Feature Extraction, Learning and Selection in Support of Patch Correctness Assessment

Viktor Csuvik[a], Dániel Horváth[b] and László Vidács[c]
Department of Software Engineering, University of Szeged, Hungary

Keywords: Automated Program Repair, Patch Assessment, Overfitting Patch, Code Features, PCC.

Abstract: Automated Program Repair (APR) strives to minimize the expense associated with manual bug fixing by developing methods where patches are generated automatically and then validated against an oracle, such as a test suite. However, due to the potential imperfections in the oracle, patches validated by it may still be incorrect. A significant portion of the literature on APR focuses on this issue, usually referred to as Patch Correctness Check (PCC). Several approaches have been proposed that use a variety of information from the project under repair, such as diverse manually designed heuristics or learned embedding vectors. In this study, we explore various features obtained from previous studies and assess their effectiveness in identifying incorrect patches. We also evaluate the potential for accurately classifying correct patches by combining and selecting learned embeddings with engineered features, using various Machine Learning (ML) models. Our experiments demonstrate that not all features are equally important, and selecting the right ML model also has a huge impact on the overall performance. For instance, using all 490 features with a decision tree classifier achieves a mean F1 value of 64% in 10 independent trainings, while after an in-depth feature- and model selection with the selected 43 features, the MLP classifier produces a better performance of 81% F1. The empirical evaluation shows that this model is able to correctly classify samples on a dataset containing 903 labeled patches with 100% precision and 97% recall on it's peak, which is complementary performance compared to state-of-the-art methods. We also show that independent trainings can exhibit varying outcome, and propose how to improve the stability of model trainings.

1 INTRODUCTION

The concept of Automated Program Repair (APR), which involves the automated resolution of software bugs, has gained significant traction alongside the growing prevalence of software usage. The predominant focus of APR research revolves around Generate-and-Validate (G&V) approaches, wherein patch candidates are generated (e.g., via genetic algorithm, heuristics, or learned code transformations) and subsequently verified against an *oracle*. If the oracle is the test suite (which is usually), the approach is referred to as test-suite-based program repair. Despite facing criticism on multiple occasions, these methods still shape the trajectory of APR research (Kechagia et al., 2022). A notable obstacle encountered in test-suite-based repair is the potential to create a patch that enables the entire test suite to pass, yet remains incorrect. This phenomenon is commonly referred to as the overfitting patch problem (Wang et al., 2021) and the goal of Patch Correcness Check (PCC) is to determine the actual correctness of a patch, without additional manual effort. We call a patch overfitting, if it only passes the test suite, but it does not fix the program. The expectation from a correct patch is that (1) the test suite passes, (2) it fixes the original fault and (3) does not introduce new bugs.

The generation of overfitting patches leads to the generation of program repair patches with limited utility, thereby substantially affecting the practical applicability of program repair. It also makes developers less confident in APR tools, thus reducing their widespread use. The use of data-augmentation techniques (Xin and Reiss, 2017a; Ye et al., 2021), and repair operator curation (Wen et al., 2018) can lead to more correct patches, but at the time of writing this paper, the classification of generated patches is the most popular research direction. Among these recent studies have introduced static methods for detect-

[a] https://orcid.org/0000-0002-8642-3017
[b] https://orcid.org/0000-0001-8855-921X
[c] https://orcid.org/0000-0002-0319-3915

Csuvik, V., Horváth, D. and Vidács, L.
Feature Extraction, Learning and Selection in Support of Patch Correctness Assessment.
DOI: 10.5220/0012746900003753

In *Proceedings of the 19th International Conference on Software Technologies (ICSOFT 2024)*, pages 23-34
ISBN: 978-989-758-706-1; ISSN: 2184-2833

ing overfitting patches, mainly because of their ease and speed of use. Xin *et al.* (Xin and Reiss, 2017b) defined 2 static features in ssFix, while 3 static features are leveraged in S3 (Le et al., 2017) and 202 in ODS (Ye et al., 2019). Another approach is the use of source code embeddings either directly (Lin et al., 2022) or by calculating similarity in the embedded vectors (Csuvik et al., 2020). Most of these works operate on distinct datasets, and approaches are in competition with each other and not complementary. In this work, our aim is to handle all of these crafted knowledge as *features* for a machine learning model and select which of these are the most useful in the PCC domain.

In our research, we used 903 patches generated by APR tools in Defects4J (Just et al., 2014) and previously labeled by researchers (Wang et al., 2021). For these patches we mapped features achieving state-of-the-art results in previous works: hand-crafted features (Xin and Reiss, 2017b; Le et al., 2017; Wen et al., 2018), static code features (Ye et al., 2019), embedding vectors (Lin et al., 2022) and similarity metrics (Csuvik et al., 2020) - thus forming a feature vector of 490 dimension. On this set, we first performed a feature selection process, which resulted in a 43-dimensional feature vector. The selected features served as the basis for our model selection, in which we trained Machine Learning (ML) models and selected MLP (Multi-Layer Perceptron) that yielded the overall best performance. In Deep Learning (DL) approaches, stability is essential because only reporting good DL performance may threaten the experimental conclusions (Liu et al., 2021a). To this end, we publish the training dataset along with fixed seeds and report the random effects of the initialization along with our results. We also propose a technique that can improve the stability of model trainings. The following points summarize our main contributions:

- **Evaluation on a Unified Benchmark:** previous works are evaluated on distinct datasets, and now we bring these to a common ground.
- **Feature selection and Combination:** approaches are handled complementary, here the best features are selected from previous works.
- **Reproducible ML Model Trainings:** ML models are selected and trained in a reproducible manner to achieve competitive performance.

2 RESEARCH OBJECTIVES

We organize our experiment around the following research questions:
RQ1. *Which features are the most useful for PCC?*
RQ2. *What ML models are the most promising?*
RQ3. *How can we further enhance the performance and stability of predictions?*

The source code and detailed experimental data can be found in the online appendix of this paper (onl, 2024).

