PATCH GENERATION WITH LANGUAGE MODELS: FEASIBILITY AND SCALING BEHAVIOR

Sophia Kolak, Ruben Martins, Claire Le Goues & Vincent J. Hellendoorn
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{sophiakolak,rubenm,clegoues,vhellendoorn}@cmu.edu

ABSTRACT

Large language models have shown a propensity for generating correct, multi-line programs from natural language prompts. Given past findings highlighting that bugs and patches can be distinguished by predictability according to simple language models, it is natural to ask if modern, large neural options lend themselves especially well to program repair without any calibration. We study this in the context of one-line bugs, by providing a series of models of varying scales (from 160M to 12B parameters) with the context preceding a buggy line in 72 Java and Python programs, and then analyze the rank at which the correct patch (and original buggy line) is generated, if at all. Our results highlight a noticeable correlation of model size with test-passing accuracy and patch ranking quality, and the propensity for especially the largest models to generate candidate patches that closely resemble (if not exactly match), the original developer patch.

1 INTRODUCTION

In recent years, language models have proven to be highly effective tools for a wide variety of code and text generation tasks Allamanis et al. (2018). A strong correlation between model size and performance has also emerged Kaplan et al. (2020); Chen et al. (2021); Austin et al. (2021), encouraging the use of multi-billion parameter models in research tools like codex (12B parameters Chen et al. (2021)) and GPT-3 (175B parameters Brown et al. (2020)). However, large language models do not achieve their performance improvements without cost. GPT3, co-BERT, and many other large language models powering popular new technologies have received both research and media attention for their environmental costs, security risks Pearce et al. (2021), and potentially biased training data Bender et al. (2021). Researchers, therefore, have a vested interest in understanding the extent to which a model's size impacts its performance. For instance Kaplan et al. (2020) evaluate the relationship between model size and loss, and Chen et al. (2021) study a similar relationship with respect to a model's success at a code generation task.

Here, we extend this work on the impact of language model size to a new domain: patch generation. This is a key part of Automated Program Repair (APR), for which language models can be highly useful by generating "natural", and ideally plausible, candidate patches for failing programs. As such, APR research is increasingly looking to language models for patch generation, yet, most current work focuses on learning from supervised datasets of buggy programs and their repairs. Datasets of real mistakes are typically limited in size (on the order of 100Ks of samples), so synthetic fault injection is often used to enrich these datasets Hellendoorn et al. (2020), which can lead to unrealistic and overly simplistic training samples. Long-standing evidence holds that even traditional, n-gram based language models of code can discriminate between buggy and correct lines Ray et al. (2016). In this paper, we revisit the use of language models for patch generation in the era of large language models.

To evaluate the potential in this space comprehensively, we include models spanning a wide range of sizes, including GPT-2 style models trained with 160M, 0.4B, and 2.7B parameters, and OpenAI's Codex (12B parameters Chen et al. (2021)). We point these models at a dataset of programs containing a known, single line bug (in both Java and Python), querying each for 100 one-line candidate

patches. We then evaluate key properties of the results, including the number of functionally correct patches, the number that matches the real-world patch exactly, and rates of syntactic similarity to the ground-truth patch. This simple series of experiments allows us to understand the base probability of a model generating a patch or reproducing a bug, as well as the variance of these trends with respect to scale. Our results show that overall, larger models are better patch generators, with significant gains in the number of successfully patched programs emerging between 2.7B and 12B parameters, and between 0.4B and 2.7B parameters for entropy rankings. We also observe that all models perform better at patch generation in Python than in Java, regardless of scale. Lastly, we find that larger models tend to generate more "natural" solutions than smaller ones, and are generally better at distinguishing between bugs and patches in their rankings.

2 METHODOLOGY

Our experiments use language models of different sizes to generate candidate patches for buggy Java and Python programs. We evaluate patch generation for four models ranging in size from 160M to 12B parameters. We begin by describing our data sources (Section 2.1); next, we describe how we prompt each model to generate candidate patches from that data (Section 2.2); and finally, we describe details of each model relevant for patch generation (Section 2.3).

