Learning Based Patch Overfitting Detection: A Survey

Xuanyan Li[1], Dongcheng Li[2], Man Zhao[1], W. Eric Wong[3], Hui Li[1]*

[1] School of Computer Science, China University of Geosciences, China
[2] Department of Computer Science, California State Polytechnic University - Humboldt, USA
[3] Department of Computer Science, University of Texas at Dallas, USA
283735194@qq.com, dl313@humboldt.edu, zhaoman@cug.edu.cn, ewong@utdallas.edu, huili@vip.sina.com

Abstract

As the complexity of software increases, the requirements for software quality and security continue to increase. However, automatically generated fix patches may suffer from patch overfitting issues, leading to instability and security vulnerabilities. This paper aims to summarize the application of machine learning in detecting overfitting problems in automatic program repair (APR) patches. We review the current state of research on automatic program repair and overfitting detection, and we summarize the machine learning techniques employed. We identified a comprehensive list of available datasets and metrics commonly used in this research. Finally, this paper discusses the challenges faced in this field and systematically summarizes the application of machine learning in detecting patch overfitting problems in automatic program repair, providing a useful reference for future research.

Keywords: Overfitting problem, Automated Patch Correctness Assessment (ACPA), Patch, Automated Program Repair (APR), Machine Learning (ML)

1 Introduction

Automated program repair is a highly researched field in computer software engineering. With the increasing complexity of industrial and internet software, the demands for software quality and security have also risen. Vulnerabilities and errors in software are inevitable, leading to the flourishing development of automatic program repair technologies. In recent years, researchers have proposed various techniques and tools for automated program repair, including Nopol, Evosuite, Randoop, Genprog, and others [1-5]. The current techniques for generating APR patches can be categorized into two main types: those based on semantic constraints and those based on heuristic search. There are also approaches based on constraints, templates, and deep learning [6]. However, the development of automatic program repair technologies has raised concerns about overfitting issues in the generated patches, leading to instability and security vulnerabilities [7]. This overfitting problem manifests as patches that may not be sufficiently universal and effective only for specific inputs. Existing APR technologies commonly exhibit this overfitting problem [8].

In past research, automated program repair and overfitting detection have been extensively discussed in the Automated Patch Correctness Assessment (APCA) domain. Most studies have focused on using traditional dynamic and static ACPA techniques, utilizing test cases, or manually marking patches. Dynamic methods rely on runtime information to determine patch correctness, which may result in lower performance. Static methods analyze patch code fragments to infer patch correctness based on program syntax, although they have higher efficiency, their accuracy is comparatively lower [9]. Notably, the Invalidator method is based on semantics and syntax to infer and automatically evaluate patch correctness. It uses a trained model on labeled patches to assess the correctness based on program syntax, evaluating patches' correctness that cannot be determined by invariants [10]. The DiffTGen method uses generated new test inputs to discover semantic differences between the original faulty program and the patched program, ultimately identifying whether the generated patch is overfitting. Generating new inputs using test cases can add them to the test suite and prevent automatic program repair technologies from generating similar overfitting patches again. However, this method may have issues such as result bias and performance deficiencies when compared to manual marking methods [11].

Researchers and engineers have also proposed using machine learning model techniques to continuously explore how to reduce the risk of patch overfitting in APR. However, despite significant progress in these studies, there are still key issues to be addressed, such as how to more effectively detect and mitigate overfitting issues in automatic program repair. Recently proposed methods to address this critical issue are numerous, but there is still a lack of an in-depth review of the latest developments in learning-based methods for patch overfitting detection. The purpose of this paper is to summarize and classify learning-based patch overfitting detection methods, outline existing research achievements and methods, and future directions to better understand and address this issue, assisting relevant research. We focus on Defects4j, QuixBugs, Bears, Bugs.jar, ManySStuBs4J and Codeflaws datasets that have been used recently by relevant researchers, making it easier for other researchers to start research

*Corresponding Author: Dongcheng Li; E-mail: dl313@humboldt.edu
DOI: https://doi.org/10.70003/160792642025012601005

without having to mix all available datasets. Additionally, our review outlines some traditional techniques and popular learning-based methods for patch overfitting detection, including comparisons of their methods and detailed analyses of each component of the learning-based patch overfitting detection process. The contributions of this paper are as following:

(1) Learning based patch overfitting detection. We systematically review learning-based patch overfitting detection, covering the most popular methods based on convolutional neural networks, generative adversarial networks, and methods to mitigate overfitting issues through automatically generated test cases for comprehensive repair techniques.

(2) Technologies and datasets. We provide detailed insights into the relevant technologies of patch overfitting detection and popular datasets for patch overfitting detection.

(3) Metrics. We thoroughly investigate how to effectively utilize machine learning methods to detect patch overfitting issues and representative evaluation metrics for this task.

(4) Outlook and challenges. We propose some suggestions for future research using learning-based methods for patch overfitting detection. The main goal of this paper is to survey and summarize the progress of these learning methods and related research, providing a reference for future studies.

The paper is structured as follows:

Firstly, Section 2 introduces the core concepts and most commonly used strategies of automated program repair patches, patch overfitting, and machine learning-related methods. Then, in Section 3, we review past research on utilizing machine learning methods to detect and mitigate overfitting issues in automatic program repair patches, analyzing shortcomings in each study. Subsequently, in Section 4, we list publicly available datasets and related evaluation standards. Finally, in Section 5, we outline the conclusions of the review and provide some promising directions for further research.

2 Background and Concepts

In this section, we will introduce some background information and common concepts in the field of patch overfitting detection based on learning technology in the rough order in which the background and concepts appear. The first is automatic program repair and patching, and the second is the overfitting problem associated with it. Next, we delve into APCA techniques for solving these problems, concluding with an introduction to machine learning and patch overfitting detection.

2.1 Automated Program Repair and Patch

Automated Program Repair (APR) aims to enhance software quality and security by automatically identifying and correcting issues in software code. APR techniques primarily use test cases with faulty inputs to induce program failures, proving the existence of faults. Patches, in the context of APR, refer to the software code or updates generated during the process of repairing, updating, or improving a software program. Different APR tools employ various methods and techniques to generate patches [12].

