Language Models Can Prioritize Patches for Practical Program Patching

Sungmin Kang
KAIST
Daejeon, South Korea
sungmin.kang@kaist.ac.kr

Shin Yoo
KAIST
Daejeon, South Korea
shin.yoo@kaist.ac.kr

ABSTRACT

The field of Automated Program Repair (APR) has seen significant growth in the past decade. As the field progressed, the number of templates used by APR tools has grown substantially to increase the number of patches included within the domain each tool finds fixable, thus increasing their fixing capability. However, this heightened potential was not free: new techniques paid by using greater computational resources and time to look over an enlarged repair space. In this paper, we look to curtail this trend by using language models (LMs) to provide guidance about whether a generated patch is *natural*. By prioritizing patches that generate natural code, which has been demonstrated in prior work to be related to correctness, we can reduce the number of patches that must be inspected to find the first correct patch. We evaluate this prioritization scheme over five APR tools, and find that we can reduce the number of patches that must be inspected in up to 70% of bugs and reduce the total number of patches inspected by up to two-thirds, paving the way for lower-cost program repair.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**.

KEYWORDS

automatic program repair, language models, naturalness

ACM Reference Format:
Sungmin Kang and Shin Yoo. 2022. Language Models Can Prioritize Patches for Practical Program Patching. In *APR '22: International Workshop on Automated Program Repair, June 03–05, 2018, Pittsburgh, PA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524459.3527343

1 INTRODUCTION

Automated Program Repair (APR) aims to automatically fix code and thus close the debugging loop. Hence, many approaches have been proposed over the years, some using constraint solving-based techniques [19, 27], while others use a predetermined set of templates to generate new patches [22, 23]. While these two techniques are different on the philosophical level, they share a common trait: they are computationally expensive to run. We are particularly interested in tools that use templates and execute multiple patches to find the correct one. As such tools aimed to patch more and more bugs with a single framework, they increased the number of templates employed: TBar [23], for example, agglomerates multiple patch templates found by the APR literature into one technique. This allows a single tool to handle a greater breadth of bugs, but also increases the number of patches considered. Without appropriate prioritization, it takes a longer time to identify the correct patch out of the multitude of generated patches.

Listing 1: TBar Math80 Patch #13.

```
  -1132,7 +1132,7  (...) {
   (...) boolean flipIfWarranted(int n, int step) {
   if (1.5*work[pingPong]<work[4*(n-1)+pingPong]) {
     // flip array
-    int j = 4 * n - 1;
+    int j = 4 * getSolver() - 1;
     for (int i = 0; i < j; i += 4) {
       for (int k = 0; k < 4; k += step) {
```

To address this issue, we focus on the observation that many patches generated by existing repair techniques result in **unnatural code**, that is, unlikely to appear in actual code. For example, note Listing 1, a patch generated by the TBar repair tool. This patch changes the variable n to the result of the method call getSolver (). Even without information about the getSolver method, to human judgment the patch is awkward, as it is unnatural to multiply a solver with an integer. This patch is actually incorrect, suggesting our intuition was meaningful. This nudges us toward the question: by capturing correlations between tokens, **could statistical language models automate the provision of similar lexical intuition**, and thus hasten the process leading to the correct patch?

In this paper, we introduce a simple yet effective method to quickly discover the correct patch: we train a language model on an easily-obtainable corpus of Java code, and use it to re-rank patches generated by various APR tools. Language models have been used to quantify code naturalness since the birth of the concept of code naturalness [11], and have been an ingredient of many algorithms [10, 14]. Pertinent to our objective is the large body of research that finds naturalness difference between buggy code and non-buggy code [16, 25, 29]: based on such work, a language model could provide information about which patch is likely to be correct.

If naturalness estimated by language models does provide meaningful information, how can we incorporate this to reduce the cost of validating patches in the generate-and-validate scheme? We propose a simple rank aggregation method to do so, and verify that combined rankings that add information from the APR tool and the language model can reduce the number of patches inspected by up to two-thirds when compared to the original ranking.

ICSEW '22, June 03–05, 2018, Pittsburgh, PA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9285-3/22/05…$15.00
https://doi.org/10.1145/3524459.3527343

The contribution of the paper is as follows:

- We verify that language models can provide complementary information as to which patch is likely to be correct. Further, they outperform a recent patch correctness predictor [34] trained on curated patch data.
- We propose simple methods that combine the language model ranking and the default ranking produced by APR tools, to reduce the amount of validation effort required until the correct patch is found.
- We empirically show language model prioritization reduces the number of patches to be inspected by up to 65%.

The remainder of the paper is organized as follows. In Section 2 we present relevant work in the field. A concrete description of our approach is given in Section 3, while the experimental protocol is illustrated in Section 4. Results from experiments are presented in Section 5. We outline the limitations of our work and future research directions in Section 6, concluding in Section 7.