2.1 DATA

We evaluate language models using the Quixbugs program repair benchmark Lin et al. (2017). This dataset contains 40 programs, each with a one line bug, implemented in both Java and Python (for a total of 80 buggy programs). For each buggy program, Quixbugs also provides a correct version of the program and a series of associated test cases. Originally, these programs were designed to challenge humans; the goal was for developers to quickly find and fix a one-line bug in the implementation of a classic algorithm. As such, each program is a stand-alone function or class.

We selected the Quixbugs benchmark for two reasons: first, because it provides analogous data across two popular programming languages (helping us generalize from our results), and second, because each program contains only a single line bug, which we assume has been localized, that requires a one line patch. We specifically sought out such bugs to study the relationship between the scale and patch generation capabilities of language models. That is, we are not assessing a language model's capabilities holistically across the entire process of program repair. Dynamic or test-driven program repair more generally comprises several stages: fault identification, fault localization, patch generation, and patch evaluation Le Goues et al. (2019). Although it is conceivable that a language model could contribute to any of these phases, we focus on their most promising usages (patch generation), and thus assume that the fault has already been localized Liu et al. (2020). Focusing on already localized one-line bugs and patches abstracts away the impact of other phases of the repair pipeline, and assures us that it is always *possible* for the model to generate a passing candidate program.

We note that while the Quixbugs data includes 40 programs, we excluded 4 programs from our analysis: `depth_first_search.py`, `reverse_linked_list.py`, `shunting_yard.py` and `wrap.py` (and their Java equivalents), because their "patches" merely involved deleting a line of code. Including bugs of this kind would complicate each of the metrics described above. Again, our goal is to study the impact of scale (as opposed to the quality of this particular methodology), so we chose to ignore these cases for the sake of simplicity and consistency.

2.2 PROMPTING TECHNIQUE & OVERVIEW

Figure 1 shows a high-level overview of our method for patch generation and evaluation. Phase 1 takes a correct and buggy version of the same program as input. Figure 2 shows an example of such an input from the dataset, *viz.* the correct and buggy version of `bitcount.py`. With the exception of one character in the highlighted line, the two programs are identical.

To represent the program as shown in phase 2, we need to determine where the buggy line is located. We do so by removing comments (if they are present) and then `diff`ing the buggy and correct files line by line. Because all Quixbugs bugs are one line long, the delta between files reveals the location

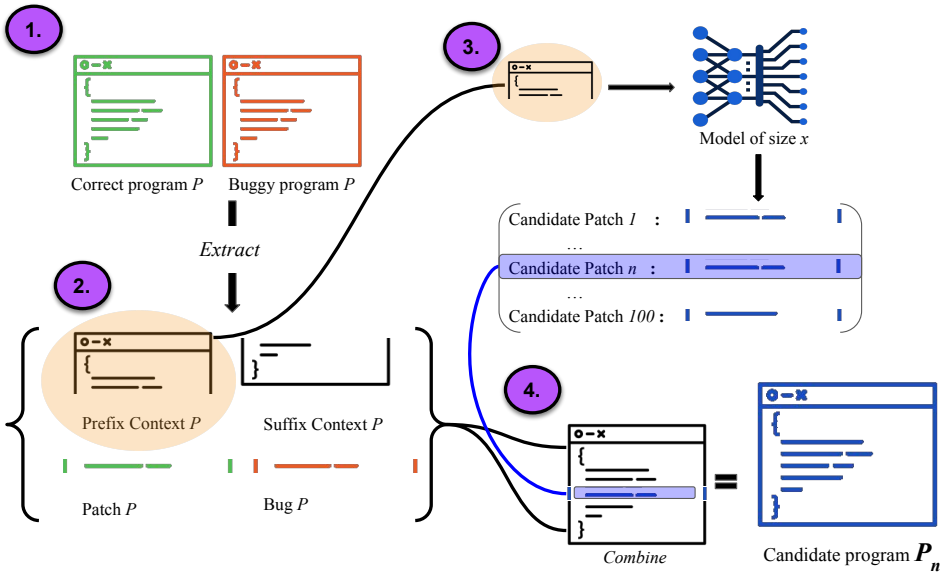

Figure 1: Visualization of the prompt extraction and patch generation procedure that is repeated for each of the models (GPT-2 160M, GTP-2 0.4B, GTP-2 2.7B, and GTP-3 12B). Each model generates 100 candidate patches per program in Java and Python (respectively).