Automated Program Repair typically consists of three parts:

Employing existing fault localization techniques to identify code segments with bugs, modifying these segments based on a set of transformation rules or patterns to generate various program variants (candidate patches) [13], and validating all candidate patches using the original test suite as an oracle [14]. Patch generation results are constrained by the effectiveness of the first part, fault detection, and localization. Accurate detection and localization of errors are crucial for generating effective repair patches, ensuring the resolution of issues in the software [15].

2.2 Overfitting Problem

The overfitting problem emerged in parallel with the development of deep learning, discussed in various research papers and projects [16]. Overfitting fundamentally involves using models or programs that violate simplicity, incorporating more terms than necessary or utilizing more complex methods. Therefore, overfitting is undesirable for both technology and methods [17].

The patch overfitting problem occurs when automated program repair systems, attempting to fix errors in software, generate repair patches that excessively adapt to specific training data, resulting in poor generalization performance on other data [18]. In other words, the repaired program may appear effective for specific defects but overly accommodates specific test cases used during the creation process [19]. Specifically, this means that the generated repair patches may perform well on some inputs but may fail to function correctly on others. The patch overfitting problem reflects the instability and lack of robustness of automated program repair systems [18]. This issue in the context of automatic program repair applies research concepts from the deep learning and neural network domains regarding model generalization, overfitting, and uneven data distribution [6].

The occurrence of the patch overfitting problem is closely related to multiple factors: Automated Program Repair systems typically use open-source code from software repositories or version control systems as training data. This data often originates from various projects and code repositories, possessing different code styles and qualities. In such cases, the distribution of training data may be uneven, with the volume of data from certain projects or domains far exceeding others [20]. This can lead to automated program repair systems performing well in some domains but exhibiting poor performance in others.

Training data for automated program repair may contain noise and errors. These errors may result from incorrect error reports or faulty repair attempts. When automated program repair systems learn these errors from training data, they can contribute to the patch

overfitting problem. Additionally, due to manual labeling by developers, label errors may exist in the training data, causing biases in model learning. Automated program repair systems must search a vast repair space, which is often extensive. Due to the complexity of the search space, systems may tend to generate repairs specific to certain domains while performing poorly in other domains. This makes automated program repair systems prone to the patch overfitting problem [21].

Automated program repair systems often rely on a series of machine learning algorithms and hyperparameter settings for learning and generating repair patches. Different algorithm and parameter choices may lead to different overfitting tendencies [22]. Therefore, algorithm selection and hyperparameter settings are also factors contributing to the patch overfitting problem. The patch overfitting problem significantly impacts the performance and usability of automated program repair systems.

2.3 APCA (Automated Patch Correctness Assessment)

Automated Patch Correctness Assessment (APCA) refers to the automated process of evaluating and determining the correctness of patches generated by Automated Program Repair (APR) techniques. In the APR environment, which involves automatically fixing software errors, APCA plays a crucial role in determining the accuracy, effectiveness, and alignment with the expected functionality of the generated patches [23]. Through automated patch correctness assessment, APCA helps minimize both false positives (misidentifying correct patches) and false negatives (failing to identify incorrect patches), which is crucial for enhancing the reliability of automated patching systems [24]. Researchers address the challenge of overfitting patches, which are often caused by insufficient test suites from real-world programs. Some propose simple methods utilizing automated test generation tools to check for patch overfitting. If a seemingly reasonable patch fails in any of these test cases, it is flagged as overfitting [19]. Current APCA methods and techniques can be broadly categorized into static and dynamic approaches:

•Static Techniques

Static techniques operate on static code patterns or features, using tools like static analyzers or code style checkers to inspect changes introduced by patches [15]. Engineering features involve selecting the most relevant ones, dimensionality reduction to eliminate less contributive features, and regularization to control model complexity. Static techniques encompass checking for common programming errors, adhering to coding standards, and identifying potential side effects. Avoiding overly complex models reduces the risk of overfitting on training data, ensuring patches execute correctly across various possible execution paths. For example, CapGen introduces context-aware repair generation technology, reducing the likelihood of generating incorrect viable patches, mitigating overfitting [25]. s3 integrates syntax-guided and semantic-guided approaches, using six features (AST differencing, Cosine similarity, Locality of variables and constants, Model counting, Output coverage, Anti-patterns) to measure the syntactic and semantic distance between candidate solutions and original error code. These features determine priority and identify correct patches [26]. ssFix filters out syntactically redundant and previously tested patches (generated by other chunks), ranks patches based on three specified rules regarding modification types and sizes, and tests and validates to identify overfitting patches [27].

•Dynamic Techniques

Dynamic techniques require running tests during program execution, analyzing the behavior of repaired code on different execution paths [28]. Using dynamic input generation technology tests the behavior of the repaired program with diverse input data, ensuring the patch works correctly and robustly. Monitoring the repaired code during program runtime captures potential exceptions, errors, or performance issues. Real-time feedback helps identify issues that repaired patches may encounter during runtime. For example, DiffTGen first discovers semantic differences between original error programs and patch programs by generating new test inputs. It then tests patch programs based on these semantic differences, generating tests to identify whether patch programs are overfitting, preventing the recurrence of similar overfitting patches by repair techniques [29]. Opad uses O-measure to determine whether a patch is overfitting based on validation results, supplementing G&V technology. If a patch is identified as overfitting, G&V technology continues searching for the next candidate patch [30]. TEST-SIM automatically determines patch correctness based on behavior similarity between program executions. While useful in supplementation, TEST-SIM cannot solve the oracle problem [31].

2.4 Machine Learning and Patch Overfitting Detection

APCA techniques based on machine learning typically involve extracting code features (e.g., static representations or dynamic execution traces) and building classifier models to directly predict patch correctness [15]. Machine learning techniques in this context encompass decision trees, neural networks, support vector machines, etc. In machine learning, supervised and unsupervised learning are two crucial paradigms. In supervised learning, models learn from labeled training data, including input features and corresponding labels. In unsupervised learning, models attempt to learn patterns and structures from unlabeled data [32]. The challenge of patch overfitting in automated program repair often involves supervised learning, as models need to be constructed based on known repair and non-repair samples to predict the effectiveness of new repairs. Feature engineering is a critical step in machine learning, involving the selection and construction of input features to aid model understanding and data prediction. In automated program repair, feature engineering may include syntax and semantic analysis of code, contextual information of code, and the version history of code changes [33]. In machine learning, selecting appropriate models and evaluation methods is crucial. Model choice depends on the nature of the task, such as decision trees, support vector machines, deep neural networks, etc.

