Evaluating Distance Measures for Program Repair

Charles Koutcheme
charles.koutcheme@aalto.fi
Aalto University
Espoo, Finland

Sami Sarsa
sami.sarsa@aalto.fi
Aalto University
Espoo, Finland

Juho Leinonen
juho.leinonen@auckland.ac.nz
The University of Auckland
Auckland, New Zealand

Lassi Haaranen
lassi.haaranen@aalto.fi
Aalto University
Espoo, Finland

Arto Hellas
arto.hellas@aalto.fi
Aalto University
Espoo, Finland

ABSTRACT

Background and Context: Struggling with programming assignments while learning to program is a common phenomenon in programming courses around the world. Supporting struggling students is a common theme in Computing Education Research (CER), where a wide variety of support methods have been created and evaluated. An important stream of research here focuses on program repair, where methods for automatically fixing erroneous code are used for supporting students as they debug their code. Work in this area has so far assessed the performance of the methods by evaluating the closeness of the proposed fixes to the original erroneous code. The evaluations have mainly relied on the use of edit distance measures such as the sequence edit distance and there is a lack of research on which distance measure is the most appropriate.

Objectives: Provide insight into measures for quantifying the distance between erroneous code written by a student and a proposed change. We conduct the evaluation in an introductory programming context, where insight into the distance measures can provide help in choosing a suitable metric that can inform which fixes should be suggested to novices.

Method: A team of five experts annotated a subset of the Dublin dataset, creating solutions for over a thousand erroneous programs written by students. We evaluated how the prominent edit distance measures from the CER literature compare against measures used in Natural Language Processing (NLP) tasks for retrieving the experts' solutions from a pool of proposed solutions. We also evaluated how the expert-generated solutions compare against the solutions proposed by common program repair algorithms. The annotated dataset and the evaluation code are published as part of the work.

Findings: Our results highlight that the ROUGE score, classically used for evaluating the performance of machine summarization tasks, performs well as an evaluation and selection metric for program repair. We also highlight the practical utility of NLP metrics, which allow an easier interpretation and comparison of the performance of repair techniques when compared to the classic methods used in the CER literature.

Implications: Our study highlights the variety of distance metrics used for comparing source codes. We find issues with the classically used distance measures that can be combated by using NLP metrics. Based on our findings, we recommend including NLP metrics, and in particular, the ROUGE metric, in evaluations when considering new program repair methodologies. We also suggest incorporating NLP metrics into other areas where source codes are compared, including plagiarism detection.

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; • **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

program repair, automated program repair, automatic program repair, distance measures, distance metrics, ROUGE, BLEU, dataset, bug fixing, feedback, natural language processing, educational data mining, computing education

ACM Reference Format:
Charles Koutcheme, Sami Sarsa, Juho Leinonen, Lassi Haaranen, and Arto Hellas. 2023. Evaluating Distance Measures for Program Repair. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1 (ICER '23 V1), August 07–11, 2023, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3568813.3600130

1 INTRODUCTION

Learning to program is associated with a wide variety of challenges [47]. When learning to program, one needs to become familiar with the notation of the programming language and its syntax [16, 42], and to learn how to work with the tools of a programmer, often including a programming environment. Although plenty of time can be invested in understanding "trivial mechanics" [66], a particular aspect that students struggle with are errors that occur during the programming process. Syntax errors in particular have received plenty of attention in CER, where researchers have observed that solving common errors takes a similar amount of time for both high-performing and low-performing students [12]

ICER '23 V1, August 07–11, 2023, Chicago, IL, USA

ACM ISBN 978-1-4503-9976-0/23/08.
https://doi.org/10.1145/3568813.3600130

and that some errors can take substantially more time to fix than others [3, 12].

Ideally, we would see teachers and teaching assistants individually guide students as they are learning – and fixing mistakes in the learning process. This is not feasible in practice in large courses, and thus, a large body of CER literature has focused on automating assessment [2, 15, 31, 56] and on improving the efficiency of providing feedback [14, 19, 26, 38, 40, 43, 53]. Feedback quality, in particular, is a crucial part of the learning process [25]; good feedback can improve learning and poor feedback may hinder it. Feedback in programming can take many forms and can be applied during or at the end of the programming process [38]. For instance, feedback can take the form of hints to help students arrive at a final solution [59, 63–65, 77] and can supplement automated graders with details on the submitted solutions [30].

One stream of research in the area is focused on helping students debug their code. Debugging and fixing issues is particularly challenging and time-consuming for novice programmers [3, 12] and thus plenty of effort has been invested in improving programming error messages [6, 44]. While the early work in this area has focused on augmenting error messages with additional details, a recent research stream has also looked into developing and applying automated program repair techniques for supporting students (Fig. 1 outlines the general idea). Examples of this include helping students with syntactic issues [1, 7, 22], semantic issues [21, 29, 78], and both [82]. Given the emergence of tools and techniques for program repair in education, there is a need to critically evaluate the performance of these techniques, since most of them do not rely on human supervision. In particular, most, if not all, studies on program repair in education have used classic edit distance metrics such as tree edit distance for choosing program repair candidates, based on which the feedback is constructed. This intuitively makes sense; a minimum amount of suggested changes should not cause significant cognitive load when contrasted with a large body of suggestions that effectively would lead to a full rewrite of the program. There seems to be a lack of empirical evaluation of the distance metrics, however.

> *There is a risk that choosing the traditional distance measures for selecting a program repair candidate might favour options that are further away from a student's intent than other repairs, due to some intrinsic aspects of the distance measure and its representation.*

In this article, we empirically evaluate the use of distance measures for comparing and selecting candidate repairs. Our overall research theme is: *How do different measures perform for evaluating program repair candidates for programming feedback?* Given the usage of edit distance measures and the recent trend of increasingly using natural language processing (NLP) and machine learning techniques in CER, there is a need to answer the following research questions:

- **RQ1** Which edit distance metrics perform better for evaluating program repair candidates?
- **RQ2** How do Natural Language Processing scoring metrics compare against edit distance metrics for selecting candidate repaired programs?

To answer the research questions, we annotated a dataset of programming solutions with expert evaluations on what the right repair to faulty programming submissions should be. We report on three experiments comparing distance and scoring measures and report which of these measures performs the best for our task. Our results show that a distance metric based on the ROUGE score is the best at evaluating closeness to the goal. As an additional contribution, on top of answering the related research questions, we release the first educational dataset (to the best of our knowledge) composed of students' buggy programs and expert annotations on their closest corrections.

This article is organized as follows. We begin by reviewing studies on novice programmer mistakes, followed by reviewing work conducted in program repair in CER. In the method section, we present the dataset used in this study, how we annotated the dataset, and which experiments we conducted to answer our research questions. The results are outlined after the method, and after outlining the results, we discuss the impact of our findings and highlight methodological considerations for using various distance and scoring measures for program repair in CER.

2 BACKGROUND

2.1 Studying novice programmer mistakes

Studying novice programmer mistakes provides insight into the issues that students are facing while learning to program [75]. Although researchers have pointed out that the way some students solve programming problems can seem random to the extent that it is almost as if they are "programming by incident" [27], some mistakes are more frequent than others [75] and some mistakes also take more time to fix than others [9, 12, 52, 71]. Having data on student mistakes can be useful, as teachers might otherwise rely on intuition on what aspects to emphasize, which might not match reality. Indeed, as pointed out by Brown and Altadmri [9], educators' beliefs on the frequency of mistakes risk not matching the data (or the beliefs of other educators). Classically, the analysis of novice programmer mistakes focused on specific problems, perhaps in part due to the focus on those by specific research groups. An example of this is the Rainfall problem, which was initially introduced as a looping problem in the early 1980s [35, 73, 74] and has since received plenty of attention in the research literature [68]. One particular stream of research has focused on studying source code snapshots and submissions [32] collected from learning environments, such as BlueJ [41]. This research stream has included quantifying and studying syntax errors that students face while programming [12, 17, 33, 51, 76] and led to the emergence of a family of methods used for detecting at-risk students based on errors observed in the programming process [5, 34, 80]. When considering the errors that students face, only some of them are syntax errors [3, 18] that can be directly identified using a compiler. The errors can also reside in the application logic where the errors have been broadly categorized into algorithmic errors (flawed approach), misinterpretation (misinterpreting the question), and misconception (flaws in programming knowledge) [18]. Access to errors also depends on the used programming language and the compiler [39], which in turn can also influence the type of feedback that can be given to the students.

