Fine-tuning GPT-2 to patch programs, is it worth it?

Márk Lajkó$^{1,2[0000-0003-0674-1275]}$, Dániel Horváth$^{1,2[0000-0001-8855-921X]}$, Viktor Csuvik$^{1,2[0000-0002-8642-3017]}$, and László Vidács$^{1,2[0000-0002-0319-3915]}$

[1] University of Szeged, Department of Software Engineering
[2] MTA-SZTE Research Group on Artificial Intelligence
`https://www.sed.inf.u-szeged.hu`
`{mlajko,hoda,csuvikv,lac}@inf.u-szeged.hu`

Abstract. The application of Artificial Intelligence (AI) in the Software Engineering (SE) field is always a bit delayed compared to state-of-the-art research results. While the Generative Pre-trained Transformer (GPT-2) model was published in 2018, only a few recent works used it to SE tasks. One of such task is Automated Program Repair (APR), where the applied technique should find a fix to software bugs without human intervention. One problem emerges here: the creation of proper training data is resource intensive and requires several hours of additional work from researchers. The sole reason of it is that training a model to repair programs automatically requires both the buggy program and the fixed one in large scale and presumably in an already pre-processed form. There are currently few such databases, so teaching and fine-tuning models is not an easy task. In this work we wanted to investigate how the GPT-2 model performs when it is not fine-tuned for the APR task, compered to when it is fine-tuned. From previous work we already know that the GPT-2 model can automatically generate patches for buggy programs, although the literature lacks of studies where no fine-tuning has taken place. For the sake of experiment we evaluated the GPT-2 model out-of-the-box and also fine-tuned it before the evaluation on 1559 JavaScript code snippets. Based on out results we can conclude that although the fine-tuned model was able to learn how to write syntactically correct source code almost on every attempt, the non-fine-tuned model lacked some of these positive features.

Keywords: Automated Program Repair · Machine learning · JavaScript · Code Refinement · GPT-2 · fine-tune

1 Introduction

Recent researches in NLP led to the release of multiple massive-sized pre-trained text generation models like the the Generative Pre-trained Transformer. There are currently three versions of it (GPT-1,2,3), from which we used GPT-2. Although OpenAI, the original creator of the GPT family, did not make the implementation of the model publicly available, thanks to the efforts of the NLP and AI research community where are several open-access implementations of it. These are pre-trained on a very large corpus of data in a self-supervised fashion. Since GPT-2 is trained to guess the next word in sentences, the training process does not require any special data, it can be easily obtained from scrapping web pages from the internet(it was originally trained on the text from 8 million websites). It is known that unfiltered data from the web is far from neutral, and the OpenAI team themselves pointed out that *"...GPT-2 do not distinguish fact from fiction, we don't support use-cases that require the generated text to be true..."* and *"...GPT-2 reflect the biases inherent to the systems they were trained on..."* [23]. Although this training procedure have some limitations, the outcome of it made the GPT family famous by writing stories about talking unicorns [2]. There are no fundamental algorithmic breakthroughs concerning GPT-2, the original model was essentially scaled-up, resulting a model with 10x more parameters than the original. Although GPT-2 is not the latest GPT model we hypothesize that the results would be roughly the same with more-recent model variants as well. 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 ease with which the GPT family can be used for a completely new unseen task is thrilling, without training for a single epoch. This combined with the availability of cheap computing capacities has led many software engineers to use these models without any special background knowledge [32]. On the other hand, fine-tuning a model requires more computational power and also competent people. While the pre-trained models are usually okay for experiments, for real production scenarios fine-tuning to the downstream task (e.g. sentiment detection, dialogue response generation, code completion, etc.) is usually recommended. This is especially true for those special cases when the downstream task is rather specific or it's domain differs from the one it was trained on. Although it is true that the training data of GPT-2 contains source code as well, natural language is present in the majority. Automated Program Repair is such a downstream task where fine-tuning might worth it, since in it the input and the output of the model is source code. The goal of it is that by given a buggy program the model should automatically create a patch for it without human intervention. This so-called patch is considered to be correct when it is syntactically identical (except for white-spaces) to the developer fix. This criterion is rather strict, 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 [17] often occurs, resulting in inadequate patches. Although there are some approaches that tried to tackle with this problem [8, 3, 5], the question of patch correctness is considered to be still open [9].

