Characterization of Data Partitioning Techniques for Ensemble Methods in Automatic Program Repair

Kevin Zhang
kevinzhang@purdue.edu
Purdue University
West Lafayette, Indiana, USA

ABSTRACT

Fixing bugs in code is a time-consuming endeavor. Automatic Program Repair (APR) seeks to autonomously fix bugs present in source code through patch generation. Recently, the application of neural networks and deep learning techniques, including neural machine translation, to this field has yielded good results, achieving state-of-the-art rates for fixes on the Defects4j and QuixBugs benchmarks. Ensemble techniques have been used to improve the learning properties of these models and achieve better results. However, a systematic measurement of the effectiveness of different ensemble methods has not been carried out. Partitioning of the dataset for training with bagging was selected as a simple and comparable ensemble method. Clustering of the bug type via human categorization and clustering via the encoder hidden state output of a pre-trained model were compared with random divisions to split the training data for ensemble models. This study then compared the results of these different ensemble methods with the same model design on the QuixBugs benchmark to determine their relative effectiveness. It was found that models trained on randomly partitioned data outperformed models trained on data clustered by both human categorization and machine embeddings, fixing 25 bugs on the QuixBugs benchmark as compared to 20 each for the two clustering methods. Further conclusions and observations about the performance of each approach, as well as recommendations for further approaches in ensemble techniques will be provided based on the comparison and analysis of results for these methods.

KEYWORDS

Automatic program repair, neural networks, ensemble

ACM Reference Format:
Kevin Zhang. 2018. Characterization of Data Partitioning Techniques for Ensemble Methods in Automatic Program Repair. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Fixing bugs manually can add many hours to development time. Automatic program repair (APR) can offer fixes to drastically reduce this debugging time and improve the health of devices operating over the internet of things [8, 12]. Early contributions to this field, such as GenProg [7] and Nopol [24] as well as its extension in DynaMoth [5] employ simple algorithmic or mutation-based fix patterns, with Nopol and DynaMoth limiting their fixes to even specific types of bugs [6]. Recently, however, deep learning and neural networks have produced a shift in the design of APR methods, with these learning-based models achieving state-of-the art results on the Defects4j and QuixBugs benchmarks. Recursive neural networks and LSTM models have enabled sequence-to-sequence learning as well as its use in neural machine translation via encoder-decoder models trained on massive datasets [20]. SequenceR [5] makes direct use of this process, introducing a translation process from a buggy line of code to a patch, but not achieving results better than pattern-based fixes. DLFix [11] looks to expand the context awareness of the model, using a multi-layered system using the abstract syntax tree to encapsulate both the buggy line and its surrounding context. Recoder [26] transfers the learning process to one of edits, using a novel provider-decider architecture to generate more correct and effective changes. RewardRepair [25] modifies the loss function in training to promote generation of changes in code which compile, are similar to developer-written code, and which is robust. These methods have used developments in computing to begin to apply the human intuition for solving problems in code, separating the processes of understanding the context of a buggy line, and of learning to produce fixed, compiling code with different metrics.

The use of ensemble methods in the training and development of deep learning models continues the trend of mimicking human learning, trying to encapsulate the idea of the wisdom of the crowd [17]. CoCoNuT [13] uses modified CNN based NMT models with ensemble learning techniques to encapsulate context and bug information more effectively. CURE [10] expands on CoCoNuT's architecture, further introducing a "programming language" model, adapted from the GPT large-scale pretrained NMT model to better capture symbols in the program and improve search space. In general, advancements in deep learning have allowed for incremental improvements by adjusting philosophy, architecture, or other details to improve the bug-fixing capabilities of deep learning-based APR techniques.

An additional direction, which has not been explored much, is the method of training. Ensemble techniques offer a fairly simple way to improve the performance of models by potentially learning

a better representation, while improving computability and generality [17]. Its use allows for a more diverse solution strategy with minimal overhead on the experimentalist's part. Betten et al. find that ensembles of local models trained via data clustering outperform global models [3], noting that this advantage depends on the clustering method and properties of the model. Furthermore, they work with linear regression models, with different convergence properties as compared to neural networks. Work done by Partridge [14] and Sharkey [18] can be generally separated into ensemble via variation of initialization weights, variation of network hyperparameters, variation of network type, and variation of training data. They find that variation of network type among ensemble methods is most effective, with variation of training data for the ensemble models being next most effective. Different network architectures and design for use in automatic program repair are still actively developed, and many require a different set of inputs with different tagging of data to be properly trained. Variation of data usage for ensemble methods allows for usage of the same training methodology within a model and offers a widely applicable and general approach towards improving the performance of existing and future techniques. The benefits and drawbacks, then, of ensemble training based upon different methods for partitioning of the data set are important concepts to be addressed.

