

Machine Learning Techniques in Software Repair and Environmental Optimization

Summary of the PhD Thesis Points

Dániel Horváth

Supervisor: László Vidács, PhD

- [1] Smarter Repair Pipelines for JavaScript APR
- [2] Representation Matters: Deep Learning-Based
Program Repair with Effective Code Encodings
- [3] Green Energy Through ML: Optimizing CO₂
Electrolyzer Cell Performance

Doctoral School of Computer Science
Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged

2026

Introduction

Machine Learning (ML) is a still emerging and expanding research field, having a wide range of scientific and even 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 life, and thus making it a highly researched topic to this day.

Among its most impactful research field there is *Automated Program Repair* (APR), where ML techniques being used to automatically identify and correct 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 are –originally designed for *Natural Language Processing* (NLP) tasks– the transformer based architectures, which have shown promising results in learning patterns of buggy and fixed code versions.

In parallel, *Genetic Algorithms* (GAs) like GenProg or Kali, have also shown promise in repairing certain kinds of software bugs. They work by mutating the buggy source codes via so-called *operators*, which act like code transform recipes. Every operator takes the code and mutates it slightly (e.g., replaces a variable name with another randomly) by transforming it in a specific way. This in turn can produce a slightly modified code, which then can 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 like DeepRepair 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 have helped researchers leap many years into the future by enabling 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 byproducts, like 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.

1. Smarter Repair Pipelines for JavaScript APR

Automated Program Repair (APR) has made notable progress since its earliest stages, but has mostly ignored the language that basically powers the web, JavaScript. Although one can find many APR tools for languages like Java and C, there are not many general-purpose tools for JavaScript that use test suites to fix bugs. This work aims to patch this, by introducing GenProgJS, a new APR tool for JavaScript that uses a genetic search approach. This thesis presents GenProgJS as a foundational tool that the JavaScript APR community can build on. It is designed to be clear and traceable. Also, it is easily interpretable, which is in direct contrast to the "black-box" nature of many learning-based models. As a benchmark, we used the BugsJS dataset, which is a collection of real reproducible bugs from well-known open-source JavaScript projects.

GenProgJS is built on three main parts: a genetic algorithm engine, a set of tools for changing code, and a testing framework that works with the BugsJS dataset. The tool uses a standard genetic algorithm, much like the one described in the original paper. It starts by creating a series of potential solutions, each one being a slightly different version of the buggy program, represented as a *Abstract Syntax Tree* (AST). The algorithm then runs for a set number of cycles (or "generations"), doing the following in each loop:

1. **Fitness Check:** Each version of the program is tested to see how well it performs. The fitness function is used to reward the algorithm for generating code with passing tests and penalizes it for failing test cases. The main goal is to guide the population to a state in which fewer and fewer tests are failing.
2. **Offspring Creation:** New potential solutions, called offspring, are made in two ways:
 - **Mutation:** We apply a set of mutation operators to the existing programs based on certain probabilities. This creates slightly modified versions of the current candidates.

- **Crossover:** We pick two parent solutions and mix their ASTs to create a new offspring that has a bit of both.
3. **Selection:** From all the new offspring, the ones that will make up the next generation are chosen. The algorithm is configured to lean toward those with better fitness scores, but some variety and exploration is also allowed.