correct `bitcount.py`

```
1 def bitcount(n):
2     count = 0
3     while n:
4         n &= n - 1
5         count += 1
6     return count
```

buggy `bitcount.py`

```
1 def bitcount(n):
2     count = 0
3     while n:
4         n ^= n - 1
5         count += 1
6     return count
```

Figure 2: Example of buggy and correct version of `bitcount.py` from the Quixbugs benchmark.

of the bug. In this figure, for instance, a diff clearly shows that the bug in bitcount is located at line 4. Using this information, we extract the lines before the bug, the bug, the patch, and the lines following the bug, which correspond to the four fields shown in phase 2. In `bitcount.py`, since the bug appears on line 4, we refer to lines 1-3 as the *prefix context*, to the buggy version of line 4 as *bug*, to the correct version of line 4 as *patch*, and to lines 5-6 as the *suffix context*. This information is then used to prompt the model and evaluate its responses in phases 3 and 4.

When prompting a model in phase 3, we give it only the prefix context as a prompt, meaning it has no knowledge of the bug, patch, or the suffix context. We then allow the model to generate 100 "patch candidates", and ensure that each candidate is exactly one line long by specifying the newline character as a stop token. In phase 4, we replace the buggy line with each of these 100 candidate patches, resulting in 100 candidate programs. We then record the candidate patch's similarity to the "ground-truth" patch, and whether it passes the provided test cases. We repeat phases 1-4 with each of the four models ranging from 160M to 12B parameters. Across all four models and all 72 program pairs (36 Java, 36 Python), there were 7,200 candidate patches and programs generated per model, and thus, 28,800 programs total were evaluated in this study.

Statement similarity was computed using BLEU score, a method originally designed for testing the quality of machine translation Papineni et al. (2002). While BLEU score is an imperfect metric overall Tran et al. (2019), it provides an approximation to syntactic similarity that may be informative when compared to test case passing behavior, as also done by Chen et al. (2021).

Table 1: Model Configurations

Model	Parameters	Layers	Hidden Dimension	Context Window	Tokens per batch	Training Steps
Small	160M	12	768	2,048	262K	150K
Medium	0.4B	24	1,024	2,048	262K	150K
Large	2.7B	32	2,560	2,048	262K	150K
Codex	12B	40	5,120	4,096	2M	50K*

*Codex was initialized from a GPT-3 model trained on a natural language corpus.

Functional plausibility was evaluated by running the candidate program on its tests, provided in the Quixbugs dataset. A patched program that passes all provided test cases is considered a *plausible* patch for a given bug. This measure does not perfectly capture correctness (see threats), but it is an important indication of the potential quality of the proposed patch, especially given the relative completeness of the QuixBugs tests.

Since all models return their candidates in order of likelihood, we processed the candidate programs as such: i.e., if the candidate at rank 3 for `bitcount.py` passed all test cases, then we do not test candidates 4-100 (though we still compute their similarity). In our results, we report the first-found (lowest ranked) candidate patch that is functionally plausible, if any, as well as the number of programs "solved" by each model per language (where "solved" means that at least one of the candidate patches the model generated passed all test cases). Rank is relevant to this analysis because it captures the sampling efficiency of a model for patching a program. In this case, rank 1 corresponds to the first and most probable candidate returned. Additionally, if a model generates duplicate patches for a given program, we test only the one returned first.

