Out of Context: How important is Local Context in Neural Program Repair?

Julian Aron Prenner
prenner@inf.unibz.it
Free University of Bozen/Bolzano
Bozen/Bolzano, Italy

Romain Robbes
romain.robbes@labri.fr
CNRS, Univ. Bordeaux, Bordeaux INP, LaBRI
Talence, France

ABSTRACT

Deep learning source code models have been applied very successfully to the problem of automated program repair. One of the standing issues is the small input window of current models which often cannot fully fit the context code required for a bug fix (e.g., method or class declarations of a project). Instead, input is often restricted to the local context, that is, the lines below and above the bug location. In this work we study the importance of this local context on repair success: how much local context is needed?; is context before or after the bug location more important? how is local context tied to the bug type? To answer these questions we train and evaluate Transformer models in many different local context configurations on three datasets and two programming languages. Our results indicate that overall repair success increases with the size of the local context (albeit not for all bug types) and confirm the common practice that roughly 50-60% of the input window should be used for context leading the bug. Our results are not only relevant for researchers working on Transformer-based APR tools but also for benchmark and dataset creators who must decide what and how much context to include in their datasets.

CCS CONCEPTS

• **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → *Machine learning*.

KEYWORDS

automated program repair, data-driven software engineering

1 INTRODUCTION

Deep learning-based methods have shown promising performance in Automated Program Repair (APR), leading to the subfield of Neural Program Repair (NPR) [2, 19, 25, 42]. NPR models are trained

in such a way that, given a snippet of buggy code as input, the model outputs fixed code. This input is usually comprised of two parts: (I) the buggy code itself and (II) the surrounding *local context* code, that is code that comes before the bug location (pre-context) and after (post-context). The bug location, together with its pre-context and post-context we refer to as the *context window*.

Importance of context. The significance of the local context is twofold. First, it identifies the purpose of a given code snippet, giving the model implicit hints as to how the bug should be fixed. Second, local context is an important source of repair ingredients, that is, code elements relevant for a fix (e.g., variable, field or methods names). Section 2 provides more background on context, context variants, and its relation to datasets and NPR models.

Example. Figure 1 shows an example. To fix the bug in the presented snippet, an APR model should replace the identifier `instructions_list` with the identifier `some_list`. Luckily, this identifier appears within the context (`some_list`) and the model is able to pick it up and correctly fix the bug. However, it is easily imaginable that there exist cases in which important identifiers (and other important fix-related knowledge) fall outside the provided context (out of context), especially if the window is small.


```
out of context
        if instruction[1] == 'dec':
            instruction[1] = '-='
        instruction = ' '.join(instruction)
        some_list[i] = instruction

pre-context
    for register in registers:
        reg_values.update({register: 0})
def run_instructions(some_list):
    setup(some_list)
    max_value_ever = 0
    for inst in instructions_list some_list:

post-context
        if eval(inst[inst.index(' if') + 4:]) == True:
            exec(inst[0:inst.index(' if') + 1])
            if max(reg_values.values()) > max_value_ever:
                max_value_ever = max(reg_values.values())
    return max(reg_values.values()), max_value_ever
print run_instructions(instructions_list)
```

Figure 1: A patch from the TSSB-3M [29] dataset. To fix the bug, the buggy token `instructions_list` has to be replaced by the correct token `some_list`, found in the context (`some_list`). The context window has a total size of 10 lines (5 lines of pre-context and 5 lines of post-context); it excludes the first 4 lines and the last line in the figure, which are not part of the model input. Note: we shortened the `inst` identifier to improve readability.

This work. So far, little work has investigated the importance of such local context in NPR and its effect on repair success. How many lines of context code do we actually need? Should we prefer context before, or after the bug location? Are there bug types that require more or less context? To answer these and more questions we train and evaluate multiple variants of a Transformer-based NPR model that *can* leverage a large context. Section 3 details our methodology to answer the following research questions:

RQ1. *How important is local context for repair success?* We study multiple context sizes, ranging from a single line up to 28 lines on both sides (56 lines) on three datasets (MegaDiff [21], TSSB [29], and ManySStuBs4J [10], see Table 1), totalling several hundreds of thousands of bugs.

RQ2. *How do different bug types and complexity (number of changes) respond to different context sizes and context window positions?* Both, MaySStuBs4J [10] and TSSM-3M [29] classify bugs into several bug types or bug patterns. We use this bug type labels to analyze how context size affects repair success for bugs of different types. We perform a similar analysis also for the number of changes of a bugfix.

RQ3. *What is the optimal context window position? In other words, given a fixed context budget, how should it be divided among pre-context and post-context?* We experiment with six different context window positions (for four different context window sizes), from only pre-context over several combinations to only post-context.

RQ4. *Is there a connection between the model size (number of parameters), the number of sampled fix candidates and context?* With more context, the amount of fix ingredients increases. We hypothesize that in order to *fully exploit* context, model size should increase, as should the number of samples. We investigate if this is indeed the case.

Findings. We find that the number of context lines strongly influences repair success, leading to relative improvements up to 29%. We observe this for context sizes up to over 50 lines (significantly larger than the current practice), however performance varies with bug types and change sizes. As for the optimal context window position, we observe allocating roughly 50-60% for pre-context and the rest for post-context yields the best performance; in addition, ensembling multiple contexts improves performance. Finally, models with more parameters or samples benefit similarly from increased context. Section 4 details our findings.

Implications. We hope that our work will help NPR researchers to make the most of their models by including a sufficient amount of local context, and clearly document their context choices. Moreover, we call on dataset creators to include more local context in their datasets. As part of this study, we re-mined the TSSM and ManySStuBs4J datasets, as the the original datasets did not provide enough context (we will release the subset of bugs used in this work with a larger context size as part of our replication package.). We discuss further challenges and opportunities with increased context in Section 5, before discussing the limitations of our work (Section 6) and concluding (Section 7).

2 BACKGROUND AND RELATED WORK

Context is an important factor for repair success in NPR [19]. For one, context may act as an abstract "description" of the code. For example, in Figure 1, identifiers such as instruction_list and register indicate that this code may be related to a register-based virtual machine implementation. This "description" may help the model to find and apply the correct fix. Second, context is an important source for ingredient code. Again referring to Figure 1, we see a variable misuse bug. Instead of the correct identifier `some_list`, the wrong identifier `instructions_list` is used. The correct variable name `some_list` does appear in the pre-context while the "buggy" identifier `instructions_list` appears neither in the pre-context nor the post-context. It is easy to see that bugs where such ingredients do not appear inside the local context, either because they happen to be far away from the bug location or because the context window is too small to include them, are very hard to fix.

2.1 Context in APR and NPR

Context as Source for Ingredients. In a previous study, Yang et al. [45] have identified several levels of ingredient code origin, among them most importantly: I) intrinsic ingredients, that is ingredients implicitly coming from the specifications of the programming language. In Figure 1 these would include Python keywords (e.g., def, for or if) or builtin functions such as eval or max). II) *method level* ingredients, which includes code elements from the method definition that contains the bug. In Figure 1, `some_list` is a method level ingredient. III) ingredients on the *file or class level*, that is, code located in the same file or class as the bug location (e.g., in Figure 1 parts of the preceding function definition appear in the pre-context) IV) donor code coming from the surrounding *package or module*, and finally V) ingredients on the *program/project level*, which includes a project's or program's entire codebase.

Use of local context. While ingredients on the class and project level are often crucial for a successful repair [45], they might spread over thousands of lines of code. Traditional generate-and-validate methods have been able to exploit project level ingredients [6, 30, 49]. However, so far, NPR systems have been quite limited in the amount of code they can consume. For instance, Chen et al. [2] use the surrounding method code as model input, but only experiment with short methods of 50 and 100 tokens of length. Using the method as a context boundary was also done in CoCoNuT [19] and later CURE [8] as well as in more recent work on large language models in APR [3, 7, 41]. In general, we find that previous work often lacks detail about context handling. For instance, RewardRepair [47] is said to use 10 lines of context, but not whether this means 10 lines in total, or for each side. Similarly, since methods can substantially vary in size, the amount of context will be highly variable.