3 BACKGROUND

Traditionally, a patch is deemed correct if it successfully passes all the test cases, unfortunately practical test suites often fail to guarantee the accuracy of generated patches. Consequently, patches that pass all tests (referred to as plausible patches) may incorrectly address the bug, fail to fully fix the issue, or disrupt intended functionalities, thus becoming overfitting patches (Lutellier et al., 2019). This work investigates the usefulness of features proposed in previous works to tackle the problem of identifying correct patches among incorrect and plausible APR-generated patches. We first provide the necessary background on the used engineered and learned features.

3.1 Features

Generally, within machine learning, a feature represents a distinct and measurable attribute or characteristic of a phenomenon, while a feature vector refers to numbers (can be binary (0-1), integer or real) in an n-dimensional space, consisting of features (Seidel et al., 2017). Features are widely used in software engineering for diverse tasks including just-in-time quality assurance (Kamei et al., 2013), fault localization (Kim et al., 2019), vulnerability prediction (Dam et al., 2021) and others. In this work, we focus on optimal feature selection in the domain of Patch Correctness Check.

3.1.1 Hand-Crafted Features

Manually crafting features so that a classifier can prioritize correct patches over overfitting ones is not uncommon in the literature. We adapt the features of ssFix (Xin and Reiss, 2017b), S3 (Le et al., 2017) and CapGen (Wen et al., 2018), as their implementation is available and because these values are available in the seminal work of Wang *et al.* (Wang et al., 2021). The

Table 1: PCC features used in this study.

Category	Feature	Origin	Description	# dim
Hand-crafted	Token- Strct/Conpt	ssFix (Xin and Reiss, 2017b)	Structural & conceptual token similarity obtained from the buggy code and the generated patch.	2
	AST/Variable- Dist	S3 (Le et al., 2017)	Number of the AST changes and distances between the vectors representing AST patch nodes.	4
	Var/Syn/Sem- Simi	CapGen (Wen et al., 2018)	Similarity between variables / syntactic structures / contextual nodes affected by the change.	3
Engineered	Code descriptor	ODS (Ye et al., 2019)	Describe the characteristics of code elements (operators, variables, statements, AST operations)	155
	Repair pattern	ODS (Ye et al., 2019)	Repair patterns based on the work of Sobreira *et al.* (Sobreira et al., 2018) as binary features.	38
	Contextual Syntactic	ODS (Ye et al., 2019)	Describe the scope, parent and children's similarities of modified statements.	24
Dynamic	Embedding	Cache (Lin et al., 2022)	Dimensions of the embedded patch.	256
	Similarity	(Csuvik et al., 2020)	Similarity metrics measured between the embeddings of the patch and the original program.	8
				490

features introduced in S3 quantify both syntactic and semantic disparities between a candidate solution and the original buggy code. Subsequently, these features are used to prioritize and discern correct patches. In ssFix, authors employ token-based syntax representation of code to pinpoint syntax-related code fragments, aiming to generate accurate patches. CapGen has proposed three context-aware models - the genealogy model, variable model, and dependency model, respectively - to prioritize correct patches over overfitting ones.

3.1.2 Engineered Features

In this study, we used ODS (Ye et al., 2019) to extract metrics from the source code. It performs static analysis of the differences in AST between the buggy and patched programs; these differences are encoded as feature vectors. The authors grouped the ODS features into three categories: code description features, repair pattern features, and contextual syntactic features. We extracted these features using Coming [1], an open source commit analysis tool. Due to space limitations, we do not describe all of the features here (they are available in the original study), just an overview in Table 1.

3.1.3 Code Embeddings and Patch Similarity

Mapping source code into vector space can be beneficial in many ways, it can grasp aspects of the program that other metrics cannot. In the domain of patch correctness assessment, these techniques assess patch correctness by embedding token sequences extracted from the changed code of a generated patch (Lin et al., 2022). Then these embeddings can be used in various ways; we consider two scenarios in our study.

1. Use the generated embeddings directly, as if a dimension of the embedded vector is a feature. Thus, each dimension is treated as a metric measured on the patch.
2. Measure the vector distance between the embedded vector of the patch and the original program. The intuition here is that correct patches are usually *simpler* (i.e. more similar to the original program) than overfitting ones. Note that in previous studies (Csuvik et al., 2021) authors used a predefined threshold to classify patches as correct or overfitting ones, while in this study the threshold is learned dynamically.

One can use diverse embedding architectures, some of which are more resource intensive than others. In this study we used the Doc2Vec (Le and Mikolov, 2014) implemented in Gensim (Rehurek and Sojka, 2010). The choice is motivated by several factors: (1) ease of use and availability; (2) have history in the PCC domain (Csuvik et al., 2021) and (3) advanced embedding techniques did not surpass it by a large margin despite being more complicated and resource intensive (Csuvik et al., 2020). Doc2vec is derived from Word2vec (Mikolov et al., 2013), which is an artificial neural network, which can transform (*embed*) documents into vector space. The main idea of it is that the hidden layer of the network has fewer neurons than the input and output layers, thus forcing the model to learn a compact representation.

3.2 Dataset

Wang *et al.* (Wang et al., 2021) published a curated dataset comprising 902 labeled patches from Defects4J bugs, generated by 19 repair tools. We used this data set, which encompasses 654 patches labeled as overfitting and 248 patches labeled as correct by the respective authors. The dataset is still actively maintained, easily available, and popular to this day. In their online appendix [2], authors have also published the measured hand-crafted features on this dataset which we used directly from there. ODS features are calculated using their tool, while dynamic features are calculated using our implementation available in the attached repository (onl, 2024).