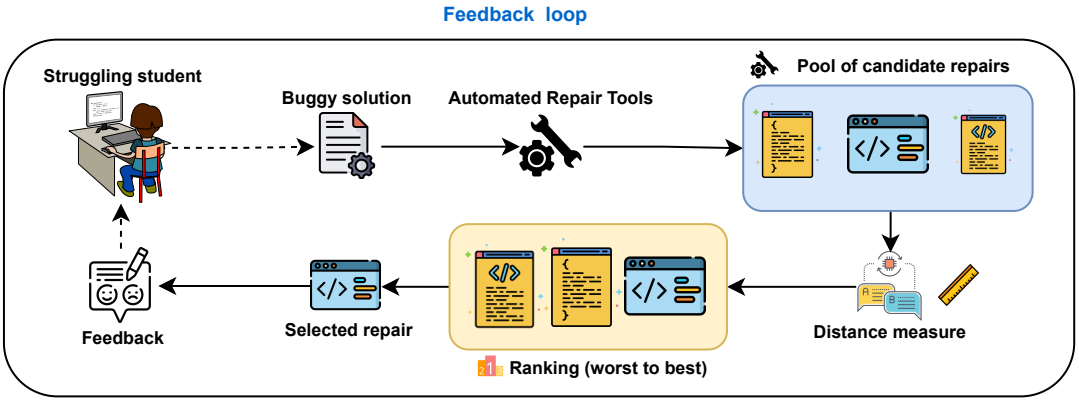

Figure 1: General idea of feedback based on automated program repair. A struggling student submits their buggy solution to a feedback system that uses one or more automated repair tools. Under the hood, the feedback system uses automated repair technique(s) to obtain potential fixes to the buggy solution. Out of the pool of candidate fixes, the system selects one fix based on comparing how far each candidate repair is from the original buggy solution, using a distance measure such as string edit distance.

Researchers have looked into enhancing feedback related to the compiler error messages [6, 17, 44] in part as the error messages do not always match the underlying cause [51]. This is still an ongoing research area [6], where one promising direction is focusing on improving the readability of the error messages [6, 13]. Improved error messages could help in the debugging process and even improve code comprehension, which has been highlighted as one of the issues with novice programmers [81].

Two key aspects of good error messages for novices are succinctness and clarity [13]. When considering that only some of the errors that students face are syntax errors [3, 18], additional ways to reach clear and succinct messages are needed. Here, one possibility is utilizing automated program repair methods for fixing the students' code [21, 29, 58, 78], which we discuss next.

2.2 Automated student program repair

Techniques from the field of automated program repair are also useful in educational contexts. While some work focuses exclusively on correcting syntactic or compilation mistakes in students' programs [1, 7, 22, 24, 67], we focus our attention on those primarily aiming at correcting semantic mistakes [21, 29, 70, 78, 82]. Our work complements prior work in hint-generation for programming that looks into providing the next step that a student could use to proceed (see e.g. [49, 50, 60, 65]) by exploring the changes needed to reach a final working solution. That is, we focus on producing a useful correct solution from an incorrect one, where the correction has *a minimal amount of modifications to the original code*. The constraint on the minimum number of changes has been emphasized algorithmically in various ways.

While the existing approaches differ in multiple aspects in the strategy used to solve the problem, all of them rely in one way or another on the availability of candidate solutions. We argue that educators are the most reliable source of corrections to students' buggy solutions. However, we also acknowledge that due to time and resource constraints, there is little possibility to primarily rely on teachers for supporting students. One approach would be to ask for a small number of annotations from the teachers and propagate those to clusters of incorrect solutions [36, 40]. The idea of clustering buggy code and propagating teachers' annotations has in general been popular for providing students with all forms of feedback [11, 19, 26, 30, 37, 40, 53, 79]. In a program repair context, teachers' work can be further minimized if we rely solely on one of the reference solutions [70].

> *Instead of depending exclusively on teachers' annotations, most automated repair systems also leverage correct solutions submitted by other students, and previous data kept by educational systems.*

Classically, program repair tools rely on rule-based systems used for constructing working solutions from existing data. This approach consists broadly speaking of two main steps: (1) searching for a small set of candidate correct solutions that match the buggy program, and (2) modifying the buggy solution to arrive at a version that matches the candidate code structure. While the algorithms vary in how they implement each step, they all heavily emphasize the constraint that the found repaired program (modified buggy code) preserves as much as possible of its original syntax. To enforce this constraint, generally, the algorithmic pipeline involves the comparison of control flow graphs [21, 29, 78] to match each buggy program with the selected set of correct candidates. For instance, Clara [21], SarfGen [78], and Refactory [29] match each buggy program with the selected correct candidates based on the code's control flow graph. While SarfGen uses a custom embedding distance, Clara and Refactory use tree edit distance. In order to reduce the number of comparisons to evaluate, some work proposes reducing the search for candidates by clustering matching correct solutions together, and only comparing the buggy program against cluster centers [21]. This strategy augments the cluster and feedback approach discussed in CER literature (c.f. [19, 40]),

without relying on teacher annotations. Instead of minimizing the number of connections, other works find better utility in comparing the buggy program against all available correct solutions and augmented versions. For instance, Refactory [29] augments the original set of correct programs to search for by refactoring programs into multiple semantically similar versions.

While most of these traditional methods rely mainly on using rule-based systems to construct working solutions from existing data, there is a transition towards the adoption of machine learning approaches. In that area, previous work has employed available data with sequence-to-sequence machine learning models for repairing programs. For instance, Pu et al. [61] trained a Recurrent Neural Network -based sequence to sequence model on students' correct solutions. The authors use their model with an enumerative repair strategy to repair buggy programs. In the last years, work has also started to leverage large pre-trained language models. For instance, Zhang et al. [82] introduced MMAPR, an automated repair technique for introductory Python programming assignments. Their approach uses correct solutions, test cases, and assignment descriptions to prompt OpenAI Codex. They evaluate their method on 286 Python programs produced by novices and show that their approach can repair up to 96.5% of the programs, with a smaller sequence edit distance compared to Refactory [29]. Similarly, Phung et al. [58] developed PyFiXV which is powered by Codex to automatically repair student programs.

Across time, automated repair techniques consistently reported their quantitative performance results in terms of the number of programs that could be effectively repaired [61] (i.e., the percentage of fixes found) as well as the average or median time to find a fix [21, 78]. Still, in the educational context, beyond the ability of a technique to find *a* repair, there remains the question and the need to ensure that the found fixes remain understandable for the student who wrote the code. This aspect is necessary to compare how well different techniques compare. For instance, one could consider the reference solution as always being a valid fix to a buggy program, but not always an understandable fix due to e.g. using a different approach [54]. Although the constraint on "understandable by student" has often been explicitly enforced in the algorithmic part by minimizing a notion of distance (classically, tree edit distance), it has not always been thoroughly examined in the evaluation of the techniques. When this necessary aspect has been reported, using a measure of distance to the original buggy program, classically, the sequence edit distance [29, 82] has been used. While there has been much work in developing automated repair techniques and adapting distances for that purpose, we found no empirical evidence for which distance measure should be adapted for evaluating and comparing these techniques in terms of the quality of the found repairs.

