Noname manuscript No.
(will be inserted by the editor)

Post Deployment Recycling of Machine Learning Models

Don't Throw Away Your Old Models!

Harsh Patel · Bram Adams · Ahmed E. Hassan

Received: date / Accepted: date

Abstract Once a Machine Learning (ML) model is deployed, the same model is typically retrained from scratch, either on a scheduled interval or as soon as model drift is detected, to make sure the model reflects current data distributions and performance experiments. As such, once a new model is available, the old model typically is discarded. This paper challenges the notion of older models being useless by showing that old models still have substantial value compared to newly trained models, and by proposing novel post-deployment model recycling techniques that help make informed decisions on which old models to reuse and when to reuse. In an empirical study on eight long-lived Apache projects comprising a total of 84,343 commits, we analyze the performance of five model recycling strategies on three different types of Just-In-Time defect prediction models (Random Forest (RF), Logistic Regression (LR) and Neural Network (NN)). Comparison against traditional model retraining from scratch (RFS) shows that our approach significantly outperforms RFS in terms of recall, g-mean, AUC and F1 by up to a median of 30%, 20%, 11% and 10%, respectively, with the best recycling strategy (Model Stacking) outperforming the baseline in over 50% of the projects. Our recycling strategies provide this performance improvement at the cost of a median of 2x to 6-17x slower time-to-inference compared to RFS, depending on the selected strategy and variant.

Harsh Patel
Queen's University, Canada
E-mail: 21hsp2@queensu.ca

Bram Adams
Queen's University, Canada
E-mail: bram.adams@queensu.ca

Ahmed E. Hassan
Queen's University, Canada
E-mail: ahmed@cs.queensu.ca

Keywords Post Deployment · Ensemble Learning · Model Recycling · Software Defect Prediction

1 Introduction

Machine Learning (ML) teams spend most of their time on model selection, model training and model fine-tuning to select the models performing best on training, validation and test datasets (Schelter et al. 2015). Once the initial training of the machine learning models is completed, the focus shifts towards integrating the models into software products, deploying said models in production, then monitoring the performance of the machine learning models in production, identifying issues affecting model performance.

One of the most common ML issues encountered in production is data drift, also known as dataset shift (Quinonero-Candela et al. 2008), which occurs when an ML model's inference data distribution changes compared to the training data distribution, leading to the model's performance degradation. Such changes can occur discretely, for example after some outside event that affects the inference data distribution, or continuously, with data gradually changing over time. When undetected, this phenomenon can have major adverse effects on model performance, as surveyed in multiple case studies by Paleyes et al. (2022). That said, drift is not necessarily a monotonous phenomenon, as it can manifest as sudden drift, gradual drift or even recurrent drift each of which requires specific monitoring and adaptation strategies to address effectively.

There are multiple techniques for adapting models to new data, including regular retraining from scratch and continual learning (Diethe et al. 2019), but both directions have important drawbacks. Retraining models from scratch (RFS) on recent data is costly, both financially, due to the cost of hardware, electricity, or cloud compute time to train, develop, and maintain, and environmentally, due to the carbon footprint required to fuel modern tensor processing hardware (Strubell et al. 2019). With RFS, knowledge or insights gained from previous model versions are not transferred to the new model; therefore, there is no continuity between models. Continual learning, also referred to as lifelong learning (Chen and Liu 2018), is a sequential learning process, requiring only a small portion of input data at once to evolve or fine-tune a model. However, it is prone to catastrophic forgetting: performance on previously learned data could significantly degrade over time as new data points are added (De Lange et al. 2021). Therefore, we focus on improving RFS model retraining in our study.

We claim that, even with data drift, old models have value, and should be reused ("*recycled*") to improve future model performance. Based on this insight, we propose to use post-deployment recycling strategies that exploit past knowledge of model performance along with knowledge of data distribution at inference time to recommend models from the past for future inference when data distribution similarities are detected. The recycled models are used in a black-box manner, i.e., without fine-tuning or training, for inference on new data. Such reuse is still a challenging task as it now requires predicting the best model for a given inference data point, which can introduce additional noise. However, it can be beneficial since it can improve the overall prediction performance, and, because it treats the model as a black box, it can be adopted easily in existing ML pipelines.

We empirically evaluate our approaches in the context of Just-In-Time (JIT) defect prediction models (Kamei et al. 2012), a popular type of ML model in the domain of software engineering, using data of 8 large, long-lived open-source projects from the ApacheJIT dataset (Keshavarz and Nagappan 2022), we address the following research questions:

– **RQ1**: Are old JIT prediction models worth saving?
– **RQ2**: What variant works best for each recycling strategy?
– **RQ3**: Which recycling strategy performs the best overall?

Our main contributions are:

– We empirically evaluate and identify the value that historical JIT models can provide to future inference on ActiveMQ, Camel, Cassandra, Flink, Groovy, HBase, Hive and Ignite projects.
– We propose five different strategies for recycling software analytics models.
– We empirically evaluate the performance of these strategies and 29 variants compared to RFS for three different types of JIT models.
– We evaluate the trade-offs between each recycling strategy in terms of time-to-inference and metric performance.

2 Related Work

We discuss the differences between our results and the closest related work on black-box reuse of models in detail in Section 7. This section discusses the related work concerning JIT defect prediction and drift in software defect prediction models.

2.1 JIT Defect Prediction

Defect prediction models have gained substantial attention in research (Zhao et al. 2023). In particular, Just-In-Time (JIT) defect prediction approaches have been proposed to help developers prioritize their limited software quality assurance (SQA) resources on the riskiest commits (Kamei et al. 2012; Kim et al. 2008), since JIT defect prediction aims to predict the defect likelihood of each software change. These predictions can be expressed as concrete work assignments for a developer to fix a defect due to a change, reducing the time for finding the developer who introduced the defect because the predictions are made early on, this ensures that the details of the code change are still fresh in the developers' minds.

The four most commonly used families of code change properties used in JIT models (Mockus and Weiss 2000; Kamei et al. 2012; Kim et al. 2008; Kononenko et al. 2015) typically measure (a) the magnitude of the change (Size, dimension in Table 1); (b) the dispersion of the changes across modules (Diffusion); (c) the defect proneness of prior changes to the modified modules (History) and (d) the experience of the author (Author Experience).

2.2 Concept Drift in Software Analytics

Similar to other fields, software analytics models are also affected by concept drift due to evolving software project data and require retraining to maintain performance. Ekanayake et al. (2009) study the effect of concept drift in software projects by evaluating the prediction quality of file-level software defect prediction models over sequential releases of software projects over time. They find that the quality of the prediction follows periods of stability and drift in software projects, suggesting that concept drift is an important factor to consider when investigating defect prediction. Stability refers to a period when the defect prediction quality of a model is continuously above a certain threshold ($AUC > 0.8$). This stability period was observed to be as long as ~ 15 months for one of the projects (Eclipse). However, the authors also found that the overall quality of the prediction drops significantly ($AUC \approx 0.5$) during periods of drift.

Our study is different in that we are evaluating JIT defect prediction models detecting bug-inducing commits in real-time, whereas Ekanayake et al. (2009) study the effect of concept drift over sequential releases of software projects using file-level features such as # revisions and # bugs fixed of critical type. While they study the effects of drift by looking at prediction performance over time using AUC, we evaluate drift by studying the feature survival rate of JIT models over time. Additionally, we evaluate each historical model over time based on its commit-level error performance compared to other historical models to identify the value of each model over time.

A study by McIntosh and Kamei (2018) suggests that fix-inducing changes are a moving target, and that JIT models lose a large proportion of their discriminatory power and calibration scores one year after being trained due to value shifts in the metrics used to train them. Their study evaluates JIT models over time and evaluates the importance scores of code change properties over time. We compare models by their commit-level error performance to quantify the model's value and correlate it with the feature survival rate of the model to understand impact of drift. Olewicki et al. (2022) study when to retrain a model for detecting brown builds in continuous integration (CI) using a priori heuristics such as comparison to a prior week's performance or to the cumulative product of performance of recent models of past few weeks, which seem effective ways to reduce retraining costs. Both of these studies are related to our work, as they study the retraining problem in software analytics models. However, as both keep retraining new models, the older models are discarded, losing potentially valuable historical knowledge that the model has learned at a specific time. Our study focuses on how to keep reusing historical models, i.e., model recycling.

There have been multiple studies on online JIT defect prediction (Cabral and Minku 2022; Tabassum et al. 2022; Tan et al. 2015). Although online learning can be an effective solution to retraining from scratch in practice because of the low cost of retraining, it however is prone to catastrophic forgetting (De Lange et al. 2021). For instance, Gao et al. (2023) found that fine-tuning CodeBERT on new datasets leads to substantial performance degradation on the initial dataset by 28.9% and 84.6% for code summarization and vulnerability detection, respectively. Olewicki et al. (2023) found that taking preventive measures for catastrophic forgetting of previously acquired knowledge, improved F1-score significantly from 4-26% for 3 out of 8 projects studied. This is particularly problematic when dealing with sequential data, as the model may prioritize recent data and forget valuable information from the past. Online JIT software defect prediction studies focus on improving the stability of predictions *over time*, while our study focuses on utilizing historical models to improve the quality of predictions *at inference time*. Therefore, our study could be complementary to these studies as it can be used along with online learning to improve the quality of predictions over time.

3 RQ1: Are old models worth saving?

This section outlines the design and results of our initial empirical study trying to understand the value of old machine learning models in the context of JIT defect prediction.

3.1 Motivation

As ML models age, their prediction accuracy is expected to decrease due to data drift (Ekanayake et al. 2009; Olewicki et al. 2022; McIntosh and Kamei 2018; Cabral and Minku 2022). The general assumption is that it is necessary to regularly retrain the machine learning model with new data to make sure the model's predictions remain relevant to newer production data. In this race to avoid the negative effects of drift, ML teams rely on this assumption and simply discard old models without any second thought. This encourages us to find out whether old machine learning models (MLMs) are really of no value in practice, or whether there is any value in preserving old models.

Table 1: Commit Metrics

Dimension	**Metric**	**Description**
Size	la	total number of lines added in commit
	ld	total number of lines deleted in commit
Diffusion	nf	number of files modified in commit
	nd	number of directories modified in commit
	ns	number of subsystems modified in commit
	entropy	distribution of change over files in commit
History	ndev	number of unique developers who previously changed the modified files of a commit
	age	average time since the previous change of the modified files of a commit
	nuc	number of unique changes to the modified files of a commit
Author Experience	aexp	number of prior commits by the commit author
	arexp	same but in the last # months
	asexp	number of prior commits to the subsystem of this commit by the commit author

3.2 Study Design

3.2.1 Studied Systems

3.2.1.1 Dataset

Several datasets are available for JIT defect prediction (Kamei et al. 2012; Jiang et al. 2013; McIntosh and Kamei 2018; Keshavarz and Nagappan 2022). Keshavarz and Nagappan (2022) built a large JIT bug prediction dataset called ApacheJIT comprising the commits of 14 popular Apache projects. ApacheJIT has 106, 674 commits, 28, 239 of which are labelled bug-inducing. Commits in ApacheJIT have the same features as Kamei et al. (2012), listed in Table 1.

In this study, we focused on utilizing commits from 8 specific projects within ApacheJIT as shown in table 2, excluding the remaining six projects for the following reasons:

– **Hadoop HDFS and MapReduce**: ApacheJIT contains commits from both the Hadoop HDFS and MapReduce projects. However, many commits from these projects are not segregated, making it challenging to accurately identify the project of origin for each commit and perform a study on these projects independently. It is plausible that this issue arose due to the

Table 2: ApacheJIT: Project Defect Distribution

Project	**# Total Commits**	**# Buggy Commits (%)**	**Time Period (Years)**
ActiveMQ	6,126	1,404 (∼22%)	2005-2019
Camel	22,700	3,078 (∼13%)	2007-2019
Cassandra	8,159	3,117 (∼38%)	2009-2019
Flink	11,691	2,811 (∼24%)	2010-2019
Groovy	8,059	1,614 (∼20%)	2003-2019
Hbase	8,730	3,782 (∼43%)	2007-2019
Hive	6,842	4,223 (∼61%)	2008-2019
Ignite	12,036	2,439 (∼21%)	2014-2019

projects' repositories being independent initially, but later consolidated into a shared repository named Hadoop.

– **Kafka, Spark, Zeppelin, and Zookeeper**: These projects have a notably low number of commits, of approximately 1,000 or even fewer. In our study, this limited number of commits translates to an insufficient number of time windows for each project for training and evaluating JIT models.

3.2.1.2 Machine Learning Models

Over the years, many machine learning algorithms have been used to predict buggy commits. However, Zeng et al. (2021) found that "*simple*" JIT defect prediction approaches outperform more complex approaches like CC2Vec and DeepJIT. Hence, we decided to use the JITLine model of Pornprasit and Tantithamthavorn (2021), which is one of the best-performing JIT defect prediction models and uses Random Forest as a classifier. To evaluate generalizability of our findings, we also evaluate our recycling strategies using a simpler model such as the Logistic Regression model inspired from Kamei et al. (2012) and a more complex Deep Neural Network model from Hoang et al. (2019) called DeepJIT. The details about the features used in each model are discussed later in Section 3.2.5.

Furthermore, since our study focuses on evaluating historical models, it is worth noting that the field of software engineering provides easy access to such models. This accessibility is facilitated by the availability of open-source software projects, which allow for the mining of chronological changes made to the software.

3.2.2 Data Preparation

This study aims to evaluate old JIT machine learning models' performance over time. Since each ApacheJIT project has a finite number of data samples, we use a sliding window technique to create more data splits. Figure 1 illustrates the sliding window sampling technique.

We first sort the commits by author date, then using a sliding window of one thousand commits that shifts by increments of two hundred, we create more windows for a dataset. The data windows mimic the real-time data stream encountered by machine learning models in production. For every window created, we establish both a training and validation set. The training set comprises 80% of the data, randomly sampled from a window, while the validation set holds the remaining 20% of the data scattered across the window.

JIT defect prediction faces verification latency (Cabral et al. 2019), which is particularly important in online learning. This means that in reality a new data point (commit) can only be used for future (online) training once its actual label is known, which could take a variable amount of time. I.e., retrospectively assuming those data points' labels to be known instantly after a prediction is not realistic. Hence, to make empirical evaluation more realistic, it is custom to choose a latency threshold to delay the use of new data points for retraining. Song et al. (2023) found that reducing this latency threshold (from 90 to 15 days) led to an increase of up to 45.86% in mislabeling defect-inducing examples. To address this threat to validity, we keep a gap of 200 samples (median of 57 days) between our training and testing sets, denoted as "Delayed Labelling (DL)" (also known as verification latency) in Figure 1. Subsequently, we use the two hundred samples after the DL gap as the test set, resulting in a test set of the same length as the validation and DL sets.

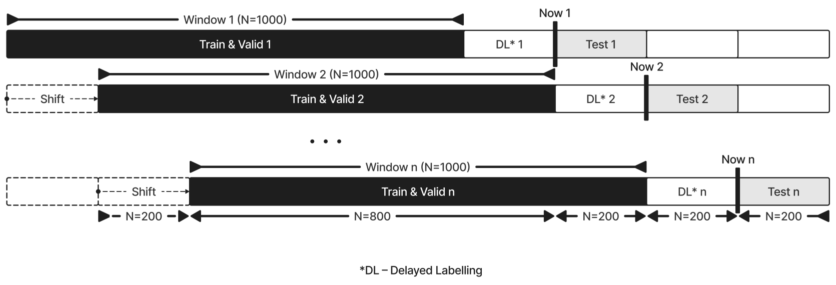


Fig. 1: Sliding Window Sampling

Based on the sliding window sampling technique, we create a total of 386 data splits across eight projects, as shown in Table 3. The minimum number of commits for a project is 6,126 (ActiveMQ), which is far greater than the number of commits required to create a window (i.e., 1,000) and results in a minimum of 26 data splits according to the following formula:

$$\#\text{ Data Splits} = \lfloor \frac{\#\text{ Commits} - \text{Window Size}}{\text{Window Shift}} \rfloor + 1$$

Table 3: Data Splits resulting from Sliding Window Sampling

Project	**# Total Commits**	**# Data Splits**	**# Evaluated Windows**
ActiveMQ	6,126	26	9
Camel	22,700	109	92
Cassandra	8,159	36	19
Flink	11,691	54	37
Groovy	8,059	36	19
Hbase	8,730	39	22
Hive	6,842	30	13
Ignite	12,036	56	39

3.2.3 Model Version History

Using the above sliding window sampling, each resulting time window serves as the training data for a new RFS model. Ekanayake et al. (2009) used a constant training period length of 2 months and evaluated defect predictions over each month while our training period is of a median of 9 months over different projects. We also maintain a history of 15 model versions consisting of 8-123 months of data depending on the project and with a median of 33 months for each window evaluated. This decision strikes a balance between excessive reliance on distant model history and retaining sufficient context.

On top of 15 windows for 15 historical models, the current window is required to train the RFS model as our baseline, the next window is skipped to address verification latency, and the window after that is used for evaluation. Therefore, the total # evaluated windows for each project is lower than the total # windows (data splits) created for the project as listed in Table 3. As the sliding window advances, the model history moves forward. When the model history size surpasses the predefined limit of 15 windows, we discard the oldest version, ensuring we continuously have updated models while maintaining a manageable history.

In section 5, in addition to the variants with limited model history, we also evaluate variants of recycling strategies without limits on the model history size for each window. This allows us to consider a broader historical perspective.