Context enrichment. To give the model access to information often not found in the local context, model input may be "enriched" with further information, in particular with carefully selected file or project level context. For instance, SequenceR [2] adds, in addition to local context, class level information such as class field declarations and method signature stubs. A similar approach was also used for RewardRepair [48]. SelfAPR [46] adds diagnostic information such as compiler or runtime errors from test executions to model

input. FitRepair [40] uses simple text similarity metrics to select possibly relevant identifiers from out of context code and includes them in the model input. Rete [24] learns a project-independent representation of variable namespaces based on def-use chains that is used to rank patches for program repair.

Code Search and Retrieval. A series of work investigates the use of code search in APR (e.g., ssFix [43], sharpFix [44], or LSRepair [17]). In a very broad sense, this extends the context to entire code corpora and millions of lines of code. However, code from different projects is unlikely to match the code under repair which necessitates adaptation and translation steps [43, 44]. Cedar [22] and RAP-Gen [36] are two APR systems that combine code LMs with retrieval.

Architectural. CoCoNuT [19] and DLFix [14] use architectures with special features to better exploit context information. Finally, a number of works explores the idea of fine-tuning the model on the project under repair in order to encode relevant project level context directly "into model weights" [40, 46].

This work. In this study, we focus on local context, that is n lines of code surrounding the bug location, either before or after. This includes intrinsic, as well as method level ingredients and in some cases ingredients at the class level (if e.g., if neighboring function definitions or field declarations fall within the context). A systematic study of enrichment techniques is left as a future work.

2.2 Context and Transformers

Transformer window size. Many state of the art neural program repair (NPR) models are based on the Transformer [35] architecture [1, 4, 42, 46, 47]. One important limitation of this architecture is its limited input window size, which is fixed at training time. Although there have been successful efforts to expand the input window of general Transformer models, recent NPR models use input windows of only several hundreds of tokens. For instance, VulRepair [4], FitRepair [40] and AlphaRepair [42], RewardRepair are all limited to only 512, SelfAPR [46] to 768 model tokens. Often, the amount of context is simply determined by the space left in the input window after the buggy code was placed in it [42].

Position embeddings. By itself a Transformer has no notion of sequences; it sees its input as a set of elements. The original Transformer architecture added *position embeddings* to give greater weight to a token's neighbour [35]. This bias allows it to model its input as sequences. The original position embeddings are *absolute* and tied to the model's input window, causing very poor generalization to longer sequences. Alternative ways to encode token position have been proposed, including relative position embedding [31], rotary position embeddings (RoPE [32]), and biases to the attention [27]. These allow fine-tuning to longer sequences, and (to a limited degree) extrapolation to longer sequences without fine-tuning [27].

The T5/CodeT5 model in NPR. T5 [28] and CodeT5 [37] a variant specifically pre-trained for code-related tasks are popular Transformer models for sequence-to-sequence (seq2seq) tasks. Since program repair can be naturally represented as a seq2seq task by letting the input sequence be the buggy code and the output sequence the fixed code, this model has been used extensively in the

recent NPR literature. Berabi et al. [1] fine-tune T5 on coding errors provided by ESLint. Ye et al. [46, 47] use CodeT5 as the basis for both, RewardRepair [47] and SelfAPR [46]. Fu et al. [4] develop VulRepair, a CodeT5-based Automated Vulnerability Repair system. Kim et al. [12] study the effect of code abstraction techniques in NPR and use T5/mT5 models in all of their experiments. Similarly, Xia et al. [40] use CodeT5 in their analysis of the plastic surgery hypothesis in NPR.

This work. In addition to its widespread use in NPR, we use CodeT5 [37] for a very specific reason: its T5 architecture [28], unlike most, uses *relative position embeddings* [31]. This allow us to fine-tune a T5 model with an arbitrary input window size [27, 28]. To our knowledge, we are the first to fine-tune a model for a larger context window in NPR. For all experiments in this work use an input window size of 1024, that can fit *twice as much* context as most of the models mentioned above.

2.3 Context and Datasets

Most existing datasets used in NPR are large collections of bug fixes mined from code repository commits. In the following, we analyze how context is handled on the "dataset level".

BFP. BFP [34] is a collection of over 65,000 bug fixes mined from GitHub. Each dataset sample includes the full method surrounding the bug location, although the dataset focuses on short methods.

CoCoNuT. In the same fashion, the dataset used by CoCoNuT [19] was mined from open-source project commits and includes context only up to the method boundary. In particular, the CoCoNuT dataset splits hunks of changes occurring in a single commit into separate dataset instances. Further, whitespace, including newlines is stripped from the dataset instances which makes estimating the context size in terms of number of lines very difficult.

ManySStuBs4J. ManySStuBs4J [10] is a dataset of over 150,000 single statement Java bugs categorized into 16 bug patterns. We find that the diffs in ManySStuBs4J have a total length of only 13 lines of code (including the bug itself). The dataset provides the commit hashes and repository identifiers for all dataset instances.

TSSM-3M. TSSM-3M [29] is a dataset of over three million single statement Python bugs classified into 20 bug patterns loosely following the categorization of ManySStuBs4J. Our analysis shows that the examples (diffs/patches) in TSSM-3M have a median length of only 9 lines (mean 8.8), that is, less than 4 lines of pre- and post-context. Here too, commit hashes and repository names are provided for all instances.

MegaDiff. MegaDiff [21] contains over 660,000 Java diffs with changes ranging from 1 to 40 lines. They come with full file-level context, containing all files affected by changes in a single diff.

Others. ManyBugs [13] is a dataset of 185 bugs in open-source C software (e.g., libdiff, zlib, PHP). For each bug, full project-level context is available. IntroClass [13] is a collection of 998 buggy simple C programs written by students. These programs are rather short (13-23 lines of code [13]), but full program code is provided.

QuixBugs [15] is a collection of 40 bugs in classical implementations of algorithms and data structures and comes in a Python and Java version.

CodeFlaws [33] and RunBugRun [26] are datasets of bugs mined from competitive programming sites. While both datasets provide full context, the contained programs solve a single problem and are thus short (e.g., 36 lines of code (avg.) in CodeFlaws [33]).

This work. We use the MegaDiff, ManySStuBs4J and TSSM-3M datasets (Table 1). MegaDiff was chosen because it provides full file level context, ManySStuBs4J and TSSM-3M because they contain all the required information for re-mining. As neither ManySStuBs4J nor TSSM-3M provide sufficient amount of context, we had to re-mine selected subsets with full file-level context (see Section 3.1). BFP and the CoCoNuT dataset were not considered in this work as they do not provide full file level context (method level only) nor were we able to find commit information necessary for re-mining. ManyBugs [13], IntroClass [13], RunBugRun [26] and QuixBugs [15] were not considered either because of dataset size or program length.

3 METHODOLOGY

Table 1: Datasets used in this work, along with the number of samples used (subset) for training and evaluation.

Dataset	Train	Test	Lang.	Labels
MegaDiff [21]	201,358	22,479	Java	✗
ManySStuBs4J [10]	-	12,714	Java	✓
TSSM-3M [29]	424,873	46,791	Python	✓

To study the effect of local context on repair success, we (I) re-mine with full context a subset of examples from existing datasets, which we pre-process for different context size (Section 3.1) (II) fine-tune various configurations of our model on the corresponding training sets (Section 3.2) and finally (III) generate and evaluate bugfixes for all bugs in the test set to carry out our experiments (Section 3.3).

3.1 Re-Mining and Pre-Processing

Datasets. We study three datasets that cover two programming languages: MegaDiff [21], ManySStuBs4J [10], and TSSM-3B [29]. In addition, ManySStuBs4J and TSSM-3B provide SStuB patterns, that is, bug type labels; we use these to analyze the effect of context size on different bug types in RQ2. For better comparability with the other datasets, MegaDiff$_{SB}$ (simple bugs) denotes the subset of MegaDiff bugs that require at most two changes to fix.

