Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models

Aidan Z.H. Yang *, Sophia Kolak †, Vincent J. Hellendoorn ‡, Ruben Martins §, Claire Le Goues ¶
Carnegie Mellon University
Pittsburgh United States
Email: *aidan@cmu.edu, †sdkolak@andrew.cmu.edu, ‡vhellendoorn@cmu.edu, §rubenm@cs.cmu.edu, ¶clegoues@cs.cmu.edu

***Abstract*—Language models have improved by orders of magnitude with the recent emergence of Transformer-based Large Language Models (LLMs). LLMs have demonstrated their ability to generate "natural" code that is highly similar to code written by professional developers. One intermediate value an LLM can emit is entropy, which measures the "naturalness" of a token of code. We hypothesize that entropy can be used to improve the performance of Automated Program Repair (APR) tasks. While much progress has been made in Automated Program Repair (APR), fault localization techniques suffer from a lack of diversity in ranking scores, patch generation tools tend to be inefficient as all tests need to run before determining if a patch is likely to be correct, and patch ranking often suffers from the test-suite over-fitting problem. However, using an LLM directly for APR introduces concerns for training data leakage. In this work, we introduce a novel way of using the entropy of LLMs in combination with prior APR tools to improve all stages of APR. By using only the prefix and suffix context of a line or block of code to describe "naturalness", we can use LLMs to localize faults and rank patches all while eliminating the dependency for test-suites. We show that entropy is highly complementary with prior fault localization tools. Our proposed re-ranking method achieves a 50% Top-5 score improvement over SBFL. We propose a patch-naturalness measurement, entropy-delta, to improve the efficiency of template-based repair techniques by ranking plausible patches before undergoing testing. When using entropy-delta for patch ranking and classification, our proposed method can rank correct patches more effectively than state-of-the-art machine learning tools with an 49% improvement in Top-1. Our work suggests that LLMs can be an effective addition to compliment prior APR tasks while minimizing both the test-suite overfitting problem and the LLM data leakage problem.**

I. Introduction

The problem of software quality has motivated the development of a variety of techniques for Automatic Program Repair (APR) [1], [2], [3], [4], [5]. At a high level, dynamic APR approaches use test cases to define a defect to be repaired and functionality to retain, and to localize the defect to a smaller set of program lines. APR techniques generate candidate patches in a variety of ways, such as by heuristically instantiating pre-defined repair template [1], [6], or by customizing symbolic techniques to synthesize new code [5], [7].

Meanwhile, the recent advances in machine learning and AI, including but by no means limited to advances in Transformer [8] based language models, have produced orders of magnitude performance improvements over previous ML techniques for code generation [9], [10]. ML therefore affords promising opportunities for program repair [2], [3], [11], [12], [13] and fault localization [14]. The applicability of language models to the repair process makes sense: these models are trained on large volumes of code in which defects are relatively rare. Since their training objective encourages next-token prediction, well-trained language models tend to simultaneously perceive faulty code as unlikely (or "unnatural") and to produce code that is correct, as correct code is more "natural" [15]. The naturalness of code and unnaturalness of buggy code is now a well-established phenomenon [16], [15]. However, the bulk of prior research on this topic relied on relatively simple n-gram language models [17]. Compared to present-day LLMs, these models provided a very poor estimator of code predictability. The "unnaturalness" of buggy lines was therefore mainly useful as an explanatory metric, but showed limited utility for precisely localizing defects, let alone repairing programs. The recent advancement of much larger and more sophisticated LLMs have decreased model perplexities by multiple orders of magnitude. This makes them a much more accurate adjunct both for estimating naturalness and for fault localization or correct patch identification [18], [19].

In this paper, we revisit the idea of (un)naturalness for program repair. The fundamental idea behind using an LM alone — even a hypothetically optimal one — for repair treats predictability as ultimately equivalent to correctness. This assumption is specious: LLMs adopt preferences based on a corpus with respect to training loss that rewards imitation. Beyond the fact that LLMs necessarily train on buggy code, LLMs generate and score text one token at the time. Given that, they may well prefer a subtly incorrect implementation spread across several readable lines over a correct but difficult-to-understand one-line solution, as the per-token surprisal of the former may be strictly lower than the latter. Judgement of code correctness requires substantially more context than an LLM has access to including, but not limited to, test cases test behavior, and developer intent. Although some of this information could be provided as context, it will lie outside the training distribution.

This implies that LLMs can only go so far on their own in reasoning about and fixing buggy code. It moreover motivates

the use of traditional tools, which compress such information, as a complement to LLMs in repair, which has indeed shown promising recent results for the patch generation stage in particular [18] (acknowledging the risk of training data leakage in any such experiment [20]).

We go beyond prior work by interrogating the role of entropy as a complement to traditional repair at every stage:

Fault localization (FL). End-to-end dynamic APR relies on fault localization to narrow a bug to a smaller set of source locations. Improving fault localization accuracy is key to improving repair efficiency [21], [22]. Although FL accuracy is improving, both commonly-used [23] and state-of-the-art ML-based techniques still suffer from the tendency to assign the same FL score to large amounts of code. For example, we find that Ochiai SBFL [23] assigns the same suspiciousness score to 1137 lines of code in the dataset *Defects4J* (an average of 2.9 ties per bug), and TransferFL [22] assigns the same suspiciousness score to 380 lines of code in the same dataset (an average of 0.96 ties per bug).

We show that by incorporating entropy into fault localization, the variance of suspicious scores increase by 87% for Ochiai SBFL, and overall accuracy increases as well (e.g., the Top-5 score improves from 94 to 145).

Plausible patch generation. APR approaches typically generate multiple potential code changes in search of *plausible* patches that cause the program to pass all tests. Executing tests (and to some extent, compiling programs to be tested) dominates repair time: the template-based approach *TBar* [1] spends about 2% of its total time creating patch templates, 6% generating patches from templates, and 92% running tests on generated patches. Regardless of the patch generation method (e.g., symbolic techniques [5], [4], [7], template instantiation [1], [6], or machine learning models [18]) repair *efficiency* is best approximated in terms of the number of patches that must be evaluated to find a (good) repair [24].

We show that entropy, when used to order candidate patches for evaluation, can improve the efficiency of generic template-based repair by 24 tested patches per bug, on average.

Patch correctness assessment. Plausible patches are not always correct, in that they can fail to generalize to the desired behavior beyond what is tested in the provided test suite [25]. Some recent work aims to address this in the form of a post-processing step that identifies (and filters) plausible-but-incorrect patches, typically by combining program analysis and machine learning [26], [27], [19]. However, techniques to date are typically trained on the same test suites used for patch generation, imposing a project-specific training burden (and an expensive one, when dynamic signals are required), and posing a significant risk of overfitting [25], [19].