3.2.4 Handling Class Imbalance

Since software projects typically have very few bugs relative to the total number of commits, JIT datasets typically are imbalanced. Several sampling strategies are available to address imbalanced class problems, like oversampling and undersampling. For the JITLine and Logistic Regression models, we chose the oversampling approach Synthetic Minority Over-sampling Technique (SMOTE) as used by Pornprasit and Tantithamthavorn (2021). The SMOTE method of over-sampling the minority class involves creating synthetic minority class examples, and it can achieve better classifier performance than only undersampling the majority class (Chawla et al. 2002). For DeepJIT, each window of data is resampled with replacement into smaller batches having equal distribution of both positive and negative labels to address the class imbalance. In our sliding window sampling shown in figure 1, we apply SMOTE or resampling to each training set only, while the validation and test sets remain unchanged.

3.2.5 Feature Selection and Engineering

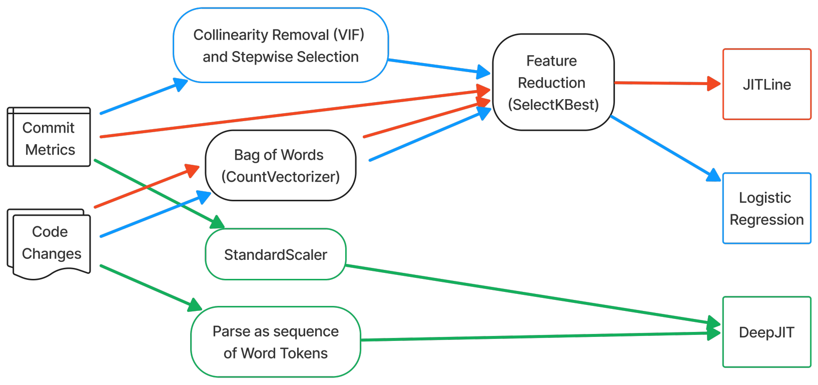

Collinearity Removal (VIF) and Stepwise Selection
Feature Reduction (SelectKBest)
JITLine
Commit Metrics
Bag of Words (CountVectorizer)
Logistic Regression
Code Changes
StandardScaler
Parse as sequence of Word Tokens
DeepJIT

Fig. 2: Feature Selection and Engineering

The pre-processing pipeline of features for each model is shown in Figure 2. We use the same features as Pornprasit and Tantithamthavorn (2021) for all models. However, there are some differences in the way these features are processed for each model, as discussed below.

JITLine – JITLine uses both commit metrics listed in Table 1 and code changes (i.e., added and removed lines of code in a commit). The code changes are converted into count vectors using the Bag of Words (BoW) technique. BoW is a common technique for extracting features from text documents that involves counting the number of times a word appears in a document, then transforming the text into a vector of word counts. Since code changes can include thousands of code tokens, including commit metrics, this leads to a very large feature space, considering our window size of 1,000 data points, and that a general rule-of-thumb calls for at least 10-15 samples per feature to avoid overfitting [1].

[1] https://postindustria.com/how-much-data-is-required-for-machine-learning

To reduce the feature space, we use scikit-learn's *SelectKBest* feature selection method (Pedregosa et al. 2011) to select the top 100 features (tokens) based on the chi-squared test. The chi-squared test is often employed for feature selection in text classification tasks because it helps identify the most relevant features by measuring the independence between each feature (term or word) and the class labels. It focuses on selecting features that have significant associations with the classes, allowing for the prioritization of terms that contribute the most discriminative information for classification.

Logistic Regression – We use the same features as JITLine, but to remove collinearity between commit metrics features, we use the Variance Inflation Factor (VIF) score to remove features having a score above 10 (James et al. 2013). Additionally, we use stepwise selection to remove features that do not contribute to the model's performance. Finally, we use the same feature selection method as JITLine to select the top 100 features.

DeepJIT – The original DeepJIT model does not use commit metrics from Table 1 as features. Instead, it uses the commit messages and code changes. However, to keep consistency across all models, we use the same features as JITLine and Logistic Regression models, i.e., both commit metrics and code changes. To improve time to convergence, we use scikit-learn's *StandardScaler* to scale the features to a mean of 0 and a standard deviation of 1 (Pedregosa et al. 2011). DeepJIT does not require feature reduction since it automatically understands the semantic features of each deleted or added line in each changed file. Then, it aggregates these features to generate a new representation (embedding) of the changed file, which is used to construct the features of the code changes (Hoang et al. 2019).

3.2.6 Evaluating Model Performance

We use standard performance metrics such as F1 score, AUC, Precision and Recall to evaluate each model's performance. Apart from these metrics, we also evaluate models using Specificity, Geometric Mean and Average Precision.

For binary classification, G-mean is the square root of the product of the minority and majority class recall values ($\sqrt{recall \times specificity}$). The minority recall is the ratio $tp/(tp+fn)$ where tp is the number of true positives and fn is the number of false negatives, quantifying the ability to avoid false negatives. Specificity is the ratio $tn/(tn + fp)$ where tn is the number of true negatives and fp is the number of false positives. It quantifies the ability to avoid false positives (Lemaître et al. 2017).

For readability purposes, we only discuss g-mean, AUC, F1 score, precision and recall in our summary plots and tables in the paper, while the replication package also includes the other performance metrics, i.e. Specificity and Average Precision (Patel 2023). Since the g-mean is a geometric mean of performance measured separately on each class, it is a better metric for the performance of imbalanced classification tasks (Kubat et al. 1997).

3.2.7 Analysis of Old ML Models

To understand how trained ML models change their performance over time, we evaluate each ML model version on test sets of future time windows. Based on their error on each data sample $Y_{error} = abs(Y_{prob} - Y_{actual})$ we find the best-performing old model for each future commit. We then lift up this information to the level of time windows by determining the model versions performing best on the largest number of commits in that window. This metric can be defined as the Temporal Commit Dominance Factor ($TCDF$), which is calculated as follows:

$$TCDF(m_i, w_j) = \frac{\#\text{ Commits in } w_j \text{ for which } m_i \text{ achieves the lowest } Y_{error}}{\#\text{ Commits in } w_j}$$

Where m_i is the model version, w_j is the window and $i \leq j$.

$TCDF$ quantifies the model's dominance and effectiveness over a range of commits within a given time window. For each model, we subsequently calculate the percentage of future windows in which that version was the best in terms of $TCDF$ in order to quantify its (future) value. This is defined as the Future Window Dominance Percentage ($FWDP$) and is calculated as follows:

$$FWDP(m_i) = \frac{\#\text{ Future Windows for which } m_i \text{ achieves the highest } TCDF}{\text{Total }\#\text{ Future Windows}}$$

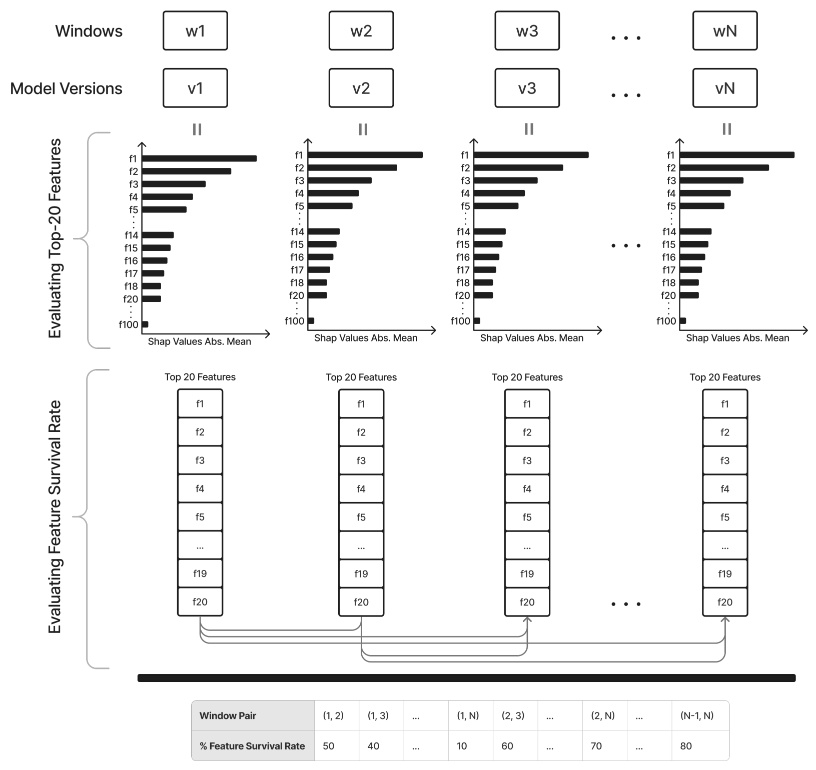
We do not use statistical tests to identify winning model versions, since even if a winning model only marginally wins on more commits, the RFS model still is not that appealing.

To understand the performance of older models in the presence of drift, we also evaluate the Feature Survival Rate (FSR) of models over time (Olewicki et al. 2022). As explained in Figure 3, we do this by first calculating, for each model of each time window, the top 20 features based on their feature importance. Then, for each pair of time windows (i, j; where, $j > i$), we find the percentage of the top 20 features from model i that survived in model j. FSR can be calculated as follows:

$$FSR(m_i, m_j) = \frac{\#\text{ Selected Top Features from } m_i \text{ that survived in } m_j}{\#\text{ Selected Top Features in } m_i (= 20)} * 100$$

Where m_i and m_j are the model versions and $j > i$.

To measure feature importance, we use the SHapley Additive exPlanation (SHAP) values proposed by Lundberg and Lee (2017), which is a Classifier Agnostic Feature Importance method. It is one of the more recent global feature importance methods that gives consistent results not impacted by feature



Window Pair	(1, 2)	(1, 3)	...	(1, N)	(2, 3)	...	(2, N)	...	(N-1, N)
% Feature Survival Rate	50	40	...	10	60	...	70	...	80

Fig. 3: Evaluating FSR of Models over Time Windows

interactions (Rajbahadur et al. 2021). We rank features based on the mean SHAP value first, then its standard deviation (from high to low) across all test samples in a time window, and select the top 20 features. The high mean is an indicator of the overall high importance of a feature across all test samples, while the high standard deviation is an indicator of the changing importance of a feature across test samples meaning that the feature may have a positive (predicting commit as buggy) or negative impact on prediction depending on the test sample.

To understand whether drift plays any role in the observation of a model being the best for a given future time window or not, we want to identify whether there is a statistically significant difference between feature survival rates of a model in the windows where the model is winning vs. not winning. For this, we perform in each studied project a statistical analysis of all models that won in at least one window. We observed that one of these two groups (typically the "*winning*" group, sometimes "*not-winning*") sometimes can have

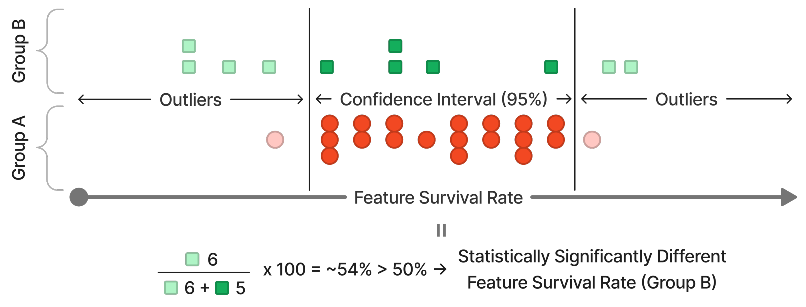

Group B
Group A
Outliers
Confidence Interval (95%)
Outliers
Feature Survival Rate
6
6 + 5
x 100 = ~54% > 50%
Statistically Significantly Different Feature Survival Rate (Group B)

Fig. 4: Identifying Statistical Significance of Feature Survival Rate Differences with the Help of Confidence Intervals (With Group A: Majority Group used for calculating Confidence Interval, Group B: Minority Group compared with Group A)

fewer than six data samples, which makes it difficult to use statistical tests like Wilcoxon signed-rank test. Therefore, as shown in Figure 4, we calculate the confidence interval ($\alpha = 0.05$) of the feature survival rate for the majority group (i.e., either the "*winning*" or "*not-winning*" group) and identify the data samples of the minority group falling outside the confidence interval of the majority group. Since our original goal is to understand the difference in statistical significance, we consider the difference in feature survival rate statistically significant if 50% or more minority data points fall outside the majority confidence interval.

Models from the last few windows have fewer chances to win due to the lack of future test windows. Hence, to ensure sufficient data points for calculating confidence intervals, we exclude model versions that are within ten windows of the end window.

Sometimes a model may win in an immediate next window but may not win far in the future. On the contrary, a model may not win immediately but may win far in the future. So, to understand the potential of model reuse, we investigate the period of potential reuse by identifying the last and first windows a model is winning in. This gives us an idea of how far in the future a model can be reused.

3.3 Results

Old models have value. Based on TCDF, we found that six projects (ActiveMQ, Camel, Cassandra, Groovy, HBase and Ignite) have an old model dominating in $\geq 52\%$ of their time windows. The other project (Flink) has an old model dominating in $\geq 32\%$ of their time

windows. Only one project (Hive) has more uniformly distributed windows where it can be difficult to predict the winning model. Figure 5 evaluates, for a given time window i of project Camel (X-axis), all past model versions, to calculate their classification error (Y_{error}) on that window's test commits (Y-axis). Each cell in the figure shows the value of $TCDF$, i.e., the ratio of commits in which that cell row (model version) outperformed all other, model versions available for that window (vertical column) in terms of absolute error. For a given time window (vertical column), the higher the number of commits for which a model is the best, the darker the colour of the cell.

The figure shows long, horizontal red lines for a minority of model versions, suggesting that these model versions continuously performed better for an extended time period. For example, model $v1$ has the longest and darkest horizontal red line expanding approximately until window 86 (covering $\sim 74\%$ of time windows, comprising commits from ~ 101 months). This effect is shifting to other model versions over time, e.g., model $v60$ and later model $v90$, following a similar trend of a continuous horizontal red line. In contrast, we note that the diagonal is not dark, indicating that RFS models do not automatically outperform older models, even though they have been trained using the most recent data for that inference time window. Figures for other projects show similar patterns and have been included in the replication package (Patel 2023).

Figure 6 shows, for each project, the $FWDP$ for the model versions (with $FWDP > 0$). As per the figure, model $v1$ for project Camel was considered the best model version for $\sim 74\%$ of the time windows, which confirms our finding above, with model $v90$ in second position winning in $\sim 8\%$ of the windows. The figure shows that old models dominated at least $\geq 52\%$ of the time windows for all projects except Flink and Hive. Although Flink did not have a single dominating old model, it had three dominating old models that constituted $\sim 78\%$ of total windows. For Hive, it was found that models $v23$ and models from $v25$ to 28 were RFS models winning their targeted window while $v7$ was the only model able to win time windows up to 3 times consecutively, whereas the remaining versions only won 1 or 2 consecutive windows. This could be because of the high ratio of buggy commits ($\sim 60\%$) present in Hive compared to other projects ($< 50\%$), which may require model adaptations that old models cannot accommodate.

As to why early models seem to dominate a substantial percentage of time windows, we have a number of hypotheses. First, the nature of software development evolves over time, especially during major release development periods marked by substantial feature and restructuring changes that affect larger segments of a codebase. Conversely, minor releases, following Lehman's fifth law and emphasizing the "conservation of familiarity", highlight that growth is constrained by the necessity to maintain familiarity (Herraiz et al. 2013). Many software projects follow an agile process with iterations of N weeks, where changes within each iteration are more cohesive than across iterations. Therefore, consecutive iterations often focus on more similar changes than

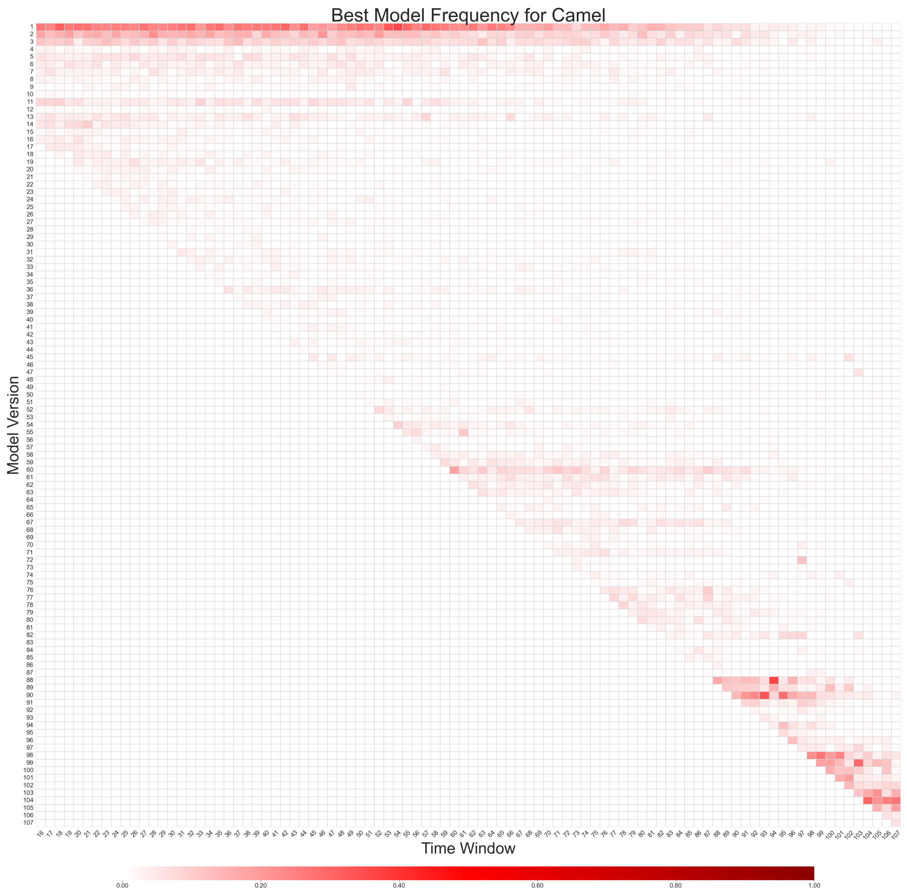


Fig. 5: Camel: Percentage of Data Samples with Best Performance (i.e., $TCDF * 100$) per Model Version over Time Windows

non-consecutive ones, allowing older models to remain relevant for extended periods.

