Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

Neural Models for Automatic Program Repair vs. Human Developers

Dominik Huber

Course of study:	Softwaretechnik M. Sc.
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Matteo Paltenghi
Started:	November 2, 2021
Completed:	May 2, 2022

University of Stuttgart
Germany

SOLA
SoftwareLab

Abstract

On the field of automatic program repair, approaches for code repair increasingly employ neural models of code for a variety of tasks. However, the repair performance of APR approaches is relatively low, and the models behave similarly to black boxes in that it is hard to determine why models make certain predictions. A better understanding of how neural models for program repair arrive at their conclusions could lead to improved training methods, model architectures, and increased trust in model predictions. Many neural models feature attention mechanisms, which are one way of gaining insight into the models' reasoning. This thesis approaches this problem by developing an approach for tracking the attention and behavior of human developers working on code repair tasks. We employ the approach in a 27-participant human study and conduct a comparative analysis of the attention gathered from the humans and that extracted from two APR approaches. Regarding repair performance, we find that human developers drastically outperform current APR approaches, with ca. 70% of fix candidates being correct for the humans and only approximately 1% being correct for the models. We further find that there is a weak to moderate but statistically significant correlation between the attention of the models and the humans (mean Spearman rank correlation between 0.34 and 0.43). The correlation analysis also shows that encoding input programs as an AST and limiting the areas of code visible to the model results in an overall more human-like attention distribution. We highlight differences in how much attention is paid to the buggy line, and recommend the introduction of a new hyperparameter to balance attention between the buggy line and the context. Finally, we identify two patterns in developer behavior that may benefit the design of future APR approaches.

Zusammenfassung

Für die automatische Reparatur von Programmcode kommen zunehmend häufiger künstliche neuronale Netze zum Einsatz. Die Leistung dieser Netze ist auf diesem Gebiet jedoch noch relativ niedrig, zumal es sehr schwierig ist, die Ausgaben solcher Netze nachzuvollziehen. Eine höhere Erklärbarkeit der Modelle könnte besssere Trainingsmethoden und Architekturen ermöglichen, sowie das Vertrauen in die Modelle erhöhen. Sog. Attention-Mechanismen stellen eine Möglichkeit dar, die Erklärbarkeit von Modellen zu erhöhen. Um dieses Problem zu lösen, erarbeitet die vorliegende Arbeit einen neuartigen Ansatz, um das Verhalten und die Aufmerksamkeitsverteilung von menschlichen Entwicklern bei der Codereparatur aufzuzeichnen. Auf Basis dieser Daten führen wir eine vergleichende Analyse zwischen 27 menschlichen Teilnehmern und zwei neuronalen Modellen durch, um Unterschiede und Gemeinsamkeiten zwischen menschlicher Aufmerksamkeit und den Attention-Mechanismen herauszuarbeiten. Bezüglich der Reparaturleistung stehen die Entwickler klar vor den aktuellen Ansätzen. Etwa 70% der Versuche von Entwicklern sind korrekt, bei den Modellen sind gerade einmal ca. 1% der Reparaturkandidaten korrekt. Weiterhin beobachten wir eine schwache bis mittelmäßige, jedoch statistisch signifikante, Korrelation zwischen der Aufmerksamkeit von Menschen und Attention von Modellen (mittlere Spearman rank Korrelation zwischen 0.34 und 0.43). Ein Einlick aus der Korrelationsanalyse ist, dass die Darstellung von Programmen als abstrakter Syntaxbaum ggf. dem menschlichen Denken ähnlicher ist, als die Darstellung als Token-Sequenz. Die Ergebnisse zeigen starke Unterschiede in der Aufmerksamkeit, die auf die fehlerhafte Zeile gerichtet ist, auf. Wir empfehlen die Einführung eines neuartigen Hyperparameters, um die Balance zwischen fehlerhafter Zeile und Kontext zu steuern. Zuletzt identifizieren wir zwei Verhaltenmuster von menschlichen Entwicklern, die für die weitere Forschung auf dem Gebiet der automatischen Programmreparatur relevant sein können.

Contents

1 Introduction

Automatic Program Repair (APR) is an emerging field of software engineering with enormous potential for a variety of applications [15]. The prospect of being able to generate correct fixes for a variety of bugs in software opens up a range of potential use cases: Continuous integration systems could suggest possible fixes if they detect a bug during software deployments, or a connected APR approach could immediately fix vulnerabilities detected by fuzzing tools [15].

A recent trend in automatic program repair is the usage of deep learning (DL) models for a variety of specialized tasks: For example, Allamanis et al. [2] introduce a graph-based neural network specializing in fixing incorrect usages of variables in program code. Bhatia et al. [6] use an approach based on recurrent neural networks to automatically generate fixes for syntax errors to enable automatic correction of student submissions in massive open online programming courses. An example of an approach for general repair is Hoppity [13], which learns a series of transformations for a graph representing the input program to fix a variety of JavaScript bugs.

However, the performance of current APR approaches has room for improvement. Considering the popular Defects4J [18] benchmark, which consists of real-world bugs extracted from open-source repositories, recent APR approaches can only produce a correct fix for under 15% of the bugs in the dataset [17, 39].

The fact that deep learning models essentially behave similar to black boxes compounds these issues: Especially for models that produce good results [35], it is very difficult to determine what influenced the model's decisions. This intransparency reduces trust in the model's decisions and makes it harder to identify erroneous relationships that the models have learned, making it harder to improve the models as a consequence [35].

Attention mechanisms are a type of neural network component modeled after the intuition of human attention. Originating from models built for sequence-to-sequence translation [4], attention mechanisms allow a neural model to "pay attention" to the input sequence, i.e., decide which parts of the input sequence are especially relevant for which parts of the output sequence. Based on the attention weights, one can interpret and make assumptions about the model's reasoning [4], and judge the importance allocated to parts of the input from a human perspective. This understanding of the models' reasoning can then be used to improve the models.

To leverage this potential for the benefit of automatic program repair, we conduct a comparative study of human developers and APR approaches. For this, we develop a methodology to track the behavior of human developers during bug fixing, including a notion of human attention. We then compare the collected human attention data to that of two neural models for automatic program repair to identify similarities and differences and suggest improvements for future APR approaches.

Additionally, we analyze the overall behavior of human developers to identify patterns that may benefit future research in this area.

The comparative analysis aims to answer the following research questions:

RQ1: How do human developers and APR approaches compare in terms of repair performance

RQ2: Do the attention vectors extracted from APR approaches correlate to those of the human participants?

RQ3: Is there a relationship between high human-model correlation and model repair performance?

RQ4: What proportion of their attention do developers and models pay to the buggy line versus the surrounding context?

RQ5: How much attention do developers and humans pay to different types of tokens?

RQ6: What patterns are there in the behavior of human developers?

Chapter 4 answers these research questions. In summary, this thesis makes the following contributions:

- An approach for recording the attention and behavior of human developers working on code repair tasks.
- An evaluation of developer performance on the Quixbugs benchmark, as a point of comparison for current and future work on automatic program repair.
- A comparative analysis of similarities and differences between the attention of neural models and human developers, highlighting areas of improvement for future APR approaches.
- An analysis of the behavior of human developers yielding two patterns of human behavior that may benefit future work on automatic program repair.
- A dataset of 99 human code editing records, which include a variety of recorded data, including attention maps and edit events.

The implementation of the approach, the tooling used in the evaluation, as well as the instrumented copies of the APR approaches are part of the GitHub repositories accompanying this thesis.

In the following, Chapter 2 introduces the relevant background knowledge for the topics covered in this thesis. Chapter 3 explains the methodology used for the experimental evaluation, the results of which are presented by Chapter 4. Chapter 5 gives an overview of relevant related work, while Chapter 6 concludes the thesis with a brief discussion of limitations, the key conclusions, and future work.

2 Background

This chapter gives an overview of the background of this thesis, starting with a general overview of the field of automatic program repair in Section 2.1. Section 2.2 introduces the neural network architectures relevant to this work, including the purpose and function of the attention layers found in each architecture, followed by the concept of copy attention. Section 2.3 follows with an overview of important bug datasets. Finally, Section 2.4 closes by distinguishing between the different notions of the correctness of a patch.

2.1 Automatic Program Repair

The goal of automatic program repair is to assist human developers by supplying potential patches for bugs in software systems. Its potential applications include the automatic suggestion of patches as part of automated build and deployment systems, the automatic repair of security vulnerabilities, automated tutoring for online programming courses, and on-the-fly repair of non-functional issues [15].

Conventional APR approaches often use test suites as a correctness criterion for generated patches. LeGouges et al. [15] distinguish between two high-level types of APR approach: Those that use heuristics to search for valid patches in the search space of all possible modifications to the input program's code, and those that use certain constraints on the resulting code to search for valid patches. In the latter strategy, the provided test suite is used as a basis for the generation of the constraints. As these two strategies share a heavy reliance on test suites, they also share a common challenge that affects APR approaches, namely overfitting to the test suite. In this phenomenon, an APR approach will generate patches that pass all tests provided to it, but are not semantically correct, and might even introduce new bugs that are not covered by the test suite.

For example, Smith et al. [29] conduct an evaluation of two APR approaches on a benchmark of 998 bugs with regard to potential overfitting of the patches. They find that about one-quarter of the patches generated by the APR approaches fail an additional test suite that covers more than the training test suite provided to fix the bug. In a more recent evaluation, Ye et al. [38] find that in a study conducted with 10 APR approaches on the Quixbugs benchmark, roughly half (53.3%) of the patches that pass all initial test cases suffer from overfitting.

A recent trend in automatic program repair is to make use of advances in the field of machine learning to build APR approaches. The following section gives an overview of the application of neural models for automatic program repair and explains neural network architectures that are important in the context of this work.

2.2 Neural Architectures

There is a variety of potential use cases for neural models of code for automatic program repair. Examples of approaches range from simple recurrent neural networks (RNN) that fix syntax errors in student code [6] to complex graph-based architectures that are capable of applying different edits to the tree representation of a program [13].

A frequent pattern in the architecture of APR approaches is to adapt architectures and concepts originally used in natural language processing. For example, Tufano et al. [33] use an encoder/decoder architecture with attention, a pattern originating in neural machine translation [4], to "translate" from buggy to fixed code, and thereby generate patches. They essentially formulate the task of repairing buggy code as a translation task that consists of translating from one kind of code to another. Analogously, tasks such as code summarization can be reformulated as a natural language summarization task, for which existing architectures can be re-used.

The following subsections explain the neural architectures and related concepts relevant to this work, namely those relevant for the two APR approaches considered in this work, SequenceR [11] and Recoder [39].

2.2.1 Encoder/Decoder with Attention

The encoder/decoder architecture is a generic sequence-to-sequence architecture that originated in the realm of neural machine translation [32]:. It consists of two connected neural networks, typically RNNs, the encoder, and the decoder. The encoder takes the input sequence as an input and generates a fixed-size vector that encodes information about the sequence. The decoder takes this vector as an input and generates the output sequence based on it. Due to the intermediate vector, there is no requirement for the input and output sequences to have the same length, which makes the architecture applicable to generic sequences. As a consequence, the encoder/decoder architecture is not specific to natural language processing tasks and can be used for arbitrary sequence-to-sequence problems. [32]

An extension of the encoder/decoder architecture is the addition of an attention mechanism introduced by Bahdanau et al. [4]: The standard encoder/decoder architecture has difficulties with very long input sequences, as the amount of information the fixed-size vector can hold is limited. The attention mechanism allows the decoder to generate its outputs based on relevant parts of the input sequence, with the attention mechanism deciding which parts of the input sequence influence the decoder at a given time step. [4]

The resulting attention weights can be interpreted to determine what the learned relationship between parts of the input and output sequences are, and provide insight into the model's reasoning to a certain extent. For APR approaches that use this architecture, this allows assessment and interpretation of what areas of the source code influence the model in producing a fix. In the context of this work, SequenceR [11] uses an encoder/decoder architecture with attention to predict a replacement for the buggy line based on the sequence of tokens of the input program.

2.2.2 Copy Attention

The term copy attention denotes a concept that originates from research related to the task of text summarization. See et al. [28] augment the encoder/decoder with attention architecture with a pointer-generator network, which enables the decoder to copy parts of the input sequence instead of generating them: specifically, it calculates a generation probability which determines whether to generate a token from the vocabulary or to copy it from the input sequence based on the attention distribution. The main application of this mechanism is mitigating the out-of-vocabulary problem: Both natural and programming languages can have an arbitrarily large amount of valid words, which poses a challenge for neural models that perform worse as vocabulary size grows. The copy attention mechanism allows the model to circumvent that problem by copying out-of-vocabulary tokens from the input. SequenceR's decoder makes use of this concept, meaning that it can copy tokens of the input program into the replacement line that it generates [11].

2.2.3 Transformer

A general property of recurrent neural networks is that they do their computations over several time steps, which correspond to tokens of the input sequence. Especially for longer input sequences, this can lead to performance issues, as it is by design not possible to parallelize the computations. To address these issues, Vaswani et al. [34] propose the *Transformer* architecture, which is an encoder/decoder architecture that exclusively relies on attention mechanisms instead of recurrent units to extract relationships between the input and the output. A key feature of the Transformer architecture is the ability to conduct many of its operations in parallel, leading to a massive performance improvement over recurrent architectures.

The Transformer employs attention mechanisms for three different purposes [34]: Analogous to the purpose of attention in the "classic" encoder/decoder architecture, the first purpose of attention in the Transformer is to allow the decoder to pay attention to tokens of the input sequence. The second use of attention is self-attention layers in the encoder, where the different parts of the input sequence can pay attention to other parts of the same input sequence, which allows the extraction of relationships between different parts of the sequence. Finally, the decoder also uses self-attention layers to allow its positions to pay attention to previous positions.

An important feature of the attention mechanism used in the Transformer architecture is the concept of multi-head attention [34], where there are multiple attention "heads" that pay attention to the input. Each head extracts different features from its input, allowing each head to specialize in certain patterns in the input, for example.

Recoder, one of the APR approaches included in this work's experimental evaluation, uses an encoder based on the Transformer architecture to encode the bug and its surrounding context [39].

2.3 Bug Datasets

On the field of automatic program repair, benchmark bug datasets are frequently used in the evaluation of APR approaches, for instance the Defects4J [18] dataset, which is used in the evaluations

of [11], [39], [23], and [17]. For the sake of comparability and reproducibility, it is sensible to re-use such datasets in experimental evaluations, as any results can be compared directly to results of other evaluations using the same dataset.

This section gives an overview of important datasets on the field of automatic program repair, especially those that are candidates for inclusion in the experimental evaluation. Table 2.1 lists the candidate datasets, along with their target language, the type, and number of bugs contained within. Section 3.3 discusses the advantages and disadvantages of each dataset in the light of the requirements set for the program repair task and explains the choice of the Quixbugs dataset for the experimental evaluation. The following Subsections 2.3.1 to 2.3.3 give more detailed information on the top three datasets in Table 2.1.

Dataset	Language	Bug Type	Number of bugs	Tests
Defects4J [18]	Java	Mixed, mined from repositories	835 (v2.0.0)	x
ManySStuBs4J [19]	Java	Single statement bugs	153,652	(x)
QuixBugs [22]	Java, Python	Single-line “challenge” bugs	40	x
ETH Py-150 [7]	Python	Mixed, mined from repositories	150,000	
Bugs.jar [27]	Java	Mixed, mined from repositories	1158	x

Table 2.1: Bug datasets.

2.3.1 Defects4J

Defects4J is a database of Java bugs introduced by Just et al. [18] in 2014 to provide a set of real-world bugs for research purposes. The database includes a high-level interface designed to abstract away the checking out, the building, and testing of various versions of the programs contained within via a dedicated test execution framework. It was introduced with the goal of enabling reproducible experiments in software engineering research [18] and is consequently used in the evaluations of most of the approaches targeting Java that are considered in this work, as per the above example.

2.3.2 ManySStuBs4J

ManySStuBs4J is a dataset first introduced by Karampatsis and Sutton in 2020 [19] as part of an investigation into the frequency of certain patterns of bugs occurring in code. It only contains instances of bugs where the bug is limited to a single statement. The dataset has four subsets, divided by size and whether the bugs contained within fit into any of the 16 bug patterns defined in that work [19, 20]:

The first subset is called the small bug dataset and contains 25,539 bugs mined from 100 open-source repositories, and includes automated build and test suites for reproducibility. The second subset called “SStuBs” (“simple, stupid bugs”), is a subset of the first and contains those bugs out of the first that fit into any of the defined bug patterns. Being a subset of the first, it also features automated build and test suites for bug reproducibility. The third subset, the large bug dataset, contains 153,652 single-statement bugs mined from 1000 open-source repositories. The fourth and final subset, “SStuBsLarge”, is a subset of the third dataset and contains “simple, stupid” bugs analogous to the second dataset.

2.3.3 QuixBugs

QuixBugs is a benchmark dataset consisting of 40 bugs taken from the so-called Quixey challenge, a hiring challenge conducted by the homonymous startup, where developers had one minute to fix the given implementations of "classic" algorithms [22]. A notable feature of the dataset is that the faulty programs are available in both Python and Java, with the dataset being originally intended to provide a multilingual benchmark for automatic program repair approaches [22]. The dataset also includes test cases and a framework for automatically running the contained programs [22].

QuixBugs is increasingly used as a benchmark for APR approaches with a focus on evaluating the ability of approaches to generalize, since it contains a category of bugs that is distinctly different from those contained in other bug datasets, such as Defects4J. Notably, Ye et al., in [38], conduct a study of 10 APR approaches on QuixBugs, finding that the approaches are capable of repairing 16/40 bugs with at least plausible fixes, and that of these fixes, 53.3% are overfitting patches. QuixBugs is also included in the experimental evaluations of newer neural APR approaches, such as Recoder [39] or CoCoNuT [23].

2.4 Correctness of Patches

As briefly discussed in Section 2.1, there are different notions of the correctness of a patch intended to repair a buggy program. To establish clear terminology, this work adopts the correctness definitions used by Ye et al. [38]. By their definition, a *plausible* patch is a patch that results in a fixed program that passes all tests of the corresponding test suite. A *correct* patch is one where the resulting program is semantically equivalent to a reference implementation. This work also counts formally correct implementations of the intended program behavior as a correct patch, even if they are not semantically equivalent to the reference patch.