[1] https://github.com/SpoonLabs/coming

[2] https://github.com/claudeyj/patch_correctness/

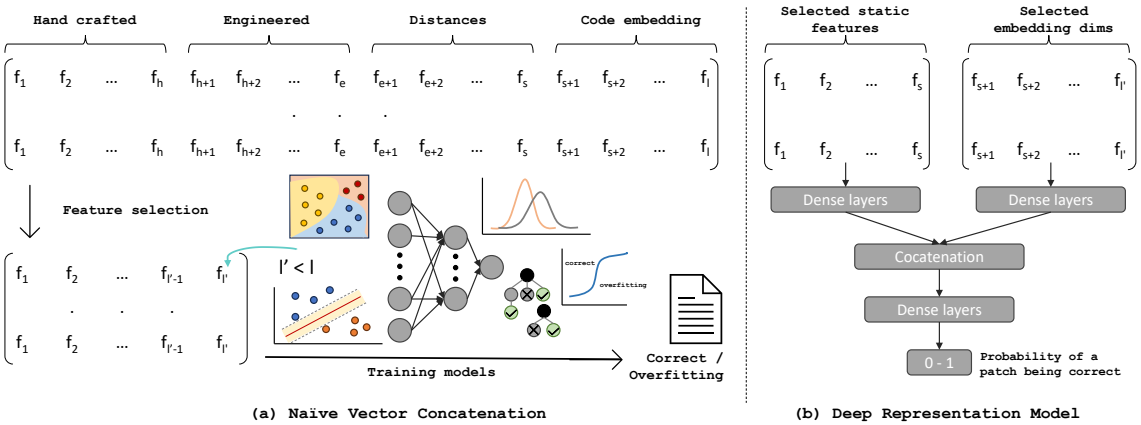

Figure 1: A high level overview of the used features and their optimization for PCC. On part (a) all features are concatenated then the most descriptive ones are selected to teach several ML models. On (b) static features (Hand-crafted, Engineered and Distances) and embeddings are first fed into dense layers and the neural network concatenates them, allowing it to learn a dynamic representation.

4 EXPERIMENT SETUP

Figure 1 shows a comprehensive overview of our study. Our goal is straightforward: find the set of features and ML models that most effectively detect overfitting patches. In Figure 1 (a) one can see that features are concatenated as is and then selected some of them, while (b) part depicts a deep representation model where features are fed into a neural architecture, allowing the net to learn the weights and biases of each feature. The obtained features from previous studies, described in Section 3.1, form a feature vector of l=*480* dimensions (composing of both static and dynamic features). Using feature selection techniques, we select l' features from these ($l' < l$) - the ones that best explain the input data. ML models are trained and evaluated on this subset to determine which yields the most optimal results. In our experiments we used a 32-bit Intel(R) Core(TM) i7-10510U CPU of 1.80GHz to train and evaluate each model and feature configuration. All of our code runs in Python 3.11.6, using the scikit-learn library version 1.4.0 (Pedregosa et al., 2011). The source code and detailed experimental data can be found in the online appendix of this paper (onl, 2024).

4.1 Feature Selection

Feature selection is the process of selecting a subset of relevant features to be used in model training (Sarangi et al., 2020). There are many available feature selection algorithms, from which we used the scikit-learn implementation of RFECV (RFECV documentation, 2024) to achieve the goal depicted in Figure 1 (a). It recursively eliminates features with cross-validation to select the most important features. The number of features selected is tuned automatically by fitting an RFE selector to the different cross-validation splits. We used a *RandomForestClassifier* as an estimator to provide information about feature importance mainly because it is a preferred model in previous PCC studies (Wang et al., 2021; Ye et al., 2019) and it has easily accessible coefficients required by the feature selection algorithm. Experiments were carried out in a *Stratified K-Fold* setting using 10 splits and the minimum number of features was required to be 10.

4.2 ML Models

The utilization of scikit-learn is motivated by its accessibility, robust performance, and inclusion of well-established, reliable models, facilitating the execution of our experiments with ease and efficacy. The following 9 models were used in part of our experiments (Figure 1 (a)): DecisionTreeClassifier, GaussianNB, KNeighborsClassifier, LinearDiscriminantAnalysis, LogisticRegression, MLPClassifier, RandomForestClassifier, SGDClassifier and SVC. The description each of these models can be found the official documentation of the scikit-learn library (Scikit-learn documentation, 2024). The concatenated features form the input for these models.

To further enhance the performance, in addition to built-in ML models, we built a neural network using Pytorch 2.1.2. It is able to combine features with learned embeddings as suggested by Tian *et al.* (Tian et al., 2023). The approach can be observed on Figure 1 (b). Note that this model operates on the al-

ready selected features, but treats embeddings dimensions and numeric features separately and concatenates them dynamically. The gist of this approach is that the neural architecture can learn the weighting of each feature and is able to inference more complex relations compared to naive concatenation. The implementation of this model is available in the online appendix of this paper (onl, 2024).

4.3 Evaluation

Previous studies have underscored the importance of PCC techniques in avoiding the dismissal of correct patches (as they are quite expensive to generate in the first place) (Yu et al., 2019). Consequently, we propose that a PCC technique is deemed effective if it produces minimal false positives while maintaining a high recall rate. To quantify our results, we introduce these metrics in the following:

- **T**rue **P**ositive: An overfitting patch is accurately identified.
- **F**alse **P**ositive: A correct patch identified as overfitting.
- **F**alse **N**egative: An overfitting patch identified as correct.
- **T**rue **N**egative: A correct patch identified as correct.

Using the above items, precision, recall, and F-meaure can be computed. Precision is the proportion between correctly classified overfitting patches among all the classified instances, while recall is the proportion between correctly classified overfitting patches and all relevant items. They are computed as:

$$precision = \frac{TP}{TP+FP} \qquad recall = \frac{TP}{TP+FN}$$

The F-measure can be defined by the two metrics above:

$$F_\beta = \frac{(\beta^2+1) * precision * recall}{\beta^2 * precision + recall}$$

β signifies the importance of precision or recall. If we want precision and recall to weigh in with exact the same importance, we simply assign the value 1 to β.