For example, in Figure 5, noticeable triangle patterns align with major version releases of Apache Camel[2], such as the periods from window 16 to 87, 88 to 97, 98 to 103, and 104 to the end. Winning models like $v1$, trained on commits from 2007-2008, coincide with the release of Camel $v1.0.0$. Similarly, models $v88$, $v98$, and $v104$, trained on commits from 2017, 2018-2019, and 2019, respectively, align with the releases[3] of Camel $v2.20.0$, $v2.24.0$, and $v3.0.0$. This suggests that the changes in subsequent releases might not be substantial enough (conservation of familiarity) to warrant retraining the model, explaining the sustained performance of older models.

[2] https://github.com/apache/camel
[3] https://camel.apache.org/releases/#camel

Table 4: Number and Proportion of Best Models per Project

Project	# Total Models	# Winning Models	% Winning Models	% RFS Models Winning[1]
ActiveMQ	26	3	11.54	0.00
Camel	109	10	9.17	4.35
Cassandra	36	2	5.56	5.00
Flink	54	11	20.37	13.51
Groovy	36	3	8.33	5.00
Hbase	39	1	2.56	4.35
Hive	30	9	30.00	🟥 38.46
Ignite	56	4	7.14	2.56

🟥 Outlier

[1] Ratio of # Winning RFS Models over # Evaluated Windows (see Section 3.2.3)

Second, JIT (Just-in-Time) software defect prediction encounters concept drift, which involves changes in the ratio of examples for each predicted class (Cabral et al. 2019). Such changes reflect shifts in the software development process, including evolving priorities or refactoring efforts. For example, a development team might initially prioritize the graphical user interface (GUI) and later shift focus to implementing business logic, introducing a concept drift in the relationship between input features and labels. Additionally, alterations in software development practices, like refactoring, can influence the rate of defect-inducing changes, resulting in shifts in the class distribution.

As seen in Figure 5, the observable triangle patterns coincide with shifts in the label distribution. For example, windows 16-87 exhibit a median of approximately 157 buggy commits, windows 88-97 show a median of about 60 buggy commits, windows 98-103 display a median of around 25 buggy commits, and windows 104 to the end demonstrate a median of approximately 20 buggy commits. Building on the earlier reasoning, the older model leading the distribution change appears to outperform the more recent RFS model, which is trained on the latest data.

For most projects, the percentage of RFS models winning was meagre($< 14\%$), except for Hive ($\sim 38\%$), where using the RFS models can contribute the most to achieving the best performance. Table 4 shows the percentage of windows where the latest RFS model outperformed the older models, when evaluated on test commits from its respective test window. Hive's ratio of best models to total models of $\sim$ 30% could be because of the high ratio of buggy commits ($\sim$ 60%) present in Hive compared to other projects ($<$ 50%), which may require model adaptations that old models cannot accommodate, as discussed above.

The median FSR values of 17 out of 26 winning model versions in winning windows consistently range from 30% to 70%. Figure 7 illustrates the FSR for model versions on their winning time windows compared to their non-winning time windows for each project. Please note that because project Hbase had only one old model version ($v1$) winning all future win-

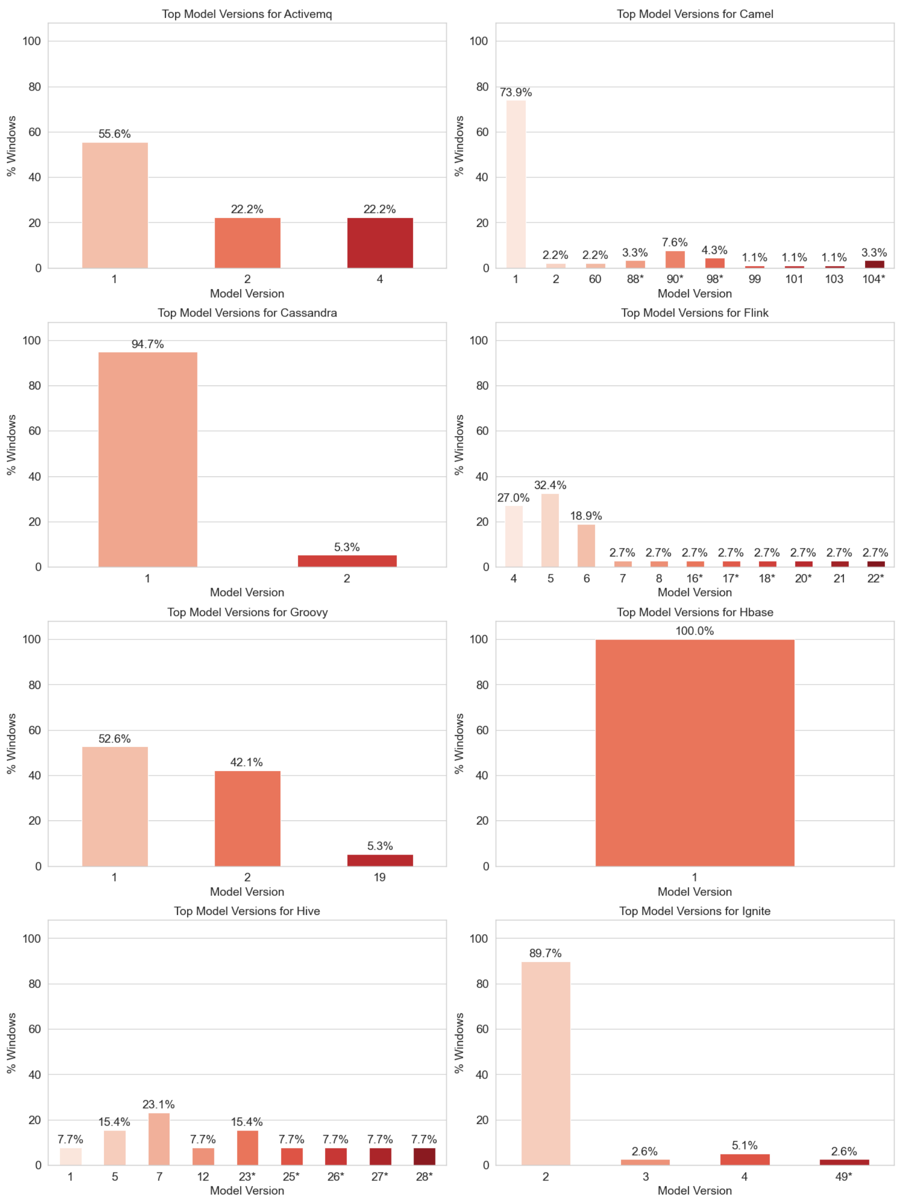


Fig. 6: Best Models in terms of $FWDP$ per Project. Starred models also won as RFS model for the window they were trained on.

dows, we do not see any data points in Figure 7 to compare with the winning windows. Overall, looking at the quartiles indicated in the figure, we can see that the FSR of models in winning time windows is higher than their FSR in non-winning time windows. At first sight, this finding aligns with the common wisdom that model retraining becomes necessary when high drift occurs. Table 5 indeed confirms that the FSR for the windows in which a model is winning is higher than for windows in which the model is not winning. This is confirmed for 17 out of 26 winning model versions by the fact that for $> 50\%$ of minority (mostly from winning windows) data points the FSR falls outside the confidence interval (OCI) of the majority group (mostly from non-winning windows).

However, the Median Feature Survival Rate (MFSR) in winning windows is still relatively low with only 7 out of 26 model versions having an MFSR of at least 50%, indicating that older models still outperform the RFS models in the presence of non-trivial concept drift. The reason behind this unexpected observation could be that the non-surviving features of these older models do not contribute significantly to the prediction and the rules learned by the model from an older time window are more fundamental and can be applied to a large proportion of new commits, indicating repeating patterns in the data.

Table 5 shows that the # of time windows of a model from its first winning window to its last winning window, ranges from 3 to 70 for more than 60% of the winning model versions. The longest winning (reuse) period was for model $v1$ for project camel, with the last winning window of that model being as far as 70 windows, despite the MFSR across the 70 winning windows being relatively low (35%). A similar pattern of long-range winning and low FSR can be seen for other projects too. Additionally, the # of winning windows for a model is more than 1 for $\sim 65\%$ of the winning model versions. The longest reuse period is indicated in bold in Table 5, and ranges from 3-71 windows, and with MFSR ranging from 0-50%. Where a model has more than 1 winning window (65% of winning models), the median reuse period is ~ 17 and the MFSR median is $\sim 35\%$. Despite the low MFSR of winning model versions, these models continue to perform well and win in future windows as far as 70 windows away from their first winning window, indicating their potential to be recycled. This intriguing outcome challenges the common belief that models must always be retrained from scratch and discarded.

Summary for Research Question 1

Old models have the best performance in $\geq 52\%$ and $\geq 32\%$ of future time windows for six and one projects respectively, up to a median of 3-70 windows in the future. This suggests that saving and potentially recycling old models to make predictions in future windows could make more sense than RFS in the context of JIT.

4 Model Recycling Algorithms

Given that RQ1 showed that old models have value, we need to find the best approach to recycle them. *Recycling* is the process of reusing old models to im-

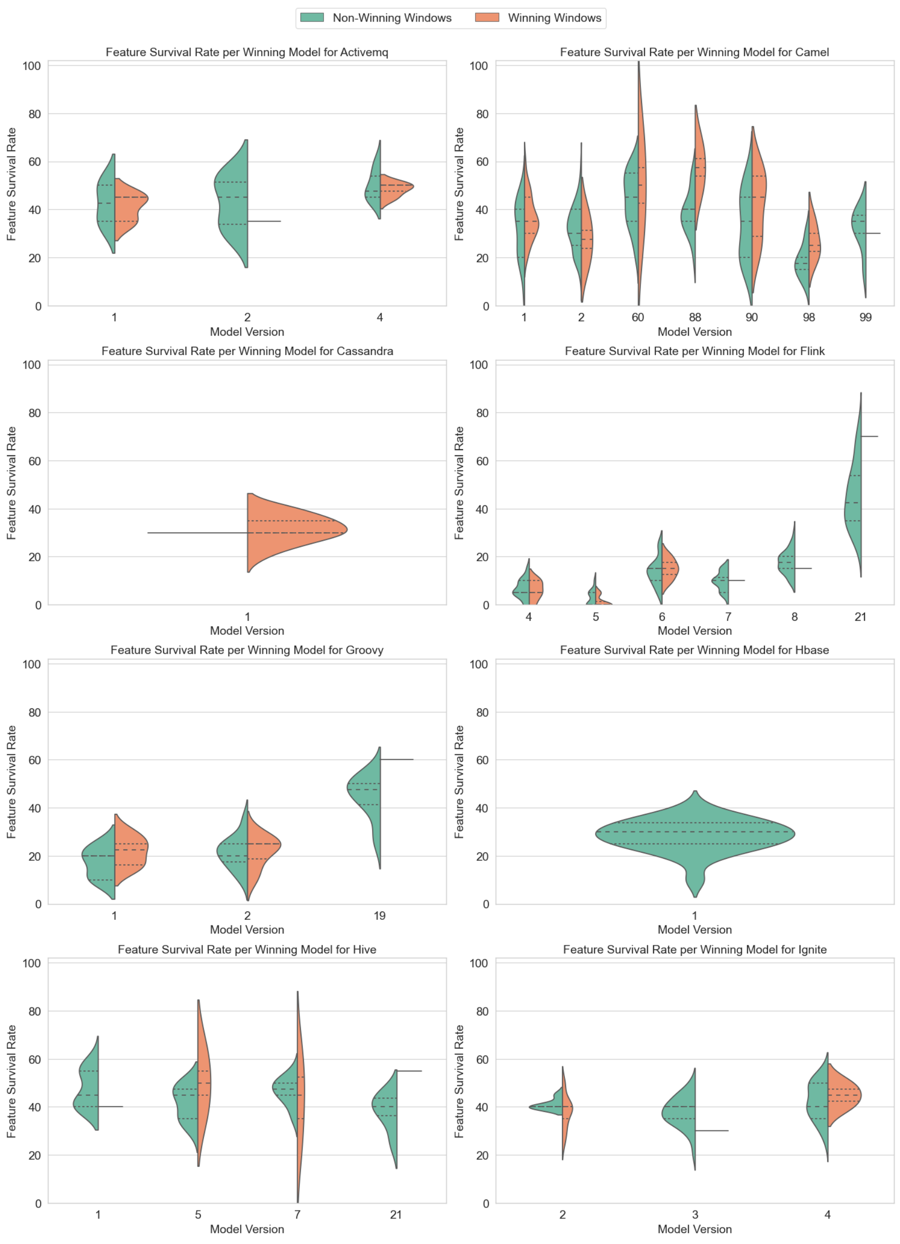


Fig. 7: FSR per Winning Model Version per Project

prove the prediction performance at a future inference time. Recycling not only emphasizes the preservation of valuable historical models, but also encourages

Table 5: Confidence interval results for FSR per reused model per project

Project	Model Version	# Windows		OCI[1]		WW[3]		MFSR[4]
		W[2]	NW[2]	#	%	# Windows	Win-dow	%
ActiveMQ	2	1	8	0	0.00	0	15	35.0
	4	3	6	0	0.00	**7**	16	50.0
Camel	1	68	24	21	87.50	**70**	15	35.0
	2	2	90	2	100.00	34	45	27.5
	60	2	45	2	100.00	14	72	50.0
	88	2	17	2	100.00	5	88	57.5
	90	6	11	3	50.00	6	90	45.0
	98	3	6	1	33.33	2	98	25.0
	99	1	7	0	0.00	0	102	30.0
Cassandra	1	18	1	0	0.00	**18**	15	30.0
Flink	4	10	27	6	60.00	**29**	20	5.0
	5	12	25	12	100.00	**29**	22	0.0
	6	7	30	4	57.14	28	18	15.0
	7	1	36	0	0.00	0	45	10.0
	8	1	36	1	100.00	0	32	15.0
	21	1	30	1	100.00	0	24	70.0
Groovy	1	10	9	4	44.44	**17**	15	22.5
	2	8	11	3	37.50	16	17	25.0
	19	1	14	1	100.00	0	19	60.0
Hive	1	1	12	1	100.00	0	16	40.0
	5	2	11	1	50.00	**3**	15	50.0
	7	3	10	2	66.67	**3**	17	45.0
	21	1	6	1	100.00	0	21	55.0
Ignite	2	34	5	1	20.00	**38**	15	40.0
	3	1	38	1	100.00	0	28	30.0
	4	3	36	1	33.33	20	29	45.0

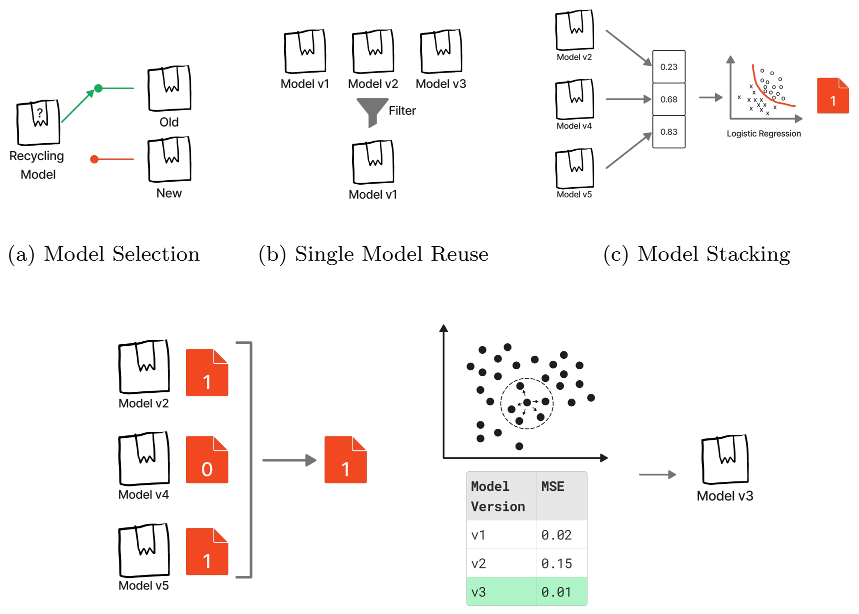
🟥 % OCI $\geq$ 50%

[1]OCI – % of minority windows where FSR lies Outside the Majority 95% Confidence Interval
[2]W – # Windows where model version is winning, NW - # Windows where model version is non-winning
[3]WW – Winning Window, i.e., the window where the given model version was winning
[4]MFSR – Median FSR on Windows in which a model version is winning

ML practitioners to derive continued benefits from these models' accumulated knowledge.

However, the process of devising effective recycling strategies is not without its challenges. The crux of the challenge lies in striking a balance between the utility of old models and the potential risks associated with their use. On the one hand, outdated models might fail to capture the nuances of evolving data patterns, leading to diminished prediction accuracy. On the other hand, deploying older models too conservatively might overlook opportunities where their insights could still prove valuable. To address this challenge, we propose a range of distinct strategies for recycling old models. The common goal behind each strategy is to make informed estimations about which old models to use and when to use them.

This section describes the general idea behind model recycling and the concrete recycling strategies we propose to implement this concept. An overview of these strategies is shown in Figure 8, and we now discuss each of these in detail.



