Computers and Artificial Intelligence; Vol.3, No.6; 2026
E-ISSN: 3006-3728 P-ISSN: 3006-371X
DOI: https://doi.org/10.70267/cai.v3n6.120132
Published by: Zeus Press

Software Defect Prediction under Label Scarcity: A Few-Shot Prototypical Learning and Adaptive Fusion Method

Chengang Zhang*

School of Software, North University of China, Taiyuan, 030051, China

**Corresponding author: Chengang Zhang.*

Abstract

Software defect prediction is a core means of software quality assurance and can reduce operation and maintenance costs by allocating testing resources in advance. However, in early project stages and cross-organizational reuse scenarios, defect labels are commonly scarce and classes are highly imbalanced. Most existing defect prediction methods assume sufficient training labels. Fixed model strategies have difficulty adapting to different label budgets, and evaluation based on random splitting can produce optimistic bias because duplicated samples leak across data subsets. This study proposes ProtoStar, a software defect prediction framework for label-scarce settings. Its core is a few-shot prototype engine that jointly trains with focal loss and supervised contrastive loss. The framework combines this engine with a label-budget-adaptive fusion mechanism selected through out-of-fold validation and an honest evaluation pipeline based on deduplicated grouped splitting, thereby achieving robust defect-risk ranking at low labeling cost. Systematic evaluation on 13 National Aeronautics and Space Administration Metrics Data Program (NASA MDP) datasets shows that the method significantly outperforms strong supervised baselines in k-shot few-shot and total-budget-constrained cold-start settings. Adaptive fusion effectively avoids ensemble overfitting. For cross-project prediction, the defect-hit enrichment among the top 20% high-risk modules reaches 1.28-1.76 times the base rate, providing a feasible technical approach to software quality assurance under limited labeling.

Keywords

prototypical networks, software defect prediction, few-shot learning, supervised contrastive learning, adaptive fusion.

1. Introduction

Software defect prediction uses historical metric data and defect labels from software modules to construct prediction models. It estimates the likelihood that new modules contain defects before testing, so that limited testing and code-review resources can be allocated first to high-risk modules. It has been an important research topic in software quality assurance and software measurement for several decades [1,2]. Since Menzies et al. [2] demonstrated that effective defect predictors can be learned using only static code attributes, module-level prediction based on static metrics has become the most commonly used experimental paradigm in this field. With the growing application of deep learning in software engineering, deep models have also been used to learn semantic representations of code automatically and improve prediction performance [3,4]. However, deploying defect prediction in a new project or at an early project stage involves two overlapping inherent

© 2026 The Author(s). Authors retain copyright in their articles.

difficulties. First, the classes are extremely imbalanced: defective modules usually account for only a small proportion of all modules. Second, labels are costly and scarce: defect labels depend on manual review and defect-tracking records. At the beginning of a project, in legacy systems, or in cross-organizational reuse scenarios, only a small number of modules often have reliable labels, resulting in few-shot or even cold-start conditions.

Existing methods show clear limitations when the label budget is constrained. Classical statistical and machine-learning models, such as naive Bayes, logistic regression, random forests, and gradient boosting, as well as deep semantic models, usually require hundreds or thousands of labeled samples for stable training [4,5,6,7]. Oversampling represented by the Synthetic Minority Over-sampling Technique (SMOTE) [8], class-imbalance learning [9], diversity-based oversampling [10], and focal loss [11] can reduce class skew, but they still require training data of a certain scale. Few-shot metric-learning methods, such as prototypical networks [12] and supervised contrastive learning [13], are more label-efficient, but in defect prediction they have mainly been used for heterogeneous cross-project transfer [14]. Their performance under label scarcity within the same project has rarely been studied. More importantly, ensembles such as stacking are usually more accurate when labels are sufficient, but can perform worse when labels are extremely scarce because the meta-model overfits. Existing systems generally use a one-size-fits-all fixed model and lack a mechanism that adapts the predictor to the label budget.

To address these problems, this study proposes ProtoStar, a software defect prediction framework for label-scarce settings, and implements it as an executable prototype system. The framework uses a few-shot prototypical network as its core engine and jointly trains the encoder with focal loss and supervised contrastive loss, enabling compact class prototypes to form even when defective samples are extremely scarce. It also retains strong supervised models, including Extreme Gradient Boosting (XGBoost) and random forests. Based on a composite metric computed from out-of-fold predictions on the validation set, the framework automatically selects either the few-shot engine directly or a stacking ensemble. Leakage-prevention mechanisms are established at both the data and process levels, and statistically rigorous conclusions are obtained through repeated multi-seed experiments and the Wilcoxon signed-rank test.

2. Related Work

2.1 Software Defect Prediction Models

According to the data source, software defect prediction can be divided into within-project defect prediction (WPDP) and cross-project defect prediction (CPDP) [1,15]. Under the module-level static-metric paradigm, Menzies et al. [2] showed at an early stage that simple models such as naive Bayes, combined with information-gain feature selection, can achieve good results. In a systematic review of 208 fault-prediction studies, Hall et al. [5] found that robust models were often based on simple techniques such as naive Bayes and logistic regression, and that methodological rigor substantially affected the conclusions. In a large-scale benchmark of 22 classifiers on 10 NASA datasets, Lessmann et al. [6] found that there were often no statistically significant differences among more than ten top-ranked classifiers, suggesting that the importance of a specific algorithm may be overestimated. After controlling for data noise, Ghotra et al. [7] revisited the issue and found that the choice of classification technique did affect performance on clean data. In recent years, Wang et al. [4] used deep belief networks to learn semantic features automatically from source code, while Kamei et al. [16] pioneered change-level just-in-time defect prediction. The overall progress of deep learning in software engineering is reviewed in. Most of these studies assume sufficient training labels and therefore complement the label-scarce setting examined in this study.

