Outlier Mining Techniques for Software Defect Prediction

Tim Cech[1][0000−0001−8688−2419], Daniel Atzberger[1],
Willy Scheibel[1][0000−0002−7885−9857], Sanjay Misra[2][0000−0002−3556−9331], and
Jürgen Döllner[1]

[1] Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam,
Potsdam, Germany
{tim.cech,daniel.atzberger,willy.scheibel,juergen.doellner}@hpi.uni-potsdam.de,
[2] Department of Computer Science and Communication, Østfold University College,
Halden, Norway sanjay.misra@hiof.no

Abstract. Using software metrics as a method of quantification of software, various approaches were proposed for locating defect-prone source code units within software projects. Most of these approaches rely on supervised learning algorithms, which require labeled data for adjusting their parameters during the learning phase. Usually, such labeled training data is not available. Unsupervised algorithms do not require training data and can therefore help to overcome this limitation.

In this work, we evaluate the effect of unsupervised learning by means of cluster-based algorithms and outlier mining algorithms for the task of defect prediction, i.e., locating defect-prone source code units. We investigate the effect of various class balancing and feature compressing techniques as preprocessing steps and show how sliding windows can be used to capture time series of source code metrics. We evaluate the Isolation Forest and Local Outlier Factor, as representants of outlier mining techniques. Our experiments on three publicly available datasets, containing a total of 11 software projects, indicate that the consideration of time series can improve static examinations by up to 3%. The results further show that supervised algorithms can outperform unsupervised approaches on all projects. Among all unsupervised approaches, the Isolation Forest achieves the best accuracy on 10 out of 11 projects.

Keywords: Software Defect Prediction, Unsupervised Learning, Outlier Mining

1 Introduction

Software defects reduce the added value for the customer or lead to an increased effort in development if they have to be corrected later in development [31]. Therefore, the timely detection of defective source code units is a central theme of code quality. Classically, *unit tests* and *integration tests* are used for the early detection of defective code units by testing the respective units for their functionality [32]. In practice, however, it is often untenable to test all functionalities, or the potential for defects is not yet known to the developers at the time the software is developed. Software metrics, which describe various aspects of the complexity and quality of the source code, can be used complementary to monitoring the development of large software systems. A distinction is made between static code metrics and process metrics. Static code metrics, such as *Lines of Code* (LOC), *McCabe Complexity* (MCCC), or *Nesting Level* (NL), measure aspects of a software project at a specific point in time, i.e., a source code revision. In contrast, process metrics describe the change between two revisions, e.g., the number of developers involved in a commit or the number of changed LOC [42]. Based on those metrics, statistical analyses can be applied to locate defect-prone source code units [21].

Various *Machine Learning* (ML) techniques have been used for detecting defect-prone source code units using software metrics. Most approaches use supervised ML techniques on static code metrics of a single revision [31,21]. Supervised approaches require a labeled training dataset, which is challenging to obtain, since usually no records of the defect history that can be used for labeling are kept [3,29]. In this case, another approach is desirable, that does not require the project to have a history of labeled defects. We focus on the usage of unsupervised training techniques, which were investigated recently to overcome this issue [37,39,27]. Previous research with unsupervised methods indicates that even basic approaches provide acceptable results [37]. However, in general, they achieve weaker results compared to supervised techniques [16]. Recent results by Moshtari et al. show that the distribution of defects within a software project follows the Pareto principle, i.e., a small fraction of the source code units contain a large part of the defects [27]. Furthermore, defective source code units are also distinct from non-defective code units in terms of metrics and can therefore be treated as outliers [27].

Motivated by this result, we extend the work of Moshtari et al. to time series of metrics by comparing unsupervised learning algorithms by means of cluster-based algorithms and outlier mining algorithms with basic supervised approaches. For it, we want to study if any approach is superior to others. Subsequently, we will not address questions about the nature of defects but inherit their definitions from the creators of each dataset. We describe each source code unit by a sequence of software metrics using sliding windows and a subsequent feature compression technique. We then apply the respective defect prediction algorithm to locate defect-prone source code units. In summary, we make the following contributions:

1. We present a computational experiment comparing basic supervised, cluster-based techniques and outlier techniques addressing common pitfalls by Feature Compression and Class Balancing.
2. We present the use of sliding windows for modeling a time series of software metrics.
3. We introduce the application of the Isolation Forest and Local Outlier Factor, as examples of outlier mining algorithms, for the task of locating defect-prone source code units.

4. We present an evaluation pipeline for comparing unsupervised and supervised defect prediction algorithms, and conduct experiments on three publicly available datasets for a total of 11 software projects.

The remainder of this work is structured as follows: Sec. 2 studies related work on defect prediction techniques. In Sec. 3, we elaborate on our approach. The computational experiment to investigate the research questions above are detailed in Sec. 4 and the results are presented in the Sec. 5. We conclude with a discussion and possible threats to validity in Sec. 6 and suggest conclusions and future work in Sec. 7.

2 Related Work

Supervised and unsupervised approaches can be distinguished in the field of ML. We focus on unsupervised software defect prediction, and thus, study the related work for unsupervised models in more detail. In contrast, we only give a high-level view of the field of supervised defect prediction. Additionally, many preprocessing methods were already evaluated in the literature, showing that the choice of the combination of the most adequate preprocessing method and model is non-trivial [4,35].

Unsupervised Approaches. Yang et al. were one of the first who applied unsupervised learning techniques for effort-aware defect prediction [39]. Effort-aware defect prediction describes the attempt to predict defects while taking into account the effort required to check them. Based on their results, the authors suggest that (basic) unsupervised models can outperform supervised models in terms of recall when taking the file size, and therefore the effort of units that need to be reviewed, into account. In subsequent research, this effect was put into perspective by Fu and Menzies as well as Huang et al. [12,16]. Before, Nam and Kim already argued that unsupervised approaches can be beneficial because they do not need historical training data, which can be hard to obtain for defect prediction [29].

Subsequently, several studies were conducted exploring different, more complex unsupervised approaches for defect prediction. Albahli combines several (also unsupervised) models resulting in an accuracy of 81% on a dataset containing seven open-source projects [2]. Xu et al. reviewed 40 cluster-based approaches and observed that these achieve similar results to typical supervised approaches [37]. Zhang et al. also found that unsupervised models are competitive to supervised models that were trained on another project [40].

Studies on 16 software projects by Moshtari et al. showed that the distribution of defects over a software project follows the Pareto principle, i.e., a large part of the defects is contained in a small part of the units [27]. Assuming that the defective units also differ significantly from the majority concerning their metrics, the localization of defects can be considered an outlier problem. Moshtari et al. investigated the use of five proximity-based outlier mining techniques. The best results were achieved using the *k nearest neighbor* (kNN) algorithm.