Evaluation methods include cross-validation, accuracy, recall, F1 score, etc. In automated program repair, choosing the right models and evaluation methods can help identify and address patch overfitting issues, crucial for detecting and resolving overfitting problems in automated program repair [34].

3 Ease of Use

In this section, we will discuss the general process of learning based patch overfitting detection and introduce some techniques based on the adopted technical features.

3.1 Automated Program Repair and Patch

Based on the learning-based patch overfitting detection, it can be divided into two phases: the Training Process and the Inference Phase [34]. Figure 1 shows the overall process of a learning based patch overfitting detection method, which consists of two process: the training process (as shown in the figure) and the inference process (as shown in the figure).

• Training Process

In the Training Process, data collection and preparation are essential. This includes gathering a substantial amount of training data, encompassing both patches generated by automatic program repair (APR) and non-repair data. Diverse data covering various projects, programming languages, and error types enhance the model's generalization. Subsequently, data cleaning, standardization, and feature engineering are conducted to ensure data quality and consistency. Feature extraction and engineering steps follow, involving extracting relevant features from repair patches, such as code syntax, semantic information, code context, and code change history. Feature engineering enhances feature quality and diversity to better capture signs of overfitting.

In the model selection and design phase, choosing an appropriate machine learning model, such as decision trees, support vector machines, or deep neural networks, is crucial for detecting patch overfitting. Designing the model architecture, including input feature dimensions, the number of hidden layers, and sizes, is equally important. Subsequent data partitioning and cross-validation split the training dataset into training and validation sets, commonly using cross-validation to assess the model's performance. This aids in detecting the model's generalization ability and reduces the risk of overfitting. Model training, a critical step in the training phase, utilizes the training dataset to teach the model how to distinguish between repair and non-repair cases. The model learns based on features and adjusts model parameters during training, guided by the performance on the validation set. Finally, the model's performance is evaluated on the validation set using metrics like accuracy, recall, F1 score, quantifying the model's performance. Analyzing the evaluation results determines if the model is accurate enough to detect patch overfitting.

• Inference Process

In the Inference Process, data preparation involves obtaining the repair patch data to be detected, typically generated by automatic program repair tools. Similar feature extraction and preprocessing steps as in the training phase ensure data consistency and formatting. Feature extraction from repair patches follows using the same methods as in training. Ensuring feature engineering consistency aligns it with the trained model. The trained model, from the training phase, is then used to predict new repair patches. The model judges whether a repair patch exhibits overfitting issues based on features. A threshold can be set, depending on the application's needs and the model's performance, using the probability values or classification labels output by the model. Lastly, interpreting the model's prediction results helps developers understand why the model identifies specific repair patches as having overfitting problems. Generating reports and marking patches that may need further review or regeneration concludes the process [15].

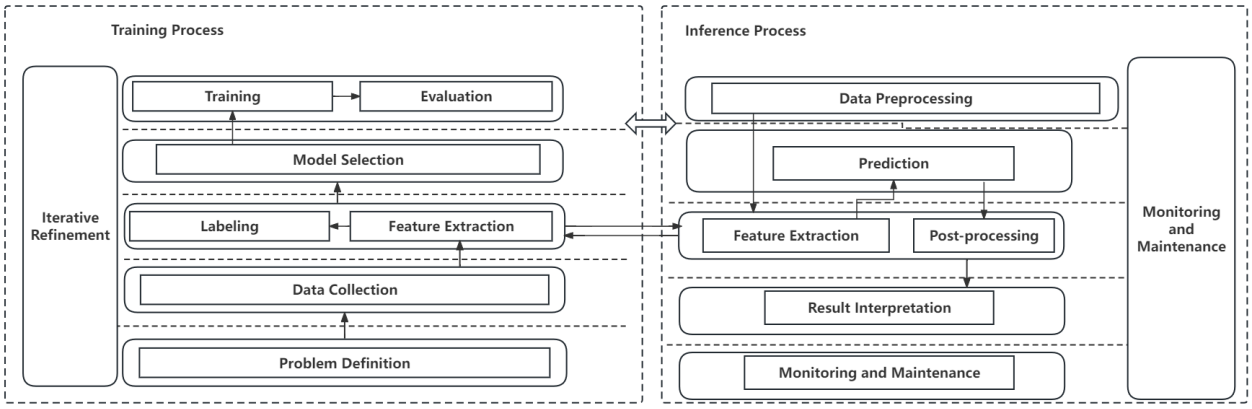

Figure 1. Architecture of learning based patch overfitting detection

3.2 Learning Based Patch Overfitting Detection Techniques

In accordance with the technical characteristics adopted in the literature on learning-based overfitting patch detection, as shown in Table 1, they are broadly categorized into two types: Code Representation and Engineered Feature. The first column in Table 1 corresponds to Technical Features, the second to Paper, the third to Language, the fourth to Year, the fifth to Model, and the sixth to Dataset.

• Code Representation

Viktor Csuvik [35] et al. applied Doc2vec and Bert embedding methods to study the reliability of using document/sentence embedding techniques on source code. Despite these methods inherently focusing on natural language text, preliminary experiments were conducted using different representations of source code. The use of exported similarity lists demonstrated that similarity-based techniques can effectively identify incorrect patches, providing a valuable alternative to patch filtering methods [35]. MIPI (Meaning-based Incorrect Patches Identifier) [36] serves as an automatic patch assessment tool designed to heuristically identify incorrect patches generated by APR tools. The core idea involves leveraging the similarity between the developer's intent, expressed in natural text (such as JavaDoc comments or the names of program entities), and the semantic meaning of the implemented code. MIPI utilizes the Code2Vec model to identify the meaning of code snippets and employs the BERT word embedding model to measure semantic similarity between natural language descriptions. MIPI identifies patch correctness by comparing the semantic meaning between the names of the already fixed method (embedding the developer's intent) and the original and fixed method bodies. MIPI filters out overfitting patches more than techniques based on automatic test generation without Oracle information, achieving higher accuracy in filtering overfitting patches while retaining more correct patches [36].