Filtering. Following previous work [2, 14, 19, 42, 47], our study focuses on single-hunk bugs. This is all the more important as the presence of multiple hunks would involve multiple contexts, which would increase the complexity of our study. As such, multi-hunk bugs in MegaDiff were filtered out. Note that we keep examples with multiple changes in a single hunk. We observed that in ManySStuBs4J, some commits included multiple bug patterns and were split in several dataset instances; such cases were also considered multi-hunk edits and consequently filtered out. No filtering was

necessary for TSSB-3M which only contains single-statement and thus single-hunk bugs; we focus on a subset of about 450,000 bugs due to limited computational resources. Table 1 shows the size of the datasets after filtering. Due to its final size, we use ManySStubs4J as a test set (with MegaDiff as training set).

Re-mining commits. As mentioned previously, in our experiments we require a large local context (up to 56 lines). Unfortunately, this is neither provided by ManySStuBs4J nor by TSSM-3B. As repository identifiers and commit hashes are provided in both cases, we re-mine dataset instances with full file level context. This was done using the GitHub API. On rare occasions, commits could not be fetched, possibly because the corresponding repository has been deleted or is no longer public. In contrast to the other two datasets, MegaDiff [21] is released with full file-level context. Thus, re-mining was not necessary.

```
def run_instructions(some_list):
    setup(some_list)
    max_value_ever = 0
<CHANGES>
    for inst in instructions_list:
<CHANGEE>
        if eval(inst[inst.index('_if') + 4:]) == True:
            exec(inst[0:inst.index('_if') + 1])
            if max(reg_values.values()) > max_value_ever:

<CHANGES>
    for inst in some_list:
<CHANGEE>
```

Figure 2: Unified bug format. Model input above the zigzag line (with three lines of pre and post context), output below. <CHANGES> and <CHANGEE> indicate the buggy code section.

Pre-Processing. We strip all empty lines from the dataset examples. To unify the examples from the three datasets we parse the diffs and select context code lines, buggy code lines and the ground-truth (i.e. fixed) code lines; we transform them into the format specified in Figure 2. In line with previous work [19, 42, 46], we assume perfect fault localization, that is, we assume the bug location is known. This avoids confounding of localization and repair performances [16]. We indicate the start and end of the buggy code section with marker tokens (<CHANGES> and <CHANGEE>), similar to SequenceR [2]. In case of multiple changes, each change is marked with tokens and surrounded by its corresponding *n* context lines. Overlapping context, that is, context lines shared by multiple changes, are fused into single blocks. The target model output consists of the fixed version of the buggy input lines (without context).

For each configuration in Table 2 we generate training test sets where we select the corresponding number of pre/post-context lines. Due to limited computational resources, we skip some TSSM context sizes. In early experiments we noticed diminishing returns above 50 lines of context, which is why we stop at 56 context lines.

3.2 Training and Evaluation

We implement all of our experiments on top of Huggingface's `transformers` library [39]. We fine-tune the 60M pre-trained CodeT5 model on the configurations, that is, different datasets and pre-context and post-context sizes, as listed in Table 2. We then generate

Table 2: Context configurations in this study (pre-context lines/post-context lines).

Datasets	Configurations
MegaDiff [21] ManySStuBs4J [10]	1/1-28/28
TSSM-3M [29]	1/1-16/16, 18/18, 20/20, 22/22, 24/24, 26/26, 28/28
All[1]	5/0, 4/1, 3/2, 2/3, 1/4, 0/5, 10/0, 8/2, 6/4, 4/6, 2/8, 0/10, 20/0, 16/4, 12/8, 8/12, 4/16, 0/20, 30/0, 24/6, 18/12, 12/18, 6/24, 0/30, 40/0, 32/8, 24/16, 16/24, 8/32, 0/40

[1] configurations for different context window positions (sizes 5, 10, 20, 30 and 40)

five fix candidates for each bug in the corresponding test set. The generated model output is evaluated using an exact-match metric.

Hyper-Parameters. To obtain a fair basis for comparison, we use the same hyper-parameters for all context configurations (unless explicitly specified). We train with FP16 precision, a learning rate of 1×10^{-4}, and a batch size of 12, accumulated over two steps on a consumer-grade NVIDIA RTX 3090 with 24GB of memory. For generation we use beam search with five beams and a maximum length of 1024 generated tokens.

Evaluation. We calculate repair success (accuracy) as fraction of examples in the test set where at least one of the five generated fix candidates (i.e. top-5) lexically matches the ground-truth fix, ignoring any whitespace. Exact matching as a metric for repair success was used in previous work [1, 4].

Tokenization. Since our model's input window is limited to a 1024 tokens, there is a limit at which further increasing context has no effect, as the exceeding input would be simply truncated away. With a median token count of 592 and an upper quartile of 699 at 56 lines of context we are well within this limit. At 56 lines of context, truncation is only necessary for 2.2% of examples. Truncation is done by dropping the last n excessive tokens.

3.3 Experiments

To answer our research questions, we carry out four main experiments: 1) we train and evaluate repair success for different context sizes (using symmetric context windows), 2) we further analyze the relationship between repair success, change type, and change size, 3) we train and evaluate repair success for different context window positions (i.e., asymmetric context window or different amounts of per-context and post-context) and finally 4) we train and evaluate repair success for a larger model (220M parameters) and for a larger number of generated fix candidates (small model).

3.3.1 Context Size (RQ1). For each context size of 1 to 28 lines of code (on both sides) and each of the three datasets (MegaDiff, TSSM-3B and ManySStuBs4J; see Table 2 for a list of all context configurations) we fine-tune and evaluate a CodeT5 Transformer as described in 3.2.

3.3.2 Change Type and Change Size (RQ2). We analyze the performance of the models from RQ1 to get finer-grained insights. We use the labels of TSSM and ManySStuBs4J to compare the performance

of models trained on different context sizes for several types of changes. In the same spirit, we compare how the performance of these models varies with change size on MegaDiff.

```
    instruction = ' '.join(instruction)
    some_list[i] = instruction
    for register in registers:
        reg_values.update({register: 0})
def run_instructions(some_list):                    5/0
    setup(some_list)                               (0%)
    max_value_ever = 0
    for inst in instructions_list some_list:
        if eval(inst[inst.index(' if') + 4:]) == True:
 0/5        exec(inst[0:inst.index(' if') + 1])
(100%)      if max(reg_values.values()) > max_value_ever:
                max_value_ever = max(reg_values.values())
    return max(reg_values.values()), max_value_ever
print run_instructions(instructions_list)
```

Figure 3: The two most extreme window positions 5/0 (0% – only pre-context ◯) and 0/5 (100% – only post-context ◯) for a context size of 5.

3.3.3 Context Window Position (RQ3). We carry out a series of experiments with asymmetric window sizes. Here, we keep the context window at a constant size, however, by adjusting pre-context and post-context, the window is slid over the bug location. For example, for a context size of 5 we evaluate at 5/0, 4/1, 3/2, 2/3, 4/1 and 0/5, where x/y denotes x lines of pre-context and y lines of post-context. Figure 3 illustrates this for the two extreme window positions 5/0 and 0/5. All intermediate positions are obtained by "sliding" the window from top to bottom. The window position can also be described as the percentage of post-context: at position $n\%$, $n\%$ of the window are filled with post-context and $100\% - n\%$ with pre-context (e.g., 0% for position 5/0, and 100% for 5/0) We do this for windows of 5, 10, 20, 30 and 40 lines and 7 sliding positions (0, 20, 40, 50, 60, 80, and 100%), adjusting the sliding step size accordingly (see Table 2 for a list of context configurations). Training and evaluation are performed as outlined above.

3.3.4 Model Size and Fix Candidate Count (RQ4). We hypothesize that a larger model, or a larger number of generated fix candidates, can make better use of larger contexts. We study whether the margins between smaller and larger models (or between different numbers of fix candidates) increase over-proportionally with larger context sizes.