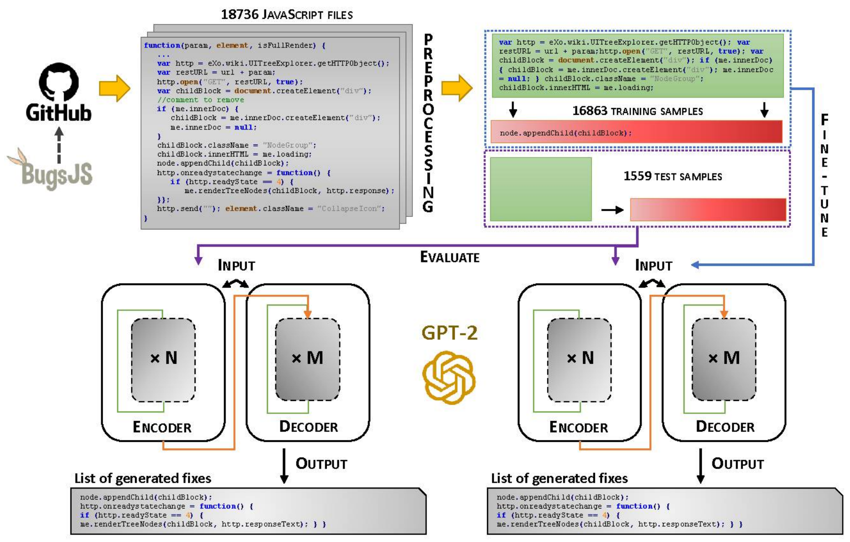
Encouraged by the excellent recent results of data-driven APR approaches [15, 4, 30, 21, 5], in this work we wanted to investigate whether is it worth fine-tuning the GPT-2 model. At the time of writing this article the top three approaches are CoTexT [22], PLBART [1] and DeepDebug [5]. Although none of these approaches use the GPT-2 model, their operating principle is similar. From previous work we know that the fine-tuned GPT model is able to repair programs automatically, although it is of question what is the performance of the raw pre-trained model on the same task. We used the GPT-2 implementation of the Hugging Face [14] community. We fine-tuned the model on JavaScript [26] samples and evaluated it, and also used the pre-trained version out-of-the-box and simply evaluated the test data on it. The choice of JavaScript is arbitrary, although it is the de-facto web programming language globally and the most adopted language on GitHub [10], the study could be executed on any other languages as well.

To be able to fine-tune 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. On the other hand, the pre-trained model was not fine-tuned, simply evaluated on the test set. Based on our experiments the fine-tuned GPT-2 was able to repair 126 programs on first try, while when no fine-tuning was applied only 10. On the other hand, when the non-fine-tuned model had more chances to generate a patch, it was able to generate fixes in 269 cases.

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 2.1 and Section 3.2 describe the preprocessing and fine-tuning 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.