2 RELATED WORK

2.1 Automated Program Repair (APR)

Automated Program Repair (APR) aims to do its namesake: given some sort of specification about correct behavior, one wants to automatically fix the program so that it correctly handles the task on hand. APR is mature as a research area and there are far too many approaches to summarize in this paper; one can refer to Gazzola et al. [9] for an overview of the field.

Some of the most promising approaches up to now have come from the template-based program repair family [17, 18, 22, 23]. These techniques are a prime subject for prioritization studies, as they can generate a large number of patches quickly, and their main bottleneck is the validation cost itself.

2.2 Code Naturalness

Since Hindle et al. [11] found code has 'naturalness', there has been much work about how naturalness interacts with other known software engineering concepts. In particular, multiple results have shown that the naturalness of buggy code and correct code are different: Ray et al. [29] find that one can use naturalness to identify buggy code with a similar accuracy as certain static bug finders. Furthermore, there has been work that learns from correct code to help patch generation [25], but the features it uses are highly idiosyncratic and difficult to apply to other tools. On the other hand, our approach that directly leverages the concrete manifestation of the patch application does not need any complicated features, and thus can be applied to any automated program repair technique.

2.3 Repair Cost Reduction

In Generate and Validate (G&V) program repair techniques, validating each patch tends to take the lion's share of the computational cost [4]. This has been known in the literature for a while; as a result many have employed a variety of cost reduction techniques. While there are many ways to reduce repair cost, here we focus on those that prioritize patches based on their 'shape' and 'ingredients'.

Regarding shape, many APR tools use abstract templates to modify the code under repair. Prioritizing which template to use is important, as each template can generate numerous patches. Prophet [25], HDRepair [20], and CapGen [35] all use mined patch statistics to help the prioritization process, but generally use hand-crafted features that may be difficult to apply to other techniques; a similar limitation applies to CapGen. Meanwhile, Tan et al. [33], ssFix [36], SimFix [13], and Asad et al. [3] mainly use non-mined heuristics to determine which patch is likely to be the fix.

Regarding ingredients, there are many strategies to estimate what is likely to go into empty slots of templates. ELIXIR [30] uses four manually constructed features to prioritize which ingredient to use in the repair tool; Hercules [31] also uses a similar machine learning technique. On a similar note, Dynamoth [7] prioritizes based on in-class token frequency statistics. Meanwhile, deep learning-based APR techniques (such as SequenceR[5]) internally prioritize patches based on their weights, but only use relatively limited patch data and are not tool-agnostic.

The language model prioritizes over the shape and synthesis space. Note that the aforementioned prioritization techniques are generally confined to the repair tool that was developed along with the prioritization technique, and are non-trivial to apply to multiple tools. To the best of our knowledge, we are the first to propose a tool-agnostic prioritization technique that can successfully prioritize in both the shape and synthesis space.

3 APPROACH

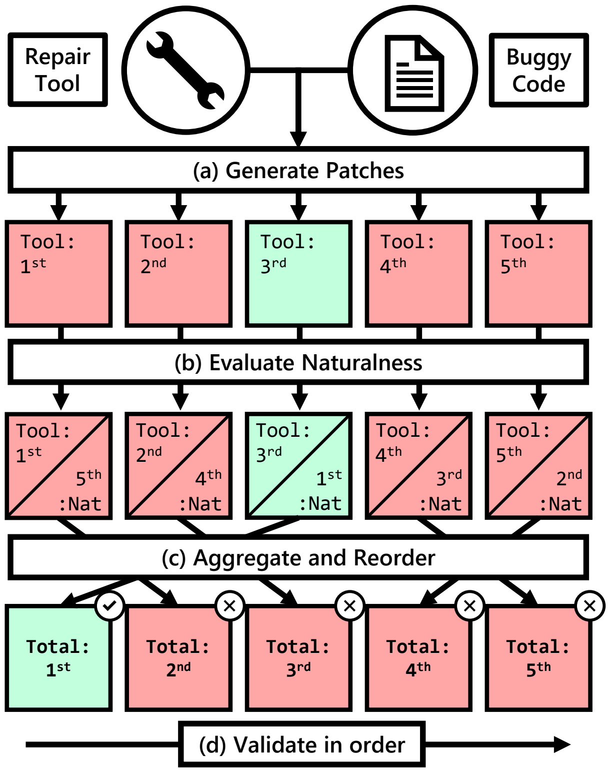

Figure 1: Overview of the patch prioritization process. Harmonic mean aggregation is used in this diagram.