2 APPROACH

The APR model used is based on a tree-decoder architecture, operating directly on the abstract syntax tree (AST) level. The function in which the bug is located is first transformed into an AST by static analysis tools, which also locate the buggy line- all of which is standard information given to current APR techniques [12], [13]. These pieces are combined to produce an AST representation of the buggy function, including the buggy index and buggy node- the location of the node representing the line where the bug occurs, and the classification of that node itself. The context AST and buggy index are transformed into a fixed length representation via an encoder before being decoded into a transformed AST to be reconstructed into Java source code. To gather data for training of the model, Java programs were scraped from GitHub and transformed into abstract syntax trees via javalang [21] and JavaParser [19]. In selecting the data, commits were scraped for information ("fix", "patch") to select buggy source code and correct fixed code. This data was then partitioned via different methods discussed hereafter to produce the variations in training data for training of an ensemble of methods. The models were then used to produce fixes on the QuixBugs Java bugs collection as a benchmark to compare their effectiveness with the same transformation into AST used as in producing the training data. A total of approximately 450,000 buggy functions and their appropriate fixes were collected and transformed into ASTs to act as the full training dataset.

2.1 Clustering Methods

To produce a fair comparison of the different partitioning methods, each method of partitioning was used to generate four different sets of training data. The training data, collected from patches scraped from GitHub focused on replacements and adjustments to code to produce fixes. However, some classes of bug require the addition of new lines and statements to properly fix. A fifth additional identical model was then added to each of the methods to account for this. Models were coded and trained via the version 1.4.0 PyTorch [2] framework with CUDA 10.0 support. The general partitioning process is detailed in Figure 1.

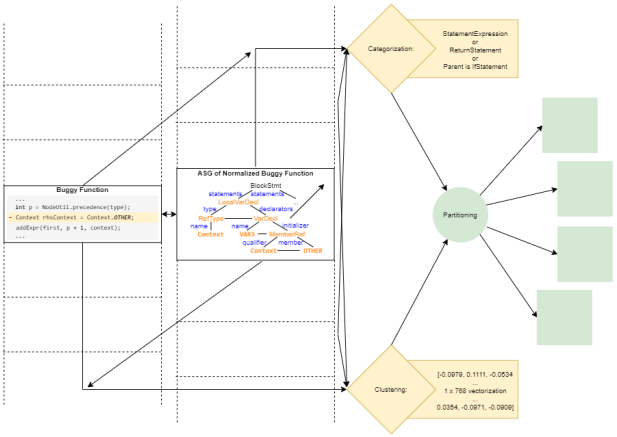

Figure 1: Overview of partitioning methods: Categorization used human selected categories on the buggy origin while Clustering used clustering methods on vectorized embeddings of the buggy source code.

Heuristic Categorization: Intuitively, it would be expected that clustering and division of the training data to train separate models would produce better results with different models specializing across a spread of different cases for the data. Human determined patterns in bug-type are therefore a potential way to divide the data. Through this process, the type of bug was determined via the type of node which was the root of the buggy subtree- found by the buggy index across all nodes.

The categories were selected based on the patterns observed through human developer experience to be necessary in fixing various types of bugs. The category names are italicized to reference for further use in the paper. Statement- buggy nodes categorized in the AST as statements, including assignment, instantiation, and method invocations represented a first class. Return- buggy nodes identified as return statements formed this second category. If/While- as bugs are often within the condition of if and while statements, this category consisted of datum where the parent of the buggy node in the AST was classified as an if or while statement. Mixed- a final miscellaneous group was selected as a combination of the remaining data.

The number of training data within each category was not equal, and to better match convergence, the models were trained with varying iterations over their datasets. The number of iterations were selected for simplicity- multiples of five- and for similar validation scores. The If/while and Return models trained for twenty epochs across their smaller datasets. The model operating on the Statement category was trained over fifteen epochs and had the largest amount of data, but converged more slowly, likely due to the disparate nature of examples covered by nodes expressed via StatementExpression in JavaParser AST representation. The Mixed

Table 1: Partitioned dataset sizes. The combined number of training data which the clustering ensemble models were trained on is significantly less than the number on which the other two ensembles trained.