We show that entropy can rank correct patches 49% more effectively (in Top-1) than state-of-the-art patch ranker Shiboleth [27], without using any project-specific training data.

In summary, we make the following contributions.

- **End-to-end entropy APR.** We propose a technique that uses a combination of an LLM's inherent ability to detect "naturalness" (i.e., entropy) and an LLM's generation ability to predict faulty lines, rank untested patches, and classify tested but potentially incorrect patches.
- **Entropy-delta for efficient template-based patch generation.** We introduce entropy-delta as a patch-naturalness measure that can rank patches before running tests. We show that entropy-delta can be used to immediately filter out test-failing patches, and on average reduce running tests for 24 patches for each bug in our dataset. We combine entropy-delta patch ranking with a prior template-based program repair technique, *TBar* [1], and release a more efficient version of *TBar* for future research.
- **Artifact availability**. Our data, tool, and results are available and will be released as open-source.[1]

```
4473   XYItemRenderer r = getRendererForDataset(d);
                                  //SBFL=0.52, E=0.48
4474         ...
4493 - Collection c = getRenderer().getAnnotations();
                                  //SBFL=0.49, E=1.59
4494 + if (r != null) {
4495 +    Collection c = r.getAnnotations();
                                  //E=1.34
4496   ...
4502 + }
```

(a) Chart 4 buggy code and developer fix.

```
4493 - Collection c = getRenderer().getAnnotations();
                                  //E=1.59
4494 + if (r == null) {
4495 +    return null;           //E=2.77
4496 + }
```

(b) Chart 4 buggy code and test failing *TBar* patch #1.

```
4493 - Collection c = getRenderer().getAnnotations();
                                  //E=1.59
4494 + if (r == null) {
4495 +    continue;              //E=1.98
4496 + }
```

(c) Chart 4 buggy code and test passing *TBar* patch #19.

Fig. 1: Chart bug 4 from *Defects4J* with its developer-written fix, a test-failing patch generated by *TBar*, and a test-passing patch generated by *TBar*. We show the InCoder-produced entropy of code in each patch.

II. Illustrative Example

Consider the buggy and fixed versions of (Chart, 4) from *Defects4J* [28], shown in Figure 1a. The original buggy code is missing a null check, which the developer fixed by adding `if(r != null)` around the implicated code at line 4493.

The *TBar* [1] template-based program repair technique produces candidate patches by repeatedly instantiating applicable templates at program statements, ordered by Ochiai SBFL suspiciousness score. For example, Figure 1b a *TBar*-generated patch that does not cause the tests to pass, and so is discarded,

[1]https://zenodo.org/records/10851256

and the search continues. Given Chart's associated test suite, the Ochiai SBFL approach [21] assigns line 4473 the highest suspicious score in Chart of 0.52; line 4493, a suspiciousness score of 0.49; and 0.03 to lines 4494 and onward. Using only SBFL for fault localization ranking, the actual faulty line at line 4493 is ranked as 10th most suspicious. This does not prevent *TBar* from considering it, but does cost time.

TBar can produce patches that pass all tests for this bug, such as the one shown in Figure 1c. In the interest of reasoning about efficiency, we hold fault localization constant [24]; given that, this is the 19th patch attempted. However, although this patch prevents the null pointer exception, it does not generalize beyond the provided tests to capture the apparent developer intent. Importantly, *TBar can* produce the correct patch (from Figure 1a), if configured to execute beyond the first test-passing patch found — it is the 70th patch attempted, but only the second that passes all tests.[2]

LLM-based entropy provides useful clues, here, however. First, consider fault location: we use InCoder [29],[3] to measure the entropy of every line in this file. Rank-ordering them, line 4473 is ranked 8th-most-surprising. This is better than the SBFL technique on face. However, their real utility appears to lie in combination: re-ranking the lines receiving the Top-10 SBFL suspicious scores by InCoder entropy values, puts line 4493 at rank 2. We investigate how entropy in conjunction with SBFL performs for fault localization across multiple bugs and projects, as well as how different LLMs affect its performance.

We can also measure the naturalness of generated patches, such as by calculating the change in entropy, which we call entropy-delta ($\triangle E$), between the original buggy line of code and the proposed patches. The $\triangle E$ for the test-failing patch is -0.39; for the test-passing but still-incorrect patch is -1.18; and for the correct patch is 0.25.

The entropy-delta scores do not perfectly predict behavior (note that the test-failing patch has a higher score than the test-passing-but-incorrect patch), but it still suggests:

1) Entropy-delta can improve efficiency by suggesting the order to test patches. Test execution time is the dominant cost in program repair. By using entropy to rank potential patches *before* testing them, to suggest the order in which to do so, both test-passing patches can be found within 6 attempts (improving on 19 to the first test-passing in the default mode, and 90 to find the second, correct patch).
2) Entropy-delta can potentially help disambiguate plausible-but-incorrect from genuinely correct patches.

We evaluate these relationships in detail in the rest of this work, showing how entropy can usefully complement traditional approaches to automatic localization and transformation in the context of program repair.

III. Approach

We ask and answer the following three research questions about the utility of LLM-entropy for APR.

- **RQ1: How can entropy improve fault localization?** We perform an empirical evaluation of prior state-of-the-art fault localization tools and observe whether and how they benefit from the use of entropy scores.
- **RQ2: How can entropy improve patch generation efficiency?** To measure how entropy can be used for patch generation efficiency, we use it to rank proposed patches generated by an APR technique before running tests.
- **RQ3: How well does entropy-deltas identify correct patches?** We investigate if entropy-delta can differentiate plausible patches (patches that passes all tests) and correct patches (patches that correctly fix the bug).

This section describes how we use entropy for fault localization (Section III-A); our development of entropy delta for evaluating patches (Section III-B); and our modifications to *TBar* to enable our study of improved patch efficiency (Section III-C). The next section describes datasets and metrics.

A. Entropy for fault localization

We integrate raw entropy scores into prior fault localization techniques. Figure 2 overviews the approach. We take suspiciousness scores provided by a given prior FL tool for a given file. For consistency with prior studies of fault localization, we focus on top-N identified lines (as developers do not typically inspect more than 5 candidates [30]). We choose 6 and 10 for N, and name the two approaches 6-filter and 10-filter, as these are two quantities near the Top-5 that can still significantly impact Top-5 scores.