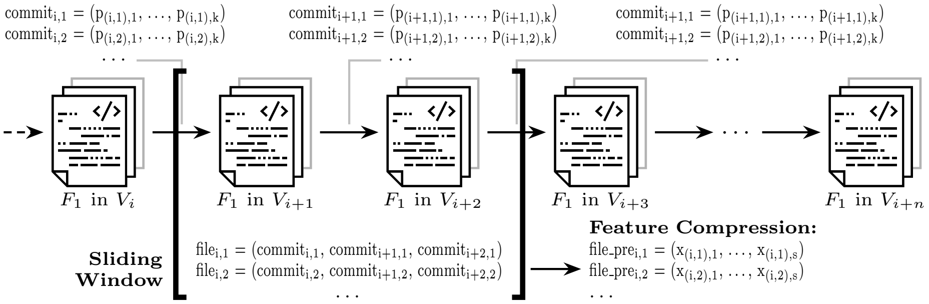

$\mathrm{commit}_{i,1} = (p_{(i,1),1}, \ldots, p_{(i,1),k})$
$\mathrm{commit}_{i,2} = (p_{(i,2),1}, \ldots, p_{(i,2),k})$
...
$\mathrm{commit}_{i+1,1} = (p_{(i+1,1),1}, \ldots, p_{(i+1,1),k})$
$\mathrm{commit}_{i+1,2} = (p_{(i+1,2),1}, \ldots, p_{(i+1,2),k})$
...
$\mathrm{commit}_{i+1,1} = (p_{(i+1,1),1}, \ldots, p_{(i+1,1),k})$
$\mathrm{commit}_{i+1,2} = (p_{(i+1,2),1}, \ldots, p_{(i+1,2),k})$
...
F_1 in V_i
F_1 in V_{i+1}
F_1 in V_{i+2}
F_1 in V_{i+3}
...
F_1 in V_{i+n}
Sliding Window
$\mathrm{file}_{i,1} = (\mathrm{commit}_{i,1}, \mathrm{commit}_{i+1,1}, \mathrm{commit}_{i+2,1})$
$\mathrm{file}_{i,2} = (\mathrm{commit}_{i,2}, \mathrm{commit}_{i+1,2}, \mathrm{commit}_{i+2,2})$
...
Feature Compression:
$\mathrm{file\_pre}_{i,1} = (x_{(i,1),1}, \ldots, x_{(i,1),s})$
$\mathrm{file\_pre}_{i,2} = (x_{(i,2),1}, \ldots, x_{(i,2),s})$
...

Fig. 1. An example visualization for our preprocessing pipeline on process metrics. The process metrics that describe the change between revision i and $i+1$ in File_j are given by $\mathrm{commit}_{\mathrm{i,j}} = (p_{(i,j),1}, \ldots, (p_{(i,j),k})$. The process metrics of three commits of File F_j are collected in the vector $\mathrm{file}_{\mathrm{i,j}} = (\mathrm{commit}_{\mathrm{i,j}}, \mathrm{commit}_{\mathrm{i+1,j}}, \mathrm{commit}_{\mathrm{i+2,j}})$. The compressed vector describing F_j is given by $\mathrm{file\_pre}_{\mathrm{i,j}} = (x_{(i,j),1}, \ldots, x_{(i,j),s})$.

Supervised Approaches. Several literature reviews were conducted, e.g., by Rathore et al., Li et al., and Wahono, summarizing various supervised approaches in different use-cases with different metrics and models [31,21]. Liu et al. already considered a time series of code and process metrics for defect prediction [23]. By training a *Recurrent Neural Network* (RNN) on nine projects of the PROMISE dataset [34], the new model was able to outperform basic models using code and process metrics in terms of cost-effectiveness, the Schott-Knott-Test, and Win/Tie/Loss. Another recently studied approach is the usage of *Cross Project Defect Prediction* (CPDP) [3]. Here, the defect prediction model is trained on another project, which has labeled data available. After the completion of the training phase, the model tries to predict defects in the target project. CPDP has only achieved limited success [42,3]. Yan et al. discovered that for a certain dataset, unsupervised within-project defect prediction outperforms supervised CPDP models [38].

Data Preprocessing. Several studies were conducted concerning the general preprocessing steps required before creating the actual defect prediction model. Mende provides a general overview of several pitfalls and aspects that should be considered when constructing a model for defect prediction [25]. In this work, we focus on a selected subset of those aspects, namely feature compression techniques for all models and balancing techniques for supervised models. Kondo et al. suggest that supervised models profit most from techniques that filter metrics, therefore preserving the original meaning [19]. In contrast, unsupervised models profited most from synthetic methods that combine several original metrics in a new synthetic one. Zhu et al. support the claim that Autoencoder can be a useful feature compression technique [41]. Another aspect of preprocessing is concerned with the question of how to handle imbalanced datasets, which are typical for defect prediction tasks [27]. Tantithamthavorn et al. investigated that the choice of the balancing technique is non-trivial and should be considered carefully [35].

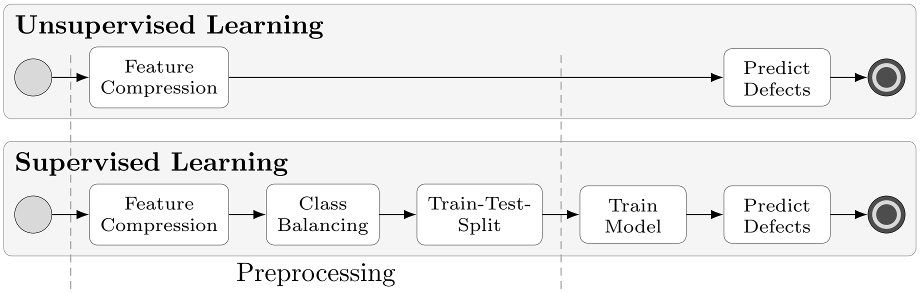


Fig. 2. Comparing the workflow for defect prediction with an unsupervised vs. supervised model. The workflow with an unsupervised model is much simpler because the imbalance of the dataset, overfitting, and the requirement of labels are not an issue.

This result is supported by Mahmood et al. who also investigated the effect of imbalance on the performance of supervised defect predictors [24].

3 Data Preprocessing and Modeling

In this section, we detail our approach for locating defect-prone source code units by using unsupervised learning algorithms on time series of source code metrics. Assuming that defects differ significantly in their metrics from non-defective samples, we adopt the idea of Moshtari et al. and apply outlier techniques for locating defect-prone source code units [27]. For this purpose, we additionally adopt an idea from Ding et al., who demonstrated in another use case that a *sliding window* can be used to capture an evolution of metrics to improve the quality of outlier mining [10]. This results in the basic workflow shown in Fig. 1.

Sliding Windows. Sliding windows are a technique known from stream data processing to analyze data-intensive data streams [9], i.e., several data samples are combined and viewed coherently [10]. For example, instead of just investigating metrics between two commits, all the changes in two consecutive commits are gathered into one feature vector. If, for example, five different process metrics are considered and the *Window Size* (WS) is three, then each file or sample is represented by a 15-dimensional vector. As this results in a high-dimensional vector, we apply feature compression techniques to reduce the size of the feature vector to overcome the *Curse of Dimensionality* [36].

Processing Pipeline. After gathering the data in an high-dimensional feature space using sliding windows, we preprocess the data to prepare them for prediction. Fig. 2 compares the different preprocessing and prediction pipelines for supervised and unsupervised learning. Both pipelines are including the task of feature compression. Afterwards, as Bennin et al. showed for supervised techniques, the balancing of the defect and non-faulty class is desirable [4]. Since unsupervised techniques do not inherently distinguish between faulty and non-faulty samples, this preprocessing step is only applicable to supervised learning. Furthermore, supervised learning distinguishes between a training phase and a test phase, which use separate datasets. A train-test split is required to avoid overfitting [13]. Finally, all models are evaluated using the test set for supervised models and the whole dataset for unsupervised models.