Ensemble Approach	Dataset	Size
Random	All	≈113,000
Categorization	If/While	47324
	Return	47899
	Statement	252070
	Mixed	106497
Clustering*	0	135847
	1	92256
	2	67086
	3	45870

category was posited to be composed of disparate bugs felt to be more difficult to fix, and so was trained over ten epochs.

Clustering via Embedding: Much work has been done in the field of software engineering to automatically classify and cluster source code programs. These methods often focus on invariants within the code through dynamic analysis [4, 9, 16, 23], a process far too computationally expensive and time consuming for APR techniques. Additionally, these approaches are usually tailored to analysis for massive open online courses on coding, analyzing source code written to solve the same problem. This specificity of code structure is in stark contrast to the wide variety of code structures and problems which automatic program repair faces. Input buggy functions, acting as context, were therefore embedded into vectorized representations by the final hidden state output of the version 4.10.0 T5EncoderModel [1], borrowing from the increasing use case of large pre-trained neural networks. The vectorized representation of each piece of training data was then padded to produce a fixed length representation and clustered via the K-means method, implemented by the Scikit learn library [15], with hyperparameter of four clusters.

Similar to the categorization by human recognized bug-types, the distribution of training data for the clusters was not even. A similar process for selecting the amount of training epochs was used as in the heuristic categorization ensemble training, namely for roundness of number and for similar convergence on validation. Then, models with data from cluster-0 and cluster-1 were trained over ten epochs, while the model trained on cluster-2 was trained over fifteen, and the model trained on cluster-3 trained over twenty.

Comparisons: Cross comparisons of the clusters produced by the three methods shows no significant overlap in the clusters produced as measured by a chi-squared test for independence. These three methods, therefore, represent clearly distinct division of the data and offer insight into the impacts on performance of these differences.

2.2 Patch Generation and Validation

Once all ensemble models had been trained, patch generation and validation was composed of three steps. First, the inference from each model on the benchmark bugs produced replacement AST-subtrees. These patches were generated node by node via a beam-search method with beam size of 1000, with each patch additionally receiving a score representing the model's confidence in the patch being a correct fix. Second, a reranking process mixed the top two results from each model into the reranked top ten, before letting other patches settle in based on their score from the respective generating model. Third, moving downwards in rank on the reranked list, each patch AST was translated into source code and the resulting patch was evaluated by the bug test cases. Once a plausible patch was found- one which passed all test cases passing before and one additional non-passing case- the validation ended, and that patch was selected as a fixing patch. Finally, returned plausible patches were manually checked and considered correct only if they were semantically equivalent to the developer patches.

3 EVALUATION AND RESULTS

On the QuixBugs benchmark of 39 bugs, the ensemble model trained on randomly partitioned data outperformed the ensemble models trained with data split via the other two methods- machine clustering and type categorization. The first was able to successfully rank and offer a fixing patch for 25 bugs out of the thirty-nine, while the latter two ensemble models were only able to successfully offer fixes for 19 bugs. All three models solve 17 common bugs, with each pair of ensemble models solving one additional bug which the third was unable to. The random model, performing the best, solved an additional 6 bugs which were not solved by either of the other models. Table 2 details which bugs were solved by which models. With the same training data and model design, the three methods provided very similar fixes, and can be seen to have solved nearly the same set of bugs. However, they can be seen to be differentiated in the fixes which they offer, as well as in operation- which can be seen through analysis of their varying attention maps to different information over the bugs within the benchmark. These results indicate that ensemble via the division of training data can have a significant impact on the final performance of the resulting model on APR tasks. Broad questions based on the significant difference in performance between the random ensemble models and the two clustering methods are addressed within the Discussion section of this paper.

3.1 Generation Differences

The different fixing patches offered in response to the QuixBugs LEVENSTHEIN bug (represented as QuixBugs ID 17), offers a good example of the differences in produced fixes. Figure 3 demonstrates that all three ensemble models were able to offer patches which correctly fixed the program, but with different patches. The cluster ensemble model produces a fix which matches the human developer-written fix. On the other hand, the random and category ensemble models produce fixes which are semantically equivalent to the correct patch but are not quite as elegant.

