter case than in real life scenario. For exact matches we did not accept identical patches where the model generated different white spaces than the original fix. To address this problem we evaluated the results with different edit distances which is a fair estimation for these cases and also show that even when the predicted line were not exactly perfect, it was not so far from the expected result. To be

precise, the edit distance is calculated between p_{fixed} = the patch created by a developer and GPT2(p_{buggy}) = the output of the model for the buggy program. In our experiment we generated patches for each bug 10 times and considered each generated line as candidate patch. Apart from that we also generated patches in multiple generations, but these generations are aggregated in a sense that the same patch cannot be generated twice. The motivation behind this approach is that we were interested whether it is more likely to generate the correct patch in separate generation or in a single generation but considering multiple lines.

In Table 1 we can see that the model managed to generate the correct patch out of all correct patches in the first generated line nearly 50 percent of the times(1. gen 1 line: 126, 5. gen 10 line: 269) and in some cases the correct patch could be found later on. This table indicate how powerful GPT-2 is in automatic code repair. We would like to point out that the more time we generate patches the less likely it is to find a new correct patch. This is due to the parameter settings of the original GPT-2 generate function. Fellow researchers can make experiments to set these parameters depending on their available time and resources.

In Table 1 we can observe that the first line in the first generation (upper left corner of the table) was accurate in 126 cases, which is 8.08%. On the contrary if we consider not just the first line but the predecessing 5 lines, the accuracy rises up to 12.89%. This of course is natural since the model has more space of guesses, but it is nice to see that the GPT is indeed able to learn the fix environment. Considering the top 5 and top 10 lines of the first generation, the number of correct patches did not rise the accuracy values that much(from 12.89% to 13.73%), which confirms the effectiveness of GPT-2 architecture.

We made additional experiments with more generations (10) but as we described earlier, the higher the number of the generation gets the less likely it is to find new correct patches and the number of total candidates grows rapidly which makes manual evaluation harder for developers. We think that the parameter settings(top_k, top_p) of the GPT-2's original generation function and the number of generations(5) and line numbers(max: 10) is optimal in a sense that the generated candidate patches are easily supervisable by developers.

4.2 Qualitative evaluation

Beside the fully identical patches we found several patches which were the same except for some white space characters and we found a lot of nearly identical correct patches. During our manual evaluation of the closest patches by edit distance we also observed that the model generates the environment variables (bug environment) very accurately, which is one of the biggest challenges in non language model based approaches. We created a text file where all expected patch (target) and generated closest patch can be found for each bug.

In **Listing 2** there is a code snippet of this text file containing incorrect patches only. We can see that there are a lot of patches which are nearly identical to the expected result. In the text file containing these results many NULL value can be seen (as in **Listing 2**, with the bug id 39). This means we could not generate any patch for the given target text, there is an example of this case in the above referenced code snippet. In this code snippet we can also find patches where there is only difference in white spaces(patch with id 178), in some cases these white space differences make our patch not compilable and in some cases they do not cause any issue at all. In bug number 135 the only difference is that the model generated == instead of ===, which is more permissive because the latter does not require the datatype of the two operands to be the same. Note that this distinction does not exist in many programming languages (e.g. Java, C), from this point of view, it seems more difficult to fix JavaScript. In bug example number 58 we can see that GPT-2 can generate regular expressions more or less accurately.

We showcase another code snippet about correct patches on **Listing 1**. Among these examples we can observe that our model is able to generate `for` loops and other complex patches requiring knowledge of the adequate name of variables and objects, the model is also able to generate human readable error messages.

The text file containing all comparison between the target text and the generated closest candidate patch is available in our GitHub repository.

```
//17
expect(console.log.calledWith(simon.match('Date'))).
    be.true;
expect(console.log.calledWith(sinon.match('Date'))).
    to.be.true;
//18
var _ = require('lodash');
var _ = require(\'lodash\');
//63
var Moment = require('./types/moment');
var Moment = require("./types/moment');
//135
if (config.archive == 2){
if (config.archive === 2){
//178
describe('Manager', function() {
describe('Manager', function () {
//47
var request = mockRequest({ method: method.
    toUpperCase(), uri: { path: path }})
var response = mockRequest({ method : method.
    toUpperCase(), uri : { path : path }})
//58
return pattern.replace(/[\/\\][^\/\\]*\*.*$/, '')
return pattern.replace(/\\/[^\\/]*\\*.*$/, '');
//187
var VERSION = require('./constants').VERSION;
var VERSION = require('../constants').VERSION;
//229
var LINK_TAG = '<link type="text/css" href="%s" rel
    ="stylesheet">';
var LINK_TAG_CSS = '<link type="text/css" href="%s"
    rel="stylesheet">';
//39
var util = require('../../util');
NULL
```

Listing 2: Examples of incorrect fixes.