2.2 Class-Imbalance Handling

The strong skew in defect data has prompted researchers to address the problem at both the data and algorithm levels. At the data level, SMOTE, proposed by Chawla et al. [8], synthesizes samples through linear interpolation between nearest neighbors in the minority class and is the most widely used oversampling method. Wang and Yao [9] systematically evaluated the role of class-imbalance learning in defect prediction. Bennin et al. [10] proposed MAHAKIL, which is based on chromosome-inspired genetic concepts and reduces overfitting by preserving diversity in the minority-class distribution. At the algorithm level, the focal loss

© 2026 The Author(s). Authors retain copyright in their articles.

proposed by Lin et al. [11] uses a modulating factor to reduce the weights of easy samples and focus on difficult samples. It was originally developed for object detection and was later widely transferred to imbalanced classification. This study uses SMOTE to balance the class distribution at the training-pool level and uses both focal loss and supervised contrastive loss at the loss-function level, thereby addressing extreme imbalance from both the data and objective-function sides.

2.3 Few-Shot and Metric Learning

Few-shot learning aims to generalize rapidly from very few labeled samples, and metric learning is one of its mainstream approaches. The prototypical network proposed by Snell et al. [12] uses the mean embedding of the samples in each class as the class prototype and classifies a query sample according to its distance from the prototypes. The structure is simple and label-efficient. Khosla et al. [13] proposed supervised contrastive loss, which explicitly pulls samples of the same class closer and pushes samples of different classes apart in the embedding space, thereby improving the discriminability of the representation. In defect prediction, few-shot methods have mainly been used in heterogeneous cross-project settings. For example, Wang et al. [14] proposed few-shot learning-based balanced distribution adaptation (FSLBDA), which simultaneously handles feature-distribution differences and class imbalance. In contrast, this study applies prototypical and supervised contrastive representations to label-scarce settings within the same project and further coordinates them with supervised ensembles in a budget-adaptive manner, an issue that has rarely been discussed in existing work.

2.4 Cross-Project Prediction, Data Quality, and Evaluation Rigor

Cross-project defect prediction transfers knowledge from source projects to alleviate insufficient labeling in the target project [15,17]. The large-scale experiments of Zimmermann et al. [18] and the study by Turhan et al. [19] on the relative value of cross-company and within-company data both show that distribution shift makes the success rate of cross-project prediction unstable. Nam et al. [17] proposed transfer defect learning to reduce distribution differences across projects. Regarding data quality, Shepperd et al. [20] explicitly noted that NASA datasets contain version differences, unreasonable values, and duplicated records, and called for complete reporting of preprocessing and cautious use of the data. This directly supports the deduplicated grouped-splitting design used in this study. Regarding evaluation methods, Tantithamthavorn et al. [21] empirically compared model-validation techniques such as repeated holdout, cross-validation, and bootstrapping, emphasizing the importance of the number of repetitions and statistical testing to the robustness of conclusions. Accordingly, this study uses repeated multi-seed experiments and paired significance tests and reports results under both random and grouped splitting.

Table 1: Positioning of the Proposed Method Relative to Representative Related Work

Method category	Representative work	Labeling requirement	Imbalance handling	Budget adaptation	Duplicate-leakage prevention
Classical statistical/machine learning	Naive Bayes, logistic regression, random forest	Large	Partial	No	Often overlooked
Deep semantic models	Semantic features from deep belief networks	Large	Partial	No	Depends on splitting
Sampling/cost-sensitive methods	SMOTE, imbalance learning, MAHAKIL, focal loss	Moderate	Yes	No	Depends on splitting
Cross-project/transfer	Transfer defect learning, CPDP, FSLBDA	Sufficient source-project data	Partial	No	Partially addressed
Few-shot metric learning	Prototypical networks, supervised contrastive learning	Small	General	No	General
ProtoStar (this study)	Prototype engine + adaptive fusion + honest pipeline	Very small ($k \geq 5$)	Yes	Yes	Yes

© 2026 The Author(s). Authors retain copyright in their articles.

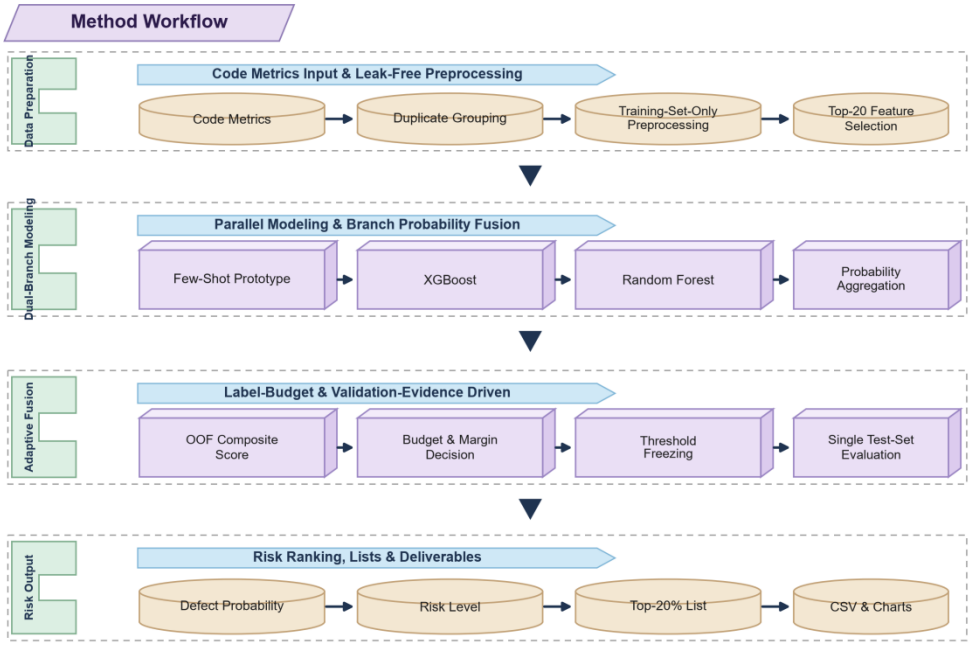
3. Methods

3.1 Problem Definition and Notation

Let the dataset be D={(x_i,y_i)}_{i=1}^N, where x_i is a d-dimensional real-valued vector of static metrics and y_i is a label in {0,1}, with 1 indicating a defective module. Let the defect rate be π=N_1/N; π is usually small in the setting considered in this study. The framework supports two label-budget-constrained settings: (1) k-shot few-shot learning, in which only k defective positive samples and a fixed set of five normal negative samples are provided as the support set, while the remaining modules are to be predicted; and (2) total-budget-constrained cold start, in which the labeling budget consists of k positive samples and 5k negative samples, labels for both classes are constrained, no oversampling is performed, and risk rankings rather than calibrated probabilities are produced. In the within-project setting, the training modules and the modules to be predicted come from the same project. In the cross-project setting, the model learns from the source project and ranks modules in the target project. The learning objective is to obtain a mapping f whose output defect probabilities or risk rankings retain high discriminability under a constrained budget.