5 RESULTS

5.1 RQ1: Feature Selection

To examine the effect of each feature, we first carried out 10 independent trainings and executed the feature selection algorithm described in Section 4.1. The results of the feature selection can be observed in Table 2. In the table, the feature set which opts for the best results (cells of color gray) and the intersection of the 10 independent trainings (cells of color violet) are included. In the performed experiments, we found that the output of the feature selection algorithm varies greatly due to the effect of random factors

Table 2: Features selected using the RFECV algorithm: gray features that yield best performance for a single execution among the 10 feature selections. violet intersection between all of the features that were selected in the 10 feature optimization turns.

Hand-crafted	Engineered	Distances	Embeddings		
s3-tool	patchedFileNo	cosine_distance	vec_dim_0	vec_dim_62	vec_dim_182
AST-tool	addLineNo	braycurtis_distance	vec_dim_5	...	vec_dim_183
Cosine-tool	rmLineNo	canberra_distance	vec_dim_6	vec_dim_84	vec_dim_184
s3variable-tool	insertIfFalse	chebyshev_distance	vec_dim_7	...	vec_dim_185
variable-tool	updIfFalse	cityblock_distance	vec_dim_8	vec_dim_90	vec_dim_186
syntax-tool	ifFalse	euclidean_distance	...	...	...
semantic-tool	dupArgsInvocation	minkowski_distance	vec_dim_12	vec_dim_108	vec_dim_192
structural_score	removeNullinCond	seuclidean_distance	...	vec_dim_109	...
conceptual_score	condLogicReduce		vec_dim_21	vec_dim_110	vec_dim_200
	insertBooleanLiteral		...	vec_dim_111	...
	insertNewConstLiteral		vec_dim_27	...	vec_dim_208
	UpdateLiteral		...	vec_dim_130	vec_dim_209
	wrapsTryCatch		vec_dim_31	...	vec_dim_210
	...		vec_dim_32	vec_dim_144	vec_dim_211
	P4J_LATER_MEMBER_VF		...	vec_dim_145	...
	P4J_LATER_MODIFIED_SIMILAR_VF		vec_dim_43	...	vec_dim_220
	P4J_LATER_MODIFIED_VF		...	vec_dim_161	vec_dim_221
	P4J_LATER_NONZERO_CONST_VF		vec_dim_51	...	...
	P4J_LATER_OP_ADD_AF		vec_dim_52	vec_dim_167	vec_dim_225
	...		...	...	...
	S6_METHOD_THROWS_EXCEPTION		vec_dim_58		vec_dim_243
9 / 7 7	217 / 2 2	8 / 1 0	256 / 33 6	**Overall: 490 /**	**43 15**

during the training phase. Despite these differences, it is clear that most features can be opted out, and that hand-crafted ones form the most important subset of such features. It is also interesting to observe that some embedding dimensions hold more valuable information than others.

> ***Answer to RQ1.*** *Based on our experiments, on the used ML models and parameter configurations, some features are not beneficial - omitting these does not affect the results negatively, quite the opposite, precision and recall improved in best scenarios. Overall 43 features have been selected by a single run and 15 joint features have been identified across all the 10 independent trainings.*

To further investigate each feature subset, 9 classifiers have been trained on each. The results are listed in Table 3. What we can see is that the *MIN* and *MAX* values vary greatly in all of the feature sets. Despite the deviations, it is evident that some features can be opted out without any negative consequences and that on average the $\text{RFECV}_{\text{intersect}}$ yielded the best result in Precision, while $\text{RFECV}_{\text{best}}$ in Recall and F1 - thus it is more suitable for PCC. Certain embedding dimensions apear to contain valuable information; however, the model fails to encompass all necessary components. Sole reliance on embeddings led to a decrease in classifier performance. A future research dimension could be to investigate what (if any) embedding dimension is equivalent to which hand-crafted/engineered feature.

The selected engineered features `rmLineNo` and `P4J_LATER_NONZERO_CONST_VF` seem to grasp an important aspect of PCC, as these are selected in all feature selection attempts. Together with the hand-crafted features, these form the most essential part of the features. Table 3 supports this observation, as the $\text{Engineered}_{\text{plus}}$ subset yields only slightly lower F1 values than the optimized sets. However, it should be noted that Engineered features only bring an additional absolute growth of 1% on average, which can also be attributed to random factors. Random interplay is reflected in huge differences in performance in our experiments. This is not unique for PCC, but for example, if we consider the subset of the distance metrics, it can be seen that in the worst-case scenario it achieved 0% precision, while on the best case 100% precision (but on average quite moderate). We did not explore the random effects on the embedding model but hypothesize that they might have a similar impact. By selecting alternative features, not limited to embeddings and distances, one may potentially mitigate this effect.

Table 3: Measures on various feature subsets.

		F1	Prec	Recall
MIN	All	.62	.65	.59
	$\text{RFECV}_{\text{best}}$	**.65**	**.69**	.59
	$\text{RFECV}_{\text{intersect}}$	.58	.58	.50
	Distances	.00	.00	.00
	Embeddings	.54	.51	.45
	Engineered	.56	.52	.55
	$\text{Engineered}_{\text{plus}}$	.63	.63	**.61**
	Hand-crafted	.17	.35	.10
MEAN	All	.80	.81	**.82**
	$\text{RFECV}_{\text{best}}$	**.81**	.84	.78
	$\text{RFECV}_{\text{intersect}}$	.80	**.85**	.76
	Distances	.18	.59	.11
	Embeddings	.70	.69	.71
	Engineered	.76	.76	.76
	$\text{Engineered}_{\text{plus}}$	.77	.77	.78
	Hand-crafted	.42	.57	.35
MAX	All	**.92**	.96	.96
	$\text{RFECV}_{\text{best}}$	.91	**1.00**	**.97**
	$\text{RFECV}_{\text{intersect}}$	.91	**1.00**	.90
	Distances	.41	**1.00**	.31
	Embeddings	.81	.88	.90
	Engineered	.89	.95	.93
	$\text{Engineered}_{\text{plus}}$	.90	.96	.93
	Hand-crafted	.59	.89	.61

The grouped rows (*MIN, MEAN, MAX*) indicate the minimum, average and maximum values of each metric we used in the 10 independent trainings. Each subset contains the followings: *All* (all 490 features), $\textit{RFECV}_{\textit{best}}$ (43 features from the feature selection algorithm), $\textit{RFECV}_{\textit{intersect}}$ (the 15 joint feature that were selected in all 10 runs), *Distances* (the 8 distance metrics), *Embeddings* (256 dimension of the embedded code vectors), *Hand-crafted* (9 hand-crafted features from previous studies), *Engineered* (217 feature from ODS) and $\textit{Engineered}_{\textit{plus}}$ (static features comprising of hand-crafted and engineered ones -$\textit{Engineered}_{\text{plus}} = \textit{Engineered} \cup \textit{Hand} - \textit{crafted}$).

