Enhancing Deployment-Time Predictive Model Robustness for Code Analysis and Optimization

Huanting Wang
School of Computer Science,
University of Leeds
Leeds, United Kingdom
schwa@leeds.ac.uk

Patrick Lenihan
School of Computer Science,
University of Leeds
Leeds, United Kingdom
p.j.lenihan1@leeds.ac.uk

Zheng Wang
School of Computer Science,
University of Leeds
Leeds, United Kingdom
z.wang5@leeds.ac.uk

Abstract

Supervised machine learning techniques have shown promising results in code analysis and optimization problems. However, a learning-based solution can be brittle because minor changes in hardware or application workloads – such as facing a new CPU architecture or code pattern – may jeopardize decision accuracy, ultimately undermining model robustness. We introduce Prom, an open-source library to enhance the robustness and performance of predictive models against such changes during *deployment*. Prom achieves this by using statistical assessments to identify test samples prone to mispredictions and using feedback on these samples to improve a deployed model. We showcase Prom by applying it to 13 representative machine learning models across 5 code analysis and optimization tasks. Our extensive evaluation demonstrates that Prom can successfully identify an average of 96% (up to 100%) of mispredictions. By relabeling up to 5% of the Prom-identified samples through incremental learning, Prom can help a deployed model achieve a performance comparable to that attained during its model training phase.

CCS Concepts: • **Software and its engineering → Software reliability**; • **Computing methodologies → Artificial intelligence**.

Keywords: Model reliability, Statistical assessment, Machine learning

ACM Reference Format:
Huanting Wang, Patrick Lenihan, and Zheng Wang. 2025. Enhancing Deployment-Time Predictive Model Robustness for Code Analysis and Optimization. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708959

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708959

1 INTRODUCTION

Supervised machine-learning (ML) is a powerful tool for code analysis and optimization tasks like bug detection [28, 39, 73, 75] and runtime- or compiler-based code optimization [18, 19, 52, 53, 68, 74, 76–78, 82, 84]. ML works by training a predictive model from training samples and then applying the trained model to *previously unseen* programs within operational environments that often have diverse application workloads and hardware [32, 79].

ML solutions, while powerful, can be fragile. Small changes in hardware or application workloads can reduce decision accuracy and model robustness [59]. This often arises from "*data drift*" [51, 70], where the training and test data distributions no longer align. Data drift can occur when the assumption that past training data reflects future test data is violated. In code optimization, this can result from changes in workload patterns, runtime libraries, or hardware microarchitectures. It is a particular challenge for ML-based performance optimizations, where obtaining sufficient performance training data is difficult [20, 48].

Existing efforts to enhance ML robustness for code optimization have predominantly focused on improving the learning efficiency or model generalization during *the design time*. These approaches include synthesizing benchmarks to increase the training data size [15, 20, 69], finding better program representations [18, 73, 75, 82], and combining multiple models to increase the model's generalization ability [43, 65, 75]. While important, these design-time methods are unlikely to account for all potential changes during deployment [42]. Although there has been limited exploration into the validation of model assumptions [25] for runtime scheduling, existing solutions assume a specific ML architecture and lack generalizability.

We introduce Prom, an open-source toolkit designed to address data drift *during deployment*, specifically targeting code optimization and analysis tasks. Prom is not intended to replace design-time solutions but to offer a complementary approach to improve ML robustness during deployment. Its primary objective is to ensure the reliability of an already deployed system in the face of changes and support continuous improvements in the end-user environment. To this end, Prom offers a Python interface for training and deploying

supervised ML models, focusing on detecting data drift post-deployment. Model and application developers can integrate Prom into the ML workflow by implementing its abstract class, usually requiring just a few dozen lines of code.

Prom adopts the emerging paradigm of *prediction with rejections* [11, 37, 63, 83], which identifies instances where predictions may be inaccurate, allowing for corrective measures when data drift occurs. For instance, an ML-based performance tuner can notify users when the model prediction, like the compiler flags to be used for a given program, might not yield good performance, prompting them to use alternative search processes [7, 74] to find better solutions [35]. Likewise, a bug detector can alert users to potential false positives for expert inspection. Essentially, this capability allows using alternative metrics when predictive model performance deteriorates. By flagging likely mispredictions, Prom supports continuous learning by using these mispredicted instances as additional training samples to enhance model performance in a production environment.

To evaluate whether a predictive model may mispredict a test input, Prom computes the *credibility* and *confidence* scores of the prediction during deployment. Credibility measures the likelihood that a prediction aligns with the learned patterns. High credibility means the test input is highly consistent with the training data, suggesting the model's prediction is likely to be reliable. Conversely, the confidence score estimates the model's certainty in its prediction. Prom uses the two scores to determine whether the model's outcome should be accepted or requires further investigation. Our intuition is that a prediction is reliable only if the model shows high confidence in its predictions and these predictions, along with the model's confidence level, are credible.

Prom employs conformal prediction (CP) theory [10] to assess the test input's *nonconformity* to compute the confidence and credibility scores of a prediction. Nonconformity is measured by comparing the test input against samples from a *calibration dataset* held out from the model training samples. The idea is to observe the ML model's performance on the calibration set and then evaluate the test input's similarity (or "strangeness") to the calibration samples.

Prom draws inspiration from recent advances in applying CP to detect drifting samples in malware prediction [11, 37] and wireless sensing [83]. While CP has shown promise in these domains, its effectiveness for code optimization tasks remains unclear. This paper presents the first application of CP to code optimization tasks like loop vectorization and neural network code generation. Doing so requires addressing several key limitations of existing approaches. One major drawback of prior methods is they rely on the entire calibration dataset to compute the nonconformity of test samples, which is ill-suited for code with diverse patterns. For example, if we want to estimate the model's accuracy on a computation-intensive program, including many calibration samples with different characteristics (e.g., memory-bound benchmarks) can mislead and bias the credibility estimation. Furthermore, previous solutions employ a monolithic nonconformity function that lacks robustness across different ML models and tasks. They also do not support regression methods and usually require changing the underlying ML model [83]. Prom is designed to overcome these pitfalls.

Unlike prior work [11, 37, 83], Prom adopts an adaptive scheme to measure a test sample's nonconformity. Instead of using the entire calibration dataset, Prom dynamically selects a subset of calibration samples with similar characteristics to the test sample in the feature space defined by the ML model. When computing the nonconformity score, Prom assigns different weights to these selected samples based on their distance from the test sample, giving higher weight to closer samples. This scheme allows Prom to construct a calibration set that closely matches the test sample distribution, thereby improving nonconformity estimation accuracy.

Prom improves conformity reliability by using multiple statistical functions to compute nonconformity scores and applying a majority voting scheme to approve or reject predictions. It is extensible, allowing easy addition of new nonconformity functions. Prom also supports regression by combining CP with clustering algorithms. Unlike [83], it uses a model-free approach instead of learning a probabilistic classifier for data drift detection.

As a potential mitigation of data drift, we showcase that Prom enables a learning-based method to enhance its robustness and maintain reliable performance over time. This is achieved by adopting incremental learning [4, 30] to retrain a deployed model using drifting samples from the production environment. Depending on the specific application, one can relabel a few test samples flagged by Prom and use the relabeled data to retrain the model, by only seeking feedback and user intervention on instances showing data drift. We stress that incremental learning is just one of the possible remedies, but such mitigations hinge on accurately detecting drifted samples - the main focus of this work.

We demonstrate the utility of Prom by applying it to 13 representative ML methods developed by independent researchers for code optimization and analysis [13, 18, 19, 28, 34, 39–41, 62, 75, 82, 84]. Our case studies cover five problems, including heterogeneous device mapping [13, 18, 19, 82], GPU thread coarsening [19, 41, 82], loop vectorization [62, 82], neural network code generation [84] and source-code level bug detection [28, 39, 40, 75]. Experimental results show that Prom can successfully identify an average of 96% (up to 100%) of test inputs where the underlying ML model mispredicts with a false-positive rate[1] of less than 14%. By employing incremental learning, Prom substantially enhances prediction performance in the operational environment, allowing the deployed model to match the performance

[1]This occurs when the ML model gives an accurate prediction, but Prom believes otherwise.

achieved during its design phase. Notably, this improvement is achieved with minimal user intervention or profiling overhead. In our evaluation, Prom requires relabeling fewer than 5% of the samples identified as drifted by Prom, which are then used to update the model through retraining.

This paper makes the following contributions:

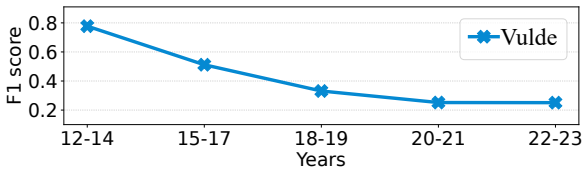
- The first framework to address ML model reliability after model deployment for code analysis and optimization;
- A collection of ready-to-use conformity measurements to support *both* classification and regression models for code optimization and analysis during deployment (Sec. 4);
- A new adaptive weighting scheme and ensemble approach when applying CP to detect data drift (Sec. 5.1.2) and an ensemble approach to detect mispredictions;
- A large-scale study validating the effectiveness of CP in code optimization and analysis tasks (Sec. 7).