Model Version	MSE
v1	0.02
v2	0.15
v3	0.01

(a) Model Selection (b) Single Model Reuse (c) Model Stacking

(d) Model Voting (e) Clustering

Fig. 8: Recycling Strategies

4.1 Model Selection

As shown in Figure 8a, the model selection strategy chooses at inference time, between the best old model thus far and the latest RFS MLM. To make this work, a model selection *recycling* model has to be trained pre-inference that predicts what version of the model (old or latest) to use for prediction at inference time. Algorithm 2 shows the process involved in this strategy, while

Algorithm 1 shows the process involved in finding the best old model from the model history (used by Algorithm 2).

pre-inference – As shown in Algorithm 2, for each window C, we first prepare (pre-inference) the training features (X_{train}) by combining the training data from the model version history including the current time window (C). The features are created from commit metrics and code changes as explained in Section 3.2.5. Second, before we can create the oracle (Y_{train}), we select the best old model from history, as shown in Algorithm 1. This old model can be chosen based on metrics like g-mean or F1-score of the historical models. Once the best old model is identified, to create the oracle (Y_{train}) we then determine the best model version between the best old model and RFS model for each commit in the training data based on their error ($abs(Y^i_{prob} - Y^i_{actual})$) on that commit. Finally, we train a *recycling* model able to select either the best old model or the RFS model using the prepared features (X_{train}) and oracle (Y_{train}).

during-inference – As new commits arrive, we use the *recycling* model to predict the best model version for each commit, then apply the predicted model version (i.e., old or new) to make predictions for the new commit.

Additionally, we also designed a variant of model selection where we do not train a *recycling* model, but during inference, we just average the probabilities of both models as $Y^{final}_{prob} = 1/2 * (Y^{v_i}_{prob} + Y^{v_j}_{prob})$. This variant requires inference from both old and new models.

Algorithm 1: Find Best Model from Model History with GMean for the Current Window (used by Algorithms 2 and 3)

```
   Input:
   $C$                                         /* Current Time Window */
1  $H_{all}$                                   /* Model Version History */
2  $Y_{pred}$                   /* Historical Models' Predicted Labels */
3  $Y_{actual}$                          /* Historical, Actual Labels */
   Output: $V$                /* Model Version with Best G-Mean Score */
4
5  Function findBestOldModel($C$, $H_{all}$, $Y_{pred}$, $Y_{actual}$):
6      $maxScore \leftarrow 0$, $V \leftarrow -1$
7
       /* Evaluate each model version one window at a time */
8      foreach $modelVersion \in H_{all}[C - 15 : C - 1]$ do
           /* Note: ground truth ($Y_{actual}$) is same for all models */
9          $score = gMean(Y_{pred}[C - 15 : C][modelVersion], Y_{actual}[C - 15 : C])$
10         if $score > maxScore$ then
11             $maxScore = score$
12             $V = modelVersion$
13     return $V$
14 End Function
```

As major drawbacks, the model selection strategy adds a layer of complexity to the inference process either by requiring training a new *recycling* model

Algorithm 2: Training a Model Selection *Recycling* Model for the Current Window

```
Input:
$C$                                        /* Current Time Window */
$K$                       /* Past Knowledge of Models' Performance */
$D$                                         /* Seen Training Data */
$H_{all}$                                /* Model Version History */
Output: $M$                              /* Model Selection Model */

Function modelSelection($C$, $K$, $D$, $H$):
    $Y_{pred} \leftarrow K[C-15:C]['predictedLabel']$
    $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
    $V \leftarrow$ findBestOldModel($C$, $H_{all}$, $Y_{pred}$, $Y_{actual}$)

    /* Prepare oracle $Y_{train}$ for model selection recycling        */
    foreach $i \in D[C-15:C]['commitID']$ do
        $Y_{train}[i] \leftarrow \arg\min(K[i]['error'][C], K[i]['error'][V])$
    /* Prepare features $X_{train}$ as per Section 3.2.5               */
    $X_{train} \leftarrow prepareFeatures(D[C-15:C])$

    /* Train RF to predict which model version to use for each commit */
    $M = RandomForestClassifier().fit(X_{train}, Y_{train})$
    return $M$
End Function
```

or by requiring inference from both old and new models to aggregate predicted probabilities. However, it can be an effective strategy if the *recycling* model can identify changes in data distribution (drift) and choose an appropriate model that performed better on similar data samples in the past.

4.2 Single Model Reuse

As depicted in Figure 8b, this strategy is designed to identify the best old model from a given model version history during pre-inference time and use the selected best old model during inference. Algorithm 3 shows the process involved in this strategy, and again leverages Algorithm 1 to find the best old model from the model history.

pre-inference – As shown in Algorithm 3, for each window C, the pre-inference step involves finding the best old model from the available model history. In addition, we also train a RFS model, to use it as a historical model in the next window, although this model is ignored for the current window C.

during-inference – At inference time, unlike the model selection strategy, the single model reuse strategy exclusively utilizes the old model to detect bug-inducing commit IDs as they arrive.

The single model reuse strategy is a simple approach that does not require any additional training or inference time. Yet, as this strategy only uses a historical model, it may not be able to always find the best historical model

Algorithm 3: Finding the Best Old Model Version for the Current Window using Single Model Reuse

```
   Input:
   $C$                                          /* Current Time Window */
1  $K$                          /* Past Knowledge of Models' Performance */
2  $D$                                         /* Seen Training Data */
3  $H_{all}$                                 /* Model Version History */
   Output: $V$                                     /* Best Old Model */
4  Function modelReuse($C$, $K$, $H_{all}$):
5      $Y_{pred} \leftarrow K[C-15:C]['predictedLabel']$
6      $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
7      $V \leftarrow$ findBestOldModel($C$, $H_{all}$, $Y_{pred}$, $Y_{actual}$)
8      return $V$
9  End Function
```

for the latest time window. For example, when the data distribution changes significantly, the historical model may not be able to make accurate predictions.

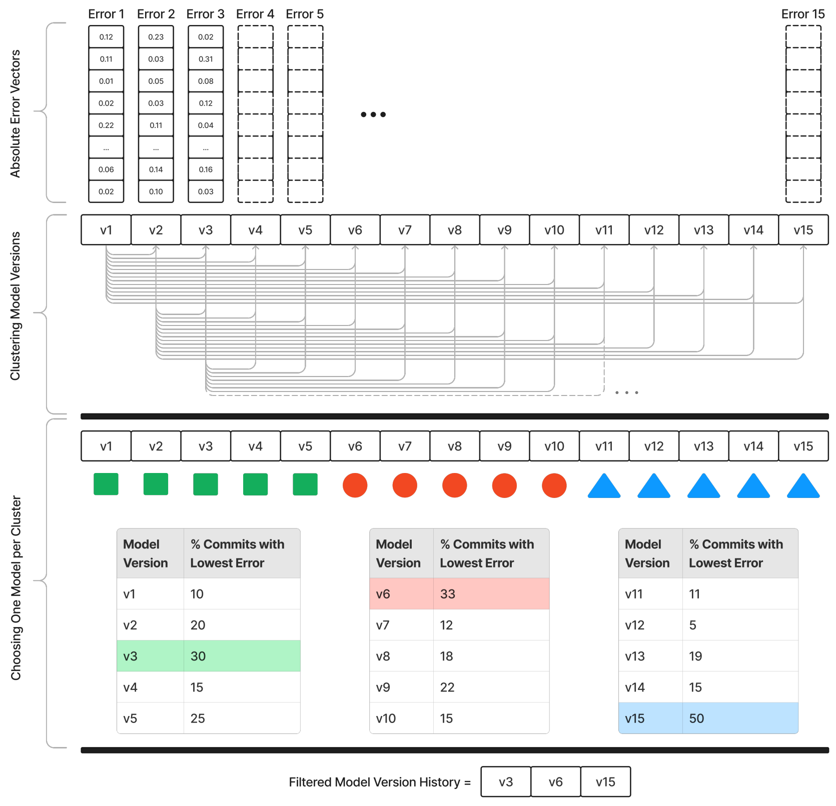
4.3 Model Stacking

As its name implies, model stacking utilizes stacked MLMs for inference. Those initially were proposed by Wolpert (1992), and found applications in various domains. The fundamental idea is to employ multiple models to generate predictions, then utilize a meta-model to combine the predictions from these base models, as illustrated in Figure 8c. In the case of JIT, the meta-model is trained to identify bug-inducing commits using the probabilities generated by the base models as features. The meta-model can be a simple linear regression or a more complex model such as a neural network.

The two main differences between traditional stacking and our stacking recycling approach are that traditional stacking re-trains both the stacked models and the meta-model simultaneously; however, we treat the stacked models as black boxes, thereby avoiding their re-training. Second, in practice, stacking combines models trained on the same data whereas we combine historical models trained on different data from different time windows up to and including the latest model.

The process can be split into two stages: *pre-inference* and *at-inference*. The pre-inference stage involves training the meta-model, and the at-inference stage involves using the meta-model to make predictions. The pre-inference stage can be further split into two stages: *filtering* of model version history and *training meta-model*.

pre-inference/filtering – Using the entire model history might not be ideal for this strategy, given that it employs all models for predictions. Hence, we employ statistical tests to identify the best models for stacking from history. As illustrated by Algorithm 4 and Figure 9, to find the most distinct set of



Model Version	% Commits with Lowest Error
v1	10
v2	20
v3	30
v4	15
v5	25

Model Version	% Commits with Lowest Error
v6	33
v7	12
v8	18
v9	22
v10	15

Model Version	% Commits with Lowest Error
v11	11
v12	5
v13	19
v14	15
v15	50

Fig. 9: Filtering Model History using Statistical Tests

models, we conduct a statistical test on the absolute error vectors ($Y_{error}(m_i)$ and $Y_{error}(m_j)$) of each pair of models. The error vectors are retrieved by finding the absolute error between actual and predicted class probabilities for both models m_i and m_j as defined earlier in Section 3.2.7, for each commit available in the model version history. The statistical tests on the error vectors help us to identify the most similar models in the model history, cluster those and select the best model from each cluster. We then use the selected models for stacking. We evaluate both the Wilcoxon signed-rank test (Woolson 2007) and the Scott-Knott Effect Size Difference (ESD) test (Tantithamthavorn et al. 2018) to reduce the size of the model history. We apply Bonferroni correction (Weisstein 2004) to adjust the p-value for multiple comparisons.

pre-inference/training meta-model – As explained in Algorithm 5, for each window C, we first prepare the training features (X_{train}) using the method explained in Section 3.2.5. Here, the oracle (Y_{actual}) includes labels for each

Algorithm 4: Filter Model Search Space using Statistical Tests for the Current Window (used by Algorithms 5 and 6)

```
   Input:
   $C$                                            /* Current Time Window */
1  $H_{all}$                                      /* Model Version History */
2  $Y_{prob}$                           /* Models' Predicted Probabilities */
3  $Y_{actual}$                                         /* Actual Labels */
   Output: $H_{filtered}$                  /* Filtered Model Version History */
4  Function filterModelHistory($C$, $H_{all}$, $Y_{prob}$, $Y_{actual}$):
5      $modelsToRemove \leftarrow []$
       /* Create all pairs of chronologically ordered model versions for
        comparison e.g. (1, 2), (1, 3), (2, 3) for models 1, 2, 3        */
6      $versionPairs \leftarrow combinations(H_{all})$
7
8      foreach $(modelX, modelY) \in versionPairs$ do
           /* If a model is already set to be filtered out, skip checking
            for statistical significance                                 */
9          if $modelX \in modelsToRemove$
10                 or
11             $modelY \in modelsToRemove$
12          then
13              continue
14
           /* Evaluate statistical similarity between the error vectors of
            the two models                                               */
15         $Y_{err}^{modelX} \leftarrow absolute(Y_{actual}[C-15:C] - Y_{prob}[C-15:C][modelX])$
16         $Y_{err}^{modelY} \leftarrow absolute(Y_{actual}[C-15:C] - Y_{prob}[C-15:C][modelY])$
17         $pValue = wilcoxon(Y_{err}^{modelX}, Y_{err}^{modelY})$
18
           /* If the models have statistically similar error vectors, remove
            one of the models in comparison, otherwise keep both since they
            both are distinct                                            */
19         if $pValue \geq ALPHA$ then
               /* Find the model version that more frequently has better
                absolute error than the other                            */
20             $modelVersion \leftarrow findLosingModelVersion(Y_{err}^{modelX}, Y_{err}^{modelY})$
21             $modelsToRemove.remove(modelVersion)$

22     $H_{filtered} \leftarrow H_{all}[C-15:C].remove(modelsToRemove)$
23     return $H_{filtered}$
24 End Function
```

commit indicating whether it is bug-inducing or not. Finally, we train the meta-model using the prepared features (X_{train}) and oracle (Y_{actual}).

at-inference – As new commits arrive, all stacked models predict probabilities for a commit being bug-inducing. We then use the meta-model to combine the probabilities from the stacked models to make predictions for the new commit.

Our historical stacking approaches leverage a simple logistic regression (LR) model or a more complex random forest (RF) model as our meta-models. We also evaluate two different feature space configurations for our meta-model.

Algorithm 5: Training Model Stacking *Recycling* Model for the Current Window

```
Input:
    $C$                                              /* Current Time Window */
    $K$                             /* Past Knowledge of Models' Performance */
    $D$                                               /* Seen Training Data */
    $H_{all}$                                       /* Model Version History */
Output: $M$                                         /* Meta Classifier Model */
Function modelStacking($C$, $K$, $D$, $H_{all}$):
    $Y_{prob} \leftarrow K[C-15:C]['predictedProbability']$
    $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
    $H_{filtered} \leftarrow$ filterModelHistory($C$, $H_{all}$, $Y_{prob}$, $Y_{actual}$)

    /* Prepare features $X_{train}$ as per Section 3.2.5                          */
    $X_{train} \leftarrow prepareFeatures(D[C-15:C])$

    /* Train meta-model                                                  */
    $M = StackingClassifier($
        /* Use filtered model versions as base estimators                */
        $baseEstimators = H_{filtered},$
        /* Use Random Forest or Logistic Regression (Section 4.3) as meta
         classifier                                                      */
        $metaEstimator = RandomForestClassifier(),$
        /* Treat the base estimators as black boxes                      */
        $fitBaseEstimators = false,$
        /* Use commit features for meta classifier                       */
        $useCommitFeatures = true,$
    )
    $M.fit(X_{train}, Y_{actual})$
    return $M$
End Function
```

One uses only base (stacked) models' prediction probabilities as features, while the other configuration adds the commit features (i.e., commit metrics and code changes) in conjunction with the base models' prediction probabilities as features. When training the meta-model with only stacked models' probabilities, we utilize LR, as this configuration involves a smaller feature space. However, we use RF when training the meta-model with stacked model probabilities and original model features. The scikit-learn library's *StackingClassifier* (Pedregosa et al. 2011) is used to implement the model stacking strategy.

Furthermore, we evaluate this strategy using different data sampling configurations, which are as follows:

1. **all** – Utilize the full aggregated training data of the model history, including the training data of the RFS model.
2. **random** – Randomly sample training data (equal to the window size) from the aggregated training data of the model history.
3. **randomly resampled** – Randomly resampled training data (equal to the window size) from the aggregated training data of the model history based on class labels. Since the source dataset is imbalanced, this configuration

ensures that the data points from each class are represented equally (50%-50%) in the training sample.

Model stacking is a more complex strategy than the other two strategies, as it requires training a meta-model and there are more than two models required for inference. However, it can be an effective strategy since involving more models trained on different data from different time windows can help the meta-model to generalize better.

4.4 Model Voting

The Model Voting strategy, similar to Model Stacking, utilizes stacked MLMs for inference. However, the key distinction lies in the approach used to combine the predictions from the base models. Figure 8d illustrates that Model Voting employs a straightforward voting mechanism. Unlike Model Stacking, there is no meta-model involved in this strategy. The process can be split into two stages: *pre-inference* and *at-inference*. The pre-inference stage involves filtering the model history and the at-inference stage involves using the filtered model history to make predictions.

pre-inference – As shown in Algorithm 6, for each window C, using the same filtering algorithm 4 used in Model Stacking, we filter the model history using statistical tests, similar to model stacking. The statistical tests help us to identify the most similar models (in terms of model predictions) in the model history as explained earlier in Figure 9. Finally, we create the voting classifier by creating an ensemble of the filtered model versions as base estimators.

at-inference – As new commits arrive, the base estimators predict probabilities for each commit. We then use the voting mechanism to combine the probabilities from the base estimators to make the final prediction for the new commit.

We implement the Model Voting strategy using mlxtend's *EnsembleVoteClassifier* (Raschka 2018). By default, the *EnsembleVoteClassifier* re-trains the base estimators, but we configure it to treat them as black boxes. So, there is no training involved in this strategy. Additionally, instead of creating an ensemble of different models trained at the same time on the same data, we create an ensemble of historical models, including the latest model, trained on different data from different time windows.

We experiment with both hard and soft voting mechanisms. Hard voting is the simplest form of majority voting, where the class label ($\hat{y}$) is predicted based on majority (plurality) voting across all classifiers C_j ($\hat{y} = \mathbf{mode}\{C_1(x), C_2(x), ..., C_m(x)\}$). On the other hand, soft voting chooses the class label with the highest cumulative probability across all models, retrieved by aggregating predicted probabilities p from each classifier C_j, i.e., $\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$, where w_j represents the weight assigned to the jth classifier and p_{ij} represents the predicted probabilities of each class label i by jth classifier. We do not explicitly provide weights to the $EnsembleVoteClassifier$

Algorithm 6: Creating Model Voting *Recycling* Model for the Current Window

```
   Input:
   $C$                                        /* Current Time Window */
1  $K$                          /* Past Knowledge of Models' Performance */
2  $D$                                         /* Seen Training Data */
3  $H_{all}$                                /* Model Version History */
   Output: $M$                              /* Meta Classifier Model */
4  Function modelVoting($C$, $K$, $D$, $H_{all}$):
5      $Y_{prob} \leftarrow K[C-15:C]['predictedProbability']$
6      $Y_{actual} \leftarrow D[C-15:C]['actualLabel']$
7      $H_{filtered} \leftarrow$ filterModelHistory($C$, $H_{all}$, $Y_{prob}$, $Y_{actual}$)
8
9      $M = EnsembleVoteClassifier($
           /* Use the filtered model versions as base estimators      */
10         $baseEstimators = H_{filtered}$,
           /* Use soft voting to combine predictions                  */
11         $voting =' soft'$,
           /* Treat the base estimators as black boxes                */
12         $fitBaseEstimators = false$,
13     )
14     return $M$
15 End Function
```

so it assigns uniform weights to each base estimator giving equal importance to predictions from each classifier.