We then query an LLM for entropy scores for each line of code in that file. We tokenize the entire file, iteratively masking each line, and querying the model for each line's entropy. We used a sliding context window with 2048 tokens (i.e., the maximum attention window of our smallest selected LLM) surrounding the mask as suffix and prefix context. This allows us to assign entropy-based scores for all code in a file, even those longer than a given LLM's context window.

We then re-rank the suspiciousness code identified by SBFL by entropy score, and validate the ranked list according to the actual fault location in the dataset.

We incorporate entropy into three previous FL techniques: SBFL using the Ochiai formula [21], TransferFL [22] and LLMAO [14]. Ochiai is a common formula in SBFL used in traditional APR practices (e.g., *TBar*). TransferFL and LLMAO are the current state-of-the-art FL techniques using transfer-learning and LLMs, respectively.

[2]Using SBFL fault localization, *TBar* produces these patches at 1076 and 1127, respectively.

[3]When prompted with the code and asked to fix the bug directly, InCoder does not produce a test-passing patch in few-shot setting. Note that GPT4 fixes the bug correctly, and reports the git commit associated with the fix, implicating data leakage.

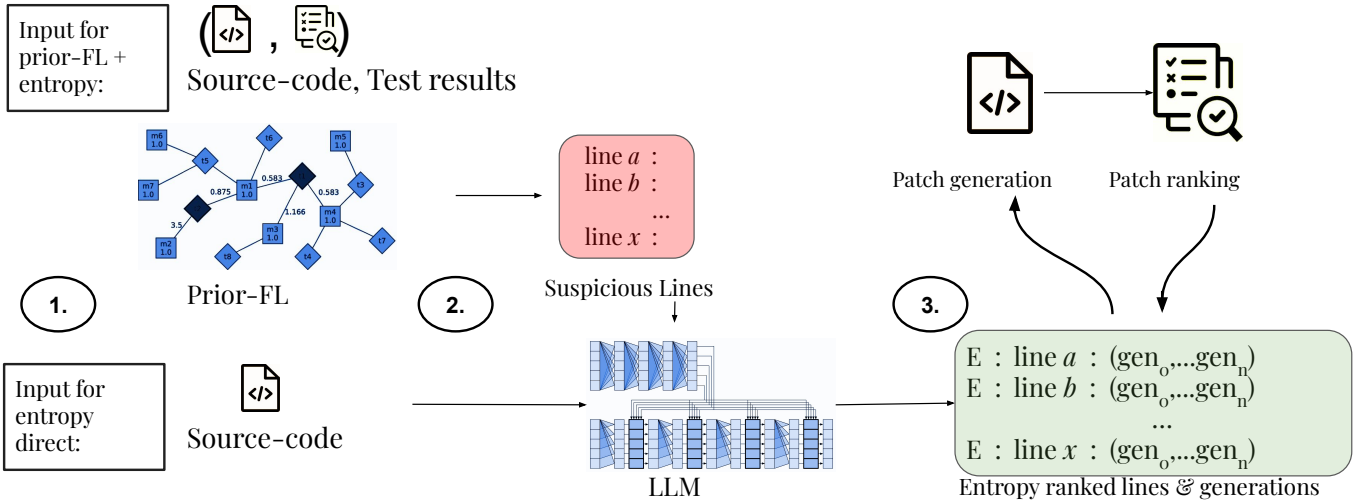
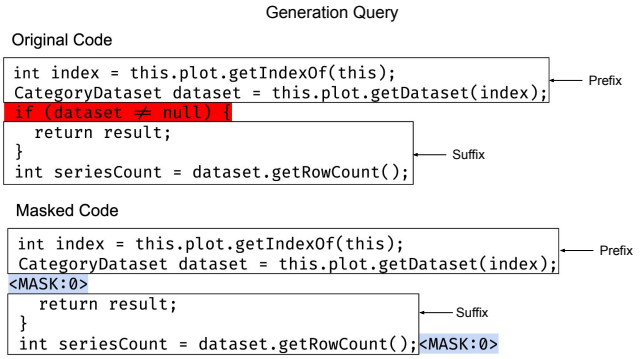
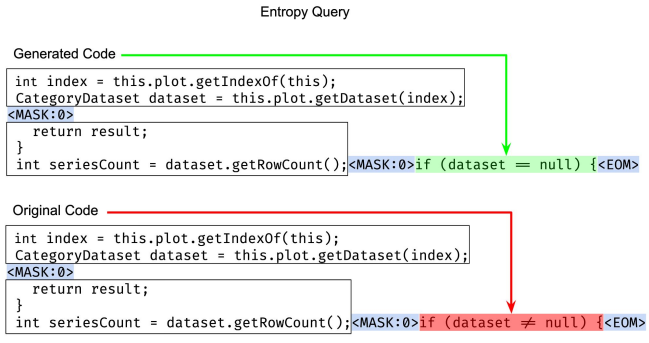


Fig. 2: Fault localization pipeline using entropy. (1) We take a prior-FL suspicious score list, (2) query each code-line for entropy values, and (3) re-rank the list using LLM entropy scores.



(a) An example of entropy-delta query from a code-line deletion patch. The entropy-delta value of the deleted line is the difference between the original line and a blank line.



(b) An example of entropy-delta query from a code-line replacement patch. The entropy-delta value of the replaced line is the difference between the original line and the replacement line.

Fig. 3: Example entropy-delta queries from an LLM. The MASK tokens enable models to learn the contextual relationships between tokens and make entropy predictions for missing, new, or replacement tokens. The EOM token is a special token that indicates the end of a mask.

B. Entropy-Delta

To evaluate patch naturalness, which we use in both patch prioritization during generation/evaluation and patch correctness prediction, we introduce the concept of an "entropy-delta". Entropy delta describes how code replacement changes the naturalness of a block of code. Figure 3a and Figure 3b give examples for our usage of entropy-delta for assigning a ranking score for patches. Figure 3a shows the process of masking out a deleted line of code and querying the LLM for the change in entropy using that mask (i.e., the change in entropy without the original line). Figure 3b shows the process of querying the LLM for the change in entropy if the tokens of the original line of code is replaced with new tokens of patch code. If the patch is an insertion of a blank new line, we query the entropy-delta between the "newline" token and the original line of code. For the case of an insertion, we measure the entropy-delta between the new code line and the original blank line.