An example of a patch that is plausible, but not correct, would be a trivial patch that outputs the expected values for each test case, and random values in all other cases. This patch trivially passes all test cases and is therefore plausible in the sense of the definition, but not a formally correct implementation of the intended behavior of the program.

Plausible but not correct patches are often denoted overfitting patches [29]. This comes from APR approaches that use the test suite as an oracle of patch correctness, and modify the input program until it passes all tests in the test suite. The approaches considered in this work do not use the test suite in that way, so they are not technically overfitting to the test suite when they produce plausible but not correct patches.

3 Methodology

This chapter presents the methodology we use for the comparative analysis of human and machine attention, which is the central focus of the experimental evaluation presented in this work. Section 3.1 starts by defining the program repair task chosen for both humans and APR approaches, as well as the rationale behind the task definition. The following Section 3.2 introduces the definition of human attention in the context of this work and explains important decisions taken for the process of human attention collection, followed by the underlying data model. Sections 3.3 and 3.4 present the bug dataset and the target language used in the experimental evaluation, and substantiate each choice.

To conclude the human part of the methodology, Sections 3.5 and 3.6 describe the concrete implementation of the Edit Attention Recorder, the platform used for collecting human attention data, and the recruitment of participants for the human study, respectively. Concerning the APR approaches, Section 3.7 explains the choice of the APR approaches used for the evaluation, while Section 3.8 lists and explains any modifications made to the approaches as part of the study. The central part of the experimental evaluation is the program repair task, which defines what problems humans and APR approaches are given to solve. The following section defines and explains the repair task.

3.1 Program Repair Task

Due to the diverse nature of software bugs, there exists a wide range of possible program repair tasks. One example is the VarMisuse task first introduced by Allamanis et al. [2], where the bug consists of the misuse of an in-scope variable. An example of a more general repair task is the task of repairing bugs in the Defects4J dataset [18], where repairs can range from the replacement of an incorrect operator to the addition of several lines of domain-specific code.

For the selection of the program repair task, three primary factors are taken into account:

1. **Human fixability:** To enable a human study, the selected repair task must be feasible for human developers using reasonable effort. Therefore, this work derives the following requirements on the complexity and the amount of required context for the repair task: The task should not require extensive domain knowledge, as that would limit the set of available participants. In addition, the task should be sufficiently easy so that human developers are able to complete it in a reasonable time, to ensure that sufficient data can be collected on different bugs.

2. **Model fixability:** The repair task must consist of bugs that are compatible with the input format taken by the selected APR approaches, and be fixable in the sense that the approaches are able to successfully repair at least a subset of the bugs.

3. **Comparability:** Another central requirement for the comparative study is to keep the concrete tasks given to models and humans as close to each other as possible, to enable a direct and fair comparison. This especially requires that any data provided to one should also be provided to the other, and vice-versa with data that is not considered by either.

Following these requirements, this work defines the following program repair task for both humans and neural approaches:

Program Repair Task: Given a faulty program in the form of a single source code file, containing a single bug that is limited to a single buggy line, fix the bug such that the program behaves as intended. Fixing the bug is possible by editing the given buggy line, and does not require any further edits outside of the buggy line. The semantics of the bug, especially the intended behavior of the program, have to be inferred from the program itself. The given inputs for the task are the buggy program and the buggy line for both neural approaches and humans, and additionally a brief natural language description of the underlying algorithm for the humans only.

```
public class BITCOUNT {
    public static int bitcount(int n) {
    int count = 0;
    while (n != 0) {
        n = (n ^ (n - 1));
        n = (n & (n - 1));
        count++;
    }
    return count;
    }
}
```

Figure 3.1: Example bug.

Figure 3.1 shows an example of a bug from the Quixbugs dataset [22], which conforms to the definition of the repair task. In this case, the bug consists of an incorrect bitwise operator in a program that counts the number of 1-bits in the binary representation of an integer. To make the program behave as intended, the `^` (bitwise XOR) operator on the line marked in red has to be replaced with `&` (bitwise AND), as indicated by the line marked in green.

The limitation to a single source code file is based on the complexity requirement, whereas the choice of single-line bugs is motivated both by the complexity requirement, as well as the

capabilities of the chosen neural approaches, one of which only processes single-line bugs. The task definition includes an additional textual description of the algorithm as part of the inputs for the human participants, which is a deviation from the principle of keeping the tasks as close as possible between humans and machines. Subsection 4.8.1 gives the reasons for this choice and discusses its implications for the validity of the experimental evaluation.

This work uses the Quixbugs [22] dataset as a source of bugs conforming to the above definition. Section 3.3 gives the rationale behind the selection of this dataset, as well as more details on the bugs contained therein. The repair task definition states that the buggy line is a part of the inputs. This is due to a common assumption in automatic program repair, namely that of perfect fault localization. The following subsection elaborates on this assumption and explains the rationale behind making it.

3.1.1 Fault Localization

Fault localization (FL) consists of identifying a program element that is the root cause of a given software failure [40]. In a comprehensive study of existing fault localization approaches, Zou et al. [40] report that across a range of families of fault localization techniques, the best-performing approaches correctly localize between 7% ($n = 1$) and 44% ($n = 10$) in a top-n metric. They conduct their evaluation using statement-level localization on the Defects4J [18] benchmark.

In automatic program repair, fault localization approaches are frequently used in conjunction with a repair approach that is typically unable to do localization by itself [36]. Both of the repair approaches considered in this work, SequenceR [11] and Recoder [39], take the location of the bug in the form of the buggy line number as an input.

Considering the above results for state-of-the-art fault localization, using a fault localization approach together with the APR approaches would incur a risk of the FL approach feeding incorrect location information to the APR approach, which would subsequently attempt to repair a non-faulty location in the input program. This work, therefore, assumes perfect bug localization on the line level, meaning that the models are always provided with the correct buggy line number. Following the requirement of equal information between models and humans, human participants are also provided with the correct buggy line number. As described above, the main benefit of assuming perfect bug localization is eliminating the possibility of models or humans trying to repair a non-faulty location in the input program.

An added benefit of perfect localization is that the attention data gathered during bug fixing relates to the same buggy location in the input program, as opposed to imperfect localization, where the attention data might relate to an incorrectly-identified location in the input program. Even though the attention vectors are distributions of scores over the entire input program, the fault location given to the APR approach matters: For example, SequenceR produces candidate patches by token-wise prediction of a replacement for the buggy line [11]. Attention vectors are extracted for each token of the output sequence, over the input sequence. By assuming perfect fault localization, it is ensured that SequenceR actually predicts a replacement for the correct buggy line and consequently, that the attention vectors are for tokens of this replacement token sequence.

Finally, assuming perfect localization to eliminate uncertainty stemming from inadequate local-

ization performance is in keeping with the current state of APR approach evaluations [39, 23, 33].

3.2 Human Attention Collection

```
package java_programs;
import java.util.*;

public class FIND_FIRST_IN_SORTED {

    public static int find_first_in_sorted(int[] arr, int x) {
        int lo = 0;
        int hi = arr.length;

        while (lo <= hi) {
            int mid = (lo + hi) / 2;

            if (x == arr[mid] && (mid == 0 || x != arr[mid-1])) {
                return mid;
            } else if (x <= arr[mid]) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }

        return -1;
    }

}
```

Figure 3.2: Example human attention map.

The concept of attention is inherently harder to define for human participants than it is for the neural models, where the attention can be extracted from the reference implementations in the form of a vector of scores. This work adapts the approximation of human attention proposed by Paltenghi and Pradel [24]. Closely related to this work, they compare attention extracted from two approaches for code summary to that of human participants in a method-naming task. They define a concept of human attention based on a specialized data collection interface called the *Human Reasoning Recorder*, where human participants are tasked with naming a method based on its source code. Initially, the tokens of the method are blurred, while participants can deblur tokens by hovering over or clicking on them. The participants' interactions with the tokens are recorded and human attention is then calculated based on the duration that individual tokens were unblurred.

Figure 3.3 demonstrates this idea with a screenshot of the interface of the Human Reasoning Recorder: The interface unblurs tokens at the position of the mouse cursor and its surroundings and blurs the rest of the tokens. In the background, it collects visibility times for each token, from which the human attention vector can be computed. Figure 3.2 shows an example human of a human attention map, visualized as a heat map over the program tokens.

Specifically, Paltenghi and Pradel [24] introduce the following definition of a human attention vector: "The human attention vector is [defined as] $\vec{h} = (h_1, h_2, ..., h_n)$, where h_i is the total time that the token at position i has been visible according to [the collected interaction events]." [24] This work adapts this definition to fit the repair task defined in Section 3.1. A key challenge in this is the fact that users have to change the input program in order to fix bugs, thereby changing the token sequence of the program. To account for these token changes, this work extends the definition by excluding tokens that have been edited from the visibility tracking. Subsection 3.2.2

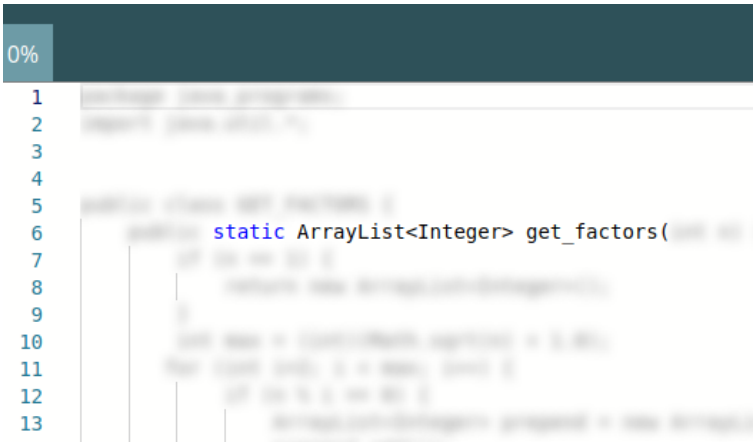


Figure 3.3: Unblurring mechanism.

explains this procedure and the collection of attention on the buggy line in detail.

The concept of collecting human attention data by selectively blurring the input is not limited to code-related tasks. For example, Das et al. [12] collect human attention during a visual question answering task by having human participants selectively deblur relevant parts of the input image.

3.2.1 Data Collection

To enable the collection of human attention data during code repair tasks, we present the *Edit Attention Recorder*, abbreviated EAR. EAR is an online data collection platform that includes a fully-featured code editor as well as a framework for recording user interaction events, from which human attention can be computed.

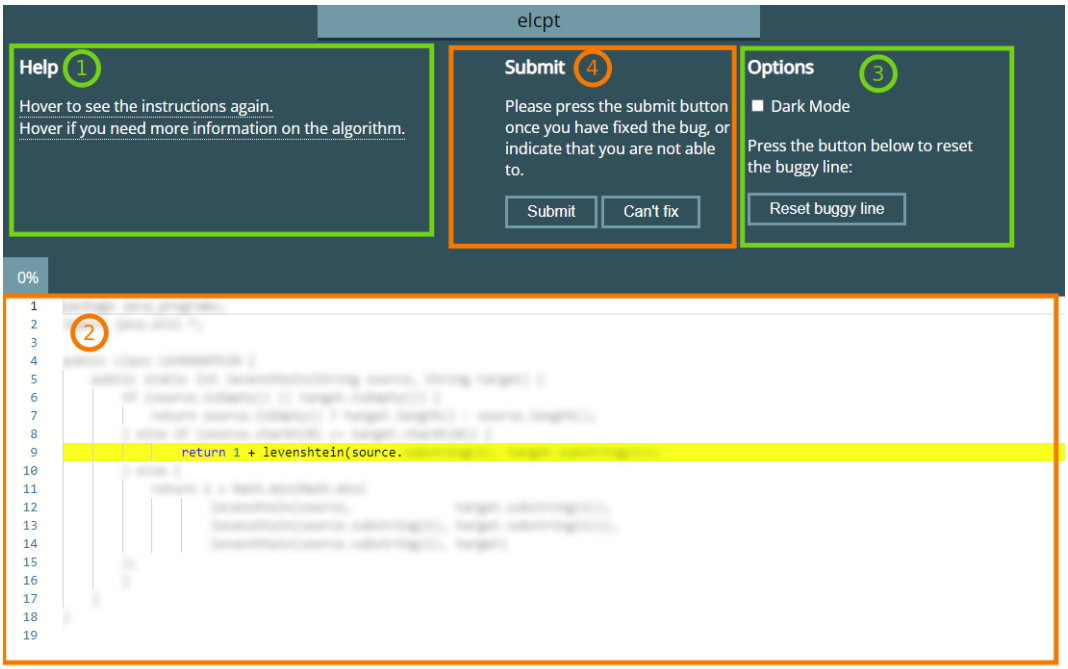


Figure 3.4: EAR user interface

Figure 3.4 shows the user interface of EAR, configured for the repair task considered in this work. The interface is divided into four main areas serving the following purposes:

1. *Help*: The help area serves to provide human participants with general instructions about the usage of the platform, as well as specific information on the task at hand, such as a general description of the underlying algorithm and input/output examples.

2. Main editing area: This area contains the modified code editor where the program tokens are initially blurred but can be deblurred by moving either the mouse pointer or the editor cursor over a token. The interface deblurs up to n tokens around the current position of either the mouse pointer or the editor's cursor. Tokens do not unblur across line boundaries, as this would result in the deblurring of the last, resp. the first tokens of the previous or the following line.

   The buggy line is highlighted in color, which serves as a graphical means of providing participants with the buggy line number. Users are able to edit the buggy line, and vice-versa, unable to edit anywhere outside of the buggy line. The insertion of additional line breaks in the buggy line is also not allowed.

3. *Options*: The Options section allows participants to switch to "Dark Mode" if they prefer this optical style of the code editor, and to reset the buggy line back to its initial state.

4. *Submit*: The Submit section allows human participants to either indicate that they have successfully fixed the bug, or that they do not know how to fix the bug. When the latter button is pressed, the reasons for their inability to fix the bug are asked as part of the exit survey.

The additional information on the algorithm provided to human participants by the help section is not provided to the APR approaches, as both selected approaches do not take such information as an input. Even though this does constitute a violation of the policy of providing humans and approaches with the same information, a pilot study conducted as part of this work found the repair task to be very hard for human participants to complete without this information, or that human participants would start internet searches for this information. The goal of providing this information outright is to minimize such searches outside of the application and to thereby maximize the interaction time and resulting attention collection.

As mentioned above, participants are able to edit the buggy line, which can change the corresponding token sequence. The implications of these changes for attention tracking are elaborated upon in the following.

3.2.2 Attention Tracking on the Buggy Line

A key challenge of the code repair task is the requirement for human participants to be able to change the source code of the input program. Source code changes also change the token sequence representing the program, which raises the question of how changed tokens should be taken into account with regard to human attention, which is defined on the basis of token visibility times.

Participants modify the source code of the input programs on a character level, meaning that individual edits may either modify, insert, or remove tokens. Also, the position of tokens after the location of the edit changes if the edit changes the length of the edited area.

As comparability with the attention data collected from the neural models is a primary concern of this work, the definition of attention on the buggy line for human participants is based on the semantics of the attention data collected from the models. SequenceR [11]'s attention scores are in the form of vectors of attention scores over the input token sequence, for each token of the output sequence. Recoder [39] employs a self-attention mechanism that provides attention scores over the so-called AST traversal sequence, which contains nodes of an abstract syntax tree constructed from the bug and surrounding context, in the order of tree traversal. Since the sequence is constructed from the bug and context, the resulting attention vector also represents scores over a representation of the input program.

Intuitively, this implies that the human attention scores should also be restricted to only input tokens and not tokens that might be considered part of the output. Therefore, this work defines the following: EAR considers tokens as *input* tokens as long as they have not been edited by the user, except for changes in token position. As mentioned above, edits are on the character level and can influence the token sequence in different ways. We introduce the following algorithm to map character-level changes in the editor to token-level changes:

Algorithm 1: Change processing algorithm

```
1  ProcessChanges (T, c)
       inputs : A sequence of tokens T, a character-level change c
       output: An updated sequence of tokens T'
2      T' ← ();
3      foreach t ∈ T do
4          if t.range intersects c.range then
5              t.excluded ← true;
6          else if t.column > c.range.end then
7              δ ← |c.text| − |c.range|;
8              t.column ← t.column + δ;
9          T' ← T' ⊕ (t);
10     end
11     return T'
```

Algorithm 1 takes as input a character-level change in the editor, as well as a sequence of tokens. A character-level change c is an object with an attribute *text* that contains the new sequence of characters at the edited location and an attribute *range* that denotes the range in the editor that was edited. The *range* attribute also has an attribute *end*, which denotes the end column of the change range. The individual tokens $t \in T$ are objects with an attribute *column* that denotes the column on the buggy line that a token begins at, an attribute *range* that denotes the range of the token, as well as an attribute *excluded* which determines whether the token is included in visibility tracking. Since edits can only take place on the buggy line, all changes and tokens have the same range by construction. The $\oplus$ operator denotes sequence concatenation.

Given these inputs, Algorithm 1 returns an updated set of tokens, in which tokens that are

determined to have been edited are marked as excluded, and the position of tokens after the change is updated. Detecting whether a token needs to be excluded is done by the condition on line 4 of the algorithm, namely by checking whether the token's range intersects that of the change. If this is the case, the token is excluded due to having been modified by the user, and no longer being an input token in the sense of the above definition.

Token positions are updated if the token is located after the change, if this is the case (condition on line 6 of the algorithm), the change δ in the token's column is calculated by taking the difference between the length of text contained in the change, and the length of the change range and adding that to the token's column (line 8 and 9).

EAR does not collect visibility times for tokens after they have been excluded according to the above algorithm, to reflect that they are no longer considered as input tokens. Attention scores are calculated from the unblurring events which are tracked by EAR, the data model of the collected events is described in the following in Subsection 3.2.3.

3.2.3 Data Model

EAR records several types of events as the user interacts with the application. Human attention vectors, as well as other statistics, are computed from the data provided by these events. This subsection defines the different types of events and how human attention vectors are computed from them. Table 3.1 gives an overview of the events collected by EAR and a short description of what information each event contains.