2.3 MODELS

We now describe each of the four language models used to generate candidate patches in our experiments. The smaller three, ranging from 160M to 2.7B parameters, are described together and with more specificity, as they are open-source and largely similar. The last model (Codex) is described briefly in a second subsection.

2.3.1 POLYCODER

We use the three publicly available versions of PolyCoder, with 160M, 0.4B, and 2.7B parameterss, which is a multi-lingual model trained solely on code that was created and open-sourced by Hellendoorn and described in Xu et al. Xu et al. (2022). Each of these is a Transformer model Vaswani et al. (2017) based on the GPT-2 architecture Radford et al. (2019), using the GPT-NeoX toolkit provided by Andonian et al. (2021). The model sizes (in terms of parameters, layers, dimensions), and training details (tokens per batch, context window, training steps, all identical) are listed in Table 1 for reference.

All three models were trained on the same large multi-lingual corpus of code spanning twelve popular programming languages and including a total of 249GB of data across 24.1M files. Java makes up the third largest proportion of these (41GB), compared to 16GB of Python data.

For this work, we modify only the text generation portion of the models, which initially returned sequences in an arbitrary order. Since Codex returns its sequences in order of decreasing likelihood, we adjusted the text generation code to do the same by summing the entropy (negative log probabilities) of all generated tokens in each sample, and then sorting the sequences to be returned in order of increasing entropy.

With respect to text generation in these experiments, for each model, we sample with a temperature of 0.8, a sample size of 100, and a prompt generated by the method outlined in the previous subsection (i.e., with all code up to the buggy line). As mentioned, we also restrict our responses to a maximum length of one line by using the newline character as a stop token.

2.3.2 CODEX

We refer to Codex as the largest (12B parameter version) of the family of GPT-3 based models trained on code by OpenAI, a version of which powers the Copilot VS-code plugin Chen et al. (2021). While developers can currently prompt and sample from a beta release of Codex via a restricted API, the source code and training data itself remain private. As such, certain details of Codex are not known to us in full.

This is relevant when comparing the amount of training code per language across models. Per the original paper (Chen et al., 2021), Codex was fine-tuned for code on a corpus with 159GB of unique Python files, a substantially larger dataset than the 16GB of Python data the other models were trained on. The only information provided about the full dataset, on which the current, multi-lingual release of Codex was trained, is that it was derived from 54 million public software repositories hosted on GitHub. It is unclear how what volume of Java and Python data this translates into. Still, going by the size of the aforementioned Python corpus alone (roughly the size of the other three models' entire multi-lingual corpus), we can reasonably assume that Codex was also trained on a far larger number of Java samples than the smaller models.

With respect to ranking, it is also somewhat opaque how the currently accessible version of Codex ranks the solutions returned for a given prompt. In our experiments, we record the sum of the entropy over each sequence that Codex returns and use that for ranking comparisons with the other models. In general, albeit with some exceptions, these solutions are returned in order of increasing entropy (or decreasing likelihood).[1]

For text generation with the Codex API, we replicated the previously described procedure as closely as possible, again setting the number of samples to 100, sampling with temperature 0.8, specifying newline as a stop token, and passing it the same prompts given to the other models.

3 RELATED WORK

Chen et al. (2021) introduce the Codex model and evaluate the Python programs it synthesizes using only doc-strings. Notably, they find that Codex generates a functionally plausible solution to 70.2% of the programs in their benchmark within 100 samples. Loss and pass-rate were analyzed with respect to scale specifically for Codex and compared against models trained only on natural language at some points in this paper, suggesting that the patterns observed by Kaplan et al. (2020) carry over for synthesis tasks, and that fine-tuning on code is a powerful technique for code generation.

Our work is comparatively more specifically targeted towards program repair. Whereas Chen et al. (2021) generate complete programs from natural language documentation, we synthesize only a repair for an already-localized bug based on a partial program (specifically, the prefix context). We also evaluate the impact of scale with respect to one-line patch generation (as opposed to full program synthesis), and compare performance on Java as well to offer results that generalize across languages.