An entropy-delta is simply the change in entropy before and after a line in code is replaced. For instance, if the line's original entropy is 1.0, and the replacement line's entropy is 0.5, then the line has an entropy-delta of +0.5, as in, replacing that line lowered entropy by 0.5. A significant reduction in entropy (large, positive entropy-delta) means that the replacement code lowered the entropy, implying both that the original statement may have been buggy and that the patch is more natural for that region of code. A large, negative entropy-delta means that the replacement code increased entropy, meaning that the patch is less natural at that location. An entropy-delta of 0 means that the patch has the exact same naturalness as the original code.

C. Modified TBar

Our patch efficiency experiments ask how entropy can speed up patch generation and evaluation. We evaluate it

in context of *TBar* [1], the best-performing template-based program repair technique in the existing literature. We avoid using ML-based APR techniques (even though some may outperform *TBar* [18], [22], [31]) because our goal is a controlled evaluation of entropy without learned patterns from the test suite. Evaluating based on a technique that otherwise also relies on trained ML models fails to isolate the effect of entropy per se.

TBar is a template-based patch generation technique integrated with *Defects4J* V1.2. Our experiments require several modifications to the codebase. First, we enable *TBar* to continue seeking patches after the first test-patching patch is found. Second, we enable *TBar* to generate patches, or evaluate them in a customized order (such as one provided by an entropy-delta ranking). Our *TBar* extension also includes some refactoring for modifiability/extensibility, as well as a more accurate patch caching mechanism (caching the patched source code, rather than the patch alone). We provide the modified code with our replication package.

IV. Datasets and metrics

In this section, we describe the models we use for entropy (Section IV-A), the bug and patch datasets considered (Section IV-B), as well as evaluation metrics (Section IV-C).

A. LLMs

We used InCoder [29], Starcoder [32], and Code-Llama2 [33]. The three LLMs were trained on open-source code and are capable of infilling with bidirectional context. The InCoder model [29] is a large-scale decoder-only Transformer model with 6.7 billion parameters. The model was trained on a dataset of code from public open-source repositories on GitHub and GitLab, as well as from StackOverflow. InCoder was primarily trained for code infilling, which involves the insertion of missing code snippets in existing code, using a causal-masked objective during training. However, its versatility enables it to be utilized in a variety of software engineering tasks, including automatic program repair. Starcoder and Llama-2 were trained with a similar autoregressive plus causal-mask objective as InCoder. Starcoder was trained with 15.5 billion parameters. Code-Llama2 have three versions available: 7B, 13B and 34B. We choose the 7B version as it is the closest in size to the other two models. Although the three LLMs were not specifically trained for repair, their large architectures and training objectives could imply that their entropy values on a particular region of code could suggest code naturalness. For all experiments, we set the LLM temperature to 0.5.

B. Dataset

We use the *Defects4J* [28] dataset as the basis of our experiments. *Defects4J* is a well-established set of documented historical bugs in Java programs with associated tests and developer patches. It is commonly used in APR, testing, and fault localization research. However, each research question requires a different subset of the data. Table I shows the number of bugs in each project that have at least one patch passing tests (for analyzing patch efficiency) and a developer fix (for analyzing patch correctness) along with plausible but incorrect patches. In total, we analyze 72 bugs from *Defects4J* V1.2 for patch efficiency and 197 bugs from *Defects4J* V2.0 for patch correctness.

TABLE I: *Defects4J* bugs with at least one patch passing tests (RQ2 - efficiency), and a developer fix (RQ3 - patch correctness).

	***Defects4J* V1.2 #bugs** Patch efficiency (RQ2)		***Defects4J* V2.0 #bugs** Patch correctness (RQ3)	
	Incl.	Total	Incl.	Total
Chart	11	26	19	26
Closure	19	133	64	174
Lang	14	65	35	64
Math	21	106	67	106
Mockito	3	38	1	38
Time	4	27	11	26
Total	72	395	197	434

We used *Defects4J* V1.2 for the fault localization and patch generation experiments. We do this because off-the-shelf TBar, as well as prior fault localization tools' replication packages, are all only compatible with *Defects4J* V1.2. The fault localization experiments consider all 395 bugs in *Defects4J* V1.2. We choose not to use *Defects4J* V2.0 for fault localization because prior tools' replication packages are only compatible with *Defects4J* V1.2.

For patch generation, the goal is to evaluate the degree to which entropy can improve repair efficiency; we therefore focus on the subset of *Defects4J* V1.2 bugs on which vanilla *TBar* succeeds at least once.

For patch correctness ranking, we use curated datasets from prior tools' replication packages directly, namely, Shibboleth [27] and Panther [26]. Shibboleth and Panther are both tools that leverage static and dynamic heuristics from both test and source code to rank and classify plausible patches, built on top of the updated *Defects4J* V2.0 dataset. We use a dataset of 1,290 plausible patches on *Defects4J* V2.0 curated by Ghanbari et al. [27]. For patch classification, we use a dataset of 2,147 plausible patches on *Defects4J* V2.0 curated by Tian et al. [26]. The patches from Tian et al. [26] were generated by seven different APR techniques. Each bug in the data set has one correct patch and several plausible (i.e., test passing) but incorrect ones. We calculate the change in entropy between the section of code in the original (buggy) file and the patched version. Note that both datasets only contain patches in projects Chart, Closure, Lang, Math, Mockito, and Time (6/17 of *Defects4J* V2.0's total projects), to compare with prior work built on *Defects4J* V1.2. Instead of the total number of bugs 835 in *Defects4J* V2.0, we only consider the 434 bugs in the 6 projects included by Shibboleth [27] and Panther [26] (shown in Table I).

C. Metrics

Fault localization and patch ranking. We measure the effectiveness of both fault localization ranking and patch ranking by counting the number of correct faults or patches that appear in

TABLE II: Top-N scores on 395 bugs from *Defects4J* V1.2.0 from 3 prior tools and re-ranking with entropy from three pre-trained LLMs: InCoder (6B), LLAMA-2 (7B), and Starcoder (15.5B)

FL type	re-rank Filter	Technique	Top-1	Top-3	Top-5
Entropy		entropy-Llama2	5	20	41
		entropy-Starcoder	9	35	55
		entropy-InCoder	**38**	**90**	**116**
SBFL		SBFL	24	61	94
	10-filter	entropy-Llama2	15	37	84
		entropy-Starcoder	15	40	88
		entropy-InCoder	50	98	133
	6-filter	entropy-Llama2	25	84	**145**
		entropy-Starcoder	28	86	144
		entropy-InCoder	**55**	**117**	141
TransferFL		TransferFL	**69**	126	145
	10-filter	entropy-Llama2	33	82	105
		entropy-Starcoder	39	92	126
		entropy-InCoder	49	114	144
	6-filter	entropy-Llama2	38	131	**184**
		entropy-Starcoder	44	138	178
		entropy-InCoder	57	**156**	182
LLMAO		LLMAO	**87**	134	**149**
	10-filter	entropy-Llama2	38	131	145
		entropy-Starcoder	45	107	144
		entropy-InCoder	77	131	146
	6-filter	entropy-Llama2	49	121	143
		entropy-Starcoder	36	84	151
		entropy-InCoder	81	**142**	**151**

the Top-N position. The Top-N measure has been widely used in APR research [34]. Existing studies [30] showed that over 70% of developers inspect only the Top-5 suggested elements. We use Top-5, Top-3, and Top-1 for fault localization ranking. We only use Top-2 and Top-3 for patch ranking following Ghanbari et al. [27], as some bugs in our dataset only have 2 plausible patches available.