Name	**Tracked Information**
Unblur Event	Unblurring and blurring of tokens
Mouse Movement Event	Movement of the mouse pointer
Edit Event	Changes made by the user
Hint Hover Event	Usage of the algorithm hint
Paste Event	Pasting of content in the editor

Table 3.1: Collected Events.

3.2.3.1 Unblur Event

Unblur events are the main foundation of human attention calculation. EAR records these events when user interaction causes tokens to be deblurred. Due to its event-driven architecture, tokens that are unblurred as part of an unblur event stay unblurred until the next event occurs, when the tokens contained therein become unblurred. Furthermore, an unblur event with no tokens visible is also recorded when the user is inactive for a configurable time, to prevent attention data from being skewed by the mouse pointer's position when the user goes inactive.

An unblur event is a tuple $e_{unblur} = (t, IDX_{vis}, s)$ where t is the UNIX timestamp when the event occurred, IDX_{vis} are the token sequence indices of the tokens that are visible as a result of the event, and s denotes the source of the event, such as mouse hovering or cursor movement. Both movements of the mouse pointer and the editor cursor result in an unblur event, but the source of the event is tracked to enrich the data available for analysis. Following the earlier definition of

the human attention vector, the human attention scores are calculated from the events as defined above by the following algorithm:

Algorithm 2: Human Attention Calculation

CalcAttention (E, T, t_s)
  inputs : A sequence of unblur events E, sequence of tokens T, start time t_s
  output: The human attention vector $\vec{h}$
  $\vec{h} \leftarrow (0, 0, .., 0)^\intercal$ such that $|\vec{h}| = |T|$;
  $T_{prev} \leftarrow \emptyset$;
  $t_{prev} \leftarrow t_s$;
  foreach $e = (t, IDX_{vis}, s) \in E$ **do**
    $\delta \leftarrow t - t_{prev}$;
    foreach $i \in IDX_{vis}$ **do**
      $\vec{h}(i) \leftarrow \vec{h}(i) + \delta$;
    end
    $t_{prev} \leftarrow t$;
    $T_{prev} \leftarrow IDX_{vis}$;
  end
  return $\vec{h}$

Algorithm 2 calculates the total visibility times by iterating over all events and calculating the elapsed time since the previous event, which is then added to the cumulative visibility scores in the human attention vector. In the algorithm, $\vec{h}(i)$ denotes the i-th element of the human attention vector. Note that in the implementation, ending the task requires exiting the editor area with the mouse pointer, which results in an unblur event with $IDX_{vis} = \emptyset$ being recorded. Thus, no explicit handling of the last event's tokens is needed, since it is guaranteed to have none.

3.2.3.2 Mouse Movement Event

Mouse movement events track movements of the mouse and are mainly kept as a sanity check and backup for the unblur events, as they occur on a lower technical level. A mouse movement event is a tuple $e_{mouse} = (t, x, y)$ where t is the UNIX timestamp when the event occurred, x is the line in the editor that the mouse pointer is at, and y is the column that the mouse pointer is at.

3.2.3.3 Edit Event

Edit events track edits that the user makes to the source code. They are a tuple $e_{edit} = (t, c)$ where t is the UNIX timestamp of when the event occurred, and c is a set of representations of changes made to the program code. In the implementation of EAR, the elements of *changes* conform to the `IModelContentChange` interface defined by the Monaco Editor API[1].

[1] `https://microsoft.github.io/monaco-editor/api/interfaces/monaco.editor.IModelContentChange.html`

3.2.3.4 Hint Hover Event

Hint hover events track usage of the hint overlay that provides information on the task such as a description of the algorithm, input/output examples, etc. A hint hover event is a tuple $e_{hint} = (t, a)$ where $a \in \{enter, leave\}$. t denotes the UNIX timestamp of when the event occurred, a denotes whether the event corresponds to the hint starting to be displayed, or ceasing to be displayed.

3.2.3.5 Paste Event

Paste events are used to additionally track edits made by the user, specifically those that involve pasting of previously copied code. They are tuples $e_{paste} = (t, r, m)$ where t is the UNIX timestamp of when the event occurred, r is the range in the editor affected by the paste operation, and m is a sequence of characters corresponding to the entire content model of the editor after the change.

3.2.3.6 Post Task Survey

In addition to these events, EAR also queries the user for additional information after the completion of each task. The post-task survey consists of a simple HTML form, which is easily replaceable based on the information that needs to be collected from the human participants. All information entered in the form is recorded as part of the records for each user-task pair.

3.3 Bug Dataset Selection

The repair task, as defined in 3.1, requires a source of single-file, single-line bugs. This section discusses the advantages and disadvantages of several bug datasets and explains the rationale behind the choice of Quixbugs as the bug dataset for the experimental evaluation. Following a preliminary inspection and the choice of Java as the target language for the APR approaches, the selected candidate datasets are the top three datasets listed in Table 2.1, as well as the training and evaluation datasets used by the selected APR approaches. To ascertain the human fixability and overall suitability of the datasets, we conduct a manual inspection of the candidate datasets, with the primary goal being high human fixability, to fulfill the requirements listed in Section 3.1.

For Defects4J, the inspection finds that most of the bugs contained within the dataset span multiple lines and that extensive domain knowledge is required to conduct successful repairs, as a result of the dataset being mined from real-life open-source projects. The inspection finds a subset of Defects4J only including single-line bugs to be more tractable than the general dataset, with the bugs often consisting of incorrect variable usages, copy-paste errors, or incorrect operator usage. However, these bugs still require extensive context information to fix, making Defects4J unsuitable for the human study.

Regarding ManySStuBs, the inspection finds that the second (the SStuBs) subset is moderately fixable for humans, with 3/10 of inspected bugs being fixable without any additional context. However, the majority of the bugs still require extensive context and domain knowledge to be fixed, due to the bugs being mined from open-source repositories. This makes ManySStuBs equally unsuitable for the human study.

Another possible source of data that fits the set criteria are the training and evaluation datasets of the two APR approaches selected for inclusion in this work, SequenceR [11] and Recoder [39]. For these, the manual inspection finds that they are also very challenging for human developers, as the bugs contained within require extensive context and domain knowledge to fix. Due to their intended use as bulk datasets for neural model training, they are also less sophisticated than the other datasets, for example containing some rows where the buggy and fixed code is the same. Additionally, they do not feature automated build and test environments, in contrast to the other datasets considered for inclusion. Finally, using these datasets would unfairly favor the corresponding approach, as it would already have "seen" the data.

In contrast to the previous datasets, Quixbugs consists of bugs that are explicitly designed as a challenge for human developers [22]. As such, manual inspection of the dataset finds that the bugs contained in Quixbugs are much more tractable for human developers, as fixing individual bugs does not require any domain knowledge beyond an understanding of what the algorithm at hand is supposed to do. It also features an automated build and test suite, in contrast to the training and evaluation datasets, streamlining the assessment of patches generated by humans and approaches. Finally, the increasing adoption of Quixbugs as a benchmark for automatic program repair approaches means that the results of the human study are directly comparable to other results and that they may serve as a baseline of human performance for existing and future work in the field.

For these reasons, this work selects Quixbugs as the source of bugs for the experimental evaluation. Specifically, we create the human study dataset by selecting a 16-file subset of Quixbugs. The limitation to 16 files is due to practical constraints of the human study: Selecting the entire dataset would have incurred a risk of not enough data being collected for meaningful analysis due to an insufficient number of participants per sample.

3.4 Language Selection

In the context of this work, the target language refers to the programming language that is used for the concrete implementation of the repair task, meaning that the buggy programs given to approaches and humans are written in that language. This section explains the choice of Java as the target language used in the experimental evaluation, based on the commonness of Java as a language both for APR approaches and bug datasets.

As listed in Table 3.2, the most frequent common language among the APR approaches considered for inclusion in this work is Java.

As the previous section states, this work selects Quixbugs as the bug dataset for the experimental evaluation. As Table 2.1 shows, that dataset, as well as the majority of other eligible datasets, are in Java as well.

Based on these facts, this work selects Java as the target language for the experimental evaluation, as a direct consequence of the target languages of both the bug dataset and the selected APR approaches.

3.5 EAR Implementation

This section documents the implementation of EAR, beginning with the overall architecture before highlighting the specifics of the central editor component, being built on top of the Monaco editor[2], followed by the selection of the task dataset samples from the full QuixBugs dataset. The section closes with a list of experimental features that were implemented, but not used in the experimental evaluation, but may be of value for future research conducted using EAR.

3.5.1 Architecture

The basic architecture of EAR is a standard three-layer web application, split into client, server, and database. Figure 3.5 visualizes this architecture and lists the concrete technology used in the implementation of each component. The Node.js runtime [3] is used to run the server component of the application.

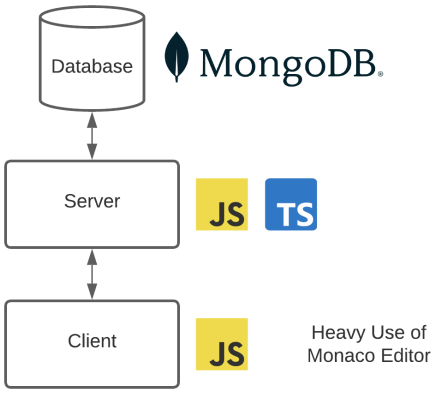


Figure 3.5: EAR Architecture.

3.5.2 Editor Component

The code editor is a central part of the client-side component of EAR, as all events defined in Subsection 3.2.3 are generated as a result of user interaction with the editor, and are also recorded by the editor component. The foundation of the editor component is an instance of the Monaco editor[4], a code editor implemented in JavaScript that also powers Microsoft's VS Code IDE. This work builds the following features on top of the editor:

- Blurring control: By default, the entire editor content is blurred but can be deblurred by movements of the mouse pointer or the editor cursor. For the experimental evaluation, the blurring mechanism is configured to unblur 3 tokens before and after the targeted token, and for the blurring not to go beyond line boundaries.

[2] https://microsoft.github.io/monaco-editor/
[3] https://nodejs.org/en/
[4] https://microsoft.github.io/monaco-editor/

- Editing restriction: This restricts edits anywhere in the editor except for on the buggy line and prevents the insertion of line breaks on the buggy line.
- Highlighting of the buggy line: The editor highlights the buggy line with a colored background, serving as an optical means of providing the buggy line number to human participants.
- Event recording: The client application of EAR registers several callbacks with the Monaco API, thereby recording the events as defined in 3.2.3.

In addition to the above features, which represent the feature set that is used for the experimental evaluation, this work introduces a number of experimental features for EAR. These features remain present in the source code and can be enabled with simple flags, to serve as a basis for future work and extensions of EAR. The experimental features are:

- Click to unblur: Enables the permanent deblurring or blurring of tokens by clicking on them, and includes the recording of a corresponding click unblur event.
- Buggy line unblurring: Permanently unblurs the buggy line to make it easier for users to read its tokens.
- Javadoc hover: Introduces a hover display showing Javadoc documentation when hovering over identifiers such as class names that belong to the Java standard library.
- Search bar: Introduces a search bar in the upper part of the application, which allows the tracking of search queries made via that search bar.

3.5.3 Task Dataset

The task dataset is a subset of the Quixbugs benchmark dataset [22]. It consists of a total of 16 buggy programs, eight of which are fixable by one of the models, and eight of which are not. Appendix B lists the exact files that make up the dataset. The application serves two fixable and two non-fixable samples to each human participant.

3.6 Human Study

This section describes the candidate recruitment process, as well as the data collected from human participants in the human study. Candidates are recruited from both academia and industry: The participants from academia consist of the members of the research group that this work is being undertaken at, as well as students taking a Master's level course on program analysis. The industry participants consist of freelance software developers, as well as members of the software engineering and working student divisions of a software firm. EAR enables participants to take part in the study by independently visiting the online interface and following the instructions from there, allowing for a decentralized running of the study. Section 4.1 gives a detailed overview of the participant numbers, their qualifications, and other demographic information.

The data collected as part of the human study consists of the events as defined in Subsection 3.2.3, as well as additional data collected via the post-task survey, and demographic data collected via an online form after the entire experiment is completed.

In the post-task survey, participants are asked to rate the difficulty of the task they just completed. If they indicate that they are unable to fix the bug, the survey also asks why they were unable to. Finally, the survey also asks participants for any aids potentially used in fixing the bug, such as internet searches, execution on paper, etc.

The post-experiment survey consists of a Google Form[5], where participants are asked questions about their demographic background, such as age, gender, and the highest level of education, as well as questions on their programming experience, especially in relation to debugging. Appendix A gives a full list of the questions and answer options in the post-experiment survey.

3.7 APR Approach Selection

The APR approaches considered in this work are chosen from a list of existing APR approaches based on a set of criteria, which are explained in this section. For gathering candidate approaches, a literature review of relevant literature on neural automatic program repair was conducted, starting with an initially provided reading list. More publications on APR approaches were collected by the following means:

- Snowballing from the references of already considered papers
- Search for papers on the *dblp* computer science bibliography[6], using the keywords *automatic program repair*, *bug fixing*, *APR*, *error repair*, and permutations thereof.
- Search for related works using the ConnectedPapers[7] tool
- Papers introduced in university courses on the subject

Following the literature review, a preliminary list of 16 APR approaches was compiled. Approaches that do not feature attention mechanisms or do not have a (publicly) available reference implementation are excluded from this list, resulting in a shortlist of seven candidate approaches. Table 3.2 lists these approaches, giving for each approach its name or authors, the architecture of the underlying neural model, the reported or experimentally determined performance on the Quixbugs dataset, and their target language(s).

Due to the choice of Java as the target language for the repair task and their performance, we choose SequenceR and Recoder as the approaches for the experimental evaluation. Both approaches have their reference implementations available on GitHub[8][9]. We modify the reference implementations to facilitate attention extraction, which the next section elaborates upon.

[5] `https://docs.google.com/forms/`
[6] `https://dblp.org/`
[7] `https://www.connectedpapers.com/`
[8] `https://github.com/pkuzqh/Recoder`
[9] `https://github.com/KTH/sequencer`

Approach	Architecture	Qxb. Perf.	Language
SequenceR [11]	Encoder/Decoder RNN	18.8%	Java
Recoder [39]	Transformer	42.5%	Java
CoCoNuT [23]	CNN Ensemble	32.5%	Java, Python, C
Hoppity [13]	GNN + LSTM	N/A	JavaScript
DeepFix [16]	Encoder/Decoder RNN	N/A	C
Tufano et al. [33]	Encoder/Decoder RNN	N/A	Java
TRACER [1]	Encoder/Decoder RNN	N/A	C

Table 3.2: Shortlist of APR Approaches.

CoCoNuT was initially also slated for inclusion, however, the pre-trained models achieving the reported performance are missing from the reference implementation's public repository. Due to the large size and the great number of models required for CoCoNuT's ensemble approach, training new models would be beyond the scope of this work in terms of available computational resources. The following section describes how we instrument and modify the approaches for the purposes of the experimental evaluation, and also gives a brief introduction of each approach.

3.8 Attention Extraction

This section briefly introduces the two selected APR approaches, before documenting any modifications done to the reference implementations of the selected approaches, starting with SequenceR. This includes instrumentation to extract the attention vectors from the neural models underlying the approaches, as well as minor functional changes that enable comparison with the developers. The section closes by indicating what hardware was used for the model evaluations.

3.8.1 SequenceR

SequenceR is an APR approach that uses a sequence-to-sequence encoder/decoder with attention architecture to produce candidate fixes [11]. It works by taking an abstracted form of the buggy program, as well as the buggy line number as an input. The underlying model predicts a replacement for the buggy line from the token sequence of the abstracted program. The approach uses beam search to search for multiple viable fix candidates, which leads to the end result being a list of patch candidates.

The implementation of SequenceR uses the OpenNMT library [21], which already includes a setting to output the attention weights. In its original form, this attention debug feature only outputs the attention corresponding to the most likely prediction, as well as the contents of that prediction. To enable more detailed analysis of the attention weights, for example for looking at relationships between the attention for predictions that results in plausible fixes and those that do not, we extend this feature. Instead of outputting the attention vector corresponding to the most likely prediction, we extend the feature to also save the entirety of the predictions and corresponding attention vectors to a file. In addition, we also extend the debug feature to save both the "regular" and the copy attention, to enable an analysis of both. As this is just an extension of an existing

debug feature, it does not change the behavior of SequenceR, except for a negligible increase in processing time.

The second modification to SequenceR's reference implementation concerns the so-called buggy context abstraction step [11]: The *abstract buggy context* is the sequence of tokens corresponding to the method containing the buggy line, select parts of the containing class, with additional marker tokens inserted around the buggy line. SequenceR truncates this sequence to a configurable limit.

As the files contained in Quixbugs are relatively short, they fit below the truncation limit set by SequenceR's reference implementation. However, parts of the buggy context abstraction step included in the reference implementation of SequenceR actually change the syntax of the input program. For example, an `else if () {...}` statement will be replaced by a `else { if (...) {...}}` statement. Even though the replacement preserves the semantics of the program, it changes the token sequence, which impacts the comparability to the human participants, as they see the original token sequence.

To enable a direct comparison, we replace the relevant parts of the buggy abstraction step with a custom part that simply adds the marker tokens to the buggy line, thereby skipping the part that rewrites the code. As the files in Quixbugs are all short enough to fall below the 1000-token truncation limit of the reference implementation, this replacement only changes the behavior related to the rewriting of some statements. In the modified form, SequenceR generates plausible fixes for only one less file than in the original form, indicating that this change does not drastically worsen its performance.

The third and final modification to SequenceR's reference implementation concerns the post-processing step: SequenceR has a copy mechanism that enables it to copy tokens from the input in place of unknown tokens [11]. In the patch post-processing, any patches that still contain unknown tokens are discarded. To correctly map between the attention vectors for all generated predictions and the resulting patches after post-processing, we add instrumentation code to the latter step to keep track of the patches that are kept. This is needed to correctly assign attention vectors to patches in the later analysis. This modification does not change the functional behavior of SequenceR, as it only reads values and writes them to a file in certain cases.

