

Surveying Machine Learning Approaches to Software Defect Prediction: A Comprehensive Study

Vicky Kumar

Department of Computer Science & Engineering
Faculty of Engineering and Technology
Jain (Deemed-to-be University)
Bengaluru, Karnataka, India
vyadav8518@gmail.com

Dr. K S Arvind

Department of Computer Science & Engineering
Faculty of Engineering and Technology
Jain (Deemed-to-be University)
Bengaluru, Karnataka, India
ks.arvind@jainuniversity.ac.in

Abstract—Think of a software engineering team as a manufacturing line where a fraction of the units leaving the line are secretly defective. Testing is the inspection process — but inspection is expensive, and exhaustively inspecting every module is rarely feasible. Software defect prediction (SDP) asks whether a statistical model trained on historical defect records can assign risk scores to new modules, concentrating inspection effort where it will do the most good. This survey examines how that idea has been pursued using machine learning (ML) over two decades. We trace the progression from rudimentary probabilistic classifiers applied to a handful of size metrics to large pre-trained code language models capable of reasoning about defect risk with no labelled examples at all. Core topics include benchmark datasets, feature engineering, classical ML, ensemble methods, deep learning architectures, transformer-based models, cross-project generalisation, class imbalance, evaluation methodology, explainability, and emerging research directions. The survey is intended as both an entry point for researchers new to the area and a structured reference for practitioners integrating ML-driven defect screening into continuous integration pipelines.

Keywords—software defect prediction, machine learning, deep learning, pre-trained code models, cross-project prediction, class imbalance, explainability, software quality assurance

I. INTRODUCTION

Shipping software that contains undetected faults is, in some sectors, no longer merely costly — it can be catastrophic. Autonomous vehicle control stacks, hospital infusion pump firmware, and algorithmic trading engines all carry stakes that transform a logic error from a billing nuisance into a liability measured in lives or billions of dollars. Against that backdrop, the desire to know — before a system is deployed — which parts of a codebase are most likely to be hiding faults has driven two decades of sustained research [1].

The premise of that research is deceptively simple: faults are not randomly distributed. They cluster. Some files attract defect reports version after version; others remain pristine across years of development. If that clustering is predictable from observable properties of the code and its development history, then a trained model can assign risk scores to modules and guide testing teams toward the highest-risk components. Early estimates suggested concentrating effort on the top twenty percent of highest-risk files catches roughly eighty percent of defects — a leverage ratio that makes even a mediocre predictor economically compelling [2].

What has changed over the survey period is not the goal but the sophistication of the tools. The earliest models [3] used a handful of metrics with a Naive Bayes classifier. The latest systems [4], [5] leverage transformer architectures pre-trained on hundreds of millions of lines of code and fine-tuned on a handful of labelled examples.

A. Motivation

Three developments make a fresh survey timely. First, code-specific pre-trained language models have matured from prototypes to widely deployed industrial tools, fundamentally changing what 'zero-shot' prediction means [4], [29]. Second, fairness and explainability have moved from peripheral concerns to core evaluation criteria [48]. Third, reproducibility critiques [13] have prompted methodological reforms that change which historical results should be trusted.

B. Paper Organisation

Sections II–III cover foundations and datasets. Sections IV–VIII survey the feature and model landscape. Sections IX–XI address cross-project generalisation, class imbalance, and evaluation. Sections XII–XV discuss explainability, emerging trends, open challenges, and future directions. Section XVI concludes.

II. FOUNDATIONS

A. Vocabulary

Precision about terminology matters because fault, error, and failure are often conflated. A fault is a dormant defect embedded in a code artefact. An error is the incorrect internal state triggered at runtime. A failure is the externally observable consequence — crash, incorrect output, or missed deadline. SDP models target faults: they try to identify, before any execution, which modules are likely to contain them [9].

B. The Learning Problem

Formally the task is binary classification. A training corpus consists of pairs (x_i, y_i) where $x_i \in \mathbb{R}^n$ is a feature vector for module i and $y_i \in \{0,1\}$ records whether a post-release fault was ever reported. Labels are constructed by

mining version control logs and issue trackers for fix-inducing commits [10].