The Model Voting strategy requires multiple JIT models for inference. It can be an effective strategy because the base models are trained on different data from different time windows, providing diverse knowledge to the ensemble, which can be beneficial. However, it can be challenging due to drift, since voting relies only on base model prediction probabilities and there is no meta-model to calibrate the predictions from the base models like the model stacking approach does.

4.5 Clustering

The clustering strategy is depicted in Figure 8e. Intuitively, this strategy leverages the *ball tree* clustering algorithm to learn the similarity between commits from the current model history based on features retrieved from their commit metrics and code changes. Then for inferencing new commits, it utilizes the prepared *ball tree* to identify the closest neighbours and selects the best-performing model on these neighbouring samples for inference. The process can be split into two stages: *pre-inference* and *at-inference*. The pre-inference stage involves training the *ball tree* algorithm responsible for finding neighbouring samples, and the at-inference stage involves using the learned *ball tree* algorithm to make predictions.

pre-inference – As shown in Algorithm 7, for each window C, we first prepare the training features (X_{train}) using the same process as before. Second, we train the ball tree algorithm using the prepared features (X_{train}). We chose the *ball tree* algorithm because it is faster than both *brute force* and the *KD tree* algorithm for high-dimensional data (with $\geq$ 100 features).

Algorithm 7: Training Ball Tree Algorithm for the Current Window

```
   Input:
   $C$                                                  /* Current Time Window */
1  $D$                                                   /* Seen Training Data */
   Output: $M$                                       /* Nearest Neighbor Model */
2  Function createNNModel($C$, $D$):
3      $X_{train} \leftarrow D[C-15:C]['features']$
4
       /* Create NN model using commit metrics and code changes as features
        */
5      $M = NearestNeighbors(algorithm =' ballTree')$
6      $M.fit(X_{train})$
7      return $M$
8  End Function
```

Algorithm 8: Inference with Clustering for the Current Window

```
   Input:
   $C$                                                  /* Current Time Window */
1  $K$                                 /* Past Knowledge of Models' Performance */
2  $N_C$                                    /* Nearest Neighbor Model for $C$ */
3  $H_{all}$                                          /* Model Version History */
4  $kN$            /* # neighboring commits to consider in model selection */
5  $X_{new}$                                               /* Unseen commit ID */
   Output: $P_{new}$                             /* Predicted Label for $X_{new}$ */
6  Function modelVoting($C$, $K$, $N_C$, $H_{all}$, $kN$, $X_{new}$):
7      $commitIDs \leftarrow getKNearestCommitIDs(N_C, X_{new}, kN)$
8      $modelVersion \leftarrow getBestModelVersionByMSE(K[C-15, C], H_{all}[C-15:$
        $C], commitIDs)$
9      $P_{new} \leftarrow predict(modelVersion, X_{new})$
10     return $P_{new}$
11 End Function
```

at-inference – At inference time, we query the *ball tree* algorithm for each new commit to retrieve the k neighbouring (i.e., most similar in terms of feature vector) commits observed in the past. The primary objective is to identify the best model from model history based on the models' performance on these neighbouring commits. To achieve this, we evaluate the mean-squared error (MSE) of all models from the model history at the current window on these neighbouring commits. Afterwards, we find the model with the best MSE from the evaluated model history on these neighbours, and use it for inference on the new commit as outlined in Algorithm 8. The implementation of the clustering strategy incorporates scikit-learn's *NearestNeighbors* (Pedregosa et al. 2011).

The process involved in inference can be seen as costly, as it requires evaluating the available model history on k neighbouring commits at inference time. To reduce the computation cost at-inference, in turn reducing delay in inference, we propose to use *absolute error lookup* by evaluating the models at the pre-inference time. At the pre-inference time, we evaluate the models on each commit in the training data seen by the available model history. We then store the absolute error of each model for each commit from the training data. At inference time, we retrieve the pre-evaluated absolute error of models for the k-neighbouring commits of the new commit. We then aggregate the absolute errors on k-neighbouring commits into mean-squared-error (MSE) for each model and use MSE to identify the best model for the new commit.

Clustering is an intuitive strategy that selects the best model based on the similarity of the new commit to the commits observed in the past. Since this strategy identifies the best model at runtime, it requires evaluating multiple models at runtime, which can be computationally expensive. These computation costs can be reduced by using the *absolute error lookup* as discussed earlier. Clustering can be an effective strategy since it selects the best model based on its local performance on the most similar commits in the past.

5 Empirical Evaluation of Recycling Strategies

In this section, we evaluate the recycling strategies proposed in Section 4 to answer RQ2 and RQ3. For each research question, we first describe the approach we use to evaluate the recycling strategies. Then, we present the results of our case study and discuss the findings.

5.1 RQ2: What variant works best for each recycling strategy?

5.1.1 Motivation

This research question aims to find out what variant of each recycling strategy exploits the most value out of old models when compared to exclusively using a model that is retrained from scratch (RFS). We do not yet compare across recycling strategies.

5.1.2 Approach

Each proposed recycling strategy can be implemented with various configurations, which we refer to as a **variant**. These variants are created based on the different configuration dimensions discussed in Section 4 as listed in Table 6. For each recycling strategy, we empirically evaluate its multiple variants to find the best variant for this research question. The data used for this evaluation is the same as in RQ1 (Section 3). The baseline (RFS) is created by using the same setup as described in Sections 3.2.1 to 3.2.6.

For each variant, we collect performance vectors by evaluating the variant on the metrics discussed in Section 3.2.6 for each time window. As a baseline, we use the respective model that is retrained from scratch (RFS) for each window, e.g., RFS Random Forest model for Random Forest model variants and RFS Logistic Regression model for Logistic Regression model variants. We use the Wilcoxon signed-rank test (Woolson 2007) to find the variants that statistically significantly perform better than the baseline, using Bonferroni correction (Weisstein 2004) to adjust the p-value for multiple comparisons. Then, to evaluate the effect size for each variant with significant change, we use Cliff's delta effect size (Macbeth et al. 2011). Cliff's delta measures the observable effect of significant differences. We followed the values provided by Hess and Kromrey (2004) for interpreting the result of this value. Values smaller than 0.147 are negligible, values in the range [0.147, 0.33) are small, values in the range [0.33, 0.474) are medium, and values greater than 0.474 are large. For detailed *p*-value statistics and the effect size of each variant, please refer to the online replication package (Patel 2023).

To identify whether a significant change is positive or negative, we calculate the difference in mean metric performance between recycling and baseline performance vectors. If the value is negative, we mark that as performance degradation, while, if positive, we mark that as performance improvement compared to the baseline. For detailed differences in the mean metric performance of each variant, we again refer to the online replication package (Patel 2023).

To summarize the statistical results for each variant, we show for each performance metric the number of projects having significant improvement or degradation out of the eight projects evaluated. To highlight whether the change is positive or negative, we show superscript "+" for improvement and "−" for degradation. We also highlight the cells based on the summarized effect size of the change. For a single variant, the summarized effect size is the most frequent effect size across all projects. In case of a tie, we superscript the cell with a "∗" and highlight the cell with the greater effect size out of the two. For example, if a variant improves a performance metric in 4/8 projects by a small effect size and in 4/8 projects by a large effect size, we highlight the cell with a large effect size and superscript the cell with a "∗".

5.1.3 Results

5.1.3.1 Model Selection

For Random Forest (RF) and Logistic Regression (LR) models, the model selection recycling variants were able to significantly improve recall and g-mean compared to RFS for at least 25% of the projects, while they suffered significant performance degradation for Neural Network (NN) models. Tables 7, 8 and 9 show the number of projects that experienced significant change per metric for the RF, LR and NN models, respectively. The RF variants show similar improvements in recall, g-mean, AUC

Table 6: List of variants evaluated for each recycling strategy

Recycling Strategy	**Variant Dimension**	**Description**	**Evaluated Options**
Model Selection	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Metric	Metric used for filtering the model search space	f1-score, g-mean
	Post Inference	Post inference method used to combine old and new model's prediction probabilities	average-probabilities
Single Model Reuse	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Metric	Metric used for filtering the model search space	f1-score, g-mean
Model Stacking	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Test	Statistical test used for filtering the model search space	Wilcoxon, ScottKnott
	With Commit Features	Flag indicating whether commit features (i.e. commit metrics and code changes) were used in training meta-model	yes/no
	Data Sample	Data sampling technique used to sample data from seen data	all, random (N=1000), randomly resampled (N=1000)
Model Voting	Model Search Space	Length of model version history	Limited (=15), All
	Filtering Test	Statistical test used for filtering the model search space	Wilcoxon, ScottKnott
	Voting Method	Voting method used to combine probabilities or votes from ensemble of classifiers	soft, hard
Clustering	k-Neighbors	# neighboring samples used to evaluate and select the best model for unseen data sample	20, 40

and F1 for 2-3 projects with effect sizes ranging from negligible to medium. However, there is a noticeable decline in precision for 1-2 projects, with negligible to small effect sizes. Determining a clear winner is challenging, as all variants display similar performance, ± one project (effect size). Among the LR variants, specifically #2, #4, #6, and #8, stand out as the top performers. They demonstrate improvements (up to a median of 2.6-4%) in precision, AUC, and F1, without any performance degradation. Other LR variants result in decreased recall (up to a median of 5.5%) while improving other metrics but only for one or two projects. In contrast, NN variants consistently exhibit performance degradation across almost all projects, with large effect sizes observed for all metrics. This can be because of the high variance in the performance of NN models, which makes it difficult to find a good old model for inference. We will not discuss NN models further in this section since it seems that the model selection strategy does not work well for NN models.

For both RF and LR, variants #5-8 were able to perform similarly as variants #1-4, despite using a simpler configuration, since they only require a *limited* model search space, in turn requiring less computations. While for RF, variants #2, #4, #6 and #8 show improvements in more projects than the other four variants, these improvements are not as significant as the other four variants because of the negligible effect size in AUC, g-mean and F1. However, variants #4 and #8 can be argued to be slightly better choices in practice due to no significant degradation in precision. Similarly, for LR, variants #2, #4, #6 and #8 show improvements in more projects than the other four variants, with variants #4 and #8 showing slightly better performance in terms of g-mean, precision and F1.

filtering metric – Out of all variants, one can see that using *g-mean* as the filtering metric resulted in slightly more projects experiencing significant improvements in precision, g-mean and F1. This can be because the g-mean gives equal importance to both classes (Section 3.2.6). This characteristic helps the filtering process in finding the best old model that performs well on both class labels, in turn helping the strategy to pick the best old model for inference.

post-inference – We also observe that using the post-inference method (i.e., *averaging probabilities of the old and new models*) instead of using a *recycling* model, improved AUC in $> 25\%$ projects for all variants (i.e., #2, #4, #6 and #8), with negligible to medium effect size and resulted in lower recall improvements. This can be because using post-inference averages the probabilities of both old and new models, which favours the prediction towards a more confident model (i.e., class label with higher probability). Since confident predictions improve the ROC curve, it results in better AUC but this does not mean the model was able to make more correct predictions. This can happen if the model adjusts its predicted probabilities across the board, making them more extreme, but still maintaining the same decision boundary for classification.

5.1.3.2 Model Reuse

The best model reuse recycling variants were able to significantly improve recall, g-mean and F1 for at least one project in case of RF, did not improve or degrade significantly in case of LR and degrade significantly in case of NN. Table 10 shows that the RF variants had very similar performance, with improvements in 2-4 (small-large effect, up to a median of 7.4-8%), 1 (small-medium effect, up to a median of 4.5-5.8%) and 1 (negligible-small effect, up to a median of 3.8-4%) project(s) for recall, g-mean

Table 7: Model Selection (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Preci- sion	Re- call	G- Mean	AUC	F1
all	1	f1 score	n/a	2^{-*}	2^{+*}	0	0	0
	2	f1 score	avg. prob.	1^{-}	2^{+*}	1^{+}	2^{+}	1^{+}
	3	g-mean	n/a	1^{-}	2^{+*}	0	0	0
	4	g-mean	avg. prob.	0	2^{+*}	2^{+}	3^{+}	1^{+}
limited	5	f1 score	n/a	2^{-*}	2^{+*}	0	0	0
	6	f1 score	avg. prob.	1^{-}	2^{+*}	1^{+}	2^{+}	1^{+}
	7	g-mean	n/a	1^{-}	2^{+*}	0	0	0
	8	g-mean	avg. prob.	0	2^{+*}	2^{+}	3^{+}	1^{+}

Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 8: Model Selection (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Preci- sion	Re- call	G- Mean	AUC	F1
all	1	f1 score	n/a	3^{+}	$2^{+}2^{-*}$	2^{+}	1^{+}	1^{+}
	2	f1 score	avg. prob.	3^{+}	2^{+}	1^{+}	6^{+}	2^{+*}
	3	g-mean	n/a	5^{+}	2^{-*}	1^{+}	0	2^{+*}
	4	g-mean	avg. prob.	4^{+}	0	4^{+*}	5^{+}	3^{+}
limited	5	f1 score	n/a	3^{+}	$2^{+}2^{-*}$	2^{+}	1^{+}	1^{+}
	6	f1 score	avg. prob.	3^{+}	2^{+}	1^{+}	6^{+}	3^{+}
	7	g-mean	n/a	5^{+}	2^{-*}	2^{+*}	0	2^{+*}
	8	g-mean	avg. prob.	4^{+}	0	4^{+}	5^{+}	3^{+}

Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

and F1, respectively. These improvements come at the cost of degradation in 3 (small effect, up to a median of 4.5-4.8%), 0-1 (medium effect, up to a median of 3.7) and 0-1 (negligible effect, up to a median of 1%) project(s) for precision, g-mean and AUC metric performance, respectively. While variants

Table 9: Model Selection (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Post Inference	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	n/a	7^-	5^-	6^-	7^-	7^-
	2	f1 score	avg. prob.	7^-	5^-	6^-	7^-	7^-
	3	g-mean	n/a	7^-	6^-	6^-	7^-	7^-
	4	g-mean	avg. prob.	7^-	6^-	6^-	7^-	7^-
limited	5	f1 score	n/a	7^-	5^-	6^-	7^-	7^-
	6	f1 score	avg. prob.	7^-	6^-	6^-	7^-	7^-
	7	g-mean	n/a	7^-	6^-	6^-	7^-	7^-
	8	g-mean	avg. prob.	7^-	6^-	6^-	7^-	7^-

Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

#1 and #3 show higher effect size improvements than the other two variants, they lack in the overall number of projects experiencing improvements.

Table 11 shows that LR variants did not make enough significant changes in the performance of any metrics to say that recycling the old model is better than retraining from scratch. Worse, Table 12 shows that the NN variants resulted in significant degradation in all projects for all metrics. Both model selection and model reuse strategies depend significantly on the selection of the best old NN model and result in significant degradation in performance due to its over-reliance on the old NN model. This suggests that old NN models are not able to generalize to other windows. This can again be because of the high variance in the performance of NN models over different windows, which makes it difficult to find a good old model for inference.

model search space – Although there is no clear winner, variants #3 and #4 use a simpler configuration, since they use a *limited* model search space, which requires less computation.

filtering metric – Similar to model selection, the model reuse recycling strategy results in slightly more projects experiencing improvements in recall and g-mean when using *g-mean* as a filtering metric, although one project had a larger effect size improvement in recall for *f1-score* filtering. This can be because of the g-mean's characteristic of giving equal importance to both classes in the score (3.2.6). Since this recycling strategy reuses the old model as is, using g-mean as a filtering metric helps in selecting the model with the best performance on both class labels.

Table 10: Model Reuse (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	3^{-}	2^{+*}	$1^{+}1^{-*}$	1^{-}	1^{+}
	2	g-mean	3^{-}	4^{+}	1^{+}	0	1^{+}
limited	3	f1 score	3^{-}	2^{+*}	$1^{+}1^{-*}$	1^{-}	1^{+}
	4	g-mean	3^{-}	4^{+}	1^{+}	0	1^{+}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 11: Model Reuse (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	2^{-}	1^{+}	0	0	0
	2	g-mean	1^{+}	0	0	1^{+}	0
limited	3	f1 score	$1^{+}2^{-}$	1^{+}	0	0	0
	4	g-mean	0	0	0	0	0

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 12: Model Reuse (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Metric	Precision	Recall	G-Mean	AUC	F1
all	1	f1 score	7^{-}	5^{-}	6^{-}	7^{-}	7^{-}
	2	g-mean	7^{-}	6^{-}	7^{-}	7^{-}	7^{-}
limited	3	f1 score	7^{-}	5^{-}	6^{-}	7^{-}	7^{-}
	4	g-mean	7^{-}	6^{-}	7^{-}	7^{-}	7^{-}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

5.1.3.3 Model Stacking

The best model stacking variant for RF uses the *Wilcoxon* test for filtering model version history, *randomly resampled* or *all* samples

for training, and both stacked model probabilities and *commit features.* In Table 13, RF variants #2, #3, #5, #6, #8, #9, #10, and #11 stand out with the highest number of projects experiencing significant improvements. They demonstrate improvements in recall (5-6 projects, medium to large effect, up to a median of 17-35.8%) and g-mean (2-4 projects, large effect, up to a median of 11.8-21.3%). However, there is a trade-off with a decline in precision (up to a median of 9%) across 2-6 projects with small effect sizes. Among these variants, #6 and #8 win by minimizing the number of projects with small effect size degradation in precision while significantly improving recall and g-mean in the majority of projects (up to 75% and 50% of projects, respectively).