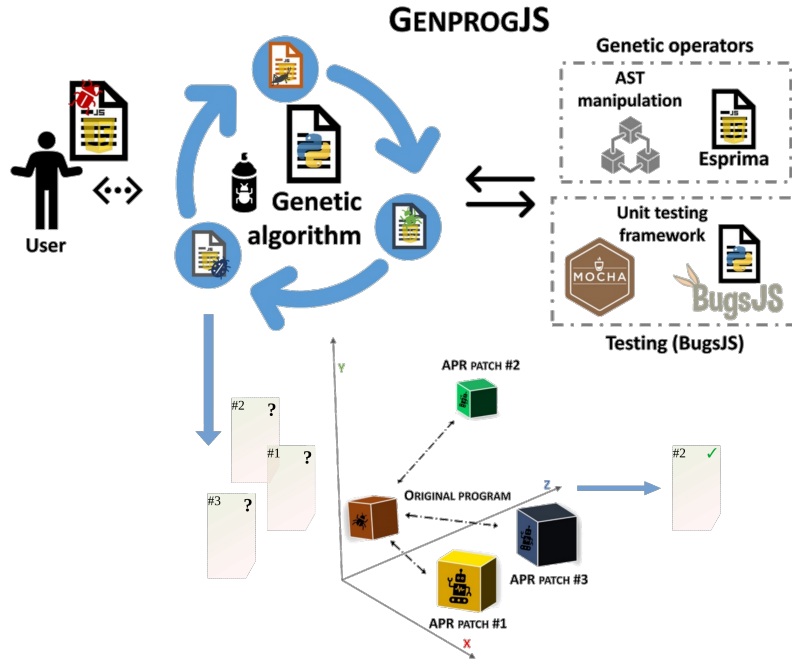


Figure 1.1: High-level overview of GenProgJS components.

If a program version passes all test cases, we consider it a plausible patch. An also notable advantage of the algorithm is that it is designed to log the whole process of patch generation. This means that one can follow the flow of applied operators and see how the patch mutated along the process, making the evolution of the patch transparent so that one can see exactly which changes led to a successful repair. This also brings us to a big part of what makes GenProgJS special. The answer also lies in its mutation operators, which were built specifically for JavaScript’s dynamic and script-like nature. Certain ideas were borrowed from C or Java-based tools such as Astor, but many were created to address the special behaviors of the language. These include:

- **Dealing with null and undefined:** The NullCheck operator adds defensive code (like `if (a && a.loc === 0)`) that uses JavaScript’s “truthy” and “falsy” values to stop errors caused by undefined variables.

- **Handling Async Code:** The `AwaitInserter` and `AsyncFunction` operators were made to fix bugs in asynchronous code, which is a huge part of modern JavaScript.
- **Working with Dynamic Types:** Operators like `VarChanger` can swap variables around freely without worrying about strict types, just as a human JavaScript developer might.

Due to the fact that these operators are designed for the language, GenProgJS can search for fixes in a way that other tools could have missed before, and can introduce potential patches that make sense in a real-world JavaScript project. In this sense, GenProgJS managed to fix 31 bugs in six different projects, of which 12 were semantically similar (or identical) to the developer’s patch. This also takes us to another theme, the thesis touches upon;

How to tell if a generated patch is actually correct? *Genetic Algorithms* can produce many plausible –meaning that those patches do indeed pass all the test cases– although not necessarily correct in a sense that they fix the actual bug. Picking the right one from several candidates is still a widely researched topic today. The work aims to tackle this problem using code similarity measured with Doc2Vec embeddings. The idea is that nicely behaving fixes are usually closer (textually, in a high-dimensional vector space, called embedding) to the buggy code than their over-fitted counterparts. By creating an open-source tool and trying new ways to validate patches, this work provides new ways and a possible baseline for JavaScript APR.

To address the problem of sorting through all the plausible patches, we proposed a similarity-based classification system. For our experiments, we limited ourselves to the ESLint project as this project contained the most single-line errors. The general steps of the algorithm are as follows.

1. We took a code snippet (the changed line plus a little context) from each plausible patch.
2. We trained a Doc2Vec model on these snippets to turn each piece of code into a dense vector (embedding).
3. We then calculated the cosine similarity between the vector for each patch and the vector for the original buggy code.
4. Finally, we ranked the patches. Our theory was that patches most similar to the original code were more likely to be the right fix.

Our experiment with ranking patches by similarity gave us mixed results. Although the developer’s fix was often textually similar to the original code, our Doc2Vec similarity metric was not always a great predictor of correctness. This tells

us that just looking at text embeddings might not be enough to understand the deeper meaning of the code, and we probably need more advanced models to do a better job of validating patches.

We also compared GenProgJS to a fine-tuned GPT-2 model. The results showed that there is a trade-off between the two methods. GPT-2 only fixed 3 bugs, but it was 100 % precise, meaning that every patch it made was correct and identical to the developer’s fix. This seems to indicate that LLMs are great at identifying common patterns, but might not be able to handle bugs that are more complex or unusual in sense that the model did not generalize to an extent to recognize such patterns. GenProgJS fixed a much wider range of bugs (12 correct fixes), showing that genetic search can, in fact, explore a larger space of possible solutions. This suggests that the future could lie in hybrid systems that combine the wide-reaching search of genetic algorithms with the pinpoint accuracy of LLMs.