Patch generation efficiency. We measure the effect of reranking generated potential patches in terms of the number of patch evaluations saved by doing so. Patch evaluations are established as a hardware- and program-independent measure for APR efficiency [24], and a proxy for compute time.

Patch correctness. For patch classification tasks, we convert entropy-delta values into binary labels. We label patches with a positive entropy-delta as "more natural" (i.e., more likely to be correct), and patches with a negative entropy-delta "less natural" (i.e., less likely to be correct). To measure entropy's ability to isolate correct and incorrect patches, we use +recall and -recall. +Recall measures to what extent correct patches are identified, while -recall measures to what extent incorrect patches are filtered out. We use accuracy, precision, and F1 scores to assess classification effectiveness over the entire dataset.

V. Results

In this section, we present results on the performance of entropy and entropy-delta on our three research questions:

RQ1: Can entropy improve fault localization?
RQ2: Can entropy improve patch generation efficiency?
RQ3: How well does entropy-deltas identify correct patches?

RQ1: Can entropy improve fault localization?

In this research question, we compared 24 different configurations for fault localization. Our analysis aims to determine the most effective approach for identifying the buggy statement in a series of one line bugs. We first measure entropy directly for fault localization with our three selected LLMs: Code-Llama2, Starcoder, and InCoder. We then measure the fault localization accuracy of three prior fault localization tools: SBFL [21], TransferFL [22], and LLMAO [14]. Finally, we use entropy to re-rank prior fault localization tools and observe that entropy re-ranking largely improves prior tools. Table II shows the Top-N scores (N = 1,3,5) on all configurations of our experiment. We observe that the entropy of InCoder, the smallest LLM in our lineup, is the most effective for fault localization. This is consistent with results from Xia et al. [18], who found that InCoder, trained with an objective of predicting missing code from a bidirectional context, is more effective at program repair tasks than larger but purely causal generative LLMs.

SBFL. From Table II, we observe an overall decrease in Top-N scores using either Code-Llama2 or Starcoder entropy with a 10-filter. However, all Top-N scores improve with the 6-filter. In particular, the Top-3 score of 84 for Llama2 entropy improves upon SBFL by 38%, and the Top-3 score of 86 for Starcoder entropy improves upon SBFL by 41%. Using InCoder entropy to re-rank SBFL shows substantial improvements across all Top-N and the two types of filters. InCoder entropy-SBFL with a 10-filter achieves a Top-1 score of 50 (108% improvement), and 5-filter achieves a Top-1 score of 55 (129% improvement). Similarly, the Top-3 and Top-5 scores improve by 61% and 41%, respectively for InCoder entropy-SBFL with a 10-filter. The Top-3 and Top-5 scores improve by 92% and 50% respectively for the 10-filter.

TransferFL. As seen in Table II, we observe an improvement from entropy on TransferFL's Top-3 and Top-5 scores using a 6-filter. In particular, 6-filter InCoder-entropy with TransferFL Top-3 is 156 (24% improvement), and 6-filter Llama2-entropy with TransferFL Top-5 is 184 (27% improvement). However, 6-filter InCoder-entropy with TransferFL yields a Top-1 score of 57, which is a 17% decrease in performance than TransferFL by itself. As compared to state-of-the-art machine learning based FL techniques, we observe that entropy scores perform worse on Top-1.

LLMAO. Similar to the results of TransferFL, re-ranking with entropy only improves fault localization results using the 6-filter. Furthermore, only entropy calculated using InCoder shows an improvement over LLMAO alone for Top-3 and Top-5, with a 8% and 1% improvement, respectively. Since LLMAO is already an LLM based FL tool, LLM entropy re-ranking shows marginal improvements as compared to prior non-LLM based FL tools. LLMAO finetunes on CodeGen 16B [35], which is a larger LLM than our three chosen LLMs.

TABLE III: entropy-delta ranking scores of 72 plausible patches generated by *TBar* per *Defects4J* project. The mean rank decrease is 24 and the median is 5.

Project	Improves ranking	Lowers ranking
Chart	11	2
Closure	15	4
Lang	11	2
Math	16	3
Mockito	1	0
Time	3	1
Total	60	12

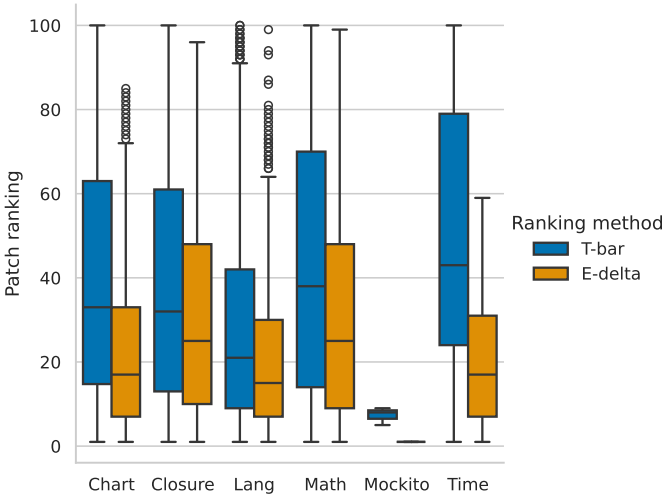


Fig. 4: Entropy-delta and *TBar* ranking (lower is better) of test-passing patches on 72 *Defects4J* bugs.

Our results indicated that SBFL benefits the most with InCoder's entropy re-ranking. Since SBFL has the most amount of tied suspicious scores (2.9 ties per bug on average), the additional suspiciousness from entropy values helps to break ties. TransferFL and LLMAO benefit from entropy re-ranking mostly when using a 6-filter. These findings suggest that incorporating entropy as a heuristic in fault localization can improve the accuracy of identifying the buggy statement, particularly when used in conjunction with SBFL.