Viktor Csuvik [37] et al. trained a Doc2Vec model on an open-source JavaScript project to validate the feasibility of reasonable patches. Existing filtering techniques were applied to JavaScript, and a thorough analysis of its usability was conducted. The motivation stemmed from the assumption that a correct program is more similar to the original program than other candidate programs. Current techniques typically involve single-line code modifications, preserving the majority of the original source code. The goal is to create simple and readable patches seamlessly integrated with the original code repository [37]. APPT (Automated Pretrained model-based Patch correctness assessment technique) [38] utilizes pre-training and fine-tuning to overcome limitations in prior work. It adopts a large pre-trained model as an encoder stack to extract code representations and employs bidirectional LSTM layers to capture dependencies between buggy and patched code snippets. A deep learning classifier predicts patch overfitting. APPT exclusively uses source code tokens as input, automatically extracting features with a well-trained encoder stack, eliminating the need for code-aware features. Implemented with the BERT model, APPT outperforms the state-of-the-art CACHE technique, improving accuracy, precision, recall, F1-score, and AUC. With different pre-trained models, APPT demonstrates enhanced performance. APPT contributes a new direction by utilizing large pre-trained models without complex feature design [38].

Quatrain (Question Answering for Patch Correctness Evaluation) [39], a supervised learning approach, utilizes a deep NLP model to classify the relatedness of bug reports with patch descriptions. Quatrain is extensively evaluated on a dataset of 9,135 plausible patches, including those written by developers or generated by APR tools. The evaluation compares Quatrain to state-of-the-art dynamic and static approaches, demonstrating comparable or superior performance in terms of AUC, F1, +Recall, and -Recall. The paper analyzes the impact of input quality on prediction performance, highlighting the potential benefits of extended research into patch summarization, also known as commit message generation, for the software engineering committee.

The authors conduct a preliminary validation study demonstrating the correlation between bug and patch descriptions in a dataset of developer-submitted patches. This initial finding suggests a novel direction for patch correctness studies utilizing bug artifacts [39]. Patcherizer [40], a novel approach addressing issues in patch representation for software engineering tasks. Patcherizer leverages a combination of context, a novel SeqIntention representation for sequential patches, and a GraphIntention representation for patches. This allows the use of powerful deep learning models like Transformers for sequence intention and graph-based models like GCN for graph intention. The model is pre-trained and task-agnostic, enabling fine-tuning for various downstream tasks. The authors extensively evaluate Patcherizer on three key patch representation tasks: generating patch descriptions in natural language, predicting patch correctness in program repair, and detecting patch intentions. Patcherizer's superiority over carefully-selected baselines and the state of the art in patch representation learning across multiple downstream tasks [40]. xTestCluster [41], a novel test-based patch clustering approach, demonstrates its ability to create at least two clusters for almost half of the bugs with multiple patches. This clustering significantly reduces the median number of patches to review and analyze by 50%, providing a time-saving advantage for code reviewers and researchers assessing patches. xTestCluster not only aids in patch evaluation but also offers valuable inputs, generated from test cases, showcasing different program behaviors for distinct patches related to a single bug. Implementation of xTestCluster for Java patches utilizes popular automated test-case generation frameworks Evosuite and Randoop, along with a patch-length-based selection strategy. The source code for xTestCluster is publicly available. Evaluation of xTestCluster using patches from 25 APR tools and 1910 plausible patches with all data are accessible in the provided appendix [41].

• Engineered Feature

Leopard [42], an innovative patch correctness

prediction framework, employs learning algorithms to infer the correctness of patches. Researchers assess the feasibility of selecting repaired code and the similarity score between chosen code snippets in the model learning embeddings. Investigating the discriminative ability of learned embeddings in Leopard's classification training pipeline, they compare it with state-of-the-art methods through ten cross-validations. To enhance the identification of correct patches, Leopard integrates learned embeddings with engineering features and implements an upgraded version called Panther. This exploratory study is supported by empirical evidence, providing experiential explanations for the feature and classifier prediction reasons behind SHAP, a machine learning technique [42]. The Overfitting Patch Detection System (ODS) [43] is designed to identify overfitting patches generated by program repair tools. ODS relies on static extraction of raw code features from Abstract Syntax Trees (AST) of buggy programs and patched programs.

Utilizing three types of features: code description features, repair pattern features, and contextual syntax features, ODS employs supervised learning for training and prediction. ODS's limitation lies in its lack of consideration for dynamic code features, relying solely on static code features, leading to a deficiency in semantic foundations when determining patch correctness [43]. Crex [44], a model designed for patch correctness identification based on learned execution semantics. Crex investigating the automation of patch correctness identification in automated program repair of C programs, addressing a research gap that predominantly focuses on Java programs. Proposing a novel perspective on patch correctness identification through micro-executions and transfer learning of execution semantics, with Crex leveraging learned embeddings to predict correctness based on functional changes. Conducting experimental validations using patches from CoCoNut, demonstrating that Crex achieves high recall (100%) and F1 (89.0%) when the classifier is trained on micro-trace-based embeddings with a Logistic regression learner [44].

As can be seen from Table 1, after Viktor Csuvik and others proposed using embedded code alternatives to replace engineering features in 2020, more and more researchers began to study code representation learning to solve the problem of patch overfitting detection. There are far more papers on technical features based on code representation than on those based on engineering features.