In summary, we tested GenProgJS on the BugsJS dataset, which includes 352 bugs from 10 different open-source projects. After five separate runs, the tool came up with plausible patches for 31 unique bugs on six of these projects. We found that GenProgJS had produced a genuinely correct patch for 12 of those 31 bugs. In five of those cases, the patch it generated was a perfect match for the one a human developer had written. Compared to GPT-2, GenProgJS clearly outperforms it by simply examining the number of patches generated. However, GPT-2 only generated perfect replicas of the developer fixes if the patch was plausible, which means that it did not generate false positives. We also found that JavaScript-specific operators, especially `NullCheck` and `VarChanger`, played a role in many of the correct fixes, which emphasize the fact that language-specific operators can play a big role in producing plausible and also correct patches.

In the end, this work successfully introduces *GenProgJS*, the first general-purpose, test-suite-based APR tool designed for JavaScript. By taking the genetic programming model and equipping it with a set of language-specific operators, it provides a foundation for future work in the field of JS APR. The comparison with GPT-2 also brings into focus an important aspect of modern APR. It shows that more traditional tools, such as GenProgJS, still have real use in modern *Automated Program Repair*, with the trade-off between exploratory power of *Genetic Algorithms*, and the precision of *Deep Learning*-based models. The results strongly point toward a promising future where hybrid approaches bring these two worlds together.

1.1 Contributions

In the first thesis group, the contributions are related to the development and evaluation of GenProgJS, a *Genetic Algorithm*-based APR designed to repair bugs potentially present in JavaScript projects, featuring novel genetic operators specifically 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 the ones 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 GA 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 pre-trained GPT-2 models by preparing data processing pipelines and mining a comprehensive dataset suitable for model fine-tuning.

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

Many studies in *Automated Program Repair* (APR) have focused on different model setups and how to train them, but the importance of how source code is represented did not get enough attention. This work emphasizes that the way source code is encoded is an important factor of *Deep Learning*-based APR models' performance. The core motivation of this thesis is to empirically investigate how different representations affect repair outcomes across various transformer-based models and programming languages (Java and JavaScript). To promote the research of the field the published works based on this thesis also include the method used for code extraction, and a publicly available dataset on Zenodo. The code extraction utility is designed in a way to be language-agnostic (supporting most, if not all popular languages), so that it could be easily implemented as a plugin to already existing mining software. The thesis also introduces a large, multi-lingual dataset mined from over 5,000 GitHub repositories. It contains over 370,000 bug-fixing commits across five popular languages (C, C++, Java, JavaScript, Python), offering granular, method-level code snippets, that models can be fine-tuned on. Unlike other datasets, it provides not just the buggy and fixed code but also the commit messages, opening possibilities for models that generate both a fix and its explanation.

To demonstrate the importance of code representation, this work presents the training of different transformer-based models on different code representations and the evaluation of their performance on multiple datasets in multiple languages. The models used in this work are as follows:

- **T5 and CodeT5:** Standard text-to-text transformer architectures, used both with and without pre-trained weights.
- **RoBERTa, CodeBERT, and GPT-Neo:** A composite architecture designed to handle structured data. RoBERTa encodes graph information (from the AST), CodeBERT encodes the source code text, and their outputs are concatenated and fed to a GPT-Neo decoder to generate the fix.

One of the key purposes is to highlight the fact that different code representations can play a significant role in how well the model is performing. This work uses three different encoding strategies for demonstration.

1. **Text-to-Text:** The baseline approach where both the buggy (input) and fixed (output) code are treated as simple sequences of tokens. The model learns to transform the entire input sequence into the output sequence.
2. **Command Sequence (cmdseq):** The input is the buggy code as text, but the model is trained to generate a sequence of edit commands ([INSERT], [DELETE]) to transform the buggy code into the fixed version. This focuses the model’s learning on the changes rather than redundant, unchanged code.
3. **AST + Text:** An enriched input representation that combines the raw text of the source code with a flattened representation of its AST. This provides the model with both lexical and structural information, theoretically offering more representative power to the encoder. The target remains the fixed code as simple text representation.