The most effective LR model stacking variant adopts the *Wilcoxon* test for filtering model version history, *randomly resampled* or *all* samples for training, and both stacked model probabilities and *commit features.* Alternatively, variants using the *ScottKnott* test with *all* samples for training, combined with both stacked model probabilities and *commit features*, also prove effective. Table 14 shows that LR variants #1, #5, #6, #8, #9, #10, and #11 have the most projects with significant improvements in precision (4-6 projects, small to large effect, up to a median 5.4-14%), g-mean (4-6 projects, medium to large effect, up to a median 5.7-8%), AUC (6-7 projects, large effect, up to a median 5.8-11%), and F1 (5-7 projects, small to medium effect, up to a median 5-8.7%). However, there is a decrease in recall (up to a median of 9%) in one project with a medium effect size. Variants #1, #6, and #8 stand out by having only one or no project with medium effect size degradation in recall while significantly improving other metrics for at least 60% of projects with significantly small to large effect sizes.

The optimal NN model stacking variant uses either the *Wilcoxon* or *ScottKnott* test for filtering model version history, *randomly resampled* or *all* samples for training, and both stacked model probabilities and *commit features.* Finally, Table 15 shows that NN variants #1, #5, #6, #8, and #11 exhibit the most projects with significant improvements in g-mean (4-6 projects, large effect, up to a median 7.9-9.8%), AUC (6-7 projects, large effect, up to a median 7.6-10%), and F1 (6-7 projects, small to large effect, up to a median 9.8-12.4%). However, they show a decline in precision (up to a median 4.3-8.8%) and recall (up to a median 22-23%) in 1-2 projects with medium to large effect sizes. Variants #1, #5, #6, and #8 excel by using simpler configuration (*limited* model search space) and providing similar improvements in g-mean and AUC for at least 50% and $> 75\%$ of projects with significantly large effect sizes, respectively.

filtering test – When comparing *ScottKnott* and *Wilcoxon* filtering test variants side-by-side, overall *Wilcoxon* performed better than *ScottKnott* by having improvements in slightly more projects with higher effect size. This can be because *Wilcoxon* resulted in more models remaining for stacking than *ScottKnott*, which means that more statistically different models provide input to the meta-model for training. These models are trained on different time windows, which can help the meta-model to generalize better on unseen commits.

data sampling technique – Out of all data sampling variants, *randomly resampled* worked best, followed by using *all* data, and using a *random sample*. This can be because the *random sample* variant outputs training data with skewed label distribution due to label imbalance in the source dataset. This label bias in data propagates to the model and negatively impacts the results. Compared to the *all* data variants, *randomly resampled* performed better while using less training data, in turn requiring less computation.

model search space – Using the full model search space ("*all*") results in a high number of models used for stacking and can increase the feature space, in turn increasing the compute time with no or negative effect on the improvements of g-mean, AUC and F1 results as shown in Tables 13, 14 and 15. This is why we did not try all possible variants under the *all* model search space variants. Between the *all* and *limited* model search space, *limited* seemed to provide improvements in more projects than using *all*.

commit features – Overall, using *commit features* along with stacked model probabilities resulted in fewer projects with degradation in precision than using only stacked model probabilities, while giving a similar performance in other metrics, or resulted in more projects with improvements across multiple metrics. This can be because *commit features* can provide additional information to the model including information about underlying feature distribution changes between time windows, which can help the model to improve on classification mistakes made by base models. However, using *commit features* can increase the feature space and hence the training time.

5.1.3.4 Model Voting

The best model voting recycling variant for RF and LR uses the *Wilcoxon* test for filtering model version history, *limited* model version history and *soft* voting, while no significant improvements were observed for NN. Table 16 shows that RF variants #1 and #2 demonstrate the highest number of projects with significant improvements and the fewest instances of precision degradation (up to a median of 3.2%). Both variants show similar improvements in recall (1 project with a medium effect, up to a median of 7.5%) and AUC (1 project with a negligible effect, up to a median of 1.5%). However, variant #2 slightly outperforms by having a negligible effect size degradation in precision.

In Table 17, LR variants #1, #2, and #3 lead in projects with improvements across various metrics: precision (2 projects with a small effect, up to a median of 6%), recall (1 project with a medium-large effect, up to a median of 10.7%), AUC (2-3 projects with a medium effect, up to a median of 5.8%), and F1 (1-3 projects with a small effect, up to a median of 4.6%). While variant #2 shows no degradation and more projects with improvements, it only marginally outperforms the others.

Table 13: Model Stacking (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	1	ScottKnott	yes	all	1^{-}	🟥 3^{+*}	🟧 2^{+*}	1^{+}	🟨 1^{+}
	2	ScottKnott	no	all	🟨 6^{-}	🟥 6^{+*}	🟥 3^{+*}	1^{+}	🟨 1^{+}
	3	ScottKnott	no	randomly resampled	🟨 6^{-}	🟥 5^{+}	🟥 2^{+*}	1^{+}	🟨 1^{+}
	4	ScottKnott	yes	random	🟨 $2^{+}1^{-}$	🟨 $2^{+}2^{-*}$	🟨 2^{-}	0	1^{-}
	5	ScottKnott	yes	randomly resampled	🟨 5^{-}	🟥 6^{+}	🟥 4^{+*}	1^{+}	🟧 1^{+}
	6	Wilcoxon	yes	all	🟨 2^{-*}	🟥 6^{+*}	🟥 4^{+*}	🟨 1^{+}	🟨 1^{+}
	7	Wilcoxon	yes	random	🟧 1^{+}	🟨 $2^{+}2^{-*}$	🟨 2^{-}	0	0
	8	Wilcoxon	yes	randomly resampled	🟨 4^{-}	🟥 6^{+*}	🟥 4^{+*}	1^{+}	🟧 1^{+}
	9	Wilcoxon	no	all	🟨 6^{-}	🟧 5^{+}	🟥 3^{+}	1^{+}	🟨 1^{+}
	10	Wilcoxon	no	randomly resampled	🟨 6^{-}	🟧 5^{+}	🟥 3^{+}	1^{+}	🟨 1^{+}
all	11	Wilcoxon	yes	randomly resampled	🟨 4^{-}	🟧 5^{+}	🟥 3^{+}	1^{+}	🟧 1^{+}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Finally, Table 18 reveals that none of the NN variants succeed in improving model performance. This might be attributed to a generalization issue, where the models struggle to generalize well on unseen commits. Alternatively, it could be that filtered models were not different enough to be useful for voting.

model search space – Similar to model stacking, using *all* model version history impacted performance negatively, so we did not test other variants with the *all* model search space option.

voting method – Using *hard* voting resulted in performance degradation for RF, improved performance for LR models and negligible improvements for NN. The reason behind this variance in performance can be that the correct predictions have a higher chance of getting overshadowed by other models in the ensemble, as the *hard* voting variant relies on majority voting using models' predicted labels.

Table 14: Model Stacking (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	**1**	ScottKnott	yes	all	🟥 6^{+*}	🟥 $1^{+}1^{-*}$	🟥 5^{+}	🟥 7^{+}	🟧 7^{+*}
	2	ScottKnott	no	all	🟨 6^{+}	0	🟥 2^{+*}	🟧 4^{+}	🟨 2^{+}
	3	ScottKnott	no	randomly resampled	🟨 6^{+}	0	🟥 2^{+*}	🟧 4^{+}	🟨 2^{+}
	4	ScottKnott	yes	random	🟥 6^{+}	🟥 $1^{+}4^{-}$	🟥 $2^{+}1^{-}$	🟥 6^{+}	🟨 $3^{+}1^{-}$
	5	ScottKnott	yes	randomly resampled	🟨 4^{+}	🟥 5^{+}	🟥 5^{+}	🟥 6^{+}	🟨 6^{+}
	6	Wilcoxon	yes	all	🟨 6^{+}	🟥 $1^{+}1^{-*}$	🟥 5^{+}	🟥 7^{+}	🟧 6^{+}
	7	Wilcoxon	yes	random	🟥 6^{+}	🟥 $1^{+}4^{-}$	🟥 $2^{+}1^{-}$	🟥 6^{+}	🟧 $3^{+}1^{-*}$
	8	Wilcoxon	yes	randomly resampled	🟨 5^{+}	🟧 3^{+}	🟥 6^{+}	🟥 7^{+}	🟧 6^{+}
	9	Wilcoxon	no	all	🟨 6^{+}	0	🟥 4^{+*}	🟥 6^{+}	🟨 5^{+*}
	10	Wilcoxon	no	randomly resampled	🟨 6^{+}	0	🟧 5^{+}	🟥 6^{+}	🟨 5^{+}
all	11	Wilcoxon	yes	randomly resampled	🟨 5^{+}	🟥 4^{+}	🟥 5^{+}	🟥 6^{+}	🟧 6^{+*}

Cell Highlights (Cliff's Delta Effect Size)	🟨 Small	🟧 Medium	🟥 Large

filtering test – Overall, using the *Wilcoxon* test for filtering model version history works slightly better for the model voting recycling strategy. This can be because the *Wilcoxon* filtering again results in more models than *ScottKnott*, i.e., more models are used in the ensemble than *ScottKnott*. This can lead to better performance, since voting relies only on the predicted probabilities of the models as input.

5.1.3.5 Clustering

Although model clustering sounds like an intuitive way for recycling, it does not result in any performance gain; instead, it degrades the

Table 15: Model Stacking (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	With Commit Features	Data Sample	Precision	Recall	G-Mean	AUC	F1
limited	**1**	ScottKnott	yes	all	🟧 5^+1^-	🟥 2^+2^-	🟥 4^+1^-	🟥 7^+	🟥 7^+
	2	ScottKnott	no	all	0	🟨 1^+	0	0	0
	3	ScottKnott	no	randomly resampled	0	🟨 1^+	0	0	0
	4	ScottKnott	yes	random	🟥 5^+1^-	🟥 2^+2^-	🟥 2^+2^-	🟥 6^+	🟥 2^+1^{-*}
	5	ScottKnott	yes	randomly resampled	🟨 1^+	🟥 4^+	🟥 6^{+*}	🟥 6^+	🟨 6^+
	6	Wilcoxon	yes	all	🟥 5^+1^-	🟥 2^+2^-	🟥 4^+1^-	🟥 7^+	🟥 7^+
	7	Wilcoxon	yes	random	🟥 5^+1^-	🟥 2^+2^-	🟥 2^+2^-	🟥 7^+	🟨 2^+1^-
	8	Wilcoxon	yes	randomly resampled	🟨 1^+	🟥 4^+	🟥 6^{+*}	🟥 6^+	🟨 6^+
	9	Wilcoxon	no	all	0	🟧 2^{+*}	0	1^+	0
	10	Wilcoxon	no	randomly resampled	0	🟨 1^+	0	0	0
all	11	Wilcoxon	yes	randomly resampled	🟨 1^+	🟥 4^+	🟥 6^{+*}	🟥 6^+	🟨 6^+

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

performance by a large margin for all models, i.e., RF, LR and NN. Tables 19, 20 and 21 show that all variants showed a significant reduction in metric performance for almost all projects, without significant improvement.

k-neighbors – Increasing the # of neighbours in variant #2 to twice that of variant #1 did not make any improvements in performance metrics. Given the inclusion of 15 different models in the search space, relying solely on the model with the lowest MSE may not always be the optimal approach. This is because the top-performing model in terms of MSE could potentially be overly specialized to the training data, leading to overfitting issues. As a result, the selection process tends to favour models that have been trained on a time

Table 16: Model Voting (RF): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Preci-sion	Re-call	G-Mean	AUC	F1
limited	1	Scot-tKnott	soft	1^{-}	1^{+}	0	1^{+}	0
	2	Wilcoxon	soft	2^{-}	1^{+}	0	1^{+}	0
	3	Wilcoxon	hard	1^{-}	2^{+}	1^{-}	1^{+}	1^{-}
all	4	Wilcoxon	soft	2^{+*}	$1^{+}4^{-}$	4^{-}	1^{+}	2^{-}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 17: Model Voting (LR): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Preci-sion	Re-call	G-Mean	AUC	F1
limited	1	Scot-tKnott	soft	2^{+*}	$1^{+}1^{-*}$	2^{+*}	2^{+}	2^{+}
	2	Wilcoxon	soft	2^{+*}	1^{+}	0	3^{+}	3^{+}
	3	Wilcoxon	hard	2^{+*}	1^{+}	0	3^{+}	1^{+}
all	4	Wilcoxon	soft	4^{+}	$1^{+}4^{-}$	1^{-}	3^{+}	1^{+}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 18: Model Voting (NN): # of projects (out of 8) experiencing significant change per evaluation metric

Model Search Space	#	Filtering Test	Voting Method	Preci-sion	Re-call	G-Mean	AUC	F1
limited	1	Scot-tKnott	soft	0	0	0	0	0
	2	Wilcoxon	soft	0	1^{-}	0	0	0
	3	Wilcoxon	hard	1^{+}	0	0	0	0
all	4	Wilcoxon	soft	2^{-*}	1^{+}	2^{-*}	2^{-*}	2^{-*}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large window containing most of the neighbouring commits. This inherent bias may help explain why the clustering strategy did not yield favourable results in our experimental setup.

Table 19: Clustering (RF): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7^{-}	7^{-}	7^{-}	7^{-}	7^{-}
2	40	7^{-}	7^{-}	7^{-}	7^{-}	7^{-}

Cell Highlights (Cliff's Delta Effect Size)	🟨 Small	🟧 Medium	🟥 Large

Table 20: Clustering (LR): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7^{-}	7^{-}	7^{-}	7^{-}	7^{-}
2	40	7^{-}	7^{-}	7^{-}	7^{-}	7^{-}

Cell Highlights (Cliff's Delta Effect Size)	🟨 Small	🟧 Medium	🟥 Large

Table 21: Clustering (NN): # of projects (out of 8) experiencing significant change per evaluation metric

#	k-Neighbors	Precision	Recall	G-Mean	AUC	F1
1	20	7^{-}	6^{-}	6^{-}	7^{-}	7^{-}
2	40	7^{-}	7^{-}	6^{-}	7^{-}	7^{-}

Cell Highlights (Cliff's Delta Effect Size)	🟨 Small	🟧 Medium	🟥 Large

Summary for Research Question 2

Model Selection and Model Reuse strategy variants performed very similarly to each other in terms of prediction performance, while Model Stacking and Model Voting strategy variants performed better when the *Wilcoxon* test was used for filtering model history. Although Clustering sounds like an intuitive way for recycling, it did not result in any performance gain; instead, it degraded the performance by a large margin. All strategies except Clustering showed significant improvements in at least one project for Random Forest (RF) and Logistic Regression (LR) models. However, none of the strategies except Model Stacking showed significant improvements in any project for Neural Network (NN) models.

5.2 RQ3: Which recycling strategy performs the best?

5.2.1 Motivation

After evaluating the variants of each recycling strategy in isolation in RQ2, we want to determine which strategy works best overall. Additionally, while a recycling strategy may work best in terms of the model's performance, we want to understand the trade-off between inference latency and model performance for each recycling strategy, because some strategies require one extra model inference to determine the best historical model, while others require inference of more than two historical models.

5.2.2 Approach

This research question builds on RQ2. Having found the best variant(s) for each recycling strategy, we compare each recycling strategy using its best variant. We reuse the performance results of RQ2, this time to determine the best variant across all recycling strategies.

As the best RF variant for each recycling strategy, we use variant #8 from model selection, variant #4 from model reuse, variant #8 from model stacking and variant #2 from model voting, respectively, because these showed the best performance in terms of # projects across evaluation metrics while using simpler configurations. We do not include any variant from clustering in this comparison because it did not show any significant improvements in any of the projects. Using the same criteria, we selected LR variants #8, #8 and #2 for model selection, model stacking and model voting, respectively. We do not include any variant from model reuse and clustering in this comparison, because they did not show any significant improvements in any of the projects. For NN, we selected #8 for model stacking only, since other strategies did not show any significant improvements in any of the projects.

Furthermore, we evaluate the inference latency of the selected best variant for each recycling strategy. As a baseline, we measure the inference latency of the RFS model on a given window. The inference latency is retrieved by measuring the time to inference on each test sample for that window, and doing this across all windows yielding $\sum_{project=1}^{project=8}(\#\text{ Evaluated Windows})_{project} *$ Test Size(=200) measurements. We only collect one time measurement per test sample since there are enough test samples for each project to generalize the latency. For example, the minimum # test samples for the project with the least # windows (9 for ActiveMQ) is $1,800$. We then compare each recycling strategy's best variant based on its metric performance and on its inference latency.

We perform statistical tests on the latency measurements of each recycling strategy's best variant against the baseline (no recycling). We use the Wilcoxon signed-rank test to determine if the latency measurements of each recycling strategy's best variant are significantly different from the baseline and we use the Cliff's Delta effect size to determine the magnitude of the difference between them. Additionally, to help understand latency differences, we measure the median latency ratio of the recycling strategy's best variant to the baseline.