A language model (LM) defines a probability distribution over sequences of words; i.e., given a sentence, it gives a probability value indicating how likely the sequence is. A common formulation is for a language model L to predict the upcoming token t_n provided preceding tokens $t_1, ..., t_{n-1}$, and deriving the probability $P((t_1, ..., t_N))$ of a sequence spanning from token t_1 to t_N as:

$$P((t_1, ..., t_N)) = \prod_{i=1}^{N} P(t_i | t_1, ..., t_{i-1}) \quad (1)$$

How can we prioritize which patches to inspect, given an LM? A graphic overview is provided in Figure 1. The prioritization process consists of three steps. First, patches are generated based on the buggy code and repair tool (Figure 1(a)). The repair tool's own patch order, denoted as *DR*, is recorded for later use. Next, the language model evaluates each patch (Figure 1(b)). While the language model cannot evaluate the correctness of a patch itself, it can evaluate code resulting from a patch. Thus, the language model is agnostic to what the buggy code was prior to the patch. Using Equation 1 (or a slight modification of it, as explained later in Section 4.3) we rank each patch in order of naturalness, denoted as *LR*. Finally, with these two rankings, we aggregate the information in the rankings to generate a final prioritization (Figure 1(c)). Specifically, using the two rankings, we calculate a score for each patch,

$$\text{score}(P) = \text{Agg}(DR(P), LR(P)) \quad (2)$$

where `Agg` is some aggregation function, such as the harmonic mean. As a lower number ranking means that the patch is of greater importance, the patches are rearranged in increasing order of score. The combined ranking is denoted with *CR*. Subsequently, we evaluate the patches against tests (Figure 1(d)). We expect to validate fewer patches under *CR* than under *DR*.

4 EXPERIMENTAL SETUP

The following experimental setup is used in the experiments described in this paper.

4.1 APR Tools Evaluated

We primarily took note of the work of Liu et al. [24] to choose APR tools to be used in our study. They report 16 APR tools could be reconfigured to operate under desired fault localization settings. Among those, we choose the five tools that correctly fix more than 20 bugs in the Defects4J [15] benchmark under perfect fault localization conditions: namely TBar [23], AVATAR [22], kPAR [21], FixMiner [18], and SimFix [13]. This is because we later evaluate the rankings based on the rank of the correct patch: evaluating on a tool that can only generate a small number of correct patches would introduce noise and threaten the validity of our results.

Among these methods, TBar, AVATAR, kPAR, and FixMiner all use greedy pattern matching to prioritize patches: starting by modifying the existing code just slightly, and subsequently progressing to patches that induce greater changes. Meanwhile, SimFix prioritizes patches based on three criteria: modification consistency, the number of modifications, and the number of replacements. Modification consistency means that for a particular donor template, the variables in that template are swapped in a consistent manner. SimFix also tries to minimize the number of modifications and replacements in considering patches to evaluate.

4.2 Research Questions

RQ1. Utility in Prioritization: Can language models provide prioritization information complementary to the default prioritization technique of each APR tool? We first confirm that language models have the potential to contribute to patch prioritization. The naturalness prioritization is compared with the default prioritization as follows. First, given perfect fault localization information, we generate all the patches that the APR tools can generate (this process naturally provides the default prioritization). The naturalness of each patched method is calculated using the language model, and a corresponding prioritization is obtained. The rank of the correct patch by default and naturalness prioritization is then compared. If there are a significant number of bugs in which the correct patch is ranked higher by the language model than by the APR tool, it would be reasonable to think that the language model can provide useful prioritization information. Note that we only check whether the language model is providing something complementary from, and not outright better than, the default tool rank. This is because the language model in this case has **no information about what patch operation was done**: it solely prioritizes based on code naturalness. Hence it does not benefit from information like what patch operator was used, which the APR tool can utilize. As such, our purpose with this RQ is to show that the language model can contribute information.

RQ2. Comparison with Correctness Classifier: Does our language model outperform a state-of-the-art baseline? To further put our performance in context, we measure performance relative to another patch evaluation algorithm that uses patch data. To this end, we compare our language models with a recent baseline from Tian et al. [34] that, given a plausible patch, predicts its correctness. This baseline uses pretrained code embeddings to first encode patches, then trains classifiers based on a patch dataset to classify which patches are correct. We compare prioritization performance with the BERT+classifier baseline, and see if there is a meaningful difference. We refer to this baseline as DLPC (Deep Learning Patch Correctness, from their repository [34]) for brevity.

RQ3. Validation Cost Reduction: Can we combine the two ranking approaches to bring the best of both worlds? Upon verifying language models can contribute to prioritization (RQ1), we combine rankings from APR tools and language models and see we can achieve better prioritizations than both. We use the same comparison with aggregated rankings, and again directly compare against the default ranking produced by the APR tools for each bug. We also investigate the validation effort for each technique; that is, how many evaluations are necessary until the correct patch is found. The validation effort VE is defined as

$$\text{VE} = \sum_{i \in \mathcal{B}} CR(P_i^c) \quad (3)$$

where $\mathcal{B}$ is the set of bugs with a correct patch, and P_i^c represents the correct patch for the ith bug. The validation effort based on the default ranking and the combined rankings is compared to evaluate whether incorporating language models can indeed result in overall validation effort reduction.