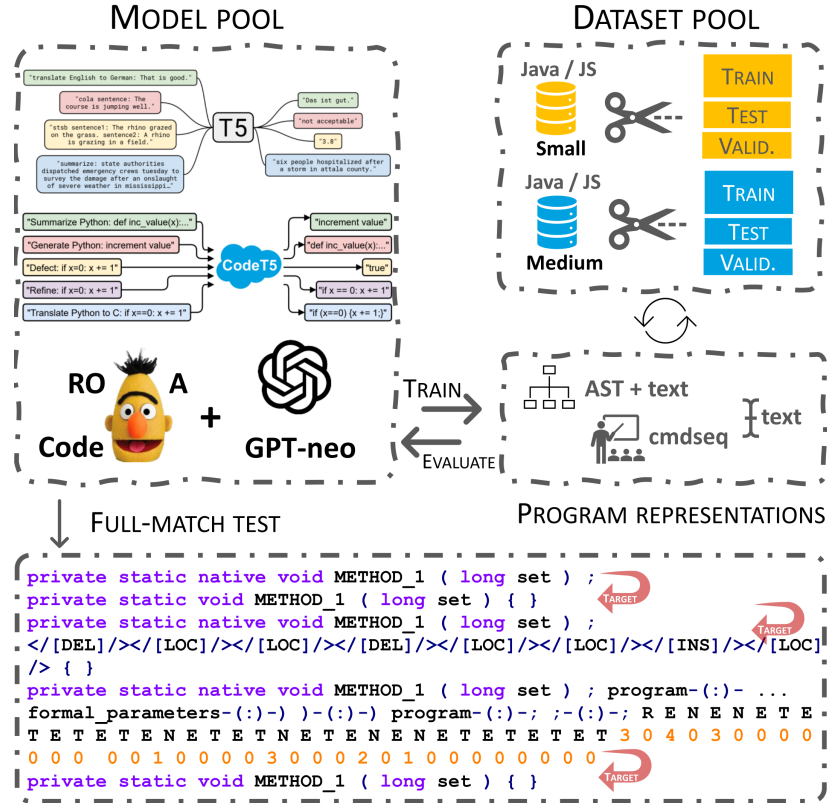


Figure 2.1: High-level approach overview of source code representation experiments.

Experiments were conducted using two primary datasets: a Java dataset from the CodeXGLUE benchmark and the FixJS dataset for JavaScript. The models were

trained on an RTX 3090 GPU, with training times varying from a few hours to over a day depending on the model size, dataset, and representation. The metric used to measure performance was the percentage of exact matches (100 % accuracy) predicted by the model, where the generated patch must be identical to the developer’s fix in the dataset. CodeXGLUE benchmark also uses this metric, and while other metrics are also reported, this was used as a fair evaluation of performance. The experimental results reveal several insights. For one, there can be a huge performance gap between two datasets. This simply means that while a model can perform relatively well on a dataset, this does not necessarily translate into the same performance on others. In our experiments, models trained on the Java dataset consistently outperformed those trained on FixJS. This is attributed to the Java dataset’s larger size and potential inclusion of duplicate samples, which may make it an “easier” learning task compared to the smaller, more strict filtering applied by the FixJS dataset.

The choice of code representation also proved to be highly influential, as no single representation is universally superior to the other. For example, the encoding strategy referred to as *cmdseq* yielded the best results on the Java dataset, significantly outperforming models trained on simple text representation. In this case, some models achieved over 30 % accuracy earning top place among other strategies. On the other hand, when this same strategy is applied on the FixJS dataset, we got mixed results, where in some cases simple *Text-to-Text* transformations outperform more sophisticated encoding strategies. This is due to the fact that the *cmdseq* representation can effectively pack information in some cases, but when there are multiple edits needed to fix the code, this advantage of information compression can quickly fade away. This highlights that the effectiveness of a representation is deeply tied to the characteristics of the dataset and the language. The *AST + Text* model, despite being theoretically more powerful due to its access to structural information, performed poorly in all experiments. We speculate that this could be due to the decoding challenge or the need for a much larger model to effectively utilize the combined information. Also, in this case, pre-trained models were not readily available, as such models are usually not pre-trained on such representation with large-scale, high-quality datasets.

Another key finding highlighted in this thesis is that fine-tuning also matters. Pre-trained models such as T5-base and CodeT5-base consistently yielded better results than training models from scratch or only training the final language model head. To take an example, CodeT5, being pre-trained on code, showed a significant edge in performance on the Java dataset.