Table 1. A summary of learning based patch overfitting detection studies

Technical features	Paper	Language	Year	Model	Dataset
Code representation	Utilizing source code embeddings to identify correct patches [35]	Java	2020	Doc2vec, BERT	QuixBugs
	Identifying incorrect patches in program repair based on meaning of source code [36]	Java	2022	Code2Vec, BERT	QuixBugs, Defects4J
	Exploring plausible patches using source code embeddings in Javascript [37]	JavaScript	2021	Doc2ve	BugsJS
	APPT: Boosting automated patch correctness prediction via pre-trained language model [38]	Java	2023	BERT	Defects4J
	Is this change the answer to that problem? Correlating descriptions of bug and code changes for evaluating patch correctness [39]	Java	2022	QA-Model, BERT	Defects4j, Bugs.jar and Bears
	Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics [44]	C	2022	Trex	Codeflaws
	Learning to represent patches [40]	Java	2023	GCN-based model, Patcherizer, BERT	Defects4j, Bugs.jar and Bears
	Test-based patch clustering for automatically-generated patches assessment [41]	Java	2022	Clustering model	Defects4J
Engineered feature	Automated classification of overfitting patches with statically extracted code features [43]	Java	2021	ODS	Defects4J, Bugs.jar, Bears
	The best of both worlds: combining learned embeddings with engineered features for accurate prediction of correct patches [42]	Java	2023	BERT	Bugs.jar, Bears, Defects4J, QuixBugs, and ManySStuBs4J

4 Dataset and Metric

In this section, we introduce existing datasets widely used in the field of learning based patch overfitting detection and discuss common evaluation metrics for evaluating patch overfitting.

4.1 Dataset

In the study of patch overfitting detection based on learning, several datasets have been instrumental in advancing research. These datasets, namely Defects4j, Quixbugs, Bears, Bugs.jar, Manysstubs4j, and Codeflaws, play a significant role in understanding and addressing patch overfitting. Here is a brief summary of each dataset:

Defects4J [3]: Defects4J is one of the most widely used datasets. Its extensibility allows seamless integration of new bugs obtained from reported fixes, ensuring continual growth. Featuring a robust test execution framework, Defects4J facilitates the implementation of tools for experiments in software testing research. This framework encompasses components for fundamental tasks such as test execution, test generation, and code coverage or mutation analysis [3].

QuixBugs [45]: QuixBugs comprises of 40 programs translated into both Python and Java. This unique characteristic renders each buggy program intriguing, and collectively, the dataset minimizes potential experimenter bias [45].

Bugs.jar [46]: Bugs.jar, a new large-scale, diverse dataset of 1,158 real bugs and patches from 8 large, popular open-source Java projects, spans 8 distinct and prominent Java application domains [46].

Bears [47]: Bears is an innovative project that creates an extensible bug benchmark to evaluate automatic program repair tools in Java. BEARS stands out for its use of CI to identify program versions, allowing bug collection from diverse projects, including those beyond mature projects with bug tracking systems [47].

ManySStuBs4J [48]: The ManySStuBs4J dataset comprises of 10,231 and 63,923 instances of single-statement bugs extracted from 12,598 and 86,771 bug-fix commits, respectively, focusing on single-statement changes for each version. The dataset is stored in JSON files, and comprehensive details are available in the GitHub repository. Each SStuB instance includes annotations for the satisfied SStuB pattern, project name, Java file name, commit hashes, bug start line, and AST subtree location [48].

Codeflaws [49]: Codeflaws is a pioneering benchmark comprising of 3902 defects extracted from 7436 programs. These defects are systematically classified across 39 defect classes, representing distinct fault types [49].

In our investigation, these datasets were used by different purposes in different studies. Some studies that use Defects4j as a training dataset mainly include training for their models, and some studies use Defects4j as an evaluation dataset that usually include a test. In Table 2, it can be observed that the Defects4J dataset is utilized in 8 research papers, showing the highest frequency among the collected studies. QuixBugs, Bears, and Bugs.jar are also popular datasets in the domain of learning-based patch overfitting detection. Codeflaws and Manysstubs4j datasets have been referenced in only one paper each, with the literature utilizing the Codeflaws dataset focusing solely on that particular dataset. Crex, designed for C programs, has chosen Codeflaws as its benchmark dataset for research. These datasets collectively enhance the comprehensiveness and diversity of the research landscape, providing valuable insights into the challenges and opportunities in patch overfitting detection suite used to verify the correctness of patches.

Table 2. Datasets usage information

Dataset	Number of paper used
Defects4J	8 [36, 38-43]
QuixBugs	3 [35-36, 42]
Bears	4 [39-40, 42-43]
Bugs.jar	4 [39-40, 42-43]
ManySStuBs4J	1 [42]
Codeflaws	1 [44]

4.2 Metric

Evaluation metrics play a crucial role in determining the performance of models in detecting overfitting issues in automated program repair (APR) patches. Key metrics include:

TP (True Positives): The number of correctly detected program defects requiring repair, indicating successful identification of issues.

FP (False Positives): The number of instances where the model incorrectly labels non-defective code as requiring repair, indicating the identification of unnecessary issues.

TN (True Negatives): The number of instances where the model correctly labels non-defective code as not requiring repair, representing the correct identification of issue-free code.

FN (False Negatives): The number of instances where the model incorrectly labels code requiring repair as not needing repair, indicating failure to detect existing issues.

These fundamental metrics are used to evaluate the model's performance in detecting overfitting issues in automated program repair patches. However, individual metrics may not provide a complete picture. Therefore, additional evaluation metrics are utilized for a comprehensive assessment:

Precision (PPV): Also called positive predictive value, the model between true positives and false positives, which is calculated as formula 1.

$$Precision = \frac{TP}{(TP + FP)}. \quad (1)$$

Higher precision indicates fewer unnecessary repairs, reducing the false-positive rate.

Recall: Recall is divided into +Recall and -Recall. +Recall is also called TPR (true positive rate), -Recall is also called TNR (True Negative Rate). +Recall quantifies

the percentage of correctly identified patches among all actual correct patches. In contrast, -Recall gauges the percentage of incorrectly identified patches that are successfully filtered out from all actual incorrect patches. The comprehensive evaluation of recall metrics contributes to a nuanced understanding of the system's performance in correctly recognizing and excluding patches. Where +Recall is calculated as formula 2.

$$+Recall = \frac{TP}{(TP + FN)}. \tag{2}$$

The -Recall is calculated as formula 3.

$$-Recall = \frac{TN}{(TN + FP)}. \tag{3}$$

Accuracy: Represents the model's ability to correctly classify samples overall, which is calculated as formula 4.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{4}$$

Higher accuracy indicates overall good performance.