Similarly, the fixing patches produced to solve the bug present in the QuixBugs FINDINSORTED benchmark bug show that the differences in patch generation also extend to the scoring and ranking process. Figure 3 shows the category and clustering ensemble models fix the bug with an additional line proposed by the insertion

Table 2: Fixed Bug IDs. Quixbug IDs were determined by their order from top to bottom appearing on the QuixBugs GitHub page (https://github.com/jkoppel/QuixBugs/tree/master/correct_java_programs), starting with ID 1 for BITCOUNT and ending at ID 39 WRAP

Fixing Ensemble Approaches	Bug IDs
Random Only	15, 19, 28, 35, 37, 38
Random and Clustering	39
Random and Categorization	20
Clustering and Categorization	23
All Approaches	1, 2, 3, 5, 6, 7, 9, 11, 12, 14, 17, 24, 25, 27, 29, 33, 34

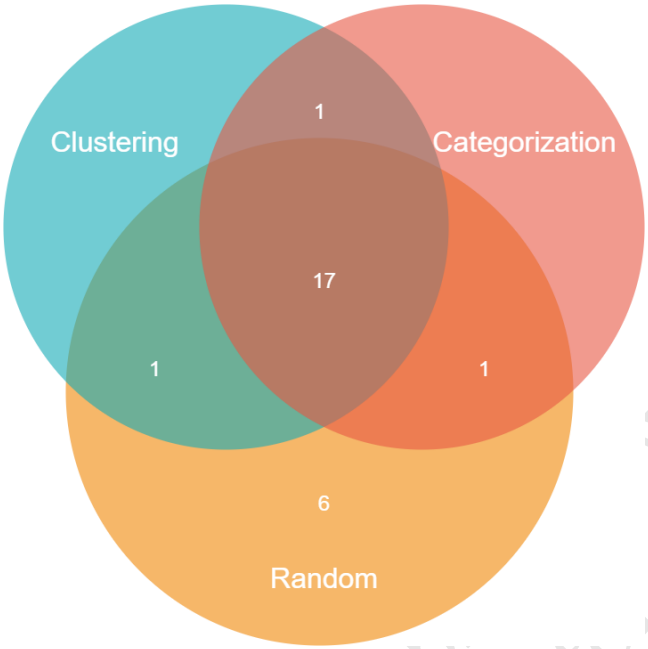

Figure 2: Number of bugs in sets by which models were able to correctly patch

model, producing a semantically equivalent fix in the bug's context, while the random ensemble model offers the proper replacement fix.

3.2 Attention Analysis

Determining why neural networks offer the results that they do is a very difficult process and is not fully understood. However, attention mappings can potentially offer insight into the inner workings of the network. Comparisons of these attention mappings for the different models within the ensembles, then, can shed some light on some of the differences in model generation. Figure 4 shows a comparison of the attentions in generating the patching sequence over both context and the buggy nodes for BITCOUNT, the first bug in the QuixBugs benchmark, and is generally representative of the attention maps of other bugs within the benchmark in the context of the following points.

Model Specificity: Analyzing the attention from the generated nodes towards the buggy sequence shows a strong pattern- the

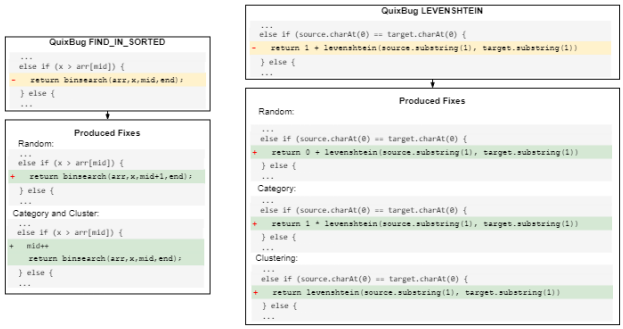

Figure 3: Fixing patches generated by the different ensemble methods for bugs within the QuixBugs benchmark.

return and statement category ensemble models have generally lower attention values to the beginning of the buggy sequence. This generally shows that those models have learned the beginning of the sequence is unimportant, instead shifting attention later in the buggy sequence. This suggests that the model specialized into the bug category which it was trained on, as the first node is what was used to define the bugs which these models trained on. This is in contrast to the if/while and mixed category models which maintain similar or slightly higher focuses as compared to both the return and statement category models, as well as the baseline attention of the random general models.

Context Attention: Analyzing the attention from the generated nodes towards the whole context function given to the model showed that all models had lower attention values for the surrounding context, focusing almost entirely on the buggy nodes anyways. With the low solution rate for more complex bugs, this attention map suggests that improvements for attention and focus on surrounding context should be sought for and explored.