RQ4. Combination Techniques: How do different rank aggregation functions affect the ranking performance? In Equation 2, we left the aggregation function open to experiment: in terms of ranking performance, which combination technique actually performs the best? We evaluate five different aggregation functions: minimum, harmonic mean, geometric mean, arithmetic mean, and maximum. These aggregation methods each have different theoretic properties: in the presented order, the methods go from being influenced more by the smaller of the two ranks to being more influenced by the larger number. RQ4 investigates how these theoretical factors play out in our situation.

4.3 Implementation Details

APR techniques. We make the perfect fault localization assumption: that is, we prioritize patches that are generated from the buggy location. This is because our approach is tangential from methods that refine fault localization accuracy on the fly, as mentioned in Section 2.3. Furthermore, much recent APR work [5, 14, 21, 23, 26] uses perfect fault localization for evaluation. We use the Defects4J dataset for evaluation and run APR tools on it. We exclude bugs that required patches outside of methods, as the language model was trained on methods alone.

Language Model. We train a single-layer 1000-neuron unidirectional LSTM [12] network to predict the next token. The naturalness is not calculated by using the exact probability multiplication form as shown in Equation 1; instead we use the average of the log likelihood of each token. Concretely, for a patch $P_i \in \mathcal{P}$ that changes the method M to M_i, and the new method M_i consists of tokens $M_i^1, ..., M_i^T$, the score of P_i is calculated as follows:

$$\text{nat}(P_i) = \frac{1}{T}\sum_{t=1}^{T}\log(p(M_i^t|M_i^1, ...M_i^{t-1})) \quad (4)$$

This is the logarithm of the naturalness of the sequence as defined in Equation 1, divided by its length. We empirically find that this length-normalized naturalness performs the best, as the non-normalized versions tend to penalize patches that add new code.

We use the `java-med` dataset from code2vec [2] as the training set, which consists of 1,000 top-starred Java projects from GitHub. We removed Defects4J-related data from the training dataset. With this training dataset, we first use byte-pair encoding (BPE) [1] to tokenize; BPE was recently found to be useful by Karampatsis et al. [16] in the software engineering context as well, as it does not suffer from any out-of-vocabulary issues. We then train the language model for naturalness calculation. Meanwhile, any ties that happen after aggregation are broken using the APR tool ranking.

DLPC. To answer RQ2, we mostly use the existing work of Tian et al. [34] as it is implemented in the official repository[1]. We use the BERT implementation as the authors report that it shows the best performance. For the classification model, we use logistic regression; we train it with data provided in the repository.

[1]https://github.com/SerVal-DTF/DL4PatchCorrectness

Environments. The machine used for experiments had a Intel(R) Core(TM) i7-6700 3.4GHz CPU, 16GB of RAM, and an NVIDIA Titan X GPU for language model training. The language model was implemented with PyTorch [28]. Java method extraction and tokenizing used the SrcML [6] framework.

5 EMPIRICAL RESULTS

This section presents the results of our empirical evaluation.

5.1 RQ1: Utility in Prioritization

RQ1 asks whether language models can contribute information that helps prioritizing the correct patch. We primarily look at the number of bugs for which the rank of the correct patch improved, remained the same, or worsened, for each tool. We also investigate how well the default tool rankings compare to a random prioritization: as under a uniform random distribution, the rank of the correct patch should come at the midpoint, we compare the original rank of the correct patch to this reference point.

The results are presented in Figure 2, where bugs in which the language model outperforms the default ordering are marked in green, while red bugs are vice-versa. Note that the language model prioritization does appear capable of contributing to the prioritization: in all APR tools examined, the rank is improved in at least 40% of all bugs. Note that we are not trying to establish that the LM ranking is strictly superior to the default ranking; rather, Figure 2 demonstrates that LMs often (but not always) outperform the default ranking, suggesting LMs can complement the default ranking and help make a better ranking in general.

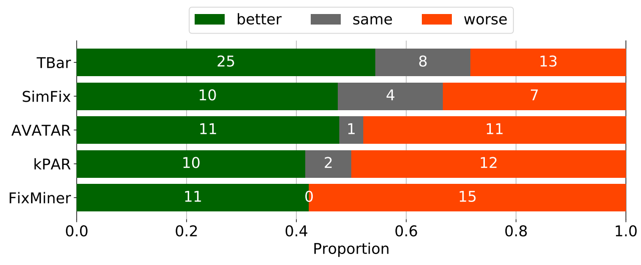

Figure 2: Count of how many bugs in which the language model ranked the correct patch at a better, same, or worse level as the default prioritization by APR tools.