Prenner & Robbes (2021) is perhaps the most similar pre-existing work to ours, as it also uses language models to patch programs in the Quixbugs Lin et al. (2017) dataset. There are two major differences between our work and this one. The first is that, like the authors of Chen et al. (2021), Prenner & Robbes (2021) attempt to generate full functions from a prompt, as opposed to attempting to repair one line at a time. The second major difference is that their work only uses Codex for program repair. We include Codex in our study, but also use three versions of open-source models trained with different parameter budgets. The benefit of this choice is two-fold: first, it allows us to conduct the subsequent scale analysis, and second, it allows us to sprovide information regarding the performance of publicly available models of code that are accessible to the research community.

Several of our findings complement those of Prenner & Robbes (2021). They found that Codex solved between 47-57% of Quixbugs programs (for Python) and between 35-45% of programs (for Java) when tasked with providing *complete functions*. In our case, the models are only tasked with generating a single patching line. This helps Codex solve 88.9% of programs in Python, and

[1]We note that the API tended to return slightly different log-likelihood scores for the same completion, varying by around 0.01 bits.

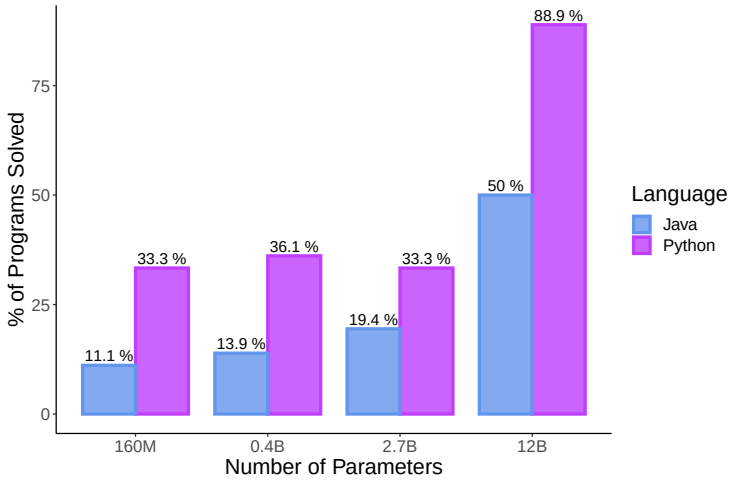

Figure 3: The percentage of programs successfully patched (test-passing) per model and language.

50% in Java. Most notably for Python, generating single line patches is nearly twice as accurate as generating a full function, whereas the Java repair rate increased comparatively less. While some increase is expected, since generating one line is comparatively simpler than generating a full function, these results provide helpful datapoints to direct this new and active area of research.

4 RESULTS

We break our results into the following three research questions, each one of which evaluates a different aspect of the feasibility and scaling behavior of patch generation with the four models.

- **RQ1:** *Are larger models better patch generators?*
- **RQ2:** *Do larger models consider test-passing patches more predictable?*
- **RQ3:** *Do larger models prefer developer patches?*

4.1 RQ1 - ARE LARGER MODELS BETTER PATCH GENERATORS?

To answer RQ1, we first discuss the number of programs that were successfully patched by each model across languages. We considered a program "successfully patched" if any one of the 100 candidate programs was functionally plausible (i.e., passed all of its test cases).

Figure 3 shows percentages of Python and Java programs successfully patched, according to their test suites by model. In both languages, the GPT-3 model (Codex) performs substantially better than the GPT-2 models, as shown by the visible jump in performance between 2.7B and 12B parameters. While the three smaller models solved up to 36% of Python programs and 19% of Java programs, Codex solved 89% of Python programs, and 50% of Java programs.

Curiously, whereas the number of Java programs solved monotonically increases as model size does (albeit, less dramatically for the GPT-2 models), for Python the 0.4B parameter model solves more than 2.7B one. In part, this may simply be a result of the small sample size. Still, there is a significant difference between the percentage of programs solved by each of the GPT-2 models when compared with GPT-3, suggesting that the size of the underlying natural language model is a highly relevant factor for performance on code related tasks, echoing recent findings from Google's PaLM (Chowdhery et al., 2022).