We investigate a selection of basic supervised and unsupervised models. We deemed those models suitable because they are used in several studies or more complex models built upon them [21,37].

Feature Compression. The gathered data suffers from high dimensionality. Many models produce worse results on a high-dimensional space, as the number of samples required to generalize grows exponentially with the number of features. This effect is denoted as the Curse of Dimensionality [36]. To avoid this, an additional preprocessing step transfers the data samples to a lower-dimensional space as shown in Fig. 1. In addition, Jiarpakdee et al. also suggest that even if the feature vector in and of itself is not already very high-dimensional, it may still be worthwhile to remove features [17]. Especially removing high correlations of individual variables from the data is usually desirable. Substantial differences can be observed between supervised and unsupervised learning techniques regarding Feature Compression [19]. For example, supervised learning predictors favor those techniques that preserve the original context of the metrics. This is also desirable in terms of the interpretability of the model. Unsupervised learning methods would benefit in particular from those techniques that construct new synthetic features from the given features. In this work, we focus on the following feature selection and synthesization techniques: The *Variance Inflation Factor* (VIF) [26], *Autoencoders* (AE) [19,41], and *Feature agglomeration* (FA). FA refers to a technique to filter correlated variables by repeated clustering. Metrics are merged if they are highly correlated leading to a more dense clustering.

Class Balancing. For supervised learning techniques, it can be beneficial to balance between the defect (faulty) class and a non-defect (non-faulty) class in the training dataset [4]. This is especially true since the defect class usually follows the Pareto principle [27]. Tantithamthavorn et al. found that the choice of a balancing technique has a significant impact onto the classification result, thus different upsampling techniques may lead to different results [35]. We considered three different upsampling techniques: *Synthetic Minority Over-sampling Technique* (SMOTE) [8], the *MAHAKIL* algorithm [5] and an *Euclidean Noise* (EN) technique. EN is a quick-to-execute naive technique. New samples are generated by offsetting each sample of the defect class with a noise signal drawn from a normal distribution.

Supervised Learning Techniques. Supervised learning is divided in a training and a test phase. In the training phase, labeled data is used for learning an abstraction of the data, which is subsequently used for predicting unseen data. We consider the following supervised learning techniques: *Random Forest* (RF), *Support Vector Machines* (SVM), *Logistic Regression* (LR), *Naive Bayes* (NB) model, and *Multi-Layer Perceptron* (MLP) [33]. In addition, we combine the

Table 1. An overview about the investigated datasets. LOC refers to the lines of code metric, P means project, and FPP means files per project. The Unified GitHub dataset and Jureczko only report on versions of projects with no specific time-frame stated.

Dataset	# P	# FPP	Time Frame	Type Metrics	jit Metric
Change Burst	1	6 728	One week	Process	Changed LOC
Unified GitHub	10	≈ 1,000	Sporadic	Static & Process	LOC
Jureczko	13	max. 250	Sporadic	Static	LOC

models above to an *Ensemble* (ES) with majority voting as the final decision rule.

Unsupervised Learning Techniques. In contrast to supervised learning, unsupervised learning does not distinguish between a training and test phase, because no labels are involved when applying them to data. Since labels are not used for unsupervised learning, we assigned which detected structure in our dataset is identified with the faulty class and the non-faulty class. We identify the smaller structure (fewer samples belong to the structure) as the cluster of the faulty class, since the number of defects is usually smaller [27]. We only had to make an exception for the *mcMMO* project, where the situation is reversed.

We consider two different unsupervised learning approaches. First, we use cluster-based techniques, which were previously already investigated e.g. by Xu et al. [37] or Li et al. [20]. Second, we additionally use the property found by Moshtari et al. that the metrics of defective source code units often have exceptional values [27]. Therefore outlier mining techniques are applicable.

Cluster-based techniques create clusters according to a criterion defined by the model, e.g., the density or similarity of sample regions. In contrast to Moshtari et al., we only investigate cluster-based techniques that allow us to set the number of clusters to the number of our target classes (defect and non-defect) [27]. Specifically, we study the following cluster-based techniques: The *k-Means* (kM) algorithm with $k = 2$, because our target has two classes [37] and the MeanShift (MS) algorithm with orphans [14]. Orphans are samples that do not belong to any density structure or would significantly change the density structure of the detected clusters if they were forcibly assigned to a cluster [14]. We identify the orphans as faulty samples and the remaining structures as non-faulty samples.

In contrast, outlier mining describes another set of techniques that focus on the process of detecting conspicuous data samples, i.e., to identify anomalies in the given dataset. Outliers are characterized as samples that are significant dissimilar to the majority of samples. So, outlier mining rather investigates dissimilarities in the dataset instead of similarities like cluster-based techniques. We investigate the following outlier mining techniques: The *Local Outlier Factor* (LOF) and an *Isolation Forest* (IF) [22].

4 Computational Experiment

We evaluate our basic models with the pipeline described in Sec. 3 on the *Change Burst* dataset [28], the *Unified GitHub* dataset [11], and the *Jureczko* dataset [18]. Our data pipeline is based on *scikit-learn*[3] and *Tensorflow with Keras*[4] library in Python. Those libraries are implementing all our investigated basic models as well as our preprocessing techniques, excluded the SMOTE and MAHAKIL algorithm for which the implementation is given in the auxiliary material. We evaluate our results in terms of precision, recall, F_1-score, accuracy as well as a *Just-in-time-accuracy* (JIT-accuracy). For consistency, we set the random seed to 42 to make our results deterministic and repeatable. For supervised models, we used a stratified train-test-split: 60% of the data was used for training and the remaining 40% for testing. From now on, we refer to a model as a specific combination of a basic supervised or unsupervised such as RF, with a feature compression—for all basic models—and a class balancing technique —for supervised models only. According to the no-free-lunch-theorem the outcome of a classification result is generally data dependent, so the evaluation on more than one project and set of metrics is necessary [1]. Therefore, for a reliable evaluation, we used the three previously mentioned publicly available datasets. For each dataset, one file of a project is considered as one sample (file-level).

We summarized our datasets in Table 1. In detail, we used the following datasets:

- The Change Burst dataset contains process metrics, hence change rates between two commits for the *Eclipse* project [28]. A so called *Burst* represents the changes during a week. We have chosen the first *Gap* containing 10 bursts, since only few defects are fixed or newly introduced into the code, providing a stample number of samples for each class (faulty and non-faulty).
- The Unified GitHub dataset captures a wide variety of both static and process-oriented metrics for more than ten projects [11]. In contrast to the Change Burst dataset, the reported revisions are further apart in time (for details, cf. auxiliary material), therefore a number of files are created or deleted between two revisions. Therefore, the number of faulty and non-faulty samples is more volatile. We will only examine projects which report at least two defective modules over all revisions, because with no or only one defective module the SMOTE and MAHAKIL technique is not applicable. Also, it is impossible to learn any abstraction from exactly one example.
- Lastly, the Jureczko dataset offers a small collection of metrics for a larger number of projects [18].