3.2 Overall Framework

Figure 1: Overall Framework of ProtoStar for Software Defect Prediction Under Label Scarcity



The end-to-end workflow of ProtoStar is shown in Figure 1 and consists of six stages: feature-based deduplication and grouped honest splitting, leakage-preventing preprocessing, parallel training of the few-shot prototype engine and supervised baselines, label-budget-adaptive fusion decisions, validation-set threshold search and freezing, and one-time test-set evaluation or cold-start risk stratification. The prototype engine and supervised baselines share the same preprocessed training pool. The fusion layer automatically selects the final predictor based on out-of-fold performance on the validation set, thereby maintaining robustness as the label budget changes.

3.3 Feature-Based Deduplication, Grouping, and Splitting

NASA data contain duplicated records with identical features and, in some cases, conflicting labels. If rows are split randomly, records from the same duplicate group may appear in both the training and test sets, allowing the model to obtain artificially high-test metrics simply by memorizing these records. Therefore, the framework first rounds the feature vector of each record to 10 decimal places and uses it as a key, binding records with the same key into one duplicate group. If labels within a group disagree, a majority vote is used; a group is labeled defective when the proportion of positive instances is at least 0.5. Grouped splitting is then

© 2026 The Author(s). Authors retain copyright in their articles.

performed under the constraint that an entire group can appear on only one side of a split. Approximately 70% of the groups are first assigned to the test set, and approximately 15% of the remaining groups are assigned to the validation set. The rest form the training set, achieving three-level isolation among training, validation, and testing. The relatively large test proportion reflects the cold-start reality of a small labeled set and a large set awaiting prediction. The framework also retains an ordinary random split as a control for quantifying overestimation caused by duplicate leakage.

3.4 Leakage-Preventing Preprocessing

All preprocessing parameters are estimated only from the training set and are then applied unchanged to the validation and test sets. Missing values are imputed using the median of each training-set feature. To compress the long tail of skewed metrics while preserving the sign, the imputed values are transformed elementwise as sign(x)·log(1+|x|). When the number of features exceeds 20, analysis-of-variance-based SelectKBest with f_classif scoring retains the 20 dimensions most strongly related to the labels; the selector is likewise fitted only on the training set. This process ensures that no information from test modules participates in feature engineering, eliminating information leakage from the workflow.

3.5 Few-Shot Prototype Engine

The few-shot prototype engine is the core of the framework and consists of four parts: a feature encoder, a joint training objective, prototype classification, and bagging.

(1) Encoder. ProtoEncoder is a two-layer feedforward network that maps the d-dimensional input through 128 hidden units with rectified linear unit (ReLU) activation and dropout of 0.3 to a 64-dimensional embedding f_θ(x). Before training, the training pool is standardized. In the standard setting, SMOTE with interpolation among the five nearest neighbors of the minority class balances the positive and negative samples to the same size.

(2) Joint training objective. To address both class imbalance and cluster compactness, the encoder and a linear classification head are trained jointly. The loss is a weighted sum of focal loss and supervised contrastive loss. Let p_t denote the predicted probability of the true class for a sample. Focal loss is defined as:

$$L_{\text{Focal}} = -\alpha(1-p_t)^{\gamma}\log(p_t)$$

The modulating factor (1-p_t)^γ reduces the weight of easy samples, causing training to focus on difficult defective samples. Supervised contrastive loss is defined on the normalized embedding z, and temperature τ controls the attention assigned to difficult negative samples:

$$L_{\text{SupCon}} = -\frac{1}{N}\sum_{i}\frac{1}{|P(i)|}\sum_{p\in P(i)}\log\frac{\exp(\text{sim}(z_i,z_p)/\tau_s)}{\sum_{a\in A(i)}\exp(\text{sim}(z_i,z_a)/\tau_s)}$$

In the equation, P(i) is the set of other samples in the same class as sample i. The final joint objective is:

$$L = L_{\text{Focal}} + 0.5 \times L_{\text{SupCon}}$$

The encoder is trained with Adam for 80 epochs using a learning rate of 0.001 and weight decay of 0.0001. The purpose of introducing the supervised contrastive term is as follows. Focal loss alone optimizes only the classification hyperplane, making it difficult to form stable class centers when samples are extremely scarce. After the contrastive constraint is added, same-class embeddings cluster in the metric space and different-class embeddings separate, making the subsequent prototypes more reliable.

(3) Prototype classification. For support set S, the prototype for class c is the mean embedding of the support samples in that class:

$$c_k = \frac{1}{|S_k|}\sum_{x_i\in S_k} f_\theta(x_i)$$

© 2026 The Author(s). Authors retain copyright in their articles.

After the encoder obtains the embedding of query sample x, its cosine similarity with the two prototypes is calculated. The similarities are scaled by temperature coefficient κ and passed through softmax to obtain the defect probability:

$$p(y=k|x)=\frac{\exp(10\cos(f_\theta(x),c_k))}{\sum_{j\in\{0,1\}}\exp(10\cos(f_\theta(x),c_j))}$$

(4) Bagging. A single episode is sensitive to support-set sampling. The framework independently runs B=15 episodes, each with a different random seed and support-set sample, and averages their defect probabilities as the final probability of the few-shot engine to reduce variance.