Table 1: Percentages of correctly fixed candidates using 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	126/1559	167/1559	181/1559	201/1559	253/1559	276/1559	214/1559	270/1559	300/1559
#2	140/1559	181/1559	195/1559	222/1559	277/1559	301/1559	237/1559	297/1559	330/1559
#3	151/1559	194/1559	211/1559	236/1559	295/1559	324/1559	255/1559	320/1559	361/1559
#4	153/1559	197/1559	214/1559	243/1559	304/1559	335/1559	263/1559	332/1559	376/1559
#5	155/1559	204/1559	221/1559	248/1559	318/1559	350/1559	269/1559	350/1559	398/1559

Results of the GPT-2 model to generate patches automatically. In each generation the model created a list of patches. 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).

5 RELATED WORK

In this work we used our own dataset to create the train-test-evaluation set of data for our model, although there are others available. Defects4J [23] is a popular dataset consisting 395 Java bugs. The ManyBugs [28] dataset contains bugs written in C - it were used to evaluate many well-known APR tools (Genprog [44], Prophet [29], etc.). Bugs.jar [40] is another well-known dataset, which is comprised of 1,158 Java bugs and their patches. From all of these we could create our training data, the choice is arbitrary. A few datasets of larger-scale is also available publicly, but the format of these are not suitable for our experiments. The CodRep [7] dataset aims at being a common playground on which the machine learning and the software engineering research communities can interact. It contains 58,069 one-liner commits. A more recent work of Karampatsis *et al.* [24] introduce a dataset of similar size consisting of 153,652 single-statement bugs mined from open-source Java projects. The seminal work of Tufano *et al.* [42] includes the creation of a dataset for Java program repair and evaluation an NMT (Neural Machine Translation) model on it. This work is also included in the CodeXGLUE benchmark [30] which includes a collection of code intelligence tasks and a platform for model evaluation and comparison. The CodeXGLUE team also operate a leaderboard of the best-performing tools, where an approach called CoTexT [36] comes first at the time of writing this paper.

CoTexT [36] is a pre-trained, transformer based encoder-decoder model that learns the representative context between natural language (NL) and programming language (PL). CoTexT follows the sequence-to-sequence encoder-decoder architecture proposed by [43]. They achieved state-of-the-art results on most of these tasks, including in code repair with an astonishing 0.226 accuracy value and 77.91 BLEU score. Other works (that are not included in the CodeXGLUE benchmark) also evaluated their approach on the dataset by Tufano *et al.* [42]. The most recent among these is DeepDebug [10], where the authors used pretrained Transformers to fix bugs automatically. Other than the described approaches several others exist Several such tools already exists, such as SequenceR [6], Hoppity [9], DLFix [46], CoCoNuT [31] or CURE [22]. Due to space limitations we won't describe these in detail since they are less related to our work.

In this paper our aim was to use the GPT-2 [2] architecture to repair bugs automatically. Although we did not achieve state-of-the-art results (although hard to compare because of the lack of publicly available datasets), to the best of our knowledge we used this model for this task first. GPT-2 was introduced in 2018 by OpenAI. Since then it has received a large amount of citations, using the model for diverse tasks. 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 [1], 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. Although these results are state-of-the-art performances, in the papers the authors did not use the GPT model for Automated Program Repair and to the best of our knowledge neither did others.

Since the original article of GPT-2, several works have investigated the capabilities and limits of the model [47]. 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]. Despite the fact that the latest descendant of the GPT model family writes better than many people [39], they were used less for software engineering tasks. In a recent work the authors introduce Text2App [20], that allows users to create functional Android applications from natural language specifications.