3 METHOD

In this section, we review our methodological procedure. We start by describing the source of our data and our annotation process. Then, we describe the experiments conducted and present the distance measures evaluated to answer our two research questions. Our annotated dataset and the code for conducting the experiments are publicly available on GitHub[1].

3.1 Data and annotation

For our experiments, we annotated a subset of a publicly available dataset comprising students' solutions to programming assignments. We refer to this dataset as "The Dublin data" since it contains data collected at Dublin City University. The original dataset released by Azcona et al. [4] contains more than half a million programming submissions (591,707) by 666 students from five Python programming courses over three academic years. The assignments vary in difficulty levels ranging from basic printing to more complex sorting algorithms. We note that the released dataset contained neither original assignment descriptions nor the original test cases. In this work, we use the version processed by Cleuziou et al. [11], who enriched the dataset semi-automatically by creating test cases for 42,487 programs.

Dataset preprocessing. We selected a subset of the assignments to annotate from this base dataset. We were interested in exercises where students had to write a single function taking one or multiple arguments as input and returning a single output. We selected assignments with an average level of complexity (i.e. involving looping or recursion and one or multiple conditional statements, excluding sorting algorithms). In total, we annotated submissions for 10 distinct assignments. Due to missing problem descriptions, we inferred the objectives of the problems based on the function name and a manual analysis of the correct and buggy submissions.

Annotation strategy. To avoid annotating similar solutions multiple times, we normalized variable names and grouped programs based on an exact matching in their Abstract Syntax Tree (AST) structure. For each group, we selected the solution with the most common original string representation as a representative, and we annotated this representative.

We adopted the following strategy for annotation. In order to keep consistency within assignments, each expert was assigned one or multiple exercises to annotate. For their exercises, each expert was tasked to *write the repair to the buggy solution that passes the unit tests while requiring minimal amounts of changes*. In other words, our goal was to keep as much of the student strategy as possible. We noticed early in the annotation process that we could not annotate many of these solutions with a repair. We highlight the following primary type of issues. In multiple cases, the assignments were in too early a stage such that completion of the buggy code would not be apparent. We marked these particular buggy programs as "partial" [63]. We also noticed that some students clearly misunderstood the scope of the assignment, either because the function definition was wrong or because the student evidently tried to solve another problem. Lastly, in some situations, we could not annotate a solution because the student's strategy was either incomprehensible or unnecessarily – or even extremely – complex. In these cases, instructors would generally recommend rewriting the program solution from scratch. We marked these particular cases and omitted them in our subsequent experiments. To ensure

[1] https://github.com/KoutchemeCharles/edmpr

that our repairs were correct, we re-executed each expert solution against the example unit tests for the particular program and re-repaired the ones that failed any of the unit tests.

We then tentatively matched the educators' annotations to all other programs having the same structure. We report the results of our experiments on the final annotated dataset.

3.2 Evaluating distance measures

Before presenting the comparison of the different measures, we formulate the problem we are trying to solve and present the experiments conducted.

3.2.1 Problem definition. Given an incorrect solution to a programming assignment, we want to find a working solution which best captures the student's intention from a pool of candidate solutions (e.g. one candidate per automated repairing technique). Since in practice such ideal repair is unknown, we select the candidate repair which has the minimum distance to the buggy program (according to a given distance measure). We seek to understand which distance measure works best for this task, given that we have access to one ideal repair. Thus, we want to choose the distance measure that, among a large pool of candidate solutions to select from, ranks one of the most suitable repairs for the buggy program (here, our experts' annotation) as having the smallest distance to the original buggy code among the candidates. This observation hints to us that we can consider the evaluation of distance measures as an information retrieval problem. We pick up on the hint and compare different distance metrics using the following experiments. Figure 2 illustrates our approach.

3.2.2 Experiment 1. We compare different distance metrics using the following strategy: for each buggy solution for each assignment in our dataset, we create a pool of potential repair candidates using automatic repair tools, and we include our expert annotation in that pool. Following previous work [23], we include all correct programs submitted by all students for a particular assignment across all academic years into the pool. We highlight that this pool might include the buggy solution's author's own written working solution to the exercise (which may or may not be similar to the buggy solution). Then, using the selected distance metric, we compute the distance between the buggy solution and each candidate repair before ranking each candidate solution from worst to best according to how small the distance value is. Finally, we use the position of the expert annotation in the ranking as an error measure, the Ranking Error (RE). For example, the expert solution being ranked first/having the smallest distance has an RE of 0. To account for the different number of candidate repairs per assignment, we normalize the error by the total number of candidates. We refer to this performance measure as the Normalized Ranking Error (NRE) for the single buggy solution. Finally, we report the Average Normalized Ranking Error (ANRE) for each assignment.

3.2.3 Experiment 2. To validate our results in a more complex setting, we examine how each distance measure ranks an expert annotation against a single other high-quality candidate repair found by a state-of-the-art automated repair technique.

We use the state-of-the-art semantic Automated Repair Tool (ART) Refactory to find a candidate repair for each incorrect solution in our annotated dataset. To obtain a high-quality repair, we run the ART giving it access to the same pool of candidate repairs as used in the first experiment (without the expert solution). Using this pool of correct programs, Refactory generates a bigger suite of semantically equivalent code by refactoring all these available working solutions to a problem. Then, given an incorrect program, Refactory analyzes its control flow structure to find a closely matching working program to compare for isolating the buggy components of the buggy solution. As such, the candidate repair generated by Refactory should be better or at least as appropriate as the best candidates in the original pool (which, once again, might contain the student's own correction to the problem).

We repeat the previous experiment using the candidate repair found for each buggy solution. The main difference with the first experiment is that we compare the expert annotation/repair against the single candidate obtained using Refactory. Therefore, the ranking error for each buggy program becomes a binary classification error. We report the total classification error – the number of times the ART candidate repair was favored over the expert annotation – for all metrics.

3.2.4 Edit distances. To answer our first research question, for each experiment, we use string edit distance [63], sequence edit distance [82], and tree edit distance [29] between the buggy code and the candidate repairs. In particular, for the sequence edit distance (and the tree edit distance respectively), we use the Python tokenizer to split code into tokens (respectively, the Python built-in parser to transform code into its AST). We compute each distance measure using the python_edit_dist package from Paaßen et al. [55].

3.2.5 Normalized edit distances. Previous work has also looked at normalizing edit distances [21, 29] for evaluation. *Clara* [21] introduces the Relative Patch Size (RPS) for evaluating the performance of program repair techniques. The RPS is simply the tree edit distance (TED) between the buggy program Abstract Syntax Tree ($ASTb$) and the proposed corrected version ($ASTc$) divided by the size of the buggy program AST:

$$RPS = \frac{TED(ASTb, ASTc)}{Size(ASTb)}$$

where $Size(AST)$ is the number of nodes in an abstract syntax tree. One will notice, however, that for our ranking experiments, dividing the distance between a proposed repair by the size of the buggy program AST will *not* influence our rankings since the normalization factor is constant across compared fixes. In other words, using the RPS or TED will yield the same results. Instead of relying on relative patch sizes, we propose to compare the edit distances against their bounded normalized versions. If $DIST(b, c)$ is the value of an edit distance between buggy code b and the repair c, we can find the bounded normalized version of that distance by dividing the distance value by the maximum size of the two inputs. $DIST\_NORM = DIST(b, c)/max(size(b), size(c))$. For example, we can obtain the normalized version of the tree edit distance using:

$$TED\_NORM = \frac{TED(ASTb, ASTc)}{max(Size(ASTb), Size(ASTc))}$$

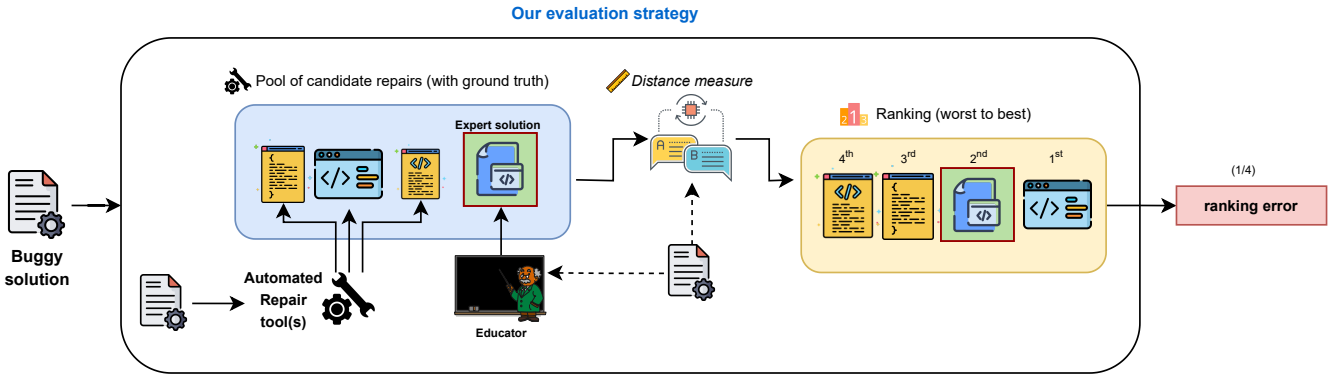

Figure 2: Illustrating our approach for evaluating distance measures. We want to quantify how good a distance measure is at finding suitable repairs. For that, we add to a large pool of potential fixes to a student's buggy solution our expert annotation on the best solution to the buggy code. We then rank all candidates from worst to best according to that given distance measure, and we use the relative position of the expert's solution as an error measure. In this example, the relative error is 0.25 (expert ranked at 2nd position out of 4 candidates).

For completeness, we also include the normalized version of the string and sequence edit distances (where we divide the edit distance by the maximum between the length of the string, and respectively the number of tokens). Although normalized edit distances have been used in hint generation systems [63], they have not been evaluated in program repair.

3.2.6 Natural Language Processing measures. To answer our second research question, we also include different variations of the two most popular performance measures used in natural language processing (NLP). We use the original BLEU [57] metric, it's adapted version for code [62], and two measures from the ROUGE [45] family. For brevity, we refer the reader to the original papers to obtain the description of the equations for computing each measure. Here, we give intuition behind the measures.

The BLEU (Bilingual Evaluation Understanding) score was originally introduced for evaluation in machine translation tasks (i.e. translating text from a source language to a target language). Intuitively, BLEU measures the proportion of words (and/or n-grams) in the *machine generated translation* (i.e., the candidate) that appear in the *human translation* (i.e., the reference). It is a precision-oriented metric. We include the more recent CodeBLEU scoring metric, which extends the original BLEU score by encapsulating code syntax similarity via abstract syntax trees (AST) and code semantics matching via data flow analysis. We use the classical implementation of BLEU as presented in the Natural Language ToolKit (NLTK) package [8] and an open reimplementation of the CodeBLEU metric [46].

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) scores are a family of metrics which were originally designed in NLP for summarization tasks. Intuitively, the ROUGE score measures the proportion of the *original text* (i.e., the reference) found in the *machine-generated text* (i.e., the candidate). We use the classical ROUGE-1 metric and the ROUGE-LCS metric for our experiments. The ROUGE-LCS metric is a variation which relies on the longest common subsequence between the sentences in the source and the target texts. We used the Google implementations [20].

There are two important aspects to take into account regarding these NLP measures. First, these evaluation scores are not distances. Fortunately, their values are bounded between 0 and 1. Thus, for our experiments, we transform these scores into a distance measure by simply computing $1 - S(b, c)$, where S is the scoring metric (e.g. BLEU), b is the buggy code and c is a candidate repair. Second, the order of the operand does matter ($S(b, c) \neq S(c, b)$). For consistency, we arbitrarily choose to always consider the buggy program b as the reference "sentence" and the proposed correction as the "candidate". Since all NLP scoring measures are based on n-grams or words, we once again use the Python tokenizer to split a code into tokens to represent "words" in the "sentence".

4 RESULTS

In this section, we first describe our annotated dataset. Then, we outline the outcomes of our two experiments. In addition to the numerical results, we visualize the distributions of the distance measures.

4.1 Dataset statistics

For the 10 selected assignments, the original dataset before removing duplicates and annotation contained 6670 submissions, 3719 of which were correct, and 2951 which did not pass all the tests. After filtering, normalization, annotation of the buggy solutions, and backpropagation of the annotations, we obtain a dataset composed of 1854 buggy programs and their instructor corrections[2]. Table 1 shows the statistics of this dataset. Some assignments are functionally equivalent but require different implementation styles. For instance, we annotated two versions (swapping, iteratively) of an exercise where students had to reverse the elements in an array.

[2]The results presented in the article include also data from assignments that were only partially annotated, i.e. the experts did not annotate every buggy program for every assignment.

Table 1: Dataset statistics. Legend: #CC: number of submitted correct solutions, #BC: number of submitted buggy solutions, #AN: number of annotated buggy solutions, #lines: average number of lines in an annotated buggy solution, #STU: number of students who submitted the *annotated* buggy solutions.

	description	#CC	#BC	#AN	#lines	#STU
count_letters	Return the number of letters in a string.	458	116	116	5.77	36
index_iter	Return the position of the letter in str, -1 if it is not there.	197	491	21	8.90	5
maximum	Return the maximum element in a non-empty list of numbers.	425	64	64	7.29	37
minimum	Return the minimum element in a non-empty list of numbers.	445	236	94	6.32	38
reverse_by_swap	Reverse a list of elements by swapping them.	176	571	269	11.20	90
reverse_iter	Reverse a list of elements iteratively.	50	75	44	6.44	14
search_iter	Return whether a letter is part of a string iteratively.	443	401	372	5.35	54
search_recur	Return whether a letter is part of a string recursively.	242	308	260	6.37	29
sumup	Return the sum of all integers up to n (the input).	266	103	102	4.51	47
swap_keys_values	Swap the keys and values of a dictionary.	542	280	162	5.32	51
total/average		3244	2649	1854	12.19	305

4.2 Evaluating distance measures

Abbreviations. We will use the following abbreviations throughout this section to discuss and show our results: (N)STR: (normalized) string edit distance, (N)SEQ: (normalized) sequence edit distance, (N)TED: (normalized) tree edit distance, ROUGE(LCS): ROUGE (Longest Common Subsequence) based distance.

4.2.1 Experiment 1. Table 2 shows the average normalized ranking error (ANRE) for each distance measure (lower is better) for each assignment. We normalized the individual ranking errors by #CC + 1, where #CC is the number of correct submitted solutions for each assignment – shown in Table 1 (#CC).

From the edit distances, we observe that the tree edit distance has consistently higher errors than all other metrics. The sequence and the string edit distances perform very similarly; the best metric varies from assignment to assignment. Looking at the normalized version of the edit distances, we can make the same observations as their non-normalized versions. When comparing both families, we observe that normalization does not bring significant benefits in retrieving the expert annotation, and can even be detrimental. Among the Natural Language Processing (NLP) based distance measures, we notice that except for the "sumup" assignment, the ROUGE-based distance measures (ROUGE and ROUGELCS) are consistently better at ranking solutions than the BLEU-based distances. Within the BLEU family, CodeBLEU performs worse than BLEU. Within the ROUGE family, both metrics perform equally well. Between edit and NLP-based distances, the reader can notice that the BLEU-based distance measures do not provide major improvements compared to the string and sequence edit distances (only BLEU achieves slightly higher results), but the ROUGE-based measures perform consistently better than all others across all assignments.