2 MOTIVATION

As a motivating example, consider training and using an ML model to detect source code bugs. In this pilot study, we consider Vulde [39], which uses a long short-term memory (LSTM) network for bug detection. Following the original setup, we train and test the model on labeled samples from the common vulnerabilities and exposures (CVE) dataset. We use the open-source release of Vulde and ensure results are comparable to those in the source publication of Vulde.

Figure 1(a) shows what happens if we train Vulde using CVE data collected between 2012 and 2014 and then apply it to real-life code samples developed between 2015 and 2023. This mimics a scenario where the code and bug patterns may evolve after a trained model is deployed. The trained model achieves an F1 score of more than 0.8 (ranging from 0 to 1, with higher being better) when the test and training data are collected from the same time window. However, the F1 score drops to less than 0.3 when tested on samples collected from future time windows. This shows that data drift can severely impact model performance, which was also reported in prior studies [49, 75].

To illustrate code pattern changes, Figures 1(b) and 1(c) show two cases of "*double-free*" vulnerabilities, one from 2012 and one from 2023. Earlier cases were simpler, where the same memory was freed twice (e.g., `name` is freed on lines 6 and 8). A model trained on these samples is unlikely to detect the later, more complex case in Figure 1(c), where a "double-free" occurs due to concurrent threads calling the same buffer-free routine. Because it is difficult to collect a training dataset which covers all possible code patterns seen in deployment time, data drift can happen in the real-life deployment of ML models for code-related tasks.



(a) Data drift leads to deteriorating performance in software vulnerability detection.

```
1  static sftp_parse_attr_3(...) {
2  ssh_string name = NULL;
3  ...
4  if ((name = buffer_get_ssh_string(buf))...)
5      {...}
6  ssh_string_free(name);
7  ...
8  ssh_string_free(name);}
```

(b) CVE-2012-4559: A "*double-free*" for `name` at lines 6 and 8.

```
1  #define NUMT 100
2
3  CURLcode curl_easy_cleanup(...){
4  sts = ...;
5  if(sts) {...
6     hsts_free(sts);}}
7  ...
8  static void* pull_one_url(...){
9  for (i = 0; i < NUMT; i++) {
10     CURL* curl = ...;
11     ...
12     curl_easy_cleanup(curl); }}
13
14 int main(...){...
15 for (i = 0; i < NUMT; i++) {
16     pthread_create(pull_one_url,...);
17 ...}
```

(c) CVE-2023-27537: A potential "*double-free*" due to multiple concurrent threads can invoke `hsts_free` at line 6 at the same time.

Figure 1. Motivation example: impact of data drift on ML models for code vulnerability detection.

3 BACKGROUND

3.1 The Need of Credibility Evaluation

To detect unreliable prediction outcomes, a straightforward approach is to analyze probability distribution given by an ML model. For instance, a high prediction probability for a specific label might suggest high confidence in the prediction. However, this alone does not fully capture the prediction's reliability, as ML models can produce skewed probabilities for rarely seen samples. Consider multi-class classification for example. An ML model predicts the likelihood, r^i, that a given input belongs to each class (c_1 to c_n). However, if the input's pattern significantly differs from training samples, the model might assign a low probability to some classes (e.g., $r^1 \approx 0.0$ for class c_1). This can disproportionately inflate the probabilities of other classes (r^2 to r^n) as the sum

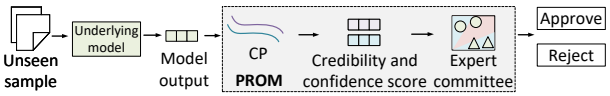


Figure 2. Workflow of PROM during deployment.

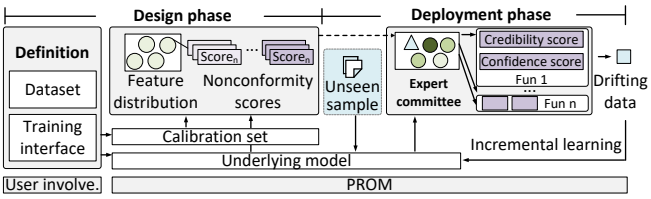


Figure 3. At design time, PROM splits the training data into training and calibration sets. During deployment, it calculates credibility and confidence scores, using majority voting to detect drifting samples. These samples can then be labeled for model updates via offline incremental training.

of probabilities needs to equal 1.0 [12]. In this case, a high probability does not equate to high prediction confidence. Therefore, assessing the model's credibility requires an approach that evaluates how well the input aligns with the training data.

3.2 Statistical Assessment

PROM uses statistical assessments to evaluate prediction credibility and confidence. Unlike typical probabilistic evaluations in ML models, which assess the likelihood of a test sample belonging to a certain class or value in isolation, statistical assessments draw from historical data distributions. They answer questions like: "*How likely is the test sample to belong to a class compared to all other possible classes*"? By framing the sample within the broader context of historical decisions and probabilistic distributions, statistical assessments quantify the uncertainty of a prediction.

3.3 Conformal Prediction

PROM is built on conformal prediction (CP) [5, 10], which, given a model g and a significance level, defines a prediction region that contains the true value with a certain probability. CP constructs this region based on training data distribution, accounting for noise and variability. CP was designed to improve prediction coverage by calculating a prediction range. PROM utilizes CP, *for a different purpose:* evaluating the reliability of a model prediction.

P-value. PROM uses the p-value [66], given by CP, to assess the prediction credibility in its predictions and the confidence of the credibility. We use it to quantify evidence that contradicts a null hypothesis. For instance, in determining the *confidence* of a classifier's prediction, the null hypothesis would assert that there is no substantial difference between the predicted class and any other class, as observed from the model's probability distribution. Likewise, in assessing the *credibility* of a prediction, the null hypothesis assumes that the prediction does not fall within a specific prediction region computed by CP. In PROM (see also Sec. 5.1.2), a high p-value suggests that the likelihood of the observed data under the null hypothesis is small, providing strong evidence against it. Conversely, a low p-value indicates a high likelihood of the observed data under the null hypothesis, thus offering weaker evidence against it. In other words, a high p-value suggests that the prediction is reliable.

```python
from prom import ModelInterface

class ModelDefinition(ModelInterface):
    def __init__(self, model, cali_dataset):
        # Setting the underlying model
        self.model = model
        self.calibration_data = cali_dataset
        super().__init__()

    def data_partitioning(self, dataset,
         calibration_ratio=0.1):
        # Splitting the training data into
             training and calibration sets
        ...
        return training_data, calibration_data

    def predict(self, X, significant_level=0.1):
        ...
        # Underlying model prediction function
             also returns a probabilistic vector
        pred, probability = self.model.predict(X)
        #Call the expert committee
        drifted = self.classifier(probability,
             significant_level)
        return pred, drifted

    def feature_extraction(self, X):
        # Convert the model input into a feature
             vector
        ...
        return self.model.feature_extraction(X)

if __name__ == '__main__':
  model = ModelDefinition(mymodel, dataset)
  pred, drifted = model.predict(sys.argv[1])
```

Figure 4. Simplified code template of PROM.

4 OVERVIEW of PROM

Figure 2 illustrates how PROM can enhance learning-based methods *during deployment*. Users of PROM are ML model or application developers. PROM requires no change to a user model's structure and working mechanism. In this paper, we refer to the user model as the "*underlying model*".

User involvement. For a given input, the underlying model works as it would without PROM during inference. Users of

Prom need only provide the model training dataset and the training interface to Prom.

Added values of Prom. Prom serves as an open-source framework that provides: an adaptive scheme for constructing the calibration set; a collection of conformal prediction methods that take as input the intermediate results (e.g., probabilities of each class label produced by the underlying model) to compute a credibility and confidence score, which is used to suggest whether to accept the prediction outcome; and an ensemble approach to detect mispredictions.

Scope. Prom goes beyond a standard CP library [44, 64] and is not limited to code-related tasks. It tackles a key challenge in ML for code analysis and optimization - insufficient data for training robust models at design time - by offering an orthogonal approach to enhance model reliability during deployment. This makes it particularly useful for ML in code analysis and optimization.

4.1 Implementation

We implemented Prom as a Python package, offering an API for use during both the ML model design and deployment phases. Prom provides interfaces to assess framework setup (Sec. 5.2), automatically searches for hyperparameter settings on the training and calibration datasets, and provides examples to showcase its utilities. These include all the case studies and ML models used in this work (Sec. 6) and simpler examples for beginners. Prom supports classification and regression methods built upon classical ML methods (e.g., support vector machines) and more recent deep neural networks. Figures 3 and 4 provide an overview of Prom's role during the model design and deployment phases, described as follows.

4.1.1 Model design phase.

As depicted in Figure 4, using Prom requires overwriting a handful of methods in Prom's `ModelDefinition` class and exposing the underlying model's internal outputs.