The Change Burst dataset allows the most stable evaluation, because the number of files is constant and also the number of faulty and non-faulty samples is comparably constant. We will also test our findings on the Unified GitHub dataset. The Jureczko dataset has the most disadvantageous characteristics to

[3] scikit-learn.org

[4] tensorflow.org and keras.io

allow a meaningful evaluation because it suffers from the same problems as the Unified GitHub dataset, i.e., the sample size total and per class is changing often and only reports on a small number of metrics. We therefore provide the results for the Jureczko dataset only in the auxiliary material for additional validation of our findings.

Quality Measures. Bowes et al. argued that such metrics should be used from which the original confusion matrix can be derived easily [7]. Therefore, we choose precision and recall for both classes as well as the weighted average. Additionally, we captured the F_1-scores for the positive and negative class. Furthermore, we captured two kinds of accuracy metrics. The classic accuracy is defined as the ratio between the sum of true positives and true negatives divided by the number of samples.

In addition, we define a JIT-accuracy (acc_{jit}). Let S be the set of all samples. Then, let $TP \subseteq S$ denote the set of all true positives, let $TN \subseteq S$ denote the set of all true negatives and let $f(s)$ be a mapping from the sample to the reciprocal normalized lines of (changed) code of the sample *((Changed) LOC)*, then the JIT-accuracy is given by:

$$acc_{jit} = \frac{\sum_{s \in TP \cup TN} s \cdot f(s)}{\sum_{t \in S} t \cdot f(t)}$$

This is to favor classifications that identify small faulty samples correctly and penalize results that incorrectly suggest a particularly large change for review. By determining precision, recall, and F_1-score for each of the classes, we show how well the predictor can handle the individual classes, and to that extent we circumvent the criticism of these values that is raised by Powers [30] or Hemmati et al. [15], for example.

Optimization of Hyperparameters. Unlike unsupervised models, supervised models usually need to be optimized during training time since they use a set of hyperparamaters [6]. For optimization, we use a random search with 50 iterations and five stratified validation folds [6]. We had to make an exception for SVMs, since their training time can be exhaustive depending on the choice of hyperparameters, therefore they are only optimized with five iterations and three folds. We have reported the full list of optimized attributes in our auxiliary material.

Additionally, we have to determine a WS for gathering the data (cf. Sec. 3). We used a *Grid Search* for optimization of this hyperparameter with the integer values 1-10 [6]. A WS of 1 means, that the model does not profit from the windowing aspect of our approach. We gathered data with according WS of 1-10 and let the models predict the defects in the next revision. We have chosen 10 as an upper bound since a burst in the ChangeBurst dataset contains ten revisions and no project in the Unified GitHub dataset report metrics for more than than ten revisions for gathering and one for testing for a total of eleven revisions. In case of WS=10 for the ChangeBurst dataset, the target variable is derived from the first revision of the second burst. Since a significant number of files is deleted

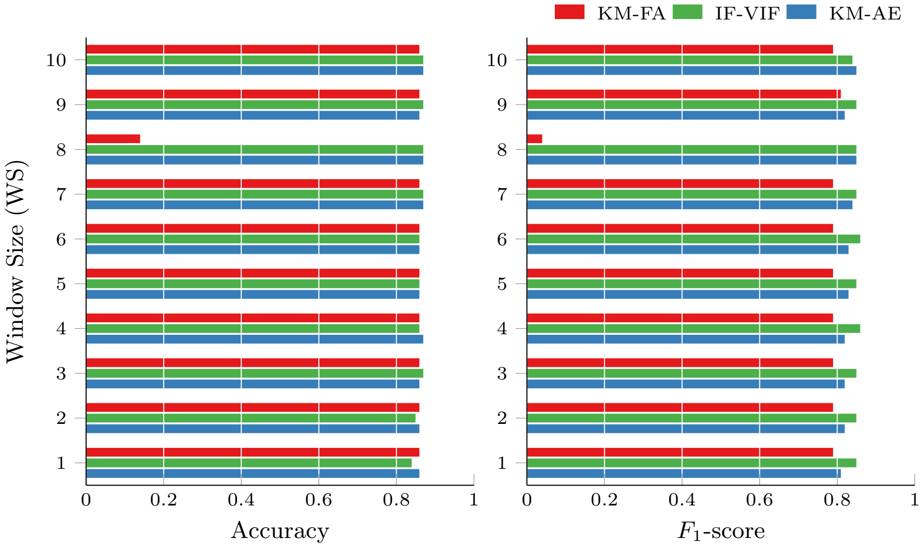


Fig. 3. Comparing WS by accuracy and F_1-score for the Eclipse project. The usage of a sliding window can improve the quality of unsupervised models by 1-3%. The positive effect is already visible if we use WS=2.

and created between revisions for projects from the Unified GitHub dataset, we only evaluated on files that are present in all revisions. Since the Change Burst dataset captures the metrics more timely, only few files are created or deleted between revisions. The number of samples overall and per-class is more stable.

5 Results

We collect results regarding three questions. First we consider which influence the choice of the WS has on the performance of our models. Regarding the second question, we examine which supervised or unsupervised model performs best for multiple projects. We investigate the ten projects from the Unified GitHub dataset and one burst from the Change Burst dataset for Eclipse for a total of 11 projects. The third question compares the performance of supervised and unsupervised models. Again, we examine our 11 projects and the Eclipse project in more detail.

In general, we only consider results, if they can exceed a F_1-score of 10% in the faulty class, because models that show worse results do not provide an useful abstraction for finding defects, which is the main task in this paper. For completeness, however, those results are included in the auxiliary material.

5.1 Choice of the Window Size

We can observe two kinds of behavior when varying the WS of our sliding window. On the one hand, Fig. 3 shows the standard case that we observed for our (here

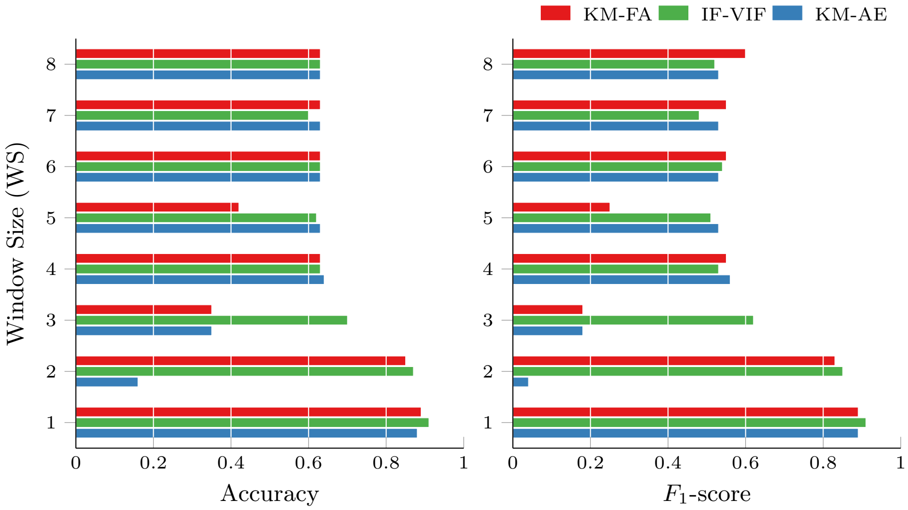


Fig. 4. For the hazelcast project the usage of a sliding window decreases the quality of unsupervised models. For WS=2 the effect is comparatively small, but the loss in quality increases for larger WS.