5.2 RQ2: Model Selection

Prior studies on PCC (Ye et al., 2019; Wang et al., 2021) have exhibited a predilection for Random Forest as the classifier of choice. Additionally, it has been widely adopted in addressing various Software Engineering-related issues due to its demonstrated efficacy across diverse tasks (Bludau and Pretschner, 2022; Bowes et al., 2018). These experiences drove our intuition to use Random Forest in feature selection, but it also raises the question of whether other classifiers might outperform it. In the subsequent experiment, we trained 9 classifiers 10 times each to obtain the results presented in Table 4. We used the $\text{RFECV}_{\text{best}}$ feature set obtained in the previous sec-

tion on all observed models. What we can see is that on average the MLPClassifier is the most harmonic: the F1 metric reaches highest values here on average. Also, apart from the GaussianNB classifier (which is insufficient in terms of Precision), MLP provides the highest Recall values.

> ***Answer to RQ2.*** *While GaussianNB consistently produced the highest Recall values, its efficacy in precisely detecting overfitting patches appears inefficient. On the other hand, as previous studies suggested, RandomForest consistently provides reliable results, however, our findings indicate that MLPClassifier outperforms it by a small margin.*

Relying only on the *MAX* values would flaw the findings of our paper, thus we try to see the whole picture and are looking for a classifier that works well in real life scenarios most of the time (even in the worst case). What we can see in Table 4 is that the MLPClassifier performs reasonably well compared to other models in the *MIN* case, also. On the other hand, the motivation behind the use of Random Forest is understandable: it provides a well-explainable output with moderate training costs. The benefit of the MLPClassifier might lie in its flexibility, as the Multilayer Perceptron is a built-in model within scikit-learn with limited possibilities for customization, building a neural network from scratch and including domain-specific knowledge might add additional value to this model. While in this section we treated every feature equally and combined them naively (i.e. forming a feature vector which includes all the features of a subset), in the following we explore the possibility to combine the selected features dynamically by expanding the MLPClassifier and implementing a Neural Network in Pytorch.

5.3 RQ3: Potential Improvements

As depicted in Figure 1 (b) and described in Section 4.2, we further try to enhance the performance. In the previous experiments we already identified the $\text{RFECV}_{\text{best}}$ feature set as the 43 features worth training on and the MLPClassifier due to its flexible nature and reliable outcomes. A Neural Network has been constructed that learns a deep representation of the input features; the measured results are shown in Table 5. Due to space limitations the exact architecture is not detailed here, but the interested Reader can find it in the online appendix (onl, 2024). The network has been trained using k-fold cross validation, when the validation loss became smaller than a certain epsilon, the training process was stopped (early stopping) - thus reducing the chance of overfitting. Having a Neural Network gives the possibility to weight input features - apart from filtering unnecessary features out this approach can give different weights to features depending on how important they are. Similarly to the previous experiments, the model was trained and evaluated 10 times, thus we display the *MIN*, *MEAN*, and *MAX* values of each metric. It is evident that the metric values did not improve on average (or at least not significantly, which cannot be attributed to random factors). On the other hand, the stability of the approach improved: the previous absolute deviation of 30% in the F1 score has been reduced to 16%, thus making the model much more reliable than before. These ML predictors are complementary to other state-of-the-art methods and similar to them in filtering out patches generated by APR tools (Tian *et al.* (Tian et al., 2023) 79%, Wang *et al.* (Wang et al., 2021) 87% F1 score).

Ensemble Learning (Polikar, 2012) and Majority Voting (Penrose, 1946) are both techniques employed

Table 4: Evaluation of the $\text{RFECV}_{\text{best}}$ feature set on 9 ML classifiers.

		F1	Prec	Recall
MIN	DecisionTree	.46	.45	.45
	GaussianNB	.44	.30	**.82**
	KNeighbors	.45	.43	.46
	LDA	.52	.46	.54
	LogRegression	**.62**	.59	.62
	MLPClassifier	**.62**	.65	.59
	RandomForest	.61	**.70**	.48
	SGDClassifier	.55	.53	.52
	SVC	.60	.60	.52
MEAN	DecisionTree	.64	.64	.65
	GaussianNB	.51	.35	**.94**
	KNeighbors	.70	.69	.71
	LDA	.67	.60	.76
	LogRegression	.74	.69	.79
	MLPClassifier	**.80**	.81	.82
	RandomForest	.77	**.86**	.71
	SGDClassifier	.71	.68	.75
	SVC	.76	.74	.79
MAX	DecisionTree	.79	.81	.89
	GaussianNB	.56	.39	**1.00**
	KNeighbors	.83	.85	.90
	LDA	.80	.72	.97
	LogRegression	.88	.89	**1.00**
	MLPClassifier	**.92**	.96	.96
	RandomForest	.91	**1.00**	.93
	SGDClassifier	.84	.89	.93
	SVC	.89	.88	.97

Table 5: Evaluation of Deep Representation Learning.

