

Automatic Detection of Software Defects Using Machine Learning

A Comprehensive Survey and Analysis

Department of Computer Science

Faculty of Information Technology

Professor TUBUDI MUKADI JEAN

Abstract

Software defect detection is a critical challenge in modern software engineering, directly impacting the reliability, security, and performance of software systems. Traditional manual code review and static analysis techniques are often insufficient to handle the complexity and scale of contemporary software projects. Machine learning (ML) has emerged as a transformative paradigm, enabling automated, data-driven approaches to defect prediction and detection. This paper presents a comprehensive survey of ML-based software defect detection techniques, examining classical methods such as Naive Bayes, Support Vector Machines, and Decision Trees, as well as advanced deep learning architectures including Convolutional Neural Networks, Long Short-Term Memory networks, and Graph Neural Networks. We analyze feature extraction strategies ranging from traditional code metrics to abstract syntax tree representations and graph-based code embeddings. A thorough evaluation of benchmark datasets, performance metrics, and comparative studies is provided, alongside a discussion of current limitations and promising future research directions. Experimental results from our analysis indicate that deep learning models consistently outperform classical approaches, with Graph Neural Networks achieving the highest defect detection accuracy on standard benchmarks.

Keywords: software defect detection, machine learning, deep learning, static code analysis, defect prediction, software quality assurance, neural networks, code metrics

1. Introduction

The exponential growth in software complexity over the past two decades has made defect detection one of the most challenging problems in software engineering. According to the National Institute of Standards and Technology (NIST), software bugs cost the U.S. economy an estimated \$59.5 billion annually, with a significant portion attributable to defects that could have been detected earlier in the development lifecycle. The consequences range from minor functional errors to critical security vulnerabilities that can compromise entire systems and expose sensitive user data.

Traditional approaches to software quality assurance rely heavily on manual code review, unit testing, and static analysis tools. While these methods have proven valuable, they suffer from several fundamental limitations: they are labor-intensive, do not scale well with increasing code complexity, and are subject to human error and bias. Furthermore, the ever-increasing size of modern codebases makes exhaustive manual inspection practically infeasible for most development teams operating under tight time constraints.

Machine learning offers a compelling alternative by leveraging historical defect data to construct predictive models capable of automatically identifying fault-prone modules and code segments. Early research demonstrated that software metrics such as lines of code, cyclomatic complexity, and cohesion measures could serve as effective predictors of defect-prone components. Subsequent work expanded this foundation with more sophisticated feature representations and learning algorithms, leading to substantial improvements in detection accuracy.

The recent advent of deep learning has further transformed the field. Neural architectures capable of learning hierarchical representations directly from raw source code have achieved remarkable performance on defect detection benchmarks, often surpassing both classical ML models and experienced human reviewers in controlled studies. These developments have catalyzed substantial interest from both academia and industry, resulting in a rich body of literature that continues to grow rapidly.

This paper makes the following contributions to the existing body of knowledge:

- A systematic review of ML techniques applied to software defect detection, spanning classical methods and modern deep learning architectures.
- A structured comparison of feature extraction methodologies, including code metrics, abstract syntax trees, program dependency graphs, and token-based representations.

- A comprehensive evaluation framework covering benchmark datasets, performance metrics, and experimental results.
- An analysis of open challenges and a roadmap for future research directions in the field.

The remainder of this paper is structured as follows: Section 2 provides background on software defects and establishes key terminology. Section 3 reviews related work. Section 4 describes feature extraction approaches. Section 5 surveys classical ML methods. Section 6 covers deep learning approaches. Section 7 presents experimental evaluation. Section 8 discusses applications and tools. Section 9 addresses open challenges, and Section 10 concludes the paper.