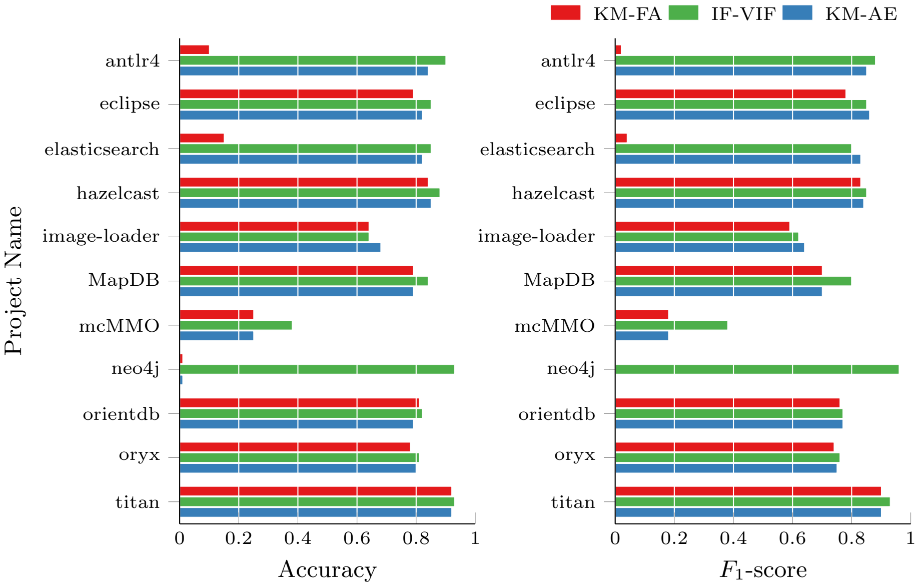


Fig. 5. The IF with VIF shows the best or one of the best accuracy for 10 out of 11 software projects and best F_1-scores for 8 out of 11 software projects. k-Means is better for the Android-Universal-Image-Loader (image-loader).

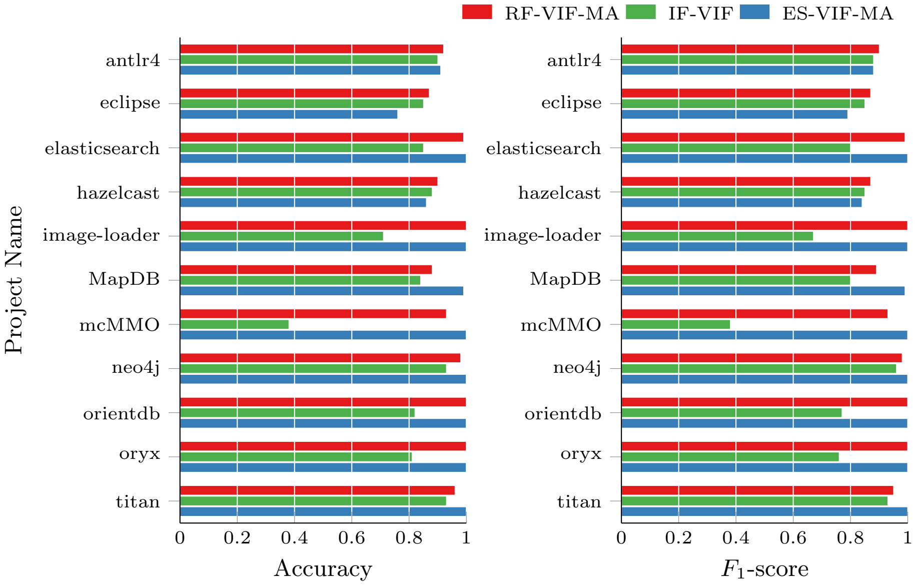


Fig. 6. The ensemble of all single supervised models (majority voting) with VIF and MA upsampling outperforms the IF in 11 out of 11 cases. Also a single model, e.g., the optimized RF with VIF and MA outperforms the best unsupervised model in 8 out of 11 cases.

unsupervised) models. The models are showing a better accuracy and overall weighted F_1-score for a WS greater 1. For WS 2 to 5 the result is similar. For greater WS, the performance do not change or became worse. On the other hand, some models have shown a different behavior on a different project. In this case, the usage of a sliding window decreased the accuracy and F_1-score. This effect is most significant for the *hazelcast* project as shown in Fig. 4. We can not test as many WS for hazelcast as for the Eclipse project, since the Unified GitHub Dataset only reports metrics for nine revisions and we require at least one revision for testing. The quality measures are decreasing for larger WS but the effect is comparably small for WS 2 and increases for larger WS. Thus, a WS greater than 1 is not beneficial for all combinations of model and project. Overall 612 (96 unsupervised and 516 supervised) different project-model combinations[5] benefit from a WS larger than 1, additional 206 (37 unsupervised and 169 supervised) neither benefit nor take a loss, and only 414 (43 unsupervised and 371 supervised) report a decrease in the macro averaged F_1-score, which additionally take into account the size of both classes, so that not already a change in the distribution between the size of the faulty and non-faulty class can influence our result. Overall, approximated 33% of the model-project combinations are not improved by our

[5] For completeness, here, we also evaluated the possibility to use no Balancing or no Feature Compression technique. Those results are—as expected—weaker (cf. auxiliary material).

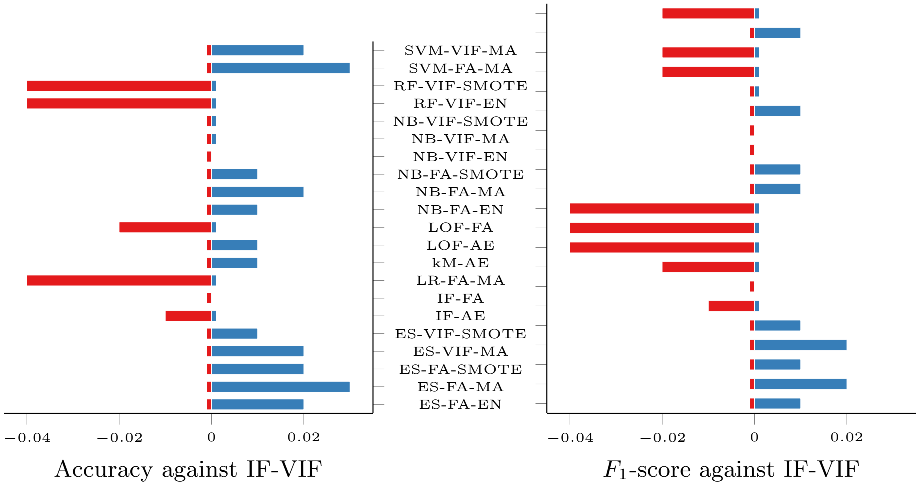


Fig. 7. Ten supervised models outperform the reference model IF-VIF (which scored 85% in all metrics) for the Eclipse project in terms of accuracy and F_1-score, but already five of them are variations of our ensemble model. Otherwise, only the NB model and some variations of SVMs are better models. Models, that are at least 3% worse in both measures than the IF-VIF model are omitted.

windowing technique, 50% do profit and about 17% are indifferent to the usage of a sliding window. Further analyses of our results show that the size of the last group shrinks with larger WS and more models either profit or not profit from the use of sliding windows. Also, as stated before, the quality decreases fast for larger WS, while the increase does not grow as fast as the decrease.