4.2.2 Experiment 2. Let us examine how different distance measures compare our educators' annotations against the repairs found by an automated repair tool (Refactory). As a reminder, we can consider this experiment as determining whether the distance measure correctly "classified" the expert annotation as being better than the candidate repair. For this reason, we show the total classification error across all buggy solutions that Refactory managed to repair, for each assignment for which there is at least one repair. Table 3 shows our results.

In these results, the tree edit distance this time shows better results in total than the other edit distance measures. However, we notice that this result fluctuates heavily between assignments. Once more, the normalized versions of the edit distances do not provide meaningful performance improvements. Regarding the NLP metrics, we notice an inverse trend within the BLEU family: the BLEU distance measure this time achieves overall comparable results to the edit distances, while the CodeBLEU distance measure performs significantly better. On the other hand, the ROUGE score-based distance measures have the lowest total classification errors and remain once again the most consistent across assignments.

We acknowledge the inherent bias in this experiment towards the Automated Repair Tool. Indeed, we gave the tool access to solutions, which, in a real-life scenario, it would not have access to. Moreover, we computed the classification errors only for the buggy programs the tool effectively managed to repair, omitting the proportion of codes it failed to repair. However, the goal of this experiment is not to reflect on the tool's performance in correcting the program. Instead, our goal is to measure the ability of distance metrics to distinguish our expert annotation among very high-quality repairs. As such, we aimed at evaluating the metrics in a worst-case scenario.

4.2.3 Distances. Figure 3 contrasts the distribution of the distance between the buggy code and the expert candidate repair against the distribution of the distances between the buggy solution and the Refactory candidate repairs for the best distance measure of each family accross our two experiments (sequence edit distance, normalized string edit distance, and NLP based distance). From the histograms, we see candidate solutions getting more zero or near zero distances than expert solutions for the SEQ and ROUGELCS metrics. On the other hand, the overall picture, emphasised by the kernel density estimate (KDE), shows the expected scenario where expert repairs receive lower distance scores and candidate repairs receive larger distance scores. We note that looking at the histograms does not easily emphasize the instances where there is

Table 2: Average Normalized Ranking Error (the lower is better) per assignment for each distance measure when searching for the expert's annotation. We highlight in bold the best average result for each family.

	TED	SEQ	STR	NTED	NSEQ	NSTR	BLEU	CodeBLEU	ROUGE	ROUGELCS
count_letters	0.301	0.076	0.041	0.367	0.095	0.048	0.018	0.049	0.010	0.010
index_iter	0.520	0.000	0.000	0.440	0.000	0.000	0.000	0.000	0.000	0.000
maximum	0.076	0.009	0.018	0.110	0.012	0.021	0.020	0.034	0.011	0.011
minimum	0.114	0.005	0.006	0.250	0.008	0.013	0.012	0.027	0.007	0.007
reverse_by_swap	0.151	0.007	0.007	0.339	0.006	0.006	0.010	0.003	0.005	0.005
reverse_iter	0.139	0.001	0.003	0.220	0.001	0.003	0.000	0.013	0.000	0.000
search_iter	0.021	0.007	0.006	0.040	0.007	0.006	0.005	0.005	0.002	0.002
search_recur	0.131	0.047	0.053	0.146	0.031	0.042	0.030	0.017	0.020	0.020
sumup	0.004	0.002	0.007	0.004	0.000	0.000	0.000	0.000	0.000	0.000
swap_keys_values	0.206	0.022	0.039	0.318	0.013	0.018	0.015	0.020	0.021	0.017
mean	0.166	**0.018**	0.018	0.223	0.017	**0.016**	0.011	0.017	0.008	**0.007**

Table 3: Classification error. Number of times that each distance metric ranked the Refactory candidate repair as being closer to the buggy student solution than our expert annotations across (#prog) the total number of programs that the tool managed to repair.

	#prog	TED	SEQ	STR	NTED	NSEQ	NSTR	BLEU	CodeBLEU	ROUGE	ROUGELCS
count_letters	12	2	0	0	2	0	0	0	0	0	0
maximum	93	2	20	18	2	18	12	12	16	12	12
minimum	131	32	22	18	31	22	18	22	22	16	16
reverse_by_swap	148	10	8	10	10	10	10	8	2	2	2
reverse_iter	38	0	1	1	0	1	1	0	0	0	0
search_iter	141	6	17	15	11	17	15	17	10	13	13
search_recur	62	31	35	37	31	35	36	36	33	33	33
sumup	36	0	2	2	0	0	0	2	0	0	0
swap_keys_values	94	11	14	18	15	14	16	14	10	15	14
total	755	**94**	119	119	**102**	117	108	111	93	91	**90**

a quality difference between the automated repair tool corrections and experts' annotations, although the KDE hints that ROUGELCS is somewhat better at distinguishing differences at lower intermediate distances than the other two metrics.

The Empirical Cumulative Distribution Functions (ECDFs) for our selected distance measures give a more clear picture of the differences between the metrics and their portrayal of the relationship between the expert and candidate repairs. An ECDF plot tells us the proportion of the observations (i.e., repairs) y having a value (i.e., distance) lower than x. In our context, the ECDF tells us *the proportion y of candidate repairs which retain at least* $(100-(x*10))$ *percent of the elements of the original buggy solutions*. For instance, the ROUGELCS ECDF plot tells us that around 82% (respectively 86 %) of the repairs found by the Refactory tool (respectively our educators) have on average 80% of the words/token in the original students' buggy programs. In particular, looking at the ECDFs of the distance measures, we can quite clearly see how the ROUGE-based measure can separate expert and candidate repair distances from buggy solutions across most values while the edit distance measures do not show much difference for the lower scores (which are the most common ones as portrayed in the histograms). In the discussion, we further discuss the usefulness of the ECDF plots for comparing repairing techniques.

5 DISCUSSION

In this section, we review the answers to our research questions, discuss the impact of the results on computing education research, and highlight the limitations of our work.

5.1 Answering research questions

5.1.1 RQ1. Our results highlight that the sequence and the string edit distances perform equivalently well for selecting a candidate repair. On the other hand, the poor consistency of the results of the tree edit distance across assignments suggests that it might not be the most adequate for selecting similar solutions, as already suggested in hint generation problems [63].

5.1.2 RQ2. Our results also show that among Natural Language Processing based distance measures, the ROUGE measure performs consistently better than edit distances for retrieving expert candidate repairs.