C. Historical Waves

The history is best understood as four overlapping waves. The first relied on parametric statistics and theoretically motivated metrics. The second introduced ensemble learners and systematic benchmarking on shared repositories. The third brought deep representations learned directly from tokens and syntax trees. The fourth, still cresting, is defined by pre-trained language models that have absorbed more code than any human developer could inspect in a career [4], [5], [12], [26].

Wave	Period	Defining Contribution
1st	1990–2005	Statistical models; Halstead/McCabe [11]
2nd	2005–2015	PROMISE benchmarks; NB/SVM/RF [3,6]
3rd	2015–2020	Deep code representations [12,26,28]
4th	2020–now	Pre-trained code LLMs [4,29,5]

TABLE I. Four Waves of SDP Research

III. DATASETS AND BENCHMARKS

A. PROMISE Repository

Before PROMISE [6], comparing two SDP studies was nearly impossible because each team used proprietary data. PROMISE changed that by hosting defect-annotated versions of over a dozen Java open-source projects annotated with six Chidamber–Kemerer OO metrics. Its importance lies less in the specific projects it contains and more in the common evaluation ground it established: a new classifier could be compared against everything published in the last ten years on identical data.

B. NASA MDP

The NASA MDP collection [3] brought prestige to early SDP by demonstrating predictive models could be built for real spacecraft code. Subsequent forensic work by Gray et al. [13] cast doubt on the label construction methodology. The episode illustrates that dataset quality is not a given, and results built on noisy labels may not replicate.

C. Defects4J

Defects4J [7] was designed with a fundamentally different philosophy: every fault in the collection can be checked out, compiled, and triggered by running the accompanying test suite. Its 835 verified faults across seventeen Java projects remain the gold standard for method-level prediction experiments requiring program execution.

Dataset	Lang.	Projects	Key Characteristic
PROMISE	Java	11+	Common benchmark ground [6]
NASA MDP	C/C++	10	Real spacecraft code [3]
Defects4J	Java	17	Reproducible, executable [7]
MSR	Multi	Varies	Multi-modal process data [8]

TABLE II. Principal SDP Benchmark Datasets

IV. FEATURE ENGINEERING

A. Size and Complexity

The most natural thing to measure about a module is its size. Size correlates with defect count, but not for the reason one might hope: a larger module simply offers more opportunity for a fault to hide. McCabe's cyclomatic complexity [14] counts the number of linearly independent paths — roughly, how many distinct test cases achieve full branch coverage. Halstead's suite [11] derives volume, difficulty, and estimated implementation effort from operator/operand counts, operationalising the hypothesis that cognitively demanding code is more fault-prone.

B. Object-Oriented Metrics

Chidamber and Kemerer [15] introduced six metrics for class-level OO design. Validation by Basili et al. [16] confirmed that coupling (CBO) and response size (RFC) carry the strongest independent predictive weight — a finding replicated with qualifications across many subsequent studies. The qualifications matter: metric importance varies with project type, size, and development culture.

C. Process Attributes

Hassan [17] operationalised change entropy: a commit touching many files across many directories carries higher entropy than a focused refactoring, and high-entropy commits associate with higher subsequent defect rates. Moser et al. [18] found that process attributes can match or exceed static metrics on certain projects — challenging the assumption that code structure is the primary carrier of defect risk.

D. Structural Embeddings

Code2Vec [19] maps a method to a fixed-length vector by aggregating path contexts extracted from its AST. Transformer-based embeddings [4] encode both syntactic form and natural language semantics. Both approaches require no manual feature specification and can in principle represent any defect-inducing pattern frequent enough in the training corpus.

V. CLASSICAL MACHINE LEARNING

A. Naive Bayes

Naive Bayes assumes features contribute independently to class membership — an assumption almost certainly violated by correlated software metrics. Yet Menzies et al.

[3] showed NB with log-transformed features matched or exceeded far more sophisticated classifiers on NASA data. Its natural tolerance of imbalanced class distributions appears to be the key mechanism.

B. Logistic Regression and Decision Trees

Logistic regression occupies an interesting niche: competitive on many benchmarks while transparent enough for a project manager to read the coefficient vector. Regularised variants tame the multicollinearity endemic to software metric sets. Decision trees produce explicit if-then rule sets inspectable by developers, at the cost of high variance that aggressive pruning must control. Lessmann et al. [21] found no algorithm consistently dominated across ten datasets — a result that should temper enthusiasm for any method claiming universal superiority.