RQ1 Summary

We leverage entropy for fault localization in *Defects4J* programs and show that, while entropy alone is only somewhat useful for finding defective lines, the measure is highly complementary when combined with prior fault localization tools, which highlights the importance of combining LLM-based methods with techniques from prior APR approaches.

RQ2: Can entropy improve patch generation efficiency?

In this section, we discuss the observed relationship of entropy and test-passing patches. We use entropy from InCoder, the most successful LLM in RQ1's fault localization. We measure the impact of entropy-delta on patch generation efficiency with two methods: (1) measuring each successful (test-passing) patch's ranking as ranked by original *TBar* and entropy-delta re-ranked *TBar*, and (2) incorporating entropy-delta into *TBar* and measuring the total number of patches generated to pass all tests.

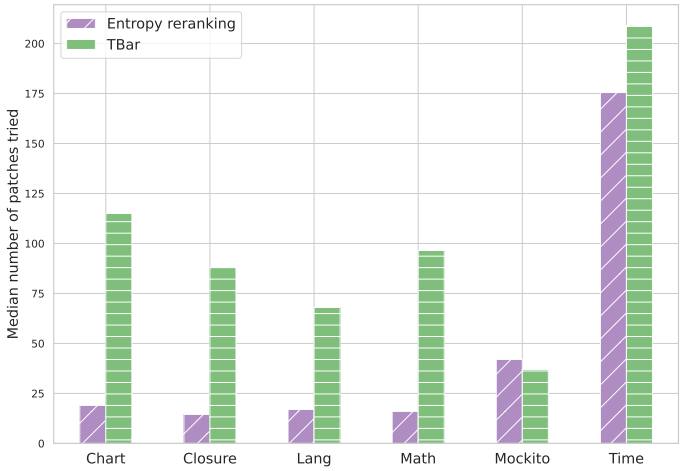


Fig. 5: Median number of patches tested (lower is better) per project before succesful patch using *TBar* original ranking and entropy-delta re-ranking of test-passing patches on 100 *Defects4J* bugs.

We first configured *TBar* to generate only 100 patches per each Defects4J bug, assuming perfect fault localization. Of the *TBar* patches we generated, 72 passed all tests contained in their bugs' respective repositories (e.g., all tests written for project Chart). Finally, we calculated the entropy-delta score for each patch, and the test-passing patch's original ranking according to *TBar*. As seen in Table III, entropy-delta improves 60 out of the 72 rankings as compared to *TBar*'s original ranking. On average, we observed a mean rank decrease of 24, meaning that using entropy-delta to rank the generated *TBar* patches can reduce a mean of 24 full test iterations (i.e., each potential patch must run through all test cases in the repository before knowing if it is a plausible patch). Liu et al. [24] compared 16 APR techniques and found that *TBar* exhibits one of the highest number of patches generated, but also the highest rate of bug fixing across *Defects4J*. We posit that entropy-delta's efficiency improvement over *TBar* significantly boosts template-based APR's overall utility.

Figure 4 compares the *TBar* ranking and entropy-delta ranking. Each bar represents the rank of test-passing patches compared to all generated patches per *Defects4J* project. A lower rank signifies a more efficient repair process, as the repair process ends when a test-passing patch is found. As seen in Figure 4, *TBar*'s original ranking for test-passing patches is higher than entropy-delta's ranking across all projects. Entropy-delta shows a higher disparity on ranking between test passing and test failing patches (i.e., a lower median rank for all test-passing patches). In particular, patches from projects Chart and Time show the largest improvement from re-ranking patches with entropy-delta. Successful patches in Chart and Time typically require multi-line edits, and with a wider range

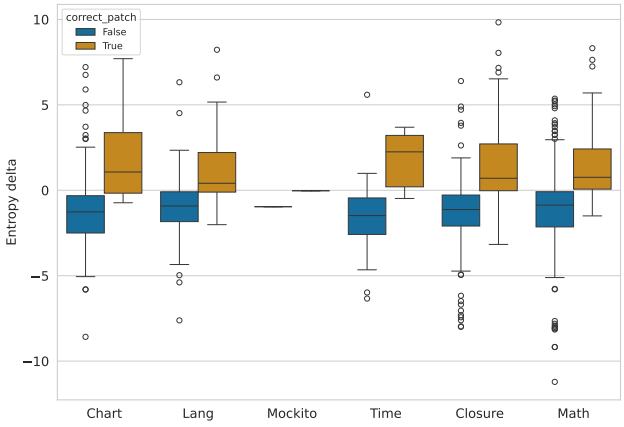


Fig. 6: Entropy-delta across correct and incorrect patches on *Defects4J* projects. A higher entropy-delta signifies a less surprising patch to the LLM, and a lower entropy (sometimes negative) entropy-delta signifies a more surprising patch to the LLM.

of templates to choose from, entropy-delta can make a greater impact in reducing the number of patches tested.

We then configured *TBar* to use entropy-delta ranked patches directly, and measured the total number of patches required until a successful bug fix (i.e., passing all tests). Figure 5 shows the median number of patches tested per project before a successful patch using *TBar* original ranking and entropy-delta re-ranking. We observe that entropy-delta re-ranking reduces the median number of patches tested across all projects except for Mockito. Mockito has only three single line bugs that *TBar* can fix. With a smaller total number of patches to try on a single template (e.g., 11 total possible patches for Mockito-26), entropy-delta re-ranking does not have as large of an impact on APR efficiency.

RQ2 Summary

We show that entropy can be used to rank patches before going through the entire test-suite, thereby reducing the test overhead for template-based repair technique *TBar* by a mean of 24 patches tested. Entropy-delta can both reduce the median number of patches tried before finding a fix, and consistently rank test patching patches higher than test-failing patches without any dependency on the test-suite. Entropy-delta is most useful for bugs that require multi-line patches.

RQ3: How well does entropy-deltas identify correct patches?

In RQ2, we saw that entropy-delta can improve the efficiency of patch generation by reducing number of patches tested. However, it is important to note that a test-passing patch is not necessarily correct. To further explore the issue of correctness, we investigated the ability of entropy-deltas to distinguish between correct and incorrect patches, both of which are test-passing.