We further post-process the extracted attention vectors. In encoder/decoder attention, there is one attention vector per output token. To obtain a single vector over the input tokens, we take the mean over all attention vectors for a single prediction, effectively turning a matrix of shape $(l_{out} \times l_{in})$ into a single vector of length l_{in}, where l_{out} is the length of the output sequence, and l_{in} is the length of the input sequence. Figure 3.6 shows an example attention map extracted from SequnceR using this process, visualized as a heat map.

3.8.2 Recoder

Recoder is an APR approach that produces a sequence of edits instead of fixed code [39]. It generally follows an encoder/decoder architecture, however, with the additional feature that the decoder is syntax-guided, meaning that it follows the target language's syntax when generating fix candidates.

Recoder consists of four major components: the code reader, the AST reader, the tree path

```java
package java_programs;
import java.util.*;

public class FIND_FIRST_IN_SORTED {

    public static int find_first_in_sorted(int[] arr, int x) {
        int lo = 0;
        int hi = arr.length;

        while (lo <= hi) {
            int mid = (lo + hi) / 2;

            if (x == arr[mid] && (mid == 0 || x != arr[mid-1])) {
                return mid;
            } else if (x <= arr[mid]) {
                hi = mid;
            } else {
                lo = mid + 1;
            }
        }

        return -1;
    }

}
```

Figure 3.6: Example SequenceR attention map.

reader, and the edit decoder [39]. Figure 3.7 gives an overview of these components, their inputs, and the neural network layers that they contain. The purpose of the code reader component of Recoder is to encode the bug and the surrounding context, for use by the other components. As it follows the Transformer architecture [34], it has a self-attention layer that allows the input sequence, in this case, the depth-first traversal sequence of the AST corresponding to the buggy method, to pay attention to itself. We select the self-attention layer of the top-most Transformer block in the code reader for extraction, as it is the only attention layer in Recoder that exclusively pays attention to parts of the input program. The code reader self-attention layer is highlighted in yellow in Figure 3.7.

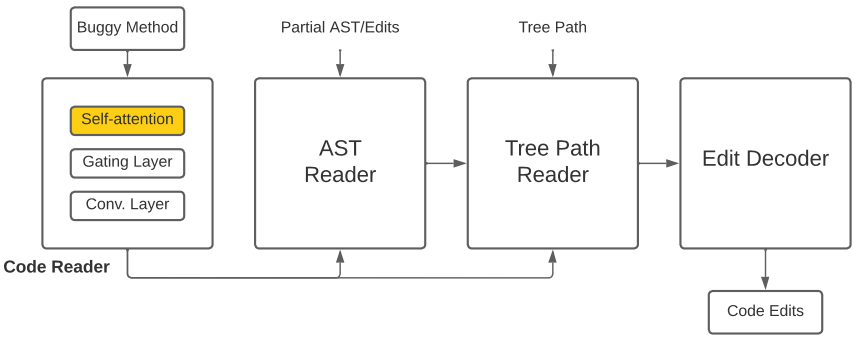


Figure 3.7: Overview of Recoder. Based on Figure 7 in [39].

We instrument the reference implementation of Recoder to write the attention tensors to disk. As the self-attention layer follows the architecture of the multi-head attention mechanism [34], which means that there are multiple attention "heads" that extract different features. Due to this, the extracted tensors contain multiple attention matrices, one for each attention head.

To obtain a single vector over the input sequence, we average them across the head axis to obtain one single matrix of shape $(l_{seq} \times l_{seq})$, where l_{seq} is the length of the input sequence in

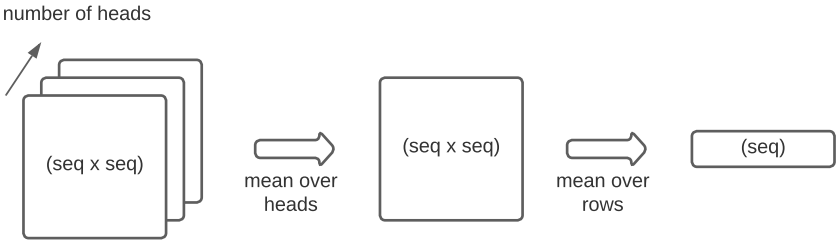


Figure 3.8: Attention post-processing for Recoder.

tokens. In the resulting matrix, the rows contain attention scores for the elements of the input sequence. Figure 3.8 illustrates this process. Analogous to the process for SequenceR, we average over the rows to obtain one single vector of weights that has the length of the input sequence. Finally, we truncate the resulting vector using the padding mask to drop any scores relating to the padding.

3.8.2.1 AST-to-Token Mapping

The resulting vector represents the (average) attention that Recoder pays to parts of the input sequence, which is the traversal sequence of the AST of the buggy method. However, the attention extracted from SequenceR, and that collected from the humans, is over the sequence of the tokens of the input program. We develop the following mapping strategy to convert Recoder's AST attention to an attention vector over the input tokens:

To enable the distribution of attention from the AST nodes, each AST node has to be assigned a token range that represents the tokens underlying the programming language construct that the node represents. To this end, we manually assign token ranges to nodes of the ASTs of the programs in our subset of Quixbugs, based on information on token ranges collected from another Java parser[10]. Figure 3.9 shows an example of this process, where token indices are assigned to AST nodes following Recoder's AST syntax. The brackets next to each AST node in the figure indicate the corresponding start and end tokens of the node. The need for manually assigning these ranges stems from the fact that the parser that Recoder uses as part of its pre-processing does not offer token range information for the AST nodes it outputs. For this purpose, we also add instrumentation code to Recoder's reference implementation that writes the ASTs going into the model to a file.

Based on the ASTs with added token range information, we distribute the attention as follows: For each AST node in the AST traversal sequence, we assign each of the tokens in its range an equal share of the AST node's attention score. Tokens accumulate the attention that they receive, meaning that they can receive attention shares from multiple AST nodes.

We also developed two other strategies for distributing attention based on the assigned tokens. The first consists of defining the attention of an AST node as the mean of the attention that its corresponding tokens receive, with this strategy allowing the conversion of a token attention vector

[10] `http://javaparser.org/`

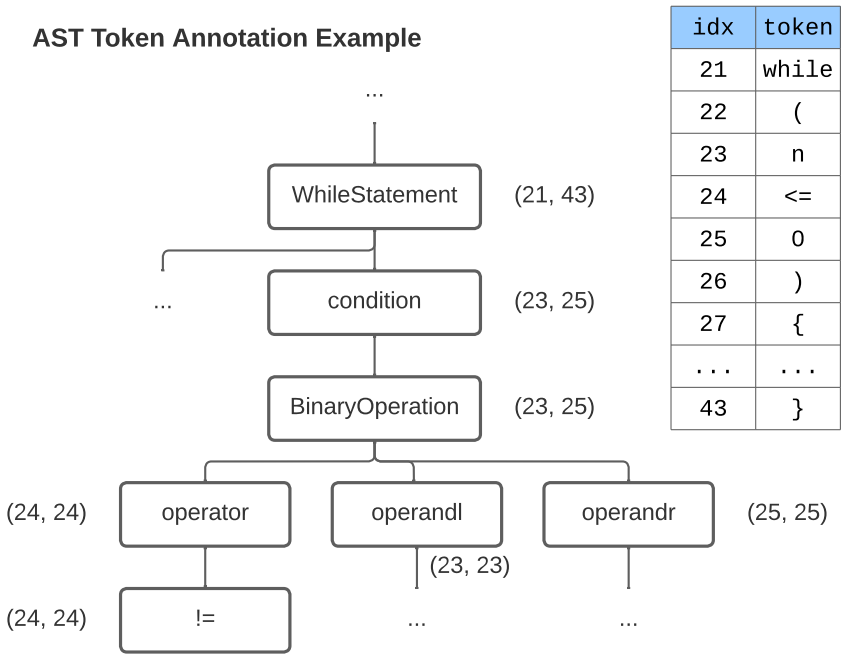


Figure 3.9: AST Annotation example.

to an AST traversal attention vector. The second strategy assigns each terminal node in the AST the attention score of the corresponding token and then propagates the scores towards the root of the tree by defining the attention of each non-terminal AST node as an aggregate of the attention of its child nodes, with the maximum function yielding the best results in preserving human-human correlation after conversion to AST traversal attention vectors. However, these additional strategies do not yield statistically significant results in the sense that correlation analysis between human and machine vectors frequently yields $p >= 0.05$. For this reason, this work only gives the results obtained using the AST-to-token strategy first described.

3.8.3 System Setup

	System 1	System 2
Operating System	Ubuntu 18.04.6 LTS	Linux Mint 20.3
Linux Kernel	4.15.0-167-generic	5.4.0-109-generic
Architecture	x86-64	x86-64
CPU	Intel Xeon Silver 4214 @ 2.20GHz	Intel Core i7-7700HQ @ 2.80GHz
GPU	2x Nvidia Tesla T4, 16 GB RAM	GeForce GTX1070 Mobile
RAM	32 GB	32 GB
CUDA Version	11.2	11.4

Table 3.3: System Setups used in the evaluation.

Table 3.3 gives the setup of the two systems used in the experimental evaluation. System 1 was used to run the timing experiments, as well as anything related to Recoder. System 2 was used

to run SequenceR except for the timing evaluation, as well as all plausibility evaluations, and the overall data analysis. EAR was deployed on the free tier of the cloud application service Heroku[11].

[11] https://www.heroku.com/what

4 Results

This chapter presents the results of the experimental evaluation conducted for this work. It begins by listing some basic statistics about the human dataset gathered as part of the 27-participant human study, before setting out to answer the research questions put forward in Chapter 1 in Sections 4.2 to 4.7.

For the comparative analysis between developers and APR approaches, we also create a dataset consisting of the extracted attention of SequenceR and Recoder. This dataset contains the "regular" and copy attention extracted from SequenceR, as well as the Transformer multi-head self-attention extracted from Recoder. We make the machine attention dataset available as part of the GitHub repository accompanying this thesis.

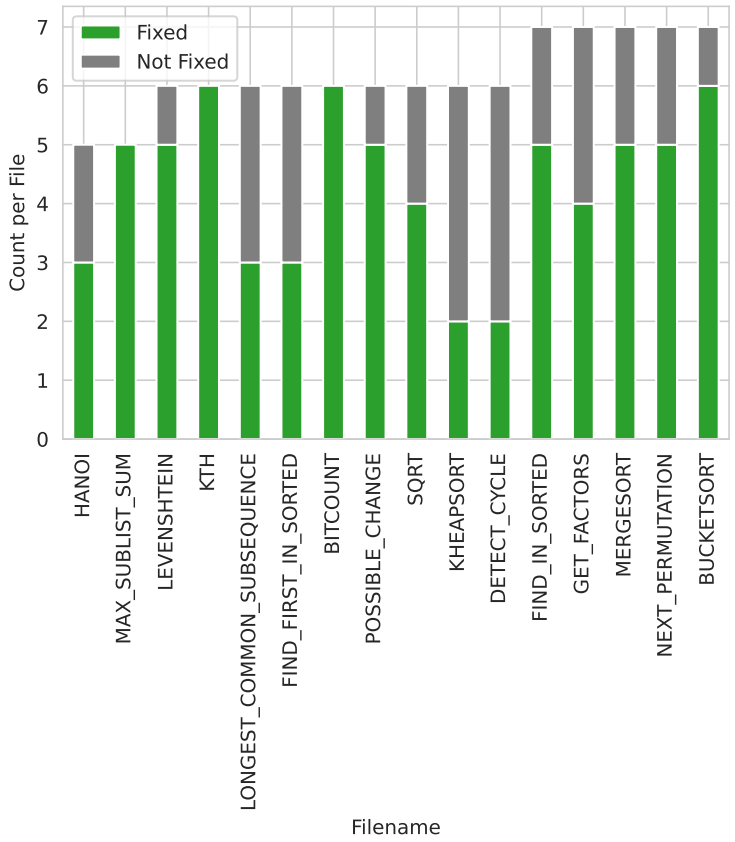


Figure 4.1: Human attempts per file.

4.1 Human Program Repair Dataset

The human data consists of attention and behavioral data gathered from 27 human developers as part of the human study. Participants worked on between one and five buggy files, for a total of 99 valid attempts. For each attempt, the dataset contains the data listed in Subsection 3.2.1.

Figure 4.1 visualizes the human attempts. It counts the number of fix attempts on the y-axis, grouped by the filename on the x-axis. The green part on each bar indicates the number of attempts that resulted in a fix, the grey part indicates the unsuccessful attempts. The repair performance varies between the different human developers, from two not being able to fix any of the files presented to them, to one being able to fix all five files. The average number of fixes per developer is 2.56.

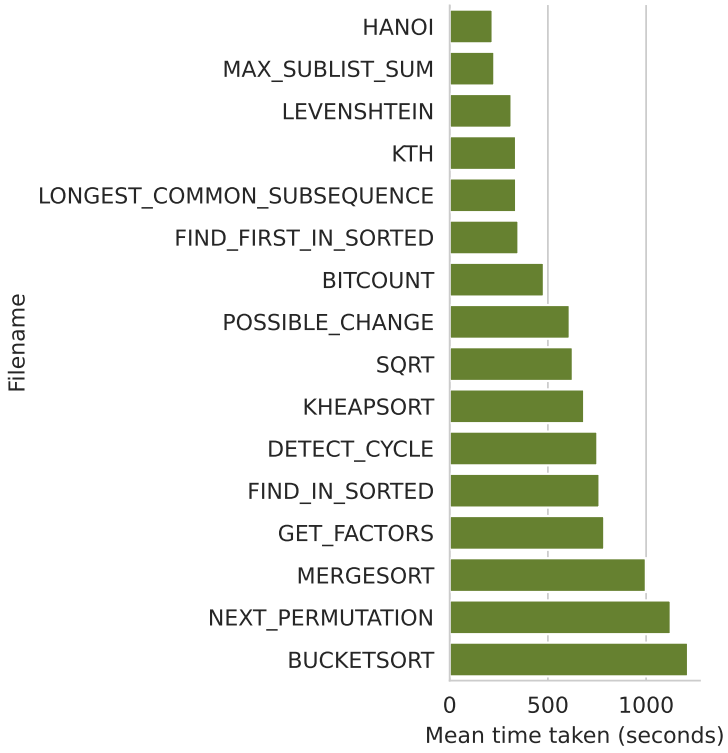


Figure 4.2: Mean time taken per file.

Figure 4.2 shows the mean time taken by the human developers for each file. The values range from 219 seconds for HANOI to 1214 seconds for BUCKETSORT on average.

Almost all of the participating developers have at least one academic degree, with 18 having a Bachelor's, and eight having a Master's degree. One participant only reported completed secondary education as their highest education level. Regarding occupation, exactly half of the participants report that they work as a software engineer or as a project lead, while the other half give their occupation as researcher or student. However, only 23.1% of the participants report their employment status as being an employee, with the rest indicating that they are students of some capacity, which includes working students. This is likely the result of the prevalence of working students in the industry, i.e., students that work freelance or at a company next to their studies. Almost all

participants (92.3%) report that they have taken a course on algorithms in the past.

The full human dataset including the demographics survey is available in the GitHub repository accompanying this work, including documentation on what data is available in what format.

4.2 RQ 1: Repair Performance: APR Approaches vs. Developers

This section explores how the selected automatic program repair approaches, SequenceR and Recoder, compare against the group of human developers in terms of repair performance. Investigating this question yields the following insights: Firstly, the performance of developers on the Quixbugs dataset provides a baseline against which current and future APR approaches can be evaluated, especially in light of the increasing adoption of Quixbugs as a benchmark for the generalizability of approaches. Secondly, significantly higher human performance would support the general argument that approaches should aim for more human-like reasoning.

We consider several aspects of performance: The first is the capability of each model and the humans to generate at least one plausible fix for a given input file. The second is the relative rate of plausible and correct fixes out of all the patches that are generated by a model, which is comparable to the relative rate of plausible and correct fixes out of all individual attempts at fixing on the human side. The third is the time taken to generate a set of fix candidates for each input file.

Approach	Plausible	Correct	Patches/Attempts	% Plaus.
SequenceR	17	15	1395	1.21%
Recoder	10	7	1381	0.72%
Humans	69	67	99	69.69%

Table 4.1: Repair Performance in terms of attempts made, 16-file subset.

The basis for the comparison is the 16-file subset of Quixbugs [22] selected for the human study. We choose the files in the subset such that there is a balanced proportion between bugs that can be fixed by the APR approaches and bugs that cannot. As a consequence, the performance results on this subset are not fully representative (6/16 (37.5%) plausible for Sequencer, and 6/16 (37.5%) plausible for Recoder), as they, by construction, approach 50.0% of bugs fixed.

In terms of how many files out of the dataset the different groups can produce a fix for, we compare the performance on the overall dataset, respectively, on appropriate subsets: For Recoder, Zhu et al. [39] report that on the entire Quixbugs dataset, the approach is capable of generating plausible fixes for 17/40 (42.5%) bugs in the dataset, with all of these fixes also being correct. We also evaluate Recoder on the subset of Quixbugs used in this evaluation and are not able to reproduce the reference patches for two files that Recoder is reported to be capable of fixing. This results in it being able to fix 6/16 instead of 8/16 files in the subset.

The original publication [11] of SequenceR does not evaluate its performance on Quixbugs. We evaluate the approach on Quixbugs both in the form of its reference implementation, and after the instrumentation for attention extraction and comparison has been added. We exclude eight bugs from the original dataset, as they are not limited to a single line or consist of adding statements,

which is out of scope for SequenceR [11]. This evaluation finds that SequenceR's reference implementation is able to generate plausible fixes for 7/32 (21.9%) bugs. In its instrumented form, namely with the modified "buggy context abstraction" step, SequenceR is still able to generate plausible patches for 6/32 (18.8%) bugs, with at least one correct patch for 5 of these files. It is noteworthy that the modified version of SequenceR produces fixes for two fewer files compared to the unmodified version, but produces fixes for a file that it previously could not fix. This suggests that the drop in performance might be an artifact of random noise, and not indicative of a general deterioration in the capabilities of the approach. For the comparison to Recoder, we also run SequenceR with its beam size set to 100.