5.2.3 Results

The model stacking recycling strategy gives the best improvements in terms of prediction performance in recall, g-mean and F1 among all recycling strategies tested. Tables 22, 23 and 24 show the significant performance metric changes for each best variant of RQ2 for all models (RF, LR and NN). For RF, out of all recycling strategies evaluated in this study, model stacking showed a significant increase in 4/8 and 6/8 projects for g-mean and recall respectively (compared to RFS), with large effect sizes, despite a decrease in precision in 4/8 projects with small effect sizes. For LR, model stacking showed a significant increase in all metrics, while improving performance for $\geq 75\%$ of projects for g-mean, AUC and F1, with medium-large effect sizes. Finally, for NN, model stacking was the only strategy that improved performance with 6/8 projects in g-mean, AUC and F1. Model stacking is an ensemble approach, and when used with multiple generalizers trained on different time windows, it seems to get the most out of old estimators across all models.

Model selection and model voting are second-best recycling strategies in terms of prediction performance. There is no clear second-best recycling strategy between model selection and model voting. For RF and LR, The model selection recycling strategy shows improvements across more projects with a negligible-medium effect size. On the other hand, for RF, model voting shows slight improvements in recall and AUC, while for LR, it shows higher effect size improvements in Recall, AUC and F1 for comparatively more projects. Finally, for NN, both model selection and model voting showed no significant improvements as discussed earlier. Model selection uses the *recy-*

cling model's prediction to decide which model to reuse for inference, whereas model voting is an ensemble approach that uses the predictions of multiple models to decide the final prediction. In other words, model selection, while using fewer models than model voting, can provide similar performance improvements if not better than model voting.

The model reuse recycling strategy comes last in terms of prediction performance. While it was able to boost the performance of recall and g-mean in some projects, it led to precision degradation in more projects than it was able to improve g-mean or F1. Additionally, it was only able to significantly improve performance for RF models, not for LR and NN. This can be because the model reuse recycling strategy's focus on reusing only the best old model can sometimes result in a model that is not well suited for the current data distribution.

Table 22: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (RF)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Selection	0	🟨 2^{+*}	2^{+}	3^{+}	1^{+}
Single Model Reuse	🟨 3^{-}	🟨 4^{+}	🟨 1^{+}	0	🟨 1^{+}
Model Stacking	🟨 4^{-}	🟥 6^{+}	🟥 4^{+}	1^{+}	🟧 1^{+}
Model Voting	2^{-}	🟧 1^{+}	0	1^{+}	0

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Table 23: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (LR)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Selection	4^{+}	0	🟧 4^{+}	🟨 5^{+}	3^{+}
Model Stacking	🟨 5^{+}	🟧 3^{+}	🟥 6^{+}	🟥 7^{+}	🟧 6^{+}
Model Voting	🟨 2^{+*}	🟥 1^{+}	0	🟧 3^{+}	🟨 3^{+}

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

Inference latency increases by 6-11x for model voting, 7-17x for model stacking, 2x for model selection, and remains the same for model reuse compared to the baseline. Figures 10, 11 and 12 show the inference latency results for each best variant under each recycling strategy tested for each model, while Tables 25, 26 and 27 show the difference factor between inference latency of baseline and recycling for each project and model

Table 24: # of projects (out of 8) experiencing significant change for the best recycling strategy variants (NN)

Recycling Strategy	Precision	Recall	GMean	AUC	F1
Model Stacking	1^{+}	4^{+}	6^{+*}	6^{+}	6^{+}

Cell Highlights (Cliff's Delta Effect Size)	🟨 Small	🟧 Medium	🟥 Large

alongside statistical test results. The baseline shows results when only using the latest model for every inference request.

Both model voting and model stacking result in significantly higher inference latency than the baseline, i.e., six to seventeen times. Between the two, model stacking requires higher inference times because of the additional computations required for meta-model inference. Tables 25, 26 and 27 and Figures 10c 11b and 12a show that these slowdowns are statistically significant in all projects and models with large effect sizes, with the median inference time ranging from 6-11x baseline for model voting and 7-17x baseline for model stacking. This is due to the ensemble nature of these two recycling strategies, i.e., we are doing inferencing on multiple models stacked together. Note that this may differ with the choice of model history filtering approach, because, in our study, we saw that the Wilcoxon test-based variant resulted in generally more than three models and sometimes up to ten or more after filtering. In contrast, the number of models stacked is significantly lower with the Scott Knott test-based variants. Even though both recycling strategy variants in the table used the Wilcoxon test for filtering, model stacking has a higher inference delay than model voting. This is because the selected model stacking recycling variant requires using the meta-model inference as well as *commit features* alongside stacked model probabilities, which increases the feature space and inference time.

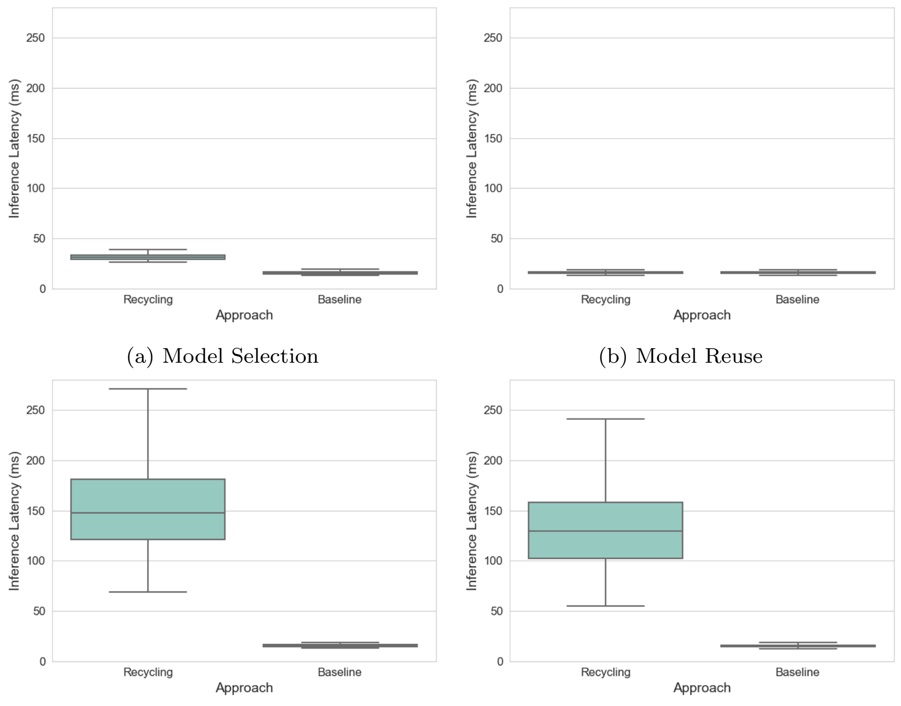
Model selection results in twice the inference latency of the baseline because of the additional inference required for each inference request. This is because the best variant tested for model selection averages probabilities of two predictions, meaning it always requires inference of two models before it can provide a final prediction. Hence, as expected, Tables 25 and 26 show that model selection has statistically significant differences in latency in all projects, with large effect sizes and with a median inference time of $\sim$ 2x baseline.

The model reuse recycling strategy results in the same inference latency as the baseline because it requires a single model inference for each inference request. For model reuse strategy, in both cases we are only doing inference with one model, i.e., inference with the old recycled model or baseline inference with the RFS model. Table 25 shows that the model reuse recycling strategy has a statistically significant difference in latency in only five projects, but with negligible effect size and median inference time being similar to baseline (1x).

The model selection and model stacking variants provide a good trade-off between inference latency and model performance. In practice, depending on the importance of inference latency or performance, an appropriate strategy can be chosen. Both model selection and model stacking are good strategies because of the trade-off between their metric performance and inference latency. Model selection, while giving performance improvements in at least 25% of projects, results in inference latency of 2x baseline. On the other hand, model stacking, while providing performance improvements in at least 50% of projects, results in inference latency of 7-17x baseline. Looking at model voting, it improves the performance in only $12 - 37\%$ of projects, while still resulting in inference latency of 6-11x baseline. Lastly, model reuse, while resulting in the same inference latency as the baseline and improving performance in $12 - 50\%$ of projects, also degrades precision for 37% of the projects.

The decision on a model recycling strategy is ultimately left to practitioners, guided by our analysis of the trade-off between predictive performance and inference latency. While inference latency remains a crucial consideration, it is not the sole determinant when opting for a recycling strategy. In scenarios where the model operates in real-time systems, inference latency takes precedence over model performance. Conversely, in batch systems, model performance outweighs inference latency.

Ensemble approaches can leverage parallel inference, diminishing the significance of inference latency in the selection of a recycling strategy. Our latency analysis reveals that some recycling strategies exhibit superior predictive performance but incur a substantial increase, ranging from 6 to 17 times, in inference latency compared to the baseline. Notably, this practical limitation can be mitigated by adopting parallel inference for stacked base models instead of a sequential approach. Furthermore, it is essential to stress that the observed increase in inference latency, although noteworthy, consistently remains within the acceptable fraction-of-a-second range ($<$ 300 milli-seconds), aligning precisely with the demands of the intended application scenarios for just-in-time software defect prediction applications. Take, for example, pull request review systems, where predictions occur while developers await the review's completion, which could take hours or even days.



(a) Model Selection (b) Model Reuse

(c) Model Stacking (d) Model Voting

Fig. 10: Inference Latency per Recycling Strategy (RF)

Table 25: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (RF)

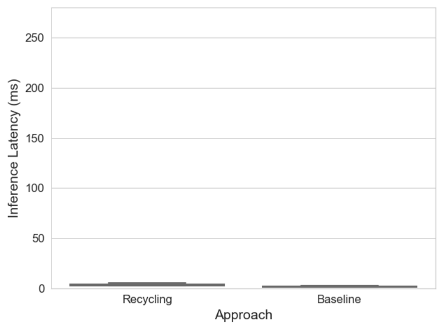
Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Selection	2.016	2.017	2.017	2.029	1.996	1.990	1.988	2.023
Model Reuse	1.001	1.004		1.012	1.003		1.004	
Model Stacking	7.598	10.356	8.889	10.676	9.287	9.001	8.598	9.325
Model Voting	6.757	9.232	7.811	9.453	8.369	8.230	7.652	7.994

Cell Highlights (Cliff's Delta Effect Size) 🟨 Small 🟧 Medium 🟥 Large

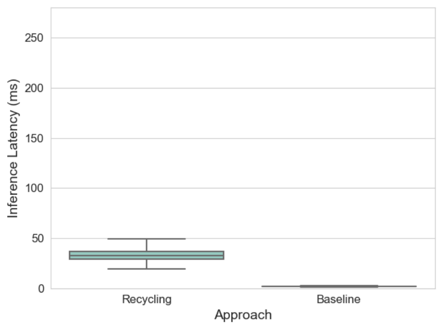
Note: *blank cells above indicate that there was no statistically significant difference between the latency of the baseline and the recycling strategy for that project.*

Summary for Research Question 3

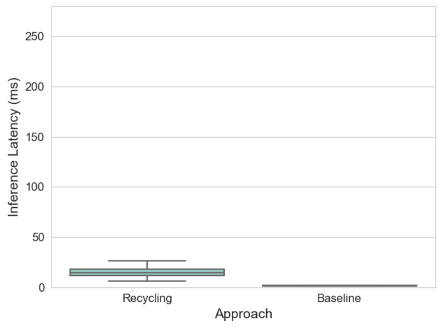
Model selection provides improvements in at least $\sim$ 25% of projects with 2x inference latency for RF and LR models, model stacking provides improvements in at least $\sim$ 50% of projects with 7-17x inference latency for all models, model voting provides improvements in $\sim$ 12-37% of projects with 6-11x inference latency for RF and LR, and model reuse provides improvements in $\sim$ 12-50% of projects with no inference latency increase for only RF. Based on the trade-off between inference latency and model performance, model selection and model stacking seem to be the best recycling strategies.



(a) Model Selection

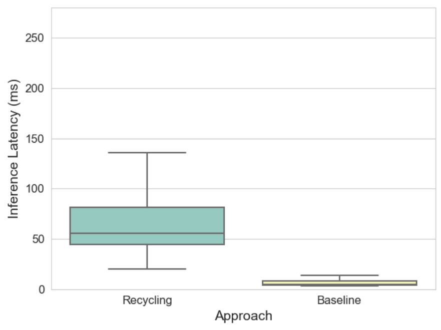


(b) Model Stacking



(c) Model Voting

Fig. 11: Inference Latency per Recycling Strategy (LR)



(a) Model Stacking

Fig. 12: Inference Latency per Recycling Strategy (NN)

Table 26: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (LR)

Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Selection	1.962	1.982	1.925	2.046	2.002	1.918	1.956	1.960
Model Stacking	16.409	17.811	15.246	16.935	16.052	15.566	12.900	17.015
Model Voting	10.105	11.612	8.141	11.723	8.974	9.856	6.921	11.057

Table 27: Median latency ratio of recycling to baseline and statistical test results for inference latency of the best variant of each recycling strategy ($\alpha = 0.05$) (NN)

Recycling Strategy	Project							
	ActiveMQ	Camel	Cassandra	Flink	Groovy	HBase	Hive	Ignite
Model Stacking	10.617	12.001	10.723	11.460	12.072	8.644	9.766	13.377

6 Threats to Validity

6.1 Internal Validity

In our experimental setup, we employed a sliding window technique to create data splits, utilizing a window size of one thousand samples. The choice of window size significantly impacts the number of samples available for training a single model. Due to the infeasibility of evaluating all possible window sizes, we addressed this potential threat to validity by conducting additional analysis using a window size of two thousand samples alongside our primary window size of one thousand samples. However, this did not show any significant improvements over the original window size. The results of these additional study results can be found in the replication package (Patel 2023).

Similarly, we maintained a window shift of two hundred samples for all analyses, which controls the overlap of data samples between consecutive windows. To explore the impact of different window shifts, we further tested our best variants with a window shift of three hundred samples. The results showed some improvements in the baseline but they are not significantly better than the original configuration settings used in this paper. The results of these additional study results can be found in the replication package (Patel 2023).

In Section 3, we propose several hypotheses to account for instances where older models outperform newer ones. While we hypothesize that such occurrences may stem from changes in project evolution, as previously discussed, our work does not provide definitive evidence to support this claim. We acknowledge the need for further investigation into this and consider it as a potential future work.

In the limited model history approaches, we set the model history size to fifteen, considering only the fifteen most recent model versions for recycling strategies. This limitation was a reasonable choice to avoid delving too far back into the history of models. Given that the cost of computing resources and time required to train and evaluate models increases with the model history size, we decided not to continue testing other variants with a larger model history size. However, to ensure a thorough investigation, we also conducted experiments without limit on the model history size, enabling us to explore potential improvements beyond the limited model history setting. Again, the results do not show improvements over the limited model history setting. This has been discussed in the results section of this paper.

We adopted hyperparameters from the JITLine model (Pornprasit and Tantithamthavorn 2021) as a baseline for our RF model training, hyperparameters from the DeepJIT model (Hoang et al. 2019) for our NN models and the default configuration for LR provided by scikit-learn (Pedregosa et al. 2011). Although hyperparameter tuning could potentially enhance the performance of our MLM, we deliberately avoided this process to maintain consistency across our experiments. Similar to Falessi et al. (2021), our focus does not lie in improving the performance of the defect prediction MLM. But rather on exploring the possibility of recycling old models to enhance the performance of existing MLMs, treating them as black boxes. By conducting these supplementary experiments and maintaining specific choices in hyperparameter tuning, we aim to provide a robust and comprehensive evaluation of our approach while mitigating potential internal threats to the validity of our study.

In our DeepJIT experiments, we do not use commit messages as features. This is because we want consistency in the features used across all models (RF, LR and NN). However, one can argue that models like DeepJIT can benefit additionally from textual features like commit messages. Therefore, we conducted additional experiments with DeepJIT using commit messages as features along with both commit metrics and code change features. The results when compared with DeepJIT without commit messages showed statistical significance, but with negligible effect size, in almost all metrics and projects. The results of these additional study results can be found in the replication package (Patel 2023).

6.2 External Validity

Our research used the ApacheJIT dataset (Keshavarz and Nagappan 2022) and Just-In-Time defect prediction models to conduct our experiments. It is essential to acknowledge that this dataset primarily comprises popular Apache projects and their associated known bugs, which may limit its representativeness compared to all software projects. Furthermore, since the projects within the dataset are written in the Java programming language, there may be concerns about the generalizability of our results to other programming languages. However, the commits level features used in this paper are language-independent and can be extracted from any software project. Since we use JIT defect prediction models to conduct our experiments, our results may not apply to other software analytics models.

However, despite these considerations, we firmly believe that our approach's fundamental principles and methodologies are applicable and generalizable beyond the confines of the ApacheJIT dataset or JIT defect prediction models. This adaptability extends to other software analytics techniques, further enhancing its potential utility. Therefore, while recognizing the specific context of our case studies, we are optimistic that our findings can serve as a valuable foundation for future research and practical applications across diverse software projects and programming languages.

Our study uses JIT defect prediction models leveraging random forest, logistic regression and neural network models. These models fall under traditional machine learning models, and are widely used for defect prediction (Zhao et al. 2023). Recently, Large Language Models (LLMs) have started to revolutionize many fields. However, even today, many companies are still training and deploying the majority of their own bread-and-butter non-LLM machine learning models (deep learning, traditional classification models, etc.). McKinsey's report[4] emphasizes that despite the rapid progress of generative AI, traditional AI applications (like advanced analytics and machine learning) still contribute more to the overall potential value (11.0-17.7 trillion dollars) compared to the incremental impact of new generative AI use cases (2.6-4.4 trillion dollars). As such model recycling techniques will remain relevant in the foreseeable future.