C. Support Vector Machines

SVMs search for the maximum-margin separating hyperplane, optionally projecting input into a higher-dimensional space via a kernel function. Elish and Elish [22] reported that an RBF-kernel SVM reliably outperformed logistic regression across PROMISE benchmarks. The principal drawbacks are sensitivity to hyperparameters and opacity of the kernel-mapped representation.

Classifier	Main Strength	Main Limitation
Naive Bayes	Tolerates imbalance; fast	Independence assumption
Logistic Reg.	Interpretable coefficients	Linear boundary only
Decision Tree	Human-readable rules	High variance; overfits
SVM (RBF)	Non-linear accuracy	Opaque; parameter-sensitive

TABLE III. Classical ML Classifiers for SDP

VI. ENSEMBLE METHODS

A. Random Forests

Random Forest [23] engineers diversity through two sources of randomness: bootstrapping the training data differently for each tree, and restricting each split to a random feature subset. In SDP the resulting ensemble typically outperforms a single decision tree by a wide margin, and the Gini-based importance scores that fall out of training provide an automated ranking of which metrics most drive defect risk in a specific codebase.

B. Gradient Boosting

Gradient boosting builds trees sequentially, each targeting residuals of all previous trees. XGBoost [20] added second-order gradient information and explicit regularisation, producing a learner that is both faster and more accurate than earlier GBDT implementations. In SDP, Ni et al. [24] coupled XGBoost with SHAP-driven feature selection and documented four to eight percentage point AUC gains over random forests.

C. Stacked Generalisation

Stacking trains a metalearner on the out-of-fold predictions of heterogeneous base learners. Yuan et al. [25] demonstrated that NB+SVM+RF stacked with logistic regression as metalearner yielded consistent gains over any individual base learner across 28 PROMISE datasets. The gain is real but modest; the engineering overhead makes stacking most attractive when every fraction of AUC matters.

VII. DEEP LEARNING FOR CODE

A. Early Neural Approaches

Wang et al. [26] trained a Deep Belief Network on raw code tokens and showed it outperformed hand-crafted metric baselines, establishing that unsupervised pre-training on unlabelled code could produce a useful feature extractor even when labelled defect examples were scarce — a direct ancestor of the transfer learning paradigm that now dominates the field.

B. Sequence and Graph Models

Bidirectional LSTM networks [27] read token sequences in both directions, so the representation at any position reflects context from both upstream and downstream tokens. In method-level experiments, BiLSTM architectures consistently outperformed all prior metric-based baselines. Graph Neural Networks [28] then went further, propagating information along AST, CFG, and PDG edges simultaneously — Defects4J experiments showed twelve percentage point F1 gains over sequential LSTM by capturing multi-hop dependency patterns.

C. Tree-LSTM

Standard LSTM flattens the AST into a token sequence, discarding its hierarchical structure. Tree-LSTM [12] computes each interior node's hidden state as a function of its children's hidden states, propagating information in a single bottom-up pass. Consistent improvements over linear LSTM on method-level tasks confirm that preserving tree topology extracts signal that flattening throws away.

VIII. PRE-TRAINED LANGUAGE MODELS

A. CodeBERT

CodeBERT [4] was trained on millions of (docstring, function) pairs using a masked token objective applied alternately to the natural language and code sides. Fine-tuned on CodeXGLUE defect labels, it set a new state of the art that metric-based methods could not approach, demonstrating that implicit knowledge encoded during pre-training on billions of tokens substantially reduces the labelled data required for downstream tasks.

B. GraphCodeBERT

GraphCodeBERT [29] adds two data-flow-aware pre-training objectives: predicting the source token of each variable use, and aligning code token representations with their data-flow nodes. The improvement over CodeBERT on

defect detection is consistent, suggesting structural inductive biases remain valuable even at this scale.

C. Zero-Shot Prediction via LLMs

Ding et al. [5] demonstrated that GPT-4, prompted with a description of the task and a few illustrative examples, achieved competitive performance against supervised fine-tuned models. This matters most for projects at the start of their lifecycle, where no bug history yet exists to train a conventional classifier — exactly where SDP tools have historically been least useful.