In contrast, Ye et al. [38] report that in an evaluation of ten non-learning based APR approaches on Quixbugs, the best-performing approach (Cardumen) is able to generate plausible patches for 5/40(12.5%) files in the dataset. When filtering for correct patches, they report that the same approach is only able to produce correct patches for 3/40 (7.5%) programs. These results indicate that both Recoder and SequenceR are capable of outperforming these approaches on Quixbugs, both in terms of plausible and correct patches.

The results of the human study yield much higher performance for the group of human developers. Overall, they are capable of producing at least one correct fix for each file in their subset, for a numerical performance of 16/16 (100%). Each file was presented to between 5 and 7 human developers, for a total of 99 fix attempts. Out of these attempts, 69 resulted in plausible fixes, for a percentage of 69.69%. Of the plausible patches, 67 (67.7%) are also correct. Valid attempts in the context of the repair performance evaluation are those attempts where the participants did not report using external repair tools, such as GitHub CoPilot[1], during the task.

If all candidate patches that are produced by the APR approaches are taken as an analog to the individual human fix attempts, the humans also outperform the approaches by a wide margin. SequenceR generates 17 plausible patches out of 1395 candidate patches, for a percentage of 1.21%. Surprisingly, Recoder performs worse than SequenceR in that category, as it only produces 10 plausible patches out of 1381 candidates (0.72%). However, over half of the plausible patches generated by SequenceR are for a single file, which suggests that this result is more of an artifact of the small dataset, and not an indicator of the overall performance of the approaches.

Another aspect of performance is the time taken for repairing a given bug. As part of the evaluation, we also time each approach to compare it against the group of developers. The timing evaluations were conducted on the hardware described in Subsection 3.8.3.

Figure 4.3 shows the time taken for each file on the x-axis, grouped by the Quixbugs filename on the y-axis. For the human group, the figure displays the mean time taken for each file. The scale on the x-axis is logarithmic, to account for the large differences in the values. The color of the bars denotes which group the bar belongs to, with blue standing for the human participants, orange for SequenceR, and green for Recoder.

Overall, SequenceR is the fastest approach, as it is roughly 35x faster than Recoder, and 128x faster than the human group. Depending on the file, SequenceR takes between four and seven seconds to finish processing one file, with the mean time being approximately 4.69 seconds. Recoder

[1] `https://copilot.github.com/`

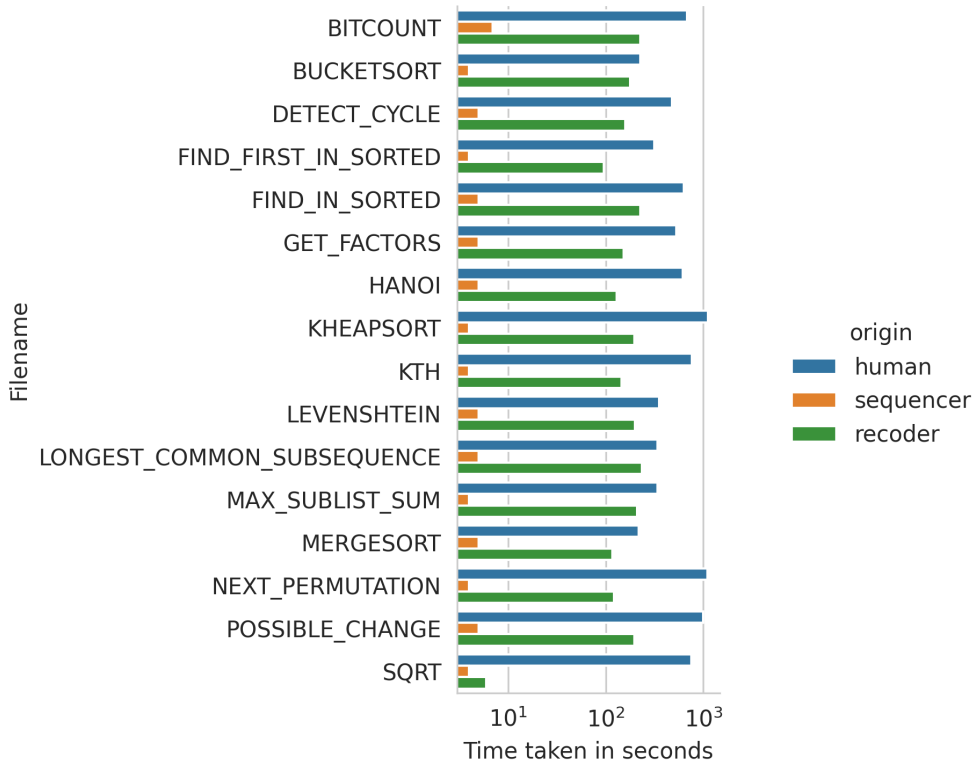


Figure 4.3: Time taken for repairs

follows as the second-fastest and is roughly 3.56x faster than the human group and takes between 6 and 236 seconds to process a file, with mean time being 162.19 seconds per file. The developers are the slowest group, taking between 218 and 1112 seconds per file, for a mean of 589.97 seconds per file.

Answer to RQ1: Human developers drastically outperform the neural approaches on the Quixbugs dataset. The group of human developers is not only able to produce at least one correct fix for each file in the selected subset of Quixbugs but also produces a plausible fix in almost 71% of the total attempts. If the returned beam search results are counted as individual attempts for the APR approaches, the approaches only produce plausible fixes for approximately 1% (SequenceR) and 0.7% (Recoder) of the total attempts.
However, high repair performance comes at a cost in terms of time taken for completing the repair: SequenceR is the fastest of the three, taking on average 4.69 seconds for one buggy file, followed by Recoder at 162.19 seconds, and the humans at 589.97 seconds.

The repair performance results indicate that state-of-the-art APR approaches still have much room for improvement before approaching human performance, except for speed. The percentage of correct fixes out of the set of all fix attempts is an especially important area of improvement. One of the envisioned use cases of automatic program repair is to improve developer productivity by providing a list of suggested fixes [15]. The two evaluated approaches produce between 50 and 100 fix candidates per buggy file, with around 1% of these being at least plausible. As these candidates sometimes occur towards the end of the list, human developers would need to search the entire

list before finding a viable patch candidate. It is unlikely that developers would do this, instead of investing the needed time in fixing the bug directly. Future work on automatic program repair could improve on this by putting an emphasis on top-k performance, for small values of k such as 5 or 10, meaning that a correct patch candidate only counts if it occurs in the first k patches of the candidate list.

4.3 RQ 2: Correlation between Human and Model Attention

This section explores the correlation between the attention collected from the human developers and that extracted from the neural models underlying the two APR approaches, SequenceR and Recoder. Subsection 4.3.1 compares the individual attention vectors collected from the developers to obtain an intuitive upper bound for the human-model correlation. Subsections 4.3.2 and 4.3.3 analyze the correlation between the human attention, and that of SequenceR and Recoder, respectively. Finally, Subsection 4.3.4 looks at the relationship between the attention of SequenceR and Recoder, before concluding the section.

4.3.1 Human vs. Human Correlation

An intuitive upper bound for the correlation between models and humans is the correlation between the individual human participants. Figure 4.4 shows the mean Spearman rank correlation coefficient between the human participants on the x-axis and the names of the files of the task dataset on the y-axis. The grey error bars denote the 95% confidence interval. The underlying data consists of the Spearman rank correlation coefficients between the possible pairs of human attention vectors for each file, as each file was presented to multiple developers. We exclude correlations for human-human pairs where $p < 0.05$. This results in between 10 and 21 pairs per file.

As the figure indicates, the human-human correlation varies between the different files, with a minimum of 0.12 and a maximum of 0.95. The overall mean human-human correlation coefficient is 0.55. Between the different files, the mean correlation coefficient ranges from 0.38 for MERGESORT to 0.74 for BITCOUNT.

4.3.2 Humans vs. SequenceR

For the comparison between the developers and SequenceR, we calculate the Spearman rank correlation coefficient between the attention vector extracted from SequenceR, and the vectors collected from the individual human participants. We exclude data points where $p < 0.05$. This results in between 3 and 7 correlation measurements per file.

Figure 4.5 shows the mean correlation coefficient between humans and SequenceR on the x-axis, grouped by the individual Quixbugs files on the y-axis. The grey error bars indicate the 95% confidence interval. The bright orange bars stand for the "regular" encoder/decoder attention extracted from SequenceR, while the dark orange bars stand for the copy attention.

Human-SequenceR correlation ranges from a minimum of 0.14 to a maximum of 0.59. The overall mean Spearman rank correlation coefficient is 0.35. The mean correlation per file ranges

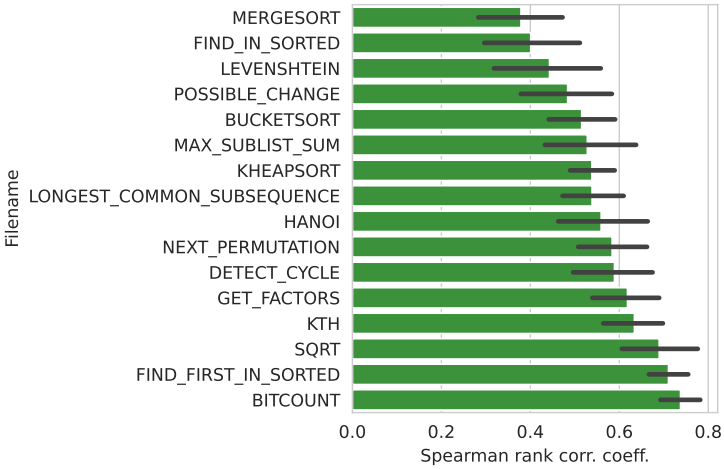


Figure 4.4: Human vs. Human Correlation

from 0.19 for MERGESORT, to 0.51 for HANOI. These results indicate that depending on the individual file, there is a weak to moderate, but statistically significant, correlation between the attention of SequenceR and that of human participants. The mean correlation between the humans and SequenceR is significantly lower than that between the individual human participants (one-sided t-test: $t = 9.655, p < 0.001$)

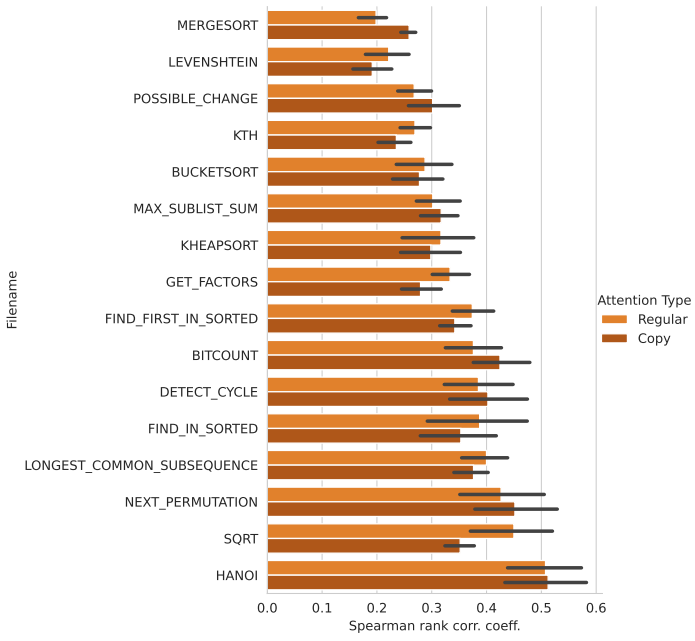


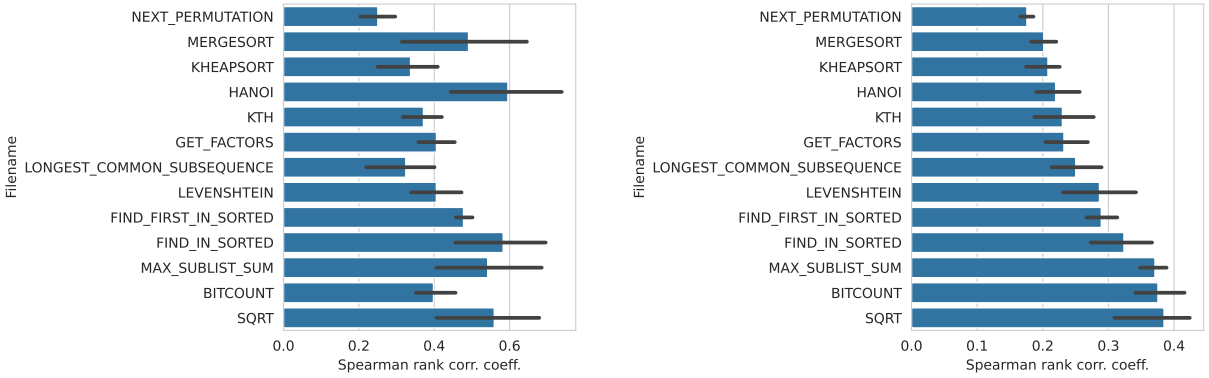
Figure 4.5: Correlation between Sequencer and Human attention

Figure 4.5 indicates that the correlation between the copy attention and the human attention is very similar to that between the encoder/decoder attention and the humans. It ranges from a minimum of 0.16 to a maximum of 0.60, with the mean being 0.34. The copy attention also correlates strongly with the encoder/decoder attention, with the Spearman rank correlation coefficient

between the two ranging from a minimum of 0.75 to a maximum of 0.97, with a mean of 0.89. The likely explanation behind this is the fact the encoder/decoder attention and the copy attention directly relate in SequenceR's architecture, as explained in Subsection 2.2.2.

4.3.3 Humans vs. Recoder

We transform Recoder's attention over the AST traversal to a token vector according to the procedure explained in Subsection 3.8.2 and compare it to the attention vectors gathered from the developer group. Recoder only pays attention to a subtree of the AST that corresponds to the method that contains the buggy line. For this reason, we compare both the full attention vectors, and shorter attention vectors that are truncated to only contain the tokens of the buggy method. In the former case, we assign tokens outside of the buggy method an attention score of 0 to reflect the fact that Recoder cannot pay attention to them. This allows an analysis of both the resulting overall attention distribution, as well as a method-level comparison.



(a) Overall

(b) Buggy Method

Figure 4.6: Human vs. Recoder Correlation.

Figure 4.6 shows the correlations for both cases, with the Spearman rank correlation coefficient on the x-axis, grouped by the Quixbugs filename on the y-axis. The color of the bars indicates whether Recoder is able to fix the respective file, with blue bars standing for those that it cannot fix, and orange bars standing for those that it can. The grey error bars indicate the 0.95% confidence interval. We exclude data points where the associated $p \geq 0.05$.

Over the entire input program, the Spearman rank correlation coefficient between Recoder's attention and the human developers' ranges from a minimum of 0.17 to a maximum of 0.80, with the mean being 0.43. Truncated to the buggy method, the Spearman rank correlation coefficient ranges from a minimum of 0.16 to a maximum of 0.46, with a mean of 0.28. The mean correlation for each file ranges from 0.25 for NEXT_PERMUTATION to 0.60 for HANOI in the overall case, and from 0.18 for NEXT_PERMUTATION to 0.38 for SQRT when only considering the buggy method.

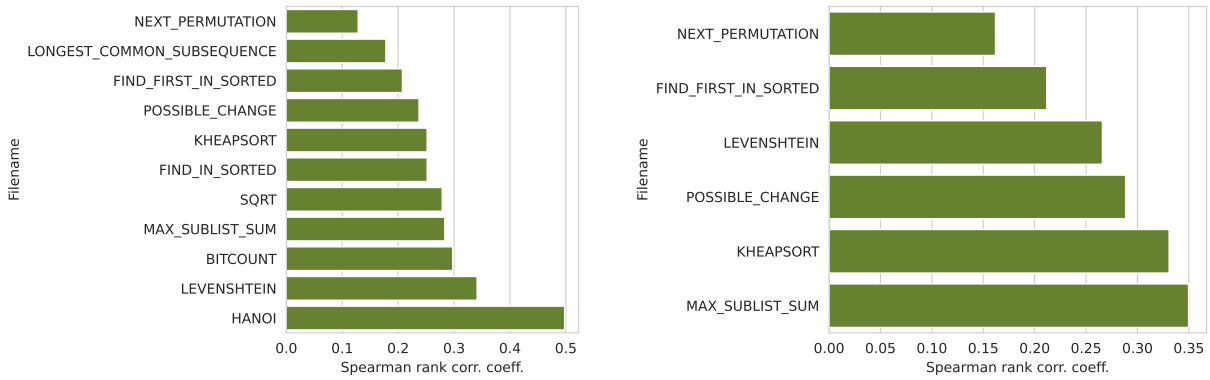
These results indicate that depending on the input file, there is a moderate to strong correlation between the attention of Recoder and that of the human developers, which is significantly higher than that between SequenceR and the humans. Looking online at the buggy method, the correlation

is still statistically significant, but weak to moderate. The construction of Recoder's attention mechanism prevents it from attending to anything but the buggy method, as well as preventing it from paying attention to some structural tokens such as separators, certain keywords, etc. The correlation results imply that this construction makes the overall attention distribution much more human-like, by blocking out parts of the input that humans also do not attend to.

We hypothesize that this restriction of what parts of the input go in the attention mechanism contributes to the higher repair performance of Recoder over SequenceR: The high performance of the developers on the Quixbugs dataset suggests using their attention as a guideline to what parts of the programs are relevant for fixing the bug. The high correlation between Recoder and the humans on the overall program would then imply that the input restriction is successful in focussing Recoder's attention on the relevant parts of the program. In contrast, SequenceR can pay attention to any token in the input sequence, which increases the risk of it attending to irrelevant parts of the input program. The lower correlation scores between the humans and SequenceR suggest that this is indeed the case. Future APR approaches could benefit from a similar input restriction strategy, where irrelevant parts of the input are blocked out, allowing the model's attention to focus more on the relevant parts of the input program.

4.3.4 SequenceR vs. Recoder

As for the human comparison, we both analyze the correlation between the attention of SequenceR and Recoder over the entire input, as well as for the buggy method only. Figure 4.7 shows the Spearman rank correlation coefficient on the x-axis, grouped by the Quixbugs filename on the y-axis, for both the overall and buggy method cases. The figure does excludes data points where $p \geq 0.05$.