Larger model. For a selected number of context sizes (1, 7, 14, 21 and 28 lines on both sides) we re-run the context size experiment on a larger version of the CodeT5 model (220M). This experiment is also only carried out for MegaDiff (training and evaluation) and ManySStuBs4J (evaluation only). We double the number of accumulation steps in order to be able to train the larger model with the same (accumulated) batch size of 12 on the same hardware.

More fix candidates. For all other experiments we use five fix candidates per bug, sampled using beam search. Here, we try 10 and 15 fix candidates per bug (evaluating using top-10 and top-15).
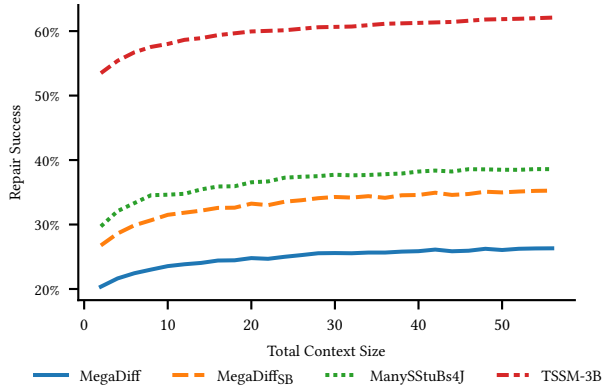
Figure 4: Repair success as a function of context size.

To account for the increased memory requirements when using more beams, we reduce the maximum length of generation from 1024 to 256 tokens. Note that the model only needs to generated the buggy code section, which rarely exceeds 256 tokens. We also enable *early stopping* to speedup the generation process. With early stopping, the search is stopped as soon as *n* complete solutions have been generated (otherwise the search may continue to find better solutions). For a fair comparison, we also regenerate the top-5 fix candidates with early stopping and a 256 token limit.

4 RESULTS

At a glance, we find that: (I) overall, more context increases repair success, but not always consistently (Section 4.1) (II) the importance of context size strongly depends on the specific bug type and change size (Section 4.2) (III) context windows which are centered around the bug location yield the best performance, but extreme windows are useful in ensembles (Section 4.3) and (IV) we see no convincing evidence that either larger models or a higher number of fix candidates can better exploit larger context sizes (Section 4.4).

4.1 Context Size (RQ1)

We analyze the symmetric context configurations (i.e., *n* lines of context in the pre-context and the post-context) to estimate the importance of local context on repair success. As shown by Figure 4, repair success steadily increases as more local context is available.

MegaDiff. For MegaDiff, repair success increases from 20.4% with a single line of context to 26.3% with 28 lines of context on both sides. For simple bugs (MegaDiff$_{SB}$) repair success is considerably higher with numbers ranging from 26.8% (one line of context) to 35.3% (28 lines of context).

ManySStuBs4J. For the ManySStuBs4J dataset, repair success ranges from 29.7% with single-line context, to 38.6% with 27 lines of context. Here, repair success does not peak at the maximum context size of 28/28, where performance is slightly lower (-0.008%).

TSSB-3M. Overall performance was best for TSSB-3M, where repair success is beyond 50% for all context sizes. Repair success starts at 53% with a single-line context and reaches 62% with 28 lines of context on both sides.

4.1.1 Consistency of Results. Our results indicate that, *in general*, growing context increases repair success. The ideal case is pictured in Figure 5: as the correct ingredients enter the context, the model succesfully use them. On the other hand, it was conjectured in previous work that too much context may "drown" the actual bug in noise, confuse the model and thus stymie repair success [19]. We do observe this, as pictured in Figure 6. Moreover, upon inspecting our results, we find that the model often "spontaneously" fails to fix a bug at a certain context size *c*, despite it correctly fixing the bug at one or more lower context sizes. To get more insight on this, we group bugs correctly fixed in at least *some* context size into four categories:

(1) bugs that are fixed for all context sizes;
(2) bugs that *improve* with context (they are not correctly fixed below a context size *c*, but for all context sizes equal or larger than *c*);
(3) bugs that *degrade* with context (they are *only* be fixed below a context size *c* and not for context sizes equal or larger than *c*);
(4) *erratic* bugs with more complex patterns (e.g., the model might correctly fix a bug at context sizes 18, 21, or 24, without clear reason of why it failed at, say, context size 20).

We find that 37% of bugs were fixed for all context sizes (case 1). Bugs with consistent repair success starting from some context size *c* (case 2, Figure 5) are surprisingly low with roughly 9%. On the other hand, we find that the number of bugs that can only be fixed at small context sizes (case 3, Figure 6) are rare (~1%). Finally, the remaining 35% show some degree of erratic behavior (case 4), which we found to be surprisingly high. In particular, roughly 5% of the bugs show an "island pattern", where a bug can be correctly fixed for a contiguous range of context sizes that is surrounded on both sides by context sizes for which models could not find a fix. The inverse case, that is, a fix can be found *except* for a contiguous block of context sizes in the middle appears with a frequency of about 4.5%. These inconsistencies indicate that the models lacks robustness; further implication of this are discussed in Section 5.

```
✓₅  game.player.updateMotion(game.player.getPosition(), v, ...);
✓₄  //other Ben's doing...
✓₃  if(!v.equals(Vec3.zero))
✓₂  {
✗₁      game.transmitPlayerPosition();
        transmittedStop = false;
✗₁  }
✓₂  else if(!transmittedStop)
✓₃  {
✓₄      game.transmitPlayerPosition();
✓₅      transmittedStop = true;
```

Figure 5: Bug from MegaDiff, where an assignment to `transmittedStop` needs to be added. To succeed, the model requires at least two lines of context on both sides (✓₂). This is likely due to `transmittedStop` coming into context (◯).

---

[1] The likely culprit here is the string `"PApplet.onDestroy() called"`: if it is included in the context, the model predicts fixes that include PApplet, such as inserting `PApplet.onDestroy();`, `pApplet.onDestroy();` or `PApplet.onResume();`

```
✗5  public void resume() {
✗4  }
✗3  public void onDestroy() {
✗2      System.out.println("PApplet.onDestroy()_called");
✓1      super.onDestroy();
        finish();
✓1  }
✗2  ////////////////////////////////////////////////////////
✗3  // ANDROID SURFACE VIEW
✗4  SurfaceView surfaceView;
✗5  SurfaceHolder surfaceHolder;
```

Figure 6: Bug from MegaDiff where a missing call to `finish()` needs to be added. This bug is correctly fixed only with a context of one line on both sides (✓1). The model is likely led astray by code highlighted in blue (◯) [1].
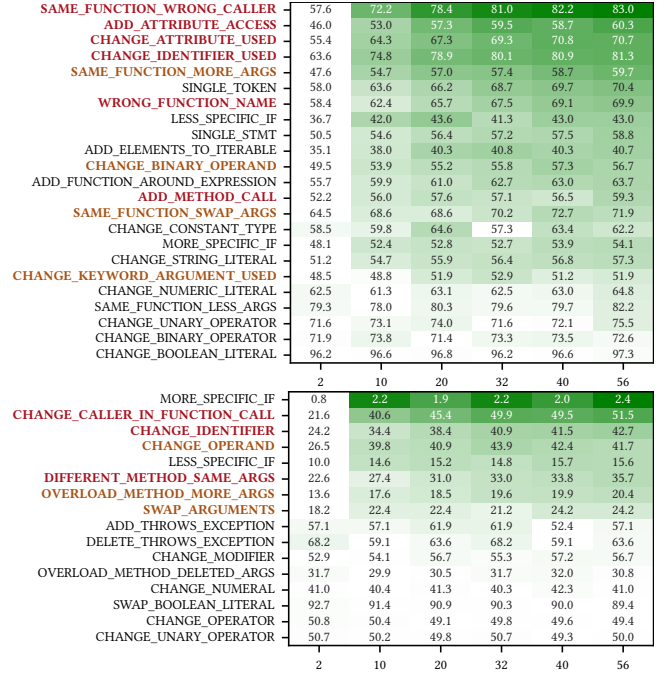
Answer to RQ1

Overall, local context has a strong effect on repair success. For MegaDiff *symmetrically* increasing total local context from two lines to 56 lines increases repair success by almost 6%. For simple bugs (ManySStuBs4J, TSSM-3M and MegaDiff$_{SB}$) performance increased by over 8%. Given this, we see that local context is an important factor for repair success. However, a large amount of bugs are inconsistently repaired across context sizes.