1) Patch ranking: We evaluate a dataset of 1,290 patches generated by 7 prior APR methods collected by Ghanbari et al. [27]. For each bug, the data set includes some number of plausible (i.e., test passing) patches, where exactly one is correct, and the rest are incorrect. We attempt to isolate the true correct patch from the incorrect patches. We then rank each patch according to its entropy-delta, querying the model for the entropy of the entire patch region before and after the replacement. Table IV shows the Top-1 and Top-2 results of our approach on the labeled dataset of 1,290 patches. We see from Table IV that entropy-delta outperforms both SBFL and Shibboleth [27] on Top-2 across all projects, and entropy-delta outperforms Shibboleth on Top-1 across all projects but Chart (10 Top-1 as compared to Shibboleth's 11 Top-1). Overall, we see that entropy-delta improves upon Shibboleth by 49% for Top-1, and 27% for Top-2.

The difference in entropy reduction between correct and plausible but incorrect patches is shown in greater detail in Figure 6. We see a clear difference in entropy-delta across correct and incorrect patches. In particular, the correct patches for all six projects have a median entropy-delta value of above 0, and the incorrect patches for all six projects have a median entropy-delta value of below 0. A correct patch tends to appear more natural to the LLM as compared to its original buggy line.

2) Patch classification: Table V shows our classification results on a labeled dataset of 2,147 plausible patches curated by Tian et al. [26] for classifying patches as correct or incorrect. Entropy-delta improves upon the accuracy score of PATCH-SIM [5] and Panther [26], but only slightly improves +recall score over both PATCH-SIM and Panther. For -recall, entropy-delta performs better than PATCH-SIM by 9%, but performs worse than Panther by 10%. Entropy-delta slightly improves accuracy over Panther by 0.6%, and 89% over PATCH-SIM. Entropy-delta improves precision over Panther by 18%, and PATCH-SIM by 267%. Finally, entropy-delta performs better than both PATCH-SIM and Panther on F1 score, by 118% and 10% respectively. As compared to the state-of-the-art, entropy improves classification performance on true positives more than true negatives.

Our analysis focused on comparing the degree of entropy reduction between true correct patches and plausible test-passing patches. As shown in Table IV, Table V, and Figure 6, our results suggest that correct patches tend to lower entropy (i.e., increase naturalness) more than incorrect patches. Specifically, entropy-delta ranks 49% more correct patches in the Top-1 than the state-of-the-art patch ranker Shibboleth, and entropy-delta can classify correct patches with an 18% higher precision than the state-of-the-art patch classifier Panther. These findings suggest that entropy-deltas can be a valuable heuristic for distinguishing between correct and incorrect patches.

TABLE IV: Ranking results of 1290 plausible patches per *Defects4J* project using ranking methods SBFL, Shibboleth, and entropy-delta

Project	#Patches	#Correct	#Incorrect	Top-N	SBFL	Shibboleth	Entropy-delta
Chart	201	19	182	Top-1	3	11	10
				Top-2	6	14	14
Closure	269	64	205	Top-1	19	27	48
				Top-2	38	47	58
Lang	220	35	185	Top-1	1	14	20
				Top-2	12	22	27
Math	541	67	474	Top-1	10	27	39
				Top-2	30	38	55
Mockito	2	1	1	Top-1	0	1	1
				Top-2	1	1	1
Time	57	11	46	Top-1	3	8	9
				Top-2	5	5	10
Total	1290	197	1093	Top-1	36	85	**127**
				Top-2	92	130	**165**

TABLE V: Classification scores of 2,147 plausible patches on *Defects4J* projects using classification methods PATCH-SIM, Panther, and entropy-delta

Score	PATCH-SIM	Panther	Entropy-delta
Accuracy	0.388	0.730	0.735
Precision	0.245	0.760	0.900
+ Recall	0.711	0.757	0.760
- Recall	0.572	0.696	0.624
F1	0.377	0.750	0.824

RQ3 Summary

The entropy-delta from an LLM distinguishes between correct and plausible (test-passing but incorrect) patches with higher precision and accuracy than state-of-the-art patch disambiguation tools.

VI. Related Work

We discuss in the following sections the most recent advances in LLM for code, fault localization, and patch ranking.

A. LLM for code

Language models have been used for code generation, bug detection, and patch generation. Recent language models finetune on code as training data and can perform code completion [36], [29], and generate code based on natural language [37] with impressive results. Large Language Models (LLMs), such as Codex [9], GPT-Neo [10], and Llama-2 [33] have raised performance on these tasks by using more trainable parameters and training data. Ray et al. [15] study the relationship between bugginess and LLM-entropy. Ray et al. empirically showed that n-gram models trained over a large corpus of code will find buggy statements more surprising, as indicated by a high entropy score. Kolak et al. [38] revisit the question of naturalness (i.e., the human-readability) of patches in the era of large language models. Kolak et al. experimented with models ranging from [160M to 12B] parameters, and measured the similarly between LLM generated patches and developer written patches. Their results show that larger models tend to generate test-passing lines at a higher rate. Additionally, LLM generated patches tend to be more similar to the human-written patch as model size increases. Xia et al. [18] directly applied LLMs for APRs and found that LLMs can suggest multi-line fixes with higher accuracy than state of the art APR tools. Our study performs an empirical evaluation of how code naturalness (i.e., entropy) can improve prior APR tools across three different stages of automated program repair: fault localization, patch generation, and patch ranking.

B. Fault localization

Prior fault localization tools use test output information, code semantics, and naturalness of code to achieve a high degree of confidence on bug detection. Spectrum-based Fault Localization (SBFL) [23], [39] uses a ratio of passed and failed tests covering each line of code to calculate its suspiciousness score, in which a higher suspiciousness signifies a higher probability of being faulty. Recent advances in deep learning created a spur of research on using graph neural networks (GNNs) [40] for fault localization. GRACE [41], DeepFL [42], and DEAR [31] encode the code AST and test coverage as graph representations before training deep learning models for fault localization. TransferFL [22] combined semantic features of code and the transferred knowledge from open-source code data to improve the accuracy of prior deep learning fault localization tools. LLMAO [14] finetuned a light-weight bidirectional layer on top of code-tuned LLMs to show that LLMs can detect both bugs and security vulnerabilities without the use of test cases. Our work builds on top of the top-performing prior fault localization tools and show that entropy can be used as a light weight re-ranking tool that improves fault localization scores without a dependency on test cases.