(a) Overall

(b) Buggy Method

Figure 4.7: Recoder vs. SequenceR Correlation

Overall, the Spearman rank correlation coefficient between Recoder and Sequencer ranges from a minimum of 0.13 to a maximum of 0.49, with the mean being 0.27. For the buggy method only, the correlation ranges from a minimum of 0.16 to a maximum of 0.35, with a mean of 0.27. This implies that there is a weak, but statistically significant, correlation between the attention of Recoder and Sequencer for 11 out of 16 files in the dataset. Focusing only on the buggy method does

not substantially change the correlation, with the overall mean correlation being approximately the same. For all other files in the dataset, the correlation is not statistically significant.

The weak correlation between the attention of SequenceR and Recoder indicates that the two approaches pay attention to different parts of the input. This is supported by the finding that the correlation does not substantially change when only looking at the tokens of the buggy method, with the correlation coefficients being very similar. As Recoder exhibits a stronger correlation with human attention than SequenceR, we interpret these results as a further indicator that Recoder pays attention to more relevant areas of code, under the assumption that human attention is a valid guideline for relevancy. Specifically, the result indicates that the representation of the input program as an abstract syntax tree more closely resembles the way that humans look at code.

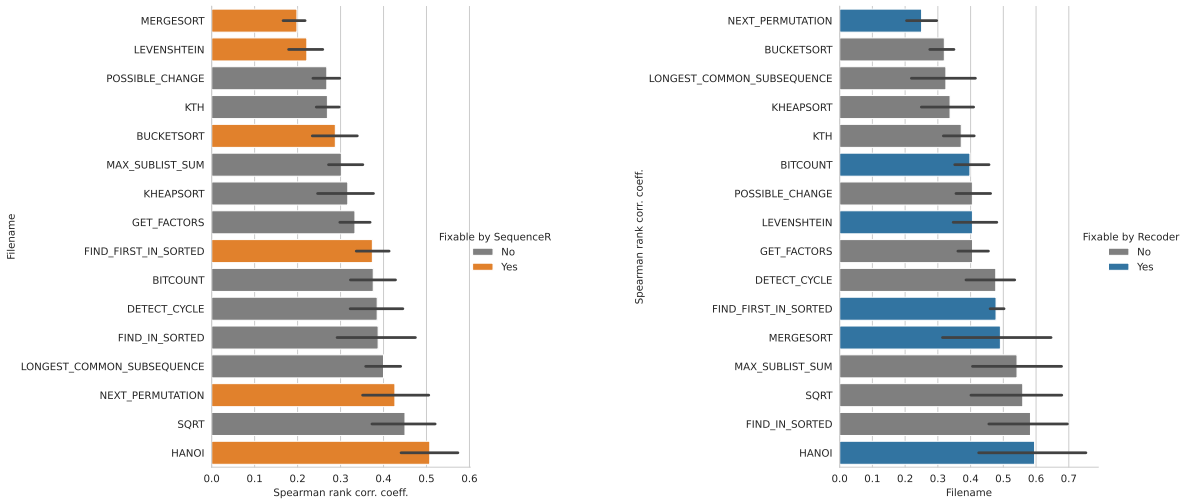
Answer to RQ2: We find that there is a weak to moderate correlation between the attention of SequenceR and that of the developers, with the mean Spearman rank correlation being 0.35. There is a similar correlation between SequenceR's copy attention and the attention of the developers, with the mean Spearman rank correlation being 0.34. For Recoder, we find that over the entire input, there is a moderate to strong correlation between its attention and that of the developers, with the mean Spearman rank correlation being 0.43. We find that the construction of Recoder's attention mechanism, which prevents it from seeing certain parts of the input program, contributes strongly to the high correlation between it and the humans over the entire input. Between SequenceR and Recoder, we find that there is a weak, but statistically significant, correlation between their attention vectors for half of the files in the dataset, indicating that they pay attention to different parts of the input.

4.4 RQ 3: Human Correlation vs. Repair Performance

The higher correlation between the human attention and that of the better-performing approach, Recoder, motivates a further investigation into the relationship between human-model correlation and repair performance. To this end, this section asks the question of whether there is a relationship between higher human-model correlation and the ability of the models to fix a given input program. We extend the correlation analysis of the previous section with labels for fixability on the side of the APR approaches. For SequenceR, this section considers the regular encoder/decoder attention.

Figure 4.8 compares the model-human correlation of both approaches side-by-side. It shows the mean Spearman rank correlation coefficient between the human developers and the model attention on the x-axis, grouped by the Quixbugs filename on the y-axis. The subfigure to the left shows the results for SequenceR, the one to the right for Recoder. For SequenceR, the orange bars indicate that it is able to generate at least one plausible fix for the corresponding file. For Recoder, the blue bars stand for files for which it can generate at least one plausible fix. The grey bars stand for files where the approaches cannot generate a fix.

As Figure 4.8 a) shows, the files that SequenceR can fix are distributed relatively evenly across the range of human-model correlations. The mean Spearman rank correlation coefficient for the files that SequenceR can fix is 0.34, while it is 0.35 for those that it cannot fix. The difference in the mean correlation is not statistically significant, with $t = -0.301, p = 0.764$.



(a) Humans vs. Sequencer (b) Humans vs. Recoder

Figure 4.8: Model vs. Human correlation, with repair performance

For SequenceR, we also analyze additional attention vectors from the beam search: The instrumented implementation of SequenceR provides an attention vector for each generated patch, of which there are up to fifty per input file. The previous analyses use the vector corresponding to the most likely plausible patch for files that the approach can fix, and otherwise the vector corresponding to the most likely patch. To use this available additional data, we investigate whether there is a significant difference in the human-model correlation between attention vectors corresponding to plausible patches and those that do not.

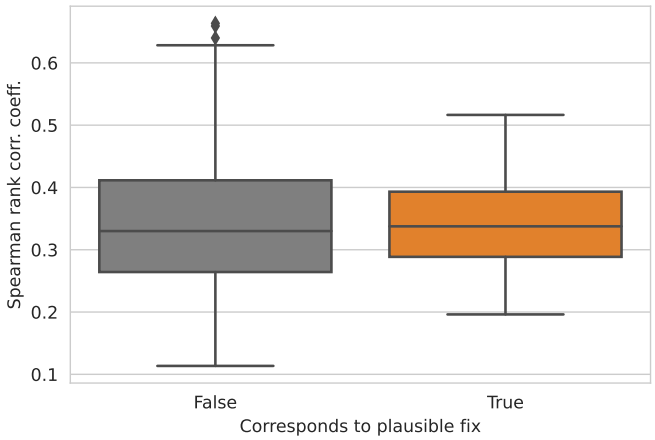


Figure 4.9: Correlation Box plot for SequenceR.

The resulting mean Spearman rank correlation coefficient between the developers and the vectors corresponding to plausible patches is 0.344, for vectors corresponding to non-plausible patches, it is 0.341. In keeping with the previous result, this difference is not statistically significant, with $t = 0.240, p = 0.405$. Figure 4.9 illustrates this, with the box plot indicating that there does not

seem to be a significant difference in the mean correlations of the two groups.

Figure 4.8 b) shows a similar distribution of the blue bars for Recoder, with half of the programs that Recoder can fix being in the top half correlation-wise, and the other half being in the bottom half of model-human correlation. The mean Spearman rank correlation coefficient for the files that Recoder can fix is 0.46, and for those that it cannot, it is 0.43. Again, this difference in the means is not statistically significant, with $t = 0.856, p = 0.198$.

Answer to RQ3: We find that for both APR approaches, there is no statistically significant difference in the mean Spearman rank correlation coefficient between model and human attention. One-sided t-tests yield $t = -0.301, p = 0.764$ for SequenceR, and $t = 0.497, p = 0.310$ for Recoder, respectively. Additionally, for attention vectors from SequenceR's beam search, there is no statistically significant difference in the mean Spearman rank correlation between vectors that correspond to plausible patches, and vectors that do not ($t = 0.240, p = 0.405$). These results suggest that for the selected approaches and input programs, there is no relationship between the models' ability to fix a program and high human-model correlation.

Intuitively, the expectation would be to find a positive relationship between high human-model correlation and the models' repair performance. However, the results presented in this section do not find a statistically significant relationship between the two. We suppose that this is a result of the small size of our dataset. Related work that finds a positive relationship between model performance and human-model correlation using a similar methodology [24] uses a human dataset that is approximately 10 times larger than the one presented by this work, which could explain the neutral results presented above.

4.5 RQ 4: Attention: Buggy Line vs. Context

This section explores the percentage of total attention devoted to the buggy line versus the surrounding context. For the human group and SequenceR, we define the attention paid to the buggy line as the sum of the attention of all tokens on the buggy line, whereas for Recoder, we define it as the sum of the attention of all AST nodes corresponding to the buggy line. We define the total attention as the sum of the elements of the respective attention vectors. The context is then simply the parts of the input that do not belong to the input, i.e. everything except the tokens or nodes of the buggy line according to the previous definitions.

Figure 4.10 shows the percentage of attention devoted to the buggy line for the two approaches and the human group on the y-axis, grouped by the Quixbugs filename on the x-axis. The bar color indicates the group that a value belongs to, with blue standing for the humans, orange for SequenceR, and green for Recoder. As the human group has multiple participants, the error bars indicate the 95% confidence interval based on the individual results.

As Figure 4.10 shows, SequenceR pays the most attention to the buggy line, with the overall mean attention on the buggy line being 67.10% of the total attention. The copy attention is focused even more on the buggy line, with the overall mean percentage on the buggy line being 76.71% of the total attention. The human group pays the second-most attention to the buggy line, with the overall mean percentage being approximately 36.85% of the total attention. Recoder pays the least

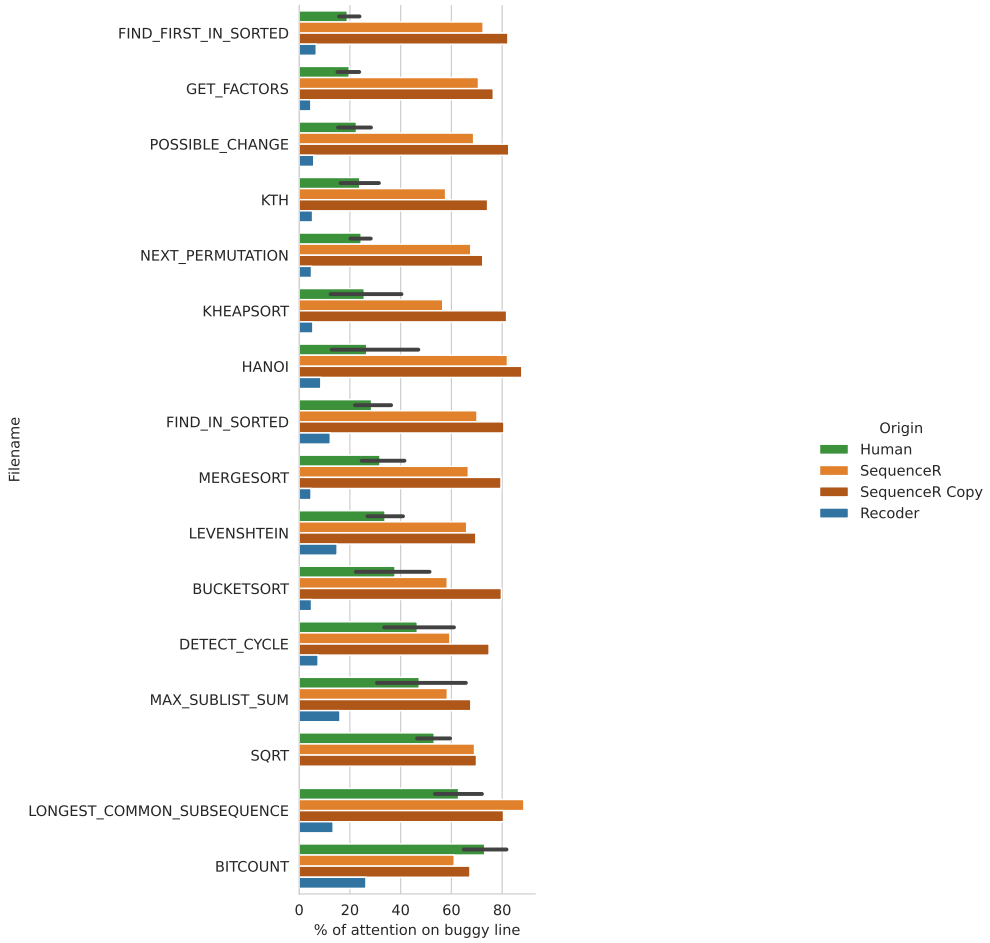


Figure 4.10: Percentage of attention on the buggy line

attention to the buggy line, the AST nodes corresponding to the buggy line only get approximately 9.42% of the total attention.

Intuitively, it could be assumed that humans, and to some extent the approaches, ignore irrelevant statements such as standard library imports or the package declaration. For short programs, this would mean that the percentage of attention on the buggy line would be higher due to the buggy line making up a greater proportion of the relevant parts of the program. To account for this possibility, we also analyze the percentage of attention on the buggy line versus the length of the input. Recall that for humans and SequenceR, the input is the token sequence, whereas Recoder takes the AST traversal sequence of the buggy method as an input.

Figure 4.11 shows the percentage of attention devoted to the buggy line on the y-axis, versus the input length on the x-axis. The coloring indicates the group that each data point belongs to, with the coloring following the color scheme established in Figure 4.10. The input length values are the length of the token sequence for the humans and SequenceR, and the length of the AST traversal sequence for Recoder.

The scatter plot shown in Figure 4.11 implies that there seems to be a negative relationship between the percentage of attention devoted to the buggy line and the input length, with the former declining as the latter grows. To strengthen this notion, we take into account the Pearson

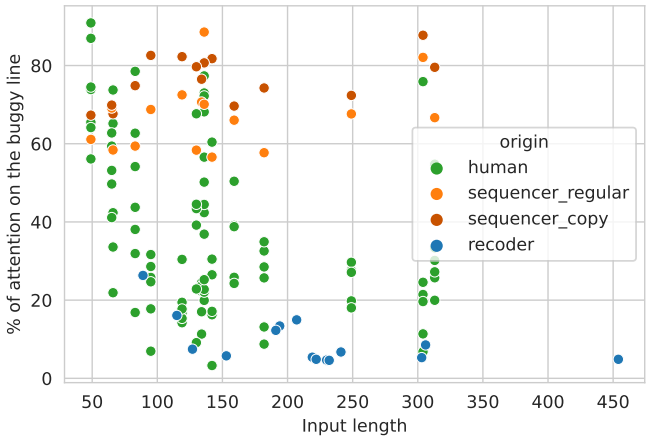


Figure 4.11: Attention on buggy line vs. input length

correlation coefficient between the buggy line percentage and the input length. For the human group, the coefficient is -0.37, $p < 0.01$, indicating a moderate negative relationship between the two variables. For SequenceR, it is at 0.30, $p \approx 0.249$, indicating a weak positive, but not statistically significant, relationship between the two variables and 0.45, $p \approx 0.08$ for the copy attention, indicating a moderate positive, but also not statistically significant relationship between the variables. For Recoder, the coefficient is -0.57, $p < 0.05$, indicating a moderate to strong negative relationship between buggy line percentage and input length. These results indicate that the percentage of attention devoted to the buggy line does indeed depend on the input length for the humans and Recoder, with the percentage declining as the input length increases. For SequenceR, no definitive statement on the relationship can be made, as the statistical uncertainty is too high.

Answer to RQ4: The percentage of attention devoted to the buggy line varies between the three groups: SequenceR pays attention almost exclusively to the buggy line, with approximately 81.25% of its attention being devoted to the buggy line. Recoder on the other hand pays attention primarily to the context, with only 10.98% of the attention being on the buggy line. The human group lies in between, at approximately 38.14% of the total attention being paid to the buggy line, and the rest to the context.
The percentage of attention paid to the buggy line does depend on the length of the input program for the humans and Recoder, with the correlation (Pearson's r) between the percentage and length being -0.36 for the humans, and -0.60 for Recoder, respectively. This indicates that the context surrounding the buggy line becomes more important as input length grows.

Out of the three groups, SequenceR allocates the most of its attention to the buggy line, in some cases even over 80%. To illustrate the extent to which SequenceR pays attention to the buggy line, we compare it to a "strawman" attention vector, where all attention has been put on the buggy line by replacing the token indices of the buggy line with one, and all other indices with zero. The mean Spearman rank correlation coefficient between SequenceR's attention and the straw man is 0.45, which is significantly higher than that between the humans and Sequencer

(t-test: $t = 3.751, p < 0.001$).

We conjecture that this is a result of the design of the approach. SequenceR produces candidate patches by predicting a replacement for the buggy line [11]. Since bugs, especially in Quixbugs, often consist of only one misplaced token, for example, a variable identifier or an operator, it essentially has to re-predict the entire buggy line, except for the buggy token. Taking into account the copy mechanism, which relies on the attention distribution to select copied tokens, this seems a likely explanation for SequenceR's strong focus on the buggy line, for both types of attention. However, even though all bugs in the dataset are limited to a single line, the information necessary for fixing the bug is often only found in the surrounding context. For example, a fix might consist of replacing an incorrectly used variable with the correct one, the identifier of which occurs somewhere else in the surrounding context, but not on the buggy line.

On the other hand, Recoder pays much more attention to the surrounding context, even significantly less so than the developers, especially considering the fact that it cannot attend to anything outside of the buggy method by design. The percentage of attention that the human developers pay to the buggy line lies in between the two APR approaches and varies heavily by file. With Recoder outperforming SequenceR, and the humans outperforming both by a wide margin, one could consider an additional hyperparameter for future APR models that controls the proportion of attention paid to the location of the bug versus the surrounding context. Optimizing this hyperparameter could allow reaching an optimal balance between buggy location and context, enabling models to tackle both bugs that are context-dependent and bugs that are not.