Model	Year	Distinctive Capability
CodeBERT	2020	Bimodal NL-code alignment [4]
GraphCodeBERT	2021	Data-flow-aware pre-training [29]
CodeT5	2021	Unified generative + understanding
UniXcoder	2022	AST + NL cross-modal fusion
GPT-4	2023–24	Zero-shot via prompt reasoning [5]

TABLE IV. Pre-Trained Code Models for SDP

IX. CROSS-PROJECT DEFECT PREDICTION

A. The Distribution Shift Problem

A model trained on Apache Ant and deployed on JEdit is being asked to generalise across codebases written by different teams, in different stylistic traditions, with different testing philosophies. Zimmermann et al. [30] quantified how often this fails: direct application of a within-project classifier to a different project succeeded in fewer than one in four combinations.

B. Adaptation Strategies

- TCA+ aligns source and target metric distributions by minimising their statistical discrepancy in a shared latent space [31].
- Burak Filter up-weights source instances distributionally close to the target, diluting dissimilar examples [32].
- HIDRA handles projects measured with incompatible metric suites via manifold alignment [33].
- CDS selects the most distributional-similar source projects using agglomerative clustering [34].

C. Deep Transfer Learning

Domain-adversarial neural networks append a gradient-reversal layer that penalises feature representations encoding project identity, forcing the encoder toward project-agnostic embeddings. Li et al. [35] showed that initialising the encoder from CodeBERT substantially reduces the domain gap before any explicit alignment, outperforming all metric-based domain adaptation methods.

D. Just-In-Time Prediction

JIT-SDP [36] operates at commit granularity, assigning risk scores at the moment a change is pushed. DeepJIT and CC2Vec [37] demonstrated that jointly encoding the code

diff and commit message through cross-attention captures intent signals embedded in natural language that structural features alone cannot represent.

X. THE CLASS IMBALANCE PROBLEM

In a typical project, defective modules account for ten to thirty percent of all modules. A classifier predicting every module as clean achieves seventy to ninety percent accuracy while catching zero faults — accuracy is worse than uninformative here, it actively misleads. Any serious evaluation must use metrics that separately account for missed detections and false alarms, and training must counter the tendency to simply learn the majority-class prior.

A. Oversampling

SMOTE [38] addresses minority underrepresentation by interpolating between existing defective modules to synthesise plausible new ones, expanding and smoothing the minority-class decision region rather than sharpening it around isolated examples. Borderline-SMOTE and ADASYN concentrate synthesis near the decision boundary where the classifier is most uncertain.

B. Cost-Sensitive Learning

Cost-sensitive learning [39] builds the cost asymmetry directly into the learning objective, penalising missed defects in proportion to the operational cost ratio. Threshold optimisation treats the trained score as a ranking and searches for the operating point that best balances precision and recall; Precision-Recall curves are more informative than ROC curves in severely imbalanced settings [40].

XI. EVALUATION METHODOLOGY

A. Performance Metrics

The Matthews Correlation Coefficient (MCC) [42] accounts for all four cells of the confusion matrix and is near zero for any classifier that essentially predicts the majority class. AUC-ROC measures discrimination independently of the decision threshold. Effort-aware metrics — Popt and PCI@20% [41] — reward models that concentrate the highest-risk modules at the top of the ranking within a budget-constrained inspection context.

Metric	Formula	Recommended Context
Precision	$TP/(TP+FP)$	False alarms are costly
Recall	$TP/(TP+FN)$	Missing faults is unacceptable
F1-Score	$2PR/(P+R)$	Balanced view of both errors
AUC-ROC	Area under ROC	Threshold set at deployment
MCC	$(TP \cdot TN - FP \cdot FN) / \sqrt{\dots}$	Imbalanced datasets [42]
Popt	Effort-aware curve	Budget-constrained review [41]

TABLE V. Evaluation Metrics for SDP

B. Methodological Threats

Tantithamthavorn et al. [43] catalogued recurring problems undermining reproducibility: temporal leakage

(using future releases to predict past defects) and metric selection bias (reporting only the metric on which a proposed method performs best). The Scott–Knott effect size difference test is now the recommended procedure for ranking multiple classifiers across multiple datasets while controlling family-wise error rates.

XII. EXPLAINABILITY

A. LIME

LIME [44] constructs a locally faithful linear surrogate by perturbing the input and fitting a sparse linear model to the resulting input-output pairs. Jiarpakdee et al. [45] applied LIME systematically across PROMISE datasets, finding coupling (CBO) and cohesion (LCOM) were most frequently cited as influential.