Training data partitioning. Prom is based on split CP [5, 10], which divides the training data into a "*training dataset*" and a "*calibration dataset*". Prom uses the calibration dataset to detect drifting test samples during deployment. By default, it randomly sets aside 10% of the training data (up to 1,000 samples) for calibration, a method shown to be effective in prior work [5, 72]. Prom also offers a way to assess the suitability of the calibration dataset (Sec. 5.2), or users can provide their own holdout calibration dataset.

Changes to user models. For classification tasks, the user model should implement a prediction function (line 15) that returns both a prediction and a probability vector. Most ML classifiers already associate probabilities with each class, which can be easily accessed. Popular frameworks like scikit-learn, PyTorch, and TensorFlow directly provide probability distributions. For example, scikit-learn's `predict_proba` method offers probabilistic values for 33 common ML models. In neural networks, probabilities can usually be extracted from the hidden layer before the output. For regression models [5, 56], Prom applies a similar approach.

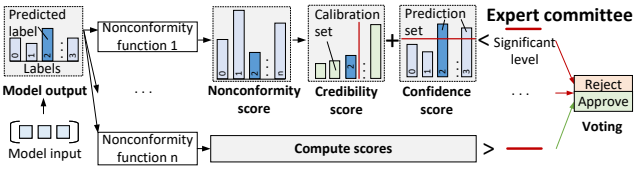


Figure 5. Prom integrates multiple nonconformity functions that vote to reject or approve the ML prediction.

Feature extraction. The user needs to provide a feature extraction function to convert the model input into a feature vector of numerical values. For example, this could be a neural network to generate embeddings of the input [22, 27] or a function to summarize the input programs into numerical values like the number of instructions [74]. Since most ML models already require this function, this requirement should not incur additional engineering effort.

Process calibration dataset. The user model is trained *outside* the Prom framework using any method the user deems appropriate. The trained model is loaded and passed as a Python object to Prom. With the calibration dataset and the trained user model, Prom *automatically* preprocesses the calibration dataset offline before deploying the ML model (Sec. 5). This is done by applying the learned model to each calibration sample and using a nonconformity function described in Sec. 5.1.1 to calculate a score. This score reflects how 'strange' or '*non-conforming*' each calibration example is compared to the learned model.

Significant level. The user can set a *significant level*, $1-\epsilon$, to determine the severity of data drifts. A smaller ϵ can reduce the probability of misprediction by Prom but can lead to a higher false positive rate. By default, Prom sets ϵ to 0.1.

Overwrite the prediction function. As a final step, the model developer needs to overwrite a prediction function (`predict`) function to return the model's prediction for a test input. The original prediction function can be invoked as a subroutine, and the Prom prediction function is used during deployment.

4.1.2 Model deployment phase.

During deployment, the user model functions as usual, taking a test sample and making a prediction. The difference with Prom is that it also suggests whether to accept or reject the prediction. The user can use this outcome to identify mispredictions and provide ground truth, and Prom will use these relabeled drifting samples to update the model.

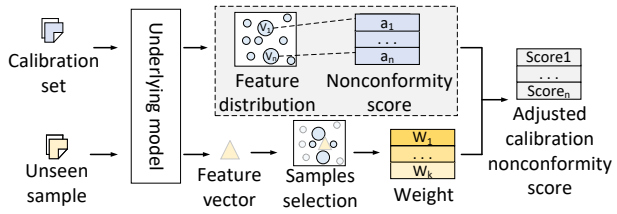


Figure 6. PROM dynamically selects a subset of the holdout calibration dataset to assess the test input's nonconformity.

5 METHODOLOGY

As shown in Figure 5, during deployment, PROM uses multiple (default: 4) nonconformity functions to independently compute the prediction's credibility and confidence scores. These scores are compared to a pre-defined significance level, $1 - \varepsilon$ (Sec. 4.1.1), to decide whether to accept the prediction. If both scores fall below the threshold, the test sample is flagged as drifting. The results are then aggregated using majority voting, where each nonconformity function (expert) decides whether the prediction should be accepted, forming an ensemble "expert committee".

5.1 Nonconformity Measures

PROM computes the p-value of a prediction using nonconformity functions, which are then used to derive credibility and confidence scores. Previous work on using CP to detect drifting samples [11, 37, 83] focuses primarily on classification tasks and does not extend to regression. PROM is the first framework to support both classification and regression for data drift detection. Additionally, prior methods only consider the predicted label, ignoring the probability distribution across labels, whereas PROM accounts for probabilities across all labels in classification tasks (Sec. 3.1).

5.1.1 Nonconformity functions. PROM integrates multiple ready-to-use nonconformity functions, and the choice of nonconformity functions can be customized by passing a list to the relevant PROM interface. By default, PROM uses 4 nonconformity functions: *LAC* [58], *TopK* [6], *APS* [57] and *RAPS* [6]. Other nonconformity functions can be easily incorporated into PROM by implementing an abstract class.

For regression tasks, our nonconformity functions compute the nonconformity score using the residual error between the prediction and the ground truth. Since we do not have the ground truth during deployment, we approximate it using the k-nearest neighbour algorithm [17, 36]. This approximation is based on the null hypothesis that the test sample is similar to those encountered during design time.

Specifically, PROM finds the k-nearest neighbors (we set k to be 3 in this work), denoted as $N_k(n+1)$, of s_{n+1}. The distance is measured by computing the Euclidean distance [21] between the test sample s_{n+1} and calibration samples on the feature space. We then approximate the true value of s_{n+1} by averaging the distance of k-nearest neighbors, $y_{s_{n+1}} = \frac{1}{k}\sum_{i \in N_k(n+1)} y_{s_i}$. The estimated value is then passed to a regression-based nonconformity function to compute the nonconformity score of the test sample. Essentially, we approximate the ground truth by assuming the samples seen at the design time are sufficient to generate an accurate prediction. If this assumption is violated due to drifting test samples, it will likely result in a large residual error (and a greater nonconformity score).

5.1.2 Computing p-value. PROM uses a *p-value* to assess whether a test sample s fits within the prediction region defined by the calibration dataset, which reflects the training data distribution. To compute the p-value, a subset of calibration samples is selected, their nonconformity scores are adjusted, and these scores are used to derive the p-value for the model's prediction.

Calibration nonconformity scores. As shown in Figure 6, PROM dynamically selects a subset of calibration samples to adjust the nonconformity score. Specifically, it computes the Euclidean distance between each calibration sample and the test input based on their feature vectors, sorting the samples by distance. By default, the closest 50% of calibration samples are selected. If the dataset contains fewer than 200 samples, all of them are selected; a threshold that can be configured via the PROM API. This nearest subset of calibration samples is chosen to estimate the nonconformity of the test data relative to the training data. These distances are also used as weights to adjust nonconformity scores.

For a calibration dataset with n samples, $(a_1, a_2, \ldots, a_n)$, PROM computes nonconformity scores and feature vectors $(v_1, v_2, \ldots, v_n)$ offline. For a new test sample s_{n+1}, it extracts the feature vector v_{n+1}, calculates the distances to calibration samples, and selects K samples. The weight w_i for each selected sample i is given by:

$$w_i = \exp\left(-\frac{\|v_i - v_{n+1}\|^2}{\tau}\right), i \in \{1, ..., k\} \quad (1)$$

where $\|v_i - v_{n+1}\|^2$ is the l2-norm, and τ is a temperature hyperparameter (default 500). This weight is used to adjust the nonconformity score: $a_i = w_i \times a_i$.

P-value for classification. After selecting the calibration samples and adjusting their nonconformity scores, PROM calculates the p-value for each test sample. First, it determines the nonconformity score $a_{n+1}^{y^p}$ for the predicted outcome y^p. Then, it evaluates the similarity of the test sample to the chosen calibration samples to compute the p-value, p_{s_i}, as:

$$p_{s_i} = \frac{\text{COUNT}\left\{i \in \{1, ..., n\} : y_i = y^p \text{ and } a_i^{y^p} \geq a_{n+1}^{y^p}\right\}}{\text{COUNT}\ \{i \in \{1, ..., n\} : y_i = y^p\}} \quad (2)$$

This counts the proportion of calibration samples with the predicted label y^p whose nonconformity scores are $\geq$ than the test sample's score. A low p-value (near $1/n$) suggests high nonconformity, meaning the test sample is significantly

different from the training samples. A high p-value (close to 1) indicates strong conformity, showing the test sample closely matches the calibration samples for label y^p.

P-value for regression. We extend classification p-values to regression tasks by generating labels in the calibration dataset using K-means clustering [9]. Specifically, we partition the calibration set into K clusters, $y_1, y_2, \ldots, y_K$, based on the feature vectors of each sample. The optimal number of clusters (K) is determined using the Gap statistic method [67], which compares the within-cluster sum of squares from K-means to that of random clustering over K values from 2 to 20. The Euclidean distance between feature vectors is used as the clustering metric. A larger gap indicates better clustering quality, and Prom selects the K with the highest gap. During deployment, test sample labels are assigned based on the nearest neighbour in the feature space. Prom then computes the *p-value*, as in classification, using Equation 2 during both design and deployment.