We further compare the performance of both the repair tool and the language model to that of a random prioritization. The results are presented in Table 1, which presents the number of bugs in which each prioritization technique was better, same, or worse than the expected ranking of a random baseline, along with the results of a binomial test. Note that the language model is statistically significantly better than random under a $p = 0.05$ power binomial test for all tools except kPAR, while the prioritization of all tools except AVATAR were also better than random. This suggests that the results of Figure 2 are unlikely to be random, and that the language model indeed has information that could contribute to prioritizing the correct patch in the studied APR tools. The fact that the language model can improve the ranking of the correct

APR Tool	Default				LM			
	B	S	L	p	B	S	L	p
TBar	29	2	15	.024	40	1	5	<.01
SimFix	16	2	3	<.01	14	2	5	.031
AVATAR	12	0	11	.5	15	1	6	.026
kPAR	17	0	7	.031	16	0	8	.075
FixMiner	20	0	6	<.01	18	0	8	.037

Table 1: Ranking comparison with a random baseline (midpoint). For each technique, the B/S/L columns show whether the ranking was better, same, or worse than the random baseline. The p columns represent the p-value of such a result under a binomial test.

patch in a significant proportion of bugs, even when compared to strong ranking baselines, further strengthens our case that language models provide meaningful information.

Answer to RQ1: Language models can contribute to patch ranking. In fact, they perform competitively against the already well-performing prioritization of existing repair tools.

5.2 RQ2: Comparison with Correctness Classifier

Next, we compare rankings of the repair tool to those produced by DLPC [34], which uses pre-trained BERT embeddings and a curated patch dataset. We use the same evaluation metrics as before; the results of the comparison are shown in Table 2. The results show that using this baseline contributes little to the overall ranking: generally less than a third of bugs see their rank improve. Looking at the direct comparison to the random baseline, or the validation cost relative to the total number of patches, the result suggests a random prioritization model would yield similar performance. Thus, we can conclude that the language model can provide information useful for patch prioritization. To be clear, this does not mean that the work of Tian et al. does not replicate: using their implementation and data, we achieved similar classifier accuracy. Rather, this is because their task was to distinguish correct patches among plausible ones, not the full patch prioritization problem. Further, these results show that the patch prioritization problem is not trivially performed by such correctness classifiers.

Answer to RQ2: In contrast to the language model evaluated in RQ1, a state-of-the-art technique that predicts the correctness of patches performs similarly to a random baseline on the raw patch prioritization task.

5.3 RQ3: Validation Cost Reduction

In RQ1, we found the default ranking and naturalness ranking differed in their strengths, suggesting integration would help performance. RQ3 studies whether combined rankings outperform both the default prioritization and the language model based prioritization: the validation cost results are shown in Figure 3, with per-bug improvement results shown in Figure 4.

APR Tool	vs Default			vs Random				Validation
	B	S	L	B	S	L	p	Cost
TBar	10	3	33	16	2	28	.97	1525 (2890)
SimFix	6	3	12	13	0	8	.19	2328 (15642)
AVATAR	8	9	6	7	3	13	.98	539 (812)
kPAR	8	1	15	9	0	15	.92	979 (1304)
FixMine	8	0	18	10	1	15	.91	18993 (38213)

Table 2: Ranking results of DLPC, compared to the default ranking from APR tools and a random baseline. The fourth column shows the validation cost based on DLPC, along with the total number of patches in parentheses.

Note that across all five tools, while at times the language model alone underperforms the default prioritization, the combined rankings require significantly less validation effort. For example, in AVATAR the combined approaches can reduce the necessary inspection cost by 65%. In FixMiner, the language model made some large mistakes in prioritization, ending up with a huge inspection cost when employed on its own. However, under the combined approaches, the language model contributes enough information to actually reduce the validation cost: using harmonic mean, the validation cost is reduced by 36%. While it may appear that the performance in kPAR is mostly similar to the original baseline, this is because most of the cost is dominated by a single bug Lang24, in which both prioritization tools perform similarly. Excluding this outlier, we can achieve a validation cost reduction of 30% over the remaining 23 bugs, from 275 patch validations to only 194. This is also evident by looking at the rank distribution on the right, which shows that the cost for inspecting many bugs has decreased. These results strongly suggest that language models can indeed contribute towards reducing validation cost.

Answer to RQ3: The combined rankings could reduce validation effort significantly over a wide range of tools, up to 65%.

5.4 RQ4: Combination Techniques

As mentioned in Section 4, different aggregation methods have widely different properties when the two ranking methods diverge from each other. Then how do these aggregation methods perform in practice? We present the validation cost by each aggregation method in Figure 3, while a comparison of the aggregated ranking on a per-bug basis is provided in Figure 4.

