

Machine Learning Techniques in Software Repair and Environmental Optimization

PhD Thesis

Dániel Horváth

Supervisor: László Vidács, PhD

Doctoral School of Computer Science

Department of Software Engineering

Faculty of Science and Informatics

University of Szeged



Szeged

2026

Contents

Introduction	5
Background	7
Traditional <i>Artificial Intelligence</i> Approaches	7
Modern <i>Deep Learning</i> Methods	8
<i>Neural Networks</i>	8
Transformers and Attention Mechanisms	8
Embeddings and Code Representation	9
Cross-Domain <i>Machine Learning</i> Applications	9
<i>Automated Program Repair</i> (APR) Applications	9
Regression Modeling and Scientific Computing	10
Contributions	11
1 Smarter Repair Pipelines for JavaScript APR	13
1.1 Introduction	13
1.2 Motivation	16
1.3 Related Work	17
1.4 Technical Approach	19
1.4.1 Genetic Algorithm	19
1.4.2 Patch Correctness of Plausible Candidates	21
1.4.3 Operators	27
1.5 Experiment Setup	28
1.5.1 BugsJS: An Executable JavaScript Dataset	28
1.5.2 Other Datasets on JavaScript	30
1.5.3 Parameter Settings	31
1.5.4 LLM-based Repair	32
1.6 Results	33
1.6.1 Repaired Bugs Using GPT-2	37
1.6.2 An Example: Express 2	40
1.6.3 Repairing on Script Language	41
1.6.4 Multi-line repairs	43

1.6.5	Insights Into the Searching Process	44
1.6.6	Operator Expedience	45
1.7	Threats to Validity	47
1.8	Discussion and Concluding Remarks	48
2	Representation Matters: <i>Deep Learning</i>-Based Program Repair with Effective Code Encodings	51
2.1	Introduction	51
2.1.1	Motivation	51
2.2	Related Work	54
2.3	Method	56
2.3.1	<i>Deep Learning</i> Models	56
2.3.2	Program Representation	58
2.4	Experimental Setup	59
2.4.1	Training Data	61
2.5	Need for Datasets	62
2.6	Mining	62
2.7	Results	65
2.7.1	Dataset Overview	66
2.7.2	Dataset Limitations	68
2.7.3	Code Representations	68
2.7.4	Empirical Evaluation and Examples	71
2.8	Discussion and Concluding Remarks	75
3	Green Energy Through ML: Optimizing CO₂ Electrolyzer Cell Performance	77
3.1	Introduction	77
3.2	Related Work	79
3.3	Method	80
3.4	Model Selection	85
3.5	Results	88
3.5.1	Evaluation on Test Set	88
3.5.2	Parameter Optimization	90
3.5.3	Evaluation on External Set	91
3.5.4	Uncertainty Analysis	94
3.5.5	Analysis of Possible Feature Correlations	103
3.5.6	Model Sensitivity Analysis	105
3.6	Discussion and Concluding Remarks	108
	Summary	111
	Smarter Repair Pipelines for JavaScript APR	111
	Contributions of the thesis	111

List of Figures

1.1	High-level overview of GenProgJS components.	16
1.2	Illustration of the process of measuring patch similarity.	23
1.3	The developer fix and patches ranked based on their similarity to the original program.	25
1.4	The nDCG metric value based on each of the examined bugs' ranked list.	26
1.5	Generated number of candidates from the 5 independent runs.	32
1.6	Probabilities of JavaScript operator usages.	35
1.7	Trace of operators applied on Express 2.	41
1.8	Operator usage per fix type.	46
2.1	High-level approach overview of source code representation experiments.	52
2.2	Overview of data mining procedure	63
2.3	AST node position extraction.	64
3.1	Aiding CO ₂ reduction processes via <i>Machine Learning</i> algorithms. . . .	78
3.2	Searching the optimal values for the operation of the CO ₂ electrolyzer cell.	82
3.3	Parameter setting combinations (left) and the measured results (right) for the electrolysis experiments.	83
3.4	Model performance mean values on the test set.	87
3.5	Model prediction results on the test set.	89
3.6	Flow chart of the NSGA-II algorithm.	91
3.7	Set parameters of the external test.	92
3.8	Measuring cell operation outputs on the external test set parameters. .	93
3.9	Prediction results on the external test set.	94
3.10	Uncertainty in the data (left) and the model predictions (right) on the full development dataset.	95
3.11	Uncertainty in the data (left) and the model predictions (right) on the test set.	96
3.12	Uncertainty in the data and the model predictions on the external test set.	100

- 3.13 Correlation between different input parameters in the model development dataset. Meas ID signifies the repeated measurement sequences used for model development, with the 0th ID corresponding to the first, and the 1st ID corresponding to the repeated measurements. The diagonal in the plot shows the distribution of the different input feature values, while the scatter plots show the individual measurement points sample-by-sample. These scatter plots help visualize obvious patterns of correlations between the input features, if any. 104
- 3.14 Local SHAP-based feature importance on the left, and global SHAP-based feature importance on the right. 107

List of Tables

1	Publications supporting the thesis points.	12
1.1	Plausible patches and their corresponding developer fix in the Eslint project.	24
1.2	Operator comparison of Astor (Java) and GenProgJS (JavaScript). . . .	27
1.3	Mutation operators of GenProgJS.	29
1.4	Systems contained by the BugsJS dataset.	30
1.5	Results of test-suite-based program repair tools.	36
1.6	Repairs on the BugsJS dataset produced by GenProgJS.	38
1.7	Repairs on the BugsJS dataset produced by the fine-tuned GPT-2 model.	39
1.8	Bugs and their corresponding plausible patches.	42
2.1	Examples of program representations.	60
2.2	Metrics describing the dataset.	66
2.3	Results on training different models on various data representations. .	69
2.4	Comparing results on Java and FixJS dataset on text and command sequence token representation.	71
3.1	Parameter settings used for the electrolysis experiments.	83
3.2	Correspondence between control- and model input features.	84
3.3	Artificial Neural Network, Random Forest Regressor, Linear Regressor, and XGBoost Regressor model predictions on the test, and external test set.	86
3.4	Comparing ANN predicted and true mean values for individual measurement sequences for each parameter setting on the test set.	98
3.5	Comparing ANN predicted and true std values for individual measurement sequences for each parameter setting on the test set.	99
3.6	Comparing ANN predicted and true mean values for individual measurement sequences for each parameter setting on the external test set.	102
3.7	Comparing ANN predicted and true std values for individual measurement sequences for each parameter setting on the external test set. . .	103
3.8	Correlation between input features, numerically representing Figure 3.13.	105

Introduction

Machine Learning (ML) is a still emerging and expanding research field that has a wide range of scientific and industrial applications. Although this study focuses mainly on the scientific side, it is evident that ML can be used in many real life tasks, ranging from autonomous driving to offering financial advice or providing targeted advertisements. It is undeniable how much ML has become part of our lives, making it a highly researched topic to this day.

Among its most impactful research fields is *Automated Program Repair* (APR), where ML techniques being used to automatically identify and rectify software bugs found in source code. The purpose of APR is to reduce human intervention by providing automatic fixes to software bugs, or to detect faulty or vulnerable code early during development, potentially saving a lot of time and effort. One particularly successful solution —originally designed for *Natural Language Processing* (NLP) tasks— is the transformer-based architecture. Transformers has shown promising results in learning patterns of buggy and fixed code versions. Such models, *e.g.*, CodeBERT [50], or CodeT5 [173] have demonstrated their ability to generate accurate patches and understand code context by training on a large scale of code repositories. Transformer models are well known for their capabilities on capturing long-range dependencies and syntactic features, which makes them an adequate choice for learning fix patterns in more complex and real-life systems. Alas, these models also have their particular shortcomings. They are highly dependent on the quality of training data, do not really "understand" concepts like humans do, and are also capable of generating nonsensical responses, which sometimes are called *hallucinations*. They are also usually very costly to train, and even fine-tuning them requires tremendous computational capacity, making them out of reach for many possible users.

In parallel, *Genetic Algorithms* (GAs) like GenProg [174] or Kali [141], have also shown promise in repairing certain kinds of software bugs. They work by mutating the buggy source codes via the so-called *operators*, which act like code transformation recipes. Every operator takes the code and mutates it slightly by transforming it in a specific way (*e.g.*, replacing a variable name with another random one). This in turn can produce a modified version of the source code, which can then be validated against existing test cases. GAs are more readily available to use due to their much

lower requirements, however, they are not without faults either. If the search space is vast, then mutation or crossover operations can take a significant amount of time to produce plausible patches, which again are not guaranteed to be "real" fixes for a specific bug. Hybrid approaches such as DeepRepair [192] also exist that try to mitigate these issues applying ML to suggest operator selection, instead of choosing randomly. This in turn can more accurately target specific operators that are more likely to produce a bug-fix.

Beyond *Software Engineering* (SE), ML can also be of use in other scientific domains ranging from biology to the creation of more sustainable alternative energy resources. Models like AlphaFold [85] have helped researchers leap many years into the future by allowing very accurate protein folding predictions, which can significantly speed up drug and molecular biology research. Similarly, *Machine Learning* is also being applied to CO₂ reduction technologies. One particular field of interest is the reduction of CO₂ through electrochemical cells, which can effectively reduce undesired greenhouse gases, while also producing highly desirable by-products, such as CO, which later can be readily used in the (petro) chemical value chain.

These applications show the versatility of ML tools and their use cases. Research of ML is capable of bringing about solutions to previously unsolved problems, making it a worthwhile topic to investigate, as it can advance many research fields to this day.

This dissertation explores the versatility of *Machine Learning* across three distinct domains: (1) traditional evolutionary approaches for program repair, (2) modern *Deep Learning* representations for code understanding, and (3) AI-guided optimization of physical systems. Together, these studies demonstrate how different ML paradigms able address complex problems in *Software Engineering* and even different scientific domains.

Background

The field of *Machine Learning* (ML) is expanding ever so quickly, bringing results to unanswered questions in various domains. The significance of *Artificial Intelligence* (AI) and ML may have never been this high as it can solve problems ranging from autonomous self-driving cars to problems like protein folding. This dissertation makes an attempt to explore and show topics much related to ML, especially *Deep Learning* (DL) models that can support tasks in fields like *Automated Program Repair* (APR), and electrochemical system optimization. The following section dives into various topics related to this work that may help to better understand the theme as a whole.

Traditional *Artificial Intelligence* Approaches

Machine Learning is a subfield of *Artificial Intelligence* that focuses on systems that learn patterns and hidden relationships in data. They usually do this by using purposefully designed algorithms to optimize for a specific task or tasks at hand. These methods usually offer concrete algorithms such as decision trees, *Neural Networks* (NNs), or combination of these models (e.g., ensemble models), while AI is more of a collection of tasks to make software or machine behave intelligently, *i.e.*, trying to mimic human behavior.

Traditional *Automated Program Repair* (APR) approaches have relied heavily on heuristic-based systems that use predefined templates or mutation strategies to generate candidate patches, which are then validated against test suites [122]. These methods take advantage of the power of search-based algorithms. Particularly *Genetic Algorithms* (GAs) and *Genetic Programming* (GP), to systematically explore the space of possible program modifications. The GA approach treats program repair as an optimization problem, where candidate solutions evolve through selection, crossover, and mutation operations until a suitable candidate is discovered.

The strength of these traditional approaches lies in their transparent behavior. Unlike black-box methods, heuristic-based repair systems provide clear traces of how patches are generated and why certain modifications are preferred over others. This is a particularly compelling aspect of such approaches, as they can give a clear explanation on how a patch is generated.

Modern *Deep Learning* Methods

Throughout all the thesis points that will follow, they all depend heavily on the predictive and generative power of the ML tools, especially in APR. Modern DL approaches have revolutionized how we approach complex pattern recognition and generation tasks.

Neural Networks

A good starting point for discussing ML algorithms would be *Neural Networks* (NNs). NN is such a construction that tries to mimic the human brain by defining many interconnected neurons capable of learning complex patterns from raw data. These neurons form the so-called layers of the *Neural Network*. The first layer of neurons is often called the input layer, whereas the last layer is called the output layer. Depending on how many hidden layers exist between the input and output layers, we can differentiate between shallow *Neural Networks* and *Deep Neural Networks*.

- **Neurons:** The basic units of *Neural Network*.
- **Connections:** Links between neurons that carry information, regulated by weights and biases.
- **Weights and Biases:** These parameters determine the strength and offset of the connections.

In *Software Engineering* NNs have been used for tasks such as defect detection and patch generation [176].

Transformers and Attention Mechanisms

Transformers are a class of DL models that rely on the Attention Mechanism [169] to process sequential input without the need to use recurrent neural units. Transformer-based models became the de facto tool for *Natural Language Processing* (NLP) as they showed promising results. Since then, transformers have been adapted to handle diverse tasks, such as understanding and generating code [50]. Their capability to capture long ranges of dependencies using the attention mechanism. This in turn also makes them suitable for learning source code.

Unlike the traditional *Recurrent Neural Networks* (RNNs) that process sequences step by step, transformers are able to account for every word (or token) at once. To determine which words play the most significant role in generating a given token, transformers use the attention mechanism by assigning scores to each part of

the sequence. With this global view of the input together with the attention mechanism, transformer-based methods have the ability to understand complex relationships throughout the input.

This makes them particularly useful for tasks such as program repair, where understanding the context is essential. Their ability to capture long-range dependencies allows models to recognize, for example, how a variable declared at the top of a file might influence a bug several functions below.

Embeddings and Code Representation

Embeddings map discrete inputs, such as tokens or nodes, to continuous vector spaces, capturing semantic relationships. Creating high-quality word and code embeddings [6, 119] is an important step for ML models to generate predictions. In code repair tasks, the choice of embeddings can significantly affect model performance, as they determine how to interpret the embedded code.

Structured formats like *Abstract Syntax Trees* (ASTs), control-flow graphs, and token sequences are commonly used [5] in many APR fields. These representations preserve syntactic and semantic features, enabling ML models to reason about code behavior.

Cross-Domain *Machine Learning* Applications

The wide range of applications of ML techniques becomes apparent when one considers their applicability in various domains. From software engineering to scientific computing, modern ML approaches show their outstanding capabilities to solve complex real-world problems.

Automated Program Repair (APR) Applications

Automated Program Repair (APR) is a subfield of *Software Engineering* that focuses on developing systems that can effectively identify and repair bugs with minimal or no human intervention. Traditional software debugging is time-consuming and error-prone, prompting the need for automated solutions, especially in large-scale, continuously evolving codebases.

APR techniques generally fall into two categories: heuristic-based and learning-based approaches. In contrast to the traditional heuristic methods discussed earlier, learning-based methods take advantage of the large volume of readily available code bases. To take it a step further, these models could later also be fine-tuned on buggy-fixed code versions to generate repaired programs [122, 168].

Recent advances in deep learning have significantly improved APR capabilities. By representing source code as sequences, trees, or graphs, modern APR systems can capture both syntactic and semantic information, allowing more accurate patch generation. Models trained on code repositories can generalize across programming languages and bug types, making APR systems more adaptable and scalable.

In this dissertation, APR is explored through two key contributions: improving repair pipelines for JavaScript (a widely used and dynamically typed language) and investigating how source code representations affect the performance of deep learning-based repair models.

Regression Modeling and Scientific Computing

For scientific applications, NNs offer data-driven modeling capabilities, making them suitable for various tasks. This task can be the performance optimization of CO₂ emissions. One promising method is the reduction of CO₂ via electrolyzer cells producing by-products that are harmless or can be further used by other industrial processes. A recently emerging technique to model such continuous real-life systems is ML-based regression methods, such as random forests, support vector regressors, and neural networks, which have been used to model complex phenomena in chemistry, materials science, and energy systems [18].

Electrochemical reduction of CO₂ is a high potential and actively researched topic that can help environmental sciences move toward carbon-neutral energy systems. Although optimizing such systems involves many interdependent variables, if enough high-quality data are present, ML models can basically act as a digital twin of the original system in simulation or experimental design loops, allowing accelerated discovery [153]. AI-guided optimization thus enables more efficient exploration of the design space, as pursued in this thesis through predictive modeling of CO₂ electrolyzers.

Contributions

The ideas, figures, tables, and results included in this dissertation were published in scientific papers (listed after the table of contents, in Sections List of Figures and List of Tables). All in all, the author is responsible for the following contributions.

Thesis 1: The author implemented genetic and mutation operators for the GenProgJS framework. Novel JavaScript operators are also introduced by GenProgJS, and are also attributed to the author. In addition to this, he was also responsible for the collection of bug types for which a specific operator or a set of operators was able to produce a patch. The author also took a significant part in the evaluation process of these operators, providing examples that demonstrate their capabilities in a JS-based environment. Furthermore, the evaluation of generated patches is also part of the contribution that is attributed to the author, as providing a method and implementation to select plausible candidate patches via Doc2Vec was also part of this research.

Thesis 2: Experiments carried out showing how different models perform on different datasets with multiple programming languages and diverse source code representations are directly attributed to the author. The results show that choosing the right representation for a specific task plays a crucial role in the performance of the model. The author is also responsible for providing the foundation dataset on which LLMs such as GPT-2 could be fine-tuned. He also provided a programming language agnostic method for extracting specific segments of source code, that can help researchers of the field to fine-tune their models on custom datasets. To showcase this, the author also provided a publicly available dataset hosted on Zenodo.

Thesis 3: The AI-guided framework for optimizing the operation of CO₂ electrolyzer cells is entirely the contribution of the author. He implemented the entire pipeline, from processing the measurement data in a way that ML models can consume and train on, to optimizing the system as a whole to find the best parameters for optimal cell operation.

Table 1: *Publications supporting the thesis points.*

	Publications					
	[29]	[27]	[93]	[73]	[74]	[151]
I.	✓	✓	✓			
II.			✓	✓	✓	
III.						✓

Roman numbers in the first column represent the number of the thesis point the different publications belong to, the other columns (separated under Publications column) show which publication supports a specific thesis point.

Publications not included in the dissertation

- András Kicsi, Péter Sánta, Dániel Horváth, Norbert Kőhegyi, Viktor Szvorenny, Veronika Vincze, Eszter Főző, and László Vidács. "Computer-Aided Forensic Authorship Identification in Criminology." In International Conference on Computational Science and Its Applications, pp. 576-592. Cham: Springer International Publishing, 2022.
- Viktor Csuvik, Dániel Horváth, and László Vidács. "Feature extraction, learning and selection in support of patch correctness assessment." In Proceedings of the 19th International Conference on Software Technologies-ICSOF, vol. 124. 2024.
- Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. "Utilizing source code embeddings to identify correct patches." In 2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF), pp. 18-25. IEEE, 2020.

Chapter 1

Smarter Repair Pipelines for JavaScript APR

1.1 Introduction

Automated Program Repair (APR) has developed greatly in recent years, with tools showing remarkable performance in numerous programming languages. But there is one crucial area that has not yet been explored: JavaScript web applications. Although JavaScript (JS) is the most popular language used for both client-side and server-side web applications, current APR methods mostly ignore its specific issues. Our work addresses this gap with the introduction and evaluation of GenProgJS, a genetic search-based APR tool specifically designed for use with JavaScript. This enables us to explore the use of test-suite-based repair methods within the field of the web's most widespread language and how the advances that have occurred in machine learning and hybrid systems over recent years can be leveraged to improve JavaScript APR.

Most of today's *Automated Program Repair* (APR) tools are implemented in such a way to repair programs written in C, Java, or even Python. Since the appearance of GenProg [174] and its genetic approach, many excellent researchers have tried to improve its performance by creating several distinct APR tools [65, 110, 111, 114, 141, 168, 185]. Many of these follow the *Generate and Validate* (G&V) approach [63], meaning; a patch is generated first and then the test suite is executed to check the correctness of the generated candidate. However, the previously mentioned programming languages are frequently used for desktop and android applications, or in research projects; their use is not widespread in web development. Python took more than a decade to surpass JavaScript as the most widely used language [159]. It has become the go-to web programming language and the most used language on GitHub based only on code pushes alone [56]. JavaScript is mostly used on the client side as a scripting language for web applications to achieve high responsive-

ness and user friendliness. In recent years, due to its flexibility and effectiveness, it has also been increasingly adopted for server-side development, leading to full-stack web applications based on JavaScript [81].

In order to fill the gap between JavaScript and the APR field, we presented and evaluated GenProgJS, a general automated repair tool that generates repairs for real-world JavaScript bugs. To our knowledge, this is the first APR tool that uses a genetic algorithm to generate patches for JS. It is built in a way that the progress of patch creation is well traceable, making it possible to further analyze the usefulness of operators and meta-parameters. This tool is an implementation of GenProg [97] for JavaScript, with additional language-specific improvements. The tool integrates with the BugsJS benchmark [66], which contains bugs from 10 popular open-source projects based on JavaScript language. Thanks to BugsJS API, GenProgJS could be evaluated on real-world bugs and compared to developer patches. We found that GenProgJS is capable of successfully fixing 31 bugs in six different projects, making it a promising baseline for future research on JavaScript APR.

Despite the popularity of JavaScript, it lacks general source code analyzer tools [49] compared to Java or C / C++; thus it is understandable why JS is missing from the list of typical languages in which APR tools operate, as JS language includes several challenges not present in other typically used programming languages. These challenges include higher-order functions, dynamic code evaluation and execution, weak typing, dynamic field access, and asynchronous control flow [81, 131, 139, 147]. On the other hand, JavaScript is designed to interact with an external entity known as the *Document Object Model* (DOM). Using DOM API methods, JavaScript is capable of retrieving elements from an HTML document and changing the DOM by modifying elements. Due to the dynamic nature of the JavaScript language and the interaction with the DOM, JavaScript-based web applications are prone to errors [129].

Recently, the field of APR has been significantly transformed by the rapid advancements in *Machine Learning*, particularly through the adoption of *Large Language Models* (LLMs) such as Codex [23], CodeT5 [173], and GPT-based architectures. By using an enormous number of code repositories for pretraining, these models can understand patterns and produce source code in a variety of languages. In addition to generating and finishing code, LLMs have shown remarkable results in detecting and fixing problematic code with little help from humans. The authors of a recent study [88] evaluated the program repair capabilities of numerous APR tools using a manually curated benchmark. Although the study's goal was to analyze all state-of-the-art methods, the authors discovered that learning-based approaches are frequently unusable due to their availability (e.g., confidentiality issues in the case of ELIXIR [150]), executability (e.g., certain tools require a specific execution environment), or configurability (e.g., additional data required from GitHub). Similar

requirements are outlined in the papers of Bui *et al.* (2023) [16] and Yang *et al.* (2022) [187]: the APR tool must be (1) publicly available, (2) have a high repair ability, and (3) be deterministic. They then eliminated learning-based methods and concentrated more on heuristic- and semantic-based methods in their experiments with various objectives. As a result, test-suite-based tools continue to be an important contributor in APR approaches due to their transparency, explainability, and ease of use [102].

Furthermore, hybrid tools are emerging that combine LLMs with symbolic techniques or genetic algorithms to harness the strengths of both paradigms. These tools offer more flexible repair strategies that balance interpretability, search efficiency, and semantic correctness. Some systems integrate source code embeddings to measure patch similarity or even prioritize candidate fixes [28], enhancing the patch selection process beyond simple test-based validation. To validate the generated patches, an oracle is needed which can reliably determine if the repair is really correct and can detect over- or under-fitted patches. Test suites are widely used as affordable approximations of this oracle in the APR field; therefore, (re-)running the tests validates whether the repair process was successful or not. In our study, a program variant is considered to be a plausible patch if it passes more tests, or at least as many test cases as the developer’s fixed version. The latter condition gives no assurance that the program is *correct*, since overfitting and underfitting [96, 141, 156] often occurs, resulting in inadequate patches. This work also looks at the issue of finding the most promising patch among the so-called mass-generated patches and provides insight into the usability of existing techniques on patch similarity assessment applied on JavaScript language.

Given these advancements, GenProgJS represents a complementary direction: it provides a traceable, *Genetic Algorithm*-based repair tool for JavaScript, in contrast to black-box ML-based methods. While LLM-powered tools continue to improve, their dependency on large training datasets and limited interpretability remain open challenges, not to mention their especially high training costs [71]. By combining the transparency of GenProgJS with insights from modern ML techniques, future research can develop hybrid APR systems that are both effective and explainable.

GenProgJS is also available in our public repository ¹ and has a website hosted by GitHub Pages ². A preconfigured Docker environment is also available to run and analyze the trace of the algorithm, and selected operators effortlessly. At this time, GenProgJS is limited to running bugs included in the BugsJS benchmark.

¹<https://github.com/GenProgJS>

²<https://genprogjs.github.io/GenProgJS/>

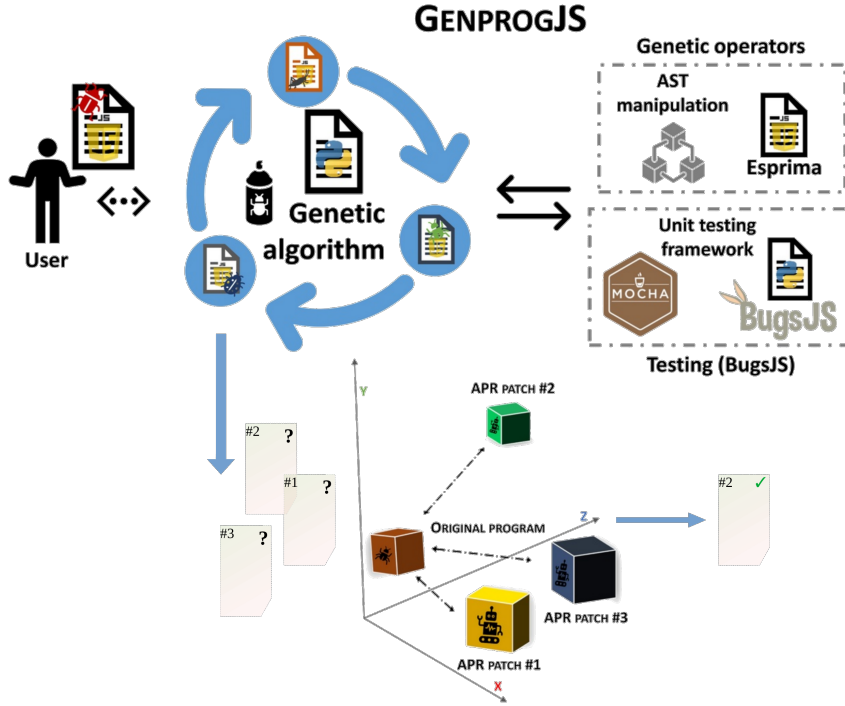


Figure 1.1: High-level overview of GenProgJS components.

1.2 Motivation

Software bugs are expensive because of the manual work that is needed to fix them. Unfortunately, errors in the code can be found in any software. Automatic fixation is an attractive idea to create a solution to fix bugs, as it reduces maintenance costs and improves the quality of software [97]. In recent years, there has been significant progress in this field [53, 122], moving from solutions for specific problems to broader methods such as genetic programming and machine learning.

Many leading *Automated Program Repair* (APR) solutions today focus on Java systems, such as Astor, which is an advanced repair framework for Java [114]. This has reduced the previous dominance of C/C++ solutions, in part by redoing their concepts. A detailed online list of repair tools shows that Java and C/C++ tools are very common³; however, tools can also be found for Eiffel, Python, and multiple languages. Despite JavaScript being one of the most popular programming languages for many years on GitHub [56] and Stack Overflow [159], there is still a shortage of general JavaScript repair tools (there are none currently listed above). The goal of GenProgJS was to create the first general solution for JavaScript, thus supporting APR research by providing a starting point for the community.

³<https://program-repair.org/tools.html>

Let us consider the following code snippet from the Karma project, a JavaScript testing environment built on the Node.js framework:

Code 1.2.1: Missing input validation problem and its human repair from the Karma project.

```
var file = findFile(path)
- if (file && file.sourceMap) {
+ if (file && file.sourceMap && line) {
  line = parseInt(line || '0', 10)
  column = parseInt(column, 10)
```

In the example of Code 1.2.1, the developers addressed a missing input validation issue by modifying a conditional statement to include a check for the `line` variable. Despite its brevity, the patch is not necessarily straightforward: resolving this issue took substantial discussion, stretching from December 2014 to February 2016 on GitHub. Note that many advanced repair techniques today still focus on similarly concise bugs, often involving a single-line change to the source code. Previous studies underscore the significance of these so-called "one-liner" fixes: they are not only prevalent [87], but can also have serious consequences, such as crashes or security vulnerabilities (e.g., due to inadequate input validation). Although some of these fixes may appear trivial or target noncritical faults, the consensus is that they are consistently worth addressing [99]. In the long term, even small patches can yield substantial reliability improvements. Still, certain issues require more extensive changes, leading to multiline (i.e., spanning several lines) or multihunk (i.e., spread across multiple code locations) modifications. To address these issues and to pave the way for JavaScript APR, we developed several JS-specific operators for use in GenProgJS.

1.3 Related Work

There have been several implementations for Java and C programming languages aimed at automatically repairing programs. GenProg [174] was one of the first tools to perform fully automatic fixes with relatively good results. Originally developed for C, it has since been ported to Java [111]. GenProg uses algorithms based on *Genetic Programming* (GP) to repair programs without the need for formal specifications. PAR is another well-known tool that uses synthesis based on human-written patches [90], applying 10 predefined templates to repair Java source code. Deep-Fix takes a different direction, using a multilayer neural network to treat the buggy program as a token sequence and generate the corrected version accordingly [65].

Some tools specialize in particular error types. For instance, Nopol improves con-

ditional control structures (*i.e.*, if-then-else statements) by modifying conditions or introducing guard clauses [185]. Kali, on the other hand, deletes or skips code to create patches, not a true repair technique, but a useful tool to identify weak test cases [141]. A more recent tool, Figra [51], improves the precision of the repair by reusing the code of the repair program in a search-based manner and outperforms several state-of-the-art approaches in benchmarks. The Astor framework [112] emerged as a way to unify multiple Java repair strategies, incorporating implementations of GenProg, Kali, Cardumen, and others.

Although there are numerous general-purpose tools for Java and C, a comparable repair framework for JavaScript remains underexplored. A notable recent attempt is Mutode [148], a generic mutation testing tool for JavaScript. Although many of its mutation operators are generic and not JS-specific, it represents a step toward systematic mutation-based repair in this language. Some earlier work has touched on JavaScript repair indirectly: Jensen *et al.* (2012) [81] transformed common uses of *eval* into safer constructs, and Selakovic *et al.* (2015) [154] tackled performance bugs through static patterns. However, these are not general-purpose APR tools.

Studies such as Kechagia *et al.* (2022) [88] provide critical information on the strengths and limitations of existing repair tools when applied to API misuses. Their evaluation in 14 Java tools found that only a fraction of the generated patches were semantically correct, underlining the challenge of correctly localizing and fixing complex bugs. Meanwhile, Hanna *et al.* (2023) [68] explored reinforcement learning to optimize mutation operator selection in heuristic-based repair systems, achieving notable improvements in bug fix success rates.

Evaluation remains a cornerstone of APR research. Several works have highlighted the overfitting problem in test-suite-based repair systems, where plausible patches may still be incorrect [121, 156]. Liu *et al.* (2021) [102] proposes eight evaluation metrics and critiques current practices, while Wang *et al.* (2020) [172] benchmarks correctness assessment techniques, advocating for combinations of static features and dynamic validation. Embedding-based ranking approaches are increasingly popular; for example, Tian *et al.* (2020) [164] compares Doc2Vec, BERT, code2vec, and CC2Vec for patch similarity scoring, while Csuvik *et al.* (2020) [28] demonstrate that natural language embeddings can help filter 45 % of incorrect patches on QuixBugs.

Some JavaScript-specific APR tools exist but are limited in scope. Vejovis [130] targets DOM interaction errors by rewriting faulty strings, while BikiniProxy [41] performs runtime HTML/JS patching via an HTTP proxy. SAFEWAPI [11] similarly targets API misuse, but focuses on parameter corrections. These approaches solve niche problems rather than offering general-purpose solutions.

Many works stress the importance of generating simple and understandable patches, as complexity can hinder maintainability [54, 117, 167, 175]. Patches that

differ syntactically but are semantically valid still pose maintenance challenges. The so-called "alien code" [122] performs the same operations as the developer fixes, although it usually does it in a roundabout way (e.g., inserting `1==0` instead of `false`). However, studies have shown that a significant portion of correct machine-generated patches differ from those written by developers [171]. Semantic approaches such as Angelix [118] and multi-hunk repair tools like Hercules continue to push the field forward. The evaluation of patch correctness remains an open issue [9, 53], recent work exploring code similarity, test enhancement, and fuzzing as possible solutions [27, 32, 163, 181, 188].

Data-driven repair methods have become increasingly prominent in recent years. Tools such as SequenceR [25], DLFix [190], CoCoNuT [109], and CURE [84] rely on training large datasets to synthesize patches. Of particular interest is Hoppity [37], a graph-based neural network that predicts AST transformations for JavaScript code. Unlike GenProgJS, which is test-suite-driven and does not require pretraining, Hoppity relies on a large dataset of 290,715 code change commits. Transformer-based architectures now dominate this subfield [91, 116], although finding the right representation for the source code remains an ongoing research challenge [74, 125].

Rather than claiming to introduce a fundamentally new APR approach, our work offers a simple and accessible test-suite-based repair tool for JavaScript, with the objective of supporting the research community in this field.

1.4 Technical Approach

GenProgJS is built around three main components: (1) a genetic algorithm responsible for managing the population, selection, *etc.* (detailed in Section 1.4.1); (2) a set of transformation operators implemented in TypeScript using the Esprima library [47], discussed in Section 1.4.3; and (3) an integrated testing framework that takes advantage of the BugsJS dataset (see Section 1.5.1). Each candidate solution in the population is represented as an *Abstract Syntax Tree* (AST), which allows for fine-grained and structured code modifications. In contrast to Java, where repairs are typically derived at the class level, JavaScript fixes in this system are extracted directly from the buggy source file.

1.4.1 Genetic Algorithm

GenProgJS extends the GenProg methodology, introducing several new operators designed to better suit the JavaScript programming language. The structure of the system aligns with the standard workflow of a genetic algorithm, as described in Algorithm 1.4.1. The process begins by initializing the population ($popul_0$), which involves loading the buggy program (P) into memory. The main loop then runs

until it reaches a predefined number of generations (MG), specified as a configurable parameter of the software.

At the heart of each iteration —starting at line 4 in the form of a while loop— the algorithm calculates the fitness values, which are being computed using a specific formula given by Equation 1.1, where P is a program variant, and W_{pos} and W_{neg} are weights for passed and failed tests, respectively. The actual parameter values used during the experimentation are described in Section 1.5.1. In this equation, $T_{pos}(P)$ and $T_{neg}(P)$ denote the sets of tests passed and failed for the program P .

$$fit(P) = W_{pos} \times |\{t|t \in T_{pos}(P)\}| + W_{neg} \times |\{t|t \in T_{neg}(P)\}| \quad (1.1)$$

After the fitness evaluation, an empty set of *offsprings* is created to store new candidate solutions. Each iteration begins with the generation of a random number (line 9), ensuring some level of stochasticity for the choice of operators represented by the set of all operators marked as *OPS*. The loop then evaluates each mutation operator in the set. If the predefined probability of the operator exceeds the generated random number, it is applied to the current population and any resulting individuals are added to the set of offsprings.

A similar process governs the application of crossover, as shown in line 14. If the crossover probability condition is met, two parents are randomly selected, and a new candidate is produced by recombining them via the so-called cross-over operator. Once all potential new candidates have been generated, the next population is selected from this offspring pool using a fitness-based selection process.

Code 1.4.1: The high-level algorithm of GenProgJS

```

1   $popul_0 \leftarrow \text{initPopul}(P)$ 
2   $i \leftarrow 0$ 
3   $REP \leftarrow \emptyset$ 
4  while  $i < MG$ ; loop
5       $\text{refreshFitness}(popul_i, T_{pos}, T_{neg})$ 
6       $\text{offsprings} \leftarrow \emptyset$ 
7      for all  $o \in OPS$ ; loop
8           $r \leftarrow \text{rand}()$ 
9          if  $r < o.\text{probability}$ ; then
10              $\text{offsprings} \leftarrow \text{offsprings} \cup o.\text{operate}(popul_i)$ 
11          end if;
12      end loop;
13       $r \leftarrow \text{rand}()$ 
14      if  $r < \text{crossover}.\text{probability}$ ; then
15           $r_1, r_2 \leftarrow \text{rand}()$ 
16           $\text{offsprings} \leftarrow \text{offsprings} \cup \text{crossover}(popul_{r_1}, popul_{r_2})$ 
17      end if;
18       $popul_{i+1} \leftarrow \text{selector}(\text{offsprings})$ 
19       $i \leftarrow i + 1$ 
20       $REP \leftarrow REP \cup \{o \mid o \in \text{offsprings} \wedge o.\text{fitness} == W_{pos} \times |T_{pos}|\}$ 
21  end loop;

```

Selection relies on fitness scores: individuals with the lowest fitness values are retained in the population, up to a maximum number defined by a configurable parameter. To maintain diversity and randomness in the population, the system first selects a larger candidate pool, typically n times the target size from which it randomly samples individuals to form the final population.

At the end of each iteration, the generation counter is incremented, and the collection of potentially improved program variants—denoted by (REP)—is updated. While in theory, each iteration should yield better-performing individuals, population limits and randomness can cause promising candidates to be discarded. There is no formal guarantee that any generated variant is fully correct, but the fitness score provides a practical signal, as such, variants without failing tests can be treated as more likely correct. We also conducted experiments on a subset of the generated candidate patches to examine how different candidates compare to the original source code, as well as the developer fixed version. In this way, another direction could be taken into account when reasoning about the true correctness of the generated patches.

1.4.2 Patch Correctness of Plausible Candidates

A possible route to evaluate the correctness of candidate patches generated by APR tools (GenProgJS in our case) is to apply similarity-based validation. Detailed information on this process can be found in Figure 1.2. GenProgJS was run for 30

generations per bug, producing a considerable number of plausible patches for each issue.

We took the faulty line from both the originally buggy program and the candidates for the patched versions, along with 3 lines before and after the faulty line. This one-liner code snippet, along with its context, is used further for similarity analysis. After training a Doc2Vec model, we encode each snippet in an N -dimensional vector representation that captures contextual, syntactic, and semantic features. These vectors are then used to calculate the similarity between the developer’s fix and the plausible patches produced by GenProgJS.

Mainly, the goal is to see whether patch similarity is an effective way to reason about patch correctness as a possible step after the generation of plausible patches. In many instances we looked into, determining correctness was relatively easy, as most patches involved single-line edits, making their impacts quite clear. However, while the determination of whether one patch is correct was often straightforward, it became more of a daunting task to decide which of two semantically valid patches was the better choice. Generally, simpler patches are preferred as a rule of thumb, this aligns well with similarity-based methods, though it is not always foolproof. The ranking system can also reveal when a developer’s patch significantly diverges from the original code, indicating a genuine design change or a major refactor instead of a mistake. Including the developer’s fix helps in making more grounded conclusions on whether the more similar a patch is to the original buggy code, it is more likely to be correct.

Our main hypothesis suggests that, according to a document embedding model like Doc2Vec, the developer’s fix will typically be situated near the original buggy code in the vector space. Since current APR techniques—including GenProgJS—sometimes face challenges with complex, real-world bugs, methods that lean towards simpler and more contextually related patches could be especially beneficial.