This effect can be caused by different aspects of the different datasets. Most of the models reporting a decrease in accuracy or F_1-score are evaluated with a project from the Unified GitHub dataset. For those projects, the sample size is very small and the share of faulty samples comparably great, since we can only evaluate on those files that are present in all revisions for consistency reasons, making the result dependent on only few samples. Also, in accordance with the no-free-lunch-theorem, a larger WS may simply be not suitable for the concrete model or set of metrics (e.g., static code metrics) [1].

It is desirable to evaluate models with a constant WS for comparability. We set our WS to 2, since—for this value—the decrease for the minority of models is relatively small, while the increase for the majority of models is almost as good as possible. In contrast, we have chosen variable WS for the Jureczko dataset to allow for the widest possible range of WS. This allows us to obtain additional validation for our findings, since we can test more WSs.

5.2 Outlier Models Compared to Other Cluster-based Models

We investigate four different unsupervised algorithms (IF, LOF, kM, MS) with three different feature compression techniques (AE, FA, VIF). Table 2 shows

	IF			kM		LOF			MS	
	AE	FA	VIF	AE	FA	AE	FA	VIF	FA	VIF
Precision	84	84	85	84	85	77	75	75	81	81
Recall	84	85	85	86	86	84	83	82	78	81
F_1-score	84	85	85	82	79	79	78	78	80	81
Accuracy	84	85	85	86	78	86	83	82	78	81

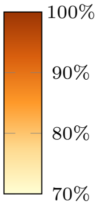


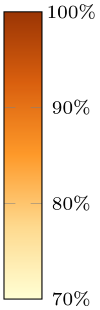
Table 2. The table shows the average weighted Precision, Recall, and F_1-score over both classes and the overall accuracy of all unsupervised models, which exceeded a weighted F_1-score of 10% in the faulty class. Isolation Forest and k-Means are the best unsupervised models.

that from all of the unsupervised models the IF and the kM algorithm performed best. Unexpectedly, the IF with the VIF performed most consistent with a score of 85% followed closely by the IF with FA and kM with AE. In contrast to other unsupervised models, IF seem to actually prefer VIF as feature compression technique. Indeed, our evaluation reveals that in 10 out of 11 projects the IF performs better or is as good with VIF rather than FA in terms of accuracy and in 9 out of 11 projects in terms of the overall weighted F_1-score of both classes. So unlike previously mentioned by Kondo et al., IF as an unsupervised outlier mining model prefers a non-synthetic feature compression technique for most of the 11 projects [19]. Table 2 and our further evaluation (cf. auxiliary material) suggests that an IF with VIF is in the context of our computational experiment the best outlier mining algorithm and kM with AE or FA are the best cluster-based models. In Fig. 5, we further investigate those three models. The figure shows that indeed the IF with VIF is the top model among those models for our 11 example projects, achieving the best predicting scores for 10 out of 11 projects in terms of accuracy and 9 out of 11 projects in terms of the overall weighted F_1-score. Therefore, an IF is indeed a strong contender for unsupervised defect prediction, since they perform consistently over several projects.

5.3 Unsupervised vs. Supervised Models

Table 3 compares quality measures from all supervised models in addition to the IF with VIF for the Eclipse project. The best model is the ensemble of all supervised models (first three rows), outperforming the IF (next three rows) as consistent as possible on 11 out of 11 projects (Fig. 6). The IF performs not worse than any single supervised model. It is as good as one of the weaker supervised models. Although, single supervised model like a RF can outperform an IF quite consistently in 9 out of 11 interms of accuarcy and in 8 out of 11 cases in terms of the F_1-score. Fig. 7 highlights that for the Eclipse project only a comparably small number of models can outperform our IF. As discussed in the previous section, the other unsupervised models can achieve a better result for this specific project and dataset in terms of accuracy, but fail to do this consistently on other

	ES Ensemble						**IF** Iso. Forest			**LR** Logistic Regression					
	FA-EN	FA-MA	FA-SMOTE	VIF-EN	VIF-MA	VIF-SMOTE	AE	FA	VIF	FA-EN	FA-MA	FA-SMOTE	VIF-EN	VIF-MA	VIF-SMOTE
Precision	86	87	86	87	87	87	84	84	85	87	88	87	87	87	87
Recall	87	88	87	78	87	86	84	85	85	79	80	78	78	78	78
F$_1$-score	86	87	86	81	87	86	84	85	85	82	83	80	81	81	81
Accuracy	87	88	87	78	87	86	84	85	85	79	80	78	78	78	78



	NB Naive Bayes						**RF** Random Forest						**SVM** Sup. Vector Machine					
	FA-EN	FA-MA	FA-SMOTE	VIF-EN	VIF-MA	VIF-SMOTE	FA-EN	FA-MA	FA-SMOTE	VIF-EN	VIF-MA	VIF-SMOTE	FA-EN	FA-MA	FA-SMOTE	VIF-EN	VIF-MA	VIF-SMOTE
Precision	86	85	86	86	87	86	86	87	86	87	87	87	86	86	86	85	85	85
Recall	86	87	86	85	85	85	79	78	80	81	76	81	75	88	75	80	87	80
F$_1$-score	86	86	86	85	86	85	81	81	82	83	79	83	78	86	78	82	83	82
Accuracy	86	87	86	85	85	85	79	78	80	81	76	81	75	88	75	80	87	80

Table 3. The table shows that many supervised models outperform an IF. We evaluated our models using over both classes averaged weighted Precision, Recall, F_1-score, and overall Accuracy. The IF is still better than some models.

projects or in terms of another metric like the F_1-score. As before, the ensembled model can achieve better results. It alone already represents half of the cases in which the IF was worse than a supervised model. Interestingly, for this project, no variation of the RF model can outperform our IF model, though here in place the NB model does so relatively consistently.

To summarize, the supervised models or at least the ensemble of supervised models show better results than an IF or any other unsupervised technique.

6 Discussion and Threats to Validity

We compared several basic models. Overall, outlier mining with IF with VIF has shown the best results among the unsupervised defect prediction techniques. However, several supervised models especially the ensemble of all supervised models seems to outperform any unsupervised technique. However, this effect is put into perspective by several additional requirements for training those models. On the one hand, we had to assume a history of recorded labels for the project, which is generally not the case in an industrial context [3,42]. Furthermore, we also favored supervised models by performing more preprocessing steps (class balancing, train-test-split, etc.), which are not required for unsupervised models. In addition, unsupervised models do not require a training phase. Therefore, we spent a significant amount of additional effort to train supervised models compared to unsupervised ones.