4.6 RQ 5: Token Type Analysis

Another dimension of both human and model attention is the type of tokens that they look at the most, and which they tend to ignore. This section analyzes this dimension by looking at which types of tokens receive more than uniform attention, and which do not. For Sequencer, this section considers the regular encoder/decoder attention. To quantify this notion, this section adopts a metric from [24], the distance from uniformity (DFU): "Given a vector of attention weights $\vec{a}$ for a sequence T of tokens, the distance from uniformity of a subset $S \subseteq T$ is: $DFU(S) = \frac{\sum_{t \in S} a_t - \frac{|S|}{|T|}}{\frac{|S|}{|T|}}$, where a_t is the attention weight assigned to token t." [24]

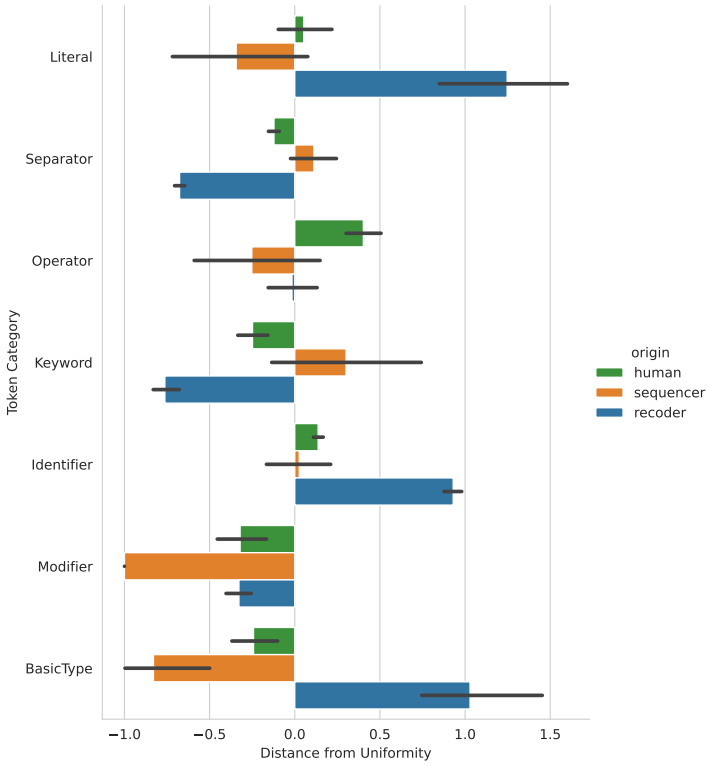


Figure 4.12: DFU for different token types

Figure 4.12 shows the mean DFU over all input files for the different types of tokens. The DFU is on the x-axis, and the y-axis denotes the name of the token type. The color of the bars stands for the group that the data comes from, with green standing for the developers, orange for SequenceR, and blue for Recoder. The grey error bars indicate the 95% confidence interval. The "Annotation" token type is excluded from the dataset, as there is only one token of that type in the dataset. Furthermore, we merge the token types corresponding to literals into the "Literal" token type. The values for Recoder are based on the token attention vector obtained via the procedure described in Subsection 3.8.2.

Overall, the human participants pay more than uniform attention only to operators, literals, and identifiers. All other types of tokens receive less than uniform attention, with basic types and modifiers being the least-attended-to tokens. SequenceR pays more than uniform attention to keywords and separators, paying less than uniform attention to all other types of tokens. Similar to the developers, it pays the least attention to modifiers and basic types. Using the converted attention vectors, Recoder pays more than uniform attention to literals, identifiers, and basic types, while paying less than uniform attention to the other token types. Recoder pays the least attention to keywords and separators, with the DFU values for the operators being too uncertain to make a definitive statement.

For Recoder, the AST-to-token mapping strategy has to be taken into account when interpreting the results. As it pays attention to a sequence of AST nodes, it can by design not pay attention to separators and keywords. The former are semantically replaced by the relationships between

the AST nodes, and the latter by distinct types of AST nodes. Furthermore, in the AST structure that Recoder uses, there are often several nodes corresponding to the same token, which could partly explain its strong focus on literals, identifiers, and basic types, as these usually consist of one token in the token sequence. These properties of the AST traversal sequence attention survive the AST-to-token mapping and are reflected in the DFU scores for each token type.

Answer to RQ5: The approaches and humans agree on paying less than uniform attention to modifiers, with SequenceR almost entirely ignoring them. Furthermore, both the humans and SequenceR pay less than uniform attention to types. SequenceR pays more than uniform attention to separators and keywords, but with high uncertainty in the data. Recoder pays much less than uniform attention to separators and keywords, and much more to literals, identifiers, and basic types.

One key takeaway from the results presented above is the fact that the design of Recoder, specifically the choice of encoding the input program as an AST instead of a token sequence, matches the way that humans look at the code to a certain extent. This may suggest that future work on APR should prefer encoding programs as ASTs, to take advantage of this similarity.

This is supported by related research on method summarization. Paltenghi and Pradel [24] report that in a similar evaluation, a Transformer-based model paid a high amount of attention to separators. Recoder, which also follows the Transformer architecture, but pays attention to the AST traversal sequence, pays almost no attention to identifiers by design.

4.7 RQ 6: Patterns in Developer Behavior

One avenue of working towards human-like performance in automatic program repair is to study the bug fixing strategies of human developers. This section analyzes the behavior of the human developers based on the data collected using the Edit Attention Recorder. The first general pattern in developer behavior relates to the distribution of the edits, which EAR records in the form of edit events.

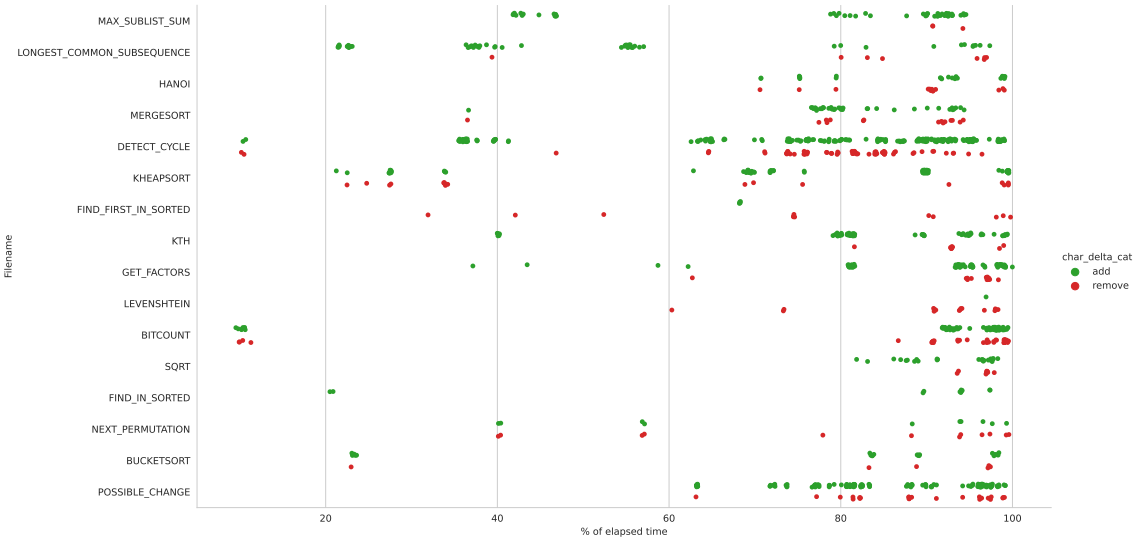


Figure 4.13: Edits over time, by file.

Figure 4.13 shows the distribution of the edit events over time. The x-axis denotes the percentage of passed time since the participant started their task, grouped by the Quixbugs filename on the y-axis. Green colored dots stand for edits that add characters, while red dots stand for events that remove characters. The general distribution of the edit events implies that the developers tend to edit the code towards the end of the task. We quantify this further by taking into account the count of edit events over time:

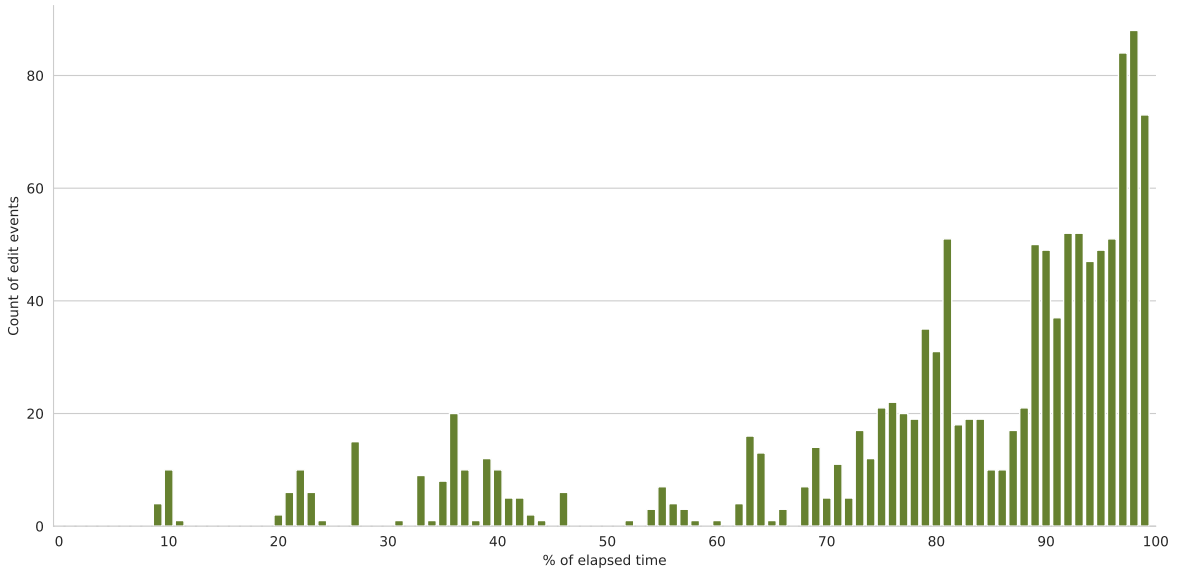


Figure 4.14: Edit counts, over time.

Figure 4.14 shows the count of edit events over time, with the percentage of elapsed time on the x-axis, and the event count on the y-axis. There is one global maximum of the event count at ca. 97% of the elapsed time, preceded by a smaller peak at around 80% of elapsed time. In

addition, there are several smaller peaks in the event count on the first half of the time axis. The global peak makes out almost half of the events that occur, with approx. 47.74% of the events occurring in the final 10% of the time. Together, the two peaks at the end of the time axis make out almost three-quarters of the edit events, with approx. 81.62% of events occurring in the final 30% of elapsed time.

However, the smaller peaks on the first half of the time axis suggest that a part of the developers follows a different strategy that involves earlier edits. A possible explanation is a strategy where developers iterate while fixing the bug, i.e. they edit the source code multiple times, thinking about the viability of their fix candidate in between edits. To quantify this idea, we divide the overall elapsed time into ten percentage "bins", with the first bin being $0\% < t \leq 10\%$, the second $10\% < t \leq 20\%$, etc., where t is the percentage of elapsed time at which an event occurs. For each fix attempt, we count the number of events that occurs during the frame of time corresponding to each bin. From this, we obtain the number of edit groups for each fix attempt in the dataset.

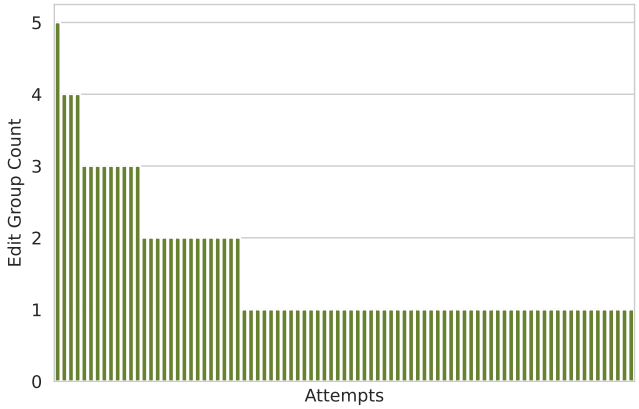


Figure 4.15: Iterated Edits

Figure 4.15 shows the number of edit groups for all fix attempts in the dataset, with the number of groups on the y-axis, and the attempts on the x-axis. The figure shows that about a third of the developers have two or more edit groups during their fix attempts, meaning that they perform at least one iteration on their fix candidate.

To further explore the overall pattern of developers first exploring the code, and then starting to edit, we consider the counts over time of three types of events, namely unblur, edit, and hint usage events. Figure 4.16 shows these counts over time. The red line stands for the unblur events, which are generated whenever tokens are unblurred due to user interaction with EAR. The blue line stands for the count of the hint usage events, which are recorded whenever users start to view the hint on what the algorithm is supposed to do, and the yellow line denotes the edit event count. The y-axis gives the event counts, whereas the x-axis denotes the percentage of elapsed time over 100 intervals.

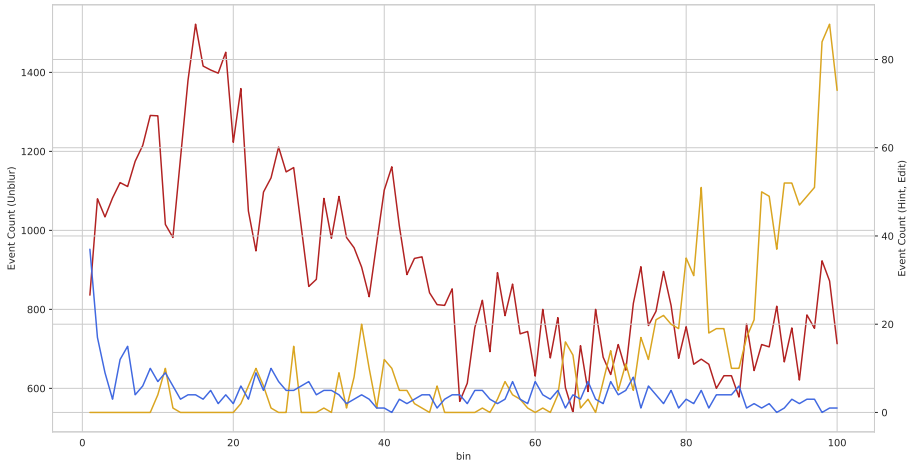


Figure 4.16: Different events over time

The red line indicates that the count of attention events per time drops as time goes on, which indicates that the users are interacting less with the application, respectively cause fewer unblur events by decreased movement of the mouse and/or cursor. At the same time, the usage of the hint drops sharply as time goes on, with most users using it during the first 10% of their sessions, and then only infrequently. Finally, as established previously, the edit events go up towards the end of the timeline.

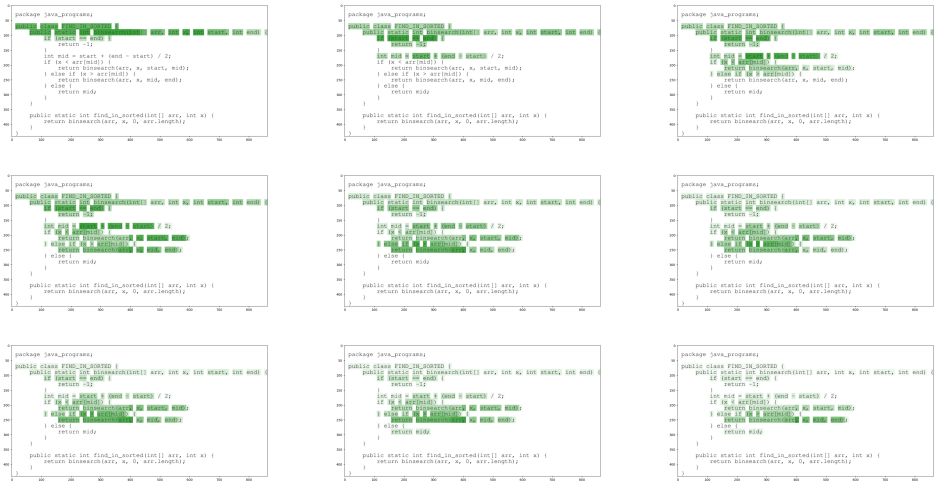


Figure 4.17: Cumulative attention over time.

These results seem to suggest a general pattern in developer behavior: they spend the first two-thirds of their session exploring the code and trying to understand the underlying algorithm, as well as the bug, before conducting the actual repair in the final third of their session. Figure 4.17 shows an example of this pattern from the human study. The grid was created using the attention video visualization tool that is part of the tooling created for this work, and shows the cumulative

attention heat map for a human participant as time goes on. The attention is relatively evenly distributed in the beginning, before focusing more and more on the area of the bug, where it is almost exclusively focused in the final frame. The attention video tool is part of the GitHub repository accompanying this work.

Answer to RQ6: We identify two patterns in developer behavior during code repair: The first relates to the total time spent exploring, versus repairing the code. Developers spend approximately two-thirds of their time on task exploring the code to gain an understanding of the underlying algorithm and the bug, before starting repairs in the last third of the time on ask. The second pattern relates to the edit strategy. A significant minority of developers have more than one group of edit events over time, meaning that they revise their fix candidate at least once.

The relatively long time spent exploring the code indicates that the developers place a lot of emphasis on understanding the code. This suggests that future APR approaches may benefit from an increased understanding of the input program and its intended behavior. One way to realize this would be to provide models with an additional input that specifies the intended behavior of the program, such as input/output examples, or a formal description of the underlying algorithm. Another option would be to use a pre-trained model that was trained on large amounts of code and fine-tune it to the task at hand. Recent research lends credence to the latter idea. Prenner and Robbes [25] evaluate the Codex [10] model, a very large model of code trained on billions of lines of code, on the Quixbugs benchmark. They find that Codex is capable of fixing 14 out of the 40 bugs in the dataset, without having been fine-tuned for APR tasks. This performance lies in between that of SequenceR and Recoder, highlighting the potential of adapting a model of that type to APR tasks.

The pattern of iterative editing has already found application on the field of automatic program repair. Recoder [39] generates the output sequence of edits over multiple queries to the model, taking the previous edits as an input for the AST reader component at each step. An approach presented by Yao et al. [37] iteratively generates a sequence of tree edits. The relatively high repair performance of both approaches indicates that the adoption of this human behavioral pattern offers benefits for the performance of APR approaches.

4.8 Threats to Validity

This section discusses potential threats to the validity of the study results presented in this chapter. The first two threats are to internal validity, namely the additional hint provided to human participants and the instrumentation code added to the APR approaches. The third threat is to external validity, namely whether results on the Quixbugs dataset generalize well to other categories of bugs.