To assess the effectiveness of similarity-based rankings, we used the well-known *Normalized Discounted Cumulative Gain* (nDCG) metric [80], which takes into account both the position and relevance of items in a ranked list. The metric is defined as:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \quad (1.2)$$

In Equation 1.2, *Discounted Cumulative Gain* (DCG) reflects the usefulness of the items based on their positions, while *Ideal Discounted Cumulative Gain* (IDCG) is the ideal DCG, *i.e.*, the highest possible score attainable by a perfect ranking. The exact formulas used are the following.

$$DCG_p = \sum_{i=1}^p \frac{2^{rel_i} - 1}{\log_2(i + 1)} \quad (1.3)$$

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{\log_2(i + 1)} \quad (1.4)$$

Each item in the ranked list was assigned a relevance score rel_i , reflecting its quality in comparison to the developer patch. The annotations were performed manually by two independent experts, with disagreements resolved by a third reviewer. The relevance values were defined as follows:

- 3: Developer fix (always considered most relevant)
- 2: Patch is syntactically identical to the developer fix (ignoring whitespace)
- 1: Patch is semantically equivalent to the developer fix
- 0: Uncertain correctness
- -1: Clearly incorrect patch (e.g., syntactically invalid)

Intermediate scores such as -0.5 were also used when the annotators were uncertain due to domain-specific uncertainties. These annotated values are published in the online appendix ⁴.

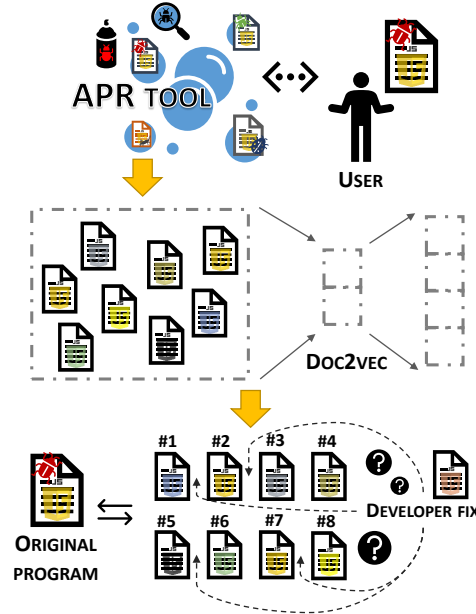


Figure 1.2: Illustration of the process of measuring patch similarity.

⁴<https://github.com/RGAI-USZ/JS-patch-exploration-APR2021>

Learning Document Embeddings

For each bug, we selected a codesnippet that includes the faulty line and three lines before and after it. These fragments were tokenized using a simple regular expression, preserving member operators (e.g., `this.bar`) as single tokens. For instance, `function foo () { return this.bar; }` would be tokenized into `['function', 'foo', '(', ')', '{', 'return', 'this.bar', ';', '}', '']`.

Doc2Vec is a single-layer neural network that learns dense vector representations for entire documents. This model was trained on the preprocessed code snippets, with a chosen vector size of 256, and a window size of 5. Tokens with frequency less than 2 were ignored during training, which we set for 50 epochs, leaving all other parameters at their default values. The similarity between the vectors of the document was calculated using the *COS3MUL* metric [101], which balances contributions in both positive and negative terms while reducing susceptibility to dominant outliers in vector space.

Sample Plausible Patches

To evaluate similarity rankings in a practical setting, we applied GenProgJS to bugs in the Eslint project. Each run was configured for 30 generations, often resulting in a large number of plausible patches per bug. A sample of 10 such bugs, along with their candidate counts and the corresponding developer fixes, is shown in Table 1.1. All considered fixes are single-line edits.

Table 1.1: *Plausible patches and their corresponding developer fix in the Eslint project.*

Bug Id	Candidates	Original line	Developer fix
Eslint 1	4	if (name === "Math" name === "JSON")	if (name === "Math" name === "JSON" name === "Reflect")
Eslint 41	3	end.column === line.length)	(end.line === lineNumber && end.column === line.length));
Eslint 47	3	column: 1	column: 0
Eslint 72	7	loc: lastItem.loc.end,	loc: penultimateToken.loc.end,
Eslint 94	14	op.type === "Punctuator" &&	(op.type === "Punctuator" op.type === "Keyword") &&
Eslint 100	12	penultimateType === "ObjectExpression"	(penultimateType === "ObjectExpression" penultimateType === "ObjectPattern")
Eslint 217	4	if (!options typeof option === "string")	if (!options typeof options === "string")
Eslint 221	221	return parent.static;	return false;
Eslint 321	5	...loc.end.line !== node.loc.end.line &&...	...loc.end.line !== node.loc.start.line &&...
Eslint 323	192	else if (definition.type === "Parameter" && node.type === "FunctionDeclaration")	else if (definition.type === "Parameter")
Total	465		

Among these, some fixes were trivial (e.g., numeric replacement in Eslint 47), while others required more structural reasoning (e.g., adding conditions in Eslint

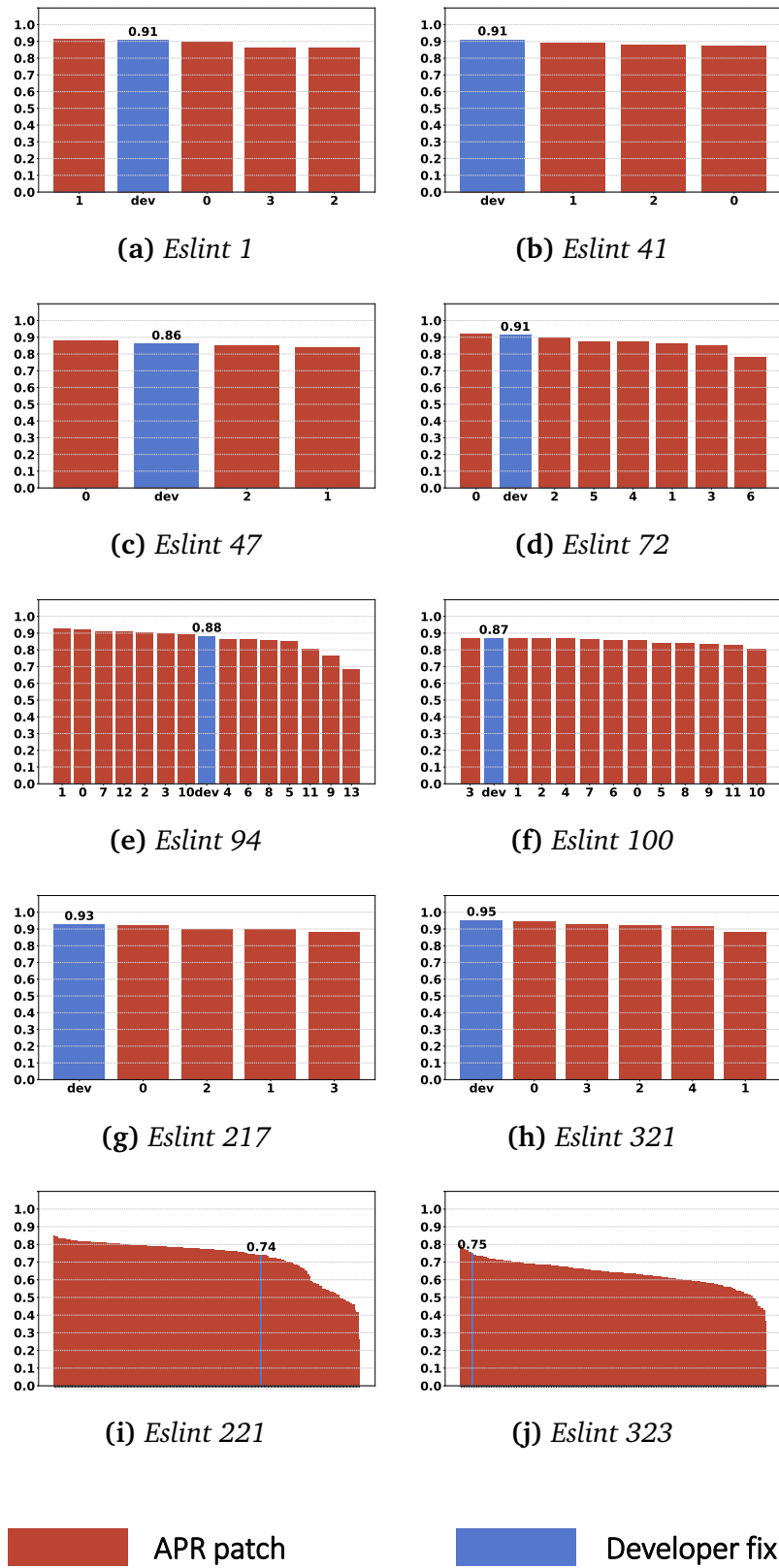


Figure 1.3: The developer fix and patches ranked based on their similarity to the original program.

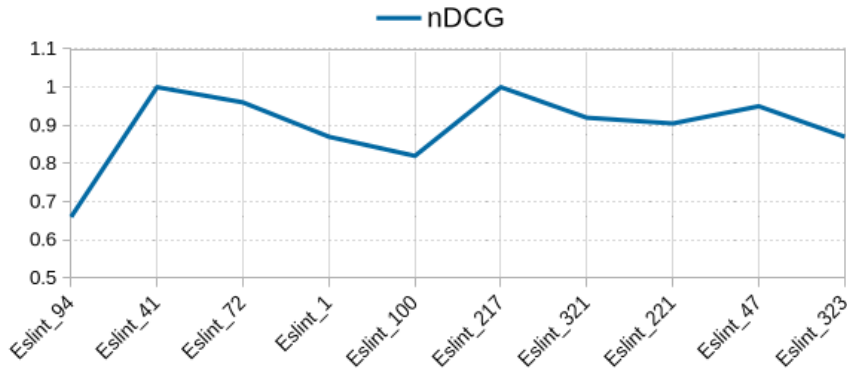


Figure 1.4: The *nDCG* metric value based on each of the examined bugs' ranked list.

100). Despite the high number of candidates (465 in total), only three were syntactically identical to the developer's fix. Nevertheless, several were semantically equivalent, indicating that although APR might not always reproduce the exact fix, it can often approximate its behavior.

Interestingly, certain bugs such as Eslint 221 and Eslint 323 had disproportionately high numbers of plausible patches. For Eslint 221, this is attributable to the nature of the required fix. In JavaScript, returning a value equivalent to `false`, is achievable through many options (e.g., `0`, `null`, `undefined`, etc.). In contrast, Eslint 323's patch count likely stems from a weak test suite, which allowed a wide range of condition variants to pass validation.

Although similarity-based validation presents an alternative perspective on evaluating patch quality, it primarily serves as a "post-processing" type of analysis. Ultimately, the structure and behavior of the repair pipeline, especially the design of mutation operators, remain the most decisive factors in the generation of patches and hence the overall quality of them. What we observe during the validation process also highlights the importance of a good operator design. Operators with greater freedom on modifications they can produce can overwhelm the search space with poor candidates, whereas those that are not generalized enough may completely overlook many possibilities of plausible code fixes.

Since JavaScript has unique characteristics that separate it from statically typed languages such as Java or C, or even scripting languages such as Python (e.g., the presence of `null` vs. `undefined`), implementing operators present in other tools could be insufficient. Therefore, we included custom operators to align with the dynamic nature of JavaScript and its common patterns, while also drawing inspiration from previously well established repair frameworks such as Astor [112, 114].

1.4.3 Operators

Table 1.2: *Operator comparison of Astor (Java) and GenProgJS (JavaScript).*

Cardumen	GenProgJS	Description
ExpressionReplaceOperator	ExprReplacer	Selects an expression and replaces it with a similar one found elsewhere in the source code.
FineGrainedExpressionReplaceOperator	CallChanger	Changes function call arguments by adding, replacing, or removing one of them.
JGenProg		
InsertAfterOp	ExprStatementInserter	Inserts an expression before or after a statement in the specified buggy line.*
RemoveOp	ExprStatementRemover	Removes a whole statement in the specified line.*
ReplaceOp	ExprStatementChanger	Changes a randomly selected expression statement in the specified buggy line.* *When more statements are present, one will be selected randomly.
JKali		
ReplaceIfBooleanOp	ConditionalChanger	Replaces condition found in an "if statement" with a <i>true</i> or <i>false</i> value.
ReplaceReturnOp	ReturnInsert	Inserts a return statement before one of the expression statements found in the specified line.
JMutRepair		
BinaryOperatorMutator	BinaryOpChanger	The operator changes a binary operator to another, entirely random one.
LogicalBinaryOperatorMutator	LogicalExprChanger	Replaces && and operators with each other.
ArithmeticBinaryOp. Mut.	ArithmeticBinaryOp. Chg.	Collects a list of <i>arithmetic binary operations</i> on the given line, randomly chooses one, and replaces it with another <i>arithmetic operator</i> . Note: only the <i>operator</i> will be replaced.
NegationUnaryOperatorConditionMut.	ConditionalInverter	Inverts the condition of an if statement by inserting or removing the ! operator.
RelationalBinaryOperatorMut.	ConditionalBinaryOpChanger	Similar to <i>BinaryOpChanger</i> , but it will only modify operators that "type" are similar to each other.

Astor serves as a framework for automatic software repair in Java, with diverse objectives. Specifically, we adapted certain mutation operators from its implementation for use in GenProgJS. Table 1.2 offers a comparison between the operators available in Astor’s implementations and those in GenProgJS. Within this table, as well as in Table 1.3, the JS column indicates whether the operator can generate patches tailored to JavaScript. The red cross indicates that this operator is not specific to the JS language, and also existed before in some form. The single tick indicates that this operator did *not* exist before in any of the examined tools and is specific to script languages, while the double tick represents the case where the operator is specific to the JS language. It should be noted that although some operators may exist in other repair tools, their behavior can differ when applied in JavaScript. To illustrate this, we describe artificial scenarios where an operator creates a patch that is logical in JavaScript but not in Java/C. GenProgJS also incorporates JavaScript-specific operators not implemented by other repair tools (refer to the language-dependent section in Table 1.3). For the implementation of these operators, we got our inspiration from studies that focused on the most prevalent JavaScript errors [67, 131, 135]. After extensive testing on open source projects, Hanam *et al.* (2016) [67] classified JavaScript bugs into six major groups, namely; (1) dereferenced non-values, (2) incorrect comparison, (3) missing argument, (4) incorrect API configuration, (5) improper *this* binding, and (6) unhandled exception. Our manual analysis of the BugsJS dataset confirms that these categories capture JavaScript-specific behaviors that can cause software errors. With this in mind, we devised the mutation

operators in Table 1.3 to address these challenges. We also developed operators that are capable of generating patches spanning across multiple lines of code (denoted as multiline repairs), detailed in the *Batch repair operators* section of Table 1.3.

The first column defines the name of the operator in GenProgJS, while the last column describes the source code modifications a given operator is able to make. The JS column denotes whether the operator is able to create a patch which makes sense in JavaScript but not in more traditional languages. In order to support our argument along JS operators, we present a simple example in these cases and describe how the given modification leverages the dynamic nature of JavaScript (see comments in green).

GenProgJS also features a dedicated operator for the crossover operation typically found in genetic algorithms. In contrast to mutation operators, *crossover operator* operates on two variants of the program, which are selected by the genetic algorithm and passed to the JavaScript program from the Python engine. First, the operator gets the AST representation of each parent individual, where the buggy lines are. Second, it selects a subtree from both of them. In the third step it replaces the subtrees with one another. Lastly, it selects one of the offsprings from the produced variants.

By design choice, all offsprings are generated by the JavaScript program featuring the individual operator implementations, hence the dedicated operator for crossover. The genetic algorithm then handles this as a new individual. The described approach is *TreeCrossover*, which extends the base *Crossover*, which represents a single crossover operation on two separate source codes (which might or might not be the same).

1.5 Experiment Setup

1.5.1 BugsJS: An Executable JavaScript Dataset

The BugsJS dataset [66] offers a collection of reproducible JavaScript bugs collected from 10 open source GitHub projects. It includes both single-line and multiline bugs. The projects included in the dataset are listed in Table 1.4.

Although the dataset originally contains a total of 453 bugs, not all of them are suitable for repair attempts. In particular, many lack failure test cases, which are essential for fitness evaluation in APR tools. As a result, only 352 of these bugs meet the criterion of having at least one failing test. Additionally, three projects, *Hessian.js*, *Shields*, and *NodeRedis*, were excluded from our experiments due to the same limitation. BugsJS provides a convenient API for retrieving buggy and fixed code versions, as well as for running the associated test suites. These features proved invaluable when evaluating automatically generated patches against developer-written fixes. GenProgJS also leverages static metadata from the dataset, such as bug location

Table 1.3: Mutation operators of GenProgJS.

Operator Name	JS	General Operators
MutExprStatementChanger	✓	Collects a list of expressions found in source code, then creates a new expression combining these expressions (AST based operation and expr. creation). The buggy expression will be replaced by the newly created one. <code>this.a = "lhs" + rhs → this.k = "stringValue" + 5</code> //replacing an AST node with another similar to the original can result in a code fragment that behaves completely different. In the example above, consider string concatenation, and the operator changes the BinaryExpression found in the code snippet. After changing the leaf nodes of the AST as well, a string and a number are being concatenated. The operator is not limited to replacing nodes with the same depth, so changes of greater extent are also conceivable (i.e., only the AST type of the nodes should match).
MutExprStatementInserter	✓	Same as described above, but instead of replacing the old expression statement with the new one, this operator inserts it before or after the buggy statement. <code>let bar = foo(); → let bar = "" "concat numbers" + 5 + 7;</code> //expressions mined from the source code are always language dependent, but in the case of JavaScript, the result expression is rather specific. For example a variable can get a default value like in this example.
NumberChanger	✗	This operator adds ± 1 to a numeric literal.
UpdateExprChanger	✗	Changes an update expression. For example <code>i++ → i-</code>
ShiftOpChanger	✗	Changes operators like <code><, >and ></code> .
VarChanger	✓	Collects a list of identifiers from the source file, and replaces the variables found in the buggy line. <code>a.map(x => x.id) → b.map(x => x.id)</code> //in JavaScript, a variable can be declared after it has been used - this is the so called hoisting. The given example can be a correct repair even if the b variable was not declared in previous lines. Due to the prototypes, accessing a class member is not obvious either, if the member does not exist we do not get an error.
Function-related modifications		
AsyncFunction	✗	Very similar to the <i>AwaitInserter</i> operator, with the exception that this one will operate on a whole <i>async</i> function. Meaning, it will use the <i>AwaitInserter</i> on some - randomly chosen - identifiers or function calls found in the function implementation.
AwaitInserter	✗	Inserts an <i>await</i> expression before an identifier or a function call, or on the right side of an assignment before an identifier or a function call. This happens regardless of the fact, that the line to be patched happens to be in an <i>async</i> function or not.
FunctionCallRemover	✓	Replaces the <i>call expression</i> with the <i>callee</i> . <code>let bar = this.foo(); → let bar = this.foo</code> //in the example removing the call expression results in an assignment, where the bar variable will contain a function reference. While this can be done in compiled languages too with the use of auto variables, it still won't compile if used incorrectly, anywhere in the source code. However, here it is fine to use it as a callable function reference once, while using it as a value elsewhere.
FunctionMaker	✓	Similarly as mentioned above, with the exception that this operator will insert a call with no arguments. <code>let bar = this.foo; → let bar = this.foo()</code> //opposed to FunctionCallRemover, here the bar variable will get the return value of the foo function. If the function does not return anything, the value of the variable will be undefined. Compared to compiled languages, where types and scopes are being checked, this code snippet might cause compilation error, while in a scripting language it might be correct.
ReturnNone	✓	This operator either replaces the buggy line with a <i>return null</i> ; or return an <i>undefined</i> constant. <code>if (sp) { break; } → return undefined;</code> //JavaScript intended undefined to refer to a value that doesn't yet exist, while null was meant to refer to a value that does exist but points to nothing. This distinction is not typical in other languages.
Batch repair operators		
CaseRemover	✗	Removes an entire <i>case-block</i> . It is worth noting that it is multi-line capable in a sense that it is able to remove multiple statements in a contiguous block.
LoopFix	✗	In a <i>for loop</i> it is usually considered a good practice not to update the <i>update variable</i> inside the <i>loop body</i> , another one is to see whether or not any of the <i>loop condition</i> variables are updated. On the other hand, in case of a <i>while loop</i> it is usually advised to update at least one of the variables found in the loop condition. When the loop condition is a <i>constant expression</i> , then we look for at least one loop-breaking statement, such as: <i>return</i> or <i>break</i> . The finishing step is to make changes on the loop based on these practices.
SwitchCaseFallthrough	✗	Removes at least one <i>switch-case breaker</i> from a selected <i>case-block</i> . By <i>switch-case breaker</i> we mean such a statement that could potentially 'break' the control flow. This can be either a <i>break statement</i> or a <i>return statement</i> .
SwitchCaseNestedRemover	✗	Given a <i>switch statement</i> , this operator will remove at least one <i>case-block</i> from the statement. Just like LoopFix, this operator is also capable of modifying the source in multiple lines, even if they are not in a contiguous block.
SwitchDefaultThrow	✗	Throws error on reaching <i>default</i> .
SwitchDiscriminant	✗	Some of the following switch operators - including this one - will insert a <i>default case</i> inside a switch statement, if it did not find any. This particular one inserts a <i>return statement</i> containing the expression found inside the <i>switch</i> .
TryCatcher	✗	Surrounds a selected <i>expression statement</i> with a <i>try catch</i> clause.
String literals		
StringCut	✗	Removes a part from a <i>string literal</i> . The length of the removed parts is random, but not longer than the actual string itself.
StringPartition	✗	Keeps only a part of the selected <i>string literal</i> . This operator differs from the previous one in a sense, that it keeps the selected substring and does not discard it.
StringRemover	✗	Replaces a whole <i>string literal</i> with an empty <i>string</i> .
StringToLowerCase	✗	For the sake of completeness: takes the lower case representation of a given <i>string</i> .
StringToUpperCase	✗	Takes the upper case representation of a given <i>string</i> .
Conditional checks and validation		
BitwiseBinaryOpChanger	✗	Operates on <i>bitwise binary</i> operators like bitwise <i>AND</i> , <i>OR</i> , <i>XOR</i> , <i>NOT</i> and replaces them with one another.
IfElseChanger	✗	The <i>Operator</i> is composed of two others: the first one replaces the last <i>else if</i> branch, if found, with an <i>else</i> . The other may delete <i>if branches</i> from the statement.
NullCheck	✓	Inserts a variable name before an expression, and may combine it with the unary <i>!</i> operator. This can be a useful check for the variable's existence. This check works especially well when working with potentially <i>undefined</i> (not only null) variables. <code>if (a.loc === 0) → if (a && a.loc === 0)</code> //in strictly typed languages conditions usually only accept boolean values, and one usually can not check for a variable's existence this way. Let us say, that a never have been defined anywhere in the code. So in the code snippet above we are actually referencing an undefined variable. This is usually not allowed even in interpreted languages. But here, JavaScript's <i>undefined</i> constant comes to the rescue, which, as a boolean, evaluates to a false value, so that we will not get a <i>ReferenceError</i>
TernaryExchange	✗	This <i>Operator</i> will simply replace the <i>alternate</i> and <i>consequent</i> nodes of a <i>ternary expression</i> . The simplest example would be to change: <code>a ? a : b → a ? b : a</code>
TernaryToIfElse	✗	This one will replace a <i>ternary expression</i> with an equivalent <i>if-else statement</i> .

Language dependent operators		
EvalMutation	✓	In JavaScript, one can evaluate another js code snippet inside the main script, with the simple use of <i>eval</i> . As this task usually proves to be highly error prone, we attempt to also fix the JavaScript source codes found inside an <i>eval</i> call. We simply get the argument of <i>eval</i> and - if it is a simple <i>string literal</i> - pass the extracted code to another of our <i>operators</i> . However, in cases where the argument of the call is not a "literal only", we try to evaluate the argument as an expression statement. This way, one can even get the source code from an argument like: <i>eval(source + code)</i> , where <i>source</i> and <i>code</i> are existing variables. <i>eval("")</i> → <i>eval("")</i> //treating a string literal as source code is unique to script languages, in compiled languages there is no similar construction.
DeclarationChanger	✓	Changes the declaration kind. For example it may replace a variable declared with the keyword <i>let</i> with a variable that has the same name but declared as a <i>var</i> . <i>var a = 0;</i> → <i>const a = 0;</i> //variables declared with <i>var</i> are processed before any code is executed, the <i>let</i> and <i>const</i> keywords also have their special meaning.
Array index safety		
ArraySubscriptCap	✗	Limits array subscript between $[0, array.length)$. The limiting is done by inserting a chained <i>ternary</i> operation. If the array is empty, the 0th index will be used.
ArraySubscriptRand	✗	If the array index is less than zero, or greater than <i>array.length</i> , a random integer will be generated between these two numbers. If the array is empty, the 0th index will be used.
ArraySubscriptRemainder	✗	If the indexing is out of bounds, then the array index will be updated according to the following rule: $index \% array.length$. Where the % sign represents the modulo operator.

indices. Consequently, the tool does not perform bug localization and focuses exclusively on generating candidate patches.

Table 1.4: Systems contained by the BugsJS dataset.

System	kLOC	#bugs	Github URL
Bower	16	3	/bower/bower
Eslint	240	290	/eslint/eslint
Express	11	27	/expressjs/express
Hexo	17	7	/hexojs/hexo
Karma	12	6	/karma-runner/karma
Mongoose	65	12	/Automattic/mongoose
Pencilblue	16	7	/pencilblue/pencilblue
Total	377	352	

1.5.2 Other Datasets on JavaScript

Although the literature contains numerous datasets related to JavaScript, locating those that are fully runnable and support automated program repair can be quite difficult. Some well-known datasets feature JavaScript examples, but often fail to provide execution-ready test environments or realistic bug scenarios relevant to large-scale software development [140, 149].

The dataset used in *CoCoNuT* [109], for instance, is based on 12 JavaScript bug patterns introduced in BugAID [67]. Although it contains useful examples for learning bug patterns, it lacks the infrastructure necessary to execute and test those programs.

Similarly, the dataset accompanying *CodeBERT* [50] suffers from the same limitation despite including buggy JavaScript code; it does not provide runnable or reproducible test cases.

RunBugRun [140] offers a collection of runnable bugs across multiple programming languages, extracted from competitive programming platforms. While this is an important step toward reproducibility, bugs found in coding challenges may not reflect the types of issues developers encounter in complex, real-world systems. Additionally, *RunBugRun* lacks a mechanism for executing bugs individually, which is problematic for G&V-based approaches that evaluate each variant of the program based on failure tests. Running the full benchmark for every candidate can quickly become impractical.

Other recent datasets such as *FixJS* [31] and the dataset introduced by *Code-grain* [73] mainly target machine learning models for code refinement. Like the CoCoNuT and CodeBERT datasets, they do not offer executable test cases that would enable full repair pipelines.

1.5.3 Parameter Settings

GenProgJS includes several configurable parameters. In our experiments, these were selected empirically, guided by existing work in the APR literature, rather than by exhaustive tuning.

Following common practice, we kept the population size relatively small, set to 20 which reflects the trade-off between search space exploration and computational cost. Because the number of individuals is modest, applying multiple operators per generation remains computationally feasible, reducing the need for many generations. We set the maximum number of generations to 30. In practice, promising candidate patches often emerge early, so this parameter had little impact. However, the stopping criterion can be adjusted depending on the task.

GenProgJS supports five stopping conditions: (1) reaching a maximum generation count, (2) exceeding a runtime limit, (3) producing a certain number of valid repairs, (4) satisfying all of the above, or (5) satisfying any one of them. In our experiments, only the generation-based criterion was used, but more complex combinations may be beneficial in different settings.

The weights used in the fitness function were borrowed from the original GenProg implementation: W_{pos} is set to 1 and W_{neg} to 10. Although alternative fitness functions have been explored in the literature [114], ranging from basic test count metrics to more sophisticated heuristics, they all share the core idea of penalizing program variants that fail tests.

Each mutation and crossover operator also has an associated usage probability. These were uniformly set to 0.8 in our experiments. Since the effectiveness of each operator is difficult to quantify independently, no fine-tuning was applied to these values. However, we did conduct an ablation study focused on the crossover operator due to its conceptual distinction from the standard mutation. The results of this

analysis are presented in subsequent sections. All experiments were run on a machine equipped with an Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz.

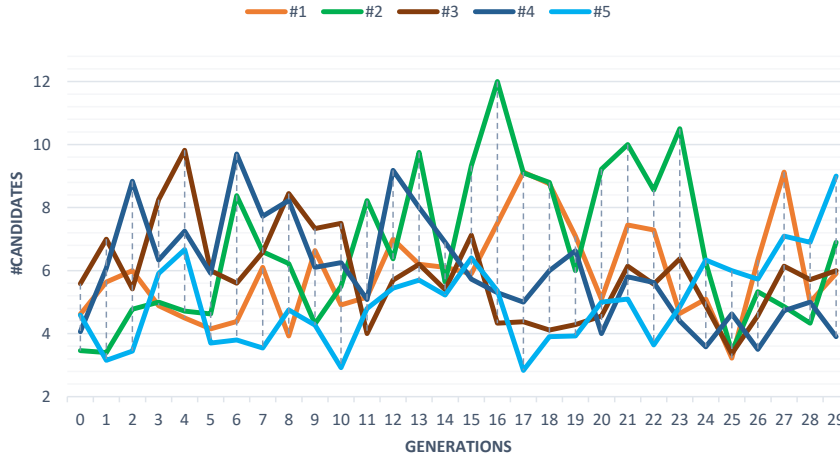


Figure 1.5: *Generated number of candidates from the 5 independent runs.*

The inherent randomness of genetic algorithms makes it challenging to draw a clear conclusion. For a slight mitigation of this issue, all experiments were executed in five independent runs. Figure 1.5 presents the number of candidate patches generated throughout these runs, broken down by generation. Although fluctuations are noticeable, the number of candidates remains broadly consistent both between generations and between separate runs. On average, between 2 and 14 patches per generation were produced when the data from all bugs were aggregated.

1.5.4 LLM-based Repair

In previous work [93, 94], we explored automatic bug fixing using the pre-trained GPT-2 model [143]. While the original pre-trained model struggled with the task, the fine-tuned variant achieved notable improvements, generating 3 patches that are an exact match of the developer fixed version. Although some projects in the current study overlap with those used during fine-tuning, most bugs, primarily from the Eslint project, were not part of the training dataset.

For the current experiments, a single candidate line was generated for each bug and evaluated using the entire test suite. In our experiments with the GPT-2 model, we configured the generation process with the parameters `do_sample=False`, `top_k=0`, `top_p=0`, `temperature=0.0`, effectively selecting the highest-probability continuation at each step. This setting ensures that the model deterministically generates the same output for each and every prompt. Each of the input prompts consisted of 900 tokens of surrounding context, and the model was tasked with

generating only the next line of code as the proposed repair.

1.6 Results

GenProgJS found plausible patches for **31** bugs in the BugsJS dataset. In Table 1.6 we collected basic data for these repairs. For each bug, the BugsJS Dissection site ⁵ provides several useful information on its nature and the type of bugfix it requires. In the table, next to the Bug Id, the bug type according to the dissection taxonomy is shown. The 31 repaired bugs cover a variety of bug types (31 bugs belong to 14 types of bugs) and belong to 6 different open-source projects, showing that the system indeed provides a general way of bugfixing. Although it is true that almost half of the fixes were generated for the Eslint project, it should be noted that in the BugsJS database there are way more bugs for this project than for others (see Table 1.4). The overall 31 bug repairs are a good starting point for JavaScript program repair, GenProg implementations for C and Java started with even fewer bug fixes [111, 174]. Considering that state-of-the-art repair tools can repair up to 30 bugs from a well-known database does not make these more efficient, since most of the fixed bugs are nonexclusives [97]. Also, many authors in the APR community reason that the number of patched bugs by each repair tool is dependent on well-known bug databases (e.g., Defects4J) [42] and their precision/performance could drop dramatically on other previously unseen databases.

The search was not stopped at the first candidate found, but each experiment continued until 30 generations were reached. This resulted in a variable number of plausible patches in each run. We experimented with two different settings, which we call (1) *generator* and (2) *mutator*. In the first case, when a plausible patch is found, it is stored and removed from the population. This approach encourages the tool to generate different patch variants. On the other hand, mutator does not remove the plausible patch from the population but checks whether the patch was already generated (thus ensuring that every plausible patch is unique). In the latter case, we observed that after the first plausible patch the process most likely re-used the plausible ones and from that point constantly generated new plausible candidates. Hence, the first plausible patch has an emphasized role in the process. Although we expected that the two settings will produce very different results, the outputs are essentially the same. The following experiments were carried out using the generator approach, although mutator would not give very different results.

We measured and reported the total runtime for all 30 iterations. Additionally, each generated candidate patch was manually evaluated for correctness, with the results of this evaluation provided in the online appendix ⁶, where each generated

⁵<https://bugsjs.github.io/dissection/>

⁶<https://genprogjs.github.io/experiments/>

patch can be observed in a diff-style comparison to the original code. In Table 1.6 next to the Fix type, the average repair time of the first repair candidate is listed in seconds. Despite the fact that the algorithm was not stopped when it found a plausible patch, the importance of the first patch found is undeniable. Although it cannot be stated in general that only those operators are useful, which were involved in the repair process of the first plausible patch, as we progress through the generations, more and more operators are employed and it is becoming increasingly difficult to decide which operator application was useful and which was not. We have also indicated in the table whether there is a plausible patch that is identical (or lies close) to the human patch.

We were also interested in the usages of JavaScript operators in the repair processes (whether an operator is capable to generate a JS patch or not can be found in Table 1.3). In 30 generations, an operator has high probability of being used, thus examining whether a JS operator was used in the whole search process holds little meaning. Investigating whether a JS operator was involved in the creation of the first patch may be more appropriate and to consider only the validated patches (*i.e.*, not plausible but correct ones). The plot in Figure 1.6 illustrates the distribution of JS operator usage. Those bugs which are not listed on the figure are either because (1) no correct patch were generated for them (2) at least one correct patch was generated, but in the process no JS operator was involved. With blue bars, we indicate the percentages that were measured through every plausible patch, while green bars indicate when we only considered the first candidate. For a given bug, these percentage values are calculated using the following formula: $|\{p|p \in P_{js}\}|/|\{p|p \in P_{all}\}|$, where P_{all} denotes the set of all patches, while P_{js} is the set in which only those patches are present in which a JavaScript specific operator made a modification. Note that this percentage can also be interpreted as the probability of applying a JS operator during the search process for a given bug.

As can be seen in Table 1.6, in the 12 cases the generated patch semantically matches the developer fix. 7 bugs are listed on Figure 1.6, meaning that if we consider every patch, 7 of the 12 repair processes involved a JS operator (58 %), while if we examine only the first candidate, JS operators have only been used 3 times (25 %). Interestingly, there are cases where almost every applied operator was JS specific (*e.g.*, Eslint 221 or Eslint 321), while no JS operator was involved during the synthesis of the first patch. During the repair process of Eslint 221, 5 JS operators were used, while for Eslint 321 only 1, which is *NullCheck*. This operator along with *VarChanger* was actually used in 6 out of 7 cases of observed bugs. From this and from other empirical observations we would like to emphasize the prominent role of these operators in JavaScript program repair. They are applicable in many real-world scenarios, and as we have seen here, they create correct patches. The *NullCheck* operator is more interesting in this sense, since it captures the dynamic na-

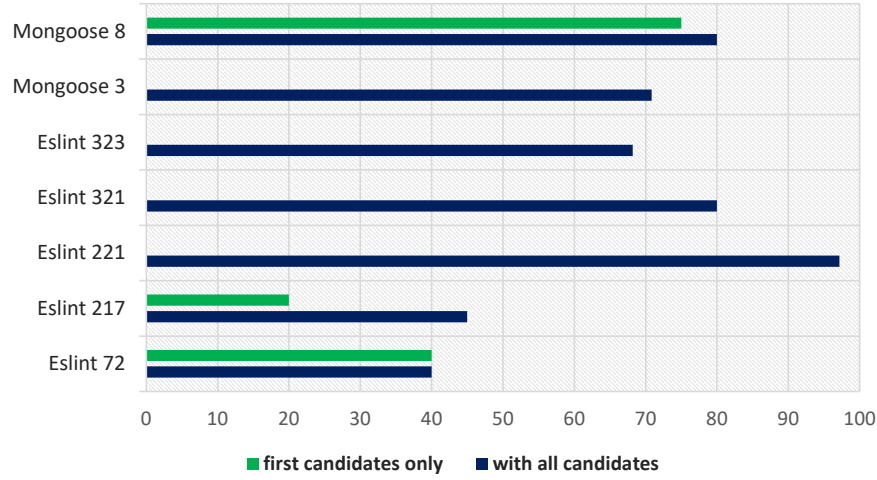


Figure 1.6: Probabilities of JavaScript operator usages.

ture of JavaScript and summarizes well the root cause of the misbehavior. Although *NullCheck* is an eminent operator, it is worth noting that if we consider only the first patch, it was involved in only 1 repair process, while *VarChanger* in 3 processes.

To put our results in context, we observed 18 state-of-the-art test suite-based repair tools. Without claiming this to be the complete list of such tools, we did our best to place the most recent works on the list. It should be noted that we intentionally excluded data-driven approaches (see Section 1.3), since the evaluation criteria of these often differ from test-based repair. We based our exploratory research on the Living Review on Automated Program Repair by Martin Monperrus [123]. The tools found and their corresponding "repair performance" can be found in Table 1.5. Precision means the precision of correctly fixed bugs in bugs fixed by each APR tool (x/y). The fix rate is the percentage of bugs for which the repair tool has generated a plausible patch ($y/\#bugs$). The column next to the tool name corresponds the total number of bugs on which the given approach was evaluated. In each row, we provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs for which a plausible patch is generated by the APR tool (*i.e.*, a patch that makes the program pass all test cases). To the best of our knowledge, GenProgJS is the first test-suite-based automated repair tool for JavaScript (and the first that was evaluated on the BugsJS dataset); we were unable to directly compare it with other approaches. On the other hand, one can see that both the Fix rate and the Precision is comparable with other state-of-the-art repair tools. Note that the implementation of GenProg both for C and Java generated less correct patches than for JavaScript. This result has little to do with the quality of the implementation; rather it is due to the unique characteristics of the observed datasets and/or the applied genetic operators. APR tools written for Java are usually evaluated on the Defects4J [86] dataset

without considering others. That is why current tools can easily have the problem of overfitting: they are biased to the database. Although the available datasets are limited to Java and have been criticized recently [102], they are relatively well tested and mature compared to what is available for JavaScript.