5.2 Initialization Assessment

Prom provides a Python function to evaluate whether the framework is properly initialized at design time after obtaining the calibration dataset (Sec. 4.1.1) and the trained underlying model. This is achieved by computing the coverage rate by performing cross-validation on the holdout calibration dataset. Specifically, Prom automatically splits the calibration dataset R times ($R = 3$ by default) into two *internal* datasets for calibration (80%) and validation (20%). It then applies the trained model to the internal validation set and calculates the coverage as:

$$\frac{1}{R}\sum_{j=1}^{R}\frac{1}{n_{\text{val}}}\sum_{i=1}^{n_{\text{val}}}\mathbb{1}\left\{y_i^{(\text{val})} \in C\left(x_i^{(\text{val})}\right)\right\} \approx 1-\varepsilon \tag{3}$$

where n_{val} is the size of the validation set, $y_i^{(\text{val})}$ is the ground truth of the ith validation example, and $C(x_i^{(\text{val})})$ is the prediction region of the ith validation example computed by Prom using the calibration data. The coverage ratio should be approximately the pre-defined significant level, $1-\varepsilon$, with minor fluctuations in deviation [5]. A large deviation indicates an ineffective initialization, which usually stems from a poorly trained or designed underlying model. In this case, Prom will alert the users when the deviation is more than 0.1, enabling them to enhance the underlying model or adjust the significance level during the design time.

A parameter selection function with a grid search algorithm is provided to help users set the optimal parameters automatically, such as the significant level and cluster size (Sec. 5.1.2). After evaluating the candidate parameters on the validation dataset, Prom will save the selected parameters and use them to predict the confidence in the underlying model at deployment time.

5.3 Credibilty and Confidence Evaluation

Credibility score. For each nonconformity function, we use the p-value (Sec. 5.1) computed for the predicted class as the credibility score. The higher the p-value is, the more likely the test sample is similar to the training-time samples, hence a higher credibility score.

Confidence score. Prom estimates the confidence score by evaluating the statistical significance of the prediction using a Gaussian function, $f(x) = e^{-\frac{(x-1)^2}{2\times c^2}}$, where c (default 3) is a constant, and x is the *prediction set size* for the test sample. The prediction set includes labels likely associated with the test sample, where the nonconformity score exceeds the significance level, $1-\varepsilon$. An empty set suggests the test sample is not linked to any known class, while multiple labels indicate uncertainty, resulting in a low confidence score. As with the credibility score, the prediction set is built from the p-value (Sec. 5.1). Regression tasks apply the same approach, using the labels introduced by clustering (Sec. 5.1.2). According to our *prediction with rejections* strategy, a sample is flagged as drifting if both scores fall below the significance level.

5.4 Improve Deployment Time Performance

Prom can enhance the performance of deployed ML systems through incremental learning [4, 30]. For example, suppose a predicted compiler option is likely to be sub-optimal. In that case, the compiler system can use auto-tuning to sample a larger set of configurations to find the optimal one. The idea is to apply other (potentially more expensive) measures to drifting samples. The ground truths can then be added back to the training dataset of the underlying model in a feedback loop for *offline* retraining. Since model retraining occurs only during instances of data drift, it reduces the overhead associated with the collection of training data.

As we will show later, updating a trained model with up to 5% of identified drifting samples significantly enhances robustness post-deployment. The goal is not to reduce training time but to provide a framework for assessing robustness. Without such a system, frequent retraining or risking performance degradation is required. In code optimization tasks, the main expense is labeling data, not training, and by focusing only on mispredicted samples (e.g. for relabeling), our approach reduces labeling overhead and shortens retraining time. By filtering out mispredictions, Prom detects ageing models and supports implementing corrective methods. This, in turn, will improve user experience and trust in ML systems.

6 EXPERIMENTAL SETUP

Evaluation methodology. As shown in Table 1, we apply Prom to detect drifting samples across 5 case studies, covering 13 representative ML models for classification and regression. We faithfully reproduced all methods following

Table 1. Case studies and their setups

#Use cases	#Test methods	Models	Tasks
C1: Thread Coarsening	Magni *et al.* [41] DeepTune [19] IR2Vec [71]	MLP LSTM GBC [45]	Class.
C2: Loop Vectorization	K.Stock *et al.* [62] DeepTune [19] Magni *et al.*	SVM LSTM MLP	Class.
C3: Heterogeneous Mapping	DeepTune [19] Programl [18] IR2Vec [71]	LSTM GNN GBC	Class.
C4: Vulnerability Detection	Vulde [39] CodeXGLUE [40] LineVul [28]	Bi-LSTM Transformer	Class.
C5: DNN Code Generation	Tlp [84]	BERT [22]	Reg.

the methodologies in their source publications and used available open-source code. We adhered to the original training methods to ensure *comparable design-time results*.

Introduce data drift. We introduce changes by separating the training and testing data. We try to mimic practical scenarios by testing the trained model on a benchmark suite not used in the training data or code samples newer than the model training data and the Prom calibration dataset. Note that our primary goal is to detect whether Prom can successfully detect drifting samples, not to improve the design of the underlying model.

Prior practices. Prior work often assumes an ideal scenario by splitting training and test samples at the benchmark or method level, where both sets may share similar characteristics [8]. In contrast, our evaluation introduces data drift to reflect real-world scenarios where workload characteristics change during deployment. As a result, baseline ML models perform worse on test samples than reported in their original publications [13, 31].

6.1 Case Study 1: Thread Coarsening

This problem develops a model to determine the optimal OpenCL GPU thread coarsening factor for performance optimization. Following [41], an ML model predicts a coarsening factor (ranging from 1 to 32) for a test OpenCL kernel, where 1 indicates no coarsening.

Underlying models. We consider three ML models designed for this problem: a Multilayer Perceptron (MLP) used in [41], a long-short-term memory (LSTM) used in DeepTune [19], and a Gradient boosting classifier (GBC) used in IR2Vec [71]. Like these works, we train and test the models using the labeled dataset from [41], comprising 17 OpenCL kernels from three benchmark suites on four GPU platforms.

Methodology. As in [19, 82], we train the baseline model using leave-one-out cross-validation, which involves training the baseline model on 16 OpenCL kernels and testing on another one. We then repeat this process until all benchmark suites have been tested once. To introduce data drift, we train the ML models on OpenCL benchmarks from two suites and then test the trained model on another benchmark suite.

6.2 Case Study 2: Loop Vectorization

This task constructs a predictive model to determine the optimal Vectorization Factor (VF) and Interleaving Factor (IF) for individual vectorizable loops in C programs [34, 47]. Following [34], we explore 35 combinations of VF (1, 2, 4, 8, 16, 32, 64) and IF (1, 2, 4, 8, 16). We use LLVM version 17.0 as our compiler, configuring VF and IF individually for each loop using Clang vectorization directives.

Underlying models. We replicate three ML approaches: K.Stock *et al.* [62] (using SVM), DeepTune [19], and Magni *et al.* [41], which use neural networks. We use the 6,000 synthetic loops from [34], created by changing the names of the parameters from 18 original benchmarks in the LLVM vectorization test suite. We used the labeled data from [82], collected on a multi-core system with a 3.6 GHz AMD Ryzen9 5900X CPU and 64GB of RAM.

Methodology. Following [82], we initially allocate 80% (4,800) of loop programs to train the model, reserving the remaining 20% (1,200) for testing its performance. To introduce data drift, we use programs generated from 14 benchmarks for training and evaluate the model on the programs from the remaining 4 benchmarks.

6.3 Case Study 3: Heterogeneous Mapping

This task develops a binary classifier to determine if the CPU or the GPU gives faster performance for an OpenCL kernel.

Underlying models. We replicated three deep neural networks (DNNs) proposed for this task: DeepTune [19], Programl [18], and IR2Vec [71]. We use the DeepTune dataset, comprising 680 labeled instances collected by profiling 256 OpenCL kernels from 7 benchmark suites.

Methodology. Following [19], we train and evaluate the baseline model using 10-fold cross-validation. This involves training a model on programs from all but one of the subsets and then testing it on the programs from the remaining subset. To introduce data drift, we train the models using 6 benchmark suites and then test the trained models on the remaining suites. We repeat this process until all benchmark suites have been tested at least once.

6.4 Case Study 4: Vulnerability Detection

This task develops an ML classifier to predict if a given C function contains a potential code vulnerability. Following [75], we consider the top-8 types of bugs from the 2023 CWE [26].

Underlying models. We replicated four representative ML models designed for bug detection: CodeXGLUE [40], LineVul [28], both based on Transformer networks, and Vulde [39] which based on a Bi-LSTM network. We evaluate this task with a dataset comprising 4,000 vulnerable C program

samples labeled with one of the eight vulnerability types, each with around 500 samples. The vulnerable code samples cover 2013 and 2023 and are collected from the National Vulnerability Database (NVD), CVE, and open datasets from GitHub.

Methodology. As with prior approaches, we initially train the model on 80% of the randomly selected samples and evaluate its performance on the remaining 20% samples. Then, we introduce data drift by training the model on data collected between 2013 and 2020 and testing the trained model on samples collected between 2021 and 2023.