CPR (Correct Patch Ratio): The ratio of correctly generated repair patches to the total generated patches, which is calculated as formula 5.

$$CPR = \frac{TP}{TP + FP}. \tag{5}$$

A higher CPR indicates more accurate generation of repair patches.

nDCGp (Normalized Discounted Cumulative Gain) [37]: DCG measures the usefulness of a document based on its position in the result list. To address the challenge of varying similarity list lengths, preventing consistent performance comparison with DCG, the cumulative gain at each position is normalized, resulting in the nDCG metric [49]. Where nDCGp is calculated as formula 6.

$$nDCGp = \frac{DCGp}{IDCGp}. \tag{6}$$

AUC (Area Under Curve): AUC quantifies the entire two-dimensional area beneath the Receiver Operating Characteristic (ROC) curve. AUC measures the probability that a classifier will prioritize a randomly selected overfitting patch over a randomly chosen correct patch. The higher the AUC, the more proficient the APCA techniques are in correctly identifying real overfitting patches as overfitting and genuine correct patches as correct. Where AUC is calculated as formula 7.

$$AUC = \frac{\sum I(P_{overfitting}, P_{correct})}{M \times N}. \tag{7}$$

DestructiveRatio [36]: The Destructive Ratio (DR) metric assesses the impact of the APCA technique on Automatic Patch Recognition (APR) tools. A DR greater than 1 signifies a higher proportion of correct patches filtered compared to overfitting patches, indicating a risk outweighing the benefit. Conversely, a DR smaller than 1 implies a smaller risk than the benefit, as the proportion of correct patches filtered is smaller than overfitting patches [36]. Where DestructiveRatio is calculated as formula 8.

$$DestructiveRatio = \frac{(1 - TNR)}{TPR}. \tag{8}$$

F1: The F1 score, a widely adopted metric in APCA, serves as a comprehensive measure that balances precision and recall. The F1 score offering a single value that considers both false positives and false negatives. Higher F1 score signifies a better balance between accurately identifying correct patches and avoiding the misclassification of incorrect ones. Where F1 is calculated as formula 9.

$$F1 = \frac{PPV * recall}{PPV + recall}. \tag{9}$$

NPV (Negative Predictive Value): NPV represents the proportion of patches classified as correct by the assessment technique that are indeed correct. A high NPV in APCA indicates a robust ability to identify true negatives, enhancing the reliability of the assessment. Researchers and practitioners can leverage NPV to understand how well an APCA technique avoids the misclassification of correct patches, contributing to the overall evaluation of its performance. Where NPV is calculated as formula 10.

$$NPV = \frac{TN}{(TN + FN)}. \tag{10}$$

BLEU (Bilingual Evaluation Understudy) [50]: BLEU is a metric commonly used in the evaluation of machine translation. In the APCA domain, BLEU is employed to quantify the similarity between the generated and correct patches. This metric aids in determining how well the automated system aligns with manually validated patches. Integrating BLEU into APCA methodologies enhances the assessment process by offering a standardized measure of correctness, contributing to the overall effectiveness of patch evaluation techniques [50]. Where BLEU is calculated as formula 11.

$$BLEU = BP \times \exp(\sum_{n=1}^{N} w_n \log p_n). \tag{11}$$

In Table 3, +Recall and -Recall are frequently utilized in the collected literature compared to other metrics. Their prominence stems from their direct reflection of a method's efficacy in identifying patches. Precision and

Accuracy each appear five times in the papers, often used concurrently. When the positive and negative sample quantities in the training and experimental datasets are comparable, Precision and Accuracy serve as suitable metrics to evaluate method performance. In studies involving fundamental metrics such as TP, FP, TN, FN, these indicators are concurrently employed. TP metric Indicates the model's capability to accurately identify positive cases. In comparison to the FN metric, it neglects false negatives. The FP metric highlights instances where the model erroneously predicts positive outcomes. However, it may be influenced by imbalanced datasets compared to other metrics. In scenarios where the training and experimental datasets exhibit a relatively balanced distribution of positive and negative samples, Precision or Accuracy can be employed to assess the method's performance. nDCGp is applicable to patch ranking scenarios but is rarely used in other contexts, with only one paper utilizing this metric. The recently proposed Destructive Ratio combines +Recall and -Recall, demonstrating heightened consideration for whether the APCA technique might result in more significant destructive or adverse impacts compared to alternative metrics. These evaluation metrics play a crucial role in machine learning studies addressing overfitting issues in automated program repair patches. They assist researchers in assessing model performance, improving algorithms, reducing false positives and false negatives, and enhancing the efficiency and effectiveness of automated program repair.

Table 3. Metrics usage information

Metric	Number of paper used
TP	3 [36, 43-44]
FP	3 [36, 43-44]
TN	3 [36, 43-44]
FN	3 [36, 43-44]
Precision (PPV)	5 [35-36, 38, 42-44]
+Recall (TPR)	8 [35-36, 38-40, 42-44]
Accuracy	5 [36, 38, 42-44]
CPR	1 [43]
BLEU	1 [40]
NPV	1 [35]
-Recall (TNR)	4 [36, 39-40, 42]
F1	5 [35, 38-39, 42, 44]
DestructiveRatio	1 [36]
nDCGp	1 [37]
AUC	2 [38-39, 42]

5 Shortcomings and Prospects

This paper has shortcomings in collecting related technologies. The methodology leans towards bias, as the systematic review of learning-based patch overfitting detection primarily explores Defects4J, potentially overlooking other potential methods. There is a need for a more comprehensive consideration of different technologies. Performance evaluation standards: There is a requirement for a more detailed discussion and comparison of performance evaluation standards for machine learning methods detecting patch overfitting issues. Future research could involve validation across multiple datasets, expanding the dataset scope to verify the robustness and applicability of the methods. Consideration of integrated methods involving different types, including semantic-based and heuristic search-based, could enhance the generality and applicability of patch generation. Developing more comprehensive evaluation metrics is suggested to more accurately assess the performance of machine learning methods in automated program repair.