Table 1.5: *Results of test-suite-based program repair tools.*

Repair tool	# bugs	x/y	Fix rate	Precision
GenProgJS	352	12/31	8.8 %	38.7 %
GenProg [174]	105	1/10	9.5 %	10.0 %
JGenProg [111]	224	5/27	12.1 %	18.5 %
Kali [141]	105	3/27	25.7 %	11.1 %
JKali [111]	224	1/22	9.8 %	4.5 %
MutRepair [34]	463	-/25	18.5 %	-
jMutRepair [112]	224	3/17	7.6 %	17.7 %
Nopol [185]	224	5/35	15.6 %	14.2 %
ARJA [195]	224	18/59	26.3 %	30.5 %
CapGen [113]	224	21/25	11.2 %	84.0 %
SimFix [83]	357	34/56	15.6 %	60.1 %
HDRRepair [95]	90	-/23	25.6 %	-
ACS [182]	224	17/23	10.3 %	73.9 %
ssFix [179]	357	20/60	16.8 %	30.0 %
ELIXIR [150]	82	26/41	50 %	63.4 %
JAID [22]	138	25/31	22.5 %	80.6 %
SketchFix [77]	357	19/26	7.3 %	73.1 %
FixMiner [92]	357	26/32	8.9 %	81.3 %
Prophet [104]	69	14/20	28.9 %	70.0 %
Average	231	14/31	17.5 %	44 %
Median	224	17/27	13.9 %	38.7 %

It took a long time for the algorithm to produce the 30 generations. The runtime depends heavily on the test suite: for instance, the total execution time of the Eslint 72 bug was 33 hours, while for Karma 4 only 9 hours. If one were only interested in the first plausible patch and stop the algorithm, then the execution time drops heavily. In addition, running only failed test cases in the algorithm is highly recommended. Although running time is far from optimal, efficiency is not a widely valued performance target in the field of APR. Authors in [97] found, that state-of-the-art APR tools are the least efficient. While execution time is an important aspect of a software artifact, it always depends on the hardware architecture. The simplest way to boost the speed of the algorithm is to put more powerful hardware behind it. However, there are other options as well. One trivial speed-up possibility is to run the test suite in a smarter way, meaning, while the patches are generated, first try to fit only on the failed test cases and later check whether the code transformation ruined the

previously positive ones. This approach could result in a significant improvement, especially in projects where there are thousands of test cases. The test execution described above is not yet supported by GenProgJS. We took steps to make it available in the future, though we encountered technical difficulties (*i.e.*, using Mocha --grep with inline tests).

Selecting the correct patches from plausible ones is a tedious task for humans [28, 89, 141, 156], which makes patch similarity another important aspect of designing good APR tools. Although most works in this field compute the similarity of software artifacts (*e.g.*, test case execution traces, or generated patches) using some similarity concept, we prioritized candidates based on textual similarity measured on the embeddings of these code snippets. We also manually checked whether there exists a close-to-human patch later in the search process. Note that we used patch prioritization only as an aid (our intuition was that correct patches were probably more similar to the original source code), and every generated plausible patch was examined later. This is marked in the sixth (Identical?) column of Table 1.6. Overall, we found that in 7 out of 31 cases the plausible patch was close or semantically identical to the human patch, and in 5 cases the generated patch is identical to the developer fix, which is at least comparable to similar results in the literature. In Table 1.6 one can examine the manual examination of the patches generated by GenProgJS.

In order to save space, the following abbreviations are being used: F. → Feature, Impl. → Implementation, Chg. → Change, Expr. → Expression, Decl. → Declaration. The shapes in the fifth column have the following meaning: ✓ - syntactically identical, ✓ - semantically identical, ✗ - uncertain. The fourth column (*Average runtime*) averages the time it took to find the first plausible patch over the five independent runs.

In the following, we examine a bug in the Express project to better understand the nature of the generated patches. In the source code examples, the original buggy line is marked with red background and - sign, the automatically generated patch with green background and + sign, while the developer fix with yellow background and > sign.

1.6.1 Repaired Bugs Using GPT-2

Large Language Model (LLM)s are becoming more and more popular and easily applicable to a wide variety of tasks. One such field that uses its capabilities is the field of APR. Nowadays, many LLMs are trained specifically for coding tasks, such as code repair, code generation, code explanation, *etc.*, however, at the time such a wide range of tools were not readily accessible. Especially for this reason, our experimental work involved fine-tuning of GPT-2. For fine-tuning, we used the BugsJS dataset, a specialized benchmark created for JavaScript bug detection and remedi-

Table 1.6: Repairs on the BugsJS dataset produced by GenProgJS.

#	Bug ID	Taxonomy	Fix Type	Average Runtime	Identical?	# Run (Plausible Patches)				
						1	2	3	4	5
1.	Bower 2	Incomplete F. Impl. → Configuration Processing	Chg. of If Condition Expr.	87646	✗	-	2	-	-	1
2.	Eslint 1	Incomplete F. Impl. → Missing Input Validation	Chg. of If Condition Expr.	15988	✗	1	3	4	6	-
3.	Eslint 7	Incomplete F. Impl. → Missing Input Validation	Multiple	51	✗	15	16	-	17	29
4.	Eslint 41	Incomplete F. Impl. → Missing Input Validation	Chg. of a return statement	35899	✗	-	2	3	-	1
5.	Eslint 47	< uncategorized >	Chg. of Class Field Decl.	113	✓	1	1	3	1	1
6.	Eslint 72	Incorrect F. Impl. → Incorrect Output Message	Chg. of Class Field Decl.	1141	✓	3	4	7	5	7
7.	Eslint 94	Incorrect F. Impl. → Missing Type Check	Chg. of If Condition Expr.	357	✗	70	94	14	67	114
8.	Eslint 100	Incorrect F. Impl. → Missing Type Check	Chg. of Assignment Expr.	1447	✓	20	26	12	9	26
9.	Eslint 122	Incorrect F. Impl. → Incorrect Input Validation	Chg. of Method Decl.	219	✗	1	-	-	1	-
10.	Eslint 130	Incorrect F. Impl. → Incorrect Data Processing	Chg. of Assignment Expr.	8293	✗	-	-	1	-	-
11.	Eslint 154	< uncategorized >	Chg. of Assignment Expr.	2155	✗	21	-	25	-	29
12.	Eslint 158	Incorrect F. Impl. → Incorrect Input Validation	Chg. of Method Call	481	✗	-	-	34	24	20
13.	Eslint 217	Generic → Typo	Chg. of If Condition Expr.	1329	✓	6	5	4	4	6
14.	Eslint 221	Incorrect F. Impl. → Incorrect Input Validation	Chg. of a return statement	278	✓	100	334	221	202	183
15.	Eslint 321	Generic → Typo	Chg. of If Condition Expr.	12058	✓	9	4	5	8	13
16.	Eslint 323	Incorrect F. Impl. → Unnecessary Type Check	Chg. of If Condition Expr.	716	✓	260	192	198	159	220
17.	Express 2	Incomplete F. Impl. → Incomplete Data Processing	Chg. of Method Call	2333	✓	2	5	4	115	19
18.	Express 3	Incomplete F. Impl. → Incomplete Initialization	Chg. of Class Field Decl.	9961	✗	1	-	-	-	3
19.	Express 5	Incomplete F. Impl. → Incorrect Filepath	Chg. of If Condition Expr.	456	✗	23	61	57	15	100
20.	Express 8	Incorrect F. Impl. → Empty Input Parameters	Chg. of If Condition Expr.	357	✓	487	350	402	284	468
21.	Express 9	< uncategorized >	Chg. of Assignment Expr.	3428	✗	-	-	-	1	-
22.	Express 16	Incorrect F. Impl. → Incorrect Data Processing	Multiple	280	✗	20	-	20	20	19
23.	Express 18	Incorrect F. Impl. → Incorrect Data Processing	Chg. of If Condition Expr.	155	✗	122	151	67	110	127
24.	Express 26	Incomplete F. Impl. → Handling Special Characters	Chg. of Assignment Expr.	8635	✗	-	1	-	-	-
25.	Karma 3	Generic → Missing Return Statement	< uncategorized >	9154	✓	1	2	3	2	-
26.	Karma 4	Incomplete F. Impl. → Missing Input Validation	Chg. of If Condition Expr.	7917	✗	-	-	1	-	-
27.	Karma 9	Incorrect F. Impl. → Incorrect Filepath	Chg. of a return statement	232	✗	28	30	22	20	31
28.	Mongoose 3	Incorrect F. Impl. → Incorrect Data Processing	Chg. of If Condition Expr.	1071	✓	166	286	226	516	270
29.	Mongoose 8	Generic → Missing Return Statement	< uncategorized >	59144	✓	-	4	9	22	12
30.	Mongoose 11	Incomplete F. Impl. → Missing Input Validation	Chg. of If Condition Expr.	8035	✗	300	477	382	588	147
31.	Pencilblue 4	Incomplete F. Impl. → Missing Type Check	Multiple	473	✗	5	3	4	4	-

ation. The results presented in Table 1.7, reveal that GPT-2 successfully produced three viable patches, each representing an exact correspondence with the respective developer-authored solutions. In particular, GPT-2 avoided generating any misleading yet plausible patches, nor created accurate patches that deviated from the original developer implementations. These results highlight GPT-2’s ability to generate high-quality patches with enough fine-tuning for a specific task, which can be highly desirable in tasks where quality over quantity is preferred, or generation of false positive patches is highly undesirable. Note that we used the same abbreviations as previously in case of Table 1.6.

Table 1.7: *Repairs on the BugsJS dataset produced by the fine-tuned GPT-2 model.*

Bug Id	Taxonomy	Fix type	Identical found?
Express 4	Incomplete F. Imp. → Missing Input Validation	Chg. of Class Field Decl.	✓
Hexo 9	Incorrect F. Imp. → Incorrect Data Processing	Chg. of of Method Call to a Class Instance	✓
Hexo 12	Generic → Data Processing	Chg. of Assignment Expr.	✓

However, GenProgJS achieved superior bug repair performance across the dataset, surpassing GPT-2’s three successful patch generations. This considerable performance gap highlights a fundamental constraint of LLMs in APR applications at the time. While LLMs like GPT-2 demonstrate proficiency in identifying patterns within their training corpus, they encounter difficulties when attempting to generalize beyond a diverse spectrum of bugs, especially those requiring more domain knowledge to produce a specific repair. The limited effectiveness of GPT-2 comes from multiple contributing factors. Initially, the BugsJS dataset, despite being adequate for assessing conventional APR techniques, may lack the variety and number of examples necessary for LLMs to develop broad generalization capabilities across distinct bug categories. LLMs, particularly compact models such as GPT-2, typically require much larger datasets to learn more universal repair methodologies.

However, GPT-2’s ability to produce accurate patches while avoiding incorrect solutions demonstrates its exceptional precision capabilities. Despite GPT-2’s modest repair success rate on the BugsJS dataset (0.85%), it maintains perfect precision at 100%. Although the 352-bug dataset proves insufficient to establish comprehensive conclusions, the precision demonstrated is of considerable significance, suggesting that GPT-2-generated patches offer exceptional reliability, potentially minimizing the likelihood of introducing additional defects. Moreover, following the initial LLM training phase (which can be computationally expensive, particularly compared to genetic approaches that do not require preliminary training), these models can produce repair recommendations with remarkable speed, while search-based techniques often require extended execution periods due to their inherently iterative search-based nature.

Although LLMs have the potential for APR, especially in applications that require

high precision, additional research remains essential to improve their generalization capabilities. Future investigations might examine the combination of LLMs with search-based or evolutionary methodologies to harness the advantages of both approaches. For example, a LLM could generate initial patch candidates for subsequent refinement through genetic algorithms, or alternatively create integrated systems capable of addressing broader bug varieties while maintaining both efficiency and accuracy.

1.6.2 An Example: Express 2

As mentioned previously, GenProgJS logs its repair trace, allowing further analysis of the obtained results. To do so, we manually picked an example of the repaired programs. It should be emphasized that this sample is from a rather huge dataset (from a human perspective), and on the sole basis of this, definite conclusions should not be drawn from this example alone. However, with further investigations, one might get an idea of how the general algorithm transformed the original source code.

Figure 1.7 illustrates the operator trace for Express 2. The trace route starts from the original source code and results in a program variant which passes all tests. As can be seen, as generations pass, the source code evolves as well: usually it becomes longer and more complex. Though in some cases an operator can remove statements from the source code or trim parts of it, our observations showed that the more advanced a population is, the more complicated the generated code becomes. By complication, we mean that the patch becomes less readable, which might be an important characteristic of a patch in real-world applications, considering maintenance and future development. Thus, reaching the first repair candidate in the process is a breakpoint in several aspects. First of all, from now on the genetic algorithm will keep this specific candidate alive with high probability. This also implies that mutation and crossover operators might create new candidates based on this variant, which of course have a higher chance to be correct as well; however, this does not promote diversification anymore.

The first patch generated for Express 2 was created in just under 1 hour, shown in Code 1.6.1. Based on this patch, we cannot be absolutely sure that a fix has actually been made here. What is certain is that the test execution proved to be correct on this variant, but from the developer's point of view it does not seem to be a valid patch. Only two operators made changes to the patch: `CallChanger` and `NullCheck`. The former made changes in the arguments of the function call, while the latter is responsible for the multiple appearance of the function signature and for the logical "and" sign. This example also shows the unique nature of JavaScript: (1) the `trust` function is called with different arguments (it is possible even if the function is not overloaded) and (2) the return type of the `trust` function call is not necessarily

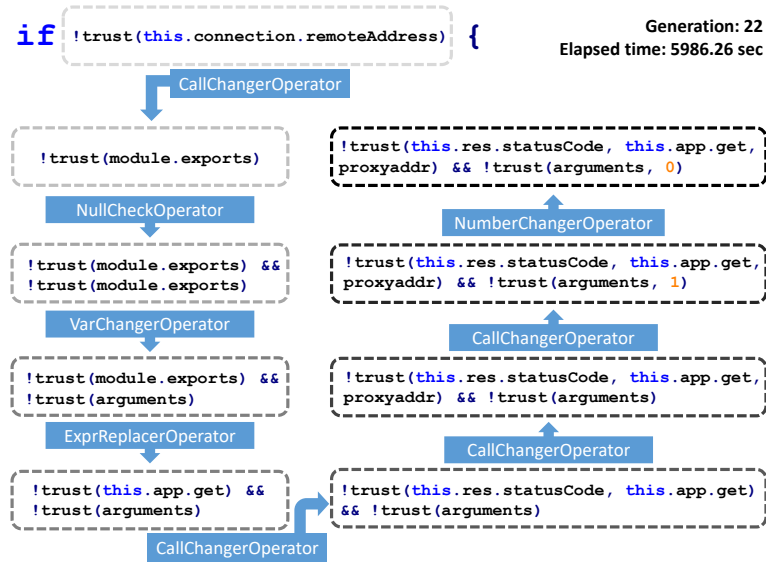


Figure 1.7: Trace of operators applied on Express 2.

boolean, it can be of any type. In the next section we further investigate the unique characteristics of the repaired patches.

Code 1.6.1: (-) Original code of Express 2, (+) the first automatically generated patch and (>) developer fix.

```

var trust = this.app.get('trust proxy fn');
- if (!trust(this.connection.remoteAddress)) {
+ if (!trust(' ', 0) && !trust('acceptedCharsets')) {
> if (!trust(this.connection.remoteAddress, 0)) {
  return proto;
}

```

1.6.3 Repairing on Script Language

JavaScript does not compile and is weakly-typed, thus tolerates partially incorrect programs and is able to run them also. This removes part of the candidate validation that the compiler typically offers for other languages such as C and Java. This lack of validation prolongs the process of patch synthesis (since the patches only rely on test executions); however, the algorithm is able to generate a whole new class of plausible patches, which are fundamentally different from repairing strongly typed compiled languages. In this section, we introduce some of these generated patches and discuss the difficulties and limitations of *Automated Program Repair* on JavaScript.

While it is generally considered good practice in scripting languages to avoid changing the type of a variable after its initial assignment, JavaScript does not en-

Table 1.8: Bugs and their corresponding plausible patches.

Bug Id	Operator	Original line	Plausible patch
Eslint 1	StringCut	if (name === "Math" name === "JSON"	if (name >= "Math" name >= "S")
Eslint 72	NullCheck	loc: lastItem.loc.end	loc: penultimateToken.loc.end node.elements && lastItem.loc.end
Eslint 130	CallChanger	notEmpty.forEach(function(x, i) { trimmedLines[i] = x;	notEmpty.forEach(function(i) { trimmedLines[i] = context;
Express 3	ShiftOpChanger	var key = req.accepts(keys);	const key = !! keys.length »re && req.accepts(keys);
Karma 4	FunctionMaker	if (file && file.sourceMap)	if (file && (line ").replace()
Mongoose 8	ReturnInserter	this.constructor.update.apply(this.constructor, args)	return this.constructor.update.apply(this.constructor, args, args)
Bower 2	ArithmeticOpChg.	if (that._options.save that._options.saveDev)	if (that._jsonFile this._manager * _direct)

The first column is the unique identifier of the bug. Next to it the name of the operator is highlighted that created the patch which we found specific for JavaScript. The remaining columns list the original source line and the automatically generated patch. Code removals and additions are highlighted with red and green background respectively.

force this convention. Consequently, we did not restrict the algorithm to only selecting variables with the same initialization type as the one being manipulated. For example, the operator responsible for modifying variables does not consider the original type of the variable, nor does it discriminate against candidate variables based on their types. Due to the genetic nature of the driver algorithm, even if a variable's type changes, it is still selected as a possible candidate without any bias regarding this property.

Table 1.8 shows some of the patches generated, which we found to be specific to JavaScript. Genetic operators in general do not take context into account: if an integer variable is compared against a number, an operator might replace the variable with another one, which holds a string value, for instance. However, in case of repair of strongly typed programming languages such as Java or C++, the developer of the APR tool can take this information into account and design the operator to maintain the type of the variable as well. The types of variables can also help to change conditional expressions, since most languages require the condition to be a type of boolean or integer. In JavaScript this intuition cannot or rather should not guide the search process. For example, comparing two strings or a string and an integer with the <= operator makes perfect sense, as demonstrated in the first row of Table 1.8. In the rows of Eslint 72 and Karma 4 we can observe the use of the logical or operator (||) that in these cases serves as a kind of default value assignment. If we consider the case of Eslint 72, the assignment works as follows: first, the value of `penultimateToken.loc.end` is tested and if it is defined, the execution goes with it; otherwise, the rest of the condition will be evaluated. Another aggravating

circumstance for scripting languages is that it is much more difficult to examine the existence of an (overloaded) function. The example of Mongoose 8 demonstrates this phenomenon: the argument list of `apply` cannot be obtained. Moreover, it can be called with an arbitrary number of parameters: if we do not specify some of the parameters, they will be undefined, but the function call is syntactically correct. In contrast, if too many arguments are given, the leftover will not be taken into account in the function.

Overall, we can conclude that for script languages the search space is orders of magnitude larger than for compiled languages. This is mainly due to the fact that these languages are generally more permissive, giving operators more space to operate. Of course, if the operators are of such that do not leverage the restricting factors detailed above, the repair process is essentially the same for compiled and script languages (except for candidate validation offered by the compiler). However, the possibility of creating more intelligent repair operators means a huge advantage as the search space is definitely huge. Another aspect of JavaScript program repair is its frequent interaction with the DOM. In this work, we were unable to focus on this kind of defects, since the BugsJS dataset does not contain bugs related to DOM manipulations. However, most string manipulating operators are suitable for repairing such dynamic errors.

1.6.4 Multi-line repairs

As described in Section 1.4.3 and Table 1.3 we designed batch operators with the aim of generating patches that span multiple lines. In addition, the original "genetic setup" is also implemented, which allows the creation of multiline fixes. Although batch operators are not able to generate multi-hunk fixes, we thought the tool could generate repairs more efficiently with these special operators (e.g., inserting a default branch in a switch-case and/or protecting a statement with a try-catch block could make sense). These approaches were used continuously in the search process. Of the 31 generated patches, 5 is a multi-line fix, thereby confirming that the approach is widely applicable. Since the human-written developer fix is available for every bug in the dataset, we compared these with the generated patches. We observed that there are cases where (1) the developer modified more lines than the GenProgJS tool and (2) the other way around: the generated patch includes multiple lines. Although the developer fix is only a single line fix (usually it was a result of a batch operator and not the standard genetic approach), we found the first option to be more interesting and depicted an example in Code 1.6.2.

The key insight of Code 1.6.2 is that the developer modified three lines in the patch, while the GenProgJS tool only one (the autogenerated patch is more concise than the developer fix). However, the generated patch is clearly incorrect, the tests

ran successfully (making it a plausible patch only). An interesting research direction may be to investigate such patches; however, this goes beyond the present study. In our experiments those automatically generated patches that modified fewer lines than the developer were all incorrect ones. This, of course, may be due to the work of chance or the unique characteristics of the BugsJS database.

Code 1.6.2: (-) Original code of Eslint 7, (+) generated patch and (>) developer fix.

```

} else if(allowDangle === "always-multiline"){
> lastTokenOnNewLine = node.loc.end.line !== penultimateToken.loc.end.line;
- if(hasDanglingComma && !nodeIsMultiLine){
> if(hasDanglingComma && !lastTokenOnNewLine){
context.report(lastItem, penultimateToken.loc.start, UNEXPECTED_MESSAGE);
+ hasDanglingComma = penultimateToken.value === "~,";
- } else if(!hasDanglingComma && nodeIsMultiLine){
> } else if(!hasDanglingComma && lastTokenOnNewLine){
context.report(lastItem, penultimateToken.loc.end, MISSING_MESSAGE);

```

1.6.5 Insights Into the Searching Process

In this section, we discuss the use of operators and the way GenProgJS found plausible patches in the search process. Figure 1.5 presents the behavior of the genetic algorithm, where the horizontal axis corresponds to the number of generations, while the vertical axis corresponds to the number of repair candidates. The values along the horizontal axis are averaged for the bugs. It is clear from this diagram that on average the same number of candidates was created in each generation (roughly between 3 and 12 candidates). This might be confusing because according to this result we can assume that a candidate for each bug is being generated in the first generation. The truth is that the figure is highly biased due to the fact that for different bugs the number of patches generated varies widely, and for *easier repairs* the tool already created patches in the first generation. For example, in the case of Eslint 323, 12 candidates were generated in the first generation, while in the case of Eslint 321 the first repair occurs only in the 23th generation. It is also apparent from our analysis that for those bugs, for which the first plausible patch was assembled in an early phase of the process, later generations created even more patches, thus obtaining a large set of repair candidates (see Table 1.6). The identification of the class of *easier repairs* mentioned above is beyond the scope of this work; however, the determination of these bugs might speed up the search process.

Another option to speed up the algorithm is to use only those mutation operators that seem to make an actual difference. That is, to give higher probability to those repair operators that eventually create patches and lower application probability to

those who do not. The speedup is due to a very simple reason: if an operator does not create program variants, the testing time of these can be spared. Although it is true that for different bug types different operators are needed, we wanted to investigate whether there is a "golden set" of genetic operators which will predictably be used in a patch candidate. To do so, we observed the distribution of the applied operators on plausible patches, which is described in the next section.

1.6.6 Operator Expedience

Choosing which operator to use is of great interest, but a far-reaching conclusion can only be drawn with a sufficiently large database. The characteristics of the BugsJS database or the JavaScript language might imply the use of some operators, thus further examination needed in the field with the involvement of other databases and programming languages as well. We know that in our case each repair category is represented by a very limited number of bugs, but the correlation of some of the fix types and operators is easily visible and clearly explainable. Thus, we attempt to illustrate the reviewed correct patches in a figure and examine the relationship between operators and fix types.

The resulting graph can be found in Figure 1.8. Operators are listed on the vertical axis of the heatmap, while on the horizontal axis the repair types can be observed. Since the number of correctly repaired candidates varied widely, the usage of operators was averaged per bug. That is, for example, in the case of a fix and two operators, let us consider that operator p_1 was used 10 times. Operator p_2 was used 15 times and a total of 10 candidates were generated. Then p_1 would have the value of 1 in the figure, while p_2 would have the value of 1.5. For the same reason, on the heatmap not just integer values are indicated (which correspond to operator usages). The darker the color, the more times an operator was used for a given fix type.

What we can see is that for each fix type which operator made attempt to repair the bug (and actually during the process the bug was repaired). So, for example, in the case of the fix type *Replacing assignment with return*, only two operators are used: *CallChanger* and *ReturnInserter* and it is clear that the latter one was used in most of the cases (its coloring is much darker). This connection seems to be quite obvious and, in fact, it is, but highlighting the coherency of the operators and fix types further validates our work and may lead us to a better understanding of the whole process. However, the case of *Change of If Condition Expression* is less self-evident: almost every operator made changes to the candidate patches. This clearly implies the fact that there was no single operator that was suited to repair this type of erroneous behavior. However, the reason for this is more nuanced. On the one hand, changing an if expression can involve a variety of changes; replacing an assignment with a return statement is quite straightforward, while changing an if expression is much

more general. However, the operators that were used most of the time here make sense as well.

- *CallChanger*: is included in most types of repairs; we attribute this to the fact that functions are heavily used in JavaScript programs.
- *ConditionalBinaryOpChanger*, *BinaryOpChanger* and *LogicalExprChanger*: changing binary and/or logical operators in an if expression makes sense - this is basically the goal of these repair operators.
- *NullCheck*: it is obvious that checking the value of a variable also makes sense in an if expression. However, thanks to the unique nature of JavaScript, this operator can be used in many scenarios - for example, for default value assignment - probably that is why it is present in other fix types as well.

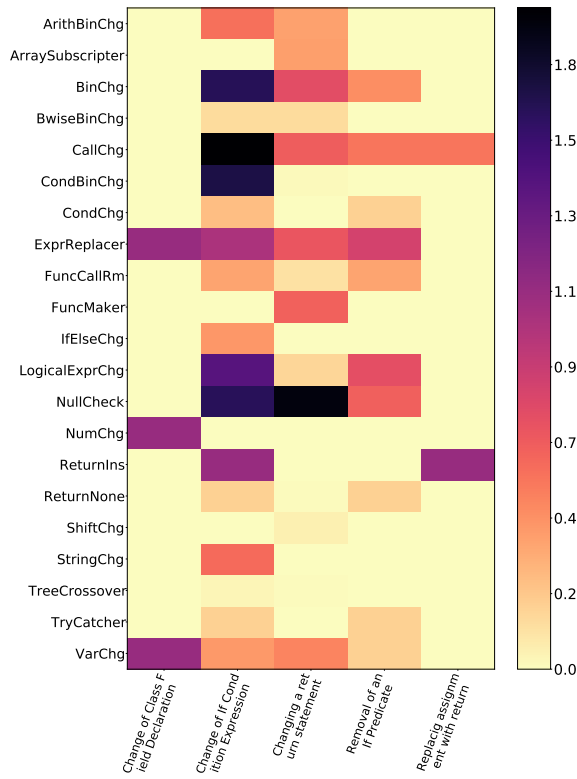


Figure 1.8: Operator usage per fix type.

Defining the operator set that will generate the correct fixes is impossible in practice, at least for now. As we saw in Figure 1.8 there are fix types for which it is quite easy to create a genetic operator, while for others it is nearly impossible to do so. Defining newer and newer operators might enhance the performance of some repair tools on a specific dataset (like BugsJS or Defect4J); however, tailoring operators

to databases is not forward-looking. The more operators are present in the repair process, the probability of each of them to operate will drop accordingly. One more complicating factor is that before creating a patch, in a real-world scenario, no one knows neither the type of bug nor the type of the fix. Nevertheless, we can argue that in some cases (e.g., when the test case clearly fails under an if condition), it may be worthwhile to use certain operators in a targeted way. In these cases, targeted operators are more likely to synthesize a patch than others. Of course, the question arises about the use of completely generic operators such as *ExprReplacer* or *ExprStatementInserter*. These kind of operators are able to fix in theory any kind of bug, but since the search space is large, they usually fail to find a plausible patch. However, their use is definitely recommended, as only they are able to generalize to some extent in the traditional genetic approach. Finally, we would like to emphasize that in this experiment we considered only those operator usages which resulted in a correct patch (and not plausible one).

1.7 Threats to Validity

GenProgJS and hence the developed operators carry several limitations that should be acknowledged. We introduced GenProgJS by conducting experiments solely on one dataset. At the time, BugJS was the only dataset that we could find that met all our criteria (e.g., that was runnable, reproducible, and in JavaScript language). While BugJS is a well-curated benchmark featuring reproducible bugs from actively maintained Node.js projects, it is also true that its focus remains on server-side applications. This fact also reflects the reality that front-end applications are usually not as thoroughly tested as server-side projects [48]. Nevertheless, these facts remain, limiting the conclusions that could be drawn about the repair if client-side applications, and also the possible differences in the usefulness of operator usage in such software.

Another threat that could come from using only the BugsJS dataset is the fact that since we only had one dataset to rely on, we drawn many conclusions on operator design based on this dataset alone, making it possibly biased toward projects that are readily present in the BugsJS benchmark. Although we tried our best to avoid such issues, by designing new operators to be as generally applicable to the JavaScript language as possible, such possibility of underlying bias should not be forgotten.

Although these threats impose a certain range of constraints on conclusions that can be drawn, they also show possible areas for further exploration and refinement in the pursuit of more intelligent and broadly applicable JavaScript repair pipelines.

1.8 Discussion and Concluding Remarks

We introduced *GenProgJS*, a test-based *Automated Program Repair* (APR) tool for JavaScript, using a *Genetic Algorithm* as its driving engine. Inspired by similar tools developed for C and Java, GenProgJS adapts these techniques but also includes operators specific to JavaScript and its overall script-like features to address the unique characteristics of the language. To the best of our knowledge, there is no prior generic test-based repair tool that was specifically designed for JavaScript, motivating the development of this tool to be able to serve as a baseline for future work. In our initial experiments, GenProgJS produced plausible patches for 31 real-world bugs across 6 Node.js projects, achieving results that are on par with previous works (in terms of generated number of plausible patches and fixed source codes) conducted in other programming languages.

To put our work in the context of today's LLMs, we also tested a fine-tuned version of GPT-2 on the same dataset. Interestingly, the GPT-2 model underperformed compared to GenProgJS in terms of the number of plausible patches; however, the two approaches show complementary behavior with each other, one identifying the correct patches that the other missed. This complementarity underscores the potential of hybrid systems that integrate search-based and learning-based repair strategies. Although we cannot ignore the inherent randomness of GenProgJS, in the sense of complementary behavior, as different runs could, in fact, produce different outcomes.

Another persistent challenge in APR is to determine the correctness of the automatically generated patches. One crucial step toward increasing trust in the APR tools is the removal of incorrect but plausible patches. To this end, we explored a similarity-based patch filtering technique using Doc2Vec embeddings to measure patch similarity. This approach may offer some leverage in cases where there is a notably high number of plausible patches—by providing a filtering criterion for generated patches—our results suggest that plain-code embeddings can fail to capture deeper code semantics and hence do to capture the true "source code" similarity of these patches. For this reason, more advanced models that possess stronger language and/or code understanding capabilities may be necessary to accurately distinguish between correct and incorrect patches.

GenProgJS gives a strong foundation for more research on the repair of JavaScript programs. The work also includes the analysis of generated patches and how it compares to a more modern approach, a fine-tuned GPT-2 model. Together with the publicized source code to promote future research, our results show the potential of more traditional heuristic-based approaches. Future work may include extending repair pipelines by adding more intelligent patch-correctness assessments. *Neural Networks* could further enhance the repair performance if combined with such techniques, as they could potentially repair a different class of bugs, as seen in this work.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

1. Novel genetic operators that are suitable for the JavaScript programming language.
2. Experimental results showing that GenProgJS can efficiently repair errors in 31 JavaScript programs, of which in 12 cases the generated patches are semantically similar to the human patch.
3. A qualitative and quantitative analysis of the generated patches, demonstrating the strengths and weaknesses of the applied genetic operators.
4. Significant contribution to creating GenProgJS, an open source tool that uses *Genetic Algorithm* to generate patches for bugs in JavaScript programs.
5. Analysis of patches generated for the Eslint project on 10 different bugs, via generating document embeddings, and calculating distances between the original faulty code and the generated plausible versions.
6. Laying the groundwork for pre-trained GPT-2 experiments in the form of data processing steps, and dataset mining, to provide the overall dataset on which the model can be fine-tuned.

This thesis point is based on scientific papers published by the author:

- Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. "GenProgJS: a Baseline System for Test-based Automated Repair of JavaScript Programs." *IEEE Transactions on Software Engineering* 01 (2024): 1-19. [29]
- Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. "Exploring plausible patches using source code embeddings in javascript." In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 11-18. IEEE, 2021. [27]
- Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. "Fine-tuning gpt-2 to patch programs, is it worth it?." In *International Conference on Computational Science and Its Applications*, pp. 79-91. Cham: Springer International Publishing, 2022. [93]

Chapter 2

Representation Matters: *Deep Learning*-Based Program Repair with Effective Code Encodings

2.1 Introduction

Previously, in Section 1.1, we looked at the current state and progress of *Automated Program Repair* (APR), focusing on how repair tools generate and validate patches. In this part of the thesis, the attention is shifted to a different —and commonly overlooked— aspect of code repair, that is, the way we represent source code and its effects on the performance of *Deep Learning*-based APR. Although many existing research focuses its attention on model architecture, and training techniques, the representation of source code, and its possible effects are often overlooked. This work highlights the importance of code representation and why it matters, advising on approaches of source code encoding that could possibly lead to better repair outcomes.

2.1.1 Motivation

The main motivation is to explore the effects of different program representations in the case of multiple *Deep Learning*-based models, with particular attention to transformer-based architectures. Despite many advancements in architectural designs, the impact of source code representation, whether abstracted, structured, or simply tokenized, is highly underrepresented in this domain. We investigate how varying representations influence repair outcomes and whether their effects generalize across different programming languages and model configurations. To this end, we evaluated multiple transformer-based models trained on three distinct source code representations, across two datasets written in Java and JavaScript, respec-

tively. Our goal is to provide empirical evidence and actionable information on the crucial but underappreciated role of input encoding in APR. All training pipelines, data splits, and code are available in an open-source repository ¹.

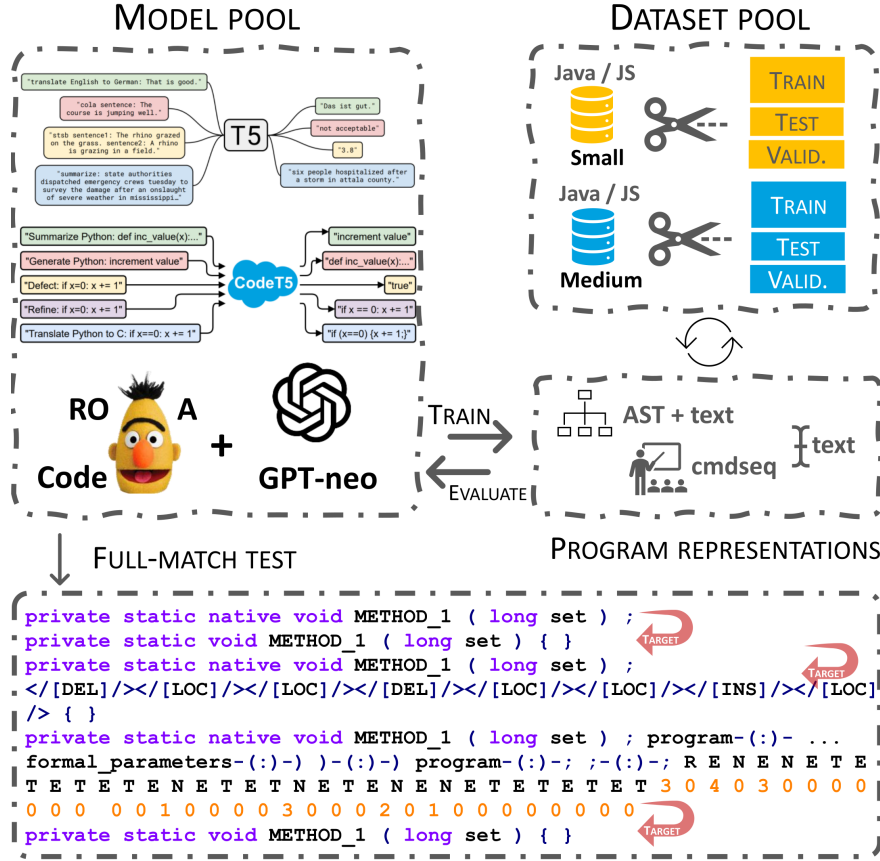


Figure 2.1: High-level approach overview of source code representation experiments.

Data-driven repair approaches [122] have recently shown promising results [25, 37, 84, 108, 190] in software engineering research. However, the most commonly used datasets for APR research (Defects4J [86], QuixBugs [189], ManyBugs [98]) have some limitations, such as being too small and not preprocessed for *Deep Learning*.

To tackle this problem, Tufano *et al.* [167] introduced a large dataset extracted from GitHub [57] commits that has become a benchmark [106] for various tasks. However, this dataset only considers learning on Java code, which limits researchers' ability to test configurations in a multilingual environment and poses significant limitations to research questions such as the ability to transfer knowledge [91] learned in one programming language to another. In addition, the dataset uses an abstracted code format [31, 167]. However, current DL models can handle large vocabular-

¹<https://github.com/AAI-USZ/APR23-representations>

ies, which may motivate the use of non-abstracted code so that models can produce usable code as-is.

Software development is a complex process that involves several stages, from the writing of code to the delivery of the final product to customers. For developers, version control systems like Git play a vital role in developing the final product. When developers complete a feature or fix a bug, they commit code changes to the version control system. Committing code involves creating a snapshot of the changes made to the codebase. This snapshot —among various other information— includes the changes made to the code and a description of the changes. The committed code is then pushed to a remote repository, *e.g.*, GitHub, where other team members can access it.

More recent work such as *PreciseBugCollector* [69] or *GitBug-Actions* [149] employs more sophisticated methods for bugfix collection. However, many seminal works [25, 39, 109, 167] use simple text-based verification of whether a commit is considered a bugfix or not. While more sophisticated methods could yield better quality bugfixes, it also limits one’s ability for large-scale mining, since it greatly restricts bugfix commit availability, *e.g.*, only commits that rely on relatively new features (such as GitHub-actions) are considered for mining.

Another issue with many of the datasets available is that they store bugfixes in raw format, without the employment of some sort of pre-processing, such as extracting smaller, but complete segments of the code, like methods or classes. *CodegrainHouse* separates these code fragments from the rest of the code. Earlier work also depended on such features, such as Lajko *et al.* [93], where only a single line of code was needed as a training target to fine-tune GPT-2. However, if one should need it, diff information is also available in the dataset which contains the modifications applied to the entire source file.

To promote future studies in the direction of source code representation in the application of *Automated Program Repair*, we propose the utility of *CodeSieve* and *CodegrainHouse*, a newly mined dataset that features our technique. *CodegrainHouse* is a multilinguistic data set consisting of five popular programming languages, extracted from more than 5,000 GitHub repositories. It contains 371,483 methods extracted from code from 250,090 bug-fixing commits. The dataset also contains the commit message, allowing researchers to not only generate the fixed code, but also include an explanation of the generated fix, which could motivate researchers to make DL models to also generate commit messages [184] when producing a fix, which also helps the automation and documentation process. *CodegrainHouse* is larger and more detailed compared to similar datasets publicly available on code repair. The data set is available on Zenodo ² in a JSONL format, which simply contains JSON-compatible strings separated by line breaks. Our methods used to extract func-

²<https://zenodo.org/records/10198721>

tions from commit diffs are also available as a small package at our public GitHub repository ³.

2.2 Related Work

Recent advancements in *Deep Learning* —particularly transformer-based architectures— brought great developments in a wide range of fields, including code repair related tasks. Many tools employ transformer-based solutions, which has proven to be an effective solution in multiple software engineering applications. Mastropaolo *et al.* [116] fine-tune a T5 model for several downstream tasks, including bug fixing, injection of code mutants, and also assertion statement and comment generation tasks. Others, like Chen *et al.* [24] focus on repairing software vulnerabilities by using transfer learning. In their studies, they train a Transformer model on a bug-fixing dataset and then they fine-tune it on a smaller vulnerability fix dataset. Their findings show the importance of model pretraining, particularly vulnerabilities found in software written in C.

Variations of the Transformer have also been proposed to better handle code structure. Yu *et al.* [193] introduce a grammar-based rule-to-rule model that employs two encoders to capture both the token sequence and the corresponding grammar rules, enhanced by a novel tree-based self-attention mechanism. Their approach outperforms previous baselines in code generation tasks. Another notable work is *DeepDebug* [39], where the authors used pre-trained transformers to fix bugs automatically. Here Drain *et al.* use the standard transformer-based encoder-decoder architecture with copy-attention. They conducted several experiments, including training from scratch, pretraining in Java or English, or both, using different embeddings. They achieved their best results when the model was pre-trained on both English and Java with an additional syntax token prediction. This model is evaluated on the dataset by Tufano *et al.* [167], which was later included in the *CodeXGLUE* (General Language Understanding Evaluation benchmark for CODE) benchmark [106].