First, observe that (particularly on the minimum aggregation side) there are bright green rectangles on almost every row indicating a large improvement; meanwhile, the red rectangles are almost always dim, indicating only a minor underperformance relative to the default ranking. This shows how the combined techniques achieve the net validation effort reduction depicted on Figure 3. On the other hand, towards the maximum side of aggregation functions, while when both rankings are performing well, the maximum function can do better than both; if only one ranking fails, maximum aggregation fails as well. This is especially clear in the maximum case of FixMiner from Figure 4: note how the number of bugs colored in bright red increases as we go toward the maximum side

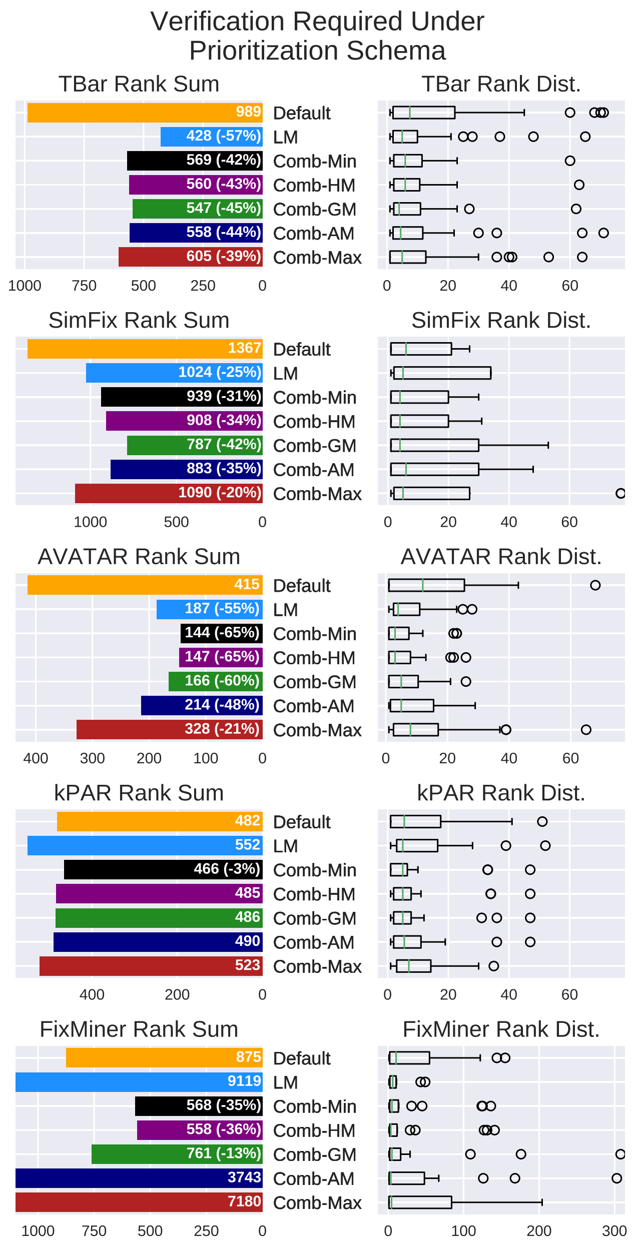

Figure 3: Validation effort by each model and combination technique. (Left) Validation cost for each technique. The number in parentheses represents cost reduction relative to the default rank. (Right) Per-bug cost distribution via box-plots; some outliers are truncated.

of aggregation functions. Thus while there is a tradeoff in which aggregation function to use, in terms of "the probability that the prioritization will be better than vanilla repair for this particular bug" and "the expected cost of fixing this bug under the new prioritization", the minimum aggregation function tends to show both an overall lower validation cost and a more stable performance.