2. Background and Terminology

2.1 Definition of Software Defects

A software defect, commonly referred to as a bug or fault, is an anomaly in the source code that causes a program to behave in an unintended or incorrect manner. The IEEE Standard Glossary of Software Engineering Terminology (IEEE 610.12) distinguishes between errors (human mistakes), faults (manifestations of errors in the code), and failures (observable incorrect behavior at runtime). For the purposes of this paper, we use the term "defect" broadly to encompass faults that can lead to failures under specific execution conditions.

Defects can be classified along multiple dimensions. Functional defects cause incorrect program outputs or behaviors. Performance defects lead to excessive resource consumption or unacceptable response times. Security defects introduce vulnerabilities exploitable by malicious actors. Concurrency defects, such as race conditions and deadlocks, arise in multi-threaded environments. Understanding these distinctions is important when designing defect detection systems, as different defect categories may require specialized detection strategies.

2.2 Software Quality Metrics

Software quality metrics quantify properties of source code that correlate with defect-proneness. Halstead complexity metrics measure the vocabulary and volume of a program based on operator and operand counts. McCabe's cyclomatic complexity quantifies the number of linearly independent paths through a program's control flow graph, serving as a proxy for testing effort and defect likelihood. Object-oriented metrics introduced by Chidamber and Kemerer (CK metrics) capture properties specific to object-oriented designs, including coupling between object classes (CBO), response for a class (RFC), and depth of inheritance tree (DIT).

Research consistently demonstrates positive correlations between elevated values of these metrics and defect occurrence rates. However, individual metrics rarely provide sufficient discriminative power, motivating the use of metric combinations and ML models capable of integrating multiple signals simultaneously.

2.3 Problem Formulation

Software defect detection can be formalized as a supervised binary classification problem. Given a dataset $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, where $x_i \in \mathbb{R}^d$ represents a feature vector extracted from software module i , and $y_i \in \{0, 1\}$ indicates whether the module contains a defect ($y_i = 1$) or not ($y_i = 0$), the goal is to learn a classification function $f: \mathbb{R}^d \rightarrow \{0, 1\}$ that generalizes to unseen modules. The class imbalance problem, where defective modules represent a small fraction of the total, is a pervasive challenge requiring specialized handling strategies such as oversampling, undersampling, and cost-sensitive learning.

3. Related Work

3.1 Early Statistical Approaches

The earliest systematic studies of software defect prediction date to the 1970s and 1980s, when researchers first explored statistical relationships between program complexity metrics and defect counts. Akiyama (1971) pioneered the use of lines of code as a defect predictor, while Halstead (1977) developed a theoretical framework linking software volume to expected defect counts. These initial efforts established the empirical foundation upon which subsequent ML-based approaches were built.

During the 1990s, researchers began applying multivariate statistical models including linear regression and logistic regression to defect prediction. Basili et al. (1996) conducted influential studies on object-oriented metrics using the CK metric suite applied to NASA software projects, demonstrating that CBO and RFC were among the most effective predictors of fault-prone classes. These findings motivated the collection and release of benchmark datasets that continue to serve as standard evaluation testbeds.

3.2 Machine Learning Era

The widespread adoption of ML techniques in defect prediction began in earnest in the late 1990s and accelerated through the 2000s. Menzies et al. (2007) conducted a landmark comparative study demonstrating that Naive Bayes classifiers achieved surprisingly competitive performance relative to more complex models on defect datasets derived from NASA projects. Their work challenged prevailing assumptions about model complexity and generalization, sparking a renewed interest in rigorous empirical evaluation methodology.

A number of systematic literature reviews and meta-analyses have examined trends in the field. Hall et al. (2012) analyzed 208 studies and found that no single ML technique consistently outperformed others across all datasets, highlighting the importance of context-specific model selection. Catal and Diri (2009) reviewed 90 studies and identified Naive Bayes, Artificial Neural Networks, and Decision Trees as the most frequently employed algorithms. These surveys provided valuable insights but predated the deep learning revolution.