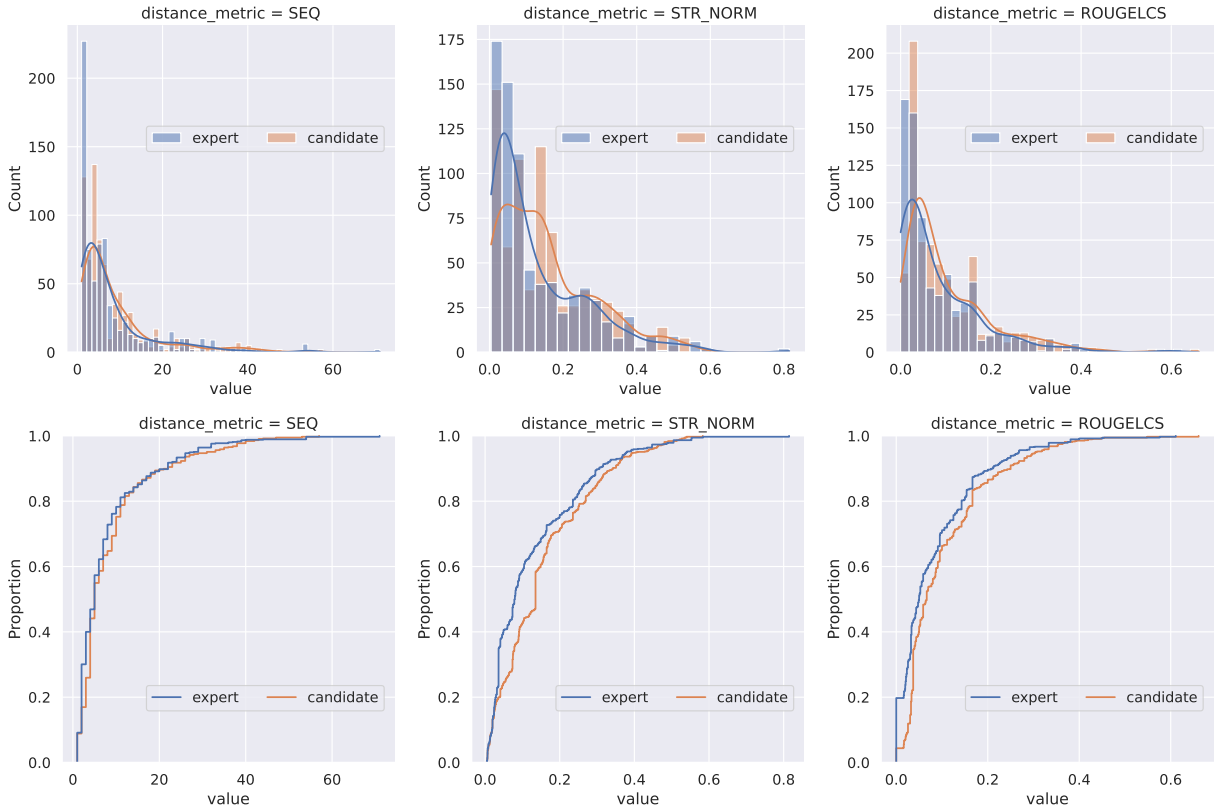

Figure 3: Visualization of distance distributions. The top row shows histograms and the bottom row the empirical cumulative distribution functions (ECDF) for the different measures. The measures are SEQ, STR_NORM, and ROUGELCS in the first, second, and third column respectively. The distances are computed between all the buggy programs and the expert annotations (in blue) and for candidate repairs (in orange) across all assignments.

5.1.3 Why ROUGE performs better. One might be intrigued by the results for RQ2. Even though the ROUGE-based measure and the sequence edit distance share many similarities (especially since they both use the token representation of code), ROUGE performed consistently better. The key underlying reason is that edit distances measure the number of operations needed to *transform* the buggy program into the candidate repair, while the ROUGE-based measure, as defined in our experiments, measures the *proportion* of the words/elements in the buggy program that can be found in the candidate (independently of whether the position of these elements has changed). As such, this hints to us that (1), the intrinsic aspect of distance captured in edit distances may be less useful in comparing repairs to students' programs as prior researchers using the metrics may have assumed, and (2) that what matters the most is that the measure captures the extent to which a correct solution retains elements from the original program. This later observation was confirmed by additional ablation experiments[3], where we observed that the BLEU-based scores achieve comparable results to the ROUGE score when considering the buggy program as the candidate "sentence" and the candidate repair as the reference "sentence".

[3]The results of all ablation experiments are available as part of the released code.

Still, we see that edit distance-based measures can be useful for small-scale errors such as fixing an extra semicolon immediately after an if statement – "`if (condition); { ... }`" [3] – although their performance in such cases would be only on par with the ROUGE-based measures. However, as novice programmers also have issues in how they structure their code (e.g. the right elements could be present but in the wrong order), they are bound to make structural changes including reordering their code. Similarly, students might decide, based on feedback, to refactor the code, e.g. by nesting functionality or by introducing functions and moving parts of the code into those functions (see e.g. [10, 72]). In such situations, NLP measures, in particular, the ROUGE-based score as presented, can be a better choice over the traditional edit distance metrics.

5.2 Methodological considerations

5.2.1 Our results are only valid for evaluation. Our results highlight the better adequacy of the ROUGE NLP measures for *evaluating* candidate repairs. These results might not hold for other aspects of the feedback pipeline, including constructing repairs. We view that there is a need for a multi-method approach for the process of *constructing a repair* to a buggy code. For small issues such as minor syntactic adjustments, edit distance metrics could still be a

viable option, while ROUGE-based measures could be meaningful if the repair would require restructuring the code. This would require additional research in determining an appropriate repair strategy.

In addition, although we observed that the tree edit distance performs inconsistently for retrieving good candidate repairs, it has often been used in the process of modifying a buggy program to match an existing correct solution [29, 48]. When working with a structured representation of the code (in most cases, the control flow graph [21, 29, 78]), leveraging distance measures which work on these same structured representations (i.e. tree edit distance) can be intuitive. One additional argument in favour of edit distances comes from their metric properties (i.e. edit/Levenshtein distances are metric in the mathematical sense), which is more advantageous for algorithms. Thus, the question of whether NLP measures can help bootstrap (rule-based) automated repair techniques remains open. Yet, we also suggest incorporating NLP measures into other areas where source codes are compared for evaluation, including plagiarism detection. Although, in the plagiarism detection case, naturally occurring similar code [69] would need to be kept in mind, as this might inflate similarity scores.

5.2.2 Choice of distance measure and ECDF. In addition to being more adequate for comparing individual candidates and selecting appropriate ones, NLP measures have other advantages as they carry more meaningful interpretations. We identify a new utility in looking at the distribution of the NLP-based distances using the Empirical Cumulative Distribution Functions (ECDF) of the ROUGE-based distance measure, shown in Figure 3, which highlights the use of ECDFs with NLP measures for understanding the overall distribution of the distances. The plots show distances for a given repair technique (in our experiment: expert annotation or Refactory) and the proportion of repairs from the technique which hold a given proportion of the original buggy code. The plots provide evidence of the number of expert annotations that retain a significantly larger proportion of the students' code compared to Refactory repairs. This allows comparing repairing techniques under a new dimension: one could interpret the ECDF plot as a success rate against a threshold, where one could decide to keep only the repairs that retain at least a given percentage of the original solution. This captures the risk/trade-off between finding a repair, and whether this repair is useful (quantity versus quality). In some aspects, this is similar to how people use precision against recall plots for comparing classifiers, for instance, for predicting various aspects of academic performance [28].

5.2.3 Handling special cases, program repair, and hint generation. During the annotation process, we observed solutions that were in a stage where finding a repair was impossible or meaningless. The two main reasons for this were (1) the solution was only partial and in a too early stage to be annotated (the student strategy to solve the problem was not clear yet), or (2) the student strategy was too far off, or the student strategy was not understandable or too complex. We see that future work should explore including a classification step to program repair in CER, where the classification step would decide whether a given program can meaningfully be repaired or not. In the case of partial programs, a better approach might be relying on a hint generation system [63] that would guide the student towards a proper and meaningful solution. In the case where a student's strategy is too far off, the approach should also be different, e.g. having an instructor intervention.

5.2.4 Multiple valid alternatives. We also observed that at times, multiple annotations or corrections were possible, all being as valid as the others. Annotators used their judgement to decide which one to choose, and different annotators might have come up with different solutions. In other words, although our expert annotations provide *one* ideal repair, other alternatives might be equally valid (and potentially closer to the buggy code). From the practical perspective, one possibility would be to provide different alternatives to students and have them pick one out of them, which would also provide code reading practice. Still, omitting the creation of multiple expert annotations in the case of multiple possibilities does not invalidate our existing results. We also note that such alternatives were present only in relatively few cases.

We note that we also conducted the same two experiments using students' corrections as an "expert repair". We did not show the results in the present work for conciseness[4]. The results confirm the relative performance of the distance measures, but absolute errors were all higher (meaning that the expert annotations are closer to the students' buggy code than the students' corrections). This observation is partly expected, as students may begin working on different parts of the code than where the problem lies, or completely change their approach to the task if they are not able to make their original strategy work, whereas expert annotators deliberately attempted to transform the original approach to a working solution while minimizing modifications.