C. Patch correctness

Similarly to prior fault localization tools, prior patch disambiguation tools leverage test output information and code information (both code syntax and code semantics) for ranking

or classifying patches. Qi et al. [43] analyzed the reported bugs of three generate-and-validate APR tools: GenProg [44], RSRepair [45], and AE [46] systems, to find that producing correct results on a validation test suite is not enough to ensure patch correctness. Smith et al. [25] performed an experiment that interrogates whether or not automatically generated patches are prone to overfitting to their test suite. Borrowing the concept of training and test sets from machine learning, they found that automated program repair (APR) typically used the same test-suite for both "training" (generating the patch), and "testing" (validation). Smith et al. found that both the coverage rate of the test-suite, as well as the assignment of test/train sets between the two suites, impact the degree of overfitting in repair. To counteract the overfitting problem, Ye et al. [47] proposed ODS (Overfitting Detection System), a novel system to statically classify overfitting patches. Xiong et al. [5] generated both execution traces of patched programs and new tests to assess the correctness of patches. Ghanbari et al. [27] used both the syntactic and semantic similarity between original code and proposed patch, and code coverage of passing tests to rank patches. Shibboleth [27] was able to rank the correct patch in Top-1 and Top-2 positions in 66% of their curated dataset. Tian et al. [26] proposed machine learning predictor with BERT transformer-based learned embeddings for patch classification. Tian et al. found that learned embeddings of code fragments with BERT [48], CC2Vec [49], and Doc2Vec [50] yield similarity scores that, given a buggy code, substantially differ between correctly-patched code and incorrectly-patched one.

The most relevant work to our study of patch correctness is Yang et al. [19]. Yang et al. [19] found that state-of-the-art learning-based techniques suffered from the dataset overfitting problem, and that naturalness-based techniques outperformed traditional static techniques, in particular Patch-Sim [5]. Our work uses 2,147 plausible patches collected in 2023 (past the LLM training data cutoff of all our chosen LLMs), which lowers the risk of LLM training data leakage. Our work performs an empirical study on entropy against the most recent state-of-the-art patch disambiguation techniques Panther [26] and Shibboleth [27], on top of Patch-Sim [5]. Motivated by Liu et al. [24], our work is the first to use LLM entropy on plausible patches before undergoing testing to achieve more efficient APR on prior template-based techniques. Finally, we introduce a new naturalness measurement for patches, entropy-delta, which achieves state-of-the-art results for plausible patch disambiguation without depending on the test-suites of buggy programs, which lowers the risk of dataset overfitting.

VII. Threats

External validity. A threat to the external validity of our study is the potential selection bias of our three selected LLMs. We chose a representative set of LLMs with a range of trainable parameters. We chose the three based on their infill ability and built-in bidirectional attention mechanism. Much larger LLMs ($>$ 20 billion parameters) might have a stronger ability to reason over faulty code lines and patches, but require much larger computation power and time for entropy calculation. Another threat to external validity is our usage of *Defects4J* data throughout our empirical evaluation. We chose *Defects4J* for our target bugs for fault localization and patch disambiguation due to the data available and the aim to compare against related work in APR. Data leakage of *Defects4J* as training data for our selected LLMs is possible. We mitigate this risk by (1) using entropy in combination with prior APR techniques instead of direct LLM prompting for patch generation, and (2) applying entropy-delta on untested or plausible patches that are not available online (i.e., recently generated and not used as an official patch for bug fixing).

Internal validity. An internal validity is the manual labeling of plausible patches. We used manually labeled data released by prior works [26], [27], in which the authors followed clear and reproducible decision criteria. Although mistakes could still be made on which plausible patches are correct or incorrect, we use the same labeling for all prior tools studied as well as entropy to create a standardized baseline on patch classification.

Construct validity. One source of construct validity is the measurements we chose for our empirical evaluation. We used Top-N as a ranking measurement for both bugs and patches, following prior APR work. To overcome some limitations of Top-N, we also use multiple patch classification measurements (accuracy, precision, recall, and F1) on a separate set of labeled patch data to strengthen generalizability.

VIII. Conclusion

In this work, we propose the use of "unnaturalness" of code for automated program repair through the measurement of entropy generated by code-tuned LLMs. We also introduce the term entropy-delta, which measures the difference in entropy between a proposed code insert (i.e., a patch) and the original code. Using three LLMs and three prior fault localization tools, we show that entropy can improve Top-5, 3, and 1 scores after re-ranking the first 6 potential bug localization. We use entropy-delta on untested patches to save an average of 24 test runs per bug for the template-based APR technique TBar. We show that entropy-delta can improve upon state-of-the-art patch ranking by 49% for Top-1, and classify plausible patches with a 18% higher precision. Our results indicate that LLMs can be a powerful addition to state-of-the-art APR tools without the dependency on tests, and the usage of LLM code-generation. The reduction in both test suites and LLM code-generation results in the reduction in model over-fitting and training data leakage.

References

[1] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[2] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[3] D. Wu and J. M. Mendel, "Patch learning," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 9, pp. 1996–2008, 2019.

[4] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[5] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 789–799.

[6] M. Kim, Y. Kim, K. Kim, and E. Lee, "Multi-objective optimization-based bug-fixing template mining for automated program repair," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 691–701.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[10] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.

[11] C. Koutcheme, S. Sarsa, J. Leinonen, A. Hellas, and P. Denny, "Automated program repair using generative models for code infilling," in *International Conference on Artificial Intelligence in Education*. Springer, 2023, pp. 798–803.

[12] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *arXiv preprint arXiv:2303.07263*, 2023.

[13] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[14] A. Z. Yang, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Large language models for test-free fault localization," *arXiv preprint arXiv:2310.01726*, 2023.

[15] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[16] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[17] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[18] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[19] J. Yang, Y. Wang, Y. Lou, M. Wen, and L. Zhang, "A large-scale empirical review of patch correctness checking approaches," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1203–1215.

[20] S. Balloccu, P. Schmidtová, M. Lango, and O. Dušek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," *arXiv preprint arXiv:2402.03927*, 2024.

[21] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 39–46.

[22] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1169–1180.

[23] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2006, pp. 39–46.

[24] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[25] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[26] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–34, 2023.

[27] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 654–665.

[28] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

[29] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[30] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.

[31] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[32] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[33] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[34] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[35] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[36] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 345–356.

[37] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[38] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=rHlzJh_b1-5

[39] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[40] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[41] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 664–676.

[42] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 169–180.

[43] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, 2015, pp. 24–36.
[44] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
[45] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "Does genetic programming work well on automated program repair?" in *2013 International Conference on Computational and Information Sciences*. IEEE, 2013, pp. 1875–1878.
[46] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.
[47] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2920–2938, 2021.
[48] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[49] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 518–529.
[50] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*. PMLR, 2014, pp. 1188–1196.