Other than data-driven approaches, using standard Generate and Validate (G&V) tools for Automated Program Repair is still widely used to this day. GenProg [44] 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]. PAR is a synthesis-based tool which leverages the knowledge of human-written patches [25]. It works in Java and repairs source code based on 10 predefined templates. In addition to completely general repair techniques, there are those that specialize only in certain error classes. An example is Nopol [45], which improves conditional control structures (if-then-else structure). Another such tool is Kali [37], which uses only deleting or skipping the source code to synthesize patches. 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. Of course, there are other approaches that tend to generate the fix from previous manual fixes [25, 26].

There are some web-based APR tools already with the aim to fix JavaScript bugs, but they are specific to special problems. Vejovis [35] 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], which focuses also on API calls, but primarily on the parametrization of these.

However data-driven repair approach does not create falsely repaired candidates (since the produced patch should be exactly the same as the developer fix), patch correctness is an important aspect of the future of program repair [3, 14]. In a recent study authors has pointed out, that the use of source code embeddings might solve this complex problem by suggesting patches, which are the most similar to the original program [8]. Nevertheless, in [15] authors highlighted that this issue is still an open question.

6 CONCLUSIONS

In this paper we used the GPT-2 medium model to fix programs automatically. First a dataset has been created from commits mined from GitHub. In the process we mined 18736 JavaScript files. From these files we created 16863 training samples for the GPT-2 model and we generated candidates for 1559 bugs. It was trained on correctly fixed code snippets and it's task is to predict patches for buggy source code. On this dataset the GPT-2 model was able to repair 126 programs on first try, while it generated correct patches in 269 cases if it had multiple chances to do so. Based on these results, we can conclude that while GPT was designed for Natural Language processing, it is also able to learn how to code and repair programs. We also concluded that larger models might achieve better results, in future work we plan to investigate these as well.

ACKNOWLEDGMENTS

The research presented in this paper was supported in part by the ÚNKP-21-3-SZTE and ÚNKP-21-5-SZTE New National Excellence Programs, by Project no. TKP2021-NVA-09 and by the Artificial Intelligence National Laboratory Programme of the Ministry of Innovation and the National Research, Development and Innovation Office, financed under the TKP2021-NVA funding scheme. László Vidács was also funded by the János Bolyai Scholarship of the Hungarian Academy of Sciences.

REFERENCES

[1] 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

[2] 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.

[3] Fatmah Yousef Assiri and James M. Bieman. 2017. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal* 25, 1 (mar 2017), 171–199. https://doi.org/10.1007/s11219-016-9312-z

[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] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Xiaodong Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *USENIX Security Symposium.*

[6] 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

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

[8] Viktor Csuvik, Deniel Horvath, Ferenc Horvath, and Laszlo Vidacs. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF).* IEEE, 18–25. https://doi.org/10.1109/IBF50092.2020.9034714

[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] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a Release History Database from Version Control and Bug Tracking Systems. *IEEE International Conference on Software Maintenance, ICSM* (2003), 23–32. https://doi.org/10.1109/ICSM.2003.1235403

[14] 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

[15] 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

[16] GHArchive 2021. GH Archive Official Website. https://www.gharchive.org.

[17] GitHub 2021. The 2020 State of the Octoverse. https://octoverse.github.com.

[18] GitHub REST API 2021. GitHub REST API Official Website. https://docs.github.com/en/rest.

[19] 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

[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] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. 2012. Remedying the Eval That Men Do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012).* Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/2338965.2336758

[22] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. (may 2021), 1161–1173. arXiv:2103.00073

[23] 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.

[24] 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

[25] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings - International Conference on Software Engineering.* IEEE, 802–811. https://doi.org/10.1109/ICSE.2013.6606626 arXiv:arXiv:1408.2103v1

[26] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 213–224. https://doi.org/10.1109/SANER.2016.76

[27] 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

[28] 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

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

[35] 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

[36] 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

[37] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

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

[39] Alec Radford, Jeffrey Wu, Dario Amodei, Jack Clark, Miles Brundage, Ilya Sutskever, Amanda Askell, David Lansky, Danny Hernandez, and David Luan. 2019. Better Language Models and Their Implications. , 12 pages.

[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] Stack Overflow 2021. Stack Overflow Developer Survey Results 2021. https://insights.stackoverflow.com/survey/2021.

[42] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology* 28, 4 (2019). https://doi.org/10.1145/3340544

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 2017-December (2017), 5999–6009. arXiv:1706.03762

[44] 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

[45] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811 arXiv:1807.00515

[46] 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

[47] 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

[48] 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.