2 Approach

In Figure 1 we depicted the high-level approach we present in this paper. First, JavaScript files are being fetched from GitHub and stored locally. Afterwards these files are preprocessed to form samples that can be fed to the GPT-2 model. These samples form a (p_{buggy}, p_{fixed}) tuple, where p_{buggy} is the state of the program before the code change, while p_{fixed} is the program after the patch has been 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. From the retrieved 18736 JS files we extracted 18422 samples (18422 (p_{buggy}, p_{fixed}) pairs). These tuples are next split into two separate parts: training and test samples. The training samples are used to fine-tune the GPT-2 model, while the test samples are for evaluation. We conducted two experiments: (1) in which we did not fine-tune the GPT-2 model, just evaluated the pre-trained model on the test samples and (2) first the model is being fine-tuned and next it is being evaluated on the same test set as the non-fine-tuned version. In both cases the output is a list of the generated patches, since on multiple runs the model gives back different results. Taking advantage of this, we handled the



```
function(param, element, isFullRender) {
   ...
   var http = eXo.wiki.UITreeExplorer.getHTTPObject();
   var restURL = url + param;
   http.open("GET", restURL, true);
   var childBlock = document.createElement("div");
   //comment to remove
   if (me.innerDoc) {
      childBlock = me.innerDoc.createElement("div");
      me.innerDoc = null;
   }
   childBlock.className = "NodeGroup";
   childBlock.innerHTML = me.loading;
   node.appendChild(childBlock);
   http.onreadystatechange = function() {
      if (http.readyState == 4) {
         me.renderTreeNodes(childBlock, http.response);
   }};
   http.send(""); element.className = "CollapseIcon";
}
```

```
var http = eXo.wiki.UITreeExplorer.getHTTPObject(); var
restURL = url + param;http.open("GET", restURL, true); var
childBlock = document.createElement("div"); if (me.innerDoc)
{ childBlock = me.innerDoc.createElement("div"); me.innerDoc
= null; } childBlock.className = "NodeGroup";
childBlock.innerHTML = me.loading;
```

```
node.appendChild(childBlock);
```

```
node.appendChild(childBlock);
http.onreadystatechange = function() {
if (http.readyState == 4) {
me.renderTreeNodes(childBlock, http.responseText); } }
```

```
node.appendChild(childBlock);
http.onreadystatechange = function() {
if (http.readyState == 4) {
me.renderTreeNodes(childBlock, http.responseText); } }
```

Fig. 1. The high-level approach of patch generation.

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. At the end, these ordered lists are compared against the developer fix and thus it can be easily decided whether the patch is correct or not. Finally we calculate the percentage of correctly patched programs in both cases and analyze that.

2.1 Dataset

For the experiments we created our own dataset based on Github projects, also contained by BugsJS [12], which contains reproducible JavaScript bugs from 10 open-source projects. However we did not restricted ourselves to bugs only. In this dataset one can find changes containing bug fixes, code refinements, refactorings, etc. For the sake of simplicity, in the paper we refer to these code-refinements, as bugs. The dataset contains both single- and multi-line bugs as well. The detailed description of these code-refinements are beyond the scope of this research, but the interested reader is encouraged to take a look at the original paper or the aforementioned projects themselves, for further details. We retrieved commits using the GitPython [11] package. First we downloaded the repository, then iterated through commits one-by-one, collecting code changes for files with *.js* extension. At the end of this phase we identified 18736 files. These files served as the basis of our preprocessing step.

Preprocessing From the mined JavaScript files every comment is being removed since they do not affect the execution. Then we split the 18736 files into 16863 training and 1559 test samples (some of the files were ignored because the code-change environment is not always adequate). Each file is preprocessed from the start until we reach the modified location and additional 10 lines. Note that for fine-tuning we picked the fixed version of the files, so the model only learns refined code and not its previous version. The evaluation samples on the other hand contain the change as well, so it can be compared to the code generated by the model. Since the model takes input sequences of fixed length, we had pad these sequences to be of equal length (2040 tokens). 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 fine-tuning and prediction depending on whether we are using the model for training or inference.

3 Experiment setup

3.1 GPT-2

The original Generative Pre-trained Transformer, or in short GPT, model was published in 2018, a descendant and improved version of it is GPT-2 [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. This property allows it to generate not just natural language but source code as well. Although the pre-trained model is suitable for some experiments tasks, it is adviseable in special cases to fine-tune it for downstream tasks.

3.2 Fine-tuning

Text sequences serve as the input of the GPT-2 model, which is usually plain English text in natural language processing. While fine-tuning the model there is no target like in classic machine learning, the model itself can learn on plain text to generate additional text (while it was trained its goal was to predict the next word in a sequence). In this paper the models input is a simple text file but instead of natural language we train on source code.