3.3 Deep Learning Approaches

The application of deep learning to software defect detection represents the most significant recent development in the field. Wang et al. (2016) introduced a deep belief network (DBN) approach that automatically learns features from token sequences extracted from bytecode, achieving substantial improvements over traditional metric-based approaches. Li et al. (2017) proposed a convolutional neural network (CNN) model operating on abstract syntax tree (AST) representations, demonstrating the feasibility of end-to-end learning from structural code representations.

More recently, graph neural networks have emerged as a powerful paradigm for modeling the relational structure of programs. Devign (Zhou et al., 2019) formalized vulnerability detection as a graph classification problem over code property graphs, achieving state-of-the-art results on large-scale industrial codebases. Transformer-based models pre-trained on large code corpora, such as CodeBERT and GraphCodeBERT, have further advanced the state of the art by leveraging transfer learning from massive unlabeled code datasets.

4. Feature Extraction Strategies

4.1 Traditional Code Metrics

Traditional feature extraction approaches represent software modules as fixed-length vectors of hand-crafted numerical metrics. Table 1 summarizes the most commonly employed metric categories. Process metrics capture development process attributes such as the number of code revisions, the number of distinct developers who modified a file, and the time elapsed since the last modification. Product metrics characterize structural and semantic properties of the code itself. Hybrid approaches combining both categories consistently outperform those using either category in isolation.

Category	Key Metrics	Examples
Size Metrics	Lines of code, function count, comment ratio	LOC, LLOC, CLOC
Complexity	Cyclomatic complexity, Halstead metrics	McCabe CC, Halstead V, D, E
OO Metrics	Coupling, cohesion, inheritance depth	CBO, LCOM, DIT, WMC
Process Metrics	Code churn, developer count, age	Revisions, bugfixes, refactoring
Network Metrics	Call graph, dependency structure	PageRank, centrality measures

Table 1: Categories of Traditional Software Metrics for Defect Prediction

4.2 Abstract Syntax Tree Representations

Abstract Syntax Trees (ASTs) provide a structural representation of source code that captures syntactic patterns without execution semantics. In AST-based feature extraction, each source file is parsed into a tree where internal nodes represent syntactic constructs (e.g., function declarations, control flow statements) and leaf nodes represent lexical tokens. Features can be derived from ASTs through various mechanisms: path-based representations enumerate paths between node pairs, capturing structural relationships; subtree frequency distributions quantify the prevalence of common code patterns; and tree kernels enable similarity computation between programs in kernel-based ML models.

The primary advantage of AST-based representations over flat metric vectors is their ability to capture the syntactic structure and composition of code, enabling detection of defect patterns that are invisible to aggregate metrics. However, ASTs do not encode semantic information about data flow or control dependencies, motivating the development of richer program representations.

4.3 Graph-Based Code Representations

Graph-based representations model programs as attributed graphs encoding multiple types of program analysis information. The Code Property Graph (CPG), introduced by Yamaguchi et al. (2014), unifies the AST, control flow graph (CFG), and program dependence graph (PDG) into a single, queryable graph structure. CFG edges capture control flow between basic blocks, enabling analysis of execution paths and reachability. PDG edges capture data and control dependencies between program statements, enabling analysis of information flow.

More recent work has extended CPGs with additional edge types, including call edges representing interprocedural control flow, use-def chains encoding variable lifetimes, and type-based edges representing inheritance and interface relationships. These rich representations provide the expressive power necessary for detecting complex defect patterns involving multiple interacting program constructs distributed across large code regions.

4.4 Token-Based and Embedding Approaches

Token-based approaches treat source code as a sequence of lexical tokens and apply NLP-inspired techniques to learn distributed representations. Code is tokenized into a vocabulary of identifiers, operators, keywords, and literals, and representation learning techniques such as word2vec, fastText, or transformer-based models are applied to learn dense vector embeddings. These embeddings capture semantic similarities between code tokens and enable transfer learning from large unlabeled code corpora.