The CodeXGLUE benchmark [106] offers a comprehensive evaluation suite for code intelligence tools, including a task named *Code Refinement (Code-Code)* that is closely related to data-driven program repair. Among the leading approaches is NSEdit [76], which predicts an edit sequence using the encoder and the decoder part of the Transformer. Their grammar consists of two actions—delete and insert, included as special tokens in the vocabulary. NSEdit achieves a state-of-the-art accuracy of 24.04 % on the small and 13.87 % on the medium dataset on the CodeXGLUE benchmark.

A key challenge in APR lies in how code is represented before being processed

³<https://github.com/AAI-USZ/codesieve>

by a DL model. Representations such as *Abstract Syntax Trees* (ASTs) are frequently used to encode syntactic structure, yet their usefulness varies depending on the task. Kim *et al.* [91], for instance, show that AST-aware Transformers improve performance of predicting the next token in autocompletion systems. In a related direction, Dinella *et al.* [37] introduce Hoppity, a graph-based *Neural Network* trained to predict AST-level changes in JavaScript. Although trained on more than 290,000 code change commits, neither the dataset nor the implementation is publicly available for replication.

Representation choices have also been specifically investigated for their impact on repair performance. Namavar *et al.* [125] examine the effect of 21 different generative models and code abstractions, including 14 homogeneous representations, 4 mixed-type representations, and 3 different embedding strategies. Their study focuses on name-based bugs and offers insights into abstraction levels in token and AST-based inputs. Unlike their work, which uses an NMT model, we rely on transformer-based architectures. Similar explorations of representation have also been conducted in other software engineering domains [28].

In addition to modeling approaches, numerous datasets have been curated to support the training and evaluation of program repair models. Several benchmarks offer high-quality, manually curated bugfix examples. *Defects4J* [86], *BugsJS* [66], and *BugsInPy* [177] cover Java, JavaScript, and Python, respectively, and are commonly used in both test-based and learning-based APR methods.

The *Codeflaws* dataset [162] benchmark contains 7,436 programs sourced from the Codeforces online database. This dataset includes 3,902 defects categorized into 40 different groups of defects. As a result, it allows for researchers to apply and test a wide range of repair tools on these bugs. Similarly, *QuixBugs* [189] comprises 40 programs from the Quixey Challenge translated into Python and Java. It is a collection of one-liner bugs defect collections, with corresponding test cases. Bugs in this dataset can be categorized into 14 different classes. The benchmark can be used to study the performance of multi-language repair tools.

ManySStuBs4J [87] focuses on simple Java bug fixes retrieved from GitHub. It identifies likely bugfixes based on commit messages and filters for minimal changes, categorizing them into 16 syntactic templates. This dataset is similar to our work, which also restricts changes to a single file per commit.

Security-specific datasets have also been developed. *Vul4J* [17] aggregates 79 reproducible vulnerabilities from 51 Java projects, each paired with a proof-of-vulnerability (PoV) test case. The dataset is derived from the Project KB knowledge base [138], which contains 1,803 fix commits addressing 912 real-world issues.

Datasets focused on JavaScript are less common, but *FixJS* [31] fills this gap. Similarly to the Java dataset [167], it filters GitHub commits using bug-related keywords to identify JavaScript bugfixes. This approach is also used in our work for data

collection.

GitBugActions [149] is a tool for building new datasets, that rely on GitHub Actions to detect and mine bugfixes including reproducible test suites. The main focus of the tool is bugs that have corresponding test suites which can be collected by locally executing them in the environment specified by the developers in the GitHub Actions workflow.

Lastly, *PreciseBugCollector* [69] is a dataset focusing on bugfix quality. They employ a precise bug collection approach comprising a bug tracker and a bug injector component. They extracted 1,057,818 bugs from 2,968 open-source projects, where 12,602 bugs come directly from bug repositories (such as NVD and OSS-Fuzz), while the remaining 1,045,216 bugs are injected via bug injection tools.

Only together can modeling techniques and datasets provide a strong foundation for training effective *Deep Learning* models. However, as we show, there is a strong connection between how data is represented and the effectiveness of trained models. Understanding this connection between program representation and the effectiveness of *Deep Learning*-based repair techniques could potentially lead future researchers in the direction of designing and training more successful APR tools. In the next section, we dive deeper into the methods used to train various models on different representations of multiple datasets.

2.3 Method

In Figure 2.1 we depicted the high-level approach of this work. First, we created a "model pool" consisting of different Transformer model architectures including T5, CodeT5, RoBERTa, and GPTNeo. Once a model is selected, it is trained with a matching code representation on each of the observed datasets (Java and JavaScript). After training, the models are evaluated using the standard evaluation procedure in learning-based APR approaches: The generated patch should be exactly the same as the one in the dataset (which is essentially the developer fix). In the following sections, we are going to introduce in detail the models and program representations used.

2.3.1 *Deep Learning* Models

In this work we experiment with different models, including T5 [144], CodeT5 [173], RoBERTa [191], CodeBERT [50], and GPTNeo [14]. T5-base and CodeT5-base are used mostly during our experimentation as pre-trained models for fine-tuning. When not using pre-trained models, we simply use the same configurations as T5-base or CodeT5-base, without initializing the model weights. However, since these models are designed to solve sequence-to-sequence tasks, when experimenting with ASTs,

to handle graph and source code information, we use RoBERTa to encode the graph part and CodeBERT to encode the source code part. These parts are concatenated, and then we use this embedding as the encoder state in GPTNeo, which acts as the decoder. When using these models, they are also initialized with pre-trained weights. Namely, for the program-encoder, we use codebert-base, and for GPTNeo we use gpt-neo-125M. Graph-encoder is initialized with empty weights since it has a custom vocabulary. In the following sections, we are going to briefly present these models.

T5 and CodeT5

Text-to-Text Transfer Transformer (T5), is a transformer-based architecture, first introduced by Raffel *et al.* [144]. It was trained on a variety of natural language text scraped from the web, introducing the dataset "Colossal Clean Crawled Corpus" (C4) [144], a dataset consisting of hundreds of gigabytes of clean English text. Every task—including translation, question answering, and classification—is fed to the model as text input training it to predict the output text. With this, the authors use the same model, and training methodologies on a diverse set of tasks. Compared to BERT, T5 adds a causal decoder to the bidirectional architecture and replaces the fill-in-the-blank cloze task with a mix of alternative pretraining tasks. CodeT5 has the same architecture, as the previously mentioned T5 model, and has been trained on CodeSearchNet [78]. It was further trained on a large number of public GitHub repositories written in C/C#. Transformer models have proven to excel in many *Natural Language Processing* (NLP) tasks, and while processing source code is different, they share many similarities.

CodeBERT, RoBERTa and GPTNeo

RoBERTa (Robustly Optimized BERT Approach) and CodeBERT have a very similar architecture to BERT, as it is derived from the same base model. Basically, it is an encoder part of a transformer model and has been proven to be a powerful tool for encoding. One key difference between RoBERTa and BERT is that RoBERTa was trained on a dataset of 160GB of text, which is more than 10 times larger than the dataset used to train BERT. Additionally, RoBERTa uses a dynamic masking technique during training that helps the model learn more robust and generalizable representations of words. To encode source code, as a pre-trained model, we chose codebert-base, which is an improved version of RoBERTa for encoding source code information [50]. To encode graph information, we used a customized RoBERTa model, with a different vocabulary size that fits the number of tokens represented by graph nodes and node-to-edge connections. Aside from this change, the model is the same as that used for text encoding. For the decoder, we use GPT-Neo [14], which is an open-source implementation of GPT-3 (Generative Pre-trained Transformer). Its architecture is

based on the Transformer, which is an attention model - it learns to focus attention on the previous words that are the most relevant to the task at hand: predicting the next word in the sentence. GPT models proven to be highly efficient in generative tasks [14]. These models are combined to encode text and AST information from the source code and generate the fixed or improved version of it, using gpt-neo-125M as the decoder.

2.3.2 Program Representation

Source code representation significantly influences the effectiveness and efficiency of the repair algorithms. A common method involves representing source code as a variant of *Abstract Syntax Trees* (ASTs). These structures capture the syntactic structure of the code. They allow manipulation and transformation to generate repair patches. Other methods use lexical representations, such as token sequences or n-grams. In fact, more advanced techniques include graph-based representations or neural network models [37]. Researchers have also examined program slicing. This approach creates a reduced version of the program. It focuses on the relevant portion of the code for the repair task [81]. These representations can be applied alone or in combination. Their selection often depends on the specific repair task, the programming language, and available resources and constraints [61]. This work employs three program representations. We summarize them in Table 2.1. The following sections describe these representations in detail.

Text

One is used as the baseline representation and is also the most common. Simply representing the source code as **text** is a straightforward yet powerful enough representation. In text representation, the source and target code are simple series of tokens, where the model needs to learn how to transform the entire input to its fixed version. However, it is usually the case that fixed versions of source code are pretty similar to the buggy version [27]. For us this means that the decoder part of the model has to generate much of the redundant parts of the code.

Command Sequence

It is also possible to have the model generate only code patches, where changes are required in the given source code [76]. This representation will be referred to as **command sequence** (cmdseq). As input, it still contains the same information as in text representation, so the task for the encoder part of the models remains the same. However, when decoding, the model needs only to predict the necessary changes to

be made to the source, to make it identical to the target. For this purpose, three special tokens are added to the dictionary, [DELETE], [INSERT], and [LOCATION] tokens. Aside from this, the dictionary is the same as in text representation. For example, if a code segment needs to be cut from the source to generate the target, then upon learning, the model needs to predict something like: [DELETE] [LOCATION] [LOCATION]. The code patch to be deleted from the source. Location tokens need to be processed and filled with true positions, by another mechanism, *e.g.*, pointer networks [76]. However, in this work, we only use the raw location tokens to see if the decoder has an easier job figuring out targets represented by cmdseq only.

Abstract Syntax Tree + Text

The third and last code representation attempts to encode more information in the input. That is, we try to encode AST information, along with the same text information as in the text representation, referred to as **AST+text**. To represent the AST we use a representation of flattened nodes and edges, with added type information. For example, we encode whether the examined token is the root of the program, a node, an edge, or a terminal (leaf) node. We also include the count of children for every node, as part of the AST information, which can be thought of as a special kind of positional embedding. The models used in this work can handle such an encoding task with a minimal configuration; however, there exist other kinds of technique to work with tree- or graph-like structures [19, 194]. The task of the decoder remains the same as in text representation. We need to predict the whole target sequence from the source. However, in theory, the encoder can have more representative power, since it also has information about the AST representation of the program. This way the encoder consists of two main parts, one for encoding the AST information, which in our case a flattened graph, with some additional information, and two for encoding textual information given by the source code.

2.4 Experimental Setup

When optimizing the models, we chose a learning rate of $5e - 5$ [155], using an Adam optimizer, leaving the default epsilon of $1e - 8$. For the small datasets we used a maximum sequence length of 256 and a batch size of 16. For the medium dataset, we used a sequence length of 384 and batch size of 8 for training. This configuration is used throughout the conducted experiments, aside from ast+text representation, where the model was proven to be too large to use batches of this size on one GPU. To train the models we used an RTX 3090 GPU. The base models working with text, and command sequence representations are of size 222M parameters. The model used on ast+text representation needs more encoding power, and it has additional layers for

Table 2.1: *Examples of program representations.*

Source Rep.	Target Rep.	Description	Example
text	text	Code is represented as is. Simple sequential text input, and output.	<pre>private static native void METHOD_1 (long set) ; => private static void METHOD_1 (long set) { }</pre>
text	cmdseq-token	Input code is represented as text, target is represented as a sequence consisting of <i>DELETE</i> and <i>INSERT</i> commands	<pre>public void METHOD_1 (final boolean VAR_1) { VAR_2 . METHOD_1 (VAR_1) ; } => </[INS]/></[LOC]/> this .</pre>
ast+text	text	Input represented as chains of AST and textual info. The output is a text rep. of the fixed source code.	<pre>boolean METHOD_1 () { } program-(:)-program program-(:)-local_variable_declaration local_variable_declaration-(:)-local_variable_declaration local_variable_declaration-(:)-boolean_type boolean_type-(:)-boolean_type local_variable_declaration-(:)-variable_declarator variable_declarator-(:)-variable_declarator variable_declarator-(:)-identifier identifier-(:)-identifier identifier-(:)-; ;-(:-); formal_parameters-(:)-formal_parameters formal_parameters-(:)- ((:-) (formal_parameters-(:)-) -(:-) program-(:)-block block-(:)-block block-(:)-{ {(-)-{ block-(:)-} }-(:-)-} R E N E T E N E T E T N E T E T E N E T E T 3 0 3 0 0 0 1 0 0 0 0 2 0 0 0 0 0 2 0 0 0 0 => boolean METHOD_1 () ;</pre>

encoding AST information. For this reason, this model is considerably bigger, with 346M parameters.

All of the models in this work come from *transformers* library [178]. For the environment, we used Python 3.8, and *PyTorch* library along with *PyTorch-Lightning*. We trained for a maximum of 50 epochs, and used early stopping with a minimum delta of 0.05 on validation loss, with patience set to 8. This means that if a model could not reduce its loss by 0.05 in 8 epochs, then we assume that the model converged, stop the training, and load the best model so far for testing. The loss function used throughout the experiments is the torch implementation of cross-entropy loss. On the Java dataset [167] the training time per epoch was mostly about half an hour on the small dataset, and one hour on the medium dataset, on text and the cmdseq-token representation. Training on text+ast representation took about 1 hour and 40 minutes on the small, and 2 hours on the medium dataset on average. Training time on the FixJS dataset [31] was significantly lower, due to the size of the dataset. It only took about 5 and 13 minutes per epoch to train the model on the small, and on the medium sequences on text and cmdseq-token representation. Also, it only took 20 minutes to train on text+ast representation in both cases. In total, the models took from 1 hour to about 1 day to train, which varied greatly with the size of the model, the dataset, and the representation used.

2.4.1 Training Data

In this section we briefly introduce the data that is used for model training and model fine-tuning. We give a short introduction to each, and provide their most important aspects, similarities, and differences. This should give a greater perspective, so that the evaluation of results can be carried out with the following information in mind.

Java

For Java, we used the dataset released by Tufano *et al.* in [167]. Later it became part of the popular CodeXGLUE benchmark [155], and can be found under the code-refinement task. The dataset contains Java source codes mined from GitHub. The programs have been normalized in a way, that they do not contain variable, method, or function names, but they have been abstracted away. For example, if a code contains a variable named *myVar*, then it will be referred to as *VARIABLE_1*. This preprocessing step is done file by file, resetting the variable index to 0, thus reducing the dictionary size. However, the dataset defines some **idioms** - common identifiers in the observed projects - which they see as "part of the language". These idioms occur frequently in all of the source codes, so they are part of the vocabulary, along with the language-specific keywords, or reserved words.

JavaScript

As for the JavaScript experiments, we used FixJS [31], which contains bug-fixing information for GitHub commits. Similarly to the Java dataset, FixJS also uses abstraction for the variable names to reduce the vocabulary size, although it includes the raw commit information as well; thus, the original names can be obtained. Note that while duplicated samples can occur in the Java dataset, the authors of FixJS filtered their data, so there is a strong guarantee that samples with different input/same output or the same input/out pair cannot occur.

2.5 Need for Datasets

As discussed in Section 2.2, readily available datasets available for fine-tuning of LLMs—or other DL-based models—are scarce. Future work could potentially benefit or even extend upon results presented by experimental results given by the study of code representations, using the dataset presented in this work.

To give context, we will start with the discussion of the mining process. The task involves multiple steps necessary to extract the desired code snippets from the source code. First, we query GitHub’s API to find the repositories that meet our filtering criteria. Once we have located these repositories, we clone them to a specified location on our system. After that, we iterate through each repository while filtering for bug-fixing commits. Finally, we extract the desired code snippets from the source code and save them to a specified location. This step is important since we are only interested in code snippets that are related to bug fixes.

2.6 Mining

Online repositories such as GitHub [55] are an abundant source of source codes. During the mining process, we used GitHub’s API [58] to query for appropriate repositories. The only API endpoint required by this approach is to use the *repositories* endpoint <https://api.github.com/search/repositories> with the appropriate query parameters. Among other things, a high-level approach to this repository filtering process can be observed in Figure 2.2. The public API is only used in the first part of the process, where repositories are queried based on some criteria, *e.g.*, the publicity, number of stars, *etc.*

After collecting the necessary information on the repositories, such as the URL, the cloning process begins. For the cloning process, we used GitPython [60], a popular Python package to interact with the git repositories. We selected more than 5,000 repositories based on the following criteria; the repositories should be public and should not be mirrors, forks, templates, or archived. Furthermore, at the time of the

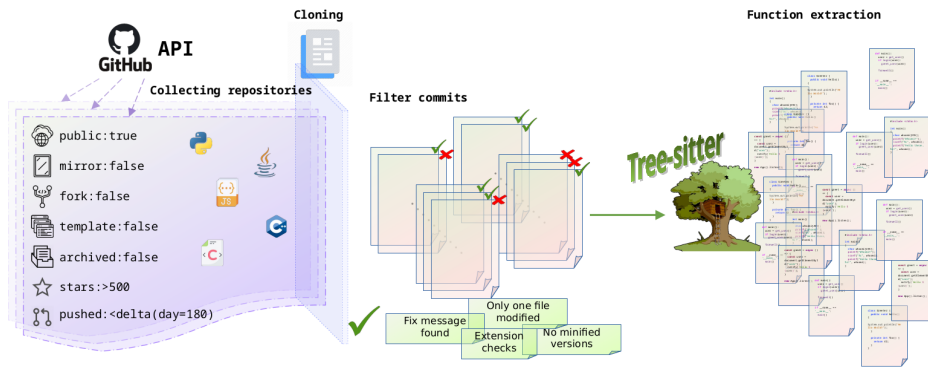


Figure 2.2: Overview of data mining procedure

collection process, there should be a push event not older than 180 days, signaling that the project is still being maintained.

We collected source codes written in five popular programming languages, C, C++, Java, JavaScript, and Python. These languages are also included in the API query criteria.

After the cloning process, one is able to freely iterate the commits without the use of GitHub's API, which we did try to minimize as much as possible throughout our experiments and the development of the data creation software. One reason for doing so is because the public API can be limiting [59] in the sense that it only allows for a small number of queries per hour. Specific commits are selected primarily by their commit message. If the message contains a "fix-like" word, such as "bug", "fix", or "improve" [167], then the commit is considered a bugfix. For our purposes, the commit should also conform to 3 other rules.

Firstly, there should be only one modified file by the commit, as currently, most APR techniques focus on single-file bugfixes.

Secondly, the modified file's extension should match one of the extensions commonly used by the specified languages. This precaution could be necessary to rule out cases where *e.g.*, only a JSON file is modified during a bugfix.

Lastly, the code should not be a minified version of itself. JavaScript code is often minified to speed up webpage loading to improve website experience. However, for our purposes, we are only interested in code that is being developed by humans, for humans. So, we rule out minified versions of the code.

Commits that abide by the above criteria are stored in JSON files. Most of the information is saved in the mentioned file, but it mainly contains the diff information, the buggy, and fixed version of the code, and the whole commit message.

Most importantly, the last step uses the Tree-sitter [166] library to extract specific code snippets from the source code. To the best of our knowledge, no other multilingual dataset uses the kind of post-processing we provide here. We generate

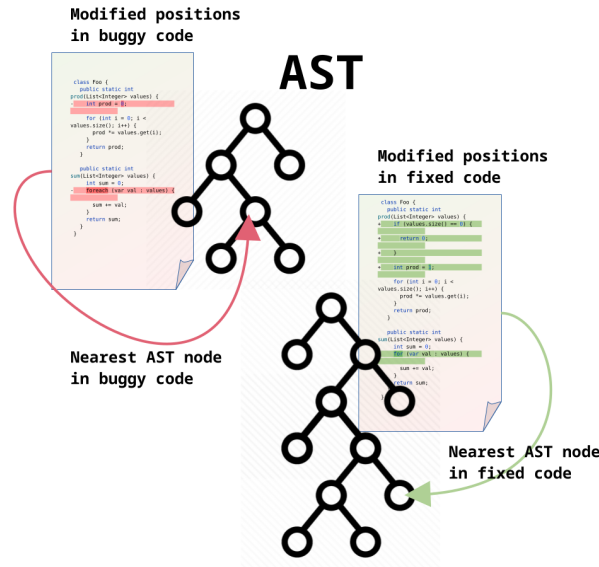


Figure 2.3: AST node position extraction.

an *Abstract Syntax Tree* (AST) representation of the buggy and fixed version of the source. We use the diff information [36] generated by comparing the fixed and buggy code, extracting the positions of the applied changes. In other words, we map the changes in the buggy file to the changes in the fixed version. This position mapping can then be used to find the closest AST nodes relative to the code changes applied, as shown in Figure 2.3. By moving up in the node hierarchy *i.e.*, querying the parent node by node, we check if a node’s type satisfies the condition of being of type, *e.g.*, function, method, or function_declaration. If the given condition is met, we store the code segment represented by that particular AST node.

Code 2.6.1: Java code example.

```

1  class AwesomeCalculator {
2      public static float divide(float a, float b) {
3          if (a == 0) {
4              return Double.NaN;
5          }
6          return a / b;
7      }
8  }

```

Consider the buggy Java program in Code 2.6.1 for a more concrete example. In this example, the fixed code should include the change in condition from $a == 0$ to $b == 0$, and from the diff information we can find its position inside the tree constructed by Tree-sitter, both in the buggy and fixed versions of the code, as shown in Code 2.6.2. From there we can climb up the tree, search for a method declaration,

and find the encapsulating function. This could be done independently of the programming language, so that this feature could easily be included in any source code mining software.

Code 2.6.2: Tree generated by Tree-sitter on Java code example 2.6.1

```

program [0, 0] - [8, 0]
class_declaration [0, 0] - [7, 1]
  name: identifier [0, 6] - [0, 23]
  body: class_body [0, 24] - [7, 1]
    method_declaration [1, 4] - [6, 5]
    ...
    parameters: formal_parameters [1, 30] - [1, 48]
    ...
    body: block [1, 49] - [6, 5]
    if_statement [2, 8] - [4, 9]
      condition: condition [2, 11] - [2, 19]
      ...
      consequence: block [2, 20] - [4, 9]
      ...
    return_statement [5, 8] - [5, 21]
    ...

```

The software could also be configured to catch the encapsulating class (or a given number of lines around a buggy segment, or any type of nodes), but for now, we chose to extract code segments on the function level, as it represents a relatively small, but complete code fragment. It is also notable that this method is language agnostic thanks to the Tree-sitter [166] library. The method presented here can also be found in our publicly available GitHub repository as an installable Python package. Outside of Tree-sitter, we do not have any special dependencies, and since it has bindings for many popular languages, our method can easily be implemented in those languages too, if one should have the need. In addition, any other existing data mining software could make use of the *function*-, *class*-, or *line-by-line* environment extraction from source codes without needing to worry about the programming language, thus providing them with yet another useful feature.

2.7 Results

In this section, a comprehensive evaluation of *Deep Learning*-based program repair is presented, showing how the selection of code representations and datasets affects model performance. We further discuss the performance differences between several models and investigate the effects of different code representations. In summary, we present the results that illustrate the importance of data representation in program repair tasks.

2.7.1 Dataset Overview

The experiments carried out in this thesis are highly dependent on the availability of high-quality datasets. In total, we collected more than 370,000 pairs of buggy-fixed functions and their corresponding commit messages. We also provide some metrics, which can be observed in Table 2.2. These metrics can help to give some insight into the size of separate parts of the dataset, which could provide useful information for future users.

Table 2.2: *Metrics describing the dataset.*

Language	#Samples	#Chars (mean)	#Chars (med)	Size (MiB)
C	63928	1612.50	885.0	196.65
C++	69905	1503.97	828.0	200.60
Java	69917	1075.18	654.5	143.58
JavaScript	66453	1196.82	541.5	151.86
Python	101280	1168.78	693.5	225.94
Summary	371483	1311.45	720.5	918.63

The table shows the number of buggy-fixed pairs extracted from commits of a specific language in column #Samples. #Chars mean, and median values provide information on the average source code length of the mined programs, while Size measures the UTF-8 encoded length of the source codes in mebibytes. The summary of sample and size information is a cumulative value of the separate samples, while both mean and median character values are averaged over the buggy and fixed codes.

Use Cases

The CodegrainHouse dataset provides preprocessed buggy functions that can be used for several *Deep Learning* and generative tasks. Some of these tasks include:

1. *Bugfix commit generation*: The dataset can be used to train *Deep Learning* models to generate commit messages [38] for bugfixes. The preprocessed buggy, fixed, or both code segments can act as input to the DL model, so it can learn and generate relevant commit messages.
2. *Code repair prediction*: The dataset can predict the changes required to fix [76, 106] the buggy functions. The preprocessed buggy functions serve as the input to the *Deep Learning* model, which can predict the appropriate code changes to repair the bugs.

3. *Code completion and suggestion*: The dataset can also be used for generative tasks, such as training *Deep Learning* models for code completion [106] and code suggestion.

4. *Buggy function detection*: The dataset can train *Deep Learning* models for buggy function detection tasks [7, 136]. The preprocessed buggy code sections can help the model learn the patterns in buggy code.

Example: for clarity, we show an example to provide more insight into the basic structure of the dataset. In Code 2.7.1 the buggy code could be observed, while Code 2.7.2 shows the fixed program. For the snippet that shows the fixed version, we put the bug-fix commit message on top; however, the dataset itself does not (necessarily) contain the commit message in the fixed source code in this format. For demonstration purposes only, here we mark such messages and include them as comments.

The examples in Code 2.7.1 and 2.7.2 show us a simple bugfix, where an unnecessary `"".join(...)` statement is removed, returning the file contents as an array. The dataset contains more complex and larger examples, too; however, for a short showcase of the bugs, we only included some that are short and relatively simple.

Code 2.7.1: Buggy Python function.

```
1  def readfile(filename):
2      try:
3          with open(filename, "r") as f:
4              line = "".join(f.readlines())
5          except Exception as e:
6              log.error('Failed to read file "%s": %s' % (filename, e))
7          return None
8          return line
```

The snippet shows the bug on line 4, where the lines read from file are concatenated, and later returned on line 8.

The snippet shows the bug on line 4, where the `readfile` function reads the file into a single string using `"".join(f.readlines())`, which returns the entire file content as a continuous string. In this version of the file, the problem is that downstream components expect this function to return the file contents as a list of lines.

Code 2.7.2: Fixed Python function.

Commit message: `firewall.functions.readfile`: Return lines read from the file as an array. This is needed for the `dbug` output in `ipXtables.set_rules`, `ebtables.set_rules` and `ipset.restore`.

```

1  def readfile(filename):
2      i = 1
3      try:
4          with open(filename, "r") as f:
5              return f.readlines()
6      except Exception as e:
7          log.error('Failed to read file "%s": %s' % (filename, e))
8      return None

```

The changes made to the fixed version ensure compatibility with functions such as `ipXtables.set_rules`, `ebtables.set_rules`, and `ipset.restore`, which rely on list-based inputs.

2.7.2 Dataset Limitations

Although the dataset introduced is large and diverse, it has its own set of limitations. Its intended use is to train and validate *Deep Learning* models. It is not suitable for *Generate and Validate* (G&V) approaches [69, 112], which require runnable test suites. Linking test cases to functions remains a challenge [30], and requiring tests would also reduce the number of usable repositories.

Another limitation of the presented dataset is that due to its size, it may also be insufficient on its own to train large LLMs [15, 165]. Currently, it supports five popular programming languages, which is a potential limitation on language diversity. While the dataset extraction process minimizes GitHub API usage, there is an increase in disk usage during repository cloning. Users should be aware that more than 100 GiB of temporary storage may be needed.

2.7.3 Code Representations

Models trained on Java [167], and FixJS [31] database. *T5-base, LM* is a pre-trained model, initialized from `t5-base`, with a language model on top. During training, only the LM-head has been optimized. Similarly, *T5-base* and *CodeT5-base* mean the same model, initialized from `t5-base` [144] and `codet5-base` [173] respectively; however, here the whole model is fine-tuned. Models with the *empty* tag are not pre-trained. *RoBERTa-base*, *CodeBERT-base*, *gpt-neo-125M* is a composite model, trained on text+ast data representation, and all of them are initialized using the same pre-trained weights from *RoBERTa-base* [191], *CodeBERT-base* [50], and *GPT-neo-125M*

Table 2.3: *Results on training different models on various data representations.*

Model	Repr.	Pretrained	Dataset	Accuracy	100 %	Epochs
T5-base, LM	text	Yes	Java small	0.9104	0.0000	42
T5-base, LM	text	Yes	Java medium	0.8577	0.0000	25
T5-base, empty	text	No	Java small	0.9371	0.0524	27
T5-base, empty	text	No	Java middle	0.9795	0.0652	31
T5-base	text	Yes	Java small	0.9756	0.1491	9
T5-base	text	Yes	Java middle	0.9864	0.0588	9
CodeT5-base	text	Yes	Java small	0.9684	0.1988	9
CodeT5-base	text	Yes	Java middle	0.9817	0.1187	9
T5-base, empty	cmdseq-token	No	Java small	0.7884	0.2598	12
T5-base, empty	cmdseq-token	No	Java middle	0.7476	0.1488	15
T5-base	cmdseq-token	Yes	Java small	0.8201	0.2946	16
T5-base	cmdseq-token	Yes	Java middle	0.8107	0.1733	18
CodeT5-base	cmdseq-token	Yes	Java small	0.8371	0.3064	12
CodeT5-base	cmdseq-token	Yes	Java middle	0.8051	0.1853	10
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	Java small	0.3862	0.0077	18
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	Java middle	0.2783	0.0000	13
T5-base, empty	text	No	FixJS small	0.8264	0.0030	24
T5-base, empty	text	No	FixJS middle	0.8107	0.0000	18
T5-base	text	Yes	FixJS small	0.9245	0.0496	1
T5-base	text	Yes	FixJS middle	0.9369	0.0262	16
CodeT5-base	text	Yes	FixJS small	0.9078	0.0754	9
CodeT5-base	text	Yes	FixJS middle	0.9207	0.0464	9
T5-base, empty	cmdseq-token	No	FixJS small	0.6917	0.0714	21
T5-base, empty	cmdseq-token	No	FixJS middle	0.6924	0.0149	21
T5-base	cmdseq-token	Yes	FixJS small	0.6502	0.0000	17
T5-base	cmdseq-token	Yes	FixJS middle	0.7265	0.0184	30
CodeT5-base	cmdseq-token	Yes	FixJS small	0.6190	0.0000	13
CodeT5-base	cmdseq-token	Yes	FixJS middle	0.7045	0.0639	16
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	FixJS small	0.2073	0.0010	24
RoBERTa-base, CodeBERT-base, gpt-neo-125M	text + ast	Yes	FixJS middle	0.1504	0.0000	19

[14] respectively.

At first glance, it is obvious that there can be huge differences, based on which dataset we train our models on. Firstly, we can deduce that it is considerably harder to train on the FixJS [31] dataset, compared to the Java dataset [167]. Models evaluated on the Java dataset outperform the evaluation metrics of models on FixJS by a significant margin. This may be due to the fact that FixJS comprises of 9,662 small data samples, and 11,410 samples in the medium dataset. Compared to this, the Java dataset contains 58,350 and 65,455 data samples in the small and medium subsets, respectively. Also, FixJS is processed in a way that every non-unique abstracted sample is filtered out, keeping only unique data. Even samples having the same target with different source representations are filtered out, which can lead to a considerable decrease in data samples, while also reducing redundancy in the database, making it *harder* to train on. To the best of our knowledge, the Java dataset [167] does not apply such filters to the data.

We can see that representing the target as a sequence of command tokens [76] can give a performance boost in some cases. In the Java dataset, this representation outperforms all the other representations by at least 6 %. However, this is not the case with the FixJS dataset, where using the text representation performs a little better on average. The comparison between the representations and the datasets can be seen in Table 2.4.

Although we only conducted one experiment, training only the *Language Model* (LM) head in a pre-trained model (T5-base, LM), it is apparent that training only the LM head is insufficient, as it performed poorly even after training for 42 and 25 epochs on small and medium Java datasets. Interestingly, the *flattened graph* model consisting of RoBERTa [191], CodeBERT-base [173], and GPT-Neo-125M [14], performed poorly compared to other models, although it encodes more information as the simple text model. However, further experimentation would be needed to draw further conclusions. Representing the source code as AST and text, concatenating the results, may have more representative power in theory; alas, decoding it may fail due to insufficient model size. Batch sizes were also needed to be halved during training to fit inside a single GPU.

Seeing Table 2.4, only a quarter of the examined cases benefit from learning on simple text representation, namely the T5-base on small and medium, and the CodeT5-base on small models trained on FixJS. In other cases, we can say that learning command sequence representation is easier for most of the models, suggesting that choosing a good representation for the given dataset is crucial. However, not all datasets benefit the same way from the a chosen representation. It is also apparent that, given the same methods, the results can vary greatly depending on the dataset.

Columns under **Text vs. Cmd.** and *Java vs. FixJS* show the differences between model performance on different data representations, and datasets, respectively. In

Table 2.4: Comparing results on Java and FixJS dataset on text and command sequence token representation.

Model	Text		Command Sequence Tokens		Text vs. Cmd.		Text	Cmd.
	Java 100%	FixJS 100%	Java 100%	FixJS 100%	Java	FixJS	Java vs. FixJS	
T5-base, empty on small	0.0524	0.0030	0.2598	0.0714	0.2074	0.0684	0.0494	0.1884
T5-base, empty on medium	0.0652	0.0000	0.1488	0.0149	0.0836	0.0149	0.0652	0.1339
T5-base on small	0.1491	0.0496	0.2946	0.0000	0.1455	-0.0496	0.0995	0.2946
T5-base on medium	0.0588	0.0262	0.1733	0.0184	0.1145	-0.0078	0.0326	0.1549
CodeT5-base on small	0.1988	0.0754	0.3064	0.0000	0.1076	-0.0754	0.1234	0.3064
CodeT5-base on medium	0.1187	0.0464	0.1853	0.0639	0.0666	0.0175	0.0723	0.1214
Average	0.1072	0.0334	0.2280	0.0281	0.1209	-0.0053	0.0737	0.1999
Min.	0.0524	0.0000	0.1488	0.0000	0.0666	0.0078	0.0326	0.1214
Max.	0.1988	0.0754	0.3064	0.0714	0.2074	0.0754	0.1234	0.3064

the *Text vs. Cmd.* column, results are positive (+) if the cmdseq representation performed better than the text representation and negative (-) otherwise. Consequently, columns found under *Java vs. FixJS* are positive (+) if a model performed better on the Java dataset, and negative (-) otherwise. *Java 100 %* and *FixJS 100 %* mean that the whole predicted sequence matches the target.

2.7.4 Empirical Evaluation and Examples

In the following paragraphs, examples generated during the experiments are presented (Code examples: 2.7.3, 2.7.5, 2.7.7, 2.7.9). In these examples, the first line is the input (the prompt for the model), the second line is the expected output, and the last line is the output of the model. To better readability of source codes, every second Code example (2.7.4, 2.7.6, 2.7.8, 2.7.10) contains the non-abstracted version of the same bugs, and the developer fixes. However, it is important to note that the models have not been fine-tuned on such representations. They are only here to present the nature of the bug to be fixed in a more readable form.

Code 2.7.3: Correctly patched program, using cmdseq-token representation.

```
public void METHOD_1 ( ) { VAR_1 . METHOD_2 ( VAR_2 ) ; VAR_2 = null ; }
// =====
<[/INS]><[/LOC]> ) { if ( ( VAR_2 ) != null<[/INS]><[/LOC]> }
// =====
<[/INS]><[/LOC]> ) { if ( ( VAR_2 ) != null<[/INS]><[/LOC]> }
```

Code 2.7.4: Non-abstracted bug, and developer fix of the example above.

```
public void unregisterNSDService() {
    mNsdManager.unregisterService(networkRegistrationListener);
    networkRegistrationListener = null;
}
// =====
public void unregisterNSDService() {
    if ((networkRegistrationListener) != null) {
        mNsdManager.unregisterService(networkRegistrationListener);
        networkRegistrationListener = null;
    }
}
```

In example 2.7.4 and its code representation 2.7.3, we can see that in the first insertion command, the model can correctly guess that the program needs to be extended with a null check. The second insertion command with the curly bracket is also a part of the previous null-check, closing the if statement. We can see that using command sequence representation, it is possible to learn multiline modification of the program using a few instructions that only concern those parts of the code which need modification, leaving the other parts of the code unchanged.

Code 2.7.5: Incorrectly patched program, using cmdseq-token representation.

```
private void METHOD_1 ( TYPE_1 VAR_1 ) {
    VAR_1. METHOD_2 ( new TYPE_2 ( ) {
        public void METHOD_3 ( TYPE_3 VAR_2 ) { METHOD_4 ( ) ; }
    } ) ;
}
// =====
</[INS]/></[LOC]/> < java.lang.String >
// =====
</[DEL]/></[LOC]/></[LOC]/>
```

Code 2.7.6: Non-abstracted bug, and developer fix of the example above.

```

private void addNotifyListener(javax.swing.JComboBox combo) {
    combo.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent e) {
            notifyChange();
        }
    });
}

// =====
private void addNotifyListener(
    javax.swing.JComboBox<java.lang.String> combo) {
    combo.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent e) {
            notifyChange();
        }
    });
}

```

Example 2.7.5 and 2.7.6 shows that the program should be patched by inserting a generic type parameter. However, according to the model, a deletion is needed. Much of the programs found in Java dataset need only a single delete command to be fixed, so it is safe to assume that the model is biased towards guessing single deletion commands (`</[DEL]></[LOC]></[LOC]>`) if it feels uncertain about the changes to be made.

Code 2.7.7: Correctly patched program, using text representation.

```

public TYPE_1 append ( int value ) {
    VAR_1. METHOD_1 ( VAR_2, value ) ;
}

// =====
public TYPE_1 append ( int value ) {
    VAR_1. METHOD_1 ( VAR_2, value ) ; return this ;
}

// =====
public TYPE_1 append ( int value ) {
    VAR_1. METHOD_1 ( VAR_2, value ) ; return this ;
}

```

Code 2.7.8: Non-abstracted bug, and developer fix of the example above.

```
public cz.lidinsky.tools.ToStringBuilder append(int value) {
    style.appendValue(sb, value);
}
// =====
public cz.lidinsky.tools.ToStringBuilder append(int value) {
    style.appendValue(sb, value);
    return this;
}
```

The patch shown in Code 2.7.7 and 2.7.8 made on the text representation correctly identifies a missing return statement. The method signature shows us that some type of value needs to be returned from the *append* function, however, the model likely encountered very similar examples during training, where a single *return this* statement was missing, as it can guess it correctly with seemingly no context suggesting this prediction.