This study demonstrates that source code representation is a critical, yet still underrepresented factor in the success of *Deep Learning*-based program repair. The experiments show that there is no solution that fits all datasets and models best; the optimal representation depends on the specific model architecture, the dataset and

its characteristics. In conclusion this work presents two key takeaways; each *Deep Learning* setting requires its own tailored data representation, and no single representation is universally optimal for every dataset and model. By open-sourcing both the dataset and the extraction tools, these works aim to provide valuable resources to the community, promoting further research into how refined code representation strategies can lead to more accurate and effective program repair systems.

2.1 Contributions

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 DL models. The method could also potentially provide an extension to the already available data mining software.

3. Green Energy Through ML: Optimizing CO₂ Electrolyzer Cell Performance

The *electrochemical reduction of CO₂* (CO₂RR) presents a promising pathway to convert a harmful greenhouse gas into valuable chemicals, used in the petrochemical value chain. However, achieving long-term, stable, and efficient cell operation remains a significant challenge to this day. The third thesis addresses this by introducing a novel methodology for *Anion Exchange Membrane* (AEM) separated electrolyzer cells with AI-guided data evaluation and optimization. The primary difficulty in optimizing CO₂RR systems lies in the complex parameter space of many controllable parameters, such as temperature, gas flow, current density, and more.

This work demonstrates that an ANN model, trained on the datasets that resulted from the logging of the steady-state operation of these electrochemical cells, can accurately predict performance under various operating conditions. The model shows high precision when predicting the behavior of the same cell under new conditions. This means, that even when predicting values that the model could never seen before, it behaves very similarly to the original cell it learned on, basically cloning its behavior. Furthermore, it also generalizes reasonably well on new cell assemblies, but otherwise identical electrolyzer cells, showcasing the potential to accelerate the optimization and industrial-scale applications of this CO₂RR technology.

A custom-built autonomous test station collected extensive experimental data by systematically varying four key process parameters: current density, cell temperature, *gas humidifier* (GH) temperature, and CO₂ inlet flow rate. This resulted in 108 unique parameter combinations. After conditioning the cell, the combination of 108 parameters was repeated twice in a row, to further analyze the reproducibility and stability of cell operation given each combination of parameters. These steps, and the core idea is presented in Figure 3.1.

The data collected was preprocessed in multiple steps. First, we filtered only for steady-state conditions, where measurements were only considered valid if parameters such as cell temperature remained stable (*e.g.*, within a $\pm 2^\circ\text{C}$ interval). Outlier

detection was also performed using Z-scores on the five target output variables (cell voltage, CO, CO₂, H₂ concentrations, and cathode gas flow rate), excluding data points with a Z-score greater than 3. These filters were designed so that they are permissive enough to include specially behaving but otherwise relevant data but strict enough to exclude data that come from *e.g.*, sensory errors.

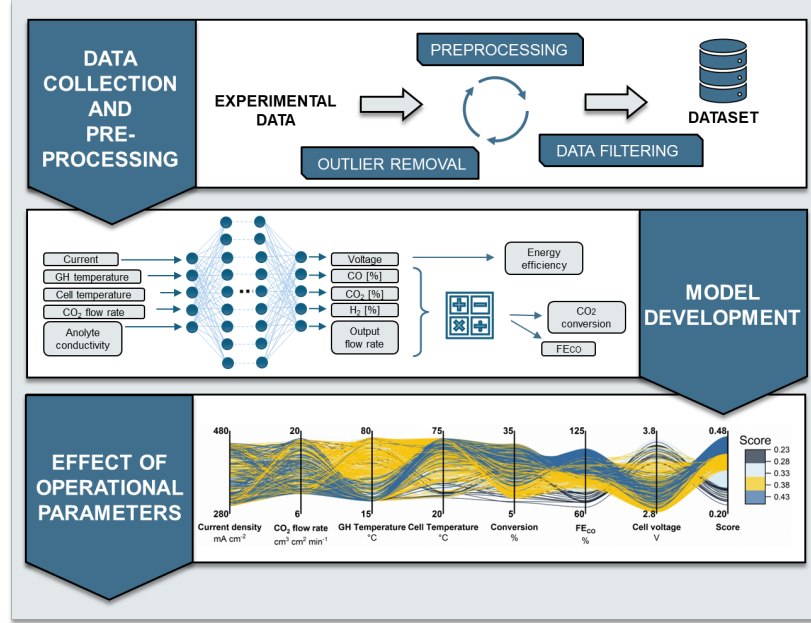


Figure 3.1: The workflow for finding the optimal values for the operation of the CO₂ electrolyzer cell.