4.2 Effect of Bug Type and Change Count (RQ2)

As mentioned previously, the labels (SStuB patterns) in Many-SStuBs4J and TSSB-3M allow us to analyze the effect of context size for different label types. Similarly, a number of bugs in MegaDiff have multi-change fixes, making a similar analysis possible for a bug's change count (i.e., the number of changed lines in the diff fixing it).

4.2.1 Bug Type. We visualize the effect of context on bug type as a heatmap highlighting relative improvements (Figure 7). While some bug types benefit from a larger context, for others, more context hardly makes a change or even slightly lowers performance.

Responding patterns. For both datasets, we see a very strong effect for the "change caller" bug types (> 25% absolute difference in repair success between 2 and 58 lines of context). For ManySStuBs4J, `MORE_SPECIFIC_IF` and `LESS_SPECIFIC_IF` are among the top-5 most responsive bug patterns. For TSSM-3B these patterns do see improvement, but to a much lesser degree and, interestingly, in reverse order. A relatively strong response can also be seen for the "more arguments" pattern, present in both datasets under slightly different names. Notably, the "opposite" pattern, that is the removal of arguments responds badly in both datasets. Bugs that require the change of an identifier (`CHANGE_IDENTIFIER` and `CHANGE_IDENTIFIER_USED`) also benefit from larger context sizes in both datasets. In general we see that bug patterns that involve or likely involve adding or changing identifiers (e.g., function names, attributes names, operands, arguments) respond strongly to context sizes. This corroborates the theory that context serves as a pool of useful ingredients.

	2	10	20	32	40	56
SAME_FUNCTION_WRONG_CALLER	57.6	72.2	78.4	81.0	82.2	83.0
ADD_ATTRIBUTE_ACCESS	46.0	53.0	57.3	59.5	58.7	60.3
CHANGE_ATTRIBUTE_USED	55.4	64.3	67.3	69.3	70.8	70.7
CHANGE_IDENTIFIER_USED	63.6	74.8	78.9	80.1	80.9	81.3
SAME_FUNCTION_MORE_ARGS	47.6	54.7	57.0	57.4	58.7	59.7
SINGLE_TOKEN	58.0	63.6	66.2	68.7	69.7	70.4
WRONG_FUNCTION_NAME	58.4	62.4	65.7	67.5	69.1	69.9
LESS_SPECIFIC_IF	36.7	42.0	43.6	41.3	43.0	43.0
SINGLE_STMT	50.5	54.6	56.4	57.2	57.5	58.8
ADD_ELEMENTS_TO_ITERABLE	35.1	38.0	40.3	40.8	40.3	40.7
CHANGE_BINARY_OPERAND	49.5	53.9	55.2	55.8	57.3	56.7
ADD_FUNCTION_AROUND_EXPRESSION	55.7	59.9	61.0	62.7	63.0	62.2
ADD_METHOD_CALL	52.2	56.0	57.6	57.1	56.5	59.3
SAME_FUNCTION_SWAP_ARGS	64.5	68.6	68.6	70.2	72.7	71.9
CHANGE_CONSTANT_TYPE	58.5	59.8	64.6	57.3	63.4	62.2
MORE_SPECIFIC_IF	48.1	52.4	52.8	52.7	53.9	54.1
CHANGE_STRING_LITERAL	51.2	54.7	55.9	56.4	56.8	57.3
CHANGE_KEYWORD_ARGUMENT_USED	48.5	48.8	51.9	52.9	51.2	51.9
CHANGE_NUMERIC_LITERAL	62.5	61.3	63.1	62.5	63.0	64.8
SAME_FUNCTION_LESS_ARGS	79.3	78.0	80.3	79.6	79.7	82.2
CHANGE_UNARY_OPERATOR	71.6	73.1	74.0	71.6	72.1	75.5
CHANGE_BINARY_OPERATOR	71.9	73.8	71.4	73.3	73.5	72.6
CHANGE_BOOLEAN_LITERAL	96.2	96.6	96.8	96.2	96.6	97.3

	2	10	20	32	40	56
MORE_SPECIFIC_IF	0.8	2.2	1.9	2.2	2.0	2.4
CHANGE_CALLER_IN_FUNCTION_CALL	21.6	40.6	45.4	49.9	49.5	51.5
CHANGE_IDENTIFIER	24.2	34.4	38.4	40.9	41.5	42.7
CHANGE_OPERAND	26.5	39.8	40.9	43.9	42.4	41.7
LESS_SPECIFIC_IF	10.0	14.6	15.2	14.8	15.7	15.6
DIFFERENT_METHOD_SAME_ARGS	22.6	27.4	31.0	33.0	33.8	35.7
OVERLOAD_METHOD_MORE_ARGS	13.6	17.6	18.5	19.6	19.9	20.4
SWAP_ARGUMENTS	18.2	22.4	22.4	21.2	24.2	24.2
ADD_THROWS_EXCEPTION	57.1	57.1	61.9	61.9	52.4	57.1
DELETE_THROWS_EXCEPTION	68.2	59.1	63.6	68.2	59.1	63.6
CHANGE_MODIFIER	52.9	54.1	56.7	55.3	57.2	56.7
OVERLOAD_METHOD_DELETED_ARGS	31.7	29.9	30.5	31.7	32.0	30.8
CHANGE_NUMERAL	41.0	40.4	41.3	40.3	42.3	41.0
SWAP_BOOLEAN_LITERAL	92.7	91.4	90.9	90.3	90.0	89.4
CHANGE_OPERATOR	50.8	50.4	49.1	50.0	49.6	49.4
CHANGE_UNARY_OPERATOR	50.7	50.2	49.8	50.7	49.3	50.0

Figure 7: Heatmaps vizualizing the effect of context (x-axis) for bugs of different bug types (SStuB patterns). Top TSSM-3B, bottom ManySStuBs4J. Bug types in red involve identifiers, those in orange likely involve identifiers. Coloring expresses the performance ratio relative to the column with minimum performance (per bug type); a ratio of 1.0 corresponds to white.

Non-responding patterns. Consistently for both datasets, operator changes range among the weakest responders, be it binary or unary operators. We also see no or only a very weak response for boolean and numeric literal changes; in contrast to string literal changes, where there is moderate response.

	2	10	20	32	40	56
1	30.8	36.1	38.3	40.2	40.8	41.4
2	25.8	30.4	32.0	32.8	33.1	33.8
3	13.9	15.9	16.4	18.0	17.7	18.2
4	11.6	12.9	13.8	13.9	14.6	14.8
5	8.7	8.4	9.6	9.5	9.4	9.6
6	10.1	10.6	10.7	11.6	11.5	11.6
7	8.0	8.5	8.5	8.5	8.9	8.0
8	7.5	8.3	9.0	8.1	8.6	9.0
9	7.1	6.5	7.1	7.1	6.5	7.1
10	9.4	10.4	10.0	10.4	10.7	11.0
11	5.1	5.6	6.0	6.0	6.0	5.6
12	10.7	10.7	11.7	13.1	12.6	11.7

Figure 8: Heatmap visualizing the effect of context for bugs that require multiple changes to fix. As change count increases, the effect of more context diminishes.

4.2.2 Change Count. Results show that as the number of changes increases, the effect of context steadily decreases. For six changes we still see a very weak improvement of 1.5% from 2 to 56 lines of

context; this reduces further for larger context sizes (Figure 8). We cannot conclusively answer why this is the case. We hypothesize that these bugs are much more difficult to fix, irrespective of context.