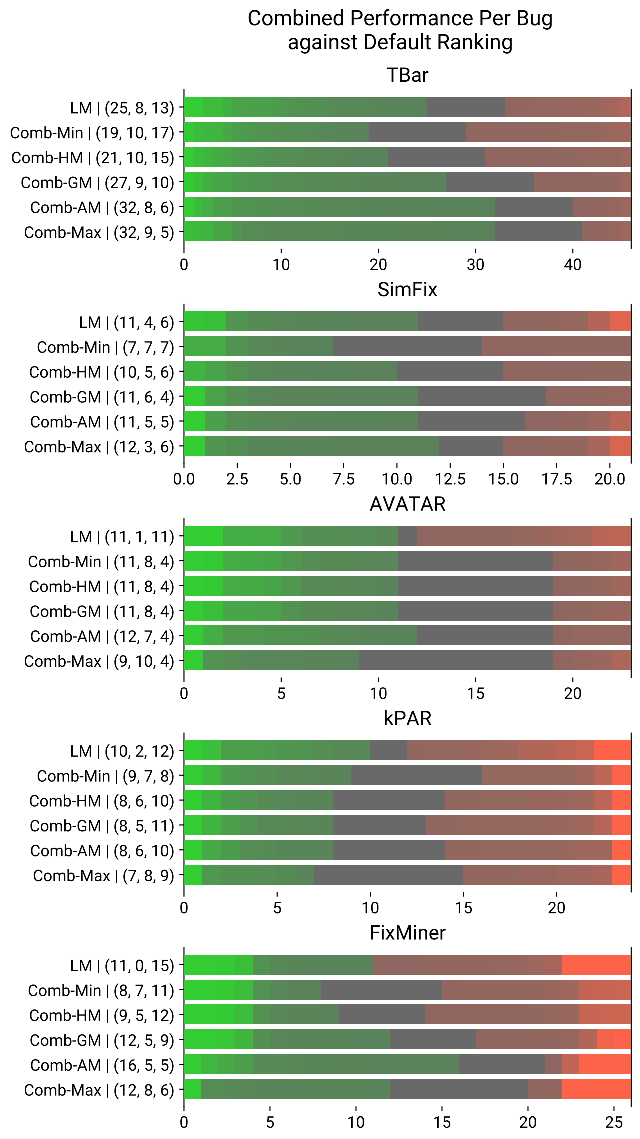

Figure 4: Comparison of per-bug performance for each aggregation technique for each tool against the default prioritization of APR tools. Green represents bugs in which the new prioritization did better than the default ranking, while red represents the opposite scenario. The color is more intense as the difference between the rankings increases.

Answer to RQ4: Different aggregation functions have different characteristics, with a tradeoff involved between cost and probability of improvement. Given that the main bottleneck of APR is validation cost itself, we would recommend using the minimum or harmonic mean when aggregating, which show both lower validation cost and overall more stable performance.

5.5 Qualitative Inspection

To conclude the section, we present examples illustrating what the LM 'thinks' about some example patches, to heighten the understanding of our approach. The listings in this section are best seen in color, as color is used to intuitively establish the likelihood of each token according to the LM. The concrete color code is provided in the first line of Listing 2. Only differing parts are highlighted, as the rest of the method is the same over all patches. The patch number presented in the listing titles is from the default prioritization of the APR tool in question.

Listing 2: TBar Lang57 Patch #1 (incorrect)

```
// Legend: token prob > 0.5  p > 0.1  p > 0.01  p <= 0.01

 -220,7 +220,7
  (...) boolean isAvailableLocale (Locale locale) {
-     return cAvailableLocaleSet.contains(locale);
+     return (cAvailableLocaleSet.contains(locale))
+             || (locale.getVariant().length() > 0);
  }
```

Listing 3: TBar Lang57 Patch #37 (correct)

```
 -220,7 +220,7
  (...) boolean isAvailableLocale (Locale locale) {
-     return cAvailableLocaleSet.contains(locale);
+     return availableLocaleList().contains(locale);
  }
```

Listing 4: SimFix Closure57 Patch #1 (incorrect)

```
   -194,9 +194,16
       Node target = callee.getNext();
  -    if (target != null) {
  -      className = target.getString();
  -    }
  +    if(child!=null){
  +      className=child.getString();
  +    }
     }
```

Listing 5: SimFix Closure57 Patch #27 (correct)

```
   -194,9 +194,16
       Node target = callee.getNext();
  -    if (target != null) {
  +    if(target!=null&&target.getType()==Token.STRING){
         className=target.getString();
       }
     }
```

TBar on Lang 57. Let us examine both an incorrect patch prioritized by TBar (Listing 2), and the correct patch placed in second place by the language model (Listing 3). Here, TBar starts off by appending some additional conditions to the existing return statement. However, the existing statement itself is buggy and unnatural to the language model. Furthermore, the language model deems the appended part unnatural: for example, the invocation of the `getVariant` method on `locale` is unlikely. (The token it suggests instead is `getAvailable`, which at least on a lexical level makes more sense.) Meanwhile, the correct patch is judged to be mostly natural, and thus prioritized as second place. The patch ranked at the top by the language model is a patch that replaces `availableLocaleList` with `availableLocaleSet`, which again is natural from a lexical perspective.

SimFix on Closure 57. Two patches are presented: the incorrect patch prioritized by SimFix (Listing 4), and the correct patch placed in second place by the language model (Listing 5). Notice how in the incorrect patch, the variable `target` is replaced with the variable `child`. The language model finds this variable change unlikely, particularly the use of the `child` variable, as the `target` variable was declared in the previous statement. Meanwhile in the correct patch, the use of the target variable is viewed upon favorably by the neural network, along with the relatively high probability use of the `getType` method, ranking this patch in the second place among the approximately 100 patches generated in this instance.

6 DISCUSSION

6.1 Threats to Validity

Internal Validity. Internal validity concerns whether the study has eliminated alternative explanations for the finding. The fact that the language model outperforms the default prioritizations of each APR tool may simply be due to randomness. To make sure that is unlikely, we performed a binomial test to verify that both the APR tool and language model are performing better than chance. Further, our technique reports performance improvement over a wide range of tools, heightening confidence in our results.

External Validity. Language models are a simple and widely applicable class of statistical models, and we believe that our technique could be widely available and useful to programming languages in which patch data is scarce. However, we have not empirically verified that language models actually improve performance in languages other than Java, and indeed it has not been verified that language models improve APR performance outside of the tools that we investigated.