	F1	Precision	Recall
MIN	72.73%	69.70%	68.97%
MEAN	81.92%	80.77%	83.68%
MAX	88.89%	91.67%	96.55%

in ML to enhance predictive performance by combining the outputs of multiple (usually weak) individual models. Through the aggregation of predictions, majority voting leverages the collective wisdom of diverse models to make decisions. As decision in a Random Forest is obtained by majority voting of the individual trees, it alone can be treated as a Majority Voting approach. However, several recent approaches integrate the learned models either by Ensemble Learning or by Majority Voting strategies. To investigate the performance of such approaches, we also combined the output of the nine observed ML models by weighting their output. We call this method *stacking*, and it is based on a StackingRegressor (Scikit-learn StackingRegressor, 2024). Stacked generalization consists of stacking the output of individual estimator and using a regressor to compute the final prediction.

The results of this approach can be observed in Table 6. Stacking allows us to use the strength of each individual estimator by using their output as input of a final estimator. Although the F1 score and Precision improved on average, Recall decreased making this method unsuitable for PCC. Another unfavorable inspection is that the deviation of all three metrics has doubled compared to the previous measurement. During the experiment, we noticed that results are close to the ones obtained with MLP. After further investigation we found that the algorithm assigns most of its weights to the MLP (on average 26%), Random Forest (38%) and SVC (32%) classifiers and relies only negligibly on other models. The implication of this observation suggests that machine learning models exhibit equal confusion regarding the remaining incorrectly classified samples, whereas the correctly classified examples are largely identical. The underlying reason might be data quality, inaccurate oracle (human error on classification), imbalanced data, suboptimal network architecture and parameterization, etc. Investigating such a problem might be a separate research direction for future work.

Table 6: Results showcasing the stacked performance of the 9 ML models.

	F1	Precision	Recall
Min	64.15%	66.67%	58.62%
Mean	82.38%	85.83%	79.68%
Max	93.10%	96.30%	93.10%

Answer to RQ3: *Improvement in stability can be achieved to a certain extent; however, the improvements may not suffice to ensure consistently reliable outcomes. Both Deep Representation Learning and Stacking failed at improving filtering out overfitting patches, although the former yields similar results with a more reliable standard deviation.*

6 DISCUSSION AND THREATS TO VALIDITY

In this study, our primary emphasis lay in optimizing the features utilized within machine learning models, with a secondary focus on enhancing performance and stability through deep learning techniques. We only examined the domain of PCC, however, this might be a general problem in Software Engineering research using ML. As we have seen, features significantly influence both the performance and stability of the applied machine learning model; however, careful construction of a neural architecture may also enhance stability. Through the application of our method, we contend that improved practices can be established for the publication of machine learning applications and the assessment of their stability in APR and also in the wider domain of Software Engineering.

An interesting insight arises from the study of Wang *et al.* (Wang et al., 2021), where they achieved 87.01% Precision and 89.14% Recall using only Hand-crafted features and a Random Forest classifier, contrasting sharply with our own results of 57% Precision and 35% Recall on average using the same features. While these figures closely approximate Wang *et al.* 's results under the best-case scenario (MAX), they remain unreproducible. Another factor to consider is the choice of library; the aforementioned article utilized the Weka app (Weka Webite, 2024), which, by its graphical interface, inadvertently undermines reproducibility, unlike our use of the scikit-learn library. Reproducibility can be significantly improved by sharing a reproduction package; however, the applicability of the proposed model remains limited without evaluating its stability. Notably, the selected feature set $\text{RFECV}_{\text{best}}$ yields more reliable results than previous iterations, and the constructed neural networks contribute to the stability and reproducibility of our study. Additionally, the online appendix of this paper offers full reproducibility of the experiments conducted (onl, 2024).

A possible threat to the validity of our research re-

lates to the benchmark we used. We only utilized the dataset of Wang *et al.* (Wang et al., 2021) for evaluation; however it is the most complete dataset, to the best of our knowledge, in the literature for conducting such studies. We believe that this choice does not impact the assessment of feature selection. Another threat might be the effect of random initialization, which can have a huge impact on the resulting model. By training the models in a cross-validation setting 10 times with different random seeds and averaging the results, we tried to mitigate this effect. Due to space constraints, we did not include the output of all the trainings, settings, and detailed experiments that were executed in the paper. However, our online appendix package (onl, 2024) ensures full reproducibility and detailed data on the research conducted.

7 RELATED WORK

Evaluating existing APR approaches is crucial, but assessing APR tools solely on plausible patches is inaccurate due to the overfitting issue inherent in test suite-based automatic patch generation. Identifying the correct patches among plausible ones requires additional developer effort. Recently several approaches have been proposed to tackle the problem of Patch Correctness Check (Yang et al., 2023).

Feature-Based PCC
Liu *et al.* (Liu et al., 2021b) proposes eight evaluation metrics for fairly assessing the performance of APR tools in addition to providing a critical review on the existing evaluation of patch generation systems. Authors in (Wang et al., 2021) benchmark the state-of-the-art patch correctness techniques based on the largest patch benchmark so far and gather the advantages and disadvantages of existing approaches. Tian *et al.* (Tian et al., 2020) use Doc2Vec, Bert, code2vec and CC2Vec to investigate the discriminative power of features. They claim that Logistic Regression with BERT embedding scored 0.72% F-Measure and 0.8% AUC on labeled dataset of 1,000 patches. Opad (Yang et al., 2017) (Overfitted Patch Detection) is another tool which aims to filter out incorrect patches. Opad uses fuzz testing to generate new test cases and employs two test oracles to enhance the validity checking of automatically generated patches. Anti-pattern based correction check is also a viable approach (Tan et al., 2016). Syntactic or semantic metrics such as cosine similarity and output coverage (Le et al., 2017) can also be applied to measure similarity, like in the tool named Qlose (D'Antoni et al., 2016). A recent study (Ismayilzada et al., 2023) presents a new lightweight specification method that enhances failing tests with preservation conditions, ensuring that patched and prepatched versions produce identical outputs under specific conditions. It also introduces a differential fuzzer for efficient patch classification, surpassing four state-of-the-art approaches, with a user study showing preference for the semi-automatic patch assessment method over manual assessment.