> **Answer to RQ2**
>
> The effect on repair success strongly depends on the bug type and the number of changes required for a fix. For high change counts (> 6) and certain bug types (e.g., boolean literals) increasing the context size does not substantially aid repair success.
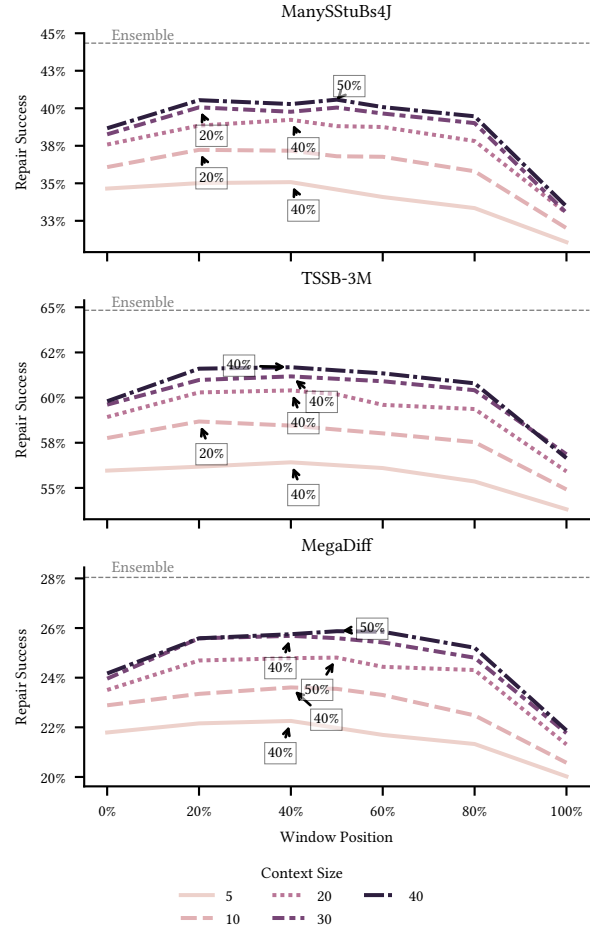
4.3 Context Window Position (RQ3)

A natural choice for filling the context window is to use 50% pre-context and 50% post-context. We confirm this choice, although in some cases, a window slightly offset from the center might perform a bit better. Figure 9 shows performance across all three datasets, five context window sizes and all context window positions.

For MegaDiff, peak performance is reached at the 40% position (i.e., 60% of post context) for window sizes 5, 10 and 30 and 50% for 20 and 40. For TSSB-3M, the 40% window was best for all context sizes, except 20. For this size, the best position lies in the other direction, at 20%. 40%-50% was the best performing position for the majority of context sizes (three out of five) in the ManySStuBs4J dataset; the other two where 20%. For all datasets and context sizes, we see that the 0% position (i.e., only pre-context) consistently outperforms the 100% position (i.e., only post-context), which performs worst in all configurations.

Complementarity. The question naturally arises, whether, at different window positions, the model is able to fix different bugs, that is, whether different window positions are complementary to each other. To answer this question, for four of the window sizes (5, 10, 20, 40) and three window positions, we analyze how many bugs are unique (fixed only at specific window position) and how many of them are common to multiple positions. For each window size we select the extreme window positions (0% and 100% position) and a position close to the middle (upper median position). The results of this analysis are visualized as Venn diagrams in Figure 10. Only 53%-57% of bugs are correctly fixed at all three window positions; 6%-7% of them are fixed by only one of the three positions.

Ensembling. Given different window positions are highly complementary, the next question that poses itself is whether models trained on window positions can be, for better performance, combined into an ensemble. When taking the five highest ranking unique (that is filtering out duplicates) predictions (XXX it *seems* ranking is by beam search score, but I'm not 100% sure, generation order is not documented. One reviewer asked for details, not sure...) from the six models trained on different window positions (but the same window size) we observe a significant boost in performance. We indicate ensemble performance of the largest context window (40 lines of code) with a horizontal line in Figure 9. Overall, we see absolute improvements between 2% and 3.8% across all context sizes.

Figure 9: Repair success for different datasets, context window positions and sizes. Position of peak performance annotated. See Figure 3 for a visualization of window positions.

> **Answer to RQ3**
>
> Our results suggest that peak repair success is reached at a window position of 20%-50%. While the best value varies across datasets and window sizes, using 60% for pre-context and 40% for post-context seems to be a good middle-ground. Further, different window positions allow the model to fix different bugs. This complementarity can be exploited by combining models trained on different window positions into an ensemble.

4.4 Sample and Model Size (RQ4)

With increasing context, the number of possible ingredients grows (Figure 13). We theorize that a larger model *may* be able to better exploit this wealth of ingredients. Similarly, raising the number of candidate fixes may allow the model to use more of the ingredients. If so, we would expect a *disproportional* increase in performance (i.e., a widening gap) for larger context sizes. However, our results do not support this hypothesis for the tested models.
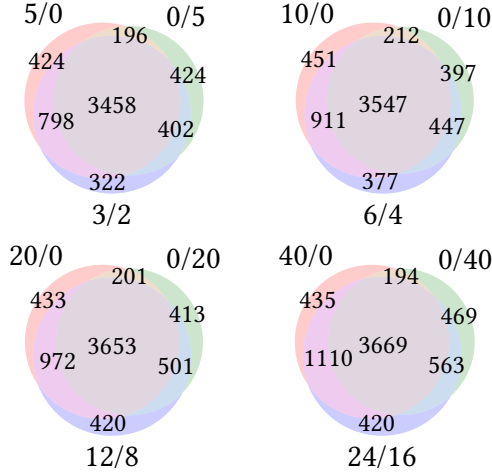
Figure 10: Are fixed bugs unique to a specific pre and post-context configuration? For four different context sizes (5, 10, 20, 40) the Venn diagrams show how many fixed bugs are unique to a specific positional configuration (non-overlapping regions) and how many bugs could be fixed at multiple positions (overlapping regions). At the center, the number of bugs correctly fixed at all positions. Circles represent positions 100%/0% (○), 0%/100% (○) and 60%/40% (○); bugs taken from MegaDiff.

Figure 11: Difference in repair success (absolute) between the smaller 60M and the larger 220M parameter CodeT5 model for different context sizes (x) and labels (y) in ManySStuBs4J. The larger model fares better for all labels except SWAP_-BOOLEAN_LITERAL.

Fix candidate count. When varying the number of candidates, we see that the margin between the two models remains largely constant (Figure 12, MegaDiff). The performance gap between 5 and 15 candidates has a mean of 5.97% and a standard deviation of 0.16%. Between 10 and 15 candidates, this difference decreases to 2% (SD 0.09%).

Model Size. Similarly, for model size we see a mostly constant gap across varying context size. For MegaDiff this gap has a mean and standard deviation of 4.7% and 0.26%, for ManySStuBs4J 5.5% and 0.15%. Figure 11 shows a per-type comparison of performance for different bug patterns in ManySStuBs4J. We cannot discern
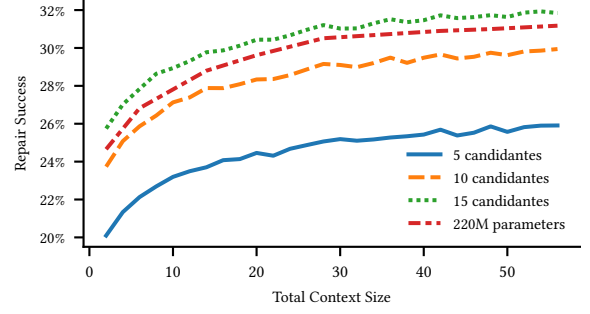
Figure 12: Repair success for different number of fix candidates (with the number of beams equal to the number of fix candidates) and the larger 220M parameter model (with 5 candidates).

a clear upwards trend for any pattern. Of note, for a single bug pattern (SWAP_BOOLEAN_LITERAL) the smaller model fared better.

> **Answer to RQ4**
>
> As expected, both, a larger model as well as an increased number of fix candidates improves overall performance. However, we do not find convincing evidence that the gap of improvement widens at large context sizes, which would indicate a better exploitation of larger contexts.