6.2 Future Directions

Naturalness as a fitness function. As explained in Section 2.1, we did not investigate the effect of using naturalness on optimization-based techniques. However, the strong performance of using naturalness on template-based APR techniques naturally brings up the question of whether language models could shape the search space of GenProg, for example. Specifically, one could use naturalness as part of a fitness function, so that the optimization process does not go far astray from what we expect natural code to look like. Given that the search space is greater in such unconstrained optimization APR techniques than in template-based APR techniques, and that optimization-based APR is prone to *bloating* [32], it is possible that naturalness would be even more helpful when applied to optimization than when applied to template-based approaches.

The use of different language models. In this paper, we used a simple unidirectional neural network. However, there are many ways to calculate naturalness, from the simplest n-gram models to the larger and more complicated models like CodeBERT [8]. To the best of our knowledge, there has not yet been deep investigation as to which language models aid software engineering tasks the most, and why; if naturalness consistently provides useful results for software engineering tools as with our results, an in-depth analysis of various language models would be of interest.

7 CONCLUSION

In this work, we propose using language models to prioritize patches generated by multiple automatic program repair techniques. We find that language models contribute information to patch ranking that is complementary to what existing models were using. Language models allow a greater range of data to be used in identifying patch naturalness, outperforming a baseline that explicitly tries to model patch correctness. Combining ranking information from both the APR tool and the language model shows that the resulting ranking can save up to 65% of patch validation effort, saving effort in up to 70% of bugs inspected. We believe this work is a starting example of many ways in which language models could be incorporated with existing software engineering tools.

8 ACKNOWLEDGMENTS

Sungmin Kang and Shin Yoo have been supported by National Research Foundation of Korea (NRF) Grant (NRF-2020R1A2C1013629), Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), as well as by Institute for Information & communications Technology Promotion grant funded by the Korean government (MSIT) (No.2021-0-01001).

REFERENCES

[1] [n.d.]. A New Algorithm for Data Compression. http://www.pennelynn.com/Documents/CUJ/HTML/94HTML/19940045.HTM. Accessed: 2021-04-19.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3 (2019), 1 – 29.

[3] Moumita Asad, K. K. Ganguly, and K. Sakib. 2019. Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2019), 328–332.

[4] L. Chen and L. Zhang. 2020. Fast and Precise On-the-fly Patch Validation for All. *ArXiv* abs/2007.11449 (2020).

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, L. Pouchet, D. Poshyvanyk, and Monperrus Martin. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *ArXiv* abs/1901.01808 (2019).

[6] Michael L. Collard and J. Maletic. 2016. srcML 1.0: Explore, Analyze, and Manipulate Source Code. *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), 649–649.

[7] Thomas Durieux and Monperrus Martin. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)* (2016), 85–91.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, X. Feng, Ming Gong, Linjun Shou, Bing Qin, T. Liu, Daxin Jiang, and M. Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP*.

[9] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.

[10] V. Hellendoorn, Sebastian Proksch, H. Gall, and Alberto Bacchelli. 2019. When Code Completion Fails: A Case Study on Real-World Completions. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 960–970.

[11] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, 837âĂŞ847.

[12] S. Hochreiter and J. Schmidhuber. 1997. Long Short-Term Memory. *Neural Computation* 9 (1997), 1735–1780.

[13] J. Jiang, Yingfei Xiong, H. Zhang, Q. Gao, and X. Chen. 2018. Shaping program repair space with existing patches and similar code. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2018).

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. arXiv:cs.SE/2103.00073

[15] René Just, D. Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014*.

[16] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1073–1085.

[17] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811.

[18] A. Koyuncu, K. Liu, Tegawendé F. Bissyandé, D. Kim, J. Klein, Monperrus Martin, and Y. Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* 25 (2020), 1980–2024.

[19] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604.

[20] Xuan-Bach D. Le, D. Lo, and Claire Le Goues. 2016. History Driven Program Repair. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 213–224.

[21] K. Liu, A. Koyuncu, Tegawendé F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (2019), 102–113.

[22] K. Liu, A. Koyuncu, D. Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2019), 1–12.

[23] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 31–42.

[24] Kui Liu, Shangwen Wang, A. Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, P. Wu, J. Klein, Xiaoguang Mao, and Y. L. Traon. 2020. On the Efficiency of Test Suite based Program Repair A Systematic Assessment of 16 Automated Repair Systems for Java Programs. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 615–627.

[25] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298âĂŞ312.

[26] Thibaud Lutellier, H. Pham, L. Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020).

[27] S. Mechtaev, J. Yi, and A. Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035.

[29] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 428–439.

[30] R. Saha, Yingjun Lyu, H. Yoshida, and M. Prasad. 2017. Elixir: Effective object-oriented program repair. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), 648–659.

[31] Seemanta Saha, R. Saha, and M. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), 13–24.

[32] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. 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 (ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543.

[33] S. Tan, H. Yoshida, M. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016).

[34] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering.*

[35] Ming Wen, J. Chen, Rongxin Wu, Dan Hao, and S. Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (2018), 1–11.

[36] Qi Xin and S. Reiss. 2017. Leveraging syntax-related code for automated program repair. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2017), 660–670.