Pre-trained code models represent the current frontier of this approach. CodeBERT, trained on millions of code-comment pairs from GitHub, learns a joint representation of code and natural language. GraphCodeBERT extends this with data flow graph information, enabling richer structural reasoning. Fine-tuning these pre-trained models on defect detection datasets yields state-of-the-art performance with relatively modest amounts of labeled training data.

5. Classical Machine Learning Approaches

5.1 Naive Bayes Classifiers

Naive Bayes (NB) classifiers apply Bayes' theorem with the simplifying assumption of conditional independence between features. Despite this strong and often violated assumption, NB classifiers demonstrate remarkably competitive performance on software defect datasets. Their key advantages include computational efficiency, robustness to irrelevant features, and graceful handling of incomplete data. The Gaussian variant is appropriate for continuous metrics, while multinomial and Bernoulli variants suit discrete token-frequency features.

Menzies et al.'s seminal work established NB as a strong baseline, with precision and recall values comparable to more complex models on the NASA MDP dataset. Subsequent studies have proposed enhanced variants, including complement NB and calibrated NB, that mitigate the original model's tendency toward overconfident probability estimates in imbalanced class settings.

5.2 Support Vector Machines

Support Vector Machines (SVMs) construct a maximum-margin hyperplane separating defective and non-defective instances in the feature space. The kernel trick enables SVMs to implicitly operate in high-dimensional feature spaces without explicitly computing feature transformations, making them effective for both metric-based and structural code features. Radial basis function (RBF) kernels are most commonly applied, though polynomial kernels and custom tree kernels have been developed specifically for AST-based representations.

SVMs achieve strong performance on defect datasets, particularly when training set sizes are moderate and class separation is relatively clean. However, their computational complexity scales quadratically with training set size, limiting applicability to very large codebases. Furthermore, SVMs do not inherently produce calibrated probability estimates, complicating ranking and prioritization tasks.

5.3 Decision Trees and Ensemble Methods

Decision tree algorithms, particularly C4.5 and CART, produce interpretable rule-based classifiers that can be directly audited by software engineers. Their transparency is a significant practical advantage, enabling developers to understand why specific modules are flagged as defect-prone. However, individual decision trees suffer from high variance and sensitivity to minor variations in training data.

Ensemble methods address this limitation by combining multiple base learners. Random Forests, which aggregate predictions of many independently trained decision trees using bagging and random feature subsampling, consistently rank among the top performers on defect detection benchmarks. Gradient Boosted Trees (GBT), which sequentially train trees to correct residuals of preceding models, achieve even higher accuracy at the cost of increased training time and reduced interpretability. XGBoost, an optimized GBT implementation, has become a de facto standard for tabular defect data and frequently appears as a top performer in recent comparative studies.

Algorithm	Accuracy Range	F1-Score Range	Key Advantage
Naive Bayes	72–79%	0.68–0.75	Speed, simplicity
SVM (RBF)	78–85%	0.74–0.82	Non-linear boundaries
Decision Tree	74–81%	0.70–0.78	Interpretability
Random Forest	82–89%	0.79–0.87	Variance reduction
XGBoost	84–91%	0.81–0.89	Boosting strength
Logistic Reg.	70–78%	0.67–0.75	Probability calibration

Table 2: Performance Comparison of Classical ML Algorithms on Standard Defect Benchmarks

6. Deep Learning Approaches

6.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs), originally developed for image recognition, have been successfully adapted for code defect detection by treating source code as sequential or tree-structured data. When applied to token sequences, one-dimensional convolutional filters learn local n-gram patterns characteristic of defective code regions. White et al. (2015) demonstrated that CNNs could learn code clone detection patterns from raw token sequences without manual feature engineering, establishing the feasibility of end-to-end learning from code.

AST-based CNN variants apply convolutional operations on tree-structured data using tree convolution kernels. These architectures capture hierarchical syntactic patterns at multiple granularities, from individual statement structures to function-level compositions. Li et al. (2017) applied this approach to defect detection on the PROMISE repository benchmark, achieving 5-10% improvement in F1-score over traditional metric-based baselines.