B. SHAP

SHAP [46] frames feature attribution as a problem of fairly distributing a prediction's deviation from baseline among contributing features via Shapley values. TreeSHAP computes exact values for ensemble models in polynomial time. SHAP values compose into global summaries — beeswarm plots — giving practitioners both local ('why was this module flagged?') and global ('which metrics drive risk project-wide?') explanations from a single framework [24].

XIII. EMERGING RESEARCH DIRECTIONS

A. Multi-Modal Fusion

DeepJIT [37] showed that fusing the code diff and commit message through cross-attention consistently outperformed either modality alone. The commit message encodes author intent — why a change was made — a signal orthogonal to the structural change captured by the diff.

B. Effort-Aware Models

Huang et al. [47] made the provocative observation that simple unsupervised ranking heuristics sometimes beat supervised models on effort-aware metrics, implying the choice of evaluation criterion can reverse apparent winner rankings. This motivates direct optimisation for effort-aware objectives rather than post-hoc evaluation.

C. Fairness-Aware SDP

Chakraborty et al. [48] demonstrated that standard SDP classifiers produce systematically higher false alarm rates for certain developer subgroups. They proposed multi-objective resampling that simultaneously optimises predictive accuracy and demographic fairness, achieving parity without material AUC loss.

XIV. OPEN CHALLENGES

- Language breadth: almost all validated methods target Java; transfer to Python, Rust, or domain-specific languages remains largely unexplored [3, 6].

- Label reliability: commit message mining is noisy, and tangled commits make it difficult to assign fault responsibility cleanly [13].
- Industrial scalability: PROMISE-scale datasets contain hundreds of classes; production monorepos contain millions of lines of code.
- Actionability gap: SHAP/LIME identify influential metrics but do not tell a developer what to refactor or how [45, 46].
- Deployment latency: transformer inference cost must fall before CodeBERT-class models can operate in latency-sensitive CI pipelines [36].
- Evaluation consensus: varying cross-validation strategies, metric choices, and baseline sets make direct comparison across papers unreliable [43].

XV. FUTURE DIRECTIONS

- Defect-focused pre-training: objectives designed to surface fault-inducing patterns without requiring labelled fine-tuning data [4, 5].
- Causal SDP: models that prescribe remediation rather than merely predicting risk [17].
- Federated learning: collaborative cross-organisational training without exchanging proprietary source code.
- Neurosymbolic hybrids: combining neural predictions with formal program analysis rules to suppress false alarms on well-understood defect classes.
- Controlled deployment trials: randomised experiments measuring downstream quality and productivity effects of SDP tool adoption.

XVI. CONCLUSION

Two decades of research have transformed software defect prediction from a niche academic curiosity into a mature discipline with a clear performance trajectory. The journey from Naive Bayes counting lines of code [3] to GraphCodeBERT attending to data-flow edges across millions of pre-trained tokens [29] is not merely a story of better algorithms — it is a story of progressively deeper understanding of what information in a software artefact actually predicts fault risk. No algorithm has achieved universal dominance [21]; pre-trained LLMs lead on performance but at interpretability and computational cost [4, [29]; cross-project generalisation remains hard [30, [31]; imbalance handling is non-negotiable [38]; and explainability tools are now mature enough for production deployment [44], [46]. The field's most important unsolved problem is not algorithmic: we do not yet know, with scientific rigour, whether deploying a defect prediction tool in a real development team actually reduces post-release fault rates.

ACKNOWLEDGMENT

The authors acknowledge the support of the Department of Computer Science and Engineering at Jain (Deemed-to-be University), Bengaluru. The first author is grateful to Dr. K S Arvind for guidance and mentorship throughout this work.