3.3 Category Ablation

An ablative study on the category ensemble models was carried out to discover insights into the relative performances of the different models within the ensemble. For each bug within the QuixBug benchmark, the reranking step was carried out with only the category which the bug would be ascribed to if it were within the training data. For both the normal reranking process involving patches generated by all models, as well as the reranking including only the patches generated by the matching category model;

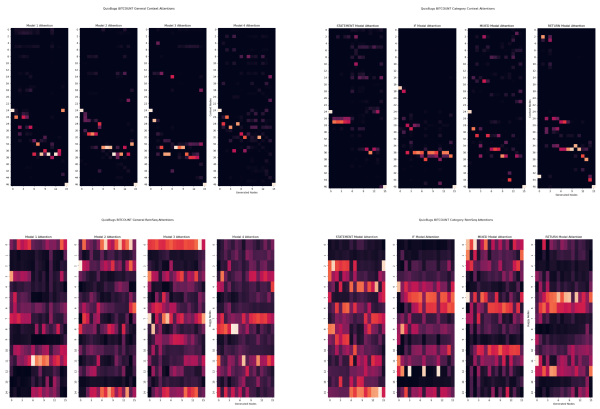

Figure 4: Various attentions for Random and Categorization models

Table 3: Ranking of perfect match patch. Category Rank denotes the ranking of the patch when only patches from the category which matched the bug were considered, while All Rank denotes the rank when patches from all category models were considered.

Bug ID & Type	Category Rank	All Rank
1-Statement	0	0
2-If/While	669	1187
3-Mixed	10	54
6-If/While	27	10
9-Return	N/A	484
11-Statement	5	25
12-Return	N/A	70
14-If	6	13
17-Return	N/A	2834
20-Statement	0	0
23-Statement	3	15
24-If/While	1	3
27-If/While	0	0
29-Statement	192	881
33-Mixed	N/A	109

this process was also only carried out for plausible matches which perfectly matched the nodes within the correct, developer-written solution for simplicity's sake. If the model performed well, the ranking of the matching patch should be lower when considering only patches generated by the model as compared to patches gathered from all models within the ensemble. As can be seen in Table 3, the return category model generally performed worse relative to other models, as many fixes on return type bugs within the benchmark were offered not by the return type model, but by other models instead. This large variation in performance across the models within the ensemble trained on categorized data likely partially explains the entire ensemble's worse performance as compared to the ensemble trained on randomly partitioned data and produces more questions which could be valid areas of further study to be further expanded upon within the Discussion section.

4 DISCUSSION

4.1 Results Significance

Human coding often focuses on decomposition of the problem and searching for patterns as main points in programming strategy [?]. With the random ensemble model outperforming both machine and human based clustering techniques, results from this experiment suggest that general domain knowledge is more important than specific patterns in fixing bugs within code, contrary to the prevailing methodology for teaching code to new students. With a significant different in performance between the random ensemble models and the two clustering ensemble methods, the reason for this disparity is a pressing question.

There are multiple potential explanations which can invite further exploration into the subject of data portioning for ensemble in learning-based APR techniques. First, unbalanced training data produced by the training data produced one or two stronger models and two or three weaker models, with the increase in strength in specific categories or clusters unable to outweigh the decrease in performance in others. This is supported by the analysis of category rankings of perfect matching patches, but could be representative of an underlying issue, where specific bug types are much more represented in training data and in practice as compared to others. Second, the clustering performed within this experiment is poor. This is certainly a possibility, and better classification of bug type deserves further exploration. Perhaps more interesting is that the earlier prognosis is true: general bug-fixing knowledge is better than more information about a specific bug type. This would then place more focus on the distillation of coding syntax and semantics into APR models and the role of context in generating fixes, as compared to the buggy line.

Related to the varying performance of categorization models, the use of more training epochs was implemented in an attempt to mitigate the differences in training data size, but results showed that the return category model performed relatively poorly compared to the other type-categorization models. With the validation score used as a comparison metric to match the learning progress of the models and its failure in this case, as well as concerns about the distribution of bugs in training data versus that which needs to be analyzed in benchmarks and use-cases, it seems that APR methods are also in need of a better metric to determine convergence.

4.2 Limitations

As can be seen from the comparison of clusters within the Methods section, bugs in the machine clustering process resulted in those ensemble models being trained overall on less data, which has a negative impact on the training process and final result. However, we hypothesize that the re-introduction of the lost training data will not close the approximately 25% increase in solved bugs which is the difference between the random and machine clustered ensemble performances on the QuixBugs benchmark.