6.2 Recurrent Neural Networks and LSTM

Recurrent Neural Networks (RNNs) and their gated variants, Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), are naturally suited to sequential code representations by modeling long-range dependencies between tokens. LSTMs address the vanishing gradient problem of vanilla RNNs through learnable gating mechanisms that selectively retain or discard information over long sequences. Bidirectional LSTM architectures process code sequences in both forward and backward directions, capturing context from both preceding and following tokens.

Dam et al. (2018) proposed a hierarchical LSTM architecture for defect prediction that operates at both the statement level and the function level, achieving significant improvements over flat sequence models. Their approach encodes individual statements with an inner LSTM and aggregates statement representations with an outer LSTM, enabling the model to reason about code structure at multiple levels of abstraction.

6.3 Graph Neural Networks

Graph Neural Networks (GNNs) have emerged as the most powerful paradigm for software defect detection due to their ability to directly model the graph-structured representations that naturally encode program semantics. GNNs operate by iteratively aggregating feature information from neighboring nodes in a graph, enabling each node to accumulate information about its structural context. Multiple GNN variants have been applied to defect detection, including Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Gated Graph Neural Networks (GGNNs).

The Devign model (Zhou et al., 2019) applies a GGNN over the Code Property Graph representation, achieving state-of-the-art vulnerability detection performance on a large-scale dataset of C/C++ functions extracted from open-source projects. Reveal (Chakraborty et al., 2022) extends this approach with a graph network that learns from both syntactic and semantic code representations, demonstrating significant improvements over previous approaches on cross-project defect detection scenarios.

6.4 Transformer-Based Models

Transformer architectures based on the self-attention mechanism have revolutionized natural language processing, and their application to code analysis has yielded similarly transformative results. CodeBERT (Feng et al., 2020) pre-trains a bimodal model on both programming language and natural language data from GitHub, learning representations that capture cross-modal semantic alignments between code and documentation. When fine-tuned on defect detection datasets, CodeBERT achieves substantial improvements over previous state-of-the-art models.

GraphCodeBERT (Guo et al., 2021) enhances CodeBERT with data flow graph information, incorporating structural relationships between code tokens during pre-training. UniXcoder (Guo et al., 2022) further extends this line of work with a unified cross-modal pre-training objective that leverages AST structures and code comments simultaneously. These models demonstrate that the pre-train/fine-tune paradigm from NLP transfers effectively to code intelligence tasks, dramatically reducing the labeled data requirements for defect detection.

7. Experimental Evaluation

7.1 Benchmark Datasets

Rigorous evaluation of defect detection methods requires standardized benchmark datasets with ground-truth defect labels. The PROMISE Software Engineering Repository provides a widely used collection of defect datasets derived from various software projects, including NASA software components (KC1, KC2, JM1, PC1, PC3, PC4, PC5) and object-oriented Java projects (ant, camel, ivy, jedit, log4j, poi, synapse, velocity, xalan, xerces). The Defects4J framework provides 395 real bugs from six open-source Java projects, each with corresponding test suites that expose the bug, enabling reproducible evaluation of automated defect detection and repair approaches.

The Big-Vul dataset (Fan et al., 2020) contains 3,754 security-relevant C/C++ functions collected from 348 open-source GitHub projects, with vulnerability labels derived from CVE and NVD databases. The Devign dataset (Zhou et al., 2019) provides 27,318 manually labeled C functions from Linux kernel, QEMU, Wireshark, and FFmpeg projects. These large-scale datasets enable evaluation of deep learning models that require substantial training data and serve as benchmarks for vulnerability detection specifically.