Dynamic PCC
Numerous studies (Mechtaev et al., 2015; White et al., 2019; Tufano et al., 2019) emphasize the importance of simplifying the generated repair patches. A recent study (Wang et al., 2019) found that 25.4% (45/177) of correct patches generated by APR techniques differ syntactically from those provided by a developer. Other methods, such as learning from human-written code (Le et al., 2016; Kim et al., 2013), have shown promise but have faced recent criticism (Monperrus, 2014). In other works (Csuvik et al., 2021) candidate patches were ranked according to their similarity to the original program and assessed as a recommendation system. Others have also used embedding techniques, but not only focusing on the changed code, but also taking into consideration the unchanged correlated part (Lin et al., 2022) by measuring the similarity between the patched method name and the semantic meaning of body of the method (Phung et al., 2022) or based their approach onthe fact that similar failing test cases should require similar patches (Tian et al., 2022). The reliability of automated annotations for patch correctness has also been proposed (Le et al., 2019). Authors compared them with a gold standard of correctness labels for 189 patches, finding that although independent test suites may not suffice as effective APR oracles, they can augment author annotations. Meanwhile, Xiong *et al.* (Xiong et al., 2018) proposed leveraging behavior similarity in test case executions to determine correct patches. By improving test suites with new inputs and using behavior similarity, they prevented 56.3% of incorrect patches from being generated. Ortin *et al.* (Ortin et al., 2020) achieved better performance than the existing systems for source code classification using embeddings, even though it has not been used for APR. Another work that is highly utilized these days in ML for source code is CodeBERT, which has been employed in many different domains, including APR (Feng et al., 2020).

This Study
In this work, we mainly focused on the optimization of the features used in ML models and partly with enhancing the performance with deep learning. We

used data from the study of Wang *et al.* (Wang et al., 2021) and the engineered features from ODS (Ye et al., 2019). Our work is fundamentally different from these: (1) we do not concentrate on the definition of features but rather use existing ones, (2) all available features are treated as complements to each other, (3) the goal was to achieve a cross-research feature set which is the most optimal to PCC. In a recent study, Tian *et al.* (Tian et al., 2023) already proposed deep-combination of features, but their approach (1) does not apply feature selection, (2) no diverse ML model training is carried out, and (3) the used static feature set is not as thorough as the ones we have presented. We also publish the training dataset along with the fixed seeds and the source code of the experiments (which is only partially true for previous studies).

8 CONCLUSIONS

In this study, we acquired 490 features, comprising both engineered features and code embeddings, to address Patch Correctness Check. Initially, a feature selection algorithm was used to extract 43 features from the extensive feature set, indicating the limited informational contribution of most original features. Subsequently, we conducted training and evaluation of nine machine learning models to discern the optimal performer. To counteract random factors, each model underwent 10 training iterations using different random seeds. Our findings suggest better performance of the models on average when utilizing the selected feature set in comparison to the entire feature set or other subsets. Among the models examined, Multi-Layer Perceptron (MLP) and Random Forest consistently exhibited the most reliable results, achieving average F1 scores of 0.8 and 0.77, respectively. However, due to random factors, the MLP score fluctuated to 0.62 in unfavorable cases or peaked at 0.92 in fortuitous circumstances. Employing a more complex neural architecture that integrates learned embeddings with other features enabled us to mitigate this variability, reducing the absolute fluctuation in the F1 score from 30% to 16%.

Our research underscores two major implications. First, the development of PCC classifiers requires careful planning of both feature selection and model construction; While hand-crafted features remain paramount, embeddings may also contain useful information. Second, machine learning methodologies must prioritize model stability, as it profoundly influences the validity and significance of results.

ACKNOWLEDGEMENTS

The research presented in this paper was supported in part by the ÚNKP-23-3-SZTE-435 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund, and 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 this work. Project no TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

REFERENCES

(2024). Supplemental material for "feature extraction, learning and selection in support of patch correctness assessment". https://github.com/AAI-USZ/PCC-2024.

Bludau, P. and Pretschner, A. (2022). Feature sets in just-in-time defect prediction: an empirical evaluation. *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*.

Bowes, D., Hall, T., and Petrić, J. (2018). Software defect prediction: do different classifiers find the same defects? *Software Quality Journal*, 26(2):525–552.

Csuvik, V., Horvath, D., Horvath, F., and Vidacs, L. (2020). Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 18–25. IEEE.

Csuvik, V., Horváth, D., Lajkó, M., and Vidács, L. (2021). Exploring plausible patches using source code embeddings in javascript. *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 11–18.

Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., and Ghose, A. (2021). Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 47(1):67–85.

D'Antoni, L., Samanta, R., and Singh, R. (2016). QLOSE: Program repair with quantitative objectives. Technical report.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Liu, T., Jiang, D., and Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages.

Ismayilzada, E., Rahman, M. M. U., Kim, D., and Yi, J. (2023). Poracle: Testing patches under preservation conditions to combat the overfitting problem of program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(2).

Just, R., Jalali, D., and Ernst, M. D. (2014). 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.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., and Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773.

Kechagia, M., Mechtaev, S., Sarro, F., and Harman, M. (2022). Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, 48(7):2658–2679.

Kim, D., Nam, J., Song, J., and Kim, S. (2013). Automatic patch generation learned from human-written patches. In *Proceedings - International Conference on Software Engineering*, pages 802–811. IEEE.

Kim, Y., Mun, S., Yoo, S., and Kim, M. (2019). Precise learn-to-rank fault localization using dynamic and static features of target programs. *ACM Trans. Softw. Eng. Methodol.*, 28(4).

Le, D. X. B., Bao, L., Lo, D., Xia, X., Li, S., and Pasareanu, C. (2019). On Reliability of Patch Correctness Assessment. In *Proceedings - International Conference on Software Engineering*, volume 2019-May, pages 524–535. IEEE Computer Society.