A fully connected ANN was chosen as the predictive model after comparing its performance against other ML algorithms such as Random Forest and XGBoost, which did not generalize well to new cell assemblies. The ANN architecture consists of:

- An input layer with 5 neurons,
- three hidden layers, 132 neurons each,
- GELU activation functions between each hidden layer, and
- an output layer of 5 neurons.

The model was trained using the AdamW optimizer and *Mean Squared Error* (MSE) as the loss function. The data was split to ensure that the test set contained parameter combinations completely unseen during training and validation, ensuring fair evaluation.

After the training phase, the model was integrated with a genetic algorithm (*Non-dominated Sorting Genetic Algorithm II* (NSGA-II)) to explore an extended parameter

space that could identify optimal operating conditions, even outside the training parameter space. The multi-objective optimization aimed to maximize the *Faraday efficiency of CO formation* ($FE(CO)$) and *Single Pass Conversion* (SPC) while minimizing cell voltage. This process generated a set of promising new parameter combinations, referred to as the "external test set", which were used to validate the model's predictive prowess on a new but otherwise identical cell assembly. The ANN model demonstrated excellent performance on the test set. The predicted values for voltage, gas concentrations, and flow rates closely matched the measured values, with R^2 scores that typically exceed a value of 0.98. This, along with other metrics, highlights the achieved performance of the model. When evaluated on the external test set, the performance of the model showed some degradation due to cell-to-cell variability, a known challenge in the field of CO₂RR. Although the error increased, the model still provided reasonably accurate predictions, especially for CO concentration and output flow rate. In particular, key performance indicators such as $FE(CO)$ and SPC were predicted with less than 5 % error (MAPE), indicating the potential of the model for cross-cell applicability. The uncertainty analysis also revealed useful information. The model's predictive uncertainty was generally low and stable for the test set, although slightly higher than the physical measurement uncertainty. This suggests that the model capably learned the system's behavior, but could benefit from a more diverse training dataset. For the external set, uncertainty increased, particularly when multiple operating parameters were pushed far beyond the original training range, highlighting the model's generalization limits.

The thesis also discusses the finding's interpretability from an electrochemical point of view. For this reason, a *SHapley Additive exPlanations* (SHAP) based sensitivity analysis was performed to understand feature importance. The analysis highlighted that the cell voltage 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, and the transport of the formed carbonate ions to the anode, described by $CO_2 + 2e^- + H_2O \rightarrow CO + 2OH^-$ and $CO_2 + 2OH^- \rightarrow CO_3^{2-} + H_2O$, respectively. 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 cor-

relation between some of the input features, namely EC and Cathode Temp, which is validated by our previous findings when analyzing for 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 system behavior.

In the end, this thesis showcases a powerful framework for accelerating the optimization of AEM separated CO₂ electrolyzer cells by integrating automated experimentation with ML modeling. In general, the developed ANN model emerged as the best, capable of accurately predicting cell performance while also generalizing well in different operating conditions and cells. However, in the latter case, some performance loss is observed due to variation from cell to cell, which is a known problem in the field of CO₂RR. The AI-guided approach presented here is a significant step toward the industrialization of CO₂RR technology; by allowing accelerated experimentation of yet unexplored parameter spaces, these methods offer a realistic and efficient path to achieve stable and long-term operation of CO₂ electrolyzer cells. In the future, *Machine Learning* methods can even pave the way to long-term operations that take into account even cell aging and are capable of reacting to these factors by continuously monitoring and controlling the parameters of such cells, possibly extending their lifespan, and keeping optimal operation even in these changing conditions.

3.1 Contributions

In the third thesis group, the contributions are related to the implementation of an AI-guided framework for optimizing the electrochemical reduction of *Anion Exchange Membrane* (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.

Summary

This dissertation stands to show the growing relevance of Machine Learning techniques through different scientific fields. In the first two thesis points it is shown that there is a growing number of innovations and breakthroughs achieved via AI tools in *Software Engineering* (SE), while the third thesis shifts its attention to other scientific fields, such as the reduction of harmful greenhouse gases such as CO₂. All thesis points dive deep into the ML techniques necessary for driving the field forward in hopes of achieving even better results, or as a simple contribution to nudge other researchers in other promising directions.