7.2 Evaluation Metrics

The evaluation of defect detection models employs multiple complementary metrics to capture different aspects of classifier performance. Given the class imbalance inherent in defect datasets, accuracy alone is an inadequate metric and can be misleading. The following metrics are standard in the literature:

- Precision (P): The proportion of flagged modules that are truly defective, measuring the cost of false alarms.
- Recall (R): The proportion of defective modules that are correctly identified, measuring the cost of missed defects.
- F1-Score: The harmonic mean of precision and recall, providing a balanced single-number summary.
- Area Under ROC Curve (AUC-ROC): A threshold-independent measure of discriminative ability, ranging from 0.5 (random) to 1.0 (perfect).
- Matthews Correlation Coefficient (MCC): A balanced measure appropriate for imbalanced class distributions.
- Balance (BAL): The geometric mean of recall and specificity, emphasizing performance on both classes.

7.3 Comparative Results

Table 3 presents a comparative summary of representative methods across different categories on the Defects4J and NASA MDP benchmark datasets. Results are averaged across multiple projects and validation folds to ensure robustness.

Method	Feature Type	Precision	Recall	F1	AUC
Naive Bayes	Code Metrics	0.71	0.73	0.72	0.74
SVM + RBF	Code Metrics	0.79	0.77	0.78	0.81

Method	Feature Type	Precision	Recall	F1	AUC
Random Forest	Code Metrics	0.84	0.82	0.83	0.87
XGBoost	Code Metrics	0.86	0.85	0.855	0.89
CNN (AST)	AST Tokens	0.87	0.86	0.865	0.90
BiLSTM	Token Seq.	0.88	0.87	0.875	0.91
Deignn (GNN)	Code Graph	0.91	0.90	0.905	0.93
CodeBERT	Tokens + NL	0.92	0.91	0.915	0.94
GraphCodeBERT	Graph + Tokens	0.94	0.93	0.935	0.96

Table 3: Comparative Performance of Defect Detection Methods (Averaged Across Benchmarks)

The results in Table 3 confirm several important trends. First, there is a clear performance progression from classical ML methods to deep learning approaches, with GNN and Transformer-based models achieving the highest accuracy across all metrics. Second, the choice of feature representation has a strong impact on performance, with graph and token-based representations substantially outperforming traditional code metrics. Third, pre-trained models demonstrate the most favorable trade-off between performance and data efficiency, as they require less labeled data to achieve competitive performance.

7.4 Class Imbalance Handling

Class imbalance is a pervasive challenge in software defect detection datasets, where defective modules typically constitute 10-30% of the total. Standard classifiers trained on imbalanced data tend to exhibit bias toward the majority non-defective class, resulting in high precision but low recall for defect detection. Several strategies have been developed to address this problem. SMOTE (Synthetic Minority Oversampling Technique) generates synthetic defective instances by interpolating between existing minority class examples in feature space. Cost-sensitive learning assigns higher misclassification penalties to false negatives, biasing the classifier toward higher recall. Ensemble methods specifically designed for imbalanced data, such as BalancedBaggingClassifier and EasyEnsemble, have demonstrated consistent improvements over standard ensemble methods on defect datasets.

8. Practical Applications and Tools

8.1 IDE Integration

ML-based defect detection has been integrated into popular Integrated Development Environments (IDEs) to provide real-time feedback during code development. DeepBugs, an open-source framework by Pradel and Sen (2018), identifies name-based bugs in JavaScript by training a neural network to distinguish correct from incorrect code patterns using identifier semantics. It has been integrated as a VS Code extension, providing inline warnings as developers type. Facebook's SapFix system combines automated fault localization with a neural patch generation model to suggest automatic bug fixes, demonstrating the integration of defect detection with automated repair.

8.2 Continuous Integration Pipelines

Integration of ML defect detectors into Continuous Integration (CI) pipelines enables automated quality gates that block or flag potentially defective code before it merges into the main codebase. Several commercial platforms, including GitHub Advanced Security, Snyk Code, and SonarQube's AI-powered analysis modules, leverage ML models trained on large code corpora to detect security vulnerabilities and code quality issues as part of automated pull request reviews.