5.2.5 An open annotated dataset of expert program repairs. In addition to our experiments showing the benefits of ROUGE NLP for evaluating program repairs, we also release the code and the annotated dataset on GitHub[5]. The data contains over one thousand expert-annotated submissions, where experts have studied students' buggy code and annotated it with the closest fix that addresses the bugs. The data will serve the CER community, supporting future program repair evaluations, and potentially also aid in other future research streams.

5.3 Limitations

Our work is not free of limitations. First of all, we annotated a selection of assignments all coming from one source. This means that the students' solutions in the data are in part driven by the tools and the pedagogy of the context in which the data was collected and that the generalizability of the results should be assessed with further annotated datasets. Although the stability of the present results over the assignments suggests that the results would hold across datasets, as does our interpretation of the underlying causes of the results, there may be tacit factors that could be revealed by data from another context. Moreover, we did not cross-validate the annotations between educators. In effect, individual educator's views on the "optimal" strategy for correcting the submission might be biased.

[4]The results of all ablation experiments are available as part of the released code.
[5]https://github.com/KoutchemeCharles/edmpr

Secondly, when contrasting the automated program repairs with the expert annotated repairs (Fig. 3), we only conducted a surface-level manual analysis of the automated repairs, providing us insight beyond the results and the visualizations and leading us to concur that the expert repairs were better. We acknowledge that we did not conduct an in-depth qualitative analysis of the automated program repairs, which could have highlighted specific cases where the automated program repairs might have been better. Future research can look into this using the data that we have released as a part of this work.

Finally, we only used one automated repair tool (Refactory) in the analysis for comparing candidate repairs. Although Refactory is a state-of-the-art (rule-based) Automated Program Repair tool, there are also non-rule-based techniques that are based for instance on machine learning models [61] or Natural Language Processing Techniques (in particular, Large Language Models) [82]. We note that these models, in general, have not released the source code and would have to be used over an API. This would require submitting student programs to external parties, which is not always possible due to privacy concerns.

6 CONCLUSION

In the present work, we explored distance metrics for evaluating program repairs. As repairs, we used both expert annotated code with close (and "ideal") program repairs as well as repairs generated using state-of-the-art rule-based program repair methods, which both built on an existing open dataset of introductory programming student code. We contrasted the commonly used edit distance metrics with NLP scoring metrics, studying to what extent the commonly used metrics apply in a context where the data comes from students who are learning to program. As the commonly used edit distance metrics, we evaluated string edit distance, sequence edit distance, tree edit distance, and explored also their bounded normalized versions. As the NLP scoring metrics, we explored the BLEU, CodeBLEU, ROUGE, and ROUGE-LCS metrics.

Our results suggest that, especially in the context of introductory programming where code can change drastically due to reordering of plans or due to refactoring, the research community should include the relative performance of proposed repair techniques by relying (also) on Natural Language Processing scores such as the ROUGE score. NLP metrics such as ROUGE provide insight into the proportion of elements present in the repair which can be especially useful if the repairs are more substantial when contrasted with the classic metrics that rely on the number of operations needed for transforming buggy programs to working programs. We further highlight the utility of ECDF plots for assessing repairs as they provide a quick visual of repair performance, and can potentially be used for creating and visualizing a threshold for the amount of code that should be retained from the original code after repairs.

In part based on the ECDF plots, we highlight the necessity to consider that not all buggy programs should be repaired. We see a need to include a new step in program repair methods, which would include first judging whether a buggy program should be repaired. Alternative options could be, for example, using a hint generation system to help the student proceed, and creating teacher interventions.

Although there is rapid development in automated repair techniques for education, we have unfortunately observed that the results of proposed repair techniques are often reported without releasing the dataset (or the code). Although the collection and storing of student submission data (and more fine-grained data) has become more common over time [32], privacy concerns have often limited the possibility of sharing such data with the public. This makes the comparison, adoption, and importantly deployment of these automated tools in real-life scenarios more challenging. As an additional contribution, we release our annotated dataset as a part of this work, in the hope that the dataset can be used to compare novel program repair techniques applied in computing education research.

Future work. As a part of our future work, we are working on annotating another dataset comprising students' buggy solutions [29], which will provide further evidence of the generalizability of our results. We are also working on feedback mechanisms based on program repairs, which in our present work focuses on identifying the key causes for the failure of the programs. We note that our released dataset already contains comments on these reasons (i.e. why the annotated code fails the given tests), and we are currently working on synthesizing them for the purposes of creating a bug classification notebook. We note that given that we possess some notion of ground truth correction to a buggy program, we can investigate bug localization techniques. While repairs to programs entail both locating and fixing the problem(s), bug localization techniques (which are only concerned with location) are also useful forms of feedback [23].

ACKNOWLEDGMENTS

We are grateful for the grant from the Ulla Tuominen Foundation to the third author.

REFERENCES

[1] Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. 78–87.

[2] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.

[3] Amjad Altadmri and Neil CC Brown. 2015. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM technical symposium on computer science education*. 522–527.

[4] David Azcona, Piyush Arora, I-Han Hsiao, and Alan Smeaton. 2019. user2code2vec: Embeddings for Profiling Students Based on Distributional Representations of Source Code. In *Proceedings of the 9th International Learning Analytics & Knowledge Conference (LAK'19)*. ACM.

[5] Brett A Becker. 2016. A new metric to quantify repeated compiler errors for novice programmers. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 296–301.

[6] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210.

[7] Sahil Bhatia and Rishabh Singh. 2016. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *ArXiv* (2016).

[8] Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit.* " O'Reilly Media, Inc.".

[9] Neil CC Brown and Amjad Altadmri. 2017. Novice Java programming mistakes: Large-scale data vs. educator beliefs. *ACM Transactions on Computing Education (TOCE)* 17, 2 (2017), 1–21.

[10] Charis Charitsis, Chris Piech, and John C Mitchell. 2023. Detecting the Reasons for Program Decomposition in CS1 and Evaluating Their Impact. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 1014–1020.

[11] Guillaume Cleuziou and Frédéric Flouvat. 2021. Learning student program embeddings using abstract execution traces. In *Proceedings of the 14th Educational Data Mining conference*. https://educationaldatamining.org/EDM2021/virtual/poster_paper70.html

[12] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. 75–80.

[13] Paul Denny, James Prather, Brett A Becker, Catherine Mooney, John Homer, Zachary C Albrecht, and Garrett B Powell. 2021. On Designing Programming Error Messages for Novices: Readability and Its Constituent Factors. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[14] Paul Denny, Jacqueline Whalley, and Juho Leinonen. 2021. Promoting early engagement with programming assignments using scheduled automated feedback. In *Proceedings of the 23rd Australasian Computing Education Conference*. 88–95.

[15] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4–es.

[16] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.

[17] Thomas Dy and Ma Mercedes Rodrigo. 2010. A detector for non-literal Java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 118–122.

[18] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*. 83–89.

[19] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[20] Google-Research. [n. d.]. Google-Research/Rouge at master · google-research/google-research. https://github.com/google-research/google-research/tree/master/rouge

[21] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. http://arxiv.org/abs/1603.03165 arXiv:1603.03165 [cs].

[22] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (July 2019), 930–937. https://doi.org/10.1609/aaai.v33i01.3301930 Number: 01.

[23] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. *Neural Attribution for Semantic Bug-Localization in Student Programs*. Curran Associates Inc., Red Hook, NY, USA.

[24] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 31, 1 (Feb. 2017). https://ojs.aaai.org/index.php/AAAI/article/view/10742 Number: 1.

[25] John Hattie and Helen Timperley. 2007. The power of feedback. *Review of educational research* 77, 1 (2007), 81–112.