Le, Q. V. and Mikolov, T. (2014). Distributed Representations of Sentences and Documents. Technical report.

Le, X.-B. D., Chu, D.-H., Lo, D., Le Goues, C., and Visser, W. (2017). S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 593–604, New York, NY, USA. Association for Computing Machinery.

Le, X. B. D., Lo, D., and Goues, C. L. (2016). History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE.

Lin, B., Wang, S., Wen, M., and Mao, X. (2022). Context-aware code change embedding for better patch correctness assessment. *ACM Trans. Softw. Eng. Methodol.*, 31(3).

Liu, C., Gao, C., Xia, X., Lo, D., Grundy, J., and Yang, X. (2021a). On the reproducibility and replicability of deep learning in software engineering. *ACM Trans. Softw. Eng. Methodol.*, 31(1).

Liu, K., Li, L., Koyuncu, A., Kim, D., Liu, Z., Klein, J., and Bissyandé, T. F. (2021b). A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817.

Lutellier, T., Pang, L., Pham, V. H., Wei, M., and Tan, L. (2019). ENCORE: Ensemble Learning using Convolution Neural Machine Translation for Automatic Program Repair.

Mechtaev, S., Yi, J., and Roychoudhury, A. (2015). DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 448–458. IEEE.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. Technical report.

Monperrus, M. (2014). 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. ACM Press.

Ortin, F., Rodriguez-Prieto, O., Pascual, N., and Garcia, M. (2020). Heterogeneous tree structure classification to label java programmers according to their expertise level. *Future Gener. Comput. Syst.*, 105(C):380–394.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Penrose, L. S. (1946). The elementary statistics of majority voting. *Journal of the Royal Statistical Society*, 109(1):53–57.

Phung, Q.-N., Kim, M., and Lee, E. (2022). Identifying incorrect patches in program repair based on meaning of source code. *IEEE Access*, 10:12012–12030.

Polikar, R. (2012). Ensemble learning. *Ensemble machine learning: Methods and applications*, pages 1–34.

Rehurek, R. and Sojka, P. (2010). Software Framework for Topic Modelling with Large Corpora. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50.

RFECV documentation (2024). Rfecv documentation. https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html.

Sarangi, S., Sahidullah, M., and Saha, G. (2020). Optimization of data-driven filterbank for automatic speaker verification. *Digital Signal Processing*, 104:102795.

Scikit-learn documentation (2024). Scikit-learn documentation. https://scikit-learn.org/stable/user_guide.html.

Scikit-learn StackingRegressor (2024). Scikit-learn stackingregressor. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingRegressor.html.

Seidel, E. L., Sibghat, H., Chaudhuri, K., Weimer, W., and Jhala, R. (2017). Learning to blame: localizing novice type errors with data-driven diagnosis. *Proc. ACM Program. Lang.*, 1(OOPSLA).

Sobreira, V., Durieux, T., Delfim, F. M., Martin, M., and de Almeida Maia, M. (2018). Dissection of a bug dataset: Anatomy of 395 patches from defects4j. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 130–140.

Tan, S. H., Yoshida, H., Prasad, M. R., and Roychoudhury, A. (2016). 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. ACM Press.

Tian, H., Li, Y., Pian, W., Kaboré, A. K., Liu, K., Habib, A., Klein, J., and Bissyandé, T. F. (2022). Predicting patch correctness based on the similarity of failing test cases. *ACM Trans. Softw. Eng. Methodol.*, 31(4).

Tian, H., Liu, K., Kaboré, A. K., Koyuncu, A., Li, L., Klein, J., and Bissyandé, T. F. (2020). 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.

Tian, H., Liu, K., Li, Y., Kaboré, A. K., Koyuncu, A., Habib, A., Li, L., Wen, J., Klein, J., and Bissyandé, T. F. (2023). The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches. *ACM Trans. Softw. Eng. Methodol.*, 32(4).

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. (2019). 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.

Wang, S., Wen, M., Chen, L., Yi, X., and Mao, X. (2019). 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-Septe. IEEE Computer Society.

Wang, S., Wen, M., Lin, B., Wu, H., Qin, Y., Zou, D., Mao, X., and Jin, H. (2021). Automated patch correctness assessment: how far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 968–980, New York, NY, USA. Association for Computing Machinery.

Weka Webite (2024). Weka webite. https://www.weka.io.

Wen, M., Chen, J., Wu, R., Hao, D., and Cheung, S.-C. (2018). Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1–11, New York, NY, USA. Association for Computing Machinery.

White, M., Tufano, M., Martinez, M., Monperrus, M., and Poshyvanyk, D. (2019). Sorting and transforming program repair ingredients via deep learning code similarities. In Wang, X., Lo, D., and Shihab, E., editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 479–490. IEEE.

Xin, Q. and Reiss, S. P. (2017a). Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 226–236, New York, NY, USA. Association for Computing Machinery.

Xin, Q. and Reiss, S. P. (2017b). Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670.

Xiong, Y., Liu, X., Zeng, M., Zhang, L., and Huang, G. (2018). Identifying patch correctness in test-based program repair. In *Proceedings - International Conference on Software Engineering*, pages 789–799. IEEE Computer Society.

Yang, J., Wang, Y., Lou, Y., Wen, M., and Zhang, L. (2023). A large-scale empirical review of patch correctness checking approaches. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1203–1215, New York, NY, USA. Association for Computing Machinery.

Yang, J., Zhikhartsev, A., Liu, Y., and Tan, L. (2017). 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.

Ye, H., Gu, J., Martinez, M., Durieux, T., and Martin, M. (2019). Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48:2920–2938.

Ye, H., Martinez, M., and Monperrus, M. (2021). Automated patch assessment for program repair at scale. *Empirical Softw. Engg.*, 26(2).

Yu, Z., Martinez, M., Danglot, B., Durieux, T., and Monperrus, M. (2019). Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system. *Empir. Softw. Eng.*, 24(1):33–67.