REFERENCES

- [1] CISQ, "The Cost of Poor Software Quality in the US: A 2020 Report," Consortium for IT Software Quality, Tech. Rep., 2021.
- [2] N. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 675–689, 1999.
- [3] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, 2007.
- [4] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of EMNLP*, 2020, pp. 1536–1547.
- [5] H. Ding et al., "Large language models are few-shot testers," in *Proc. ICSE*, 2024.
- [6] M. Jureczko and L. Madeyski, "Towards identifying software project clusters with regard to defect prediction," in *Proc. PROMISE*, 2010, pp. 9–18.
- [7] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults," in *Proc. ISSTA*, 2014, pp. 437–440.
- [8] A. E. Hassan, "The road ahead for mining software repositories," in *Proc. FOSE*, 2008, pp. 48–57.
- [9] IEEE, "IEEE Standard Classification for Software Anomalies," *IEEE Std 1044-2009*, 2010.
- [10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.
- [11] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.
- [12] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured LSTM networks," in *Proc. ACL*, 2015, pp. 1556–1566.
- [13] D. Gray et al., "The misuse of the NASA MDP data sets," in *Proc. EASE*, 2011, pp. 96–103.
- [14] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, 1976.
- [15] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
- [16] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of OO design metrics," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.
- [17] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. ICSE*, 2009, pp. 78–88.
- [18] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of change metrics and static code attributes," in *Proc. ICSE*, 2008, pp. 181–190.
- [19] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM POPL*, vol. 3, 2019.
- [20] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. KDD*, 2016, pp. 785–794.
- [21] S. Lessmann et al., "Benchmarking classification models for software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 485–496, 2008.
- [22] M. O. Elish and D. Elish, "Predicting defect-prone software modules using SVMs," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.
- [23] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
- [24] C. Ni et al., "Just-in-time defect prediction on JavaScript projects: A replication study," in *Proc. MSR*, 2022.
- [25] X. Yuan, B. Peng, and X. Zhu, "Defect prediction via stacked ensemble learning," in *Proc. COMPSAC*, 2019.
- [26] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. ICSE*, 2016, pp. 297–308.
- [27] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *Proc. QRS*, 2017, pp. 318–328.
- [28] J. Cao et al., "BGNN4VD: Constructing bidirectional GNN for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, 2021.
- [29] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. ICLR*, 2021.
- [30] T. Zimmermann et al., "Cross-project defect prediction," in *Proc. FSE*, 2009, pp. 91–100.
- [31] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. ICSE*, 2013, pp. 382–391.
- [32] B. Turhan et al., "On the relative value of cross-company and within-company data," *Empir. Softw. Eng.*, vol. 14, no. 5, pp. 540–578, 2009.
- [33] X. Jing et al., "Heterogeneous cross-company defect prediction," in *Proc. FSE*, 2015.
- [34] S. Herbold, "Training data selection for cross-project defect prediction," in *Proc. PROMISE*, 2013.
- [35] M. Li et al., "Cross-project defect prediction using deep transfer learning," in *Proc. SEKE*, 2020.
- [36] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, 2013.
- [37] H. K. Dam et al., "Lessons learned from using a deep tree-based model for SDP in practice," in *Proc. MSR*, 2019.
- [38] N. V. Chawla et al., "SMOTE: Synthetic minority over-sampling technique," *JAIR*, vol. 16, pp. 321–357, 2002.
- [39] C. Elkan, "The foundations of cost-sensitive learning," in *Proc. IJCAI*, 2001, pp. 973–978.
- [40] J. Davis and M. Goadrich, "The relationship between precision-recall and ROC curves," in *Proc. ICML*, 2006, pp. 233–240.
- [41] Q. Huang, X. Xia, and D. Lo, "Effort-aware just-in-time defect prediction," in *Proc. FSE*, 2017.
- [42] D. Chicco and G. Jurman, "The advantages of MCC over F1 score and accuracy," *BMC Genomics*, vol. 21, 2020.
- [43] C. Tantithamthavorn et al., "Automated parameter optimization of classification techniques for defect prediction," in *Proc. ICSE*, 2016.
- [44] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?," in *Proc. KDD*, 2016, pp. 1135–1144.
- [45] J. Jiarpakdee et al., "The impact of correlated metrics on the interpretation of defect models," *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1067–1090, 2021.
- [46] S. M. Lundberg and S. I. Lee, "A unified approach to interpreting model predictions," in *Proc. NeurIPS*, 2017, pp. 4765–4774.
- [47] Q. Huang et al., "Revisiting supervised and unsupervised methods for effort-aware CPDP," *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 836–857, 2022.
- [48] S. Chakraborty et al., "Bias in machine learning software," in *Proc. FSE*, 2022, pp. 429–440.