As to the applicability of model recycling to LLMs, the emergence of techniques like frugalGPT (Chen et al. 2023) offer an efficient and cost-effective approach to using LLMs by sequentially cascading multiple LLMs, from cheapest to most expensive. With various LLM APIs available, each with unique performance and costs, strategic selection based on query requirements can reduce expenses and enhance performance. The LLM cascade strategy sequentially sends a query to multiple LLM APIs, returning the first reliable (cheapest) response and avoiding more expensive queries, showing how multiple inferences with LLM models are not unrealistic. As such, one could imagine our model recycling strategies to be applied to LLMs as well, either to improve model performance (focus of the current paper) or to reduce inference costs (Chen et al. 2023). We leave this follow-up work as a potential future work.

6.3 Construct Validity

ApacheJIT uses the SZZ algorithm to find bug-inducing changes. The SZZ algorithm tends to label many clean commits as bug-inducing. However, Ke-

[4] https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/the-economic-potential-of-generative-ai-the-next-productivity-frontier#key-insights

shavarz and Nagappan (2022) use additional filtering steps that are shown to be effective in removing safe changes that are identified as bug-inducing by SZZ, and they utilized GumTreeDiff (Falleri et al. 2014) to remove trivial changes. Therefore, we believe that the bug-inducing changes in ApacheJIT are of high quality.

We use the sliding window technique to create data splits to mimic release cycles in real-world ML applications. This technique may not exactly represent every nuance of the real-world scenario, but it provides a structured and controlled way to simulate the progression of time and evolving data distributions. Despite its slight deviations from reality, we firmly believe that the sliding window technique is a valuable approximation.

7 Discussion

The reuse of historical models in general is not new. There have been several studies done in the past to exploit historical models to some extent for concept drift adaptation. We have listed them in Table 28 along with other related work. There are multiple distinctive features of our study compared to these studies, which we discuss in detail in this section.

Black-box – A major distinctive feature of our study is that we treat the historical models as black boxes. This is important, because retraining historical models can be costly and time-consuming, and would change the known, possibly certified behaviour of an existing model. Furthermore, training data or scripts of old models might no longer be available, while the historical, black-box model could still be valuable.

Time Window – Another distinctive feature of our study is that we limit the number of historical models to be reused, which in turn limits the historical data to a specific time window. This is because, as we add models to the ensemble, it increases complexity and it is impractical to load all historical models in memory at inference. However, we also evaluated variants without limiting the number of models where filtering is applied, which not only reduces historical models but also selects models trained on diverse data distributions. We believe that filtering can help in selecting the model trained on distinct data trends which can help the ensemble generalize better on unseen data.

Filtering History – The third distinctive feature of our study is that we filter the historical models before reusing them. This is because we want to avoid reusing models that are too similar to other models in the ensemble and might reduce the ensemble's performance. This is important because if we reuse similar models, it can lead to overfitting and reduce the diversity of the ensemble. Another advantage of filtering historical models is that it reduces the number of models to be reused, which reduces the number of models to be inferenced per commit in turn reducing the inference time.

While many studies do limit the historical models using a time window, they do not filter the models. They use weighted voting, which adjusts the weights of the models based on their performance on the latest data. This is done to avoid reusing models that are not performing well on the latest data. However, this approach does not consider the diversity of the models, which is important to avoid overfitting. This also means that all models in the ensemble are inferenced every time even if their weight in prediction is negligible, leading to higher latency.

Filtering Method – Filtering can be done in multiple ways. On the one hand, one could use statistical tests such as the Wilcoxon or ScottKnott test to cluster similar models and select models with statistically different error performances. On the other hand, one could also use clustering techniques or simply use performance metrics such as g-mean or f1-score to find the best historical model. We believe that filtering can be done in multiple ways and can be further explored in future work.

Domain – While prior research has explored historical model reuse for concept drift adaptation, our study is the first to assess its applicability in the context of software analytics models. It is important to note that the outcomes of black-box model reuse can vary across domains. For instance, in tasks like news article classification, as seen with TIX (Forman 2006), it consistently achieved over 10 points of F-measure improvement. In contrast, experiments by Tsymbal et al. (2008) using synthetic data sets emulating abrupt and gradual concept drifts, as well as a real-world antibiotic resistance data set, revealed that dynamically integrating classifiers built over small time intervals or fixed-sized data blocks can outperform methods like majority voting and weighted voting. As such, it is important to evaluate the performance of model recycling in the domain of software analytics.

In our study, we evaluate the performance of five model recycling strategies on Just-In-Time defect prediction models. When we compare the performance of recycling strategies across projects, we see a variance in performance. This is not only a problem with our study but also with other studies in the literature. For example, JITLine's (Pornprasit and Tantithamthavorn 2021) F1-score between OpenStack and QT differs by 10%. Similarly, in the study by Kamei et al. (2016) on JIT defect prediction, within-project JIT defect prediction F1-score varied up to 47% between eleven different projects studied. This variance in performance can be due to the difference in data distributions between projects. This is also the reason why we see a variance in performance across projects in our study. However, we believe that our study is still valuable because we see a consistent improvement in performance across projects for the best recycling strategy (Model Stacking). This also holds for performance degradation variants from the worst recycling strategy (Clustering).

Furthermore, we perform the first large-scale comparative study of multiple recycling strategies. We also evaluate the trade-offs between each strategy in terms of time-to-inference and metric performance. We believe that our study can be a valuable foundation for future research and practical applications across diverse software projects and programming languages.

Table 28: Distinctive Features: Prior Work vs. Our Approaches

Strategy	**Related**	**Description**	**Domain**	**Using Historical Models**	**Black-box**	**Filtering History**	**Filtering Method**	**Time Window**
Model Selection	**Our work**	**See Section 4.1**	**Software Analytics**	✓	✓	✓	**G-mean, F1-score**	✓
Model Reuse	**Our work**	**See Section 4.2**	**Software Analytics**	✓	✓	✓	**G-mean, F1-score**	✓
Model Stacking	TIX (Forman 2006)	New model is trained using augmented features and with historical model output	News Article Classification	✓	✓			✓
	Our work	**See Section 4.3**	**Software Analytics**	✓	✓	✓	**ScottKnott Wilcoxon**	✓
Model Voting	SEA (Street and Kim 2001)	Combines historical model output with new model output using majority voting	Salary Prediction, Cancer Detection, Web User Behaviour Prediction	✓	✓			✓
	Learn++ (Polikar et al. 2001)	Ensemble of weak learners trained on different data distributions combined using majority voting	OCR, Vehicle Classification, Gas Sensing, Synthetic (Numeric)	✓	✓			
	Learn++ .NSE (Elwell and Polikar 2011)	Combines historical model output with new model output using weighted voting where weights are calculated based on models' MSE on latest and previous data	Weather Prediction, Synthetic (Numeric, Image)	✓	✓			
	AUE2 (Brzezinski and Stefanowski 2013)	Combines historical model output with new model output using weighted voting where weights are calculated based on models' MSE on latest data only	Synthetic (Numeric), Energy Price Prediction, Forest Cover Type Prediction, Poker Hand Classification, Airline Delay Prediction	✓				✓
	DTEL (Sun et al. 2018)	Preserves a diverse set of historical models and retrains them on new data. Employs weighted voting scheme used by AUE2	Synthetic (Numeric), Energy Price Prediction, Forest Cover Type Prediction, Poker Hand Classification, Web User Behaviour Prediction, Tweet Classification	✓		✓	Yules Q-statistic	✓
	Our work	**See Section 4.4**	**Software Analytics**	✓	✓	✓	**ScottKnott Wilcoxon**	✓
Clustering	DIC (Tsymbal et al. 2008)	Combines historical models output with new model output • Dynamic Selection (DS): model with the best local performance is selected • Dynamic Voting (DV): combines models output using majority voting • DV with Selection (DVS): half of the models are selected using DS and then combined using DV	Synthetic (Numeric), Antibiotic Resistance in Patients	✓	✓			✓
	two-step (Cruz et al. 2022)	A two-step approach for model selection that involves choosing the most suitable model from the model pool	Micro-milling of aerospace components					
	Our work	**See Section 4.5**	**Software Analytics**	✓	✓			✓

8 Conclusion

In summary, this study challenges the conventional practice of discarding older Machine Learning (ML) models once newer ones are deployed. It highlights the substantial value that older models still hold compared to freshly trained RFS counterparts. This paper introduces novel post-deployment model recycling techniques, offering a more informed approach to deciding which old models to reuse and when to do so.

Through an empirical investigation across eight long-lived Apache projects, encompassing a substantial number of commits, this study analyzes the performance of five model recycling strategies on three different types of models in the domain of Just-In-Time defect prediction, a popular subdomain of software analytics. Our findings indicate that our approach significantly outperforms the traditional RFS approach in terms of recall, g-mean, AUC and F1 performance metrics. Our best recycling strategy (Model Stacking) outperforms the baseline in over 50% of the projects across all models. These results may differ based on the choice of the project and recycling strategy.

While these recycling strategies showcase improved model performance, it is important to consider that they come at the cost of negligible to low effect size degradation in precision and increased time-to-inference compared to RFS. Nevertheless, this research underscores the potential of thoughtfully retaining and reusing older models, enhancing the effectiveness of ML models in real-world settings.

9 Conflict of Interests

All authors declare that they have no conflicts of interest.

10 Data Availability Statement

The replication package for this project which contains the code and data used can be found here Patel (2023).

References

Brzezinski, D., Stefanowski, J.: Reacting to different types of concept drift: The accuracy updated ensemble algorithm. IEEE transactions on neural networks and learning systems **25**(1), 81–94 (2013)

Cabral, G.G., Minku, L.L.: Towards reliable online just-in-time software defect prediction. IEEE Transactions on Software Engineering **49**(3), 1342–1358 (2022)

Cabral, G.G., Minku, L.L., Shihab, E., Mujahid, S.: Class imbalance evolution and verification latency in just-in-time software defect prediction. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 666–676. IEEE (2019)

Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: synthetic minority oversampling technique. Journal of artificial intelligence research **16**, 321–357 (2002)

Chen, L., Zaharia, M., Zou, J.: Frugalgpt: How to use large language models while reducing cost and improving performance. arXiv preprint arXiv:2305.05176 (2023)

Chen, Z., Liu, B.: Lifelong machine learning, vol. 1. Springer (2018)

Cruz, Y.J., Rivas, M., Quiza, R., Haber, R.E., Castaño, F., Villalonga, A.: A two-step machine learning approach for dynamic model selection: A case study on a micro milling process. Computers in Industry **143**, 103,764 (2022)

De Lange, M., Aljundi, R., Masana, M., Parisot, S., Jia, X., Leonardis, A., Slabaugh, G., Tuytelaars, T.: A continual learning survey: Defying forgetting in classification tasks. IEEE transactions on pattern analysis and machine intelligence **44**(7), 3366–3385 (2021)

Diethe, T., Borchert, T., Thereska, E., Balle, B., Lawrence, N.: Continual learning in practice. arXiv preprint arXiv:1903.05202 (2019)

Ekanayake, J., Tappolet, J., Gall, H.C., Bernstein, A.: Tracking concept drift of software projects using defect prediction quality. In: 2009 6th IEEE International Working Conference on Mining Software Repositories, pp. 51–60. IEEE (2009)

Elwell, R., Polikar, R.: Incremental learning of concept drift in nonstationary environments. IEEE Transactions on Neural Networks **22**(10), 1517–1531 (2011)

Falessi, D., Ahluwalia, A., Penta, M.D.: The impact of dormant defects on defect prediction: A study of 19 apache projects. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(1), 1–26 (2021)

Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, pp. 313–324 (2014)

Forman, G.: Tackling concept drift by temporal inductive transfer. In: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, pp. 252–259 (2006)

Gao, S., Zhang, H., Gao, C., Wang, C.: Keeping pace with ever-increasing data: Towards continual learning of code intelligence models. arXiv preprint arXiv:2302.03482 (2023)

Herraiz, I., Rodriguez, D., Robles, G., Gonzalez-Barahona, J.M.: The evolution of the laws of software evolution: A discussion based on a systematic literature review. ACM Computing Surveys (CSUR) **46**(2), 1–28 (2013)

Hess, M.R., Kromrey, J.D.: Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In: annual meeting of the American Educational Research Association, vol. 1. Citeseer (2004)

Hoang, T., Dam, H.K., Kamei, Y., Lo, D., Ubayashi, N.: Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 34–45. IEEE (2019)

James, G., Witten, D., Hastie, T., Tibshirani, R., et al.: An introduction to statistical learning, vol. 112. Springer (2013)

Jiang, T., Tan, L., Kim, S.: Personalized defect prediction. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 279–289. Ieee (2013)

Kamei, Y., Fukushima, T., McIntosh, S., Yamashita, K., Ubayashi, N., Hassan, A.E.: Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering **21**, 2072–2106 (2016)

Kamei, Y., Shihab, E., Adams, B., Hassan, A.E., Mockus, A., Sinha, A., Ubayashi, N.: A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering **39**(6), 757–773 (2012)

Keshavarz, H., Nagappan, M.: Apachejit: a large dataset for just-in-time defect prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories, pp. 191–195 (2022)

Kim, S., Whitehead, E.J., Zhang, Y.: Classifying software changes: Clean or buggy? IEEE Transactions on software engineering **34**(2), 181–196 (2008)

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W.: Investigating code review quality: Do people and participation matter? In: 2015 IEEE international conference on software maintenance and evolution (ICSME), pp. 111–120. IEEE (2015)

Kubat, M., Matwin, S., et al.: Addressing the curse of imbalanced training sets: one-sided selection. In: Icml, vol. 97, p. 179. Citeseer (1997)

Lemaître, G., Nogueira, F., Aridas, C.K.: Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. Journal of Machine Learning Research **18**(17), 1–5 (2017). URL `http://jmlr.org/papers/v18/16-365`

Lundberg, S.M., Lee, S.I.: A unified approach to interpreting model predictions. Advances in neural information processing systems **30** (2017)

Macbeth, G., Razumiejczyk, E., Ledesma, R.D.: Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica **10**(2), 545–555 (2011)

McIntosh, S., Kamei, Y.: Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. In: Proceedings of the 40th International Conference on Software Engineering, pp. 560–560 (2018)

Mockus, A., Weiss, D.M.: Predicting risk of software changes. Bell Labs Technical Journal **5**(2), 169–180 (2000)

Olewicki, D., Habchi, S., Nayrolles, M., Faramarzi, M., Chandar, S., Adams, B.: Towards lifelong learning for software analytics models: Empirical study on brown build and risk prediction. arXiv preprint arXiv:2305.09824 (2023)

Olewicki, D., Nayrolles, M., Adams, B.: Towards language-independent brown build detection. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2177–2188 (2022)

Paleyes, A., Urma, R.G., Lawrence, N.D.: Challenges in deploying machine learning: a survey of case studies. ACM Computing Surveys **55**(6), 1–29 (2022)

Patel, H.: Post-deployment model recycling. `https://github.com/SAILResearch/replication-23-harsh-model_recycling` (2023)

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)

Polikar, R., Upda, L., Upda, S.S., Honavar, V.: Learn++: An incremental learning algorithm for supervised neural networks. IEEE transactions on systems, man, and cybernetics, part C (applications and reviews) **31**(4), 497–508 (2001)

Pornprasit, C., Tantithamthavorn, C.K.: Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 369–379. IEEE (2021)

Quinonero-Candela, J., Sugiyama, M., Schwaighofer, A., Lawrence, N.D.: Dataset shift in machine learning. Mit Press (2008)

Rajbahadur, G.K., Wang, S., Oliva, G.A., Kamei, Y., Hassan, A.E.: The impact of feature importance methods on the interpretation of defect classifiers. IEEE Transactions on Software Engineering **48**(7), 2245–2261 (2021)

Raschka, S.: Mlxtend: Providing machine learning and data science utilities and extensions to python's scientific computing stack. Journal of open source software **3**(24), 638 (2018)

Schelter, S., Biessmann, F., Januschowski, T., Salinas, D., Seufert, S., Szarvas, G.: On challenges in machine learning model management (2015)

Song, L., Minku, L., Yao, X.: On the validity of retrospective predictive performance evaluation procedures in just-in-time software defect prediction. Empirical Software Engineering (2023)

Street, W.N., Kim, Y.: A streaming ensemble algorithm (sea) for large-scale classification. In: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 377–382 (2001)

Strubell, E., Ganesh, A., McCallum, A.: Energy and policy considerations for deep learning in nlp. arXiv preprint arXiv:1906.02243 (2019)

Sun, Y., Tang, K., Zhu, Z., Yao, X.: Concept drift adaptation by exploiting historical knowledge. IEEE transactions on neural networks and learning systems **29**(10), 4822–4832 (2018)

Tabassum, S., Minku, L.L., Feng, D.: Cross-project online just-in-time software defect prediction. IEEE Transactions on Software Engineering **49**(1), 268–287 (2022)

Tan, M., Tan, L., Dara, S., Mayeux, C.: Online defect prediction for imbalanced data. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 99–108. IEEE (2015)

Tantithamthavorn, C., McIntosh, S., Hassan, A.E., Matsumoto, K.: The impact of automated parameter optimization on defect prediction models. IEEE Transactions on Software Engineering **45**(7), 683–711 (2018)

Tsymbal, A., Pechenizkiy, M., Cunningham, P., Puuronen, S.: Dynamic integration of classifiers for handling concept drift. Information fusion **9**(1), 56–68 (2008)

Weisstein, E.W.: Bonferroni correction. https://mathworld. wolfram. com/ (2004)

Wolpert, D.H.: Stacked generalization. Neural networks **5**(2), 241–259 (1992)

Woolson, R.F.: Wilcoxon signed-rank test. Wiley encyclopedia of clinical trials pp. 1–3 (2007)

Zeng, Z., Zhang, Y., Zhang, H., Zhang, L.: Deep just-in-time defect prediction: how far are we? In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 427–438 (2021)

Zhao, Y., Damevski, K., Chen, H.: A systematic survey of just-in-time software defect prediction. ACM Computing Surveys **55**(10), 1–35 (2023)