6 Conclusion

This paper provides a review of learning-based patch overfitting detection, covering the most popular automated repair techniques, deep neural networks, and their construction mechanisms. The mentioned change detection methods indicate that deep learning technology has successfully propelled the development of change detection, achieving significant progress. However, due to the diversity of requirements and the complexity of data, overfitting detection still faces many challenges. Therefore, it is strongly recommended to propose several suggestions for the future research direction of patch detection to pay more attention to these challenges. First, appropriately combining static and dynamic code features in research directions. Second, more effectively utilizing machine learning methods to detect and alleviate patch overfitting issues in automated program repair. Third, simultaneously detecting patch overfitting and more effectively use machine learning techniques to improve the quality of patches in automated program repair.

References

[1] G. Fraser, A. Arcuri, Evosuite: automatic test suite generation for object-oriented software, *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, Szeged, Hungary, 2011, pp. 416-419.

[2] C. Le Goues, T. V. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, *IEEE transactions on software engineering*, Vol. 38, No. 1, pp. 54-72, January-February, 2012.

[3] R. Just, D. Jalali, M. D. Ernst, Defects4J: A database of existing faults to enable controlled testing studies for Java programs, *Proceedings of the 2014 international symposium on software testing and analysis*, San Jose, CA, USA, 2014, pp. 437-440.

[4] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, M. Monperrus, Nopol: Automatic repair of conditional statement bugs in java programs, *IEEE Transactions on Software Engineering*, Vol. 43, No. 1, pp. 34-55, January, 2017.

[5] G. Fraser, A. Arcuri, A large-scale evaluation of automated unit test generation using evosuite, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 24, No. 2, pp. 1-42, December, 2014.

[6] X. B. D. Le, F. Thung, D. Lo, C. Le Goues, Overfitting in semantics-based automated program repair, *Empirical Software Engineering*, Vol. 23, No. 5, pp. 3007-3033, October, 2018.

[7] C.-H. Lee, C.-Y. Huang, Applying Cluster-based Approach to Improve the Effectiveness of Test Suite Reduction, *International Journal of Performability Engineering*, Vol. 18, No. 1, pp. 1-10, January, 2022.

[8] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, H. Jin, Automated patch correctness assessment: How far are we?, *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, Virtual Event, Australia, 2020, pp. 968-980.

[9] J. Yang, Y. Wang, Y. Lou, M. Wen, L. Zhang, A Large-Scale Empirical Review of Patch Correctness Checking Approaches, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, San Francisco, CA, USA, 2023, pp. 1203-1215.

[10] L.-C. Thanh, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, Q.-H. Bui, Q.-T. Huynh, Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning, *IEEE Transactions on Software Engineering*, Vol. 49, No. 6, pp. 3411-3429, June, 2023.

[11] Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, L. Zhang, A syntax-guided edit decoder for neural program repair, *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Athens, Greece, 2021, pp. 341-353.

[12] F. Logozzo, T. Ball, Modular and verified automatic program repair, *ACM SIGPLAN Notices*, Vol. 47, No. 10, pp. 133-146, October, 2012.

[13] C.-E. Lai, C.-Y. Huang, Developing a Modified Fuzzy-GE Algorithm for Enhanced Test Suite Reduction Effectiveness, *International Journal of Performability Engineering*, Vol. 19, No. 4, pp. 223-233, April, 2023.

[14] T. Durieux, M. Monperrus, Dynamoth: dynamic code synthesis for automatic program repair, *Proceedings of the 11th International Workshop on Automation of Software Test*, Austin, TX, USA, 2016, pp. 85-91.

[15] Q. Zhang, C. Fang, Y. Ma, W. Sun, Z. Chen, *A Survey of Learning-based Automated Program Repair*, November, 2023. https://arxiv.org/abs/2301.03270

[16] T. Dietterich, Overfitting and undercomputing in machine learning, *ACM computing surveys (CSUR)*, Vol. 27, No. 3, pp. 326-327, September, 1995.

[17] D. M. Hawkins, The problem of overfitting, *Journal of chemical information and computer sciences*, Vol. 44, No. 1, pp. 1-12, January, 2024.

[18] X. Ying, An overview of overfitting and its solutions, *Journal of physics: Conference series*, Vol. 1168, Article No. 022022, 2019.

[19] E. K. Smith, E. T. Barr, C. Le Goues, Y. Brun, Is the cure worse than the disease? overfitting in automated program repair, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, Bergamo, Italy, 2015, pp. 532-543.

[20] A. Nilizadeh, Automated program repair and test overfitting: measurements and approaches using formal methods, *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, Valencia, Spain, 2022, pp. 480-482.

[21] T. Durieux, F. Madeiral, M. Martinez, R. Abreu, Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts, *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn, Estonia, 2019, pp. 302-313.

[22] M. Lim, G. Guizzo, J. Petke, Impact of test suite coverage on overfitting in genetic improvement of software, *12th International Symposium on Search-Based Software Engineering (SSBSE 2020)*, Bari, Italy, 2020, 188-203.

[23] H. Ye, M. Martinez, M. Monperrus, Automated patch assessment for program repair at scale, *Empirical Software Engineering*, Vol. 26, No. 2, pp. 1-38, March, 2021.

[24] Z. Qi, F. Long, S. Achour, M. Rinard, An analysis of patch plausibility and correctness for generate-and-validate patch generation systems, *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, USA, 2015, pp. 24-36.

[25] M. Wen, J. Chen, R. Wu, D. Hao, S.-C. Cheung, Context-aware patch generation for better automated program repair, *Proceedings of the 40th International Conference on Software Engineering*, Gothenburg, Sweden, 2018, pp. 1-11.

[26] X. B. D. Le, D. H. Chu, D. Lo, C. Le Goues, W. Visser, S3: syntax-and semantic-guided repair synthesis via programming by examples, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 593-604.

[27] Q. Xin, S. P. Reiss, Leveraging syntax-related code for automated program repair, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, Urbana, IL, USA, 2017, pp. 660-670.

[28] A. Nilizadeh, G. T. Leavens, X. B. D. Le, C. S. Păsăreanu, D. R. Cok, Exploring true test overfitting in dynamic automated program repair using formal methods, *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, Porto de Galinhas, Brazil, 2021, pp. 229-240.

[29] Q. Xin, S. P. Reiss, Identifying test-suite-overfitted patches through test case generation, *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, Santa Barbara, CA, USA, 2017, pp. 226-236.