In our experiments we used HugginFace [14] implementation of the pre-trained GPT-2 model. It was used in two scenarios: generation without and with fine-runing. The fine-tuning took place on am Nvidia GeForce RTX 3090 with batch size of 7 due to the limited GPU memory. Fine-tuning took 3 hours and 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 linear learning rate scheduler with warmup (warmup_steps = 1e2, total_steps = len(train_dataloader) * epochs). As early stopping parameter we used patience

Table 1. Results of the GPT-2 model to generate patches automatically. The upper table shows the results of the fine-tuned model and the lower table shows the results of the pre-trained model. 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).

	GPT-2 fine-tuned								
	Top_1			Top_5			Top_{10}		
Generation	# EM	# ED_5	ED_{10}	# EM	# ED_5	ED_{10}	# EM	# ED_5	ED_{10}
#1	8.08	10.71	11.61	12.89	16.23	17.7	13.73	17.32	19.24
#2	8.98	11.61	12.51	14.24	17.77	19.31	15.2	19.05	21.17
#3	9.69	12.44	13.53	15.14	18.92	20.78	16.36	20.53	23.16
#4	9.81	12.63	13.73	15.59	19.5	21.49	16.87	21.3	24.12
#5	9.94	13.09	14.18	15.91	20.4	22.45	17.25	22.45	25.53
	GPT-2 pre-trained								
#1	0.64	1.15	2.5	1.22	1.86	5.52	1.48	2.25	7.12
#2	0.71	1.41	3.08	1.28	2.37	6.86	1.54	2.82	8.92
#3	0.77	1.48	3.78	1.35	2.69	8.21	1.67	3.34	10.78
#4	0.83	1.67	4.17	1.41	2.95	9.17	1.73	3.66	11.93
#5	0.83	1.67	4.17	1.41	2.95	9.17	1.73	3.66	11.93

3. We set 100 as maximum number of epochs. Additional parameters of the GPT-2 model: top_k=50, top_p=0.8, do_sample=True, max_length=1024, num_return_sequences=1.

3.3 Patch Generation

First we expanded the GPT-2 models generate function so that the function returns a list of lines of the generated code without the input given to the model. For every bug we called this generate function 10 times which means we generated 10 patches for every input sample. The expanded generate function returns 124 tokens each time it is being called, thus the number of generated lines vary by sample and generation. In every generation we compared each generated line to our target text, 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. By doing so we ended up with a lot of candidates, but in an ordered manner: the first line in the first generation is treated as the "first guess of the model", so it has a privilege in some sense. As we move forward in generations and lines, the less focus is on the candidate. This patch generation process is executed for both the pre-trained and the fine-tuned model which are evaluated separately. In the next sections we compare the results in more details.

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 are going to compare the results of the pre-trained and the fine-tuned version of the model then analyze the results manually. The model, the preprocessing and the generation of the candidates are essentially the same the only difference is whether we are using fine-tuning or not. Before our experiment we expected the fine-tuned model to be much more effective and we also supposed that the fine-tuned version was going to be a lot better in predicting variable names as well.

```
//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++) {
```

```
//2
let renderer = renderer
    .render.init()

//113
var errors = error.
    errors();

//115
it('should invoke the
    callback when
    complete', function
    (done){
//180
console.log('Called')


//261
module.exports.forEach
    = function(name,
    data) {



//354
if (args.length > 0) {
```

Listing 1: Examples of correct fixes generated by the fine-tuned GPT-2 model (left) and fixes for these samples generated by the pre-trained model (right).

For our quantitative evaluation we used edit distance. Our strictest condition was to generate identical patches, in this case we didn't accept patches that would be identical without white spaces. We evaluated the results with different edit distances so that we can see how close were our candidates to the correct patch. For both cases (pre-trained and fine-tuned model) we generated patches for one bug 5 times and considered each generated line as a candidate. The results are aggregated by generations so in each following generations there are greater or equal number of correct patches, it is also obvious that the more generations we use the less likely we are to find new additional correct patches (for example after the first generation we are more likely to find correct patches than after the 4th generation).

```
//12
if (2 == arguments.
    length) {
//213
register('data');
//214
if (browser) {
//720
else {
//914
'',
//1097
app.get('/movie',
    function(req, res){
```

```
//12
return this;

//213
''
//214
} : () => {
//720
else {
//914
{

//1097
NULL
```