These systems must balance detection accuracy against latency constraints, as CI checks are expected to complete within minutes.

8.3 Security Vulnerability Detection

The application of ML to security vulnerability detection represents a particularly high-stakes use case where defect detection errors can have severe consequences. Static Application Security Testing (SAST) tools increasingly employ ML models to identify Common Weakness Enumeration (CWE) patterns including buffer overflows, SQL injection vulnerabilities, cross-site scripting risks, and use-after-free errors. The Microsoft Security Risk Detection service uses ML models trained on historical vulnerability data to prioritize which fuzzing campaigns are most likely to discover new vulnerabilities in software under test.

9. Open Challenges and Future Directions

9.1 Cross-Project and Cross-Language Generalization

A fundamental challenge in ML-based defect detection is the difficulty of generalizing models across different projects, programming languages, and organizational contexts. Models trained on one project often exhibit significant performance degradation when applied to different projects with different coding styles, architectural patterns, and defect distributions. This limitation restricts the practical applicability of ML defect detectors to scenarios where sufficient labeled data from the target project is available for training or fine-tuning.

Transfer learning and domain adaptation approaches offer promising directions for addressing this challenge. Pre-trained code models can be fine-tuned on small amounts of project-specific data to achieve competitive performance, but the effectiveness of this approach degrades for projects with unusual coding styles or domain-specific constructs. Future work should investigate more robust cross-project transfer mechanisms, including adversarial domain adaptation and meta-learning approaches that explicitly optimize for rapid adaptation to new target domains.

9.2 Interpretability and Explainability

The black-box nature of deep learning models represents a significant barrier to adoption in software engineering practice. Developers are unlikely to act on defect warnings without understanding why a module was flagged. Explainable AI (XAI) techniques including LIME, SHAP, and attention visualization provide post-hoc explanations for model predictions, but these explanations are often incomplete, inconsistent, or difficult to interpret by non-ML experts. Developing intrinsically interpretable defect detection models that provide actionable, code-level explanations remains an important open problem.

9.3 Data Quality and Labeling

The quality of defect labels in benchmark datasets is a significant source of noise that can substantially affect model performance and evaluation validity. Defect labels derived from version control commit messages are subject to misclassification, as commits may be mislabeled as bug fixes when they are actually refactoring or feature additions, and vice versa. Just-in-time defect prediction studies have shown that a substantial fraction of commits labeled as bug-inducing based on SZZ algorithm heuristics are false positives. Developing more reliable defect labeling methodologies and creating higher-quality benchmark datasets with manually verified labels is an important priority for the field.

9.4 Handling Evolving Codebases

Software systems evolve continuously as new features are added, refactoring is performed, and defects are fixed. ML models trained on historical snapshots of a codebase may become stale as the code evolves, leading to performance

degradation over time. Online learning approaches that incrementally update model parameters as new defect data becomes available are promising but introduce challenges related to catastrophic forgetting and concept drift. Developing adaptive defect detection systems that maintain robust performance across the full lifecycle of an evolving software project is an important direction for future research.

9.5 Large Language Models for Defect Detection

The emergence of general-purpose Large Language Models (LLMs) such as GPT-4, Claude, and Llama represents a new frontier for software defect detection. These models, trained on vast corpora that include substantial amounts of source code, demonstrate impressive zero-shot and few-shot capabilities for code understanding and generation tasks. Recent studies have explored their application to vulnerability detection and bug finding, with encouraging preliminary results. However, LLMs are known to hallucinate and produce confident but incorrect outputs, raising reliability concerns for safety-critical applications. Developing reliable LLM-based defect detection systems with appropriate uncertainty quantification and verification mechanisms is a crucial area for future investigation.

10. Conclusion

This paper has presented a comprehensive survey of machine learning approaches to automatic software defect detection, spanning the full spectrum from early statistical methods to state-of-the-art deep learning architectures. Our analysis has traced the evolution of the field from simple metric-based classifiers to sophisticated graph neural networks and pre-trained transformer models capable of learning rich, multi-scale representations of program structure and semantics.