3.6 Supervised Baselines and Stacking Ensemble

To provide strong controls and support fusion, the framework trains two supervised models on the same training pool: XGBoost with 100 trees, a maximum depth of 4, a learning rate of 0.1, and class weighting through scale_pos_weight according to the positive-to-negative sample ratio; and a random forest with 100 trees and class_weight=balanced. Fixed stacking concatenates the probabilities from the few-shot engine, XGBoost, and random forest into meta-features and uses logistic regression with a maximum of 2,000 iterations as the meta-learner to output the fused probability.

3.7 Label-Budget-Adaptive Fusion

Fixed stacking is usually more accurate when labels are sufficient. However, when the label budget is constrained, the meta-learner sees only a very small number of validation samples and can easily overfit, producing worse results than direct use of the few-shot engine. Therefore, this study proposes label-budget-adaptive fusion. Its core is to compare the two options without bias on the validation set and select the better option conservatively.

To avoid within-sample optimistic bias caused by training and scoring on the same validation set, the framework performs five-fold cross-prediction on the stacking meta-features and obtains an out-of-fold probability for each validation sample. After thresholds are searched separately for the out-of-fold stacking probabilities and the few-shot engine probabilities, the average score composed of the F1 score (F1) and Matthews correlation coefficient (MCC) is calculated.

4. Experimental Results and Analysis

4.1 Datasets

The experiments use 13 public NASA MDP datasets from the Predictor Models in Software Engineering (PROMISE) repository: CM1, JM1, KC1, KC3, KC4, MC1, MC2, MW1, and PC1-PC5. Each dataset contains module-level static code metrics, such as Halstead metrics, McCabe complexity, and lines of code, together with binary defect labels. Table 2 presents dataset statistics calculated directly from the original Attribute-Relation File Format (ARFF) files. The 13 datasets contain 51,006 modules and 3,683 defective modules. Dataset size ranges from 125 modules in KC4 to 17,186 in PC5, and the defect rate ranges from 0.41% in PC2 to 48.80% in KC4. This provides sufficient sample diversity for examining the robustness of the method across different scales and degrees of imbalance.

Table 2: Statistics of the 13 NASA MDP Datasets (Calculated from the Original ARFF Files)

Dataset	Modules	Features	Defective modules	Defect rate
CM1	505	40	48	9.50%
JM1	10,878	21	2,102	19.32%
KC1	2,107	21	325	15.42%
KC3	458	40	43	9.39%
KC4	125	40	61	48.80%
MC1	9,466	39	68	0.72%
MC2	161	40	52	32.30%
MW1	403	40	31	7.69%
PC1	1,107	40	76	6.87%

© 2026 The Author(s). Authors retain copyright in their articles.

Dataset	Modules	Features	Defective modules	Defect rate
PC2	5,589	40	23	0.41%
PC3	1,563	40	160	10.24%
PC4	1,458	40	178	12.21%
PC5	17,186	39	516	3.00%
Total	51,006	—	3,683	7.22%

4.2 Experimental Settings and Hyperparameters

All methods are implemented in Python using PyTorch and scikit-learn. Grouped splitting, as described in Section 3.3, is used as the primary protocol, while random splitting is used as the control. Few-shot, cold-start, and cross-project experiments are repeated with five random seeds, and the mean and variation are reported. Paired metrics for two methods on the same dataset are evaluated using a two-sided Wilcoxon signed-rank test. The number of pairs, median difference as the direction of the effect size, and p-value are reported, with the significance level set to 0.05. The main hyperparameters are fixed according to the system source code and are listed in Table 3. Except for the sensitivity analysis in Section 4.9, no additional tuning is performed outside the validation set, ensuring fair and reproducible comparisons.

Table 3: Experimental Environment and Main Hyperparameter Settings

Component/stage	Parameter	Value
Encoder ProtoEncoder	Network / embedding dimension / dropout	d→128→64 / 64 / 0.3
Optimizer	Algorithm / learning rate / weight decay / epochs	Adam / 1e-3 / 1e-4 / 80
Focal Loss	α / γ	0.75 / 2.0
Supervised contrastive loss	Temperature τ/ joint weight λ	0.1 / 0.5
Prototype classification	Temperature κ/ support set	10 / 5 positive and 5 negative (k positive for k-shot)
Bagging	Number of rounds B	15
SMOTE	Minority-class neighbors	5
Feature selection	Method / retained dimensions	SelectKBest(f_classif) / 20
XGBoost	Trees / maximum depth / learning rate	100 / 4 / 0.1
Random forest	Trees / class weight	100 / balanced
Stacking meta-model	Maximum logistic-regression iterations	2 000
Adaptive fusion	Margin η/ pass-through budget / minimum validation positives	0.05 / 20 / 10
Threshold search	Range / step / objective	[0.05,0.95] / 0.01 / (F1+MCC)/2
Data splitting	Test proportion / validation proportion	Approximately 70% / 15% of training side
Repeated experiments	Random seeds / significance test	5 / two-sided Wilcoxon

4.3 Effect of Deduplicated Grouping on the Evaluation Protocol

Table 4 compares the median area under the receiver operating characteristic curve (AUC) across 13 datasets for eight models under random and grouped splitting. The eight models are one-nearest neighbor (1NN), k-nearest neighbors (KNN), naive Bayes (NB), logistic regression (LR), random forest (RF), Extreme Gradient Boosting (XGB), the few-shot engine, and fusion. For every model, the AUC under random splitting is not lower than that under grouped splitting, with overestimation ranging from 0.004 to 0.033. The 1NN model, which is most sensitive to duplicated samples, shows the greatest overestimation at 0.033, followed by XGBoost at 0.023. The few-shot engine at 0.007 and naive Bayes at 0.004 are affected less. This systematic difference confirms Shepperd et al.'s concern about duplicated NASA records: random splitting does inflate metrics through duplicate leakage across subsets. Grouped splitting provides a more conservative and credible performance estimate. Therefore, all subsequent full-baseline results in this study use grouped splitting.

© 2026 The Author(s). Authors retain copyright in their articles.