5 IMPLICATIONS

5.1 Opportunities of Context

Context has a clear impact on performance. Through systematically varying the context size fed to NPR models from 2 to 56 lines, we observe *relative* improvements in repair success ranging from 16% (TSSB-3M) to 29% (MegaDiff, ManySStuBs4J). Extending the context size from 10 to 56 lines of code still yields *relative* increases between 7% and 11.7%. To put these changes in perspective, this performance variation comes close to the improvements offered by some components of APR approaches, as identified through ablations. For instance, RewardRepair's semantic training improves performance by 7% Ye et al. [47], CoCoNuT's context-awareness feature by 16%, and the use of diagnostics in SelfAPR 37% (all relative). The improvement is also comparable to improvements one can leverage by increasing the size of the model (~28%, relative), or the number of generated samples (23%, relative), measured on MegaDiff with the largest context size of 56 and when moving from 5 to 15 fix candidates. Importantly, the improvement obtained by increasing context appears to be orthogonal to the improvements obtained by increasing model size or number of samples.

Context as a source of ingredients. As mentioned in Section 2, context is an important source for fix ingredients [45]. Furthermore, in RQ2, we identified that the changes for which the model improves most are the ones for which it needs to leverage identifiers in its context. To estimate the extent to which context ingredients could be responsible for the increased repair success we compute the overlap of identifiers in the context and the ground-truth fix. We define overlap as $(IDs(fixed) \cap IDs(context))/|IDs(fixed)|$, where $IDs(c)$ denotes the set of all identifiers in c. Figure 13 shows

identifier overlap as a function of context size for MegaDiff. This supports the hypothesis observed from the change types in RQ2: an increased pool of ingredients, particularly identifiers, is a major factor for improved performance with larger context. Further, since both pre and post context are sources of distinct ingredients, this supports the observation in RQ3 that different context window fix different bugs. Similarly, Xia et al. [41] find that feeding post-context in addition to pre-context increases repair performance and lowers syntactic and semantic errors in the generated code.
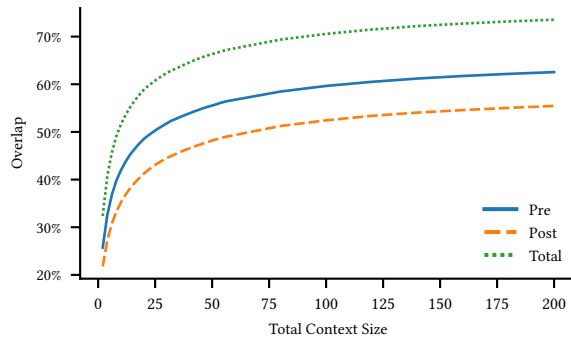
Figure 13: Overlap between identifiers in the context and the ground-truth fix for different context sizes (MegaDiff).

Context can scale further. Figure 13 indicates that context can scale further: the proportion of ground-truth identifiers that enter the context continues to grow well beyond the context sizes that we studied (28 lines pre/post). At 28 lines of pre-post context, 67% of ground truth identifiers are in context. While there are diminishing returns, this grows to 74% if the context is extended to 200 lines of context. Even simple approaches can show appreciable benefits to leverage this additional context. By combining multiple 40-line context windows in an ensemble of models, we were able to increase repair success rate by 2 to 3.8% (~5-8% relative improvement) compared to the best 40-line window (Figure 9). In essence, our ensemble leverages a context of up to 80 lines of context, and does so in a very straightforward fashion; other ensembling or selection strategies may further improve performance.

Local context can be combined. Finally, we note that even relatively large context sizes may not fill up the entire context window of even a modestly sized model. As mentioned in Section 3.2, 56 lines of context have a median token count of 592 and an upper quartile of 699. A T5 model fine-tuned with a 1024 token window comfortably fits this, leaving space for additional context enriching techniques such as SequenceR [2], or other techniques described in Section 2.1; this is even more the case for larger models.

5.2 Challenges with Context

Context is not a panacea. While context is helpful *overall*, it is not always so. For every 10 bugs that can be reliably solved by increasing context size, there is one bug for which more context causes a regression (see Figure 6 for an example). Moreover, an even larger proportion of bugs are sensitive to the context in ways that are hard to predict: the model may succeed to solve them at some context

sizes, and not others. This is an example of the lack of robustness of neural models of code. Other work has found comparable rates of differences for both NPR [5], and code generation [20]. This clearly calls for increased research on more robust NPR models.

We also note a silver lining: while the best performance we obtain on MegaDiff was of 26.3% with a single model, up to 35.2% of the bugs in MegaDiff can be fixed at one context size or another (RQ3). Combined with our finding that a straightforward ensembling technique for window sizes yielded benefits, one avenue for future research is to investigate whether ensembling models of different context sizes may help performance, and possibly alleviate the robustness issues we observed.

The limits of local context. While we see further potential in extending the window size as mentioned above, there are two important caveats. First, according to Figure 13, even when extending the context to large sizes (200 lines or more), roughly one quarter or identifiers in the ground-truth fix are not found in the context. This finding echoes other studies that found that a large proportion of method calls are non-local [11]. Approaches that leverage other kinds of context (Section 2.1) can complement the local context. Second, while there is room to improve with the local context, whether the current crop of models can exploit it is another question: as mentioned earlier, we observed diminishing returns with more than 50 lines of context. One way forward might be models that can better generalize to longer contexts, using e.g., RoPE [32] or ALiBi [27]; we are unaware of such models being used in APR, beyond CodeT5's relative position embeddings.

Some changes are challenging, regardless of context. Finally, while we observe some improvements when adding context, some categories of changes benefit far more than others (RQ2). In particular, the models struggled with larger changes, regardless of context size. Context size does offer some improvements for larger changes (e.g 5 or more lines), but they are slight. Similarly, while some categories of changes benefit from an increased context, for other categories context made little difference or was detrimental.

5.3 Implications on Research Practices

Context should be systematically documented. Given the impact that context has on the performance of NPR models, clearly and thoroughly documenting the context size that was used in any experiment is crucial. However approaches from the literature are not always clear. For instance, in RewardRepair, "the context code is considered as 10 lines of code surrounding the buggy code" [47]. This is ambiguous: it can be interpreted either as 5 lines on each side (totalling 10), or 10 lines on each side (totalling 20). Other work simply adds context until the input window is filled up (e.g., AlphaRepair [42]), or uses the surrounding method as a context boundary (e.g., CoCoNuT [19]). In both cases the context size is highly variable as it depends on the size of the bug and the size of the surrounding method. Finally, since the context window position matters, clearly specifying how the context is distributed in pre and post context is important as well. We thus call on the community to document their choices in terms of context as clearly as possible.

Datasets should include enough context. Last but not least, it is data that makes machine learning approaches possible. As mentioned in Section 2.3, several NPR datasets did not forecast the need for a larger context. This includes method-level datasets such as BFP [34] and CoCoNuT [19], and change-level datasets with a truncated context such as ManySStuBs4J [10] and TSSM-3B[29]. Only MegaDiff had enough information for our study [21]. We hypothesize that the dearth of adequate datasets has limited studies of models with larger contexts. Indeed, we consider that re-mining the ManySStuBs4J and TSSM-3B datasets to include this additional context is one of our most important contributions. We call on the community to provide this information in future datasets, or, at a minimum, to include the information necessary for such a re-mining to take place (e.g., repository identifiers, commit hashes).

6 THREATS TO VALIDITY

Bugs. Despite due diligence we cannot fully preclude software bugs in our training and evaluation scripts.

Single architecture. The scope of this work is limited to a single Transformer model (CodeT5). We chose CodeT5 as its relative position embeddings allow us to fine-tune to a larger input window size, which was a strict requirement for this study. While this model has seen widespread use in NPR, other Transformer types or neural architectures have been used (e.g., CURE [8] uses a GPT-style Transformer, CoCoNuT [19] uses convolutional networks). To which degree our results apply to them is subject to future work. With over 100 different models trained in this work, adding other architectures would have not only gone far beyond the scope of this paper but also exceeded our hardware capabilities. Instead, we focused on depth, by studying different aspects of context such as window positions, bug types and change count as well as multiple programming languages. The experiments in RQ4 show that the effects described are likely not dependent on model size or the number of candidates generated. This suggests (pending confirmation by further work) that results might extrapolate well to other Transformer types and possibly even other network types.