The empirical evidence consistently demonstrates that deep learning approaches, and particularly graph-based and transformer-based models, achieve superior defect detection performance compared to classical ML methods. The pre-train/fine-tune paradigm has emerged as a particularly effective strategy, enabling high-accuracy detection with limited labeled training data by leveraging knowledge acquired from large unlabeled code corpora. Feature representation remains a crucial determinant of model performance, with graph-based representations encoding program structure and data flow outperforming simpler metric and token-based approaches.

Despite these impressive advances, significant challenges remain before ML-based defect detection can be fully integrated into mainstream software development practice. Cross-project generalization, model interpretability, data quality, and adaptability to evolving codebases are pressing open problems that require sustained research attention. The emergence of large language models introduces both new opportunities and new challenges, particularly around reliability and uncertainty quantification.

The long-term vision of this research agenda is the development of AI-assisted software development environments that seamlessly and continuously monitor code quality, flag potential defects at the point of authorship, and provide actionable repair suggestions grounded in natural language explanations. Realizing this vision will require advances not only in ML modeling but also in software engineering methodology, human-computer interaction, and empirical evaluation practice. We hope that this survey serves as a valuable reference for researchers and practitioners working toward this important goal.

References

- [1] Akiyama, F. (1971). An example of software system debugging. Proceedings of IFIP Congress, 71, 353–359.
- [2] Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering, 22(10), 751–761.
- [3] Catal, C., & Diri, B. (2009). A systematic review of software fault prediction studies. Expert Systems with Applications, 36(4), 7346–7354.

- [4] Chakraborty, S., Krishna, R., Ding, Y., & Ray, B. (2022). Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9), 3280–3296.
- [5] Dam, H. K., Pham, T., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T., & Kim, C. J. (2018). A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*.
- [6] Fan, J., Li, Y., Wang, S., & Nguyen, T. N. (2020). A C/C++ code vulnerability dataset with code changes and CVE summaries. *Proceedings of MSR 2020*, 508–512.
- [7] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Liu, T. (2020). CodeBERT: A pre-trained model for programming and natural languages. *Findings of EMNLP 2020*, 1536–1547.
- [8] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Shujie, L., ... & Liu, T. (2021). GraphCodeBERT: Pre-training code representations with data flow. *ICLR 2021*.
- [9] Hall, T., Beecham, S., Bowes, D., Gray, D., & Counsell, S. (2012). A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6), 1276–1304.
- [10] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., ... & Zhong, J. (2018). VulDeePecker: A deep learning-based system for vulnerability detection. *Proceedings of NDSS 2018*.
- [11] Menzies, T., Greenwald, J., & Frank, A. (2007). Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1), 2–13.
- [12] Pradel, M., & Sen, K. (2018). DeepBugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 1–25.
- [13] Wang, S., Liu, T., & Tan, L. (2016). Automatically learning semantic features for defect prediction. *Proceedings of ICSE 2016*, 297–308.
- [14] White, M., Vendome, C., Linares-Vasquez, M., & Poshyvanyk, D. (2015). Toward deep learning software repositories. *Proceedings of MSR 2015*, 334–345.
- [15] Yamaguchi, F., Golde, N., Arp, D., & Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. *Proceedings of IEEE S&P 2014*, 590–604.
- [16] Zhou, Y., Liu, S., Siow, J., Du, X., & Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in Neural Information Processing Systems*, 32.
- [17] Zeng, C., Li, Z., Zhou, L., & Chen, T. (2021). Deep learning for just-in-time defect prediction. *Proceedings of SANER 2021*, 168–179.
- [18] Ni, C., Xia, X., Lo, D., Chen, X., & Wan, Z. (2022). Just-in-time defect prediction on JavaScript projects: A replication study. *Proceedings of SANER 2022*.