We premised several assumptions to make our sliding window technique applicable to our dataset. First, we have not distinguished whether there is one or more defects present in a given file. If a dataset reported more than one defect for the file, we assigned it the label for the faulty class. Second, the datasets reported a numerical label for a defect. Therefore, we could not track the case that a defect may be fixed inside of our window and another one is newly introduced. We only predict whether or not a defect is present in the file in the next revision of the software project. We set the random seed during our data processing to a constant value anywhere where randomness was involved. This, on the one hand, allows us to reproduce our results easily, but on the other hand raises the concern that we could only achieve those results for those specific seeds. However, we achieved our results analyzing several projects and since we did not optimize the seed for each project, it is very unlikely that this seed is especially beneficial for all our investigated projects. Usually, especially for supervised models one would use a random train-test split and a k-fold cross validation. However, here this step would increase the training time for our supervised models even further by the factor k, because also feature compression and class balancing would have to be repeated for each fold. This step would therefore undue favor our supervised models, since no training, and thus, no validation is required for unsupervised models. This is especially true since we already used a simple k-fold cross validation during optimization (only one iteration per hyperparameter set with 5 respectively 3 stratified folds).

In general, it is not possible to transfer results from one dataset to another due to the no-free-lunch theorem without restrictions [1]. Consequently, our results could be different on a different dataset or with different optimization techniques or, for example, different class balancing techniques. However, in the context of our computational experiment, it seems highly recommended to also seriously consider unsupervised techniques for defect prediction because we achieved our results from the evaluation of 11 projects and different sets of metrics. Here, the IF with VIF were also able to achieve a fairly consistent classification result. If the results of Bowes et al. can be extended to unsupervised techniques in future work, unsupervised techniques could also be used to reliably detect a certain class of defects [7]. Supervised models could subsequently specialize in those types of defects that are not so easily detectable by outlier mining techniques. In this way, a combination of supervised and unsupervised techniques could represent a simplification of previous methods, as already noted by Fu and Menzies [12].

We found some anomalies during our research. For example, as shown in Figure 3, KM-FA did especially poorly for window size 8. As stated before, there is no guarantee that a model performing well on one dataset will perform as well on a similar but different dataset [1]. We suspect that this anomaly is an example of such an effect. Furthermore, most of our techniques maximized around 85% in any metric, which could be considered unacceptable for practical use. But, we purposefully concentrated on studying only basic models to get a basic overview of the performances of different methods so that a lower-than-usual quality can be expected.

In summary, it has been demonstrated that it can be beneficial to use unsupervised techniques, especially an IF, for defect prediction especially if no training labels are available and a fast prediction is desirable. This contrasts earlier results, which suggests that supervised techniques will usually outperform unsupervised techniques (cf. Huang et al. [16]).

7 Conclusions and Future Work

Using software metrics for defect prediction is widely used and typically viewed as a classification task, i.e., a supervised learning task. However, supervised learning techniques require a training dataset with training labels and require more (pre-) processing steps. Consequently, we proposed an approach using outlier mining techniques on time series of process metrics to reduce the processing steps.

The evaluation indicates that the proposed method is competitive to single basic supervised models. However, an ensemble of all basic supervised models outperforms any single unsupervised technique on all 11 projects. We still advocate the use of unsupervised models. Due to their ease of use and simplicity, they are more suitable as they require less development and execution time and no labels are required for training. In detail, the experiments are suggesting that an Isolation Forest with the Variance Inflation Factor used for feature compression is the most consistent option for predicting defects in an unsupervised fashion. However, other unsupervised models, e.g., the k-Means algorithm may perform better in certain cases.

Future research has to find a criterion to decide when which technique should be used to further improve an ensemble of models. We also suggest that further research should be conducted to investigate more complex unsupervised techniques for defect prediction, more preprocessing dimensions, and the explainability of models. Future research might also investigate more quality measures, e.g., the number of falsely reported files until the first hit when ordering the files according to their lines of code. In addition, a quantitative study evaluating our approach in an industrial setting may be beneficial.

Acknowledgements. We thank the anonymous reviewers for their valuable feedback. This work was partially funded by the German Ministry for Education and Research (BMBF) through grants 01IS20088B (“KnowhowAnalyzer”) and 01IS22062 (“AI research group FFS-AI”).

Auxiliary Material. The auxiliary material to this paper is provided under: https://t1p.de/SoftwareQualityDays.

References

1. Adam, S.P., Alexandropoulos, S.A.N., Pardalos, P.M., Vrahatis, M.N.: No Free Lunch Theorem: A Review, pp. 57–82. Springer (2019). https://doi.org/10.1007/978-3-030-12767-1_5
2. Albahli, S.: A deep ensemble learning method for effort-aware just-in-time defect prediction. Future Internet **11**(12), 246:1–13 (2019). https://doi.org/10.3390/fi11120246
3. Amasaki, S.: Cross-version defect prediction using cross-project defect prediction approaches: Does it work? In: Proc. 14th International Conference on Predictive Models and Data Analytics in Software Engineering. p. 32–41. PROMISE '18, ACM (2018). https://doi.org/10.1145/3273934.3273938
4. Bennin, K.E., Keung, J., Monden, A., Kamei, Y., Ubayashi, N.: Investigating the effects of balanced training and testing datasets on effort-aware fault prediction models. In: Proc. 40th Annual Computer Software and Applications Conference. pp. 154–163. COMPSAC '16, IEEE (2016). https://doi.org/10.1109/COMPSAC.2016.144
5. Bennin, K.E., Keung, J., Phannachitta, P., Monden, A., Mensah, S.: Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Transactions on Software Engineering **44**(6), 534–550 (2018). https://doi.org/10.1109/TSE.2017.2731766
6. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. Journal of Machine Learning Research **13**(10), 281–305 (2012), http://jmlr.org/papers/v13/bergstra12a.html
7. Bowes, D., Hall, T., Gray, D.: Comparing the performance of fault prediction models which report multiple performance measures: Recomputing the confusion matrix. In: Proc. 8th International Conference on Predictive Models in Software Engineering. pp. 109–118. PROMISE '12, ACM (2012). https://doi.org/10.1145/2365324.2365338
8. Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: Smote: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research **16**(1), 321–357 (2002). https://doi.org/10.1613/jair.953
9. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R.: Catch the moment: maintaining closed frequent itemsets over a data stream sliding window. Knowledge and Information Systems **10**(3), 265–294 (2006). https://doi.org/10.1007/s10115-006-0003-0
10. Ding, Z., Fei, M.: An anomaly detection approach based on isolation forest algorithm for streaming data using sliding window. IFAC Proceedings Volumes **46**(20), 12–17 (2013). https://doi.org/10.3182/20130902-3-CN-3020.00044
11. Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., Gyimóthy, T.: A public unified bug dataset for java. In: Proc. 14th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 12–21. PROMISE '18, ACM (2018). https://doi.org/10.1145/3273934.3273936
12. Fu, W., Menzies, T.: Revisiting unsupervised learning for defect prediction. In: Proc. 11th Joint Meeting on Foundations of Software Engineering. pp. 72–83. ESEC/FSE '17, ACM (2017). https://doi.org/10.1145/3106237.3106257
13. Hawkins, D.M.: The problem of overfitting. Journal of Chemical Information and Computer Sciences **44**(1), 1–12 (2004). https://doi.org/10.1021/ci0342472
14. He, Z., Fan, B., Cheng, T., Wang, S.Y., Tan, C.H.: A mean-shift algorithm for large-scale planar maximal covering location problems. European Journal of Operational Research **250**(1), 65–76 (2016). https://doi.org/10.1016/j.ejor.2015.09.006

15. Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., Godfrey, M.W.: The msr cookbook: Mining a decade of research. In: Proc. 10th Working Conference on Mining Software Repositories. pp. 343–352. MSR '13, IEEE (2013). https://doi.org/10.1109/MSR.2013.6624048
16. Huang, Q., Xia, X., Lo, D.: Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In: Proc. International Conference on Software Maintenance and Evolution. pp. 159–170. ICSME '17, IEEE (2017). https://doi.org/10.1109/ICSME.2017.51
17. Jiarpakdee, J., Tantithamthavorn, C., Hassan, A.E.: The impact of correlated metrics on the interpretation of defect models. IEEE Transactions on Software Engineering **47**(2), 320–331 (2021). https://doi.org/10.1109/TSE.2019.2891758
18. Jureczko, M., Madeyski, L.: Towards identifying software project clusters with regard to defect prediction. In: Proc. 6th International Conference on Predictive Models in Software Engineering. PROMISE '10, ACM (2010). https://doi.org/10.1145/1868328.1868342
19. Kondo, M., Bezemer, C.P., Kamei, Y., Hassan, A.E., Mizuno, O.: The impact of feature reduction techniques on defect prediction models. Empirical Software Engineering **24**(4), 1925–1963 (2019). https://doi.org/10.1007/s10664-018-9679-5
20. Li, N., Shepperd, M., Guo, Y.: A systematic review of unsupervised learning techniques for software defect prediction. Information and Software Technology **122** (2020). https://doi.org/https://doi.org/10.1016/j.infsof.2020.106287
21. Li, Z., Jing, X.Y., Zhu, X.: Progress on approaches to software defect prediction. IET Software **12**(3), 161–175 (2018). https://doi.org/0.1049/iet-sen.2017.0148
22. Liu, F.T., Ting, K.M., Zhou, Z.H.: Isolation forest. In: Proc. 8th International Conference on Data Mining. pp. 413–422. ICDM '08, IEEE (2008). https://doi.org/10.1109/ICDM.2008.17
23. Liu, Y., Li, Y., Guo, J., Zhou, Y., Xu, B.: Connecting software metrics across versions to predict defects. In: Proc. 25th International Conference on Software Analysis, Evolution and Reengineering. pp. 232–243. SANER '18, IEEE (2018). https://doi.org/10.1109/SANER.2018.8330212
24. Mahmood, Z., Bowes, D., Lane, P.C.R., Hall, T.: What is the impact of imbalance on software defect prediction performance? In: Proc. 11th International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 1–4. PROMISE '15, ACM (2015). https://doi.org/10.1145/2810146.2810150
25. Mende, T.: Replication of defect prediction studies: Problems, pitfalls and recommendations. In: Proc. 6th International Conference on Predictive Models in Software Engineering. pp. 1–10. PROMISE '10, ACM (2010). https://doi.org/10.1145/1868328.1868336
26. Miles, J.: Tolerance and Variance Inflation Factor. John Wiley & Sons (2014). https://doi.org/10.1002/9781118445112.stat06593
27. Moshtari, S., Santos, J.C., Mirakhorli, M., Okutan, A.: Looking for software defects? first find the nonconformists. In: Proc. 20th International Working Conference on Source Code Analysis and Manipulation. pp. 75–86. SCAM '20, IEEE (2020). https://doi.org/10.1109/SCAM51674.2020.00014
28. Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., Murphy, B.: Change bursts as defect predictors. In: Proc. 21st International Symposium on Software Reliability Engineering. pp. 309–318. ISSRE '10, IEEE (2010). https://doi.org/10.1109/ISSRE.2010.25
29. Nam, J., Kim, S.: Clami: Defect prediction on unlabeled datasets. In: Proc. 30th IEEE/ACM International Conference on Automated Software Engineering. pp. 452–463. ASE '15 (2015). https://doi.org/10.1109/ASE.2015.56

30. Powers, D.M.W.: Evaluation: From precision, recall and f-measure to roc., informedness, markedness & correlation. Journal of Machine Learning Technologies **2**(1), 37–63 (2011)
31. Rathore, S.S., Kumar, S.: A study on software fault prediction techniques. Artificial Intelligence Review **51**(2), 255–327 (2019). https://doi.org/10.1007/s10462-017-9563-5
32. Runeson, P.: A survey of unit testing practices. IEEE Software **23**(4), 22–29 (2006). https://doi.org/10.1109/MS.2006.91
33. Saravanan, R., Sujatha, P.: A state of art techniques on machine learning algorithms: A perspective of supervised learning approaches in data classification. In: 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS). pp. 945–949 (2018). https://doi.org/10.1109/ICCONS.2018.8663155
34. Sayyad Shirabad, J., Menzies, T.: The PROMISE repository of software engineering databases. School of Information Technology and Engineering, University of Ottawa, Canada (2005)
35. Tantithamthavorn, C., Hassan, A.E., Matsumoto, K.: The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. IEEE Transactions on Software Engineering **46**(11), 1200–1219 (2020). https://doi.org/10.1109/TSE.2018.2876537
36. Verleysen, M., François, D.: The curse of dimensionality in data mining and time series prediction. In: Cabestany, J., Prieto, A., Sandoval, F. (eds.) Computational Intelligence and Bioinspired Systems. pp. 758–770. Springer (2005). https://doi.org/10.1007/11494669_93
37. Xu, Z., Li, L., Yan, M., Liu, J., Luo, X., Grundy, J., Zhang, Y., Zhang, X.: Clustering-based unsupervised models, data analytics for defect prediction, empirical study. Journal of Systems and Software **172**(110862) (2021). https://doi.org/10.1016/j.jss.2020.110862
38. Yan, M., Fang, Y., Lo, D., Xia, X., Zhang, X.: File-level defect prediction: Unsupervised vs. supervised models. In: Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 344–353. ESEM '17, IEE/ACM (2017). https://doi.org/10.1109/ESEM.2017.48
39. Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H.: Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In: Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 157–168. FSE '16, ACM (2016). https://doi.org/10.1145/2950290.2950353
40. Zhang, F., Zheng, Q., Zou, Y., Hassan, A.E.: Cross-project defect prediction using a connectivity-based unsupervised classifier. In: Proc. IEEE/ACM 38th International Conference on Software Engineering. pp. 309–320. ICSE '16 (2016). https://doi.org/10.1145/2884781.2884839
41. Zhu, K., Zhang, N., Ying, S., Zhu, D.: Within-project and cross-project just-in-time defect prediction based on denoising autoencoder and convolutional neural network. IET Software **14**(3), 185–195 (2020). https://doi.org/10.1049/iet-sen.2019.0278
42. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., Murphy, B.: Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In: Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. pp. 91–100. ESEC/FSE '09, ACM (2009). https://doi.org/10.1145/1595696.1595713