[26] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[27] Kenny Heinonen, Kasper Hirvikoski, Matti Luukkainen, and Arto Vihavainen. 2014. Using codebrowser to seek differences between novice programmers. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 229–234.

[28] Arto Hellas, Petri Ihantola, Andrew Petersen, Vangel V. Ajanovski, Mirela Gutica, Timo Hynninen, Antti Knutas, Juho Leinonen, Chris Messom, and Soohyun Nam Liao. 2018. Predicting Academic Performance: A Systematic Literature Review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education* (Larnaca, Cyprus) *(ITiCSE 2018 Companion)*. Association for Computing Machinery, New York, NY, USA, 175–199. https://doi.org/10.1145/3293881.3295783

[29] Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based Program Repair applied to Programming Assignments. In *2019 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. IEEE/ACM, 388–398.

[30] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning MOOC. *AIED 2013 Workshops Proceedings Volume* 1009 (Jan. 2013), 25.

[31] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[32] Petri Ihantola, Arto Vihavainen, Alireza Ahadi, Matthew Butler, Jürgen Börstler, Stephen H Edwards, Essi Isohanni, Ari Korhonen, Andrew Petersen, Kelly Rivers, et al. 2015. Educational data mining and learning analytics in programming: Literature review and case studies. *Proceedings of the 2015 ITiCSE on Working Group Reports* (2015), 41–63.

[33] Matthew C Jadud. 2005. A first look at novice compilation behaviour using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40.

[34] Matthew C Jadud. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*. 73–84.

[35] W. Lewis Johnson, Elliot Soloway, Benjamin Cutler, and Steven Draper. 1983. *Bug Catalogue: I*. Technical Report. Yale University, YaleU/CSD/RR #286.

[36] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-Supervised Verified Feedback Generation. arXiv:1603.04584 [cs.SE]

[37] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2014. Strategy-based feedback in a programming tutor. In *Proceedings of the computer science education research conference*. 43–54.

[38] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43.

[39] Tobias Kohn. 2019. The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 524–530.

[40] Teemu Koivisto and Arto Hellas. 2022. Evaluating CodeClusters for Effectively Providing Feedback on Code Submissions. In *2022 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–9.

[41] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4 (2003), 249–268.

[42] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. *Acm sigcse bulletin* 37, 3 (2005), 14–18.

[43] Juho Leinonen, Paul Denny, and Jacqueline Whalley. 2022. A comparison of immediate and scheduled feedback in introductory programming projects. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*. 885–891.

[44] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A Becker. 2023. Using large language models to enhance programming error messages. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 563–569.

[45] Chin-Yew Lin. 2004. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*. Association for Computational Linguistics, Barcelona, Spain, 74–81. https://aclanthology.org/W04-1013

[46] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]

[47] Andrew Luxton-Reilly, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, and Claudia Szabo. 2018. Introductory programming: a systematic literature review. In *Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*. 55–106.

[48] Yana Malysheva and Caitlin Kelleher. 2022. An Algorithm for Generating Explainable Corrections to Student Code. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '22)*. Association for Computing Machinery, New York, NY, USA, Article 13, 11 pages. https://doi.org/10.1145/3564721.3564731

[49] Samiha Marwan, Nicholas Lytle, Joseph Jay Williams, and Thomas Price. 2019. The impact of adding textual explanations to next-step hints in a novice programming environment. In *Proceedings of the 2019 ACM conference on innovation and technology in computer science education*. 520–526.

[50] Jessica McBroom, Irena Koprinska, and Kalina Yacef. 2021. A survey of automated programming hint generation: The hints framework. *ACM Computing Surveys (CSUR)* 54, 8 (2021), 1–27.

[51] Davin McCall and Michael Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. IEEE, 1–8.

[52] Davin McCall and Michael Kölling. 2019. A new look at novice programmer errors. *ACM Transactions on Computing Education (TOCE)* 19, 4 (2019), 1–30.

[53] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*. 491–502.

[54] Henrik Nygren, Juho Leinonen, and Arto Hellas. 2019. Non-restricted Access to Model Solutions: A Good Idea?. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 44–50.

[55] Benjamin Paaßen, Bassam Mokbel, and Barbara Hammer. 2015. A Toolbox for Adaptive Sequence Dissimilarity Measures for Intelligent Tutoring Systems.

In *Proceedings of the 8th International Conference on Educational Data Mining (EDM 2015)* (2015-06), Olga Christina Santos, Jesus Gonzalez Boticario, Cristobal Romero, Mykola Pechenizkiy, Agathe Merceron, Piotr Mitros, Jose Maria Luna, Christian Mihaescu, Pablo Moreno, Arnon Hershkovitz, Sebastian Ventura, and Michel Desmarais (Eds.). International Educational Datamining Society, 632–632. http://www.educationaldatamining.org/EDM2015/uploads/papers/paper_257.pdf

[56] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education (TOCE)* (2022).

[57] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Philadelphia, Pennsylvania, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[58] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. *arXiv preprint arXiv:2302.04662* (2023).

[59] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*. 195–204.

[60] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29 (2019), 368–395.

[61] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. *arXiv:1607.02902 [cs]* (July 2016). http://arxiv.org/abs/1607.02902 arXiv: 1607.02902.

[62] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]

[63] Kelly Rivers and Kenneth R Koedinger. 2013. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, Vol. 50. 50–59.

[64] Kelly Rivers and Kenneth R Koedinger. 2014. Automating hint generation with solution space path construction. In *Intelligent Tutoring Systems: 12th International Conference, ITS 2014, Honolulu, HI, USA, June 5-9, 2014. Proceedings 12*. Springer, 329–339.

[65] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27 (2017), 37–64.

[66] Anthony Robins, Patricia Haden, and Sandy Garner. 2006. Problem distributions in a CS1 course. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 165–173.

[67] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 311–322.

[68] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. 87–96.

[69] Simon, Oscar Karnalim, Judy Sheard, Ilir Dema, Amey Karkare, Juho Leinonen, Michael Liut, and Renée McCauley. 2020. Choosing code segments to exclude from code similarity detection. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 1–19.

[70] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[71] Rebecca Smith and Scott Rixner. 2019. The error landscape: Characterizing the mistakes of novice programmers. In *Proceedings of the 50th ACM technical symposium on computer science education*. 538–544.

[72] Elliot Soloway. 1986. Learning to program= learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858.

[73] Elliot Soloway, Jeffrey G. Bonar, and Kate Ehrlich. 1983. Cognitive strategies and looping constructs: An empirical study. *Commun. ACM* 26, 11 (1983), 853–860. https://doi.org/10.1145/182.358436

[74] Elliot Soloway, Kate Ehrlich, Jeffrey G. Bonar, and Judith Greenspan. 1982. What do novices know about programming? In *Directions in Human–Computer Interactions*, Albert Badre and Ben Shneiderman (Eds.). Vol. 6. Ablex Publishing, 27–54.

[75] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[76] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th koli calling international conference on computing education research*. 109–116.

[77] Arto Vihavainen, Thomas Vikberg, Matti Luukkainen, and Martin Pärtel. 2013. Scaffolding students' learning using test my code. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 117–122.

[78] Ke Wang, RIshabh Singh, and Zhendong Su. 2017. Data-Driven Feedback Generation for Introductory Programming Exercises. *arXiv:1711.07148 [cs]* (Nov. 2017). http://arxiv.org/abs/1711.07148 arXiv: 1711.07148.

[79] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. https://doi.org/10.48550/arXiv.1711.07163 arXiv:1711.07163 [cs].

[80] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*. IEEE, 319–323.

[81] Jacqueline Whalley, Amber Settle, and Andrew Luxton-Reilly. 2021. Novice reflections on debugging. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 73–79.

[82] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. https://doi.org/10.48550/ARXIV.2209.14876