Listing 2: Examples of correct patches generated by the pre-trained model (left) and fixes for these samples generated by the fine-tuned model (right)

Observing Table 1 we can clearly see that the fine-tuned model performed much better than the pre-trained one (upper table: fine-tuned, lower table pre-trained). The pre-trained model was able to generate 10 correct patches in the first generation and the first line, while the fine-tuned model was able to generate 126 correct patches in the first generation and the first line. As we mentioned earlier the results are aggregated by generation so in each generation the number of found identical patches can not be less than before. From the two observed tables we can also see that the total generated identical patches are 27 for the pre-trained model and 269 for the fine-tuned one. We can also observe that the less strict we are concerning the number of candidate lines per generation the better results we get. It is also clear that the more additional candidate lines we consider per generation better the results get. As we stated above we generate more than just the first line after the input code snipped given to our model (these are the candidate lines) despite the location of the one-liner bug is right after the input code snippet. Because gpt-2 is capable of understanding the input code snippet the first candidate line is the most likely to be a correct patch and as we check for later candidate lines the less likely the model is to generate the identical patch.

Next we analyze the generated patches manually and we are going to see that both the pre-trained and the fine-tuned model are really good at generating correct variable names and human readable error messages, although the fine-tuned model can generate more complex patches. On **Listing 1** we can see the identical candidate patches generated by the fine-tuned model compared to the fixes generated by the pre-trained model. Among these correct patches we can see that the fine-tuned version was able to generate more complex identical patches than the pre-trained one. Both the pre-trained and the fine-tuned model can predict correct variable names, and there are also examples of the fine-tuned model generating human readable error messages. On **Listing 2** we can find the identical fixes of the the pre-trained model compared to the fixes generated by the fine-tuned model. It seems like that the pre-trained model only generated easy fixes (i.e. short patches). Interestingly the fine-tuned model seems to "over learn" on these simple cases. In future research we plan to investigate it in more details. On the other hand on **Listing 3** we listed only incorrect patches generated by the pre-trained model, but as can be seen they are really close to the target. As we described earlier our evaluation was strict so we didn't consider these patches correct. Knowing the fact that the pre-trained model was not trained on any hexo (JS project) files, we can state that the reason for generating so accurate regular expressions (bug 37) is not data leakage. Note that the patches generated by the pre-trained model make sense in most of the cases, even a developer cannot decide whether it is correct or not without knowing the context.

```
//37
var rSwigVar = /\{[\s\S]*?\}\}/g;           //Pre-trained
var rSwigComment = /\\{#[\\s\\S]*?#\\}/g;  //Fine-tuned
var rSwigComment = /\{#[\s\S]*?#\}/g;      //Target
//270
Resolver.prototype.resolveTarget=function() // pre-trained
Resolver.prototype.getDependencies=function() // fine-
    tuned
Resolver.prototype.getDependencies=function() // target
//1065
app.get('/error', function(err, res){ // pre-trained
app.get('/error', function(req, res){ // fine-tuned
app.get('/error', function(req, res){ // target
//1096
var paths = require('path'); // pre-trained
var pathspec = require('pathspec'); // fine-tuned
var pathspec = require('pathspec'); // target
```

Listing 3: Patches generated by the observed models that are nearly identical to the developer change.

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 [16] is a popular dataset consisting 395 Java bugs. The ManyBugs [18] dataset contains bugs written in C - it were used to evaluate many well-known APR tools (Genprog [28], Prophet [19], etc.). Bugs.jar [25] is another well-known dataset, which is comprised of 1,158 Java bugs and their patches. Hovewer, despite its popularity, none of the aforementioned datasets contain bugs for JavaScript. The seminal work of Tufano *et al.* [27] 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 [20] 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 NSEdit [29] comes first at the time of writing this paper.