Code 2.7.9: Incorrectly patched program, using text representation.

```
public int METHOD_1 ( java.lang.Object VAR_1 ) { return 0 ; }
// =====
public int METHOD_1 ( java.lang.Object VAR_1 ) {
    return list. METHOD_1 ( VAR_1 ) ;
}
// =====
public int METHOD_1 ( java.lang.Object VAR_1 ) { return 1 ; }
```

Code 2.7.10: Non-abstracted bug, and developer fix of the example above.

```
public int indexOf(java.lang.Object o) {
    return 0;
}
// =====
public int indexOf(java.lang.Object o) {
    return list.indexOf(o);
}
```

Lastly, we can examine a patch represented by examples 2.7.9 and 2.7.10 that was incorrectly predicted in the text representation. Although the model correctly guessed that the return statement needs to be changed and even the type is identified correctly, this case also shows symptoms of overfitting. Simple patches like this can be found throughout the dataset, where a simple *return 1* statement is good enough. However, the generalization of more complex statements such as this seems to fail.

If we only considered the Java dataset, it is clear that the CodeT5-base is superior on the command-sequence representation compared to others. However, the same

does not seem to be true for the FixJS dataset. As per our previous discussion, this might be a consequence of the learning problem being harder in the latter case (*i.e.*, the nature of the programming language makes it hard to predict). It is also possible that the model could not achieve comparable results due to fewer training samples. In either case, it is worth noting that we used the same model size in each case, which is fair to compare, but not optimal from the data point of view. However, optimizing the size of the model to better fit the data could be another interesting future research direction.

2.8 Discussion and Concluding Remarks

For a conclusion of the presented works, this thesis investigated the role of source code representations in the context of *Deep Learning-based Automated Program Repair*. We conducted experiments using 3 different code encodings, applied in 8 *Deep Learning* model configurations, and evaluated them on two datasets including Java and JavaScript bug fixes. The results suggest that choosing the right representation for a specific learning task can play a crucial role in the resulting model performance and, hence, the quality of repaired programs. We showed that there are cases where edit-based transformation sequences can give a significant boost to performance; however, it should be noted that there were cases where simple text representation outperformed others. In this regard, we concluded that the emphasis on syntactic transformations is not always necessary for optimal model performance, demonstrating that there is no universally optimal representation. Based on the results, we conclude that (1) each *Deep Learning* setting requires its own data representation, thus (2) no representation fits every dataset and every model.

To support continued research in this area, we also introduced a new dataset and a new method to extract code fragments at multiple levels of granularity using Tree-sitter [166]. This method enables researchers to capture code changes at the level of lines, methods, or classes, offering flexibility in how repairs are framed and learned. The dataset introduced includes five different programming languages and is publicly available as a compressed archive at Zenodo. Also, all tools used for the extraction code snippets have been open-sourced on our GitHub repository.

Together, our experiments and published tools contribute to the understanding of how different code encodings can further improve *Deep Learning-based Automated Program Repair*, and pave the way for the building of more accurate and task-tailored repair systems in the future.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

1. Comparison of different code representation strategies with different DL models on different datasets.
2. Novel language-agnostic source code granulate extraction method.
3. Novel dataset including extracted functions for fine-tuning DL models.
4. Providing the dataset to fine-tune GPT-2 pre-trained model, which supports that fine-tuning on code fragments can significantly improve model performance.

This thesis point is based on scientific papers published by the author:

- Dániel Horváth, Viktor Csuvik, Tibor Gyimóthy, and László Vidács. "An extensive study on model architecture and program representation in the domain of learning-based automated program repair." In 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 31-38. IEEE, 2023. [74]
- Dániel Horváth, and László Vidács. "Yet another miner utility unveiling a dataset: Codegrain." In International Conference on Data Technologies and Applications. 2024. [73]
- Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. "Fine-tuning gpt-2 to patch programs, is it worth it?." In International Conference on Computational Science and Its Applications, pp. 79-91. Cham: Springer International Publishing, 2022. [93]

Chapter 3

Green Energy Through ML: Optimizing CO₂ Electrolyzer Cell Performance

3.1 Introduction

While the previous chapters have focused on applying *Machine Learning* techniques within the domain of *Software Engineering* and *Automated Program Repair*, this chapter demonstrates the broader applicability and versatility of these methodologies across entirely different scientific domains. The transition from optimizing source code representations and repair algorithms to optimizing electrochemical processes for CO₂ reduction illustrates a fundamental principle: *Machine Learning* techniques, particularly *Artificial Neural Network* (ANN)s, can effectively model and optimize complex systems regardless of their underlying domain-specific characteristics. Just as we explored how different code representations affect the performance of *Deep Learning* models in program repair, here we investigate how ML approaches can learn and predict the intricate relationships between operational parameters in CO₂ electrolyzer systems. This shift in domain shows the universal nature of *Machine Learning* as a powerful tool for understanding and optimizing complex, multi-parameter systems, whether they involve abstract software artifacts or physical electrochemical processes.

Although the number of reports on the electrochemical reduction of carbon dioxide is increasing at an ever-accelerating rate, achieving long-term stable, selective, and energy-efficient operation is still challenging. This can be attributed mainly to the short duration of laboratory-scale measurements and the complexity of the cell operation parameters. Here, we introduce a high-throughput cell operation testing methodology, including data evaluation and optimization by machine learning algorithms. The experimental data was collected using an autonomously operating test

station, allowing the collection of enough data to develop an *Artificial Neural Network* (ANN) model. We show that when the model is trained on a fraction of a large dataset, predictions for the operation of the same cell under different conditions are very precise. When a calibrated model for a given cell was used to evaluate results gathered with a newly assembled, but otherwise identical, electrolyzer cell under conditions that were not part of model training combinations, all important process metrics (except the overestimated H₂ amount) were predicted with relatively high precision.

The opportunity of simultaneously reducing the emission of a harmful greenhouse gas and forming valuable chemicals attracted many researchers to focus on the *electrochemical reduction of CO₂* (CO₂RR) [12, 46, 160]. This resulted in rapid advancement of the field, considering all important process metrics (reaction rate, selectivity, stability, energy efficiency) [35, 100]. This progress was facilitated by the improvement of the systems, cell architectures, catalysts, and other structural elements used [8, 10, 152]. However, achieving the optimal combination of these *Key Performance Indicators* (KPIs) is a grand challenge to be solved, so that CO₂RR can mature into an industrially applied technology at scale [12, 62, 145].

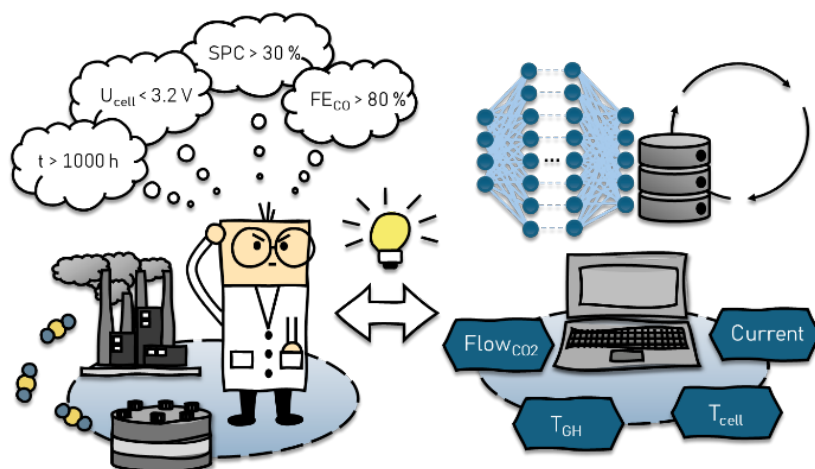


Figure 3.1: Aiding CO₂ reduction processes via Machine Learning algorithms.

Despite continuous collection of experimental data (at a still accelerating rate), modeling cell operation is still in its infancy with scarce reports [2, 43, 103]. In addition, there is still no general model that would fully describe the operation of CO₂RR cells and would aid in optimizing their operation in a continuous parameter space. This is not surprising considering the large number of operational parameters [115], including cell temperature, gas humidification, gas flow rate, anolyte composition, and cell voltage/current [4, 142, 161, 183, 196]. In addition, there is no standard test protocol, which also means that different research groups measure data under very different circumstances, which is hard to compare. Aging is another factor to be

considered: if a cell was operated under conditions that cause (ir)reversible changes, the subsequently recorded results might be misleading. Typical examples are electrode flooding and salt precipitation, both of which alter cell chemistry [72, 124]. Furthermore, a transient behavior might be captured over very short measurements; hence, the conclusions derived can be less relevant for continuously operating electrolyzer cells.

3.2 Related Work

Machine Learning (ML) methods, especially *Artificial Neural Networks* (ANNs), are increasingly used to model and optimize electrochemical energy devices. In CO₂ electrolyzers and *electrochemical reduction of CO₂* (CO₂RR) systems, data-driven models have been developed to capture complex reaction and transport phenomena. Richard *et al.* (2023) studies a smart manufacturing approach for CO₂ electrolysis, arguing that large, high-quality datasets and AI modeling are needed to decouple the kinetics and the effects of mass/heat transport in the scaling of systems to a level where they could be industrialized [146]. They demonstrated that *Feedforward Neural Networks* (FNNs) can accurately capture nonlinear relationships in CO₂ reactor data when in a steady state (*i.e.*, no changes in reactor operation). Recurrent networks (RNNs/LSTMs) have been applied to model the time-dependent dynamics of a rotating cylinder CO₂ electrolyzer: an LSTM-based model successfully learned the reactor's transient behavior from sparse gas-product measurements and was used in a multi-input-multi-output control scheme. These studies illustrate that ANN-based surrogate models can predict CO₂RR reactor outputs (*e.g.*, product rates) from inputs (voltage, flow, catalysts) with high fidelity, enabling downstream optimization.

ANN models have similarly been used in the production of H₂ in water electrolysis research. Early work by Chávez-Ramírez *et al.* constructed a *Multi-Layer Perceptron* (MLP) to model a three-cell *Proton Exchange Membrane* (PEM) (*a.k.a.* Polymer Electrolyte Membrane) electrolyzer stack. Using current and temperature as input, their ANN predicted the voltage of each cell with a good correlation rate ($R > 0.97$ in the train, validation and test splits) with less than 2 % error [21]. Recently, studies are also using ML to optimize even the design parameters of PEM electrolyzers. Mohamed *et al.* trained polynomial and logistic regression models to predict 11 design parameters (anode type, cathode type, anode catalyst, cell temperature, number of cells, *etc.*) from 4 input features such as the hydrogen production rate (mL/min), cathode area (mm²), anode area (mm²), and type of cell design (*e.g.*, single or bipolar) [120]. Their ML-based design yielded model predictions that closely matched the experimental electrolyzer performance, demonstrating the validity of such approaches. Ozdemir *et al.* (2024) evaluated multiple ML techniques (MLP, SVM, Random Forest) for a PEM electrolyzer dataset and found that ANN gave the best

current density predictions (MAE of 0.03819 and 0.04036 for train and test sets respectively) [134].

Neural networks have also been applied extensively to fuel cells. Abbade *et al.* (2025) developed a *Deep Neural Network* (DNN) to predict the voltage measured on a NEXA 1200W [127] PEM fuel cell stack. Training on experimental operating data, their model achieved an MSE and MAPE score of 0.0429 and 1.05 % respectively. They also achieved a low RMSE of 0.2071, and a high *Coefficient of Determination* (R^2) rate of 0.9071 on the test data [1]. For introducing non-linearity to their systems, they use the *Rectified Linear Unit* (ReLU) activation function. Their design successfully captures both the linear and the non-linear effects of the input parameters on the fuel cell voltage without explicitly using any electrochemical equations. Another example is the study of Napoli *et al.* emphasizing that the establishment of mathematical models applicable on PEMFCs is often not a trivial task. To mitigate this issue, they also employ ML-based methods, namely they experiment with classical NNs, and compare them with different stacking strategies, showing that both methods are capable of predicting the output variables under various conditions, provided as input features to the learning algorithm [126]. *Machine Learning*, and especially ANN-based architectures are being used in many studies ranging from optimization of cell control to optimization of cell design. These studies show that data-driven ANNs can replace (in case of lack of domain knowledge) or even complement physics-informed or physics-based fuel cell models by learning these complex and often hidden relationships from experimental data.

In summary, the literature demonstrates the effectiveness of ML and ANNs in electrochemical system modeling and optimization. Across CO₂RR, water electrolysis, and fuel cell domains, ANN-based surrogate models can predict key outputs (voltage, current density, hydrogen production, *etc.*) from device inputs with high accuracy [1, 21]. They have been integrated into design and control workflows to reduce experimental costs and explore high-dimensional parameter spaces [120]. These works also motivated research of AEM separated electrolyzer cells through the use of various ML techniques. The next sections present this development process of the AI-guided optimization framework for CO₂ electrolyzer cells that use data-driven approaches to model and optimize these complex systems.

3.3 Method

Advanced computational methods offer a solution to circumvent such difficulties, by evaluating a large amount of experimental data together, as a whole. *Machine Learning* (ML) is getting widely applied for predicting and optimizing the operation of water electrolyzer cells and fuel-cells, which could serve as a guide for the field of CO₂ electrolysis [82]. Such methods are already in use in some domains; such as

identifying catalysts and additives for CO₂RR [64, 75].

Formulating exact theoretical equations for such a complex system as CO₂RR is often challenging, but this challenge can be tackled using ML [180]. *Artificial Neural Network* (ANN) is a particularly useful ML tool to uncover complex and non-linear relationships between variables in vast datasets. These methods rely on models that are built upon a large amount of experimental data and are validated under previously unseen, randomly chosen conditions. In addition, such data analysis can be used (i) to diagnose cell status, (ii) to ensure dynamic continuous operation, with parameters adjusted to the cell properties, and (iii) to forecast (and possibly avoid) cell failure. These use cases have been investigated with respect to hydrogen production, but have yet to be explored for CO₂RR [52].

Here we report on the collection of a large amount of reliable experimental data using our custom-built and autonomously operated electrolyzer test station. The task-tailored software allowed for filtering data that was collected during parameter settling, hence providing reproducibility. The filtering process at the test station is a relatively straightforward process, as it only considers a small range of variance that is allowed during the measurement process. Data points that fall outside these predefined intervals (*e.g.*, cell temperature deviates by more than 2 °C from the target setting), then that particular data point will not be considered for future learning processes. In this context, a data point (or measurement point) means a single row of collected values recorded at a single point in time (*e.g.*, current, voltage, product composition and flow rate).

The large amount of data was processed using ML algorithms to develop a model for continuous description of cell operation in the entire parameter space, instead of scattered experimental points. Furthermore, the ANN-based evaluation made it possible to identify the parameters limits too narrow for optimal cell operation (*e.g.*, too high a minimum humidification temperature of the gas), as the model points beyond the parameter space that we originally defined based on our previous experience and state of the art results. ANN-based optimization of cell operation does not require exact knowledge of all physical/chemical phenomena that occur in the electrolyzer cell, nor does it require the formulation of exact equations. Accounting for cell aging is also possible via ANN-based methods, even without identifying the cause of performance fading. Implementing ANN-based operation in process control systems for continuous long-term operation of the CO₂ electrolyzer is therefore one of the most realistic routes for industrialization of CO₂RR. We note that this is not in contrast to the semiempirical physics-based description of cell operation. [2, 43, 103] On the contrary, the parallel development of these methods and their final merging can provide a full description.

Our workflow consists of three key steps (Figure 3.2). First, raw data collected from the electrochemical measurements were pre-processed to remove outliers and

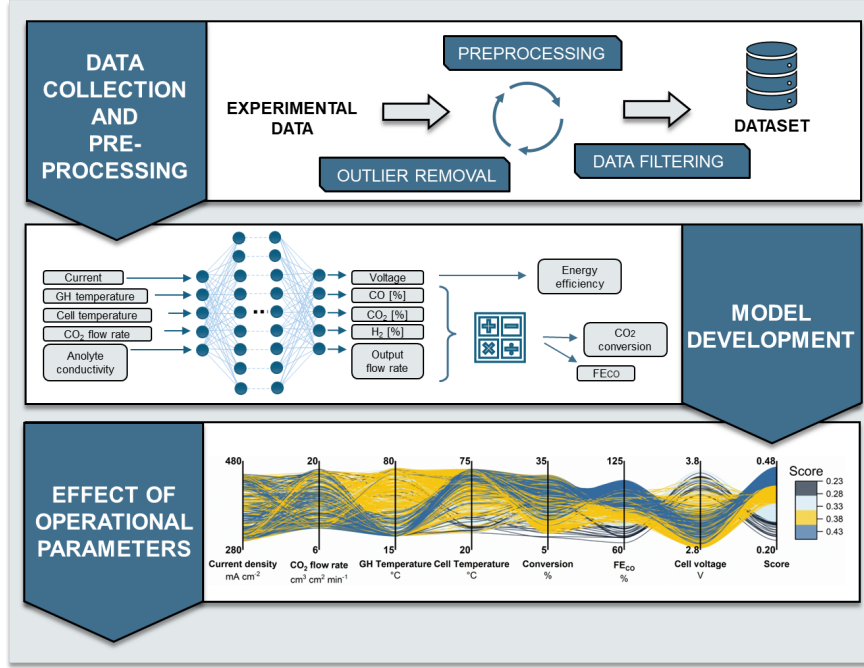


Figure 3.2: Searching the optimal values for the operation of the CO₂ electrolyzer cell.

faulty samples (e.g., parameter combinations that did not reach steady state conditions within the specified time limit), since these can deteriorate model prediction performance. When applying outlier detection, we only consider our 5 target variables (cell voltage, CO₂ concentration, CO concentration, H₂ concentration, and cathode gas flow rate). Outlier detection is performed by calculating the Z-scores of every individual output value, defined by the following equation.

$$Z = \frac{x - \mu}{\sigma}, \quad (3.1)$$

where x is the measured value of an individual data point, μ is the mean and σ is the standard deviation value of the distribution from which the specific value x came from, which is defined by the entire measurement sequence of a single parameter setting. We defined a Z-score tolerance of 3. If even one of the measured values falls outside this interval, it will be excluded. The data was split into train, validation, and test sets. In the next step, an ANN model was trained to predict cell voltage, gas composition (V/V % of CO, CO₂, and H₂), and output flow rate (flow out) using the input parameters: current, gas humidifier (GH) temperature (determining the humidity level of the CO₂ inlet feed), anolyte's electric conductivity (EC), cell temperature, and CO₂ inlet flow rate. In the last step, the trained ANN was combined with a genetic algorithm to explore an extended parameter space and find the best combination of parameters for the optimal operation of the electrochemical cell.

This last step was only achieved in part in this study. The reason behind this is the experienced cell-to-cell variation, as in Table 3.1.

Table 3.1: *Parameter settings used for the electrolysis experiments.*

Current density $\left[\frac{\text{mA}}{\text{cm}^2}\right]$	300	350	400	
Cell Temperature $[\text{°C}]$	60	65	70	
GH Temperature $[\text{°C}]$	55	60	65	
CO ₂ feed flow rate $\left[\frac{\text{cm}^3}{\text{min}\cdot\text{cm}^2}\right]$	9.375	10.375	11.500	12.500

The different parameter settings seen in Table 3.1 show the values that were pre-defined before starting the measurement. Each of the values were visited separately while fixing the other values, such as the case with grid search, resulting in 108 different parameter combinations. This grid is also demonstrated by the left side of Figure 3.3.

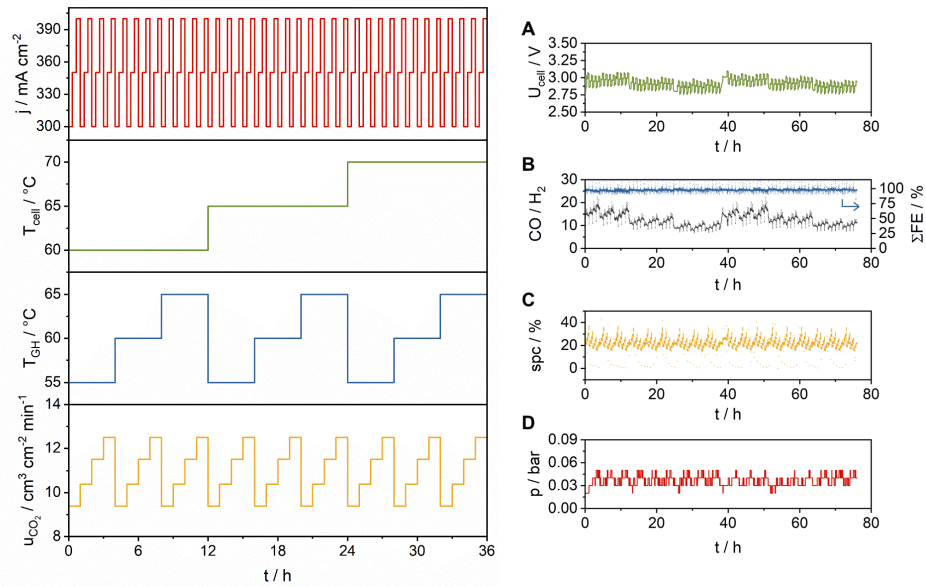


Figure 3.3: *Parameter setting combinations (left) and the measured results (right) for the electrolysis experiments.*

On the right side of Figure 3.3 we can observe the measurement data that was accumulated for 1200 seconds with 5 second intervals per parameter setting. In

Table 3.2: Correspondence between control- and model input features.

Control Variable	Input Feature
Current Limit Setpoint [A]	Current [A]
Cell Temp Setpoint [deg C]	Cathode Temp [deg C]
GH Temp Setpoint [deg C]	GH Temp [deg C]
MFC Setpoint [sccm]	Flow [sccm]

the data splitting step, the individual data points are already treated as sequences (groups) and are further split into train, validation, and test sets accordingly. This data handling strategy prevents potential data leaks and ensures a fair evaluation of the models. All data points recorded for a single parameter setting were evaluated together, adding up to ~ 240 readings per parameter setting (which might be decreased during data filtering in case faulty readings are observed). The measurements were repeated twice in a row. In the figure, we can see the (A) cell voltage (B) selectivity (C) *Single Pass Conversion* (SPC) and (D) the pressure measured upstream of the electrolyzer cell.

The link between the parameter settings shown in Table 3.1 and represented by the left side of Figure 3.3, can be observed in Table 3.2. The control variables are settings that can be set during the measurement phase. The system validates these settings by measuring them in real time and also reporting the true values of these values. Our models use the column of input features as their training data, with *every* sample as a different measurement point.

Table 3.2 represents the link between controllable settings and the values measured by the system in real time. The control settings and their measured values can experience certain deviations. This is an important aspect, and also a limitation of the current training and measurement procedure. Although there is a limitation on how far the control variables can be from their corresponding measured values, it is also true that there is a range in which these values are accepted (e.g., temperatures are accepted within a ± 2 °C interval). As such, we opted for training the model on the real measurement data instead of training the model to predict the mean of the outputs produced by certain parameter settings. The result of this design is more training data; however, it is also true that in this sense, this model will *expect* that the settings it trained on are reproducible, *i.e.*, and the parameter settings are precisely controllable.

Model training was carried out by systematically varying four process parameters (current density, cell temperature, GH temperature and CO₂ inlet flow rate), performing experiments in all possible combinations (see Table 3.1), adding up to 108

data points in a single measurement sequence (Figure 3.3). In the model development phase, a fully connected neural network was used to model the relationship between input and target variables. The model consists of three key parts: the input layer, the hidden layers, and the output layer. The input layer has a neuron number equal to the number of input variables, while the output layer maps the weights of the hidden layers to the target variables. In this case, both the number of input and target variables was 5. Between the input- and output layers there are 3 hidden layers with 132 neurons each. To handle non-linear relationships in the data, the network implements GELU [70] activation function between each layer [52]. The number of hidden layers and the number of neurons were selected based on multiple hyper-parameter optimization runs. The parameters with the best results have been selected to serve as model hyperparameters for the experiments. To train the model, we used *Mean Squared Error* (MSE) as the loss function and chose AdamW as the optimization algorithm [105].

In the data preprocessing step, the test split was carefully separated from the rest of the data, avoiding any overlap between the training and test data sets. This means that parameter combinations used for testing are in no way part of the training or the validation sets, ensuring that the model testing gives a fair picture of the model's performance. The test data set was 20 % of the total of unique parameter combinations (*i.e.*, 21 of the 108 different parameter combinations; see Table 3.1 and Figure 3.3). The remaining 80 % of the parameter combinations were further split into training-, (80 %) and validation datasets (20 %).

3.4 Model Selection

Selecting a baseline model for our case study was not a trivial task, as the number of ML algorithms is continuously increasing, with newer and more sophisticated strategies. It is outside the scope of this study to list all possible algorithms to choose from; however, we did conduct a series of experiments to get insight into the intricacies of different models. During these experiments, our goal was to select the predictive model that would offer the best trade-off between accuracy and generalizability across different cell assemblies, also keeping in mind the ease of use and the possibility to further upgrade the selected model.

Several model types were considered ranging from simpler linear regression models to *Random Forest Regressor* (RFR), XGBoost and *Artificial Neural Network* (ANN) models. All models performed surprisingly well on the test set (see Table 3.3); however, they mostly failed on the external set (except ANN), especially in terms of the average R^2 value. In this sense, RFR is the only model (apart from ANN) that is capable of producing a positive value on the average R^2 metric (see Table 3.3). This implies that most of the models fail to generalize well on different cell assemblies.

Table 3.3: Artificial Neural Network, Random Forest Regressor, Linear Regressor, and XGBoost Regressor model predictions on the test, and external test set.

	Test set					External test set				
	R ²	MSE	RMSE	MAE	MAPE	R ²	MSE	RMSE	MAE	MAPE
Artificial Neural Network										
Voltage [V]	0.9892	9.6347E-05	9.8157E-03	7.8167E-03	2.5955E-03	0.8184	5.0718E-03	7.1216E-02	5.6806E-02	1.8082E-02
CO [%]	0.9960	1.8197E-01	4.2658E-01	3.1806E-01	1.0431E-02	0.9119	6.4593E+00	2.5415E+00	1.6839E+00	6.7276E-02
CO ₂ [%]	0.9963	1.6060E-01	4.0075E-01	3.0577E-01	5.2516E-03	0.7823	1.1114E+01	3.3338E+00	2.4628E+00	3.8523E-02
H ₂ [%]	0.9843	1.8863E-02	1.3734E-01	9.9188E-02	4.3357E-02	0.0455	1.0530E+00	1.0262E+00	9.7229E-01	1.5113E+14
Flow out [sccm]	0.9943	3.2587E+00	1.8052E+00	1.3539E+00	1.1354E-02	0.9614	7.4411E+01	8.6262E+00	5.0597E+00	3.8184E-02
FE(CO) [%]	0.5356	1.8961E-04	1.3770E-02	9.9178E-03	1.1313E-02	-3.5247	2.2081E-03	4.6991E-02	3.9980E-02	4.4964E-02
SPC [%]	0.9904	1.1653E-05	3.4136E-03	2.4724E-03	1.1313E-02	0.9593	1.0702E-04	1.0345E-02	8.9142E-03	4.4964E-02
Average	0.9266	5.1720E-01	3.9955E-01	2.9959E-01	1.3659E-02	0.1363	1.3292E+01	2.2366E+00	1.4692E+00	4.1999E-02
Random Forest Regressor										
Voltage [V]	0.9726	2.4340E-04	1.5601E-02	1.1846E-02	3.8987E-03	0.5076	1.3756E-02	1.1729E-01	8.3694E-02	2.6422E-02
CO [%]	0.9949	2.3293E-01	4.8263E-01	3.6682E-01	1.2029E-02	0.8407	1.1679E+01	3.4174E+00	2.5349E+00	1.1545E-01
CO ₂ [%]	0.9969	1.3354E-01	3.6543E-01	2.7229E-01	4.6328E-03	0.7234	1.4122E+01	3.7579E+00	2.7860E+00	4.3569E-02
H ₂ [%]	0.9442	6.7066E-02	2.5897E-01	1.8267E-01	7.5441E-02	-0.4426	1.5914E+00	1.2615E+00	1.1980E+00	4.5967E+14
Flow out [sccm]	0.9944	3.2141E+00	1.7928E+00	1.2682E+00	1.0606E-02	0.5577	8.5238E+02	2.9195E+01	1.5493E+01	8.3024E-02
FE(CO) [%]	0.3100	2.8175E-04	1.6786E-02	1.2812E-02	1.4593E-02	-2.7019	1.8066E-03	4.2504E-02	3.3550E-02	3.7752E-02
SPC [%]	0.9847	1.8647E-05	4.3182E-03	3.2579E-03	1.4593E-02	0.9609	1.0300E-04	1.0149E-02	7.8817E-03	3.7752E-02
Average	0.8854	5.2117E-01	4.1950E-01	3.0255E-01	1.9399E-02	0.0637	4.5679E+00	5.4003E+00	3.1624E+00	5.7329E-02
Linear Regressor on 2 nd Order Polynomial Features										
Voltage [V]	0.9791	1.8557E-04	1.3622E-02	1.0138E-02	3.3645E-03	0.7776	6.2122E-03	7.8818E-02	6.4854E-02	2.0647E-02
CO [%]	0.9959	1.8858E-01	4.3425E-01	3.1341E-01	1.0103E-02	0.7232	2.0288E+01	4.5042E+00	2.2266E+00	1.2630E-01
CO ₂ [%]	0.9954	2.0108E-01	4.4842E-01	3.5982E-01	6.0944E-03	-0.1793	6.0203E+01	7.7591E+00	4.0991E+00	6.0248E-02
H ₂ [%]	0.9695	3.6681E-02	1.9152E-01	1.3175E-01	5.4210E-02	-1.7186	2.9992E+00	1.7318E+00	1.3298E+00	1.6309E+15
Flow out [sccm]	0.9939	3.4739E+00	1.8638E+00	1.4171E+00	1.1833E-02	0.9785	4.1364E+01	6.4315E+00	4.3075E+00	2.8344E-02
FE(CO) [%]	0.6046	1.6147E-04	1.2707E-02	9.3193E-03	1.0615E-02	-288.1289	1.4110E-01	3.7563E-01	1.3007E-01	1.4735E-01
SPC [%]	0.9911	1.0840E-05	3.2923E-03	2.3700E-03	1.0615E-02	0.1647	2.1978E-03	4.6880E-02	1.9242E-02	1.4735E-01
Average	0.9328	5.5722E-01	4.2395E-01	3.2055E-01	1.5262E-02	-41.0547	1.7858E+01	2.9897E+00	1.7396E+00	8.8375E-02
XGBoost Regressor										
Voltage [V]	0.9826	1.5467E-04	1.2437E-02	9.7584E-03	3.2270E-03	0.4073	1.6556E-02	1.2867E-01	1.0201E-01	3.2380E-02
CO [%]	0.9932	3.1466E-01	5.6094E-01	3.5828E-01	1.1500E-02	0.8937	7.7882E+00	2.7907E+00	2.3703E+00	1.0600E-01
CO ₂ [%]	0.9950	2.1749E-01	4.6636E-01	2.7687E-01	4.6876E-03	0.8942	5.4002E+00	2.3238E+00	1.7454E+00	2.6027E-02
H ₂ [%]	0.9590	4.9297E-02	2.2203E-01	1.5148E-01	6.3985E-02	-0.0883	1.2006E+00	1.0957E+00	1.0581E+00	4.7134E+14
Flow out [sccm]	0.9927	4.1835E+00	2.0454E+00	1.2760E+00	1.0954E-02	0.6012	7.6864E+02	2.7724E+01	1.4308E+01	7.5049E-02
FE(CO) [%]	0.4773	2.1345E-04	1.4610E-02	1.1122E-02	1.2654E-02	-6.7081	3.7617E-03	6.1333E-02	5.2848E-02	5.8807E-02
SPC [%]	0.9883	1.4240E-05	3.7736E-03	2.8300E-03	1.2654E-02	0.9107	2.3506E-04	1.5332E-02	1.2952E-02	5.8807E-02
Average	0.9126	6.8077E-01	4.7507E-01	2.9805E-01	1.7095E-02	-0.4413	2.4016E+00	4.8771E+00	2.8071E+00	5.9511E-02

The average values are tinted with hues that correspond to the color of these results shown in Figure 3.4. The blue color corresponds to the results achieved by the ANN model, the orange color matches the results of the Random Forest Regressor, the green color shows the results for the Linear Regression model, while the red color corresponds to the XGBoost model.

Figure 3.4 provides a visual comparison of the average performance of the ANN and the alternative models on the test-, and external test set. Note that the hue of the last rows of Table 3.3 matches the color of the average metric value presented in the figure. Another notable piece of information in the table is the gray color of cells for the predicted H_2 [%] value. These values are outliers that are mainly the result of the sensitivity limitations of the gas analyzer and data pre-processing. In cases where the amount of H_2 in the output gas decreased below the sensitivity threshold of the gas analyzer, it returned unrealistically high values. These faulty values are filtered out by replacing them with zero. As a result of this data preprocessing step, the MAPE metric increases unrealistically, and as such, we left out these values for the average calculations. Even with this allowance in mind, MAPE still performed significantly worse (2-4 times) on the external test set for all algorithms used.

Figure 3.4 illustrates the overall performance of the model averaged over all separate output values on the test set and the external test set. The y-axis shows the different metrics used to evaluate the fitness of the model, while the x-axis shows the values of these metrics. The different colors help distinguish between the various models and their performance. Note that, for practical purposes, we used a log-scale on the x-axis, as some values could show notable differences. The values of the metrics are also shown to aid in the comparison of performances. On the external test set, the R^2 values for the Linear Regression and XGBoost models are negative, implying poor correlation. The negative values are replaced by 0 and not shown in the plot, due to the logarithmic scaling of the x-axis.

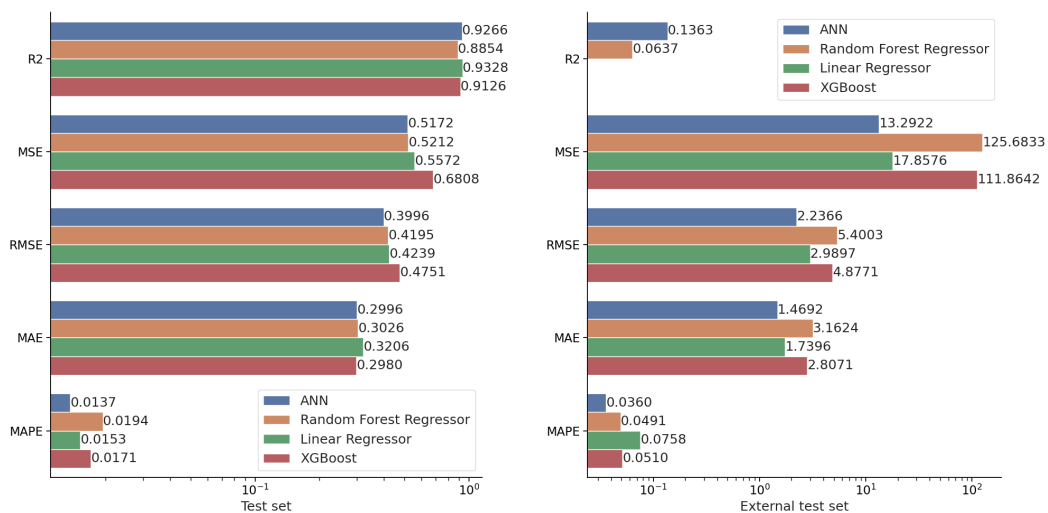


Figure 3.4: Model performance mean values on the test set.

Ultimately, a relatively simple *Artificial Neural Network* architecture emerged as the most effective model for our experiments. The chosen model achieved the best

test results among other ML-based methods and provided a strong balance between flexibility and generalization. In addition, this model has been proven to be the most robust in terms of predictive capacity across different cell assemblies.

In conclusion, while a variety of modeling techniques were evaluated, the ANN model provided the best overall performance and generalizability across different cells. The models seen in Table 3.3 are documented for reference and comparison purposes, but were ultimately not selected for the final optimization processes.

3.5 Results

Several experiments were conducted to evaluate the performance of the model. To see how well the model performs given a cell assembly, we split the train data in a way that separates data corresponding to test parameter settings, and train-val parameter settings. As discussed in Section 3.4, an ANN-based architecture emerged as the most efficient model among others. This model is evaluated against the test set (same cell) and also the external test set (same cell assembly, different cell) to see how well this model generalizes across separate but identical electrolyzer cell assemblies.

3.5.1 Evaluation on Test Set

Figure 3.5 shows the evaluation of the ANN model on the test set. It can be deduced that the model performs well, based on the shown R^2 values, which basically tells us how well a statistical model predicts the dependent variables based on the given independent variables. Normally, the closer the R^2 value is to 1, the better the prediction prowess of the model.

Figure 3.5 illustrates the validity of the developed model on the test set. The model was trained through the measurement sequence built from varying experimental parameters according to Table 3.1 and Figure 3.3. The test set was 20 % of the total number of parameter combinations, carefully separated from the rest without any overlap.

Quantifying the accuracy of model prediction, the trained model predicts target features with high precision on the test set, even though the model never saw any of these parameter combinations. Note that the test set was chosen from experiments run on the same electrolyzer cell. If one examines the voltage values predicted by the model, they are closely matched with the actual values of the experiment. This is supported by a low value of *Mean Absolute Error* (MAE) and a high *Coefficient of Determination* (R^2). *Mean Absolute Percentage Error* (MAPE) is also of low value, showing that the predicted values do not deviate significantly from the true target variable (see all parameters in Table 3.3). One can reason similarly for the predicted

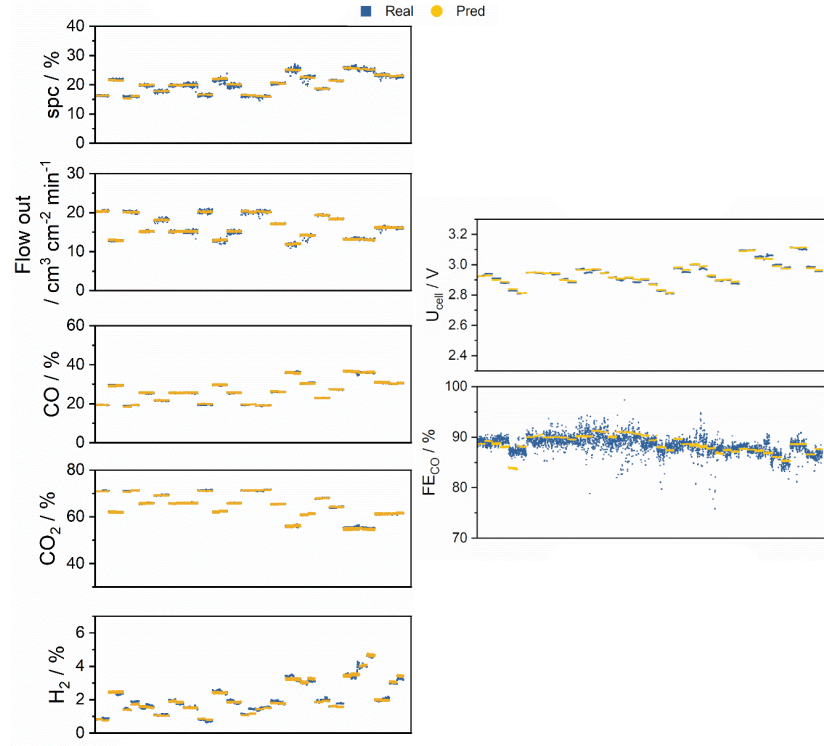


Figure 3.5: Model prediction results on the test set.

CO, CO₂, H₂ concentrations, and the flow rate of the product stream (flow out), as these predictions also indicate a high correlation with the target variables. However, $FE(CO)$ behaves differently. Note that this value is calculated from previous values predicted by the model, so it is affected by prediction errors. Despite this, MAE and MAPE scores show that $FE(CO)$ could be closely estimated using this model. The R^2 value is not so encouraging; however, it could be reasoned that $FE(CO)$ is of high variability with many outlier-like artifacts (Figure 3.5 (right)). For this reason, the R^2 value is bound to be lower for this target variable.