4.8.1 Hint Usage

One deviation from the principle of providing equal information to both humans and approaches is the hint on the algorithm at hand provided to the human participants. EAR provides this hint

because a pilot study found that human participants will just acquire information on the algorithm via an internet search or similar, and providing the hint allows tracking the usage of the hint. It furthermore may also limit the additional information provided to human participants to the known quantity of the hint itself, whereas humans may otherwise search for reference implementations of the algorithm, other occurrences of the bug, etc.

However, repair performance results indicate that this additional knowledge does not substantially change the performance of the human participants: 18 participants did not use the hint at all, with 15 of them also reporting no internet searches. Out of these 15 participants, 11 did produce a plausible fix for their respective task (73.33%). The performance of this group is in line with the performance of the overall group, which is at approximately 70%. This indicates that the availability of the hint does not drastically change human repair performance on Quixbugs.

4.8.2 Model Utilization

Two additional threats to internal validity lie in the instrumentation that is added to the approaches, and in the way that extracted attention is post-processed. For SequenceR, we modify its buggy context abstraction step to not replace certain parts of the source code with semantically but not syntactically equivalent replacements. After this replacement step, SequenceR is able to produce plausible patches for two fewer files in the dataset but gains the ability to fix a file that it previously could not fix. This might indicate that the one-file drop is merely a result of the minor changes in the input, and not that SequenceR has actually degraded in repair performance.

For Recoder, a potential threat to the validity of the results is the AST-to-token mapping strategy. Information may be lost in the mapping process, and inevitable ambiguities in the semantics of AST nodes vs. tokens may distort the results. However, we observe that certain features resulting from attention over the AST traversal sequence, such as Recoder's inability to see separators or keywords, translate well into the token vector, with the token type analysis yielding that it pays less than uniform attention to these token types, as would be expected. We additionally computed heat maps of Recoder's attention over the AST nodes and over the token sequence and performed a qualitative inspection of whether the two are in agreement.

4.8.3 Quixbugs

The final threat to validity relates to the usage of the Quixbugs dataset. Due to it originating from a hiring challenge [22], the bugs in it may not accurately reflect real-world bugs that are different in scope and size. However, even though the source code files are of similar small size, the bugs contained within are often relatively complex due to recursion or nested loops [38]. Furthermore, the dataset also contains 17 different bug types spread over 40 bugs. The diversity in the dataset mitigates this concern. Finally, due to the widespread usage of Quixbugs as a benchmark dataset, the results of this study are immediately comparable to other results on the same benchmark. We argue that this benefit outweighs the remaining deficits in the generalizability of the dataset.

5 Related Work

This chapter gives an overview of relevant related work, starting with the research also studying attention mechanisms to interpret neural models in Section 5.1. Section 5.2 covers research relating to human behavior while working on code repair tasks. Closing the chapter, Section 5.3 touches on the subject of explainable AI (XAI).

5.1 Comparing Attention

In closely related work, Paltenghi and Pradel [24] conduct a comparative analysis between neural models for code summarization and human participants. They use a platform called the *Human Reasoning Recoder*, a direct ancestor of the Edit Attention Recorder, to record the attention of human developers during a method naming task, with a similar blurring mechanism as the basis of attention calculation. On the field of visual question answering, Das et al. [12] conduct a comparative study of humans and attention-based models, where human attention maps are also collected via a blurring-based interface. In their approach, humans and models have to answer questions about an image, with humans being able to selectively deblur parts of the image, similar to how the human participants selectively deblur parts of the code in EAR.

Attention weights are frequently interpreted to judge what the model has learned, and the quality of what it has learned. Bahdanau et al. [4], who introduce the attention mechanism for encoder/decoder architectures, also inspect the attention weights to assess the quality of the alignments between input and output sequence words that the model has learned. On the field of automatic program repair, Gupta et al. [16] also interpret the attention weights of their sequence-to-sequence model to conjecture about the reasoning of the model when producing a fix. Lutellier et al. [23] also interpret the attention maps of their model, to gain a better understanding of why it produces certain fixes, and to explain why the model performs well for some fix types, and bad for others.

Another avenue to capture the attention of humans is to use eye-tracking data. Sood et al. [31] compare the attention of several types of neural models for text comprehension to the visual attention of human developers, which they gather by eye-tracking. Eye-tracking data is also frequently used in human behavioral studies, which the following section elaborates upon.

5.2 Behavioral Studies involving Human Developers

Related research also includes the study of the behavior of human developers, and their strategies in bug repair and code comprehension. Busjahn et al. [8] investigate the linearity of code reading between novice and expert programmers using eye and mouse tracking data, finding that expert developers exhibit a less linear order of code reading than novices. On a human neural level, Castelhano et al. [9] investigate activation patterns in the brains of developers searching for a bug, describing a distinct "eureka" moment when developers suspect that they have found a bug. Future work could incorporate findings from brain research into the design of new neural model architectures for automatic program repair, similar to how the concept of convolutional neural networks stems from the visual cortices of cats [14].

5.3 Explainable AI

Finally, the topics considered in this work also overlap with the field of explainable artificial intelligence (XAI). The discipline of XAI aims to improve the interpretability of (deep) neural models by developing explanation techniques. Explanation techniques are used to supply an explanation with each prediction that a model makes, increasing trust in the model, and allowing for the identification of deficiencies in what the model has learned [35].

One example of an explanation technique is LIME [26], an explanation technique that is capable of explaining the predictions of arbitrary classifiers by computing a local approximation of the classifier function, treating the underlying model as a black box. Layer-wise relevance propagation [3] is another explanation technique that outputs the relevance of each pixel of an input image for a classification, which allows human inspection of the resulting heat map.

6 Conclusion

This chapter concludes this thesis by briefly discussing limitations, before concluding with the key findings and insights in Section 6.1. One key limitation of the results presented in this work relates to the small size of the dataset. To ensure that enough participants are available for each file, the evaluation used a subset of the Quixbugs dataset, consisting of 16 files. Due to the diverse nature of software bugs, this likely limits the types of bugs that were covered by the human study.

Furthermore, the complex nature of setting up and instrumenting the reference implementations of APR approaches led to only two approaches being evaluated as part of this work, limiting the available model attention data. This could be alleviated in future work by extracting the attention of existing and upcoming APR approaches and comparing it to the human data presented in this work, which is publicly available.

6.1 Conclusion

In conclusion, this work presents a novel approach for tracking human attention and other behavioral data during code editing task, the Edit Attention Recorder. We conduct a 27-participant human study using EAR to gather attention data of developers fixing bugs out of the Quixbugs dataset, for comparison against attention data gathered from two neural APR approaches, SequenceR and Recoder. The analysis yields the following results: We find that humans still drastically outperform state-of-the-art APR approaches, with the developer group being able to produce at least one correct fix for each file out of the Quixbugs subset. The developers also produce a correct fix in approximately 70% of all fix attempts, in stark contrast to the APR approaches, where only around 1% of the fix candidates are at least plausible. Based on the latter finding, we recommend that future evaluations of APR approaches should also take into account top-k metrics, to push toward practical usability of APR approaches.

Regarding model-human correlation, we find that there is a statistically significant weak to moderate correlation between the attention of SequenceR and the humans, a moderate to strong correlation between humans and Recoder on the overall attention profile, and a weak to moderate correlation on the buggy method only. The high correlation between Recoder and the humans on the overall profile implies that the design of Recoder's attention mechanism, namely ignoring anything but the buggy method and encoding the input program as an abstract syntax tree, results in more human-like attention.

Considering attention on the buggy line, we find that SequenceR pays the majority of its attention to the buggy line, which is likely a result of its design. Recoder pays more attention to

the surrounding context, and the humans are in the mid-field between the two approaches. For the humans, the percentage of attention paid to the buggy line also depends heavily on the input file. On the basis of these results, we suggest that future models for APR could be extended with an additional hyperparameter that controls the proportion of attention paid to the buggy line versus the surrounding context, as both are important for the task of repairing bugs.

Regarding the types of tokens that humans and approaches attend to, we find that the design of Recoder's attention mechanism results in it not seeing keywords and separators, which is in line with how humans at look at code, as they also pay less than uniform attention to these token types. The humans are however alone in paying more than uniform attention to operators, which highlights a possible area for improvement. A possible solution for this could be to pre-assign higher attention scores to token types that are relevant in bug fixing, possibly on the basis of human data.

Finally, we identify two patterns in the behavior of human developers. Firstly, there is a division in a code reading and a code editing phase, according to the interaction events gathered using EAR. Secondly, a significant minority of the developers revise their code between 1 and 4 times, which indicates an iterative bug-fixing strategy.

6.2 Future Work

On the field of automatic program repair, future work could consist of realizing and evaluating the suggestions for improvements put forward throughout this work. In addition, future models for APR could also explore the idea of providing the models with human attention data during training, an idea that has found application in other fields. For example, on the field of visual question answering, Sood et al. [30] set a new state of the art in terms of performance by augmenting training with human attention data.

Another avenue for future work is the further validation of the approach for human attention collection put forward in this work. This could, for example, consist of a comparative study of data gathered using EAR and an existing dataset of eye-tracking information collected during programming tasks, such as EMIP [5], or of a separate parallel eye-tracking and EAR-based experiment.

The reference implementation of EAR supports arbitrary code repair tasks and can be adapted to other code editing tasks with minimal effort. Therefore, a modified version could be integrated into the standard workflow of a development team as part of an industrial study, allowing data on much more complex bugs to be gathered.

Finally, future work could also consist of conducting a more detailed analysis of the attention extracted from the neural models. This work uses averaging over the attention heads and the output sequence to obtain a high-level vector representing what parts of the input sequence the models are paying attention to. Especially for the Transformer architecture, where each attention head is designed to extract different features [34], this results in the loss of detail information. Future work could therefore aim at conducting a detailed analysis of attention vectors for each token and attention head, and comparing those to human reasoning.

To leverage the advancements in explainable artificial intelligence, a similar study to the one in this work could also be done on the basis of one or more explanation techniques, in which the

explanations could be compared to human data collected using EAR. This would allow analyzing models that do not use an attention mechanism and would add another dimension of understanding the models' reasoning for models that do feature one.

A Post-experiment Survey Questions

The post-experiment survey consisted of the following questions, and if applicable, answer options:

1. Nickname
2. Age
3. Gender: Female, Male, Diverse, Prefer not to say
4. Highest level of education: Intermediate secondary education, Secondary education, Bachelor's degree, Master's degree, PhD
5. Occupation: Software Developer / Engineer, Software Tester / QA, Project Lead / Management, Researcher, Other (text box)
6. Employment status: Student, Employee, Freelance / Self-employed, Unemployed, Other (text box)
7. Years of programming experience: 0-1 years, 2-4 years, 5-7 years, 8-10 years, 11+ years
8. Years of Java experience: 0-1 years, 2-4 years, 5-7 years, 8-10 years, 11+ years
9. Acquisition of experience: School education, University education, Self-learning, Apprenticeship or similar, Lateral entry into the field, Internship, Work Experience, Other (text box)
10. Course on algorithms: Yes, No
11. Java in day-to-day work: Daily, 2-3 times per week, Once per week, Once per month, Once per quarter, Once per six months or rarer
12. Debugging frequency: Daily, 2-3 times per week, Once per week, Once per month, Once per quarter, Once per six months or rarer
13. Bug Reports assigned: None, 1-10, 11-50, 51-100, 100+
14. Bug Reports filed: None, 1-10, 11-50, 51-100, 100+

All questions with answer options were single-choice only, except for question 9.

B Subset of Quixbugs

Our dataset consists of the following files out of the Quixbugs dataset [22]:

- FIND_FIRST_IN_SORTED
- MERGESORT
- HANOI
- NEXT_PERMUTATION
- BUCKETSORT
- LEVENSHTEIN
- DETECT_CYCLE
- BITCOUNT
- SQRT
- FIND_IN_SORTED
- POSSIBLE_CHANGE
- MAX_SUBLIST_SUM
- GET_FACTORS
- KHEAPSORT
- KTH

It is important to note that the experimental evaluation uses the improved version of the Quixbugs files introduced in [38], which can be found at `https://github.com/KTH/quixbugs-experiment`.

Bibliography

[1] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani. Compilation error repair: for the student programs, from the student programs. In P. Lago and M. Young, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 78–87. ACM, 2018.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[3] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLOS ONE*, 10:1–46, 07 2015.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[5] R. Bednarik, T. Busjahn, A. Gibaldi, A. Ahadi, M. Bieliková, M. E. Crosby, K. Essig, F. Fagerholm, A. Jbara, R. Lister, P. A. Orlov, J. H. Paterson, B. Sharif, T. Sirkiä, J. Stelovsky, J. Tvarozek, H. Vrzakova, and I. van der Linde. EMIP: the eye movements in programming dataset. *Sci. Comput. Program.*, 198:102520, 2020.

[6] S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 60–70. ACM, 2018.

[7] P. Bielik, V. Raychev, and M. T. Vechev. PHOG: probabilistic model for code. In M. Balcan and K. Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pages 2933–2942. JMLR.org, 2016.

[8] T. Busjahn, R. Bednarik, A. Begel, M. E. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm. Eye movements in code reading: relaxing the linear order. In A. D. Lucia,

C. Bird, and R. Oliveto, editors, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, pages 255–265. IEEE Computer Society, 2015.

[9] J. Castelhano, I. C. Duarte, C. Ferreira, J. Duraes, H. Madeira, and M. Castelo-Branco. The role of the insula in intuitive expert bug detection in computer code: an fMRI study. *Brain Imaging and Behavior*, 13(3):623–637, may 2018.

[10] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[11] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. Software Eng.*, 47(9):1943–1959, 2021.

[12] A. Das, H. Agrawal, L. Zitnick, D. Parikh, and D. Batra. Human attention in visual question answering: Do humans and deep networks look at the same regions? In J. Su, X. Carreras, and K. Duh, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016*, pages 932–937. The Association for Computational Linguistics, 2016.

[13] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[14] K. Fukushima and S. Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognit.*, 15(6):455–469, 1982.

[15] C. L. Goues, M. Pradel, A. Roychoudhury, and S. Chandra. Automatic program repair. *IEEE Softw.*, 38(4):22–27, 2021.

[16] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. Deepfix: Fixing common C language errors by deep learning. In S. P. Singh and S. Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 1345–1351. AAAI Press, 2017.

[17] N. Jiang, T. Lutellier, and L. Tan. CURE: code-aware neural machine translation for automatic program repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 1161–1173. IEEE, 2021.

[18] R. Just, D. Jalali, and M. D. Ernst. Defects4j: a database of existing faults to enable controlled testing studies for java programs. In C. S. Pasareanu and D. Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 437–440. ACM, 2014.

[19] R. Karampatsis and C. Sutton. How often do single-statement bugs occur?: The manysstubs4j dataset. In S. Kim, G. Gousios, S. Nadi, and J. Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 573–577. ACM, 2020.

[20] R. M. Karampatsis and C. Sutton. Manysstubs4j dataset, Feb. 2020.

[21] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. Rush. OpenNMT: Open-source toolkit for neural machine translation. In *Proceedings of ACL 2017, System Demonstrations*, pages 67–72, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[22] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In G. C. Murphy, editor, *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, pages 55–56. ACM, 2017.

[23] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: combining context-aware neural translation models using ensemble for program repair. In S. Khurshid and C. S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 101–114. ACM, 2020.

[24] M. Paltenghi and M. Pradel. Thinking like a developer? comparing the attention of humans with neural models of code. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 867–879. IEEE, 2021.

[25] J. A. Prenner and R. Robbes. Automatic program repair with openai's codex: Evaluating quixbugs. *CoRR*, abs/2111.03922, 2021.

[26] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should I trust you?": Explaining the predictions of any classifier. In B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen, and R. Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 1135–1144. ACM, 2016.

[27] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In A. Zaidman, Y. Kamei, and E. Hill, editors, *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 10–13. ACM, 2018.

[28] A. See, P. J. Liu, and C. D. Manning. Get to the point: Summarization with pointer-generator networks. In R. Barzilay and M. Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, pages 1073–1083. Association for Computational Linguistics, 2017.

[29] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In E. D. Nitto, M. Harman, and P. Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 532–543. ACM, 2015.

[30] E. Sood, F. Kögel, P. Müller, D. Thomas, M. Bace, and A. Bulling. Multimodal integration of human-like attention in visual question answering. *CoRR*, abs/2109.13139, 2021.

[31] E. Sood, S. Tannert, D. Frassinelli, A. Bulling, and N. T. Vu. Interpreting attention models with human visual attention in machine reading comprehension. In R. Fernández and T. Linzen, editors, *Proceedings of the 24th Conference on Computational Natural Language Learning, CoNLL 2020, Online, November 19-20, 2020*, pages 12–25. Association for Computational Linguistics, 2020.

[32] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, pages 3104–3112, 2014.

[33] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4):19:1–19:29, 2019.

[34] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[35] F. Xu, H. Uszkoreit, Y. Du, W. Fan, D. Zhao, and J. Zhu. Explainable AI: A brief survey on history, research areas, approaches and challenges. In J. Tang, M. Kan, D. Zhao, S. Li, and H. Zan, editors, *Natural Language Processing and Chinese Computing - 8th CCF International Conference, NLPCC 2019, Dunhuang, China, October 9-14, 2019, Proceedings, Part II*, volume 11839 of *Lecture Notes in Computer Science*, pages 563–574. Springer, 2019.

[36] D. Yang, Y. Qi, X. Mao, and Y. Lei. Evaluating the usage of fault localization in automated program repair: an empirical study. *Frontiers of Computer Science*, 15:151202, 09 2020.

[37] Z. Yao, F. F. Xu, P. Yin, H. Sun, and G. Neubig. Learning structural edits via incremental tree transformations. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[38] H. Ye, M. Martinez, and M. Monperrus. A comprehensive study of automatic program repair on the quixbugs benchmark. *CoRR*, abs/1805.03454, 2018.

[39] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 341–353. ACM, 2021.

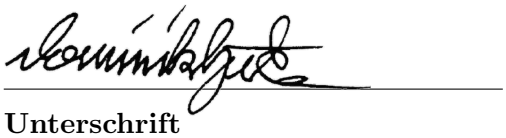
[40] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2021.

Selbsständigkeitserklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

02.05.2022

Datum



Unterschrift