6.5 Case Study 5: DNN Code Generation

This task builds a *regression-based* cost model to drive the schedule search process in TVM [16] for DNN code generation on multi-core CPUs. The cost model estimates the potential gain of a schedule (e.g., instruction orders and data placement) to guide the search.

Underlying model. We apply Prom to Tlp [84], a cost model-based tensor program tuning method integrated into the TVM compiler v0.8 [16]. We use $2,308 \times 4$ samples collected from 4 Transformer-based BERT models of different sizes in the TenSet dataset [85].

Methodology. For the baseline, we trained and tested the cost model on the BERT-base dataset, where the model is trained on 80% (400K) randomly selected samples and then tested on the remaining 20% (100K)samples. To introduce data drift, we tested the trained model on the other three variants of the BERT model. We ran the TVM search engine for around 8 hours (4,000 iterations) for each DNN on a 12-core 2.7GHz AMD EPYC 9B14 CPU server.

6.6 Performance Metrics

Performance to the oracle. For code optimization tasks (case studies 1 to 3), we compute the ratio of the predicted performance to the best performance obtained by exhaustively trying all options. A ratio of 1.0 means the prediction leads to the best performance given by an "oracle" method.

Misprediction threholds. For code optimization, we consider a prediction to be a misprediction if runtime performance is 20% or more below the Oracle performance (case studies 1–3) or if predicted performance deviates by 20% or more from profiling results (case study 5). For bug detection (case study 4), a misprediction happens when the model misclassifies a test input.

Coverage deviation. This "*smaller-is-better*" metric measures the difference between the confidence level and Prom's true coverage rate on the model. A zero deviation means the coverage rate matches the predefined significance level.

Metrics for data drift detection. We consider the following "*higher-is-better*" metrics for detecting drifting samples:

Table 2. Summary of our main evaluation results

Perf. to the Oracle			Prom performence			
Training	Deploy.	Prom on deploy.	Acc.	Pre.	Recall	F1
0.836	0.544	0.807	86.8%	86.0%	96.2%	90.8%

Accuracy. The ratio of the number of correctly predicted samples to the total number of testing samples.

Precision. The ratio of mispredicted samples that were correctly rejected to the total number of mispredicted samples. This metric answers questions like "*Of all the rejected predictions, how many are actually mispredictions?*". High precision indicates a low *false-positive* rate, meaning Prom rarely incorrectly rejects predictions.

Recall (or Sensitivity). The ratio of mispredicted samples that were correctly rejected to the total number of mispredicted samples. This metric answers questions like "*Of all the mispredicted test samples, how many are rejected by Prom?*". High recall suggests a low *false-negative* rate, indicating Prom can identify most mispredictions.

F1 score. The harmonic mean of Precision and Recall, calculated as $2 \times \frac{Recall \times Precision}{Recall + Precision}$. It is useful when the test data has an uneven distribution of drifting and normal samples. The highest possible F1 score is 1.0, indicating perfect precision and recall.

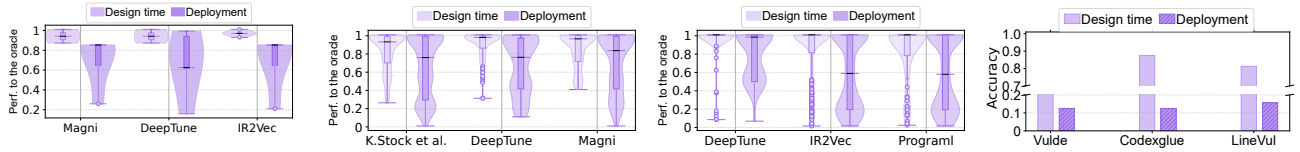
7 EXPERIMENTAL RESULTS

Highlights. Table 2 summarizes the main results of our evaluation. All the tested models were impacted by changes in the application workloads (Sec. 7.1), where the performance relative to the Oracle predictor drops significantly from training time to deployment time. Prom can detect 96.2% of the drifting samples on average (Sec. 7.2). When combined with incremental learning, Prom enhances the performance of deployed models, improving prediction performance by up to 6.6x (Sec. 7.3). Prom also outperforms existing CP-based methods and related work (Sec. 7.5).

7.1 Impact of Drifting Data

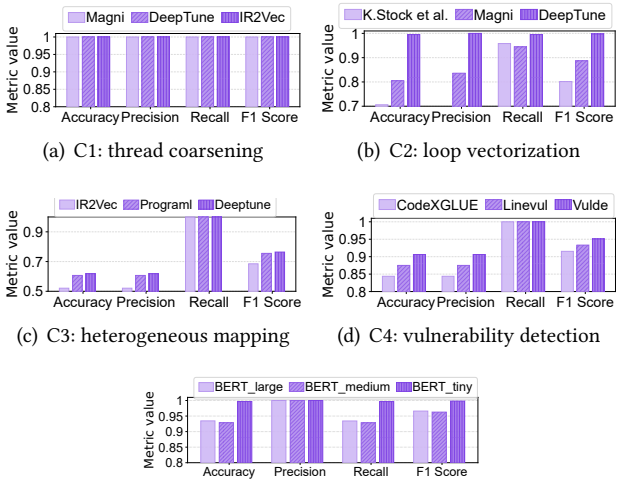
This experiment assesses the impact of data drift by applying a trained model to programs from an unseen benchmark suite or CVE dataset (Sec. 6). For code optimization tasks (case studies 1-3 and 5), we report the performance ratio relative to the oracle (Sec. 6.6). In case study 4, we report the accuracy of bug prediction.

Figure 7 shows the classifiers' performance (case studies 1-4) during design and deployment, while Table 3 presents the regression model's performance (case study 5). The violin diagrams in Figure 7 show the distribution of test sample performance, with the violin's width representing the number of samples in each range. The line inside shows the median, and the top and bottom lines indicate the extremes. Outliers



(a) C1: thread coarsening (b) C2: loop vectorization (c) C3: heterogeneous device mapping (d) C4: vulnerability detection

Figure 7. The resulting performance when using an ML model for decision making. The performance of all learning-based models can suffer during the deployment phase when the test samples significantly differ from the training data.



(a) C1: thread coarsening (b) C2: loop vectorization

(c) C3: heterogeneous mapping (d) C4: vulnerability detection

(e) C5: DNN code generation

Figure 8. Prom's performance for detecting drifting samples across case studies and underlying models (higher is better).

are marked as circles. Ideally, a model's violin would be wide at the top, reflecting good performance for most samples.

Design time performance. For case studies 1-4, we assess design-time performance by holding out 10% of the training samples as a validation set and applying the trained model to it. In case study 5, the training data covers all DNN models, but the model is tested on samples from unseen schedules. This process is repeated 10 times, and the average performance is used as the design-time result. This *ideal setting* assumes that validation and training samples come from the same project or benchmark, with similar workload patterns, yielding comparable results to those reported in the original model publications.

Deployment time performance. An ML model's robustness can suffer during deployment. From Figure 7, this can be observed from the bimodal distribution of the violin shape or a lower prediction accuracy at the deployment stage. From the violin diagrams, we observe a wider violin shape towards the bottom of the y-axis and a lower median value compared to the design-time result. From Table 3, this also can be seen from a lower deployment-time prediction accuracy than the

Table 3. C5: DNN code generation (performance to the oracle ratio) - trained on BERT-base and tested on BERT variants.

Network	BERT-base	BERT-tiny	BERT-medium	BERT-large
Native deployment	0.845	0.224	0.668	0.703
Prom assisted deployment	/	0.794	0.810	0.808

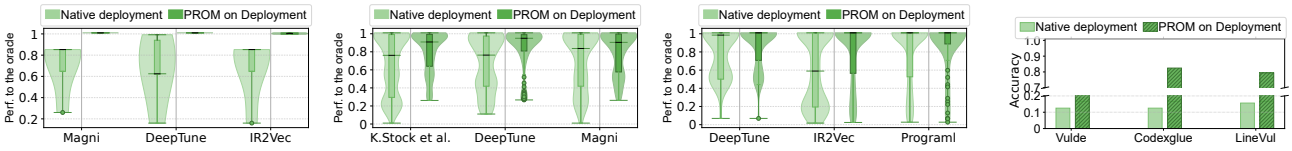
design-time performance. The impact of drifting samples is clearly shown in vulnerability detection of Figure 7(d), where the prediction accuracy drops by an average of 62.5%. For DNN code generation (Table 3), the accuracy of performance estimation can also drop from 84.5% to as low as 22.4%. The results highlight the impact of data drift.