Additionally, the QuixBugs benchmark is a very small benchmark, composed only of 39 very simple bugs. Thus, the results on this benchmark are not guaranteed to be generalizable. As a future

extension of this work, the application of these models to the much larger and more comprehensive Defects4j Java bugs benchmark may change observed patterns and will increase the generalizability of the combined results

5 CONCLUSION

A brief foray into the systematic testing of ensemble approaches using data partitioning techniques is presented. With randomly partitioned data performing best, we tentatively hypothesize that general domain knowledge is more important than specific categorical bug type data in the realm of learning-based approaches for Automatic Program Repair. Comparisons of model performances and characteristics also allow us to characterize thoughts and directions for future research into training methodology for APR models, especially in the development of better metrics for determining convergence, due to the difficulties of using validation score with potential drift between testing set data and real-world applications.

ACKNOWLEDGMENTS

To Professor Lin Tan, and Nan Jiang for their mentorship and guidance throughout this project.

Furthermore, to Purdue University EURO and the SURF Program for offering support in writing and presenting, and to University of Waterloo for the use of its computing servers to train the models tested within the experiment.

REFERENCES

[1] 2011. *T5.* https://huggingface.co/docs/transformers/model_doc/t5#transformers.T5EncoderModel

[2] 2022. *PyTorch.* https://pytorch.org/

[3] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E. Hassan. 2015. Towards improving statistical modeling of software engineering data: think locally, act globally! *Empirical Software Engineering* 20 (4 2015), 294–335. Issue 2. https://doi.org/10.1007/S10664-013-9292-6

[4] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding automatically-generated patches through symbolic invariant differences. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 411–414. https://doi.org/10.1109/ASE.2019.00046

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2018. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. (12 2018). https://doi.org/10.1109/TSE.2019.2940179

[6] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic code synthesis for automatic program repair. *Proceedings - 11th International Workshop on Automation of Software Test, AST 2016*, 85–91. https://doi.org/10.1145/2896921.2896931

[7] Claire Le Goues, Thanh Vu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72. Issue 1. https://doi.org/10.1109/TSE.2011.104

[8] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62 (11 2019), 56–65. Issue 12. https://doi.org/10.1145/3318162

[9] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53 (12 2018), 465–480. Issue 4. https://doi.org/10.1145/3296979.3192387

[10] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. *Proceedings - International Conference on Software Engineering* (2 2021), 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[11] Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. *Proceedings - International Conference on Software Engineering* (6 2020), 602–614. https://doi.org/10.1145/3377811.3380345

[12] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 118–129. https://doi.org/10.1109/SANER.2018.8330202

[13] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. *ISSTA 2020 - Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 101–114. https://doi.org/10.1145/3395363.3397369

[14] Derek Partridge. 1996. Network generalization differences quantified. *Neural Networks* 9, 2 (1996), 263–271. https://doi.org/10.1016/0893-6080(95)00110-7

[15] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, null (nov 2011), 2825–2830.

[16] David M. Perry, Roopsha Samanta, Dohyeong Kim, and Xiangyu Zhang. 2019. SemCluster: Clustering of imperative programming assignments based on quantitative semantic features. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 860–873. https://doi.org/10.1145/3314221.3314629

[17] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8 (7 2018), e1249. Issue 4. https://doi.org/10.1002/WIDM.1249

[18] Amanda Sharkey. 1996. On Combining Artificial Neural Nets. *Connect. Sci.* 8 (12 1996), 299–314. https://doi.org/10.1080/095400996116785

[19] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. 2019. *JavaParser.* https://github.com/javaparser/javaparser

[20] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems* 4 (9 2014), 3104–3112. Issue January. https://arxiv.org/abs/1409.3215v3

[21] Chris Thunes. 2020. *javalang.* https://github.com/c2nes/javalang

[22]]compthinking Rutgers University. [n. d.]. *What is Computational Thinking?* https://ctpdonline.org/computational-thinking/

[23] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (11 2017). https://doi.org/10.48550/arxiv.1711.07163

[24] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43 (1 2017), 34–55. Issue 1. https://doi.org/10.1109/TSE.2016.2560811

[25] He Ye, Matias Martinez, and Martin Monperrus. 2021. Neural Program Repair with Execution-based Backpropagation. (5 2021). https://doi.org/10.1145/3510003.3510222

[26] Qihao Zhu, Zeyu Sun, Yuan An Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 341–353. https://doi.org/10.1145/3468264.3468544