NSEdit [29] is a pre-trained, transformer based encoder-decoder model that predicts an editing sequence that can fix bugs. The encoder parts are initialized using weights from the pre-trained CodeBERT [7] model, while the decoder weights are initialized using CodeGPT [20]. They achieve an astonishing result of 24.04% fix rate on the small -, and 13.87% on the medium CodeXGLUE [20] dataset.

In this paper our aim was to use the GPT-2 [2] architecture to repair bugs automatically. While 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. In the previously mentioned CodeXGLUE benchmark [20] 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. Although these results are state-of-the-art performances, in the papers the GPT model was not used for Automated Program Repair.

Since the original article of GPT-2, several works have investigated the capabilities and limits of the model [31]. 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 [6]. Despite the fact that the latest descendant of the GPT model family writes better than many people [24], they were used less for software engineering tasks. In a recent work the authors introduce Text2App [13], that allows users to create functional Android applications from natural language specifications.

6 Conclusions

Although it is known from the literature that GPT-2 can be used for coherent natural text generation without fine-tuning, it is little known whether its source code generation capabilities improve significantly with fine-tune or not. To follow up on this issue, we evaluated both the pre-trained (non fine-tuned) and the fine-tuned GPT-2 model on a dataset that has been created from Github commits. The fine-tuned model was trained on 16863 JavaScript samples, while the pre-trained model was used out-of-the-box. The models were evaluated on the same set of test samples, and it turned out that both are able to generate syntactically and semantically correct source code. While the fine-tuned model was able to correctly refine 126 programs on first try, the pre-trained only in 10 cases. When both models had multiple chances to generate patches, the fine-tuned generated correct pathes in 269 cases, while the pre-trained in 27 cases. Although the GPT-2 model was designed for Natural Language processing and its training data mostly consists of natural language texts, based on the results, we can conclude that without fine-tuning it is still able to generate source code as well. On the other hand, it seems that fine-tuning it to this downstream task boosts its performance significantly, thus in this special case it did worth the extra computational power. We also concluded that both the pre-trained and the fine-tuned model are effective for using existing variable names, creating human readable error messages and creating reasonable complex regular expressions.

Acknowledgements

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. Ahmad, W., Chakraborty, S., Ray, B., Chang, K.W.: Unified Pre-training for Program Understanding and Generation pp. 2655–2668 (mar 2021). https://doi.org/10.18653/v1/2021.naacl-main.211
2. Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, I.S.: [GPT-2] Language Models are Unsupervised Multitask Learners. OpenAI Blog **1**(May), 1–7 (2020)
3. Csuvik, V., Horvath, D., Horvath, F., Vidacs, L.: Utilizing Source Code Embeddings to Identify Correct Patches. In: 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF). pp. 18–25. IEEE (2020). https://doi.org/10.1109/IBF50092.2020.9034714
4. Dinella, E., Dai, H., Brain, G., Li, Z., Naik, M., Song, L., Tech, G., Wang, K.: Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. Tech. rep. (2020)
5. Drain, D., Wu, C., Svyatkovskiy, A., Sundaresan, N.: Generating bug-fixes using pretrained transformers. MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021 pp. 1–8 (jun 2021). https://doi.org/10.1145/3460945.3464951