Evaluation metric and datasets. We use exact match as an evaluation metric. Exact matching might underestimate the performance of a model, as a semantically correct fix can be expressed in multiple syntactic forms [26]. Benchmarks such as Defects4J [9] or BugsInPy [38] come with test cases, which would alleviate this issue. However these benchmarks range are much smaller (hundreds of bugs). The large weight assigned to any single bug leads to concerns about noise in measurements, and associated issues with overfitting to smaller datasets, which is a concern in APR [18, 23]. We rely on two orders of magnitude more data to limit this.

7 CONCLUSION

In this work we studied the effect of local context in NPR from multiple perspectives. On multiple datasets, we first find that varying context size yields sizeable improvements (a 16–29% relative improvement), that continue well beyond typical context local contexts used in NPR (e.g., a relative 7–11% improvement going from 10 to 56 lines of context). Moreover, different context configurations yield different results, with straightforward ensembling techniques

giving a further 5–8% relative improvement. Our analysis of bug patterns shows that bug types involving identifiers particularly benefit from increased context. Improvements from model size and number of samples appear orthogonal to those gained from context.

Our results have multiple implications. First, there is further room for improvement as local context can be leveraged further. On the other hand, context come with challenges, particularly in terms of robustness to sometimes small variations in context. Finally, given that performance variations due to context are comparable to some improvements in the literature, we call on the community to clearly document the context they use, as well as ensuring that datasets come with enough context. We provide training and evaluation scripts as well as datasets allowing to expand this work with larger models and other neural architectures. They can be downloaded from https://github.com/giganticode/out_of_context_paper_data.

REFERENCES

[1] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.

[2] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* 47, 9 (Sept. 2021), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[3] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1469–1481. https://doi.org/10.1109/ICSE48619.2023.00128

[4] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 935–947.

[5] Hongliang Ge, Wenkang Zhong, Chuanyi Li, Jidong Ge, Hao Hu, and Bin Luo. [n. d.]. RobustNPR: Evaluating the Robustness of Neural Program Repair Models. *Journal of Software: Evolution and Process* n/a, n/a ([n. d.]), e2586. https://doi.org/10.1002/smr.2586

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[7] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. https://doi.org/10.48550/arXiv.2302.05020 arXiv:2302.05020 [cs]

[8] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[9] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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 (ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[10] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 573–577. https://doi.org/10.1145/3379597.3387491

[11] Anjan Karmakar, Miltiadis Allamanis, and Romain Robbes. 2023. JEMMA: An Extensible Java Dataset for ML4Code Applications. *Empirical Software Engineering* 28, 2 (March 2023), 54. https://doi.org/10.1007/s10664-022-10275-7

[12] Kicheol Kim, Misoo Kim, and Eunseok Lee. [n. d.]. Systematic Analysis of Defect-Specific Code Abstraction for Neural Program Repair. In *2022 29th Asia-Pacific Software Engineering Conference (APSEC)* (2022-12). 81–89. https://doi.org/10.1109/APSEC57359.2022.00020 ISSN: 2640-0715.

[13] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec. 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[14] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[15] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941

[16] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. [n. d.]. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)* (2019-04). 102–113. https://doi.org/10.1109/ICST.2019.00020 ISSN: 2159-4848.

[17] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 658–662. https://doi.org/10.1109/APSEC.2018.00085

[18] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F Bissyandé. 2021. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software* 171 (2021), 110817.

[19] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (IS-STA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[20] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2149–2160. https://doi.org/10.1109/ICSE48619.2023.00181

[21] Martin Monperrus, Matias Martinez, He Ye, Fernanda Madeiral, Thomas Durieux, and Zhongxing Yu. 2021. Megadiff: A Dataset of 600k Java Source Code Changes Categorized by Diff Size. *arXiv:2108.04631 [cs]* (Aug. 2021). arXiv:2108.04631 [cs]

[22] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2450–2462. https://doi.org/10.1109/ICSE48619.2023.00205

[23] Kunihiro Noda, Yusuke Nemoto, Keisuke Hotta, Hideo Tanida, and Shinji Kikuchi. 2020. Experience report: How effective is automated program repair for industrial software?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 612–616.

[24] Nikhil Parasaram, Earl T. Barr, and Sergey Mechtaev. 2023. Rete: Learning Namespace Representation for Program Repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1264–1276. https://doi.org/10.1109/ICSE48619.2023.00112

[25] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs? An Evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair (APR '22)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3524459.3527351

[26] Julian Aron Prenner and Romain Robbes. 2023. RunBugRun – An Executable Dataset for Automated Program Repair. https://doi.org/10.48550/arXiv.2304.01102 arXiv:2304.01102 [cs]

[27] Ofir Press, Noah A. Smith, and Mike Lewis. 2022. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. https://doi.org/10.48550/arXiv.2108.12409 arXiv:2108.12409 [cs]

[28] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. [n. d.]. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. https://doi.org/10.48550/arXiv.1910.10683 arXiv:1910.10683 [cs, stat]

[29] Cedric Richter and Heike Wehrheim. 2022. TSSB-3M: Mining Single Statement Bugs at Massive Scale. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 418–422. https://doi.org/10.1145/3524842.3528505

[30] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. 2017. Elixir: Effective Object-Oriented Program Repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659. https://doi.org/10.1109/ASE.2017.8115675

[31] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. 2018. Self-Attention with Relative Position Representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*. Association for Computational Linguistics, New Orleans, Louisiana, 464–468. https://doi.org/10.18653/v1/N18-2074

[32] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2022. RoFormer: Enhanced Transformer with Rotary Position Embedding. https://doi.org/10.48550/arXiv.2104.09864 arXiv:2104.09864 [cs]

[33] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[34] M. Tufano, C. Watson, G. Bavota, M. di Penta, M. White, and D. Poshyvanyk. [n. d.]. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2018-09). 832–837. https://doi.org/10.1145/3238147.3240732 ISSN: 2643-1572.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv:1706.03762 [cs]* (Dec. 2017). arXiv:1706.03762 [cs]

[36] Weishi Wang, Yue Wang, Shafiq Joty, and Steven C.H. Hoi. 2023. RAP-Gen: Retrieval-Augmented Patch Generation with CodeT5 for Automatic Program Repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 146–158. https://doi.org/10.1145/3611643.3616256

[37] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. [n. d.]. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. https://doi.org/10.48550/arXiv.2109.00859 arXiv:2109.00859 [cs]

[38] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1556–1560. https://doi.org/10.1145/3368089.3417943

[39] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[40] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. https://doi.org/10.48550/arXiv.2303.10494 arXiv:2303.10494 [cs]

[41] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[42] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[43] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 660–670.

[44] Qi Xin and Steven P. Reiss. 2019. Better Code Search and Reuse for Better Program Repair. In *Proceedings of the 6th International Workshop on Genetic Improvement (GI '19)*. IEEE Press, Montreal, Quebec, Canada, 10–17. https://doi.org/10.1109/GI.2019.00012

[45] Deheng Yang, Kui Liu, Dongsun Kim, Anil Koyuncu, Kisub Kim, Haoye Tian, Yan Lei, Xiaoguang Mao, Jacques Klein, and Tegawendé F. Bissyandé. 2021. Where Were the Repair Ingredients for Defects4j Bugs? *Empirical Software Engineering* 26, 6 (Sept. 2021), 122. https://doi.org/10.1007/s10664-021-10003-7

[46] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. https://doi.org/10.48550/arXiv.2203.12755 arXiv:2203.12755 [cs]

[47] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1506–1518. https://doi.org/10.1145/3510003.3510222

[48] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22).* Association for Computing Machinery, New York, NY, USA, 1506–1518.   https://doi.org/10.1145/3510003.3510222

[49] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (Oct. 2020), 1040–1067.   https://doi.org/10.1109/TSE.2018.2874648