7.2 Detecting Drifting Samples

Figure 8 reports Prom's performance in predicting drifting samples across case studies and underlying models. For all tasks, Prom achieves an average precision of 0.86 with an average accuracy of 0.87. This means it rarely filters out correct predictions. For the binary classification task of heterogeneous mapping (Figure 8(c)), Prom achieves an average F1 score of 0.74. In this case, Prom sometimes rejects correct predictions. This is because the probability distribution of binary classification is often less informative for CP than in multiclass cases [64]. For the regression task of case study 5, Prom can detect most of the drifting samples with a recall of 0.95 and an average precision of 1. Furthermore, the underlying model's quality also limits the performance of Prom. When the information given by the underlying model becomes noisy, Prom can be less effective. Averaged across case studies, in detecting mispredictions, Prom achieves a recall of 0.96, a false-positive rate of 0.14 and a false-negative rate of 0.04, suggesting that Prom is highly accurate in detecting drifting samples.

7.3 Incremental Learning

In this experiment, we use Prom to identify drifting samples and update the underlying models by retraining with a small set of Prom-identified samples. Prom preserves the performance of the methods close to their original levels, as shown by improved accuracy (Figure 9(d)), the performance-to-oracle ratio (Table 3), and violin plots (Figures 9(a) to



(a) C1: thread coarsening (b) C2: loop vectorization (c) C3: heterog. mapping (d) C4: vuln. detection

Figure 9. PROM enhances performance through incremental learning in different underlying models.

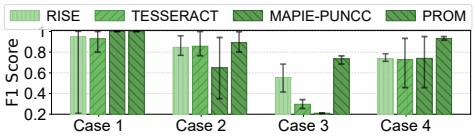


Figure 10. Geometric mean and variances across models of the F1 score in classification tasks.

9(c)), where test sample distributions shift towards higher performance with a better median than native deployment without PROM. Overall, PROM requires labeling at most 5% (sometimes just one) of drifting samples to update the model. Without it, one would need to label random test samples, leading to higher maintenance costs and unnecessary user confirmations for samples the model can predict correctly.

7.4 Individual Case Studies

We now examine case studies showing how PROM improves the underlying model with incremental learning.

Case study 1: thread coarsening. In this experiment, we tested the trained model on kernels from OpenCL benchmarks unseen at the model training stage. Figure 7(a) and 9(a) show that the performance of all ML models drops as the test dataset changes. By relabeling just one drifting sample using incremental learning, PROM improves the performance to the oracle ratio from an average of 77.6% to 99.0% (21.4% improvement) during deployment.

Case study 2: loop vectorization. This experiment introduced changes by testing the underlying model on loops extracted from unseen benchmarks. Figure 7(b) and 9(b) show that drifting data led to a performance reduction for all methods, averaging 9%. Retraining the model using only 5% of the PROM-identified drifting samples helps restore the underlying model's initial performance, leaving only a 1.6% gap to the design-time performance.

Case study 3: heterogeneous mapping. This experiment tests the underlying models on OpenCl kernels from *unseen* benchmarks. Figures 7(c) and 9(c) show that all ML models deliver low performance on *unseen* benchmarks. PROM also successfully detects 100% of the drifting samples (recall) on average with an accuracy rate of 58%. Further, by utilizing incremental learning on 5% of the drifting samples, PROM improved the performance to oracle ratio of all systems from 63.1% to 78.9% (15.8% improvement) on average.

Case study 4: vulnerability detection. In this experiment, we tested a trained model on a vulnerability dataset from a time period not covered by the training data. Figures 7(d) and 9(d) show that all models initially had low prediction accuracy, ranging from 12.5% to 15.6% when facing new code patterns. PROM correctly identified all mispredictions with a recall of 1. By relabeling up to 5% of the PROM-identified drifting samples and updating the model, we improved the accuracy from an average of 13.5% to 72.5%, achieving 95.3% of the design-time performance.

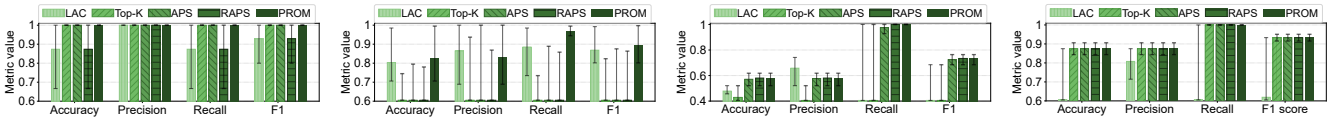
Case study 5: DNN code generation. In this experiment, we apply the cost model trained on the Bert-base dataset to three other variants of the BERT network. Once again, from Table 3, we see the performance of the trained model experiences a reduction from 84.5% to 53.2%. For BERT-tiny, the performance drops as much as 65.3%. PROM can detect 95.3% of drifting data and achieve a precision of 1. After profiling just 5% of the PROM-identified drifting data and using them to retrain the cost model online during the TVM code search process, the average performance of the cost model improves to 80.4%, resulting in a 2.0x enhancement.

7.5 Comparison to CP Libraries and Methods

We compare PROM with MAPIE [64] and PUNCC [44], two CP libraries used for outlier detection, as well as RISE [83] and TESSERACT [49], which use a single nonconformity function. RISE also trains an SVM for misprediction detection but, like TESSERACT, supports only classical classifiers, so we evaluate them on cases 1 to 4. Figure 10 shows average results, with min-max bars indicating variation across models. PROM outperforms TESSERACT by 17.6%, achieving a higher F1 score due to its improved nonconformity strategy. RISE struggles with uneven data or tasks with many labels, while PROM's model-free ensemble approach handles these cases better. Naive CP yields the lowest F1 score, showing its limitations in detecting drift in large-scale tasks.

7.6 Further Analysis

Nonconformity functions. Figure 11 shows the average performance of the four default PROM functions in detecting



(a) Case study 1: thread coarsening (b) Case study 2: loop vectorization (c) Case study 3: heterog. mapping (d) Case study 4: vuln. detection

Figure 11. Performance of individual nonconformity functions. Min-max bar shows the performance across ML models.

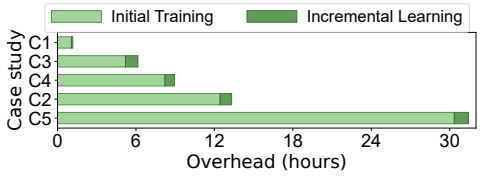
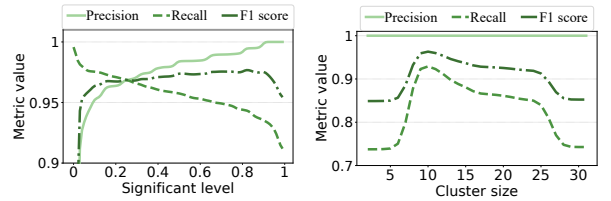
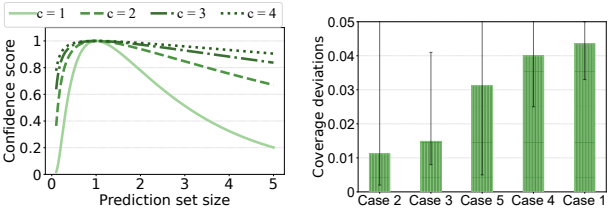


Figure 12. The average training and incremental learning overhead of individual case studies.



(a) PROM performance as the threshold increases in loop vectorization (b) PROM performance as cluster size increases in regression task



(c) PROM performance as Gaussian scale parameter changes (d) Coverage deviations across 5 case studies

Figure 13. Sensitive analysis of PROM hyperparameters.

drifting samples across case studies, with min-max bars indicating variance across models. PROM's ensemble strategy outperforms individual functions in all metrics, demonstrating that no single function performs well across all case studies. By combining multiple functions, PROM enhances the generalization of statistical assessments.

Sensitive analysis. Figures 13(a) to 13(c) show how the significance threshold and other parameters (e.g. cluster size and Gaussian scale) impact PROM's performance for data drift detection. A larger significance threshold reduces false positives, improving precision but potentially lowering recall. PROM automatically determines the optimal cluster size using the Gap statistic method [67]. Deviations from the optimal cluster size affect detection performance. For the Gaussian scale parameter, prediction set sizes smaller or larger than 1 lead to reduced confidence by suggesting too few or too many classes for a sample.

Coverage deviation. Figure 13(d) shows the coverage deviation across five cases. The min-max bar represents the variance across the underlying models. The geomean of deviation is 2.5%, which is a benign fluctuation and indicates a good fit for conformal prediction on the underlying models. The thread coarsening task shows a 4.4% deviation due to the small calibration dataset, which could be mitigated by adding more calibration samples.

Training overhead. Figure 12 reports the overhead of initial model training and incremental learning. Initial training takes several hours to one day, while incremental learning with fine-tuning takes less than one hour on a multi-core CPU or a desktop GPU - a negligible overhead compared to the initial training time.

Runtime overhead. During deployment, the main runtime overhead in PROM comes from computing nonconformity scores and detecting drifting samples. This overhead is minimal: on a low-end laptop, computing confidence and credibility scores and performing drift detection take less than 10 and 2 milliseconds, respectively.

8 DISCUSSIONS

Naturally, there is room for improvement and further work. We discuss a few points here.

Overfitting. To mitigate overfitting, PROM retrains the model using the original dataset and a few PROM-identified mispredictions. Incorporating additional data from the deployment environment likely enhances the model's generalization. In our experiments, cross-validation shows that this retraining approach effectively improves generalization.