6. Elkins, K., Chun, J.: Can GPT-3 Pass a Writer's Turing Test? Journal of Cultural Analytics (sep 2020). https://doi.org/10.22148/001c.17212
7. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: Codebert: A pre-trained model for programming and natural languages (2020). https://doi.org/10.48550/ARXIV.2002.08155, `https://arxiv.org/abs/2002.08155`
8. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: A survey. IEEE Transactions on Software Engineering **45**(1), 34–67 (2019). https://doi.org/10.1109/TSE.2017.2755013
9. Gazzola Luca, Micucci Daniela, M.L.: Automatic Software Repair: A Survey. IEEE Transactions on Software Engineering **45**(1), 34–67 (jan 2019). https://doi.org/10.1109/TSE.2017.2755013
10. The 2020 state of the octoverse. `https://octoverse.github.com` (2021)
11. Gitpython home. `https://gitpython.readthedocs.io/en/stable/` (2021)
12. Gyimesi, P., Vancsics, B., Stocco, A., Mazinanian, D., Beszedes, A., Ferenc, R., Mesbah, A.: BugsJS: A benchmark of javascript bugs. In: Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019. pp. 90–101 (apr 2019). https://doi.org/10.1109/ICST.2019.00019
13. Hasan, M., Mehrab, K.S., Ahmad, W.U., Shahriyar, R.: Text2App: A Framework for Creating Android Apps from Text Descriptions (2021)
14. Hugging face website. `https://huggingface.co` (2022)
15. Jiang, N., Lutellier, T., Tan, L.: CURE: Code-Aware Neural Machine Translation for Automatic Program Repair pp. 1161–1173 (may 2021)
16. Just, R., Jalali, D., Ernst, M.D.: 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. pp. 437–440. Association for Computing Machinery, Inc (jul 2014)
17. Le, X.B.D., Thung, F., Lo, D., Goues, C.L.: Overfitting in semantics-based automated program repair. Empirical Software Engineering **23**(5), 3007–3033 (oct 2018). https://doi.org/10.1007/s10664-017-9577-2
18. Le Goues, C., Holtschulte, N., Smith, E.K., Brun, Y., Devanbu, P., Forrest, S., Weimer, W.: The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. IEEE Transactions on Software Engineering **41**(12), 1236–1256 (dec 2015). https://doi.org/10.1109/TSE.2015.2454513
19. Long, F., Rinard, M.: Automatic patch generation by learning correct code. Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016 pp. 298–312 (2016). https://doi.org/10.1145/2837614.2837617
20. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., Li, G., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, M., Duan, N., Sundaresan, N., Deng, S.K., Fu, S., Liu, S.: CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. undefined (2021)
21. Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L.: 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**, 101–114 (2020)
22. Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., Ye, Y.: CoTexT: Multi-task Learning with Code-Text Transformer pp. 40–47 (may 2021). https://doi.org/10.18653/v1/2021.nlp4prog-1.5

23. Radford, A., Narasimhan, T., Salimans, T., Sutskever, I.: [GPT-1] Improving Language Understanding by Generative Pre-Training. Preprint pp. 1–12 (2018)
24. Radford, A., Wu, J., Amodei, D., Clark, J., Brundage, M., Sutskever, I., Askell, A., Lansky, D., Hernandez, D., Luan, D.: Better Language Models and Their Implications (2019)
25. Saha, R.K., Lyu, Y., Lam, W., Yoshida, H., Prasad, M.R.: Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. Proceedings - International Conference on Software Engineering pp. 10–13 (2018). https://doi.org/10.1145/3196398.3196473
26. Stack overflow developer survey results 2021. `https://insights.stackoverflow.com/survey/2021` (2021)
27. Tufano, M., Watson, C., Bavota, G., Penta, M.D., White, M., Poshyvanyk, D.: 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
28. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering. p. 364–374. ICSE '09, IEEE Computer Society, USA (2009). https://doi.org/10.1109/ICSE.2009.5070536
29. Yaojie, H., Xingjian, S., Qiang, Z., Lee, P.: Fix Bugs with Transformer through a Neural-Symbolic Edit Grammar
30. Yi, L., Wang, S., Nguyen, T.N.: Dlfix: Context-based code transformation learning for automated program repair. In: Proceedings - International Conference on Software Engineering. pp. 602–614. IEEE Computer Society (jun 2020). https://doi.org/10.1145/3377811.3380345
31. Zhao, T.Z., Wallace, E., Feng, S., Klein, D., Singh, S.: Calibrate Before Use: Improving Few-Shot Performance of Language Models (2021)
32. Zhuang, Y., Cai, M., Li, X., Luo, X., Yang, Q., Wu, F.: The Next Breakthroughs of Artificial Intelligence: The Interdisciplinary Nature of AI. Engineering **6**(3), 245–247 (mar 2020)