[30] J. Yang, A. Zhikhartsev, Y. Liu, L. Tan, Better test cases for better automated program repair, *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany, 2017, pp. 831-841.

[31] Y. Xiong, X. Liu, M. Zeng, L. Zhang, G. Huang, Identifying patch correctness in test-based program repair, *Proceedings of the 40th international conference on software engineering*, Gothenburg, Sweden, 2018, pp. 789-799.

[32] T. Mikolov, K. Chen, G. Corrado, J. Dean, *Efficient estimation of word representations in vector space*, September, 2013. https://arxiv.org/abs/1301.3781

[33] K. Liu, D. Kim, T. F. Bissyandé, T. Kim, K. Kim, A. Koyuncu, S. Kim, Y. Le Traon, Learning to spot and refactor inconsistent method names, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, Montreal, QC, Canada, 2019, pp. 1-12.

[34] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, T. F. Bissyandé, Evaluating representation learning of code changes for predicting patch correctness in program repair, *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Virtual Event, Australia, 2020, pp. 981-992.

[35] V. Csuvik, D. Horváth, F. Horváth, L. Vidács, Utilizing source code embeddings to identify correct patches, *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing*

(IBF), London, ON, Canada, 2020, pp. 18-25.

[36] Q. N. Phung, M. Kim, E. Lee, Identifying Incorrect Patches in Program Repair Based on Meaning of Source Code, *IEEE Access*, Vol. 10, pp. 12012-12030, January, 2022.

[37] V. Csuvik, D. Horváth, M. Lajkó, L. Vidács, Exploring plausible patches using source code embeddings in Javascript, *2021 IEEE/ACM International Workshop on Automated Program Repair (APR)*, Madrid, Spain, 2021, pp. 11-18.

[38] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, Z. Chen, *APPT: Boosting Automated Patch Correctness Prediction via Pre-trained Language Model*, January, 2023. https://arxiv.org/abs/2301.12453

[39] H. Tian, X. Tang, A. Habib, S. Wang, K. Liu, X. Xia, J. Klein, T. F. Bissyandé, Is this change the answer to that problem?: Correlating descriptions of bug and code changes for evaluating patch correctness, *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Rochester, MI, USA, 2022, pp. 1-13.

[40] X. Tang, H. Tian, Z. Chen, W. Pian, S. Ezzini, A. K. Kabore, A. Habib, J. Klein, T. F. Bissyande, *Learning to Represent Patches*, October, 2023. https://arxiv.org/abs/2308.16586

[41] M. Martinez, M. Kechagia, A. Perera, J. Petke, F. Sarro, A. Aleti, *Test-based Patch Clustering for Automatically-Generated Patches Assessment*, July, 2022. https://arxiv.org/abs/2207.11082

[42] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, T. F. Bissyandé, The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches, *ACM Transactions on Software Engineering and Methodology*, Vol. 32, No. 4, pp. 1-34, July, 2023.

[43] H. Ye, J. Gu, M. Martinez, T. Durieux, M. Monperrus, Automated classification of overfitting patches with statically extracted code features, *IEEE Transactions on Software Engineering*, Vol. 48, No. 8, pp. 2920-2938, August, 2022.

[44] D. Yan, K. Liu, Y. Niu, L. Li, Z. Liu, Z. Liu, J. Klein, T. F. Bissyandé, Crex: Predicting patch correctness in automated repair of C programs through transfer learning of execution semantics, *Information and Software Technology*, Vol. 152, Article No. 107043, December, 2022.

[45] D. Lin, J. Koppel, A. Chen, A. Solar-Lezama, QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge, *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, Vancouver, BC, Canada, 2017, pp. 55-56.

[46] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, M. R. Prasad, Bugs. jar: A large-scale, diverse dataset of real-world java bugs, *Proceedings of the 15th international conference on mining software repositories*, Gothenburg, Sweden, 2018, pp. 10-13.

[47] F. Madeiral, S. Urli, M. Maia, M. Monperrus, Bears: An extensible java bug benchmark for automatic program repair studies, *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019, 468-478.

[48] R. M. Karampatsis, C. Sutton, How often do single-statement bugs occur?: the manysstubs4j dataset, *Proceedings of the 17th International Conference on Mining Software Repositories*, Seoul, Republic of Korea, 2020, pp. 573-577.

[49] S. H. Tan, J. Yi, Yulis, S. Mechtaev, A. Roychoudhury, Codeflaws: a programming competition benchmark for evaluating automated program repair tools, *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina, 2017, pp. 180-182.

[50] K. Papineni, S. Roukos, T. Ward, W.-J. Zhu, Bleu: a method for automatic evaluation of machine translation, *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, Philadelphia, Pennsylvania, 2002, pp. 311-318.

Biographies

Xuanyan Li is a graduate student in computer science at China University of Geosciences (Wuhan). His research interests include intelligent algorithms and software security detection.

Dongcheng Li earned his Ph.D. and M.S. degrees in Computer Science from the University of Texas at Dallas and holds a B.S. in Computer Science from the University of Illinois Springfield. Currently, he serves as an Assistant Professor in the Department of Computer Science at California State Polytechnic University, Humboldt. His research is centered on search-based software engineering, test generation, program repair, and intelligent optimization algorithms.

Man Zhao is currently an associate professor at China University of Geosciences (Wuhan). Her main research directions are computer science and technology, intelligent computing and artificial intelligence.

W. Eric Wong received his Ph.D. in computer science from Purdue University. He was at Telcordia Technologies (formerly, Bellcore) as a Senior Research Scientist and a Project Manager, where he was in charge of dependable telecom software development. He is currently a Full Professor and the Founding Director of the Advanced Research Center for Software Testing and Quality Assurance, Computer Science Department, The University of Texas at Dallas. He also has an appointment as a Guest Researcher with the National Institute of Standards and Technology (NIST). His research focuses on software testing, debugging, risk analysis/metrics, safety, and reliability. In 2014, he was named as the IEEE Reliability Society Engineer of the Year.

Hui Li is now a professor at China University of Geosciences (Wuhan). Her main research direction is to use intelligent computing theories and methods to solve various complex problems in the aerospace field.