Robustness. PROM employs a model-free approach to detect drifting samples, avoiding reliance on supervised learning. The calibration dataset can be updated during incremental learning, and our detection framework is updated during ML model retraining. Thus, PROM can adapt over time to changes in model training samples.

Combining with reinforcement learning. PROM is a good fit for RL in code optimization, which usually requires a cost model to estimate the reward to avoid expensive program profiling. PROM can assess the RL cost model's robustness,

guiding RL to profile only samples likely to be mispredicted by the cost model (similar to case study 5). This profiling information can then be used to update and improve the cost model during the RL search.

Applicability. Prom targets probability-based classifiers [23, 24, 55, 80], making it broadly applicable across ML algorithms. It can also be used during training to evaluate whether adding new data would improve performance, helping reduce costly data labeling for code optimization [48].

Rejection costs. Handling rejected predictions varies by application. Simple cases may involve manual review or other optimization strategies, such as iterative compilation for optimal compiler settings [38, 46]. Rejections incur costs, and Prom allows users to adjust the significance level to balance benefits against these costs. While Prom does not eliminate human intervention, it detects data drift and ageing ML models post-deployment, only requesting user verification when data drift is likely. User feedback is then added to the training dataset to retrain the model.

Privacy. Since Prom only uses information available to the underlying ML model, it should not raise privacy concerns.

Integration with non-Python environments. Prom is easy to integrate into non-Python compilers. For example, for C/C++ code, Prom provides a pyblind11 API [2] to take the probabilistic vector of the model prediction as input and returns a boolean value to suggest whether the prediction should be accepted.

9 RELATED WORK

Supervised ML is a powerful tool for code analysis and optimization [18, 19, 28, 39, 75, 82, 84]. It relies on the assumption that training data will closely resemble future test data [54]. However, this assumption can be violated in deployment environments due to evolving hardware and workloads, leading to compromised ML model robustness [3, 42, 60].

Efforts to enhance ML model robustness during design time include data augmentation through code synthesis [15], learning program representations [18, 75], and tuning ML model architectures [74]. Prom complements these design-time methods by improving trained model performance at deployment without altering the model architecture.

Some recent works evaluate DNN prediction accuracy using test data from the operational environment [33], requiring representative data collection from the deployment environment first. Prom, however, assesses prediction reliability in real-time. Other methods quantify prediction uncertainties using entropy and mutual information [29] or train DNN models to estimate prediction confidence for specific applications [61]. Unlike Prom, which is model-agnostic, these methods depend on specific DNN operators.

Prom focuses on detecting incorrect ML predictions caused by changes in input characteristics, combining anomaly detection [11, 50, 83] and adversarial attack analysis [14]. Grounded in Conformal Prediction (CP), guarantees confident predictions. While standard CP libraries like MAPIE [64] and PUNCC [44] estimate where the ground truth likely lies, Prom uses CP differently—assessing the credibility of predictions to reject unreliable ones. Additionally, it offers a framework to automate CP setup and usage.

Prom improves upon prior CP methods [37, 49, 81, 83], which rely on full calibration datasets and single nonconformity measures. By using adaptive weighting and multiple nonconformity functions, it enhances misprediction detection. Additionally, Prom extends CP beyond classification to support regression tasks.

10 CONCLUSIONS

We have introduced Prom, an open-source library designed to enhance the robustness of learning-based models during deployment for code-related tasks. Prom measures the credibility and confidence of predictions to detect likely mispredictions. We evaluated Prom on 13 ML models across five code optimization and analysis tasks. Prom can effectively detect samples that an ML model can mispredict by successfully identifying an average of 96% of mispredictions. It also improves deployed model performance by updating the trained model with a few identified test samples. As an example of the practical use of Prom, we are working with two companies to integrate PROM into their internal tools to enhance the reliability of ML models for code optimization.

There is growing interest in applying ML to systems research [59, 79], yet prior work has primarily focused on model optimizations during the design phase. Prom addresses post-deployment optimization. While this work focuses on code-related tasks, Prom can be applied to a wider range of problems with a generic API. We hope that Prom will be an enabling technology to support deployment model optimization, thereby enhancing the ML model robustness across various domains.

Data-Availability Statement

The open-source release of Prom is available at https://github.com/HuantWang/PROM. Additionally, an archival copy of our Artifact can be downloaded from [1] with detailed step-by-step instructions for replicating our results using a Docker image can be found at https://github.com/HuantWang/PROM/blob/ae_cgo/AE.md.

Acknowledgments

This work was supported in part by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant agreements EP/X018202/1 and EP/X037304/1. For any correspondence of this work, please contact Zheng Wang (Email: z.wang5@leeds.ac.uk).

References

[1] [n. d.]. PROM: Enhancing Deployment-Time Predictive Model Robustness for Code Analysis and Optimization - Research Paper Artifact. https://doi.org/10.5281/zenodo.14077780

[2] [n. d.]. pybind11. https://github.com/pybind/pybind11.

[3] Samuel Ackerman, Orna Raz, Marcel Zalmanovici, and Aviad Zlotnick. 2021. Automatically detecting data drift in machine learning classifiers. *arXiv preprint arXiv:2111.05672* (2021).

[4] RR Ade and PR Deshmukh. 2013. Methods for incremental learning: a survey. *International Journal of Data Mining & Knowledge Management Process* (2013).

[5] Anastasios N Angelopoulos and Stephen Bates. 2021. A gentle introduction to conformal prediction and distribution-free uncertainty quantification. *arXiv preprint arXiv:2107.07511* (2021).

[6] Anastasios Nikolas Angelopoulos, Stephen Bates, Michael Jordan, and Jitendra Malik. 2020. Uncertainty Sets for Image Classifiers using Conformal Prediction. In *International Conference on Learning Representations*.

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Edmonton, Canada. http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

[8] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[9] David Arthur and Sergei Vassilvitskii. 2007. K-means++ the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. 1027–1035.

[10] Vineeth Balasubramanian, Shen-Shyang Ho, and Vladimir Vovk. 2014. *Conformal prediction for reliable machine learning: theory, adaptations and applications*. Newnes.

[11] Federico Barbero, Feargus Pendlebury, Fabio Pierazzi, and Lorenzo Cavallaro. 2020. Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv* (2020).

[12] Christopher M Bishop. 2006. *Pattern recognition and machine learning*. Springer.

[13] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*. 201–211.

[14] Wieland Brendel, Jonas Rauber, and Matthias Bethge. 2018. Decision-Based Adversarial Attacks: Reliable Attacks Against Black-Box Machine Learning Models. In *International Conference on Learning Representations*.

[15] Ferhat Ozgur Catak, Javed Ahmed, Kevser Sahinbas, and Zahid Hussain Khand. 2021. Data augmentation based malware detection using convolutional neural networks. *Peerj computer science* 7 (2021), e346.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[17] Giovanni Cherubin, Konstantinos Chatzikokolakis, and Martin Jaggi. 2021. Exact optimization of conformal predictors via incremental and decremental learning. In *International Conference on Machine Learning*. PMLR, 1836–1845.

[18] Chris Cummins, Zacharias V Fisches, Tal Ben-Nun, Torsten Hoefler, Michael FP O'Boyle, and Hugh Leather. 2021. Programl: A graph-based program representation for data flow analysis and compiler optimizations. In *International Conference on Machine Learning*. PMLR, 2244–2253.

[19] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-end deep learning of optimization heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[20] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 86–99.

[21] Per-Erik Danielsson. 1980. Euclidean distance mapping. *Computer Graphics and image processing* 14, 3 (1980), 227–248.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[23] Annette J Dobson and Adrian G Barnett. 2018. *An introduction to generalized linear models*. CRC press.

[24] Sonia Domínguez-Almendros, Nicolás Benítez-Parejo, and Amanda Rocío Gonzalez-Ramirez. 2011. Logistic regression models. *Allergologia et immunopathologia* 39, 5 (2011), 295–305.

[25] Murali Krishna Emani and Michael F. P. O'Boyle. 2015. Celebrating diversity: a mixture of experts approach for runtime mapping in dynamic environments. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 499–508. https://doi.org/10.1145/2737924.2737999

[26] Common Weakness Enumeration. 2023. 2023 CWE Top 25 Most Dangerous Software Errors. https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[27] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[28] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[29] Jakob Gawlikowski, Cedrique Rovile Njieutcheu Tassi, Mohsin Ali, Jongseok Lee, Matthias Humt, Jianxiang Feng, Anna Kruspe, Rudolph Triebel, Peter Jung, Ribana Roscher, et al. 2023. A survey of uncertainty in deep neural networks. *Artificial Intelligence Review* 56, Suppl 1 (2023), 1513–1589.

[30] Alexander Gepperth and Barbara Hammer. 2016. Incremental learning algorithms and applications. In *ESANN*.

[31] Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jerónimo Castrillón. 2019. A case study on machine learning for synthesizing benchmarks. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*. ACM, 38–46.

[32] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. 2013. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 1–10.