In contrast to overall performance, which notably improves as size does, the ratio of performance between Python and Java programs is largely independent of scale: each model solved between a third and half as many programs in Java as it did in Python regardless of its size.

Key Insight: Overall, larger models were more successful patch generators. A significant leap in performance was observed between 2.7B parameters (GPT-2) and 12B parameters (GPT-3). The largest and most successful model patched 89% of Python programs, and all models patched 2-3 times more programs in Python than Java.

4.2 RQ2 - Do larger models consider test-passing patches more predictable?

The previous research question discussed the rate at which any one of a model's 100 solutions passed all tests. Here we analyze the entropy ranking of *test-passing solutions only*. Entropy captures how "surprising" or "unnatural" a model finds a solution, allowing us to examine the relationship between scale and and the naturalness of patches. Prior work showed that n-gram language models generally consider bugs less natural Ray et al. (2016). We evaluate whether the inverse is true with respect to transformer models.

As described in Section 2, each model ranks its solutions according to entropy, where a rank 1 solution is least entropic (most likely, returned first), and rank 100 is most entropic (least likely, returned last). By examining how "natural" or likely each model considered its test-passing patches, we can evaluate its sampling efficiency, as well as the extent to which its rankings are meaningful in the context of patch generation.

Figure 4 shows the distribution of the ranks of all test-passing programs generated by each model (median ranks emphasized for clarity). In Python, when the larger two models did produce a test-passing patch, they often did so with only one attempt (72% of the time by Codex, 73% of the time by the 2.7B parameter model). This rate was much lower for the smaller two models (25% at 0.4B, 27% by 160M), suggesting that larger models are better able to differentiate bugs from patches via entropy rankings.

In Java, the smallest two models solve very few programs (11.1% and 13.9% of programs, respectively), but when they did produce a passing patch, these solutions were typically also ranked near or at the top by these models. Due to the extremely small set of successful Java patches generated by the two smaller models, their rank distribution is interesting but not particularly meaningful.

In Python, we previously observed that the number of programs solved by the GTP-2 models was practically identical. Here, however, we see that the median rank of their test-passing patches changed substantially between 0.4B and 2.7B parameters – the latter ranks test patching patches substantially higher than the former. Unlike functional plausibility 4.1, where the major leap emerged between GPT-2 and GPT-3, here we observe that "meaningful" entropy rankings (with respect to patch generation) emerge at GPT-2 with 2.7B parameters.

Key Insight: Larger models generally considered patches more natural. A significant decline in the entropy of test-passing patches emerged between 0.4B (GTP-2) and 2.7B (GTP-2) in Python, demonstrating that sampling efficiency increases with model size.

4.3 RQ3 - Do larger models prefer developer patches?

For the final research question, we discuss the BLEU score similarity of the candidate patches generated by each model to the ground-truth (the original human-created bug and patch from the benchmark). While BLEU scores in the range (0-99) are not necessarily meaningful, a BLEU score of 1 is relevant as it means a model perfectly recreated the original bug or patch. We analyze the rank of the candidates with a BLEU score of 1 compared to the GT bug as well as those with a BLEU score of 1 compared to the GT patch. The results of this comparison are shown in figure 5.

A distinction again emerges according to scale, with the two larger models ascribing a lower entropy to the developer patch than to the bug in both languages. In Java, all models prefer (as in, rank higher) the Ground Truth (GT) patch, whereas in Python, only the larger two models do so. Still, true bugs were rarely assigned a lower entropy than true patches by any model for either language,