As shown in Figure 3.5, the predicted values of the model match the measured results very closely (R^2 values above 0.98 were reached for all the targets with the exception of $FE(CO)$ [%]). Table 3.3 also shows these trends. This model (trained on a dataset from repeated measurements of the 108 on the same cell twice in a row) was used by the NSGA-II algorithm [33] to identify combinations of favorable parameters in an almost continuous parameter space. The optimization budget was defined in terms of the maximum number of jobs, that is, the number of parameter combinations that the algorithm is allowed to test before termination. We set this budget at 10,000. The parallel coordinates plot shown in Figure 3.2, where only results with 1,000 test runs are shown for better visibility, represent these different scenarios, where the cell operation is predicted by the model for each combination of

connected input parameters. The last 3 axes before the score column represent the predicted values of the different target variables, while the score metric itself signifies the overall goodness of the given parameter combination. The score metric is not a target variable; it is simply calculated by taking the mean value of the individual scores of the 3 target variables.

In this way, 8 more parameter combinations were chosen from the parameter optimization experiment to investigate the validity of the model and whether the parameter space is worth extending in any direction. We refer to these parameter combinations as the external test set. 7 out of the 8 combinations contained at least one parameter that had a value outside of the originally defined parameter space. After a short (1h long) conditioning step under conditions that allowed stable cell operation, [145] the measurement sequence generated from this combination of parameters was performed with a newly assembled (but otherwise identical) CO₂ electrolyzer cell, twice in a row (without stops, see Figure 3.7). Note that although only 16 parameter combinations were defined (8 settings, repeated twice in a row), and the results were only collected for 20 minutes at each setting, the duration of this whole experiment was almost 24 hours. This is an important attribute of our experimental setup and control software: All important process parameters are continuously monitored, and data are only collected when the target values are reached and stabilized. This is very important, especially in the case of changing the temperature of the electrolyzer cell and/or the GH, devices of rather large heat capacitance. In other cases, for example, when only the applied current density is changed, reaching the desired setting is almost instantaneous.

3.5.2 Parameter Optimization

In the parameter optimization step, the goal was to test the validity of the model predictions in an extended parameter space and find the optimal settings for the electrochemical cell. We used Optuna's Python package to guide this process [3]. The maximum number of trials (the overall budget) was set to 10,000. The optimization metrics were set to be $FE(CO)$ [%] and *Single Pass Conversion* [%], while also optimizing for Cell Voltage [V]. The first two values were set for maximization, while the Cell Voltage was minimized. When optimizing for multiple targets, Optuna defaults to using the so-called *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) [33] with its default parameter settings of population size of 50, crossover probability of 90 % and mutation probability of 1 divided by the length of trial parameters to be optimized (3 in our case).

In the optimization process, the trained model was employed to predict target variables for thousands of previously —highly likely— unseen input variable combinations. Unlike single-objective optimization, multi-objective optimization aims to

find solutions that have the best trade-offs between different objectives. The set of these solutions is also known as the Pareto front, where it can be said that improving one objective leads to degradation of others. In other words, neither of the solutions can be said to be better than the other. The main steps of the algorithm is as follows: (1) initialization of the first population, (2) fitness evaluation, (3) non-dominated sorting, (4) calculation of crowding distance, (5) selection of parents, (6) application of genetic operators (crossover, mutation), (7) creation of the next generation (using elitism), (8) process termination if the criteria are met. To better understand the procedure, these steps can be seen in Figure 3.6.

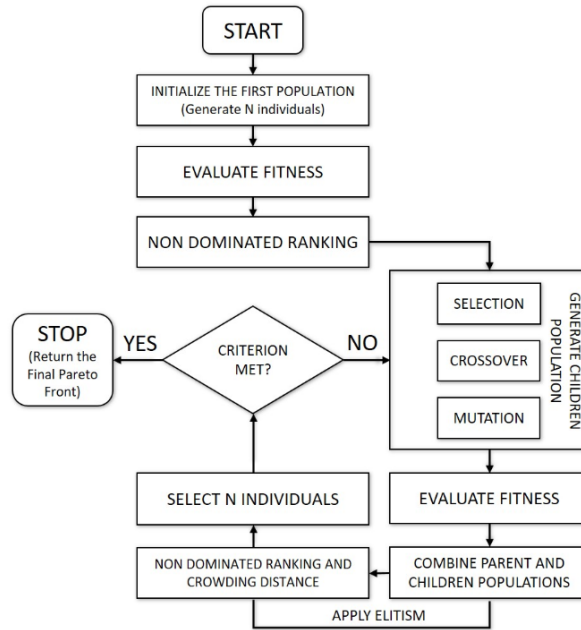


Figure 3.6: Flow chart of the NSGA-II algorithm.

When the optimization process was over and all 10,000 trials were run successfully, we selected 8 different parameter sets provided by the selection procedure. When selecting these parameters, a significant selection criterion was to include 4 combinations of parameters from the lower-end optimization trials and 4 from the higher-end trials, while also considering that the selected combinations should be of different settings, if possible. This meant that we would have 8 distinct parameter combinations, which are considerably different from each other, for the external test set.

3.5.3 Evaluation on External Set

For the external test set, data collection was performed on another cell assembly, but otherwise identical to the one we used for the development of the model. The pa-

parameter setting with the highest gas flow rate (150 sccm) was invalidated in the first measurement, as this parameter combination never settled in a stable state, rendering it unreliable. The second trial was kept, as it may contain useful information on cell behavior outside the typical parameter settings. Note that, for this reason, only 15 parameter settings remained after all the data preprocessing steps, keeping two repetitions for 7 of the parameter combinations and one single measurement for 1 parameter combination.

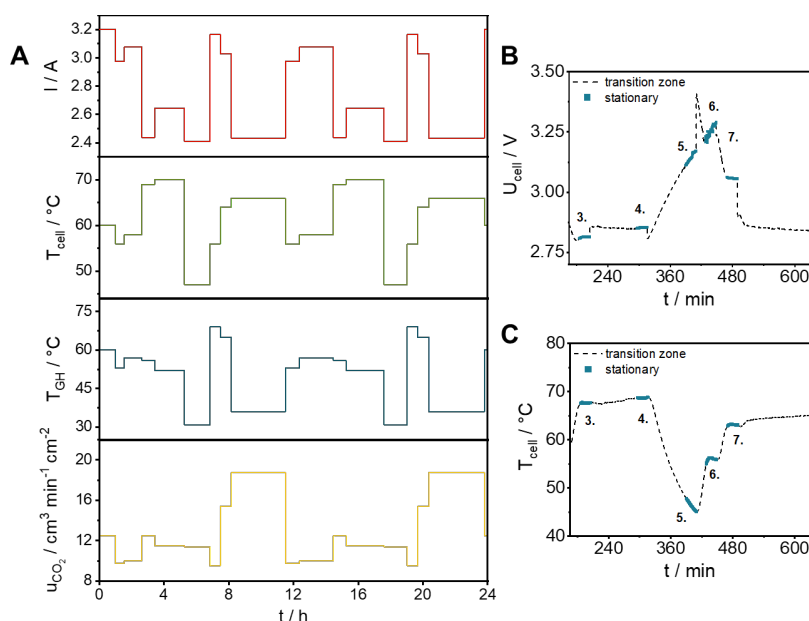


Figure 3.7: Set parameters of the external test.

In Figure 3.7 one can observe the set parameters of the measurement conducted to be our so-called external test set. (A) shows the experimental measurement sequence, which was used for model validation (external test set). (B) is the typical cell voltage and (C) is the cell temperature curves recorded during such measurement, indicating short and long transition times between different experimental settings.

Many general notions of the CO₂RR community were confirmed when running these parameter combinations. The cell voltage scales with the applied current and is inversely proportional to the cell temperature. The electrolyzer cell voltage varied between 2.8 and 3.3 V, with a $FE(CO)$ variation of 90-100 %, while the *Single Pass Conversion* (SPC) values spanned between 15-35 %. At lower GH temperatures, CO formation is preferred, resulting in very high selectivity (expressed by the CO/H₂ ratio). These trends are precisely captured by our sensitivity analysis (refer to Section 3.5.6, and see Figure 3.14), performed on the dataset used for model training.

The increase in cathode pressure occurred at low GH and cell temperatures, which is attributed to precipitate formation and its inefficient removal from the cathode

gas diffusion electrode [45]. This further causes a cell voltage increase, which is attributed to the decreasing electrochemically accessible surface area. This notion is supported by the rapid decrease in cell voltage when the current is reduced (at *ca.* 8 hours in the dataset), although the increase in CO₂ inlet pressure was not affected by this change. Importantly, the pressure increase attributed to precipitate formation did not cause irreversible damage to the electrolyzer cell. The repeated measurement of the given set of parameter combinations (from *ca.* 12 hours), almost completely overlap with those of the first run.

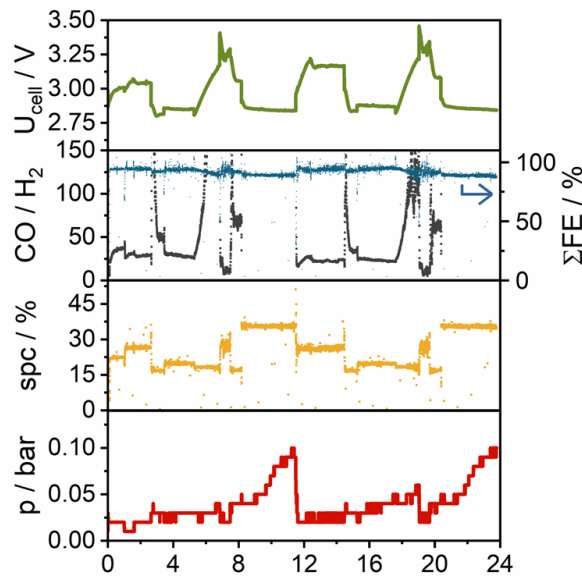


Figure 3.8: *Measuring cell operation outputs on the external test set parameters.*

Figure 3.8 shows the results gathered during the experiment of the external test set. These results were gathered during the operation of a different, but (ideally) identical cell assembly. Note that this cell assembly is "parametrically" the same as the original, however, not the exact same cell as the one that the model training was performed upon. This allows for the research and analyzation of the cell-to-cell variability commonly observed in this field.

Upon also examining Figure 3.9, or Table 3.3 for the overall metrics and Table 3.6 for comparison of the mean values of the predictions and the mean of the measured values, we can conclude that the model loses some of its performance when applied to a different cell assembly. However, some of the values remain close to their original performance. For example, CO [%] and Flow out [sccm] give above 0.9 value for R^2 with MAPE values of 0.0673 and 0.0382, indicating a good predictive capacity. Voltage and CO₂ are moderately well predicted compared to other values such as H₂. Among the derived values, the single-pass conversion is well explained by the model according to the R^2 metric, and a MAPE score of 0.0496 also shows a low error rate

in the calculated value. A low error rate can also be observed in the calculated value of $FE(CO)$ [%], however, due to the already high variance of this value, R^2 shows significant degradation. Due to its high variance, even in the training data, R^2 alone should not be used to judge model performance, hence the inclusion of other metrics to complement each other.

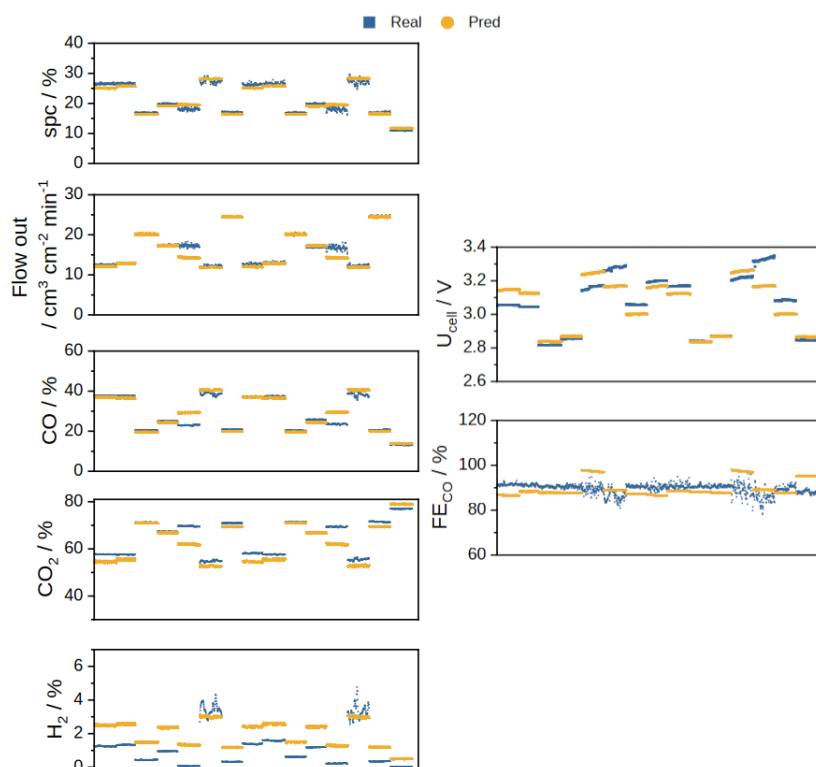


Figure 3.9: Prediction results on the external test set.

Overall, our conclusion is that even for different cell assemblies, some of the values can be predicted more efficiently (e.g., Flow out, CO and CO₂), while others are affected more negatively (H₂, and $FE(CO)$) by applying the same model on a different assembly.

To further support the reliability of the model predictions, a detailed uncertainty analysis was performed. This analysis quantifies the prediction confidence intervals of the input and target variables within single parameter settings. This offers additional insight into the robustness of the model and compares it to the true results of the measurements.

3.5.4 Uncertainty Analysis

This section analyzes the uncertainty in the input features measured during different parameter settings to provide insights into the stability of these specific measure-

ments. We also analyze these uncertainties in the model output, compared to the true measured values, so that we can analyze possible differences and similarities between the learned system and the behavior of the data provided by the measurements.

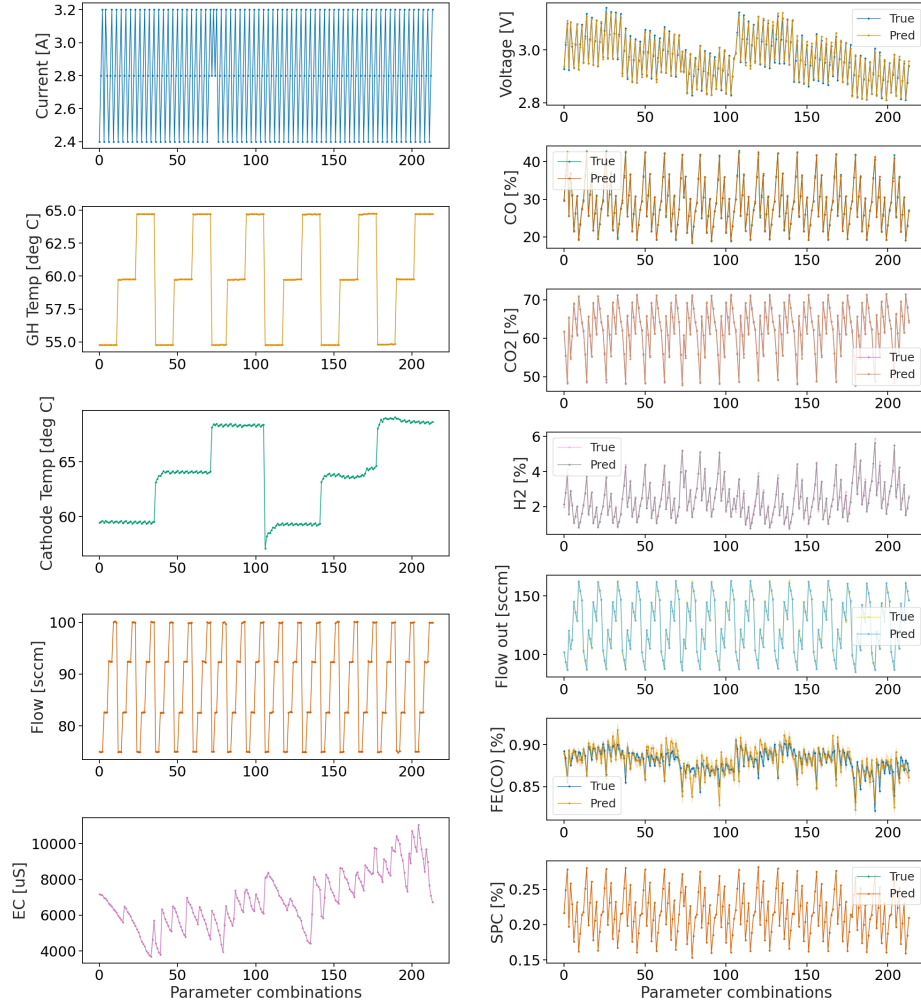


Figure 3.10: *Uncertainty in the data (left) and the model predictions (right) on the full development dataset.*

On the left side of Figure 3.10, means and standard deviations are shown for the input features, while on the right, the output features can be observed for the ANN predictions, and for the true values as well. Note that this figure contains all the parameter combinations used for the development of the model, hence model predictions for the train, validation and the test sets are also included.

As every parameter setting has multiple measurement points, it is reasonable to assume some level of uncertainty. Although we attempted to minimize the variance in the input and output features, some deviation from the measurement sequence

mean is expected. If one examines Figure 3.10 it is apparent that the cell operation for input features, as well as the targeted output features, could be considered stable. We plotted the mean of every parameter setting (ordered by time) and drawn the full scale of data spread (confidence interval of 100 %) for every setting which is represented by a band surrounding the mean value. The lines connecting the dots only help to visualize the ordering of the different measurement sequences. One can see that the largest uncertainty is associated with $FE(CO)$. Note that $FE(CO)$ is calculated from the gas flow rate and the gas composition, and therefore the uncertainty of these two is reflected in this together. To better understand the behavior of the model on previously unseen data, Figure 3.11 visualizes the uncertainty in the data and in the model predictions in the test set.

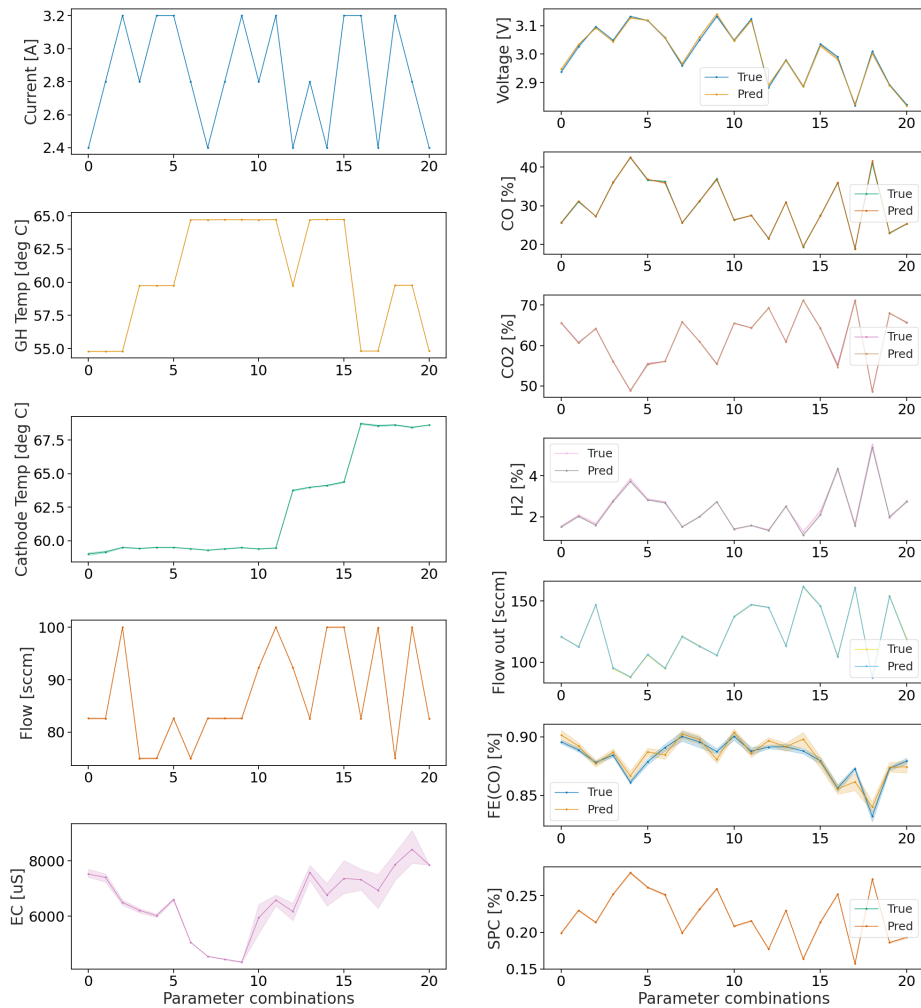


Figure 3.11: Uncertainty in the data (left) and the model predictions (right) on the test set.

The left side of Figure 3.11 shows the mean and the confidence interval for the

input values, while on the right side one can observe these values for the ANN predictions, and the true values for the test set. The stability of the input and output values is comparable to that of the full dataset. To see the numerical representation of these values, see Table 3.4.

Based on the colorings, and the last row that shows the average distance of the predicted means from the true values, it can be said that the model predictions are indeed close to the true values measured on the test station. Note that these parameter combinations have never been seen during the training phase; however, they were produced by the same cell configuration on which the model was trained. Based on the average distances, we can conclude that the ANN model captures the behavior of a particular cell configuration relatively well, as these values are close to zero. For example, the highest average distance (0.3330) can be observed in the flow out column; however, this is still less than 1 % of the column mean (122.89). We reason that while 0.3330 is indeed the largest of the average distances, it is still close to the real values relative to the scale of the Flow out values. The most significant difference (~ 2.6 %) in this sense is observed in the prediction of the H_2 value, which has a column mean of 2.3835, and an average distance of 0.0627.

Table 3.5 presents the standard deviation of the measured and predicted values in the test set. The observed experimental uncertainty, expressed as the standard deviation of repeated measurements, remains low and stable in most output variables with the exception of the EC value, which exhibits comparatively higher variance. However, this deviation is not considered impactful, as SHAP-based sensitivity analysis (using the PartitionExplainer method) indicates that EC has minimal influence on model output (discussed later). Furthermore, EC shows only moderate correlation with the cathode temperature, further limiting its importance in the predictive framework. $FE(CO)$ also shows higher uncertainty, but this is also the case for the measurements conducted on the different cell assembly and stands true for the entirety of the training dataset.

The predictive uncertainty of the model (probably epistemic in nature) is consistently higher (except for a small set of pairs) than the uncertainty of the physical measurement. This possibly reflects the model's limited confidence due to the limited number of parameter settings in the training dataset. The higher predictive uncertainty suggests that the model has not fully learned the underlying dynamics for all operational conditions, which can be mitigated in the future with the addition of data of higher diversity and volume, which should help close the gap between model and measurement uncertainty, improving both the reliability and generalizability of the predictions.

Due to the commonly known cell-to-cell variability in the field of CO₂RR [40], it is expected that the model exhibits greater uncertainty in the external test set. Figure 3.12 visualizes the uncertainty in the data and in the predictions of the model

Table 3.4: Comparing ANN predicted and true mean values for individual measurement sequences for each parameter setting on the test set.

#	Current [A]	GH Temp [°C]	Cell Temp [°C]	Flow [sccm]	EC [μs]	Voltage [V]	CO [%]	CO ₂ [%]	H ₂ [%]	Flow Out [sccm]	FE(CO) [%]	SPC [%]	Type
0	2.40	54.78	59.01	82.62	7522	2.94	25.55	65.63	1.55	120.79	0.90	0.20	True
						2.95	25.69	65.57	1.51	121.04	0.90	0.20	Pred
1	2.80	54.78	59.17	82.61	7403	3.03	31.03	60.77	2.09	112.91	0.89	0.23	Pred
						3.03	31.21	60.65	2.02	112.63	0.89	0.23	Pred
2	3.20	54.80	59.50	99.99	6499	3.10	27.32	64.11	1.68	146.77	0.88	0.21	True
						3.09	27.30	64.22	1.59	146.66	0.88	0.21	Pred
3	2.80	59.74	59.43	74.99	6210	3.05	36.09	56.01	2.81	94.88	0.88	0.25	True
						3.04	36.02	56.09	2.75	95.43	0.89	0.25	Pred
4	3.20	59.74	59.50	75.01	6024	3.13	42.44	48.80	3.85	87.76	0.86	0.28	True
						3.13	42.57	48.89	3.74	88.16	0.87	0.28	Pred
5	3.20	59.76	59.50	82.62	6602	3.12	36.64	55.55	2.87	105.91	0.88	0.26	True
						3.12	36.79	55.28	2.82	106.39	0.89	0.26	Pred
6	2.80	64.70	59.41	74.98	5064	3.06	36.33	56.09	2.74	94.88	0.89	0.25	True
						3.06	35.97	56.12	2.70	95.46	0.89	0.25	Pred
7	2.40	64.71	59.30	82.62	4551	2.96	25.59	65.88	1.53	121.22	0.90	0.20	True
						2.97	25.64	65.83	1.52	121.13	0.90	0.20	Pred
8	2.80	64.71	59.40	82.62	4445	3.05	31.18	61.00	2.02	113.21	0.90	0.23	True
						3.06	31.25	60.94	2.01	113.10	0.90	0.23	Pred
9	3.20	64.71	59.49	82.63	4343	3.13	37.04	55.52	2.73	105.70	0.89	0.26	True
						3.14	36.70	55.32	2.74	105.69	0.88	0.26	Pred
10	2.80	64.71	59.39	92.35	5949	3.05	26.33	65.46	1.39	137.22	0.90	0.21	True
						3.12	27.54	64.36	1.59	147.12	0.89	0.22	Pred
11	3.20	64.72	59.46	100.03	6582	3.12	27.46	64.37	1.58	147.00	0.88	0.22	Pred
						2.88	21.50	69.30	1.38	144.87	0.89	0.18	True
12	2.40	59.75	63.75	92.37	6174	2.89	21.61	69.28	1.34	144.86	0.90	0.18	Pred
						2.98	30.96	60.99	2.51	113.42	0.89	0.23	True
13	2.80	64.71	63.99	82.60	7577	2.98	30.97	60.90	2.52	113.45	0.89	0.23	Pred
						2.89	19.37	71.18	1.28	161.37	0.89	0.16	True
14	2.40	64.74	64.13	100.03	6770	2.89	19.53	71.28	1.12	162.04	0.90	0.16	Pred
						3.04	27.49	64.19	2.28	145.76	0.88	0.21	True
15	3.20	64.74	64.38	100.04	7365	3.03	27.55	64.35	2.12	146.12	0.88	0.21	Pred
						2.99	35.90	55.31	4.30	105.11	0.86	0.25	True
16	3.20	54.81	68.73	82.61	7323	2.98	36.11	54.61	4.37	104.48	0.86	0.25	Pred
						2.82	19.07	71.22	1.68	161.09	0.87	0.16	True
17	2.40	54.82	68.58	99.97	6936	2.82	18.81	70.95	1.56	160.48	0.86	0.16	Pred
						3.01	41.01	48.71	5.51	87.68	0.83	0.27	True
18	3.20	59.76	68.64	75.05	7862	3.00	41.52	48.47	5.38	87.17	0.84	0.27	Pred
						2.89	22.93	67.95	1.96	154.19	0.87	0.19	True
19	2.80	59.77	68.46	100.00	8417	2.89	23.03	67.99	2.02	153.78	0.87	0.19	Pred
						2.82	25.34	65.76	2.78	119.19	0.88	0.19	True
20	2.40	54.81	68.64	82.61	7860	2.82	25.44	65.63	2.73	118.46	0.88	0.19	Pred
						Column mean						3.00	29.86
Averaged distance of values						0.0050	0.1542	0.1366	0.0627	0.3330	0.0048	0.0012	

For every two rows found on the right side there is a single corresponding input setting, which is the measured mean value of the specific settings. The first row of every pair represents the true (measured) mean value of the output features, while the second row is the predicted mean value of the model. Values in ■ light blue indicate lower values for that particular column or feature, with ■ neutral gray coloring indicating values in the middle range, and ■ red indicating values in the higher range of that setting. The purpose of coloring is twofold: for one, they indicate where to search for parameters that are possibly outside the range of the trained parameter settings, and two, they are comparable by pair, meaning that similar if for one index two colors are similar in shade, they are also similar in value. The last row (averaged distance of values) indicates the averaged distance between every pair of values per column, showing how far the prediction means are from the true measurement means on average.

Table 3.5: Comparing ANN predicted and true std values for individual measurement sequences for each parameter setting on the test set.

#	Current [A]	GH Temp [°C]	Cell Temp [°C]	Flow [sccm]	EC [μs]	Voltage [V]	CO [%]	CO ₂ [%]	H ₂ [%]	Flow Out [sccm]	FE(CO) [%]	SPC [%]	Type
0	0.0002	0.0298	0.4755	0.4780	580	0.0157	0.0893	0.2656	0.0389	0.8067	0.0068	0.0019	True
						0.0135	0.3638	0.4191	0.0972	1.6048	0.0155	0.0036	Pred
1	0.0002	0.0297	0.3813	0.4746	612	0.0147	0.1049	0.2458	0.1060	0.7911	0.0075	0.0023	True
						0.0109	0.4307	0.3877	0.0990	1.5168	0.0133	0.0038	Pred
2	0.0002	0.0230	0.1080	0.5013	359	0.0011	0.2502	0.2430	0.0931	0.9727	0.0055	0.0018	True
						0.0064	0.4081	0.4006	0.0702	1.9901	0.0176	0.0044	Pred
3	0.0002	0.0177	0.1067	0.5028	342	0.0041	0.3609	0.2989	0.1064	1.0263	0.0085	0.0027	True
						0.0053	0.5243	0.4933	0.0821	1.5295	0.0131	0.0038	Pred
4	0.0001	0.0378	0.1166	0.5015	272	0.0059	0.4080	0.3507	0.1213	1.0636	0.0085	0.0032	True
						0.0067	0.6581	0.6636	0.1132	1.6011	0.0154	0.0053	Pred
5	0.0002	0.0205	0.1038	0.4900	203	0.0058	0.3992	0.3366	0.1012	1.3143	0.0106	0.0035	True
						0.0067	0.5610	0.4923	0.0883	1.5353	0.0152	0.0049	Pred
6	0.0003	0.0233	0.0860	0.5080	30	0.0159	0.3839	0.2851	0.1124	1.3551	0.0124	0.0038	True
						0.0062	0.5691	0.5606	0.0908	1.6494	0.0137	0.0041	Pred
7	0.0002	0.0292	0.0926	0.4837	28	0.0116	0.2031	0.1973	0.0407	2.1901	0.0181	0.0042	True
						0.0054	0.4688	0.4468	0.0649	1.7756	0.0168	0.0040	Pred
8	0.0003	0.0286	0.0891	0.4767	50	0.0095	0.2721	0.2048	0.0650	2.4875	0.0207	0.0054	True
						0.0053	0.4423	0.4700	0.0711	1.7409	0.0148	0.0040	Pred
9	0.0001	0.0287	0.0970	0.4777	86	0.0136	0.2038	0.1741	0.0362	1.2230	0.0103	0.0033	True
						0.0076	0.6054	0.4981	0.0943	1.6644	0.0152	0.0049	Pred
10	0.0003	0.0852	0.0880	0.4884	2124	0.0145	0.1902	0.1736	0.0620	2.3005	0.0160	0.0039	True
						0.0151	0.3972	0.4183	0.0702	1.9341	0.0149	0.0037	Pred
11	0.0002	0.0300	0.1025	0.4526	867	0.0139	0.1735	0.1290	0.0661	2.1370	0.0139	0.0034	True
						0.0084	0.3749	0.4028	0.0728	1.8038	0.0158	0.0040	Pred
12	0.0004	0.0188	0.2535	0.4945	1283	0.0096	0.0706	0.0779	0.1007	0.9193	0.0069	0.0017	True
						0.0077	0.3606	0.4202	0.0715	1.8460	0.0165	0.0034	Pred
13	0.0002	0.0263	0.1016	0.4602	1163	0.0156	0.2172	0.1827	0.0768	1.0687	0.0116	0.0032	True
						0.0106	0.3961	0.3995	0.0674	1.4860	0.0126	0.0035	Pred
14	0.0002	0.0301	0.1914	0.4643	1593	0.0129	0.1405	0.1017	0.1683	1.6904	0.0097	0.0018	True
						0.0196	0.3708	0.4947	0.0822	2.3039	0.0209	0.0038	Pred
15	0.0002	0.0602	0.2675	0.4573	2530	0.0170	0.2447	0.1774	0.2266	2.1668	0.0123	0.0031	True
						0.0271	0.4412	0.4737	0.1583	1.9536	0.0167	0.0043	Pred
16	0.0000	0.0355	0.2773	0.4242	1734	0.0108	0.2053	0.1475	0.3077	1.2440	0.0125	0.0039	True
						0.0099	0.5685	0.5724	0.3313	1.7762	0.0168	0.0052	Pred
17	0.0003	0.0365	0.3560	0.4236	2924	0.0082	0.1581	0.0983	0.2216	1.7136	0.0070	0.0014	True
						0.0152	0.6146	0.6243	0.1856	2.4844	0.0321	0.0059	Pred
18	0.0002	0.0810	0.1948	0.4194	2162	0.0050	0.7164	0.4255	0.4650	0.9320	0.0192	0.0061	True
						0.0147	0.7343	0.6980	0.3039	1.7018	0.0192	0.0063	Pred
19	0.0002	0.0306	0.1764	0.4431	2279	0.0118	0.1911	0.2644	0.1092	1.0977	0.0070	0.0017	True
						0.0113	0.4807	0.5566	0.1664	2.2130	0.0211	0.0047	Pred
20	0.0004	0.0571	0.0365	0.3849	13	0.0010	0.0729	0.0721	0.0222	1.2292	0.0091	0.0022	True
						0.0063	0.3930	0.4305	0.0711	1.5438	0.0170	0.0037	Pred

For more information on table colorings refer to Table 3.4.

on the external test set.

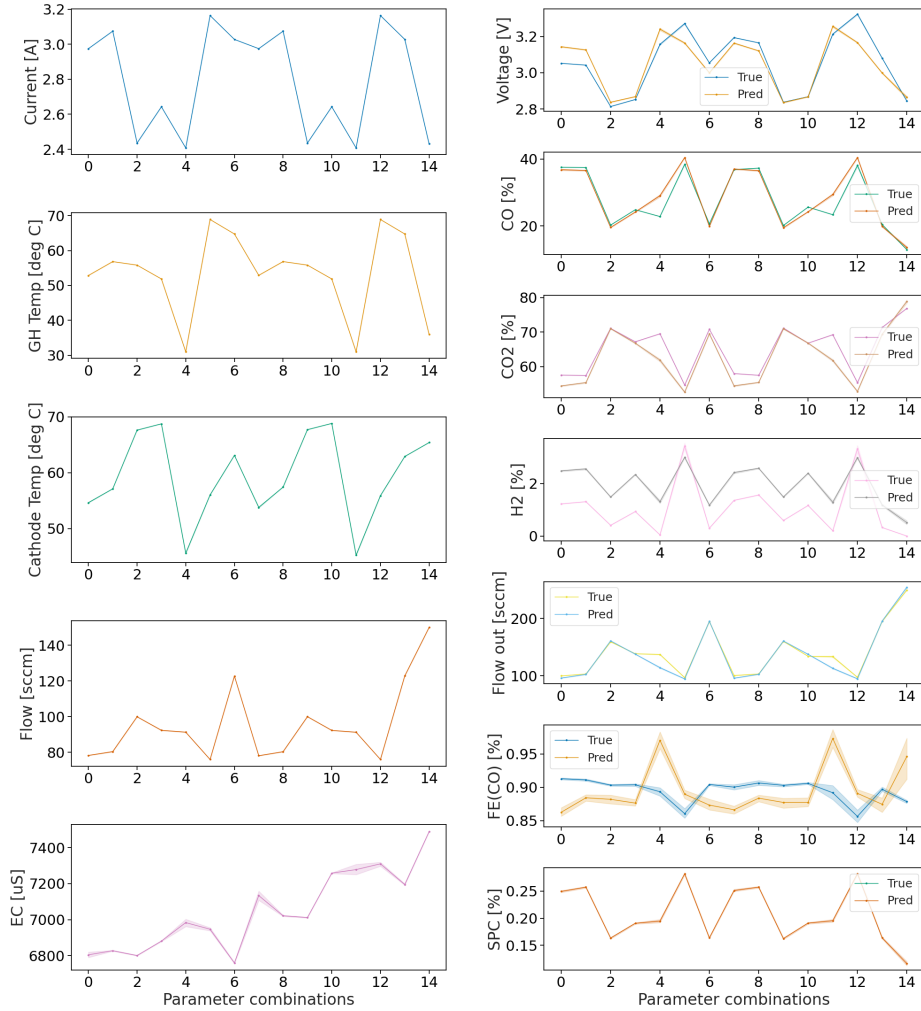


Figure 3.12: *Uncertainty in the data and the model predictions on the external test set.*

Figure 3.11 represents the uncertainty in the data and the model predictions on the external test set. The left side shows the mean and the confidence interval for the input values, while on the right side one can observe these values on the ANN predictions, and the true values.

The uncertainty of the measured input parameter settings is comparable to the uncertainty in the case of the test set (Figure 3.11), with the only notable difference being the value of *electric conductivity* (EC), which has higher variation in the test set. This is discussed later in the section dedicated to model sensitivity analysis, however, it should be noted here also that EC does not have high importance in determining either of the output features and as such, greater variance in the output features should not be attributed to this fact alone.

In the case of the external test set, a considerably higher error can be observed on the model side, especially in the case of the predicted H_2 value. An increase in error is expected in this case due to the fact that in most cases at least one parameter setting is outside the range that the model was trained on, which in turn can cause the model to respond with a higher variability in the outputs even for small change in the inputs. Also, a high uncertainty in the predicted $FE(CO)$ value in the last parameter combination with the following settings is especially conspicuous:

- Cell Temp Setpoint [deg C] = 66
- GH Temp Setpoint [deg C] = 36
- MFC Setpoint [sccm] = 150
- Current Limit Setpoint [A] = 2.432

This can be attributed to the fact that GH and MFC setpoint both significantly fall outside the range on which the model was trained, causing greater uncertainty in the model predictions even for smaller variations in the input data (see Table 3.7).

The prediction performance of the model shows degradation compared to the test set. However, examining the target values $FE(CO)$ [%] and CO_2 Conversion Rate [%], one can see an overlap in these values. The mean of *Faraday Efficiency* and the conversion rate are moderately well predicted with average distances of 0.0397 and 0.0088 which are ~ 4 % of the corresponding column means (0.8961 and 0.2084, respectively). In the case of cell voltage, the average distance (0.0565) is ~ 2 % of the corresponding column mean (3.0479), which suggests that a varying EC value does not necessarily imply unstable cell operation, even in the case of cell voltage, where EC is measured to be the third most important feature in predicting this value (see Section 3.5.6), although still with a relatively low feature importance value compared to the first two (Current and Cell temperature).

In Table 3.6 it can be seen that the model can struggle to predict parameter settings when two or more parameters fall outside the range of the training data. For example, the value setting for the 14th index produces high uncertainty in the model (as shown on Figure 3.12), with a mean value also being off by a higher margin as opposed to other predictions, where the GH temperature and the Flow rate is out of bounds. Similar effects can be observed when both temperature values fall outside the specified range for the 4th and 11th indexed rows, since here the predicted output flow rate is ~ 15 % less than the true output flow, resulting in a higher Faraday efficiency prediction. In the 0 and 1 indexed rows we can see that the model performs significantly better, even though the cell temperature falls below 60 °C (the minimum cell temperature in the training data). This can be observed in setting 3 and 5 too, where inputs are relatively close to ranges of the training data. This

Table 3.6: Comparing ANN predicted and true mean values for individual measurement sequences for each parameter setting on the external test set.