Table 4: Comparison of Median AUC Values under Random and Grouped Splitting (13 Datasets)

Model	Random-split AUC	Grouped-split AUC	Overestimation
1NN	0.695	0.662	0.033
KNN	0.760	0.740	0.019
NB	0.817	0.812	0.004
LR	0.809	0.802	0.007
RF	0.813	0.796	0.017
XGB	0.818	0.795	0.023
Few-shot engine	0.816	0.809	0.007
Fusion	0.807	0.801	0.006

4.4 Full Baseline Comparison

Table 5 reports the AUC values of eight models on all 13 datasets under grouped splitting. The dataset appears in the first column of each row, the best value is bold, and the last row gives the median across datasets. Table 6 further summarizes the median F1, AUC, and MCC values of each model. The results show two points that should be reported directly. First, under the standard supervised setting using all training labels, no model performs best on every dataset. Naive Bayes has the highest median AUC at 0.812, followed by the few-shot engine at 0.809. This agrees with Lessmann et al.'s conclusion that differences among leading classifiers are limited and indicates that simple probabilistic models are already strong when low-dimensional numerical metrics and sufficient labels are available. Second, the few-shot engine has the highest median F1 at 0.381, clearly above XGBoost at 0.316 and random forest at 0.294. This indicates that, for threshold-dependent defect identification, the engine provides a better balance between precision and recall. The comparative advantage of the few-shot engine becomes fully evident mainly under label scarcity. This is why Sections 4.5 and 4.6 specifically examine k-shot and cold-start experiments, and it also supports the need for budget-adaptive predictor selection.

Table 5: AUC Comparison of Eight Models on 13 Datasets under Grouped Splitting (Best Values in Bold)

Dataset	1NN	KNN	NB	LR	RF	XGB	Few-shot engine	Fusion
CM1	0.612	0.701	0.751	**0.829**	0.730	0.746	0.809	0.776
JM1	0.596	0.664	0.710	**0.735**	0.713	0.712	0.730	0.724
KC1	0.641	0.740	**0.812**	0.798	0.796	0.795	0.809	0.805
KC3	0.697	0.758	0.848	0.816	0.807	0.814	**0.849**	0.832
KC4	0.728	0.800	0.814	0.791	0.827	**0.835**	0.807	0.801
MC1	0.693	0.776	**0.912**	0.841	0.896	0.904	0.856	0.887
MC2	0.662	0.636	0.687	0.657	0.667	0.659	**0.693**	0.621
MW1	0.645	0.734	0.751	0.712	**0.770**	0.749	0.763	0.769
PC1	0.673	0.758	0.719	0.786	0.758	0.763	**0.800**	0.782
PC2	0.609	0.732	**0.902**	0.866	0.845	0.880	0.887	0.861
PC3	0.626	0.723	0.787	**0.802**	0.754	0.761	0.798	0.781
PC4	0.767	0.867	0.844	0.918	0.905	0.919	0.919	**0.924**
PC5	0.772	0.885	0.941	0.945	0.935	0.947	0.956	**0.956**
Median	**0.662**	**0.740**	**0.812**	**0.802**	**0.796**	**0.795**	**0.809**	**0.801**

Note: The data were aggregated from Table 2_Baseline Comparison.csv; the last row is the median across the 13 datasets.

Table 6: Cross-Dataset Median F1, AUC, and MCC Values under Grouped Splitting

Model	Median F1	Median AUC	Median MCC
1NN	0.352	0.662	0.249
KNN	0.359	0.740	0.211
NB	0.344	0.812	0.251
LR	0.366	0.802	0.281
RF	0.294	0.796	0.214
XGB	0.316	0.795	0.242
Few-shot engine	0.381	0.809	0.262
Fusion	0.333	0.801	0.249

© 2026 The Author(s). Authors retain copyright in their articles.

4.5 k-Shot Few-Shot Experiments

The k-shot experiments use only k defective positive samples and five negative samples for training, simulating the early project stage in which only a very small number of defects have been confirmed. Figure 2 shows the F1 and AUC curves of the few-shot engine and XGBoost as k changes. Table 7 gives the results of two-sided Wilcoxon tests at k=5, 10, and 20. Under all three budget levels, the few-shot engine significantly outperforms XGBoost in both F1 and AUC. At k=5, the median F1 difference is 0.079 (p=0.002), and the median AUC difference is 0.065 (p<0.001). At k=10, the median F1 difference is 0.078 (p=0.002), and the median AUC difference is 0.068 (p<0.001). At k=20, the median F1 difference is 0.071 (p=0.039), and the median AUC difference is 0.040 (p=0.016). As k increases, the absolute performance of both models increases and the gap narrows, which is consistent with the expectation that conventional supervised models catch up as more labels become available. However, in the extremely low-budget range of k≤20, the advantage of the prototype engine remains stable and significant, confirming the label efficiency of metric learning.

Figure 2: Changes in F1 and AUC with the Number of Labeled Samples k for the Few-Shot Engine and XGBoost in the k-Shot Setting