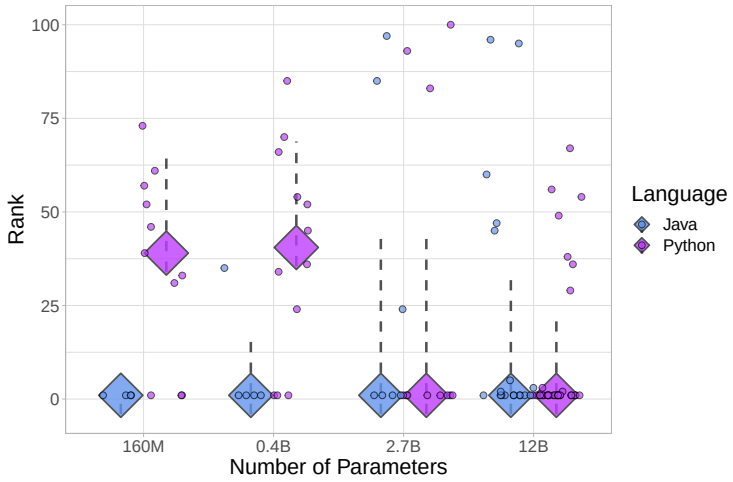

Figure 4: Rank distribution for the test-passing patches generated by each of the four models per language. Each point corresponds to the rank of one correct candidate patch, and diamonds correspond to the median of these points.

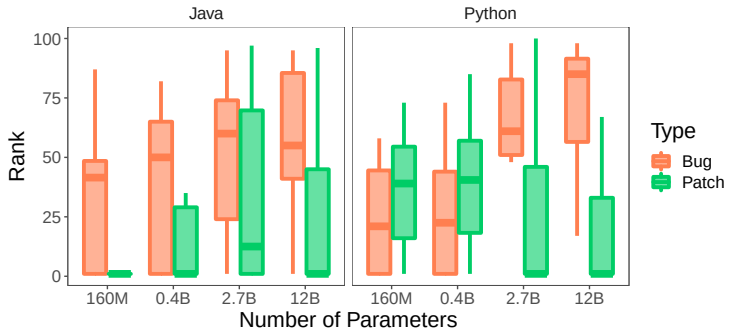

Figure 5: The distribution of ranks for candidate patches that had a BLEU score of 1 (i.e., matched perfectly) with either the ground truth patch or bug for Java and Python samples (respectively).

further supporting the notion that patches tend to be more "natural", and that these patterns are emphasized as model size increases.

Figure 6 shows the BLEU scores for all 100 candidates (whereas figure 5 includes only candidates with a BLEU score of 1). Candidates at each rank were compared to both the GT bug and patch, and then averaged over the total number of programs. Java is excluded from this plot for the sake of visibility, however the results were similar.

Note that as entropy increases, both patch and bug BLEU score also decline. Notably, however, while the 12B parameter model and 2.7B parameter model both consistently generate solutions more similar to the true patch than the true bug, as represented by the triangular points, this relationship flips for the two smaller models.

Key Insight: Larger models tend to consider developer patches more natural. In Python, the larger two models (12B and 2.7B) assign true patches lower entropy than bugs, whereas the smaller two (0.4B and 160M) assign true bugs lower entropy.

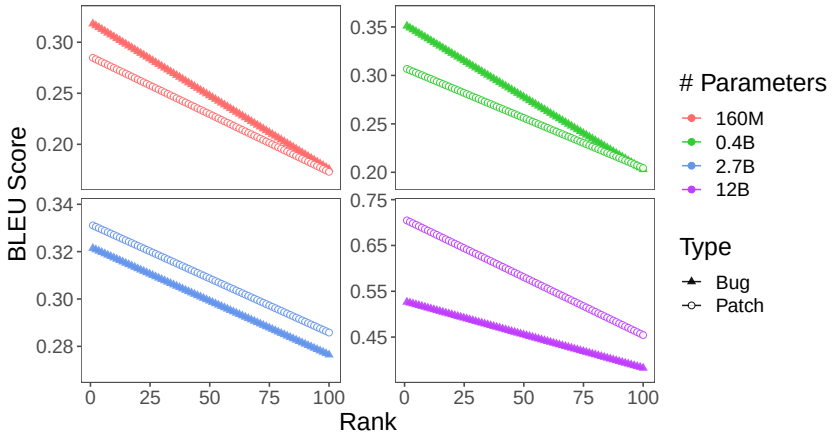

Figure 6: The average BLEU score of all 100 candidates generated per model compared to the GT patch and bug in Python. On average, the larger two models (12B and 2.7B) generate solutions more similar to the patch, whereas the smaller two generate solutions more similar to the bug.