#	Current [A]	GH Temp [°C]	Cell Temp [°C]	Flow [sccm]	EC [μs]	Voltage [V]	CO [%]	CO ₂ [%]	H ₂ [%]	Flow Out [sccm]	FE(CO) [%]	SPC [%]	Type
0	2.97	52.86	54.64	78.28	6805	3.05	37.56	57.52	1.22	100.00	0.91	0.26	True
						3.14	36.83	54.40	2.47	96.22	0.86	0.25	Pred
						3.04	37.50	57.42	1.31	103.32	0.91	0.26	True
1	3.08	56.84	57.11	80.44	6828	3.13	36.52	55.39	2.56	102.78	0.88	0.26	Pred
						2.81	20.24	71.05	0.41	159.42	0.90	0.17	True
						2.84	19.52	70.94	1.49	160.80	0.88	0.16	Pred
2	2.44	55.83	67.61	99.96	6801	2.85	24.85	67.17	0.93	138.62	0.90	0.20	True
						2.87	24.18	66.80	2.36	137.72	0.88	0.19	Pred
						3.16	22.79	69.49	0.04	137.40	0.89	0.18	True
4	2.41	30.99	45.60	91.31	6984	3.25	29.27	61.90	1.34	113.82	0.97	0.20	Pred
						3.27	38.44	54.64	3.43	97.01	0.86	0.27	True
						3.17	40.39	52.59	3.00	94.73	0.89	0.28	Pred
5	3.16	68.92	56.02	76.01	6947	3.06	20.61	70.79	0.30	194.65	0.90	0.17	True
						3.00	19.86	69.43	1.18	195.57	0.88	0.16	Pred
						3.19	36.86	57.97	1.36	100.66	0.90	0.26	True
7	2.97	52.91	53.79	78.11	7134	3.16	37.01	54.23	2.41	95.55	0.86	0.25	Pred
						3.17	37.30	57.47	1.56	103.32	0.91	0.26	True
						3.12	36.63	55.31	2.59	102.57	0.88	0.26	Pred
8	3.08	56.85	57.42	80.39	7022	2.84	20.13	71.16	0.59	160.22	0.90	0.17	True
						2.83	19.56	71.04	1.49	161.06	0.88	0.16	Pred
						2.87	25.63	66.77	1.17	134.25	0.91	0.20	True
10	2.64	51.88	68.81	92.33	7258	2.87	24.19	66.81	2.40	137.59	0.88	0.19	Pred
						3.21	23.35	69.22	0.20	133.57	0.89	0.18	True
						3.26	29.52	62.00	1.32	113.75	0.98	0.20	Pred
11	2.41	30.98	45.25	91.28	7278	3.32	38.10	55.34	3.32	97.55	0.86	0.27	True
						3.17	40.41	52.75	2.97	95.05	0.89	0.28	Pred
						3.08	20.41	71.31	0.33	195.10	0.90	0.17	True
13	3.03	64.76	62.90	122.87	7195	3.00	19.81	69.39	1.18	195.31	0.87	0.16	Pred
						2.84	12.88	76.85	0.00	249.60	0.88	0.11	True
						2.87	13.76	78.90	0.49	255.78	0.97	0.12	Pred
Column mean						3.05	28.14	63.87	1.51	138.77	0.90	0.21	
Averaged distance of values						0.06	1.65	2.44	0.97	4.66	0.04	0.01	

For more information on table colorings refer to Table 3.4.

demonstrates the model's ability to generalize to some extent even outside the training data range. However, it also shows its limitations *i.e.*, when the model is used well beyond the training data ranges, it can produce unrealistic results.

Table 3.7: Comparing ANN predicted and true std values for individual measurement sequences for each parameter setting on the external test set.

#	Current [A]	GH Temp [°C]	Cell Temp [°C]	Flow [sccm]	EC [μs]	Voltage [V]	CO [%]	CO ₂ [%]	H ₂ [%]	Flow Out [sccm]	FE(CO) [%]	SPC [%]	Type
0	0.0003	0.0322	0.1759	0.4244	52	0.0009	0.0617	0.0665	0.0173	0.4732	0.0048	0.0020	True
						0.0091	0.6501	0.6542	0.1069	1.9128	0.0191	0.0054	Pred
1	0.0001	0.0352	0.0327	0.4541	7	0.0005	0.0749	0.0849	0.0158	0.4181	0.0046	0.0020	True
						0.0064	0.5199	0.4818	0.0766	1.5030	0.0143	0.0044	Pred
2	0.0001	0.0334	0.0495	0.4990	6	0.0005	0.0479	0.0696	0.0149	0.6288	0.0043	0.0011	True
						0.0067	0.4023	0.5193	0.0633	2.2711	0.0221	0.0042	Pred
3	0.0002	0.0205	0.0444	0.4646	8	0.0010	0.0919	0.1025	0.0137	0.8864	0.0061	0.0016	True
						0.0049	0.3779	0.5045	0.0669	1.9836	0.0202	0.0043	Pred
4	0.0003	0.0245	0.3618	0.4489	58	0.0108	0.1759	0.1449	0.0165	2.4406	0.0167	0.0035	True
						0.0199	1.0380	1.1355	0.1949	3.4511	0.0415	0.0085	Pred
5	0.0002	0.0335	0.1201	0.3982	29	0.0105	0.6885	0.3153	0.3509	1.5382	0.0221	0.0072	True
						0.0097	0.6921	0.5881	0.1106	2.0319	0.0215	0.0070	Pred
6	0.0004	0.0198	0.0333	0.5913	11	0.0009	0.0753	0.0861	0.0111	0.7739	0.0045	0.0011	True
						0.0095	0.5254	0.5212	0.0931	2.6940	0.0237	0.0045	Pred
7	0.0002	0.0302	0.2718	0.3870	86	0.0036	0.1987	0.1370	0.0187	1.3220	0.0119	0.0037	True
						0.0105	0.6108	0.5614	0.1020	1.7756	0.0183	0.0052	Pred
8	0.0002	0.0345	0.0289	0.4204	10	0.0014	0.0953	0.0913	0.0243	1.2470	0.0113	0.0037	True
						0.0061	0.4699	0.4318	0.0820	1.4458	0.0146	0.0045	Pred
9	0.0000	0.0348	0.0281	0.5035	8	0.0008	0.0660	0.0691	0.0091	1.0800	0.0062	0.0013	True
						0.0061	0.3575	0.4546	0.0662	2.0721	0.0188	0.0035	Pred
10	0.0002	0.0157	0.0326	0.4942	7	0.0009	0.0577	0.0871	0.0137	0.6400	0.0047	0.0015	True
						0.0047	0.3950	0.4513	0.0767	1.8108	0.0185	0.0042	Pred
11	0.0000	0.0286	0.3554	0.4768	90	0.0068	0.2181	0.1822	0.0201	4.2099	0.0311	0.0063	True
						0.0196	1.0621	0.9466	0.1921	3.8217	0.0435	0.0088	Pred
12	0.0002	0.0344	0.1049	0.3917	37	0.0114	0.9477	0.4185	0.4276	1.7241	0.0282	0.0092	True
						0.0103	0.7184	0.6648	0.1116	1.7780	0.0187	0.0061	Pred
13	0.0004	0.0231	0.0315	0.5465	13	0.0028	0.1593	0.1522	0.0150	1.3048	0.0092	0.0019	True
						0.0090	0.4897	0.5877	0.0875	3.1620	0.0270	0.0050	Pred
14	0.0002	0.0299	0.0336	0.3126	8	0.0008	0.0710	0.0768	0.0000	1.3996	0.0071	0.0009	True
						0.0197	1.2631	1.3293	0.2443	5.2965	0.0954	0.0117	Pred

For more information on table colorings refer to Table 3.4.

From Table 3.7 we can see that the model generally has more uncertainty (*e.g.*, parameter setting #14) than the real system itself. This is still the case, even when the predicted mean is close to the true value. This is a limitation of the current model, which could later be mitigated by including additional high-quality data points in the training set.

3.5.5 Analysis of Possible Feature Correlations

It is known that not all model sensitivity analysis tools can be used safely in the presence of stronger input correlations. To mitigate this problem, we analyzed the possible correlations between the input data (Figure 3.13) to ensure the validity of our chosen method.

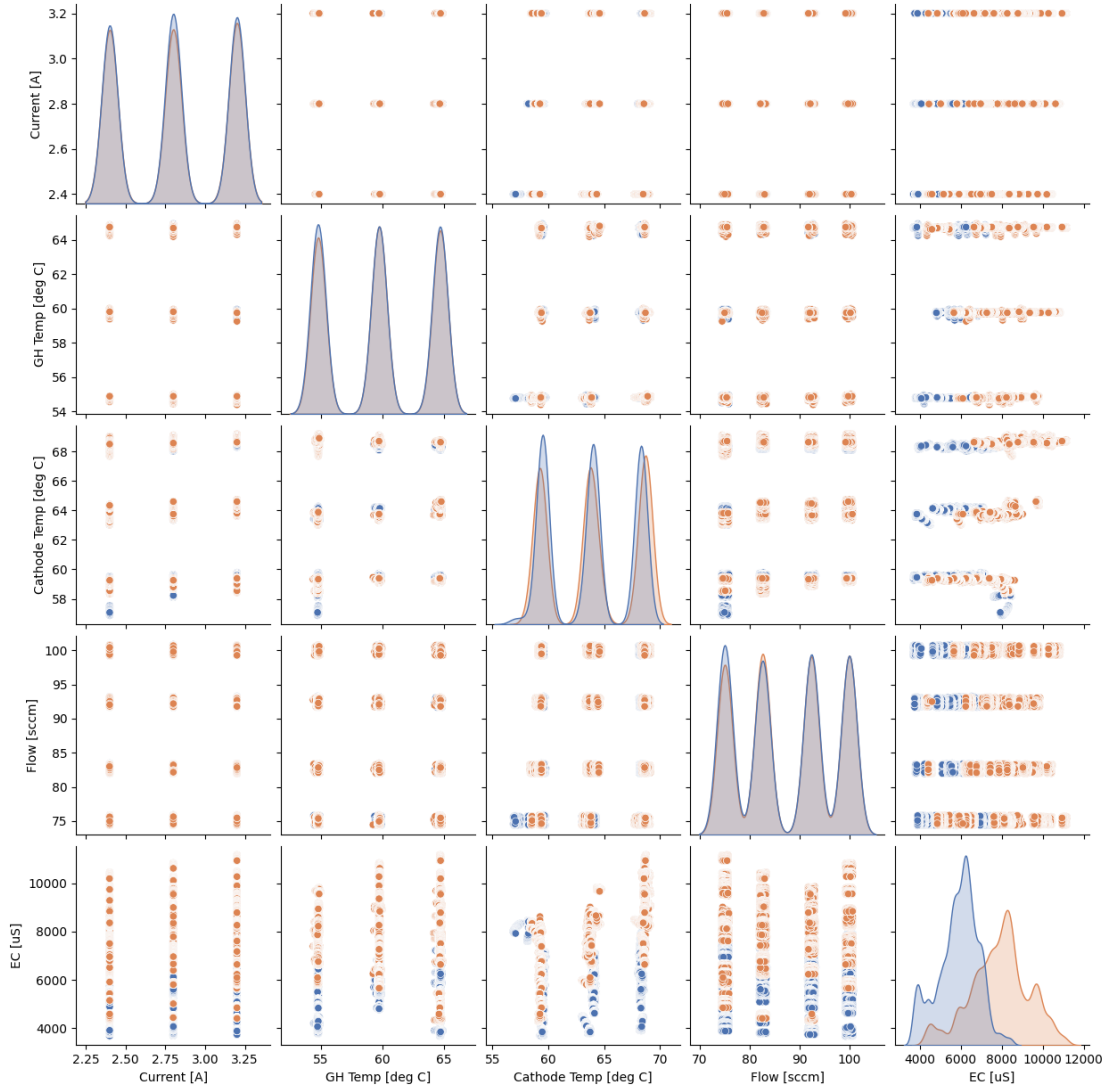


Figure 3.13: Correlation between different input parameters in the model development dataset. Meas ID signifies the repeated measurement sequences used for model development, with the 0th ID corresponding to the first, and the 1st ID corresponding to the repeated measurements. The diagonal in the plot shows the distribution of the different input feature values, while the scatter plots show the individual measurement points sample-by-sample. These scatter plots help visualize obvious patterns of correlations between the input features, if any.

The distribution plots in Figure 3.13 show the different control variable settings, since the majority of samples gather around the specific settings and are easily separated by the different maximums. These local maximums correspond to the settings defined as the basis for the measurement of the training data (see Table 3.1). There are no obvious correlations between any of the input features based on the plot; however, a moderate positive correlation between EC and Cathode Temp could be observed, implying the temperature dependence of the measured conductivity values (despite using a temperature-compensated conductivity probe). To quantify this correlation, we calculated the *Pearson correlation coefficient* (PCC) [26] for each pair of input features.

	Current [A]	GH Temp [$^{\circ}$ C]	Cell Temp [$^{\circ}$ C]	Flow [sccm]	EC [μ s]
Current	1.0000	-0.0147	0.0371	-0.0131	0.0309
GH Temp	-0.0147	1.0000	0.0298	-0.0036	-0.0222
Cell Temp	0.0371	0.0298	1.0000	0.0237	0.3744
Flow	-0.0131	-0.0036	0.0237	1.0000	-0.0368
EC	0.0309	-0.0222	0.3744	-0.0368	1.0000

Table 3.8: Correlation between input features, numerically representing Figure 3.13.

Table 3.8 shows that there is no significant correlation between most of the input features, although a moderate correlation between Cathode Temp and EC can be observed. Note that although EC appears as a model input, it is not a controllable parameter.

3.5.6 Model Sensitivity Analysis

In traditional *Sensitivity Analysis* (SA) methods such as *one-at-a-time* (OAT), only one factor of the model is changed at a time while others are kept fixed. In each iteration, the algorithm selects an independent variable (in our case input) to perturb so it can analyze how that particular variable affects the model outcome. This method is model agnostic; however, one significant drawback is that it cannot detect feature interactions or higher-order, nonlinear effects. Another option is to use Sobol' indices; however, classical Sobol' sensitivity indices are defined under the assumption that the input variables are independent [157]. For models featuring dependent inputs, this may lead to a wrong interpretation because the sensitivity induced by the dependence between two factors is implicitly included in their Sobol' indices [20].

A different approach is to use Shapley effects instead of Sobol' indices [158]. The Shapley value concept from cooperative game theory allocates the total output vari-

ance (the "value") among inputs fairly, accounting for correlations and interactions. Shapley indices have the advantages of always summing to one and being well defined under dependence. Iooss *et al.* show that Shapley effects allocate the mutual contribution (due to correlation and interaction) of a group of inputs to each individual input, effectively disentangling shared effects [79]. Shapley-based measures also give more interpretable attributions when inputs are dependent [13, 79]. In practice, many modern SA toolkits (*e.g.*, SHAP) now offer Shapley-effect estimation as the recommended approach for correlated inputs when using typical machine learning models [107].

When applying Shapley values to machine learning, it can be interpreted by making the following equivalences.

- **Features = Players:** Input features are considered to be players of a cooperative game.
- **Model Prediction = Payout:** The prediction of the model could be thought of as the payout.
- **Baseline/Reference Value:** SHAP often uses the average model prediction (over the training data, or an optionally aggregated subset (*e.g.*, with K-means) of it) as a baseline. SHAP values then explain how each feature shifts the prediction from this baseline expectation to the predicted value of the specific instance.

Eventually, we chose SHAP for its proven ability to explain even black-box models, such as *Deep Neural Networks* (DNNs). SHAP [186] also has a proven record of being consistent and accurate [44, 128] even in the presence of feature interactions. Furthermore, for a small number of input features (such as the case in this work), it could also be argued that it is a more adequate choice over other methods, such as constructing Sobol' indices, due to its compelling properties [132, 137, 170].

With the use of SHAP values we were able to rank each input feature based on its importance score for a specific output feature. In the following sections, we discuss how different control parameters might affect the experiment based on the extracted Shapley values.

In selecting a specific SHAP method to approximate Shapley values, we considered the fact that the input features (Current, GH Temp, Cell Temp, and Flow) except for the non-controllable EC parameter are largely independent, with no strong evidence of direct interactions, with EC and Cathode Temp being moderately dependent on each other. Some implementations (*e.g.*, KernelSHAP) make the assumption of feature independence, and as such, we only considered methods that account for possible feature correlations [107].

The Partition SHAP method provides a model-agnostic approach to explaining model output by computing the Shapley values through a hierarchical clustering of characteristics, resulting in the Owen values from cooperative game theory [133]. Unlike Kernel or Sampling-based SHAP methods, which scale exponentially with the number of features, Partition SHAP achieves exact computation with quadratic complexity when employing a balanced partition tree. A notable advantage of this approach is its ability to account for feature dependencies. When correlated features are grouped within the hierarchy, Partition SHAP allocates their collective contribution without requiring perturbations that disrupt their statistical relationships.

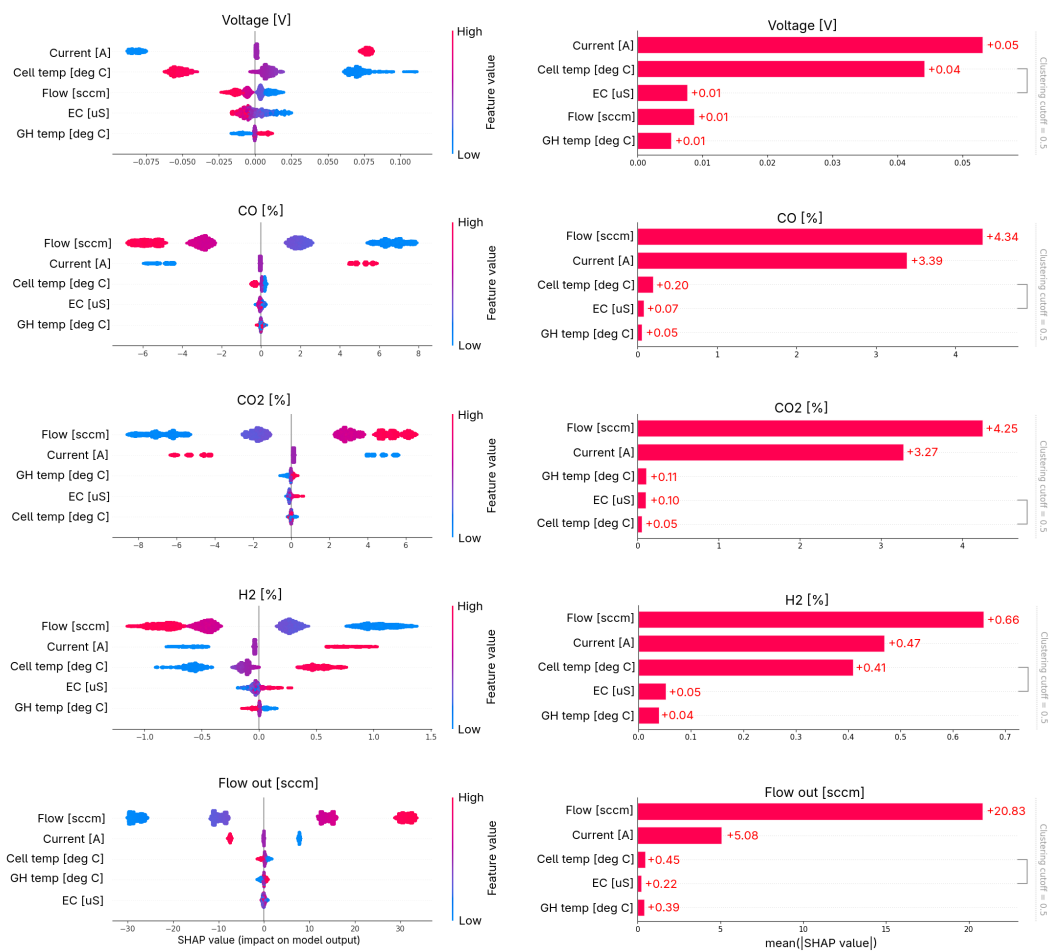


Figure 3.14: Local SHAP-based feature importance on the left, and global SHAP-based feature importance on the right.

On the left side of Figure 3.14, we can see the SHAP based ANN model sensitivity where all the input parameters are sorted based on their impact on model output which can be observed on the right colorbar. On the right, SHAP values are aggregated over individual input parameters (taking the mean), so that they show their global feature importance, and also show the groupings that the Partition SHAP

method accounted for, which may cause some reordering compared to the local contributions, as Partition SHAP groups correlating features together.

Examining the figure one can immediately see how each input feature contributes to different model outcomes. In the first row, we can see that the cell voltage is most affected by the applied current and the temperature of the cell. Positively correlating in the case of electric current while negatively correlating with the temperature. The grouping in values (on the left figure) can be attributed to the fact that the total of 108 parameter combinations were formed of 4 flow rates and 3 different levels for the other 3 parameters ($3^3 \times 4$ combinations). However, the order of the features depends on the mean attribution of one particular input feature. For example, this means that the cell voltage in the first row is mostly affected by the applied current. The concentration of gaseous elements depends mostly on the flow rate and is highly affected by the current. This is caused by both the different dilution changes and the selectivity changes at different flow rates together. The concentration of H₂ in the gas stream increases with the cell temperature, most probably due to the greater amount of water that crosses the membrane from the anode to the cathode, hence promoting the formation of H₂. Lastly, the output flow rate is mostly affected by the inlet flow rate, with a positive correlation, while also negatively affected by the set current. The latter is caused by the loss of carbon dioxide due to its reaction with electrochemically generated hydroxide ions ($CO_2 + 2e^- + H_2O \rightarrow CO + 2OH^-$, $CO_2 + 2OH^- \rightarrow CO_3^{2-} + H_2O$), and the transport of the formed carbonate ions to the anode.

Overall, the SHAP analysis provided clear and intuitive insights into some of the input features' influence on model predictions. The strong effects of current and cell temperature reflect their effect in driving the electrochemical processes, which matches well with our current understanding. The grouped values showed correlation between some of the input features, namely EC and Cathode Temp, which is validated by our previous findings when analyzing possible input correlations. These values in turn play a pivotal role in determining the cell voltage. These findings demonstrate that SHAP is not only effective for model interpretation but also valuable for exploring how key parameters shape the behavior of the system.

3.6 Discussion and Concluding Remarks

The thesis explores the optimization of AEM separated electrolyzer cells, and discusses the performance of different ML methods, ultimately selecting an ANN network in the process. For model training, a dataset of 108 different parameter combinations was used and evaluated on identical and different cell assemblies as well. The predictive capacity of the developed model was investigated with the parameter settings carefully separated from the training data, and was also evaluated on a

dataset produced after selecting 8 more combinations on another cell assembly (external set). Compared to the performance achieved on the test set, on the external set R^2 values dropped moderately for cell voltage, amount of CO and CO₂, and output flow rate. The model performed worse in predicting the amount of H₂ and $FE(CO)$. In terms of MAE, there is an order of magnitude increase in all predicted variables compared to the test set. However, this increase is still relatively low if we consider the effect of cell-to-cell variations, typically reported in the field. MAPE also shows a slight deterioration in model performance for all target variables, most notably for the H₂ evolution rate. For the rest of the target variables, MAPE increased by only a few percentage points. It should be noted that both $FE(CO)$ and SPC predictions were less than 5 % away from the measured values based on MAPE alone.

As noted above, we believe that the differences in the measured and predicted values are rooted in the cell-to-cell variation. Although this marks the path for important future research regarding the reduction of this variation, continuing to further develop the models and methodology presented is also of great importance. The latter includes defining and running shorter cell diagnostic measurements, which can be used to "calibrate" the ML model. Note that this would also allow accounting for cell aging, and therefore these tools can also aid long-term CO₂ electrolysis, under conditions continuously adapted to the current state-of-health of the electrolyzer cell/stack. The implementation of ANN-based methods in the process control architecture as one of the most realistic scenarios for the industrialization of CO₂RR.

This work shows the versatility of machine learning techniques and their applicability across a multitude of fields. In earlier chapters we looked at using *Genetic Algorithms* and *Deep Learning* techniques to handle abstract software elements. Now, our optimization of the CO₂ electrolyzer makes it clear that core machine learning ideas, like pattern recognition, optimization, and predictive modeling, work just as well to model complex electro-chemical processes in real-life physical systems.

The ANN approach succeeded in the prediction of electrolyzer performance, thus allowing to save time and resources spent on real-life experimentations. When paired with SHAP-based analysis for interpretability, it demonstrates how modern *Machine Learning* delivers precise predictions along with useful insights. These benefits hold up across very different areas.

This kind of transferability, however; can signify something bigger. Advances in machine learning, from *Software Engineering* to optimization strategies and interpretability tools, reach well beyond their starting points. Our work promotes the efforts of many researchers in the field, who are constantly working to achieve more sustainable energy resources. By optimizing CO₂ reduction, while also creating high-value byproducts, which later could be used in many industrial processes, we also take part in adding value to this highly researched field.

The work presented in this thesis is the result of several years of work. Improve-

ments are continuously made on the test station to provide stable and reliable operation of electrolyzer cells, which makes it possible for models to learn the behaviors of such electrochemical systems even more precisely, with less uncertainty. These results show that even these complex systems could be modeled using ML techniques, making it possible for further experimentation, such as applying *Reinforcement Learning* (RL) to continuously monitor and adjust system parameters to maintain the cell at optimal operational levels. This could not only optimize efficiency but also possibly prevent premature cell failures, extending the lifespan of such systems. For the future, we are planning to extend our modeling capabilities so that our models can continuously adapt to changing conditions, such as cell aging and other uncontrollable but otherwise measurable factors.

The author of this PhD thesis is responsible for the following contributions presented in this chapter:

1. Optimization of CO₂ reduction of AEM separated electrolyzer cells through *Machine Learning* techniques.
2. Implementation of the AI-guided framework for optimizing the CO₂ reduction of AEM separated electrolyzer cells.
3. Devising and conducting experiments on parameter setting selection optimization.
4. Qualitative and quantitative evaluation of model performance on different cell assemblies.
5. Contributing to the explainability of the real-life system via performing Shapley-value-based evaluation of model predictions.

This thesis point is based on scientific papers published by the author:

- Angelika A. Samu, Dániel Horváth, Balázs Endrődi, László Vidács, and Csaba Janáky. "Neural Algorithm Aided Operation of CO₂ Electrolyzers." In ACS Energy Letters. 2025. [151]

Summary

This dissertation discusses three main topics, all related to *Machine Learning* (ML) and its applications. It is apparent that ML plays a significant role in the advancement of many fields of today's science including automated patch generation for buggy programs and even the performance optimization of electrochemical cells. Although this study may be short for presenting all the possibilities and advancements made by *Artificial Intelligence* (AI) it still shows the significance of such studies and proves the capabilities of smarter, AI-guided frameworks, and the wide range of fields to which it could potentially be applied.

Smarter Repair Pipelines for JavaScript APR

Chapter 1 addresses a significant gap in the field of *Automated Program Repair* (APR) by focusing on JavaScript, the most widely used programming language for web development, yet mostly overlooked in existing APR research. To this end, the thesis presents GenProgJS, a genetic search-based APR tool tailored for JavaScript through its operators specifically designed to also patch JavaScript-specific bugs. The tool builds on the principles of traditional GenProg while also considering language-specific intricacies, and is evaluated against a benchmark of real JavaScript bugs. This work also places GenProgJS within the broader context of recent advancements in *Machine Learning* and hybrid APR systems. It shows that traditional approaches are not necessarily outdated but complementary to recent advances, comparing the tool's performance with LLMs, such as GPT-2. The work also explores the possibility of the extension of APR pipelines with patch validation based on similarity as a possible step. The research presented could potentially lead to more effective and explainable repair systems for dynamic languages such as JavaScript.

Contributions of the thesis

In the **first thesis group**, the contributions are related to the development and evaluation of GenProgJS, a tool *Genetic Algorithm*-based APR designed to repair bugs potentially present in JavaScript projects, featuring novel genetic operators specifi-

cally tailored to suit the unique characteristics and flexible nature of the programming language. He contributed to the results of GenProgJS, repairing errors in 31 JavaScript programs, of which in 12 cases the generated patches were semantically similar to those written by human developers. A detailed qualitative and quantitative analysis was carried out to assess the generated patches, highlighting the strengths and weaknesses of the applied genetic operators. A major contribution of this thesis is the creation of GenProgJS itself, an open-source repair tool built around *Genetic Algorithm* principles, designed to generate patches for bugs in JavaScript code. Additionally, the author analyzed the patches generated for the ESLint project, covering 10 different bugs, by computing document embeddings and measuring the distances between faulty and plausible code versions. He also laid the foundation for future experiments using pretrained GPT-2 models by preparing data processing pipelines and mining a comprehensive dataset suitable for model fine-tuning. Detailed discussion can be found in Chapter 1.

In short, the author made the following contributions:

- I/1. Novel genetic operators that are suitable for the JavaScript programming language.
- I/2. Experimental results showing that GenProgJS can efficiently repair errors in 31 JavaScript programs, of which in 12 cases the generated patches are semantically similar to the human patch.
- I/3. A qualitative and quantitative analysis of the generated patches, demonstrating the strengths and weaknesses of the applied genetic operators.
- I/4. Significant contribution in creating GenProgJS, an open source tool that uses *Genetic Algorithm* to generate patches for bugs in JavaScript programs.
- I/5. Analysis of patches generated for the Eslint project on 10 different bugs, via generating document embeddings, and calculating distances between the original faulty code and the generated plausible versions.
- I/6. Laying the groundwork for pre-trained GPT-2 experiments in the form of data processing steps, and dataset mining, to provide the overall dataset on which the model can be fine-tuned.

Journal publications

- Viktor Csuvi, Dániel Horváth, Márk Lajkó, and László Vidács. "GenProgJS: A Baseline System for Test-Based Automated Repair of JavaScript Programs." *IEEE Transactions on Software Engineering* 01 (2024): 1-19.

Full papers in conference proceedings

- Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. "Exploring Plausible Patches Using Source Code Embeddings in JavaScript." In 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 11-18. IEEE, 2021.
- Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. "Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?." In International Conference on Computational Science and Its Applications, pp. 79-91. Cham: Springer International Publishing, 2022.

Representation Matters: *Deep Learning*-Based Program Repair with Effective Code Encodings

Chapter 2 focuses on the role of source code representation in *Deep Learning*-based *Automated Program Repair* (APR). This aspect of software repair is often overlooked by previous studies —only focusing on improving model architecture— although it clearly plays a significant role in determining overall model performance. To demonstrate this, experiments with multiple transformer-based models were conducted on three different representations of source code. Their performance was tested on Java and JavaScript datasets. To support researchers of this field, a multilingual dataset, *CodegrainHouse* was introduced —comprising more than 370,000 method-level bug fixes extracted from 5000+ GitHub repositories— together with a new language-agnostic methodology for extracting different levels of code granulates from source code. These contributions support future work in *Automated Program Repair* and highlight the importance of data representation to achieve more desirable outcomes.

Contributions of the thesis

In the **second thesis group**, the contributions are related to the empirical comparison of various source code representation strategies using multiple *Deep Learning* models evaluated on two different datasets with different languages, highlighting the impact of input encoding on repair performance. Furthermore, the author introduced a novel, language-agnostic method to extract different levels of granulate of code segments, and published a new dataset containing such extracted functions to demonstrate the form of datasets that could be created using the method in support of the fine-tuning of *Deep Learning* models. The method could also potentially provide an extension to the already available data mining software. Detailed discussion can be found in Chapter 2.

In short, the author made the following contributions:

- II/1. Comparison of different code representation strategies with different DL models on different datasets.
- II/2. Novel language-agnostic source code granulate extraction method.
- II/3. Novel dataset including extracted functions for fine-tuning DL models.
- II/4. Providing the dataset to fine-tune the GPT-2 pre-trained model, which supports that fine-tuning on code fragments can significantly improve model performance.

Full papers in conference proceedings

- Dániel Horváth, Viktor Csuvi, Tibor Gyimóthy, and László Vidács. "An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-Based Automated Program Repair." In 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 31-38. IEEE, 2023.
- Márk Lajkó, Dániel Horváth, Viktor Csuvi, and László Vidács. "Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?." In International Conference on Computational Science and Its Applications, pp. 79-91. Cham: Springer International Publishing, 2022.

Short papers in conference proceedings

- Dániel Horváth, and László Vidács. "Yet Another Miner Utility Unveiling a Dataset: Codegrain." In International Conference on Data Technologies and Applications. 2024.

Green Energy Through ML: Optimizing CO₂ Electrolyzer Cell Performance

Chapter 3 focuses on the electrochemical reduction of CO₂ via AEM separated electrolyzer cells, through the use of *Machine Learning* (ML) techniques. The thesis highlights the relevance of ML beyond common software engineering tasks—such as *Automated Program Repair*—underscoring its possible applications in different scientific domains. The work explores use cases in which predictive models of complex physicochemical systems can significantly help reduce experimental time and costs. This could be achieved by enabling a form of virtual experimentation through accurate digital representations of the real-world process. To demonstrate this, the

thesis presents a trained model and a parameter optimization experiment (using the trained model), where 10,000 individual combinations of parameters were tested to find superior parameter settings for optimal cell operation.

Contributions of the thesis

In the **third thesis group**, the contributions are related to the implementation of an AI-guided framework for optimizing the electrochemical reduction of AEM separated CO₂ electrolyzer cells. The design, implementation and execution of these experiments are entirely the work of the author, also focusing on parameter setting optimization, and the qualitative and quantitative assessment of model performance across different cell assemblies. He also conducted experiments on model *Sensitivity Analysis* through the use of Shapley-values, to enhance the explainability of model predictions in the context of real-world systems. Detailed discussion can be found in Chapter 3.

In short, the author made the following contributions:

- III/1. Implementation of the AI-guided framework for optimizing the CO₂ reduction of AEM separated electrolyzer cells.
- III/2. Devising and conducting experiments on parameter setting selection optimization.
- III/3. Qualitative and quantitative evaluation of the performance of the model across different cell assemblies.
- III/4. Contributing to the explainability of the real-life system via performing Shapley-value-based evaluation of model predictions.

Journal publications

- Angelika A. Samu, Dániel Horváth, Balázs Endrődi, László Vidács, and Csaba Janáky. "Neural Algorithm Aided Operation of CO₂ Electrolyzers." In ACS Energy Letters. 2025.

Unified Discussion

The three chapters presented in this dissertation collectively demonstrate the remarkable versatility and cross-domain applicability of *Machine Learning* techniques, while revealing fundamental insights about how different methodological approaches address distinct classes of optimization and prediction problems. Each contribution

represents a different facet of the modern *Machine Learning* landscape: evolutionary computation for transparent, search-based optimization; representation learning for capturing complex patterns in structured data; and neural network modeling to understand intricate physical processes.

Comparative Analysis of Methodological Approaches

The methodological diversity throughout this work highlights the strengths, while also discussing limitations of different *Machine Learning* paradigms when applied to complex real-world problems. The genetic algorithm-based approach in GenProgJS shows the power of *Genetic Algorithms* when transparency, explainability, and domain-specific customization are necessary. The explicit design of JavaScript-specific mutation operators demonstrates how traditional optimization techniques can be enhanced through careful engineering specialized for a certain domain, achieving competitive results while maintaining full interpretability of the search process. For future research, understanding the *why* behind the emergence of a particular solution is just as important as producing the correct solution itself.

In addition to this, the exploration of *Deep Learning*-based program repair reveals the critical importance of data representation in determining model performance. The finding that no single representation strategy universally outperforms others in different datasets and model architectures underscores a fundamental challenge in modern *Machine Learning*: the representation problem often determines the limit of what can be achieved, regardless of the model architecture. This insight extends beyond program repair to show how representation choices fundamentally shape the learning landscape for any *Machine Learning* application.

The chapter on CO₂ electrolyzer optimization work demonstrates yet another dimension of *Machine Learning* applicability, where neural networks excel at capturing complex, non-linear relationships in multi-parameter physical systems. The success of the *Artificial Neural Network* approach, combined with SHAP-based interpretability analysis, shows how modern *Machine Learning* can provide accurate predictions and meaningful insights into the behavior of the underlying system, bridging the gap between black-box prediction and scientific understanding.

Broader Implications for *Machine Learning* Applications

This work demonstrates that distinguishing between "traditional" and "modern" *Machine Learning* approaches may be less relevant than understanding how different methodologies complement each other. The success of *Genetic Algorithms* in GenProgJS, the importance of representation engineering in *Deep Learning*-based repair, and the effectiveness of *Neural Networks* in electrolyzer optimization suggest that

methodological diversity —rather than convergence toward any single approach— can be key to addressing the full spectrum of real-world *Machine Learning* applications.

Furthermore, cross-domain applications demonstrate that *Machine Learning*'s impact extends well beyond traditional computer science boundaries. The successful application of similar algorithmic principles to software repair and electrochemical optimization suggests that *Machine Learning* methodologies are becoming fundamental tools for scientific discovery and system optimization in various fields. This universality implies that advances in *Machine Learning* theory and practice will continue to have far-reaching impacts on scientific research and technological development.

The collective insights from this dissertation thus contribute to both the theoretical understanding of *Machine Learning* methodologies and their practical application in diverse domains, while highlighting the continued importance of methodological diversity in the addressing of the complex challenges of modern science and engineering.

Összefoglalás

Jelen disszertáció három fő témát tárgyal, melyek mindegyike gépi-tanulási módszerekhez és azok alkalmazásaihoz kapcsolódik. A gépi-tanulási módszerek jelentős szerepet játszanak számos tudományos terület fejlődésében, beleértve programok automatikus javítását vagy akár elektrokémiai cellák teljesítményének optimalizálását. Habár jelen disszertáció kevés mindezen lehetőségek és eredmények bemutatására, mégis segít rávilágítani ezen kutatások jelentőségére, és demonstrálja a mesterséges intelligencián alapuló rendszerekben rejlő képességeket, valamint azok alkalmazás területeiben rejlő potenciálokat.

Intelligensebb Folyamatok JavaScript Rendszerek Automatikus Javításához

Az 1. fejezet egy jelentős hiányosságra világít rá az *automatikus programjavítás* területén azzal, hogy a JavaScript nyelvre —a webfejlesztés során egyik leggyakrabban használt programnyelvre— fekteti a hangsúlyt. A JavaScript egy elterjedt, ám mégis jórészt figyelmen kívül hagyott nyelv. E hiányosság indokán készült a GenProgJS elnevezésű genetikusan alapuló automatikus programjavító eszköz. A szoftver kifejezetten JavaScript-hez készült operátorokat használ, hogy ezáltal is képes legyen nyelvspecifikus hibák javítására. Programunk hagyományos GenProg alapelvekre épül, de figyelembe veszi a JavaScript nyelv sajátosságait is. Kiértékelése valódi JavaScript hibák gyűjteményéből készült adatbázison került elvégzésre. A tézis ezen túlmenően a bemutatott eszközt, egyéb gépi-tanuló, valamint hibrid alapon nyugvó rendszerek kontextusába is helyezi azáltal, hogy összehasonlítja annak teljesítményét nagy nyelvi modellekkel, valamint tárgyalja a hasonlósági alapon nyugvó program-helyesség ellenőrzés lehetőségeit is. Rávilágít, hogy a hagyományos megközelítések nem feltétlenül elavultak, hanem kiegészítik a modern változataikat. A bemutatott kutatások ezáltal hozzájárulhatnak hatékonyabb programok javítására szolgáló fejlesztések kutatásában.