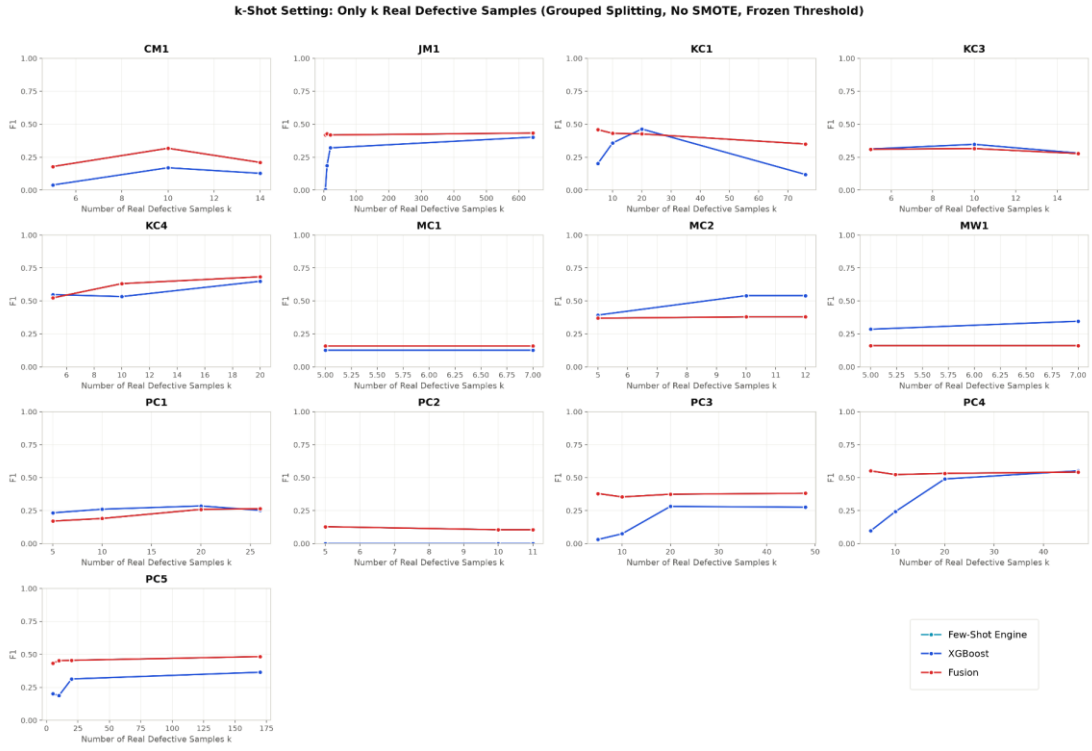


Table 7: Wilcoxon Tests of the Few-Shot Engine Relative to XGBoost in the k-Shot Setting

Budget level	Metric	Paired datasets	Median difference	p-value	Conclusion
k=5	F1	13	+0.079	0.002	Significantly better
k=5	AUC	13	+0.065	<0.001	Significantly better
k=10	F1	11	+0.078	0.002	Significantly better
k=10	AUC	11	+0.068	<0.001	Significantly better
k=20	F1	7	+0.071	0.039	Significantly better
k=20	AUC	7	+0.040	0.016	Significantly better

Note: Data are from Table 8_Significance Test.csv. The number of pairs decreases with the number of available positive samples. The median difference is the few-shot engine minus XGBoost.

4.6 Total-Budget-Constrained Cold-Start Experiments

The cold-start experiments limit the total labeling budget to k positive samples and 5k negative samples, where k=5, 10, or 20. Labels for both classes are scarce and no oversampling is used, making the setting closer to a real new project. Figure 3 shows the change in AUC across budget levels, and Table 8 gives the test results. The few-shot engine is consistently and significantly better in AUC at all three levels: the median difference

© 2026 The Author(s). Authors retain copyright in their articles.

is 0.043 at k=5 (p<0.001), 0.028 at k=10 (p<0.001), and 0.025 at k=20 (p=0.039). The F1 results must be distinguished accurately. At k=20, the few-shot engine is significantly better, with a median difference of 0.037 and p=0.008. At k=5 (p=0.342) and k=10 (p=0.416), the F1 difference is not significant. This indicates that when labels for both classes are extremely scarce, the prototype engine first improves threshold-independent overall ranking ability, as measured by AUC. Threshold-sensitive F1 improves significantly only after the budget has accumulated to some extent. Because the positive-sample budget does not exceed 20 in this setting, adaptive fusion passes the few-shot engine through directly according to its rule, exactly as designed.

Figure 3: Change in AUC with Budget Level k under Total-Budget-Constrained Cold Start

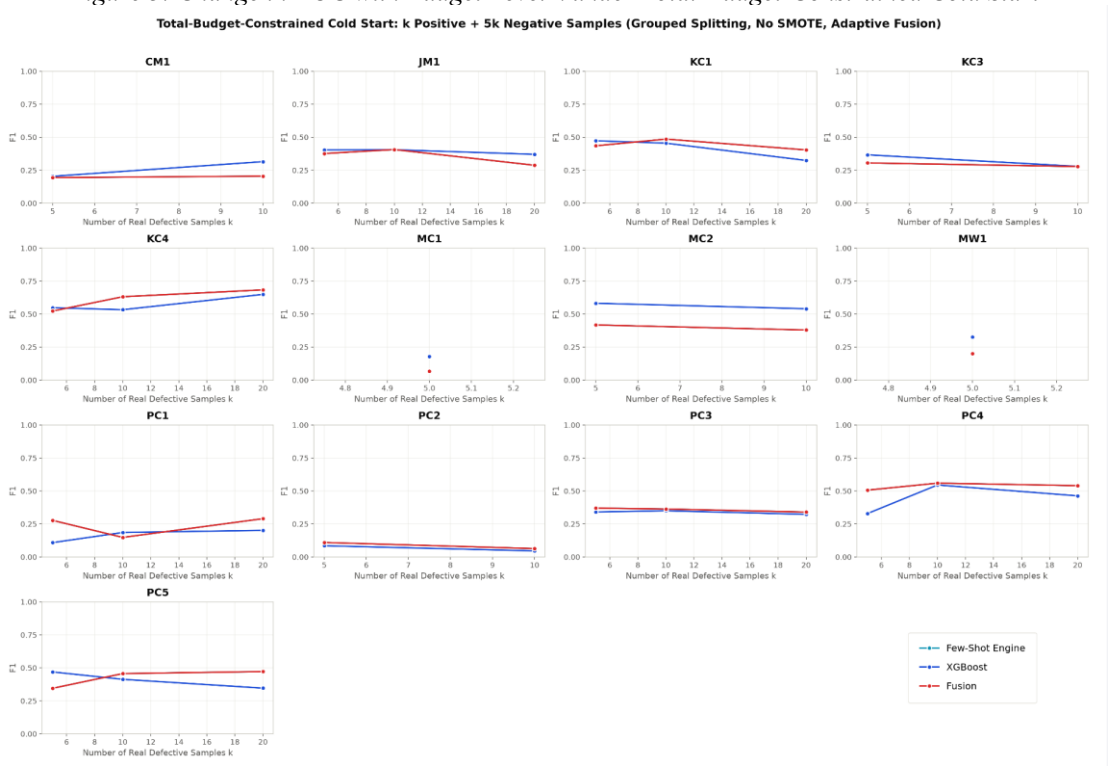


Table 8: Wilcoxon Tests of the Few-Shot Engine Relative to XGBoost under Total-Budget-Constrained Cold Start

Budget level (positive + negative)	Metric	Paired datasets	Median difference	p-value	Conclusion
k=5 (5+25)	F1	13	+0.008	0.342	Not significant
k=5 (5+25)	AUC	13	+0.043	<0.001	Significantly better
k=10 (10+50)	F1	13	+0.001	0.416	Not significant
k=10 (10+50)	AUC	11	+0.028	<0.001	Significantly better
k=20 (20+100)	F1	7	+0.037	0.008	Significantly better
k=20 (20+100)	AUC	7	+0.025	0.039	Significantly better

Note: Data are from Table 8_Significance Test.csv. SMOTE is not used for cold start, and risk rankings are produced.