The first thesis point takes the course of repairing JavaScript programs using different toolsets, such as Genetic Algorithms (GAs), and in its core genetic mutation operators, and more recent techniques such as *Large Language Models* (LLMs), currently in the form of GPT-2. It shows that JavaScript can benefit from more thorough research focused on the specifics and intricacies of the programming language. With our novel toolkit, called GenProgJS, we achieve comparable results achieved by previous tools, filling the gap of JavaScript repair along the process. The thesis examines how different mutation operators play a role in repairing JS programs and shows the benefit of using operators considered to be JS-specific. We also give a perspective of how these results compare in the world of LLMs, more specifically, we compare the results from GenProgJS to results of repair candidates produced via GPT-2. We show that in real-world applications GAs can even outperform tools like GPT-2 in program repair tasks, signifying the relevance of the tool as a baseline for JavaScript APR. To further support the quality of the patches produced and to discuss possible patch validation techniques, we examined the patch candidates for the ESLint project and performed an analysis of patch similarity. We ranked each generated patch against the original program and showed that the closer the patch is to the original program, the more likely it may be that it is correct, showing that this could even be used as a useful metric to extend the arsenal of patch correctness assessment.

The second thesis shifts the focus on code representation and code extraction. This thesis is in contrast with the first in a sense that the emphasis here is not solely on architectural designs and toolkit comparison. It examines whether existing techniques could benefit from more educated choices on code representation in the field of *Automated Program Repair*. We show that not all models benefit from the same

representations and also point out that the dataset or chosen language could also play a crucial role on how one chooses to represent source code. One other aspect of the thesis is the data, and its mining procedure as well. It offers analysis tools and extraction techniques that would allow other great miner tools to be implemented as a plugin. The software could then be easily used in a programming language agnostic way to extract code segments from any source code necessary. To give an example, these works also introduced a publicly available dataset on Zenodo.

The third thesis, although may differ in theme, also uses ML techniques in other scientific domains. That is, the focus is now on the reduction of CO₂ emission, through AEM separated electrolyzer cells. While conducting real-life experiments in the hopes of achieving optimal cell performance is costly and time-consuming, ML techniques can help mitigate these issues. Via learning on data collected from the electrolyzer cell's test station, this work basically makes a digital copy of the real system, mocking its behavior, enabling rapid tests and optimizations on said cells. Producing results in this way is not only less time consuming, but the cost of cell assemblies and operation costs could also be cut. This work experiments with multiple models and techniques to achieve the best possible cell mocking behavior and uses modern optimization techniques to achieve high-quality results on the real-life cell assembly.

Table 3.1: *Publications supporting the thesis points.*

	Publications					
	[2]	[6]	[5]	[3]	[4]	[1]
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 included in the dissertation

- [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. – D1 (Impact Score: 17.51)
- [2] 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. – Q1 (Impact Score: 6.78)

- [3] Dániel Horváth, and László Vidács. "Yet another miner utility unveiling a dataset: Codegrain." In International Conference on Data Technologies and Applications. 2024.
- [4] 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.
- [5] 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.
- [6] Viktor Csuvi, 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.

Publications not included in the dissertation

- [1] Viktor Csuvi, 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.
- [2] András Kicsi, Péter Sánta, Dániel Horváth, Norbert Kőhegyi, Viktor Szvoren, 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.
- [3] Viktor Csuvi, 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.

Összefoglalás

Ez a disszertáció azt kívánja bemutatni, hogy a gépi tanulás (Machine Learning, ML) technikái egyre inkább nagyobb jelentőséggel bírnak a tudomány számos területén. Az első két tétel a mesterséges intelligencia (MI) eszközeinek szoftverfejlesztésben (Software Engineering, SE) elért eredményeit tárgyalja, míg a harmadik tétel egyéb tudományos területekre koncentrál. Például olyan káros üvegházhatású gázok, mint a CO₂ kibocsátásának csökkentésére. A tételek mélyen elemzik azokat az ML-technológiákat, amelyek az adott szakterület előrehaladását szolgálják, akár az eredmények további javításának reményében.