[33] Antonio Guerriero, Roberto Pietrantuono, and Stefano Russo. 2021. Operation is the hardest teacher: estimating DNN accuracy looking for mispredictions. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 348–358.

[34] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. NeuroVectorizer: end-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[35] Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. 2006. On the impact of data input sets on statistical compiler tuning. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 8–pp.

[36] Vladislav Ishimtsev, Alexander Bernstein, Evgeny Burnaev, and Ivan Nazarov. 2017. Conformal k-NN Anomaly Detector for Univariate Data Streams. In *Conformal and Probabilistic Prediction and Applications*. PMLR, 213–227.

[37] Roberto Jordaney, Kumar Sharad, Kumar Dash Santanu, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. 2017. Transcend: Detecting Concept Drift in Malware Classification Models. In *USENIX Security*.

[38] Toru Kisuki, Peter MW Knijnenburg, and Michael FP O'Boyle. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00622)*. IEEE, 237–246.

[39] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings of the NDSS* (2018).

[40] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[41] Alberto Magni, Christophe Dubach, and Michael O'Boyle. 2014. Automatic optimization of thread-coarsening for graphics processors. In *Proceedings of the 23rd international conference on Parallel architectures and compilation techniques*. 455–466.

[42] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. 2022. Matchmaker: Data drift mitigation in machine learning for large-scale systems. *Proceedings of Machine Learning and Systems* 4 (2022), 77–94.

[43] Vicent Sanz Marco, Ben Taylor, Barry Porter, and Zheng Wang. 2017. Improving Spark Application Throughput via Memory Aware Task Co-Location: A Mixture of Experts Approach. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 95–108. https://doi.org/10.1145/3135974.3135984

[44] Mouhcine Mendil, Luca Mossina, and David Vigouroux. 2023. PUNCC: a Python Library for Predictive Uncertainty Calibration and Conformalization. In *Conformal and Probabilistic Prediction with Applications*. PMLR, 582–601.

[45] Alexey Natekin and Alois Knoll. 2013. Gradient boosting machines, a tutorial. *Frontiers in neurorobotics* 7 (2013), 21.

[46] Ricardo Nobre, Luiz GA Martins, and João MP Cardoso. 2016. A graph-based iterative compiler pass selection and phase ordering approach. *ACM SIGPLAN Notices* 51, 5 (2016), 21–30.

[47] Dorit Nuzman, Ira Rosen, and Ayal Zaks. 2006. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices* 41, 6 (2006), 132–143.

[48] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Minimizing the cost of iterative compilation with active learning. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 245–256.

[49] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *USENIX Security*.

[50] Marco A.F. Pimentel, David A. Clifton, Lei Clifton, and Lionel Tarassenko. 2014. A review of novelty detection. *Signal Processing* 99 (2014), 215–249. https://doi.org/10.1016/j.sigpro.2013.12.026

[51] Joaquin Quinonero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D Lawrence. 2008. *Dataset shift in machine learning*. Mit Press.

[52] Jie Ren, Ling Gao, Hai Wang, and Zheng Wang. 2017. Optimise web browsing on heterogeneous mobile platforms: a machine learning based approach. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[53] Jie Ren, Ling Gao, Xiaoming Wang, Miao Ma, Guoyong Qiu, Hai Wang, Jie Zheng, and Zheng Wang. 2021. Adaptive computation offloading for mobile augmented reality. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 5, 4 (2021), 1–30.

[54] Mauro Ribeiro, Katarina Grolinger, and Miriam AM Capretz. 2015. Mlaas: Machine learning as a service. In *2015 IEEE 14th international conference on machine learning and applications (ICMLA)*. IEEE, 896–902.

[55] Irina Rish et al. 2001. An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, Vol. 3. 41–46.

[56] Yaniv Romano, Evan Patterson, and Emmanuel Candes. 2019. Conformalized quantile regression. *Advances in neural information processing systems* 32 (2019).

[57] Yaniv Romano, Matteo Sesia, and Emmanuel Candes. 2020. Classification with valid and adaptive coverage. *Advances in Neural Information Processing Systems* 33 (2020), 3581–3591.

[58] Mauricio Sadinle, Jing Lei, and Larry Wasserman. 2019. Least ambiguous set-valued classifiers with bounded error levels. *J. Amer. Statist. Assoc.* 114, 525 (2019), 223–234.

[59] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. (2014).

[60] Siddharth Singhal, Utkarsh Chawla, and Rajeev Shorey. 2020. Machine learning & concept drift based approach for malicious website detection. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*. IEEE, 582–585.

[61] Andrea Stocco, Michael Weiss, Marco Calzana, and Paolo Tonella. 2020. Misbehaviour Prediction for Autonomous Driving Systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 359–371.

[62] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. 2012. Using Machine Learning to Improve Automatic Vectorization. *ACM Trans. Archit. Code Optim.* 8, 4, Article 50 (jan 2012), 23 pages. https://doi.org/10.1145/2086696.2086729

[63] Wenlong Tang and Edward S Sazonov. 2014. Highly accurate recognition of human postures and activities through classification with rejection. *IEEE journal of biomedical and health informatics* (2014).

[64] Vianney Taquet, Vincent Blot, Thomas Morzadec, Louis Lacombe, and Nicolas Brunel. 2022. MAPIE: an open-source library for distribution-free uncertainty quantification. *arXiv preprint arXiv:2207.12274* (2022).

[65] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive deep learning model selection on embedded systems. In *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 31–43.

[66] Ronald A Thisted. 1998. What is a P-value. *Departments of Statistics and Health Studies* (1998).

[67] Robert Tibshirani, Guenther Walther, and Trevor Hastie. 2001. Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 63, 2 (2001), 411–423.

[68] Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael FP O'Boyle. 2009. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *ACM Sigplan notices* 44, 6 (2009), 177–187.

[69] Foivos Tsimpourlas, Pavlos Petoumenos, Min Xu, Chris Cummins, Kim Hazelwood, Ajitha Rajan, and Hugh Leather. 2023. BenchDirect: A Directed Language Model for Compiler Benchmarks. In *The 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

[70] Alexey Tsymbal. 2004. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin* 106, 2 (2004), 58.

[71] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (dec 2020), 27 pages. https://doi.org/10.1145/3418463

[72] Vladimir Vovk. 2012. Conditional validity of inductive conformal predictors. In *Asian conference on machine learning*. PMLR, 475–490.

[73] Huanting Wang, Zhanyong Tang, Shin Hwei Tan, Jie Wang, Yuzhe Liu, Hejun Fang, Chunwei Xia, and Zheng Wang. 2024. Combining Structured Static Code Information and Dynamic Symbolic Traces for Software Vulnerability Prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[74] Huanting Wang, Zhanyong Tang, Cheng Zhang, Jiaqi Zhao, Chris Cummins, Hugh Leather, and Zheng Wang. 2022. Automating reinforcement learning architecture design for code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. 129–143.

[75] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. 2021. Combining Graph-Based Learning With Automated Data Collection for Code Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 1943–1958. https://doi.org/10.1109/TIFS.2020.3044773

[76] Yiming Wang, Meng Hao, Hui He, Weizhe Zhang, Qiuyuan Tang, Xiaoyang Sun, and Zheng Wang. 2024. DRLCap: Runtime GPU Frequency Capping with Deep Reinforcement Learning. *IEEE Transactions on Sustainable Computing* (2024).

[77] Zheng Wang and Michael FP O'Boyle. 2009. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 75–84.

[78] Zheng Wang and Michael FP O'Boyle. 2010. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 307–318.

[79] Zheng Wang and Michael O'Boyle. 2018. Machine learning in compiler optimization. *Proc. IEEE* 106, 11 (2018), 1879–1901.

[80] Petros Xanthopoulos, Panos M Pardalos, Theodore B Trafalis, Petros Xanthopoulos, Panos M Pardalos, and Theodore B Trafalis. 2013. Linear discriminant analysis. *Robust data mining* (2013), 27–33.

[81] Xueshuo Xie, Zongming Jin, Jiming Wang, Lei Yang, Ye Lu, and Tao Li. 2020. Confidence guided anomaly detection model for anti-concept drift in dynamic logs. *Journal of Network and Computer Applications* (2020).

[82] Guixin Ye, Zhanyong Tang, Huanting Wang, Dingyi Fang, Jianbin Fang, Songfang Huang, and Zheng Wang. 2020. Deep Program Structure Modeling Through Multi-Relational Graph-Based Learning. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques* (Virtual Event, GA, USA) *(PACT '20)*. Association for Computing Machinery, New York, NY, USA, 111–123. https://doi.org/10.1145/3410463.3414670

[83] Shuangjiao Zhai, Zhanyong Tang, Petteri Nurmi, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2021. RISE: Robust Wireless Sensing Using Probabilistic and Statistical Assessments. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (New Orleans, Louisiana) *(MobiCom '21)*. Association for Computing Machinery, New York, NY, USA, 309–322. https://doi.org/10.1145/3447993.3483253

[84] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.

[85] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*.

Received 2024-09-12; accepted 2024-11-04