4.7 Comparison of Label-Budget-Adaptive Fusion Strategies

Table 9 compares fixed stacking, the selection-based adaptive fusion proposed in this study, and direct pass-through of the few-shot engine in terms of median performance and pass-through rate across 13 datasets. Relative to fixed stacking, selection-based adaptive fusion increases median F1 from 0.331 to 0.403, AUC from 0.793 to 0.809, and MCC from 0.249 to 0.262. Paired tests show that the AUC improvement is significant, with a median difference of 0.009 and p=0.020, while the F1 improvement does not reach the 0.05 significance level, with a median difference of 0.019 and p=0.073. Under the standard protocol for this batch of datasets, the selection strategy passes through the few-shot engine in approximately 97% of runs. This indicates that fixed stacking is indeed often impaired by the ensemble in these settings, which mainly involve low-dimensional metrics and limited validation samples. Therefore, the value of adaptive fusion should be understood accurately as a safety net. When the few-shot engine is more stable, adaptive fusion automatically avoids ensemble overfitting and significantly improves ranking. When future data contain sufficient labels and stacking is genuinely superior, out-of-fold selection can switch automatically to the ensemble, providing two-

© 2026 The Author(s). Authors retain copyright in their articles.

way adaptability. This section compares fusion strategies. Separate component-by-component ablation of focal loss, supervised contrastive loss, deduplication, and other individual components has not been conducted in the current experiments. This study therefore does not claim independent contributions for these components, and component-level ablation is left for future work.

Table 9: Median Performance of Three Fusion Strategies on 13 Datasets

Strategy	Median F1	Median AUC	Median MCC	Pass-through rate
Fixed stacking	0.331	0.793	0.249	0.00
Selection-based adaptive fusion (this study)	0.403	0.809	0.262	0.97
Few-shot engine pass-through	0.403	0.809	0.262	1.00

Note: Data were aggregated from Table 3_Label-Budget-Adaptive Fusion.csv. The pass-through rate is the proportion of runs that directly selected the few-shot engine. For selection-based adaptive fusion versus fixed stacking, AUC p=0.020 and F1 p=0.073.

4.8 Cross-Project Cold-Start Risk Ranking

Cold-start ranking ability is evaluated in two cross-project configurations, KC1→JM1 and JM1→KC1. Table 10 reports the Top-20% hit rate, enrichment factor, and AUC averaged over five random seeds under deduplicated grouped splitting. The enrichment factor at each budget level is approximately 1.28-1.76. For example, for KC1→JM1 at k=20, the Top-20% hit rate is 0.389, which is 1.75 times the base rate of 0.222. For JM1→KC1 at k=20, the hit rate is 0.420, which is 1.66 times the base rate of 0.253. The hit rate and enrichment generally increase with k. These results show that, even under cross-project distribution shift, the few-shot engine can still rank defective modules toward the front effectively, which has practical value for allocating testing resources. It should be noted that the enrichment factor appears higher under the non-deduplicated random protocol, reaching approximately 2.37 at most, but this includes duplicate leakage and is therefore optimistically biased. The deduplicated protocol is used as the credible conclusion in this study. The cross-project AUC values are in the moderate range of 0.57-0.65, which also objectively reflects the inherent difficulty of cross-project transfer.

Table 10: Cross-Project Cold-Start Risk-Ranking Results (Deduplicated Protocol, Average over Five Seeds)

Configuration	Budget k	Defect base rate	Top-20% hit rate	Enrichment factor	AUC
KC1→JM1	5	0.222	0.285	1.285	0.569
KC1→JM1	10	0.222	0.361	1.628	0.617
KC1→JM1	20	0.222	0.389	1.755	0.644
KC1→JM1	All	0.222	0.385	1.739	0.626
JM1→KC1	5	0.253	0.357	1.412	0.585
JM1→KC1	10	0.253	0.371	1.468	0.580
JM1→KC1	20	0.253	0.420	1.661	0.628
JM1→KC1	All	0.253	0.420	1.661	0.646

Note: Data are from Table 7_CPDP_Five Seeds.csv. Enrichment factor = Top-20% hit rate / defect base rate.

4.9 Sensitivity Analysis of the Conservative Margin

A sensitivity analysis is conducted using four values of the conservative margin η for adaptive fusion: 0, 0.03, 0.05, and 0.1. Table 11 reports the mean F1, AUC, and pass-through rate across 13 datasets. As η increases, the fusion decision becomes more conservative and the pass-through rate rises from 0.85 to 0.99, while mean F1 at 0.363-0.365 and AUC at 0.819-0.822 remain nearly unchanged. This shows that fusion performance is insensitive to the margin parameter. η mainly adjusts how cautiously stacking is enabled and does not cause performance oscillation within a reasonable range. The default value η=0.05 is a robust choice.

Table 11: Sensitivity Analysis of Conservative Margin η (Average over 13 Datasets)

Margin η	Mean F1	Mean AUC	Pass-through rate
0.0	0.363	0.819	0.85
0.03	0.365	0.819	0.92
0.05	0.364	0.822	0.97
0.1	0.363	0.822	0.99

Note: Data were aggregated from Table 9_Sensitivity Analysis.csv.

© 2026 The Author(s). Authors retain copyright in their articles.