5 THREATS TO VALIDITY

Fault Localization: Our analysis sought to isolate patch generation from other phases of program repair. As such, we assumed fault localization had already been conducted prior to this phase. Still, it is debatable whether or not patch-generation can or should be treated as a stand alone technique, and whether or not these scaling results would hold if combined with fault localization is unclear.

One line bugs: Additionally, we study only one line bugs, which comes from a dataset of relatively simple programs. In practice, bugs are rarely so neatly localized to a single location. While a robust methodology for program repair was not the intention of this paper, we do draw a conclusion about the likelihood of bug versus patch generation that may not generalize to more complex cases where the "ground-truth" is unclear or a patch alters multiple statements in the program.

Functional plausibility: A candidate program was deemed "functionally plausible" when it succeeded in passing its associated test cases, however, the extent to which functional correctness corresponds to expected behavior is always dependent on test quality. While we did not write or manually evaluate these test cases, they have been used in numerous APR studies, so we assume they are a sufficient approximation for correctness. Still, if the tests are flawed, it is possible that some of the "functionally correct" programs do not behave as expected in all cases.

6 CONCLUSION

We presented an analysis of large language models for program patch generation. By focusing specifically on one line bugs and patches, we shed light on the impact of model scale on both the rate and rank of test-passing patches, which correlate with size and climb especially quickly for the largest models, and on the remarkable relationships between model size and proclivity for echoing the original, human-written patch. It is perhaps not coincidental that, compared to other test-passing tests, the latter is guaranteed to generalize and arguably the most natural – a property that language models naturally achieve. These results show the promising role of especially very large (10B+ parameter) language models in guiding patch selection in Automated Program Repair work.

ACKNOWLEDGEMENTS

This work was partially supported under NSF Grant No. CCF-1762363, and CMU-Portugal project ANI 045917 funded by FEDER and FCT.

REFERENCES

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Alex Andonian, Quentin Anthony, Stella Biderman, Sid Black, Preetham Gali, Leo Gao, Eric Hallahan, Josh Levy-Kramer, Connor Leahy, Lucas Nestler, Kip Parker, Michael Pieler, Shivanshu Purohit, Tri Songz, Phil Wang, and Samuel Weinbach. GPT-NeoX: Large scale autoregressive language modeling in pytorch, 2021. URL http://github.com/eleutherai/gpt-neox.

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

Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? . In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, pp. 610–623, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383097. doi: 10.1145/3442188.3445922. URL https://doi.org/10.1145/3442188.3445922.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

Vincent J. Hellendoorn. Large Models of Source Code. URL https://github.com/VHellendoorn/Code-LMs.

Vincent J Hellendoorn, Petros Maniatis, Rishabh Singh, Charles Sutton, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations (ICLR)*, 2020.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. SPLASH Companion 2017, pp. 55–56, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355148. doi: 10.1145/3135932.3135941. URL https://doi.org/10.1145/3135932.3135941.

Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In Gregg Rothermel and Doo-Hwan Bae (eds.), *International Conference on Software Engineering (ICSE'20)*, pp. 615–627. ACM, 2020.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pp. 311–318, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://doi.org/10.3115/1073083.1073135.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *arXiv preprint arXiv:2108.09293*, 2021.

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

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 428–439, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884848. URL https://doi.org/10.1145/2884781.2884848.

Ngoc Tran, Hieu Tran, Son Nguyen, Hoan Nguyen, and Tien N. Nguyen. Does bleu score work for code migration? In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pp. 165–176. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00034. URL https://doi.org/10.1109/ICPC.2019.00034.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of NeurIPS*, 2017.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022.