Az első tétel JavaScript programok javításával foglalkozik különböző eszközkészletek segítségével. Ilyen például a genetikai algoritmusok (Genetic Algorithms, GAs) és azok alapját képező genetikai mutációs operátorok, valamint újabb módszerek, például nagyméretű nyelvi modellek (Large Language Models, LLMs), jelen esetben a GPT-2 formájában. A tétel rámutat, hogy a JavaScript nyelv sajátosságaira összpontosító kutatások komoly előnyöket rejthetnek. Az általunk kidolgozott új eszközkészlettel, a GenProgJS-sel, olyan eredményeket értünk el, amelyek összevethetők a korábbi eszközök teljesítményével, és ezzel hozzájárulunk a JavaScript program javítás területének fejlődéséhez. A dolgozat megvizsgálja, hogy a különböző mutációs operátorok milyen szerepet játszanak a JavaScript programok javításában, és bemutatja, hogy a JavaScript-re szabott operátorok alkalmazása milyen előnyöket hordoz. Emellett összehasonlítjuk az így elért eredményeket a GPT-2 által generált javítási javaslatokkal. A vizsgálatok alapján kimutatható, hogy a valós környezetben végzett alkalmazások során a GA-alapú megközelítések akár jobban is teljesíthetnek, mint a GPT-2, ami alátámasztja a GenProgJS eszköz relevanciáját, mint a JavaScript-APR egyik kiindulópontját. A javítások minőségének további alátámasztása és a validációs módszerek vizsgálata érdekében a ESLint projekthez generált javítási javaslatokat is elemeztük, különös tekintettel a hasonlóságukra az eredeti programhoz. Az eredmények azt mutatják, hogy minél közelebb áll egy patch az eredeti kódhoz, annál valószínűbb, hogy helyes. Ez a megfigyelés akár új mérőszámként is szolgálhat a javítások helyességének értékelésében.

A második tétel a kód reprezentációjára és kinyerésére helyezi a hangsúlyt. Ez eltér az első tételtől abban az értelemben, hogy itt nem kizárólag az eszközarhitektúrák és teljesítménybeli különbségek állnak a középpontban. A cél annak

vizsgálata, hogy a meglévő módszerek miként profitálhatnak a forráskód reprezentációjának tudatosabb megválasztásából az automatizált programjavítás (Automated Program Repair, APR) területén. Bemutatjuk, hogy nem minden modell részesül előnyben ugyanabból a reprezentációból, és azt is kiemeljük, hogy a választott programnyelv és adathalmaz szintén kulcsfontosságú tényezők lehetnek ebben a döntésben. A tétel egy másik hangsúlyos eleme az adatok feldolgozása és bányászata. Elemző eszközöket és kinyerési technikákat kínálunk, amelyek segítségével bővíthető és testreszabható bányászmodulok készíthetők bővítményként. A fejlesztett szoftver nyelvfüggetlen módon is képes kódrészleteket kinyerni, így bármilyen forráskód feldolgozására alkalmas lehet. Ennek szemléltetéseként kutatásunk során egy nyilvánosan elérhető adathalmazt is közzétettünk.

A harmadik tétel, bár tematikájában eltérő, szintén ML-technológiákra támaszkodik, csak épp más tudományos területen. A figyelem itt a CO₂-kibocsátás csökkentésére irányul, mégpedig anion cserélő membránnal (Anion Exchange Membrane, AEM) elválasztott elektrolizáló cellák révén. Mivel a valóságos kísérletek végrehajtása költséges és időigényes, az ML-technológiák segíthetnek ezeknek a kihívásoknak a mérséklésében. Az elektrolizáló cella tesztállomásán gyűjtött adatok felhasználásával a kutatás egy digitális másolatot készít a valódi rendszerről, annak működését szimulálva, így lehetővé téve gyors tesztek és cella optimalizálást. Az ilyen módon születő eredmények nemcsak kevésbé időigényesek, de az üzemeltetési költségek is jelentősen csökkenthetők (cellák összeállítása, tesztállomás üzemeltetése). A munka többféle modellt és technikát is kipróbál a legpontosabb szimuláció elérése érdekében, és modern optimalizációs módszerekkel igyekszik megtalálni a cella működtetéséhez szükséges lehető legoptimálisabb paramétereket.

Acknowledgments

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. Supported 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