5. Conclusions

This study addresses the common problems of scarce defective samples, severe class imbalance, and high labeling costs in software defect prediction during early project stages and cross-organizational reuse. It proposes and implements ProtoStar, a framework that combines few-shot prototypical learning with label-budget-adaptive ensembles. The framework centers on a few-shot prototype engine jointly trained with focal loss and supervised contrastive loss. It uses an out-of-fold composite score and a conservative margin to select between the few-shot engine and a stacking ensemble in a data-driven manner. It also includes an honest evaluation pipeline with deduplicated grouped splitting, leakage-preventing preprocessing, and validation-set threshold freezing. Systematic evaluation on 13 NASA MDP datasets containing 51,006 modules shows that, under label-scarce settings such as k-shot few-shot learning and total-budget-constrained cold start, the proposed engine achieves significant advantages over the strong supervised baseline XGBoost in F1 and AUC, as confirmed by Wilcoxon tests. The advantage remains stable in the extremely low-budget range of $k \leq 20$. Adaptive fusion significantly improves ranking ability without reducing F1. In cross-project cold start, the defect-hit rate among the top 20% high-risk modules is enriched by 1.28-1.76 times relative to the base rate, and overall performance is insensitive to key hyperparameters. These results show that ProtoStar provides an effective technical approach to software defect prediction under limited labels, with high label efficiency, good robustness, and direct practical applicability. Extension to richer code-semantic features, cross-language data, and component-level ablation will be examined in future work.

References

[1] X. Tian, J. Chang, C. Zhang, et al., Survey of open-source software defect prediction method, *Journal of Computer Research and Development 60* (7) (2023) 1467–1488 (in Chinese). https://doi.org/10.7544/issn1000-1239.202221046.

[2] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, *IEEE Transactions on Software Engineering 33* (1) (2007) 2–13. https://doi.org/10.1109/TSE.2007.256941.

[3] Y. Yang, X. Xia, D. Lo, J. Grundy, A survey on deep learning for software engineering, *ACM Computing Surveys 54* (10s) (2022) Article 206. https://doi.org/10.1145/3505243.

[4] S. Wang, T. Liu, J. Nam, L. Tan, Deep semantic feature learning for software defect prediction, *IEEE Transactions on Software Engineering 46* (12) (2020) 1267–1293. https://doi.org/10.1109/TSE.2018.2877612.

[5] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, *IEEE Transactions on Software Engineering 38* (6) (2012) 1276–1304. https://doi.org/10.1109/TSE.2011.103.

[6] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, *IEEE Transactions on Software Engineering 34* (4) (2008) 485–496. https://doi.org/10.1109/TSE.2008.35.

[7] B. Ghotra, S. McIntosh, A.E. Hassan, Revisiting the impact of classification techniques on the performance of defect prediction models, in: *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015, pp. 789–800. https://doi.org/10.1109/ICSE.2015.91.

[8] N.V. Chawla, K.W. Bowyer, L.O. Hall, W.P. Kegelmeyer, SMOTE: synthetic minority over-sampling technique, *Journal of Artificial Intelligence Research 16* (2002) 321–357. https://doi.org/10.1613/jair.953.

[9] S. Wang, X. Yao, Using class imbalance learning for software defect prediction, *IEEE Transactions on Reliability 62* (2) (2013) 434–443. https://doi.org/10.1109/TR.2013.2259203.

[10] K.E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, MAHAKIL: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, *IEEE Transactions on Software Engineering 44* (6) (2018) 534–550. https://doi.org/10.1109/TSE.2017.2731766.

© 2026 The Author(s). Authors retain copyright in their articles.

[11] T.-Y. Lin, P. Goyal, R. Girshick, K. He, P. Dollár, Focal loss for dense object detection, in: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 2999–3007. https://doi.org/10.1109/ICCV.2017.324.

[12] J. Snell, K. Swersky, R.S. Zemel, Prototypical networks for few-shot learning, in: *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 30, 2017, pp. 4077–4087.

[13] P. Khosla, B. Teterwak, C. Wang, et al., Supervised contrastive learning, in: *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 33, 2020, pp. 18661–18673.

[14] A. Wang, Y. Zhang, H. Wu, K. Jiang, M. Wang, Few-shot learning based balanced distribution adaptation for heterogeneous defect prediction, *IEEE Access 8* (2020) 32989–33001. https://doi.org/10.1109/ACCESS.2020.2973924.

[15] S. Pal, A. Sillitti, Cross-project defect prediction: a literature review, *IEEE Access 10* (2022) 118697–118717. https://doi.org/10.1109/ACCESS.2022.3221184.

[16] Y. Kamei, E. Shihab, B. Adams, et al., A large-scale empirical study of just-in-time quality assurance, *IEEE Transactions on Software Engineering 39* (6) (2013) 757–773. https://doi.org/10.1109/TSE.2012.70.

[17] J. Nam, S.J. Pan, S. Kim, Transfer defect learning, in: *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 382–391. https://doi.org/10.1109/ICSE.2013.6606584.

[18] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction: a large scale experiment on data vs. domain vs. process, in: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009, pp. 91–100. https://doi.org/10.1145/1595696.1595713.

[19] B. Turhan, T. Menzies, A.B. Bener, J. Di Stefano, On the relative value of cross-company and within-company data for defect prediction, *Empirical Software Engineering 14* (5) (2009) 540–578. https://doi.org/10.1007/s10664-008-9103-7.

[20] M. Shepperd, Q. Song, Z. Sun, C. Mair, Data quality: some comments on the NASA software defect data sets, *IEEE Transactions on Software Engineering 39* (9) (2013) 1208–1215. https://doi.org/10.1109/TSE.2013.11.

[21] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, *IEEE Transactions on Software Engineering 43* (1) (2017) 1–18. https://doi.org/10.1109/TSE.2016.2584050.

Funding

This research received no external funding.

Conflicts of Interest

The authors declare no conflict of interest.

Acknowledgment

This paper is an output of the science project.

Copyrights

Copyright for this article is retained by the author (s), with first publication rights granted to the journal. This is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/4.0/).

© 2026 The Author(s). Authors retain copyright in their articles.