A tézis hozzájárulásai

Az **első téziscsoportban** a hozzájárulások a GenProgJS fejlesztésére és kiértékelésére vonatkoznak. A szoftver egy genetikus algoritmuson alapuló kódjavító eszköz, amely JavaScript projektekben előforduló hibák automatikus javítására készült. Az eszköz soha nem látott operátorokat is tartalmaz, melyek kifejezetten a JavaScript nyelv sajátosságait is tekintetbe véve készültek. Eredményeink alapján 31 JavaScript programban sikerült kijavítani a hibát, melyek közül 12 esetben a generált javítások szemantikailag is hasonlóak voltak az emberi fejlesztők által készített javításokhoz. A generált javításokról részletes elemzés történt, kiemelve az alkalmazott genetikus operátorok előnyeit és esetleges korlátait. A tézis egyik fő eredménye maga a GenProgJS eszköz, egy nyílt forráskódú programkód javító rendszer, amely genetikus algoritmus alapján —operátorok használatával— készít javításokat. Ezen felül elemeztük az ESLint projekt alá tartozó, 10 különböző hibára generált javításokat is, melyeket dokumentumbeágyazások segítségével rendeztünk hasonlósági sorrendbe, evvel következtetve helyességükre. A szerző továbbá fontos hozzájárulásokat tett a GPT-2-re épülő további kísérletekben is, ahol adatfeldolgozási módszerekkel hozta létre azon finomhangoláshoz szükséges adatbázist, melyen a modell később tanulhatott. Részletes kifejtés a 1. fejezetben található.

Folyóiratcikkek

- [1] Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. "GenProgJS: A Baseline System for Test-Based Automated Repair of JavaScript Programs." *IEEE Transactions on Software Engineering* 01 (2024): 1–19.

Teljes terjedelmű konferenciatickek

- [1] Viktor Csuvik, Dániel Horváth, Márk Lajkó, and László Vidács. "Exploring Plausible Patches Using Source Code Embeddings in JavaScript." In *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pp. 11–18. IEEE, 2021.
- [2] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. "Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?." In *International Conference on Computational Science and Its Applications*, pp. 79–91. Cham: Springer International Publishing, 2022.

A Reprezentáció Számít: Mélytanulás-alapú Programjavítás Hatékonyabb Forrás Kódolással

A 2. fejezet a forráskód reprezentáció szerepére összpontosít a mélytanulás-alapú automatikus kódjavító rendszerekben. A szoftverjavításnak ezen szemlélete azonban gyakran háttérbe szorul a kutatásokban, többnyire csak a modellarchitektúra fejlesztésére koncentrálva. Kísérleteink során azonban bemutatjuk, hogy a bemenetek megfelelő reprezentációja kulcsszerepet játszik a modell teljesítményének meghatározásában. Ennek bemutatására több transformer-alapú modell került kiértékelésre három különböző kódábrázolási formátumot használva, Java és JavaScript adathalmazokon. Későbbi kutatások támogatására bemutatásra került egy több programozási nyelvet tartalmazó adatbázis is, amely több mint 370'000 metódusból álló hibajavítást tartalmaz több mint 5'000 GitHub projektből bányászva. Emellett egy új, nyelvfüggetlen módszertan is ismertetésre került különböző kódrészletek kinyerésére. Ezek az eredmények új irányokat nyithatnak meg az automatikus kódjavítás kutatásában, és kiemelik a kódreprezentáció fontosságát.

A tézis hozzájárulásai

A **második téziscsoportban** a hozzájárulások különféle forráskód-reprezentációs stratégiák empirikus összehasonlítására irányulnak többféle mélytanuló modell segítségével, különböző nyelvű adatbázisokon. Kiemelésre kerül a tanító adatokon végbevitt kódábrázolások hatása a javítási teljesítményen. Továbbá egy új, nyelvfüggetlen módszer is bemutatásra kerül kódszegmensek válogatására, valamint közzétett egy olyan új adatbázist, amely ilyen módon kiválogatott függvényeket tartalmaz különböző mélytanuló modellek finomhangolására. Ezen felül, ez a módszer kiegészítheti a jelenlegi adatbányászó szoftvereket is. Részletes kifejtés a 2. fejezetben található.

Teljes terjedelmű konferenci cikkek

- [1] Dániel Horváth, Viktor Csuvi, Tibor Gyimóthy, and László Vidács. "An Extensive Study on Model Architecture and Program Representation in the Domain of Learning-Based Automated Program Repair." In 2023 IEEE/ACM International Workshop on Automated Program Repair (APR), pp. 31–38. IEEE, 2023.
- [2] Márk Lajkó, Dániel Horváth, Viktor Csuvi, and László Vidács. "Fine-Tuning GPT-2 to Patch Programs, Is It Worth It?." In International Conference on Computational Science and Its Applications, pp. 79–91. Cham: Springer International Publishing, 2022.

Rövid konferenciatickek

- [1] Dániel Horváth, and László Vidács. "Yet Another Miner Utility Unveiling a Dataset: Codegrain." In International Conference on Data Technologies and Applications. 2024.

ML és Zöldenergia: CO₂ Elektrolizáló Cellák AI-alapú Vezérlése és Optimalizálása

A 3. fejezet az CO₂ elektrokémiai redukciójára koncentrált anioncserélő membrán-elválasztású elektrolizátor cellák segítségével, gépi-tanuló technikák alkalmazásával. A tézisben megmutatjuk, hogy a gépi-tanuló algoritmusok nem csupán a szoftver-fejlesztési feladatokban —mint például az automatikus kódjavítás— játszhatnak szerepet, hanem más tudományterületeken is hasznosak lehetnek. Ezen fejezet olyan eseteket vizsgál, ahol a komplex fizikai-kémiai rendszerek megfelelő modelljei jelentősen csökkenthetik a kísérletekre fordított időt, és ezáltal költségeket is. Ez lehetővé teszi a folyamatok virtuális térben történő mérését, melyek becslést adhatnak a valós rendszerben zajló folyamatokról. A fejezetben vizsgálunk egy ilyen módon tanított modellt, valamint egy paraméteroptymalizálási kísérletet is, ahol 10'000 különböző paraméterkombinációt teszteltünk az optimális cellaműködés elérése érdekében.

A tézis hozzájárulásai

A **harmadik téziscsoportban** a hozzájárulások a mesterséges intelligencia által vezérelt keretrendszer megvalósítására vonatkoznak anioncserélő membrán által elválasztott CO₂ elektrolizátor cellák optimalizálásához. A kísérletek megtervezése, kivitelezése és végrehajtása teljes mértékben a szerző munkája. Különös figyelmet kapott a paraméterbeállítási optimalizáció, valamint a modell teljesítményének kiértékelése különböző cellakonfigurációkon. További kontribúció még a modell szenzitivitásának analízise és vizsgálata az ún. Shapley-value-k segítségével is, ezzel is elősegítve a modellek magyarázhatóságát a valós rendszerek kontextusában. Részletes kifejtés a 3. fejezetben található.

Rövid terjedelmű folyóiratcikkek

- [1] Angelika A. Samu, Dániel Horváth, Balázs Endrődi, László Vidács, and Csaba Janáky. "Neural Algorithm Aided Operation of CO₂ Electrolyzers." In ACS Energy Letters. 2025.

Acknowledgments

While it is practically impossible to list everyone who helped me get to this point, I would like to express my gratitude to all the people who supported me throughout my academic journey. I may not have been here as long as others and, as such, I am especially thankful to everyone who welcomed me with open arms. In this regard, I would like to emphasize that producing this dissertation would not have been possible without the help and guidance of my supervisor, László Vidács who offered a place under his supervision, András Kicsi who introduced me to him and encouraged me in taking this path of scientific journey.

I am also very grateful to my coauthors and colleagues, especially Viktor Csuvik and Márk Lajkó, as they are undoubtedly a huge part of the success of this research process. Also, cheers to Márk, without whose kind encouragement this dissertation might still be unwritten. Furthermore, I would like to add Péter Pusztai, whose guidance and mentorship was invaluable throughout the work presented in the 3rd thesis. This dissertation should also stand to acknowledge his nameless contributions to this 3 years of continuous research. Last but not least, I am grateful for the support of my wife, family, and friends who encouraged me during my time here.

Let this dissertation be proof of what we can achieve together, as I will *never* forget the people helping me to be here today.

The results presented in this dissertation were in part supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory. The national project TKP2021-NVA-09 also supported the work. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. This work is also supported by project no. RRF-2.3.1-21-2022-00009, titled National Laboratory for Renewable Energy implemented with the support provided by the Recovery and Resilience Facility of the European Union within the framework of Programme Széchenyi Plan Plus. eChemicles (also a supporter of said thesis) acknowledges funding from the European Union's Horizon Europe research and innovation programme under grant agreement no. 101099284. Sup-

ported by the KDP-2021 program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation fund.

This dissertation was refined on multiple occasions by accepting minimalistic suggestions on grammar, clarity, and use of words with the support of Writefull's language model ¹ integration with Overleaf ². This also involved accepting paraphrased versions of multiple sentences, where the original was unclear compared to the suggested one. Furthermore, Hungarian summaries are also translated/generated with the usage of LLM tools, then paraphrased by the author, where "non-human" writing style was disturbingly apparent.

¹<https://www.writefull.com/>

²https://www.overleaf.com/learn/how-to/Writefull_integration

Bibliography

- [1] Hafsa Abbade, Hassan El Fadil, Abdellah Lassioui, Abdessamad Intidam, Ahmed Hamed, Yassine El Asri, Abdelouahad Fhail, and Anwar Hasni. Deep learning-based performance modeling of hydrogen fuel cells using artificial neural networks: A comparative study of optimizers. *Processes*, 13(5):1453, 2025.
- [2] Matteo Agliuzza, Candido Fabrizio Pirri, and Adriano Sacco. A comprehensive modeling for the co2 electroreduction to co. *Journal of Physics: Energy*, 6(1):015004, 2023.
- [3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.
- [4] Shima Alinejad, Jonathan Quinson, Yao Li, Ying Kong, Sven Reichenberger, Stephan Barcikowski, Peter Broekmann, and Matthias Arenz. Optimizing the use of a gas diffusion electrode setup for co2 electrolysis imitating a zero-gap mea design. *Journal of Catalysis*, 429:115209, 2024.
- [5] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.
- [6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, jan 2018.
- [7] Kamel Alrashedy. Language Models are Better Bug Detector Through Code-Pair Classification, 2023.
- [8] Claudio Ampelli, Francesco Tavella, Daniele Giusi, Angela Mercedes Ronsisvalle, Siglinda Perathoner, and Gabriele Centi. Electrode and cell design for co2 reduction: A viewpoint. *Catalysis Today*, 421:114217, 2023.

- [9] Fatmah Yousef Assiri and James M. Bieman. Fault localization for automated program repair: effectiveness, performance, repair correctness. *Software Quality Journal*, 25(1):171–199, mar 2017.
- [10] Mohd Monis Ayyub, Attila Kormányos, Balázs Endrődi, and Csaba Janáky. Electrochemical carbon monoxide reduction at high current density: Cell configuration matters. *Chemical Engineering Journal*, 490:151698, 2024.
- [11] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. *SAFEWAPI: Web API Misuse Detector for Web Applications*. FSE 2014. Association for Computing Machinery, New York, NY, USA, 2014.
- [12] Blanca Belsa, Lu Xia, and F. Pelayo García de Arquer. Co2 electrolysis technologies: Bridging the gap toward scale-up and commercialization. *ACS Energy Letters*, 9(9):4293–4305, 2024.
- [13] Nazih Benoumechiara and Kevin Elie-Dit-Cosaque. Shapley effects for sensitivity analysis with dependent inputs: bootstrap and kriging-based algorithms. *ESAIM: Proceedings and Surveys*, 65:266–293, 2019.
- [14] Gao Leo Black Sid, Leahy Connor Wang Phil, and Biderman Stella. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, March 2021. If you use this software, please cite it using these metadata.
- [15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models Are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS’20, Red Hook, NY, USA, 2020. Curran Associates Inc.
- [16] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. Apr4vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. *Empirical Software Engineering*, 29, 12 2023.
- [17] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference*

- on *Mining Software Repositories (MSR)*, pages 464–468. Institute of Electrical and Electronics Engineers Inc., 2022.
- [18] Keith T Butler, Daniel W Davies, Hugh Cartwright, Olexandr Isayev, and Aron Walsh. Machine learning for molecular and materials science. *Nature*, 559(7715):547–555, 2018.
- [19] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. CODIT: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, apr 2022.
- [20] Gaelle Chastaing, Fabrice Gamboa, and Clémentine Prieur. Generalized sobol sensitivity indices for dependent variables: numerical methods. *Journal of statistical computation and simulation*, 85(7):1306–1333, 2015.
- [21] AU Chavez-Ramirez, R Munoz-Guerrero, V Sanchez-Huerta, Juan M Ramirez-Arredondo, R Ornelas, LG Arriaga, S Siracusano, G Brunaccini, G Napoli, V Antonucci, et al. Dynamic model of a pem electrolyzer based on artificial neural networks. *Journal of New Materials for Electrochemical Systems*, 14(2):113–119, 2011.
- [22] Liushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 637–647. IEEE, 2017.
- [23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [24] Zimin Chen, Steve James Kommrusch, and Martin Monperrus. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering*, apr 2022.
- [25] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering*, 47(01):1–1, sep 2019.
- [26] Israel Cohen, Yiteng Huang, Jingdong Chen, Jacob Benesty, Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. *Noise reduction in speech processing*, pages 1–4, 2009.

- [27] Viktor Csuvi, Dániel Horváth, Márk Lajkó, and László Vidács. Exploring plausible patches using source code embeddings in javascript. *CoRR*, abs/2103.16846, 2021.
- [28] Viktor Csuvi, Deniel Horvath, Ferenc Horvath, and Laszlo Vidacs. Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 18–25. IEEE, 2020.
- [29] Viktor Csuvi, Dániel Horváth, Márk Lajkó, and László Vidács. Genprogjs: A baseline system for test-based automated repair of javascript programs. *IEEE Transactions on Software Engineering*, 51(2):325–343, 2025.
- [30] Viktor Csuvi, Andras Kicsi, and Laszlo Vidacs. Source code level word embeddings in aiding semantic test-to-code traceability. In *Proceedings - 2019 IEEE/ACM 10th International Workshop on Software and Systems Traceability, SST 2019*, pages 29–36. Institute of Electrical and Electronics Engineers (IEEE), sep 2019.
- [31] Viktor Csuvi and László Vidács. FixJS: A Dataset of Bug-Fixing JavaScript Commits. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, MSR '22, pages 712–716, New York, NY, USA, 2022. Association for Computing Machinery.
- [32] Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. QLOSE: Program repair with quantitative objectives. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9780, pages 383–401. Springer, 2016.
- [33] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.
- [34] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pages 65–74, 2010.
- [35] Remko J. Detz, Claire J. Ferchaud, Arie J. Kalkman, Jasmin Kemper, Carlos Sánchez-Martínez, Marija Saric, and Manoj V. Shinde. Electrochemical co2 conversion technologies: state-of-the-art and future perspectives. *Sustainable Energy Fuels*, 7:5445–5472, 2023.
- [36] difflib. <https://docs.python.org/3.11/library/difflib.html>, 2023.

- [37] Elizabeth Dinella, Hanjun Dai, Google Brain, Ziyang Li, Mayur Naik, Le Song, Georgia Tech, and Ke Wang. Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. In *Iclr 2020*, pages 1–17, 2020.
- [38] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. FIRA: fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 970–981, New York, NY, USA, 2022. Association for Computing Machinery.
- [39] Dawn Drain, Chen Wu, Alexey Svyatkovskiy, and Neel Sundaresan. Generating bug-fixes using pretrained transformers. *MAPS 2021 - Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming, co-located with PLDI 2021*, pages 1–8, jun 2021.
- [40] Harry Dunne, Weiming Liu, Mohammad Reza Ghaani, Kim McKelvey, and Stephen Dooley. Sensitivity analysis of one-dimensional multiphysics simulation of co2 electrolysis cell. *The Journal of Physical Chemistry C*, 128(27):11131–11144, 2024.
- [41] T. Durieux, Y. Hamadi, and M. Monperrus. Fully Automated HTML and Javascript Rewriting for Constructing a Self-Healing Web Proxy. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (IS-SRE)*, pages 1–12, 2018.
- [42] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*, page 302–313, New York, NY, USA, 2019. Association for Computing Machinery.
- [43] Jonathan P Edwards, Théo Alerte, Colin P O'Brien, Christine M Gabardo, Shijie Liu, Joshua Wicks, Adriana Gaona, Jehad Abed, Yurou Celine Xiao, Daniel Young, et al. Pilot-scale co2 electrolysis enables a semi-empirical electrolyzer model. *ACS Energy Letters*, 8(6):2576–2584, 2023.
- [44] Ali El Bilali, Taleb Abdeslam, Nafii Ayoub, Houda Lamane, Mohamed Abdellah Ezzaouini, and Ahmed Elbeltagi. An interpretable machine learning approach based on dnn, svr, extra tree, and xgboost models for predicting daily pan evaporation. *Journal of Environmental Management*, 327:116890, 2023.

- [45] Balázs Endrődi, A Samu, E Kecsenovity, T Halmágyi, D Sebők, and Cs Janáky. Operando cathode activation with alkali metal cations for high current density operation of water-fed zero-gap carbon dioxide electrolyzers. *Nature energy*, 6(4):439–448, 2021.
- [46] B. Endrődi, G. Bencsik, F. Darvas, R. Jones, K. Rajeshwar, and C. Janáky. Continuous-flow electroreduction of carbon dioxide. *Progress in Energy and Combustion Science*, 62:133–154, 2017.
- [47] Esprima official website. <https://esprima.org>, 2024.
- [48] Amin Milani Fard and Ali Mesbah. JavaScript: The (Un)Covered Parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 230–240. IEEE, 2017.
- [49] Asger Feldthaus, Max Schafer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings - International Conference on Software Engineering*, pages 752–761, 2013.
- [50] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics.
- [51] Alcides Fonseca and Máximo Oliveira. Figra: Evaluating a larger search space for cardumen in automatic program repair. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 24–30, 2022.
- [52] Nikola Franić, Ivan Pivac, and Frano Barbir. A review of machine learning applications in hydrogen electrochemical devices. *International journal of hydrogen energy*, 102:523–544, 2025.
- [53] L. Gazzola, D. Micucci, and L. Mariani. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, jan 2019.
- [54] Ali Ghanbari. Revisiting Object Similarity-based Patch Ranking in Automated Program Repair: An Extensive Study. *Proceedings - International Workshop on Automated Program Repair, APR 2022*, pages 16–23, 2022.
- [55] Octoverse: The state of open source and rise of AI in 2023. <https://octoverse.github.com>, 2023.

- [56] The 2024 state of the octoverse. <https://github.blog/news-insights/octoverse/octoverse-2024/>, 2024.
- [57] GitHub. <https://github.com/>, 2025.
- [58] GitHub REST API Official Website. <https://docs.github.com/en/rest>, 2023.
- [59] Rate limits of GitHub’s REST API. <https://docs.github.com/en/rest/overview/rate-limits-for-the-rest-api?apiVersion=2022-11-28>, 2023.
- [60] GitPython. <https://github.com/gitpython-developers/GitPython>, 2023.
- [61] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.
- [62] Maxwell Goldman, Aditya Prajapati, Eric Duoss, Sarah Baker, and Christopher Hahn. Bridging fundamental science and applied science to accelerate co2 electrolyzer scale-up. *Current Opinion in Electrochemistry*, 39:101248, 2023.
- [63] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019.
- [64] Ying Guo, Xinru He, Yuming Su, Yiheng Dai, Mingcan Xie, Shuangli Yang, Jiawei Chen, Kun Wang, Da Zhou, and Cheng Wang. Machine-learning-guided discovery and optimization of additives in preparing cu catalysts for co2 reduction. *Journal of the American Chemical Society*, 143(15):5755–5762, 2021.
- [65] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI’17, page 1345–1351. AAAI Press, 2017.
- [66] Peter Gyimesi, Bela Vancsics, Andrea Stocco, Davood Mazinanian, Arpad Beszedes, Rudolf Ferenc, and Ali Mesbah. BugsJS: a Benchmark of JavaScript Bugs. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation, ICST 2019*, pages 90–101, apr 2019.
- [67] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 144–156, 2016.
- [68] Carol Hanna, Aymeric Blot, and Justyna Petke. Reinforcement learning for mutation operator selection in automated program repair, 2023.

- [69] Ye He, Zimin Chen, and Claire Le Goues. PreciseBugCollector: Extensible, Executable and Precise Bug-Fix Collection: Solution for Challenge 8: Automating Precise Data Collection for Code Snippets with Bugs, Fixes, Locations, and Types. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1899–1910, 2023.
- [70] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.
- [71] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. An empirical analysis of compute-optimal large language model training. *Advances in neural information processing systems*, 35:30016–30030, 2022.
- [72] Lucas Hoof, Kevinjeorgios Pellumbi, Stephan Heuser, Daniel Siegmund, Kaijunge Puring, and Ulf-Peter Apfel. Water management as a key parameter for scaling up a co2 electrolyzer. *Chemie Ingenieur Technik*, 95(5):668–674, 2023.
- [73] Dániel Horváth and László Vidács. Yet another miner utility unveiling a dataset: Codegrain. In *International Conference on Data Technologies and Applications*, 2024.
- [74] Dániel Horváth, Viktor Csuvi, Tibor Gyimóthy, and László Vidács. An extensive study on model architecture and program representation in the domain of learning-based automated program repair. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 31–38, 2023.
- [75] Erhai Hu, Chuntai Liu, Wei Zhang, and Qingyu Yan. Machine learning assisted understanding and discovery of co2 reduction reaction electrocatalyst. *The Journal of Physical Chemistry C*, 127(2):882–893, 2023.
- [76] Yaojie Hu, Xingjian Shi, Qiang Zhou, and Lee Pike. Fix bugs with transformer through a neural-symbolic edit grammar, 04 2022.
- [77] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 12–23, New York, NY, USA, may 2018. Association for Computing Machinery (ACM).
- [78] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

- [79] Bertrand Iooss and Clémentine Prieur. Shapley effects for sensitivity analysis with correlated inputs: comparisons with sobol'indices, numerical estimation and applications. *International Journal for Uncertainty Quantification*, 9(5), 2019.
- [80] Kalervo Järvelin and Jaana Kekäläinen. Ir evaluation methods for retrieving highly relevant documents. In *ACM SIGIR Forum*, volume 51, pages 243–250. ACM New York, NY, USA, 2017.
- [81] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remediating the eval that men do. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis, ISSTA 2012*, page 34–44, New York, NY, USA, 2012. Association for Computing Machinery.
- [82] Sung Eun Jerng, Yang Jeong Park, and Ju Li. Machine learning for co2 capture and conversion: A review. *Energy and AI*, page 100361, 2024.
- [83] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, 18:298–309, 2018.
- [84] Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE '21*, page 1161–1173. IEEE Press, 2021.
- [85] et. al. John Jumper. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.
- [86] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 437–440. Association for Computing Machinery, Inc, jul 2014.
- [87] Rafael-Michael Karampatsis and Charles Sutton. How Often Do Single-Statement Bugs Occur?: The ManySStuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20*, page 573–577, New York, NY, USA, may 2020. Association for Computing Machinery.
- [88] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, 48(7):2658–2679, 2022.

- [89] Besma Khairreddine, Matias Martinez, and Ali Mili. Program Repair at Arbitrary Fault Depth. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 465–472. IEEE, apr 2019.
- [90] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings - International Conference on Software Engineering*, pages 802–811. IEEE, may 2013.
- [91] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 150–162. IEEE Computer Society, may 2021.
- [92] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, may 2020.
- [93] Márk Lajkó, Dániel Horváth, Viktor Csuvik, and László Vidács. Fine-tuning gpt-2 to patch programs, is it worth it? In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, and Chiara Garau, editors, *Computational Science and Its Applications – ICCSA 2022 Workshops*, pages 79–91, Cham, 2022. Springer International Publishing.
- [94] Márk Lajkó, Viktor Csuvik, and László Vidács. Towards javascript program repair with generative pre-trained transformer (gpt-2). In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 61–68, 2022.
- [95] Xuan Bach D. Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, mar 2016.
- [96] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering*, 23(5):3007–3033, oct 2018.
- [97] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings - International Conference on Software Engineering*, pages 3–13. IEEE, jun 2012.

- [98] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and Intro-Class Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, dec 2015.
- [99] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, January 2012.
- [100] Grazia Leonzio, Anna Hankin, and Nilay Shah. Co2 electrochemical reduction: A state-of-the-art review with economic and environmental analyses. *Chemical Engineering Research and Design*, 208:934–955, 2024.
- [101] Omer Levy and Yoav Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the eighteenth conference on computational natural language learning*, pages 171–180, 2014.
- [102] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, jan 2021.
- [103] Marco Löffelholz, Jens Osiewacz, Alexander Lüken, Karen Perrey, Andreas Bulan, and Thomas Turek. Modeling electrochemical co2 reduction at silver gas diffusion electrodes using a tffa approach. *Chemical Engineering Journal*, 435:134920, 2022.
- [104] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, pages 298–312, 2016.
- [105] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.
- [106] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *undefined*, 2021.
- [107] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

- [108] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair. *arXiv preprint arXiv:1906.08691*, 2019.
- [109] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 20:101–114, 2020.
- [110] Sonal Mahajan, Abdulmajeed Alameer, Phil McMinn, and William G. J. Halfond. Automated Repair of Internationalization Presentation Failures in Web Pages Using Style Similarity Clustering and Search-Based Techniques. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*, pages 215–226. IEEE Computer Society, 2018.
- [111] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, aug 2017.
- [112] Matias Martinez and Martin Monperrus. ASTOR: A program repair library for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444, jul 2016.
- [113] Matias Martinez and Martin Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of Astor. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11036 LNCS:65–86, dec 2018.
- [114] Matias Martinez and Martin Monperrus. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, feb 2019.
- [115] Richard I Masel, Zengcai Liu, Hongzhou Yang, Jerry J Kaczur, Daniel Carrillo, Shaoxuan Ren, Danielle Salvatore, and Curtis P Berlinguette. An industrial perspective on catalysts for low-temperature co2 electrolysis. *Nature nanotechnology*, 16(2):118–128, 2021.
- [116] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the

- usage of text-to-text transfer transformer to support code-related tasks. *Proceedings - International Conference on Software Engineering*, pages 336–347, may 2021.
- [117] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458. IEEE, may 2015.
- [118] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.
- [119] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space, jan 2013.
- [120] Amira Mohamed, Hatem Ibrahim, Rui Yang, and Kibum Kim. Optimization of proton exchange membrane electrolyzer cell design using machine learning. *Energies*, 15(18):6657, 2022.
- [121] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 234–242, New York, New York, USA, 2014. ACM Press.
- [122] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018.
- [123] Martin Monperrus. *The living review on automated program repair*. PhD thesis, HAL Archives Ouvertes, 2018.
- [124] Asger B Moss, Sahil Garg, Marta Mirolo, Carlos A Giron Rodriguez, Roosa Ilvonen, Ib Chorkendorff, Jakub Drnec, and Brian Seger. In operando investigations of oscillatory water and carbonate effects in mea-based co2 electrolysis devices. *Joule*, 7(2):350–365, 2023.
- [125] Marjane Namavar, Noor Nashid, and Ali Mesbah. A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair. *Empirical Software Engineering*, 27(7), dec 2022.
- [126] Giuseppe Napoli, Marco Ferraro, Francesco Sergi, Giovanni Brunaccini, and Vincenzo Antonucci. Data driven models for a pem fuel cell stack performance prediction. *International journal of hydrogen energy*, 38(26):11628–11638, 2013.

- [127] Nexa® 1200 Technical Data Sheet | Fuel Cell System. https://www.i-en.eu/uploads/tx_etim/Datenblatt_Nexa1200_EN_1109.pdf.
- [128] Yasunobu Nohara, Koutarou Matsumoto, Hidehisa Soejima, and Naoki Nakashima. Explanation of machine learning models using shapley additive explanation and application for real data in hospital. *Computer Methods and Programs in Biomedicine*, 214:106584, 2022.
- [129] Frolin Ocariza, Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An empirical study of client-side JavaScript bugs. In *International Symposium on Empirical Software Engineering and Measurement*, pages 55–64, 2013.
- [130] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Vejovis: suggesting fixes for JavaScript faults. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 837–847, New York, New York, USA, 2014. ACM Press.
- [131] Frolin S. Ocariza Jr., Karthik Pattabiraman, and Benjamin Zorn. JavaScript Errors in the Wild: An Empirical Study. In *Proceedings of the 2011 IEEE 22nd International Symposium on Software Reliability Engineering, ISSRE '11*, page 100–109, USA, 2011. IEEE Computer Society.
- [132] Art B Owen. Sobol’indices and shapley value. *SIAM/ASA Journal on Uncertainty Quantification*, 2(1):245–251, 2014.
- [133] Guillermo Owen. Values of games with a priori unions. In *Mathematical economics and game theory: Essays in honor of Oskar Morgenstern*, pages 76–88. Springer, 1977.
- [134] Safiye Nur Ozdemir and Oguzhan Pektezel. Performance prediction of an experimental pem electrolyzer using machine learning algorithms. *Fuel*, 378:132853, 2024.
- [135] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, jun 2009.
- [136] Long Phan, Hieu Tran, Daniel Le, Hieu Nguyen, James Annibal, Alec Peltekian, and Yanfang Ye. CoTexT: Multi-task learning with code-text transformer. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 40–47, Online, August 2021. Association for Computational Linguistics.
- [137] Elmar Plischke, Giovanni Rabitti, and Emanuele Borgonovo. Computing shapley effects for sensitivity analysis. *SIAM/ASA Journal on Uncertainty Quantification*, 9(4):1411–1437, 2021.

- [138] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*, May 2019.
- [139] Michael Pradel, Parker Schuh, and Koushik Sen. TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324. IEEE Computer Society, 2015.
- [140] Julian Aron Prenner and Romain Robbes. Runbugrun - an executable dataset for automated program repair. *ArXiv*, abs/2304.01102, 2023.
- [141] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, ISSTA 2015, pages 24–36, New York, New York, USA, 2015. ACM Press.
- [142] Maximilian Quentmeier, Bernhard Schmid, Hermann Tempel, and Rüdiger-A Eichel. Modular co₂-to-co electrolysis short-stack design– impact of temperature gradients and insights into position-dependent cell behavior. *ACS Sustainable Chemistry & Engineering*, 12(9):3876–3885, 2024.
- [143] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.
- [144] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.
- [145] Andrés Raya-Imbernón, Angelika A Samu, Stefan Barwe, Giuseppe Cusati, Tamás Földi, Balázs M Hepp, and Csaba Janáky. Renewable syngas generation via low-temperature electrolysis: opportunities and challenges. *ACS Energy Letters*, 9(1):288–297, 2023.
- [146] Derek Richard, Joonbaek Jang, Berkay Çıtmacı, Junwei Luo, Vito Canuso, Prakashan Korambath, Olivia Morales-Leslie, James F Davis, Haresh Malkani, Panagiotis D Christofides, et al. Smart manufacturing inspired approach to research, development, and scale-up of electrified chemical manufacturing systems. *iScience*, 26(6), 2023.

- [147] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, New York, New York, USA, 2010. ACM Press.
- [148] Diego Rodríguez-Baquero and Mario Linares-Vásquez. Mutode: Generic JavaScript and node.js mutation testing tool. In *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 372–375, New York, New York, USA, jul 2018. Association for Computing Machinery, Inc.
- [149] Nuno Saavedra, André Silva, and Martin Monperrus. Gitbug-actions: Building reproducible bug-fix benchmarks with github actions. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 1–5, 2024.
- [150] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. Elixir: Effective object-oriented program repair. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659. Institute of Electrical and Electronics Engineers Inc., nov 2017.
- [151] Angelika A. Samu, Dániel Horváth, Balázs Endrődi, László Vidács, and Csaba Janáky. Neural algorithm aided operation of co2 electrolyzers. *ACS Energy Letters*, 10:3845–3850, 2025.
- [152] Angelika Anita Samu, Imre Szenti, Ákos Kukovecz, Balázs Endrődi, and Csaba Janáky. Systematic screening of gas diffusion layers for high performance co2 electrolysis. *Communications Chemistry*, 6(1):41, 2023.
- [153] Benjamin Sanchez-Lengeling and Alán Aspuru-Guzik. Inverse molecular design using machine learning: Generative models for matter engineering. *Science*, 361(6400):360–365, 2018.
- [154] M. Selakovic and M. Pradel. Poster: Automatically Fixing Real-World JavaScript Performance Bugs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 811–812, 2015.
- [155] Daya Guo Shuai Lu, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

- [156] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 532–543, New York, New York, USA, 2015. ACM Press.
- [157] Ilya M Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and computers in simulation*, 55(1-3):271–280, 2001.
- [158] Eunhye Song, Barry L Nelson, and Jeremy Staum. Shapley effects for global sensitivity analysis: Theory and computation. *SIAM/ASA Journal on Uncertainty Quantification*, 4(1):1060–1083, 2016.
- [159] Stack overflow developer survey results 2024. <https://survey.stackoverflow.co/2024/technology#2-programming-scripting-and-markup-languages>, 2024.
- [160] Ifan E L Stephens, Karen Chan, Alexander Bagger, Shannon W Boettcher, Julien Bonin, Etienne Boutin, Aya K Buckley, Raffaella Buonsanti, Etosha R Cave, Xiaoxia Chang, See Wee Chee, Alisson H M da Silva, Phil de Luna, Oliver Einsle, Balázs Endrődi, Maria Escudero-Escribano, Jorge V Ferreira de Araujo, Marta C Figueiredo, Christopher Hahn, Kentaro U Hansen, Sophia Haussener, Sara Hunegnaw, Ziyang Huo, Yun Jeong Hwang, Csaba Janáky, Buddhinie S Jayathilake, Feng Jiao, Zarko P Jovanov, Parisa Karimi, Marc T M Koper, Kendra P Kuhl, Woong Hee Lee, Zhiqin Liang, Xuan Liu, Sichao Ma, Ming Ma, Hyung-Suk Oh, Marc Robert, Beatriz Roldan Cuenya, Jan Rossmesl, Claudie Roy, Mary P Ryan, Edward H Sargent, Paula Sebastián-Pascual, Brian Seger, Ludmilla Steier, Peter Strasser, Ana Sofia Varela, Rafaël E Vos, Xue Wang, Bingjun Xu, Hossein Yadegari, and Yuxiang Zhou. 2022 roadmap on low temperature electrochemical co2 reduction. *Journal of Physics: Energy*, 4(4):042003, oct 2022.
- [161] Siddhartha Subramanian, Joost Middelkoop, and Thomas Burdyny. Spatial reactant distribution in co 2 electrolysis: balancing co 2 utilization and faradaic efficiency. *Sustainable Energy & Fuels*, 5(23):6040–6048, 2021.
- [162] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182, 2017.
- [163] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th*

- ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, pages 727–738, New York, New York, USA, 2016. ACM Press.
- [164] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 981–992, 2020.
- [165] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.
- [166] Tree-sitter. <https://tree-sitter.github.io/tree-sitter/>, 2023.
- [167] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 25–36. IEEE Press, 2019.
- [168] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4), sep 2019.
- [169] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017-December:5999–6009, 2017.
- [170] Bruno Vuillod, Marco Montemurro, Enrico Panettieri, and Ludovic Hallo. A comparison between sobol’s indices and shapley’s effect for global sensitivity analysis of systems with independent input variables. *Reliability Engineering & System Safety*, 234:109177, 2023.
- [171] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. How Different Is It between Machine-Generated and Developer-Provided Patches? : An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques. In *International Symposium on Empirical Software Engineering and Measurement*, volume 2019-Sept. IEEE Computer Society, sep 2019.

- [172] Shangwen Wang, Ming Wen, Bo Lin, Xiaoguang Mao, Hongjun Wu, Deqing Zou, Hai Jin, and Yihao Qin. Automated Patch Correctness Assessment: How Far are We? *ASE 2020 - Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1166–1178, 2020.
- [173] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.
- [174] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 364–374, USA, 2009. IEEE Computer Society.
- [175] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In Xinyu Wang, David Lo, and Emad Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 479–490. IEEE, 2019.
- [176] Martin White, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, Massimiliano Penta, Christopher Association for Computing Machinery-Digital Library, Mario ACM Special Interest Group on Software Engineering., and Denys Poshyvanyk. Toward Deep Learning Software Repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*, pages 334–345. IEEE Press, 2015.
- [177] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging Studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020*, page 1556–1560, New York, NY, USA, 2020. Association for Computing Machinery.
- [178] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz,

- Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.
- [179] Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670, 2017.
- [180] Lei Xing, Hai Jiang, Xingjian Tian, Huajie Yin, Weidong Shi, Eileen Yu, Valerie J Pinfield, and Jin Xuan. Combining machine learning with multi-physics modelling for multi-objective optimisation and techno-economic analysis of electrochemical co2 reduction process. *Carbon Capture Science & Technology*, 9:100138, 2023.
- [181] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings - International Conference on Software Engineering*, pages 789–799. IEEE Computer Society, may 2018.
- [182] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering, ICSE '17*, page 416–426. IEEE Press, 2017.
- [183] Qiucheng Xu, Björt Óladóttir Joensen, Nishithan C Kani, Andrea Sartori, Terry Wilson, John R Varcoe, Luca Riillo, Anna Ramunni, Jakub Drnec, Ib Chorkendorff, et al. Electrolyte effects in membrane-electrode-assembly co electrolysis. *Angewandte Chemie International Edition*, page e202501505, 2025.
- [184] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *IJCAI*, 2019.
- [185] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, 2017.
- [186] Changlan Yang, Xuefeng Guan, Qingyang Xu, Weiran Xing, Xiaoyu Chen, Jinguo Chen, and Peng Jia. How can shap (shapley additive explanations) interpretations improve deep learning based urban cellular automata model? *Computers, Environment and Urban Systems*, 111:102133, 2024.

- [187] Deheng Yang, Yan Lei, Xiaoguang Mao, Yuhua Qi, and Xin Yi. Seeing the whole elephant: Systematically understanding and uncovering evaluation biases in automated program repair. *ACM Transactions on Software Engineering and Methodology*, 32, 09 2022.
- [188] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 831–841, 2017.
- [189] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. *IBF 2019 - 2019 IEEE 1st International Workshop on Intelligent Bug Fixing*, pages 1–10, may 2019.
- [190] Li Yi, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings - International Conference on Software Engineering*, pages 602–614. IEEE Computer Society, jun 2020.
- [191] Myle Ott Yinhan Liu, Jingfei Du Naman Goyal, Danqi Chen Mandar Joshi, Mike Lewis Omer Levy, and Veselin Stoyanov Luke Zettlemoyer. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.
- [192] Bing Yu, Hua Qi, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, and Jianjun Zhao. Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment. *IEEE Transactions on Reliability*, 71(4):1401–1416, 2022.
- [193] Long Zhou Yu Tang, Shujie Liu Ambrosio Blanco, Ming Zhou Furu Wei, and Muyun Yang. Grammar-Based Patches Generation for Automated Program Repair. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 1300–1305, 2021.
- [194] Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin. Circle: Continual repair across programming languages, 2022.
- [195] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, 2018.

- [196] Shuang Zong, Aibing Chen, Marek Wiśniewski, Lebohang Macheli, Linda L Jewell, Diane Hildebrandt, and Xinying Liu. Effect of temperature and pressure on electrochemical co₂ reduction: A mini review. *Carbon Capture Science & Technology*, 8:100133, 2023.