Software Quality Journal (2023) 31:1281–1302
https://doi.org/10.1007/s11219-023-09629-1

RESEARCH

Just-in-time defect prediction for mobile applications: using shallow or deep learning?

Raymon van Dinter[1,2] · Cagatay Catal[3] · Görkem Giray[4] · Bedir Tekinerdogan[1]

Accepted: 12 April 2023 / Published online: 9 June 2023
© The Author(s) 2023

Abstract

Just-in-time defect prediction (JITDP) research is increasingly focused on program changes instead of complete program modules within the context of continuous integration and continuous testing paradigm. Traditional machine learning-based defect prediction models have been built since the early 2000s, and recently, deep learning-based models have been designed and implemented. While deep learning (DL) algorithms can provide state-of-the-art performance in many application domains, they should be carefully selected and designed for a software engineering problem. In this research, we evaluate the performance of traditional machine learning algorithms and data sampling techniques for JITDP problems and compare the model performance with the performance of a DL-based prediction model. Experimental results demonstrated that DL algorithms leveraging sampling methods perform significantly worse than the decision tree-based ensemble method. The XGBoost-based model appears to be 116 times faster than the multilayer perceptron-based (MLP) prediction model. This study indicates that DL-based models are not always the optimal solution for software defect prediction, and thus, shallow, traditional machine learning can be preferred because of better performance in terms of accuracy and time parameters.

Keywords Just-in-time defect prediction · Shallow learning · XGBoost · Deep learning · Imbalanced learning

✉ Cagatay Catal
ccatal@qu.edu.qa

Raymon van Dinter
raymon.vandinter@wur.nl; raymon.van.dinter@sioux.eu

Görkem Giray
gorkemgiray@gmail.com

Bedir Tekinerdogan
bedir.tekinerdogan@wur.nl

[1] Information Technology Group, Wageningen University & Research, Wageningen, The Netherlands

[2] Sioux Technologies, Apeldoorn, The Netherlands

[3] Department of Computer Science and Engineering, Qatar University, Doha, Qatar

[4] Izmir, Turkey

1 Introduction

Google Play Store hosts 3.48 million Android applications as of the first quarter of 2021 (Statista Research Department, 2021b). In 2020, approximately 112.6 thousand Android applications were released monthly on average through Google Play Store (Statista Research Department, 2021a). If not all, most of these applications are being updated frequently for corrective or perfective maintenance reasons, including the need for adding new features, removing outdated features, fixing bugs, and improving performance. As a good practice, developers tend to release frequent updates to obtain quick feedback from users and keep their applications up to date. This practice adds additional complexity to the software development process since frequent changes may lead to more defects and require additional effort for developing new test cases. However, there are many other reasons, such as complicated internal logic, why defects still occur in software systems (Wang et al., 2021). Time to market pressure, tight deadlines, and a limited budget for testing are some of the other potential causes of software defects.

To efficiently allocate resources to defect-prone components in software systems, software defect prediction models have been developed using statistical techniques and mostly machine learning algorithms since the 1990s by software engineering researchers (Alan & Catal, 2011). Most of the machine learning models assumed that there is a sufficient number of labeled data points for building supervised machine learning models; however, in some cases, there might be very few labeled data (a.k.a., semi-supervised defect prediction) or even no labeled data exists (a.k.a., unsupervised defect prediction). In these cases, supervised machine learning models cannot be applied because of the missing labels in the data. Researchers built different kinds of models to address these cases (Catal, 2014; Catal & Diri, 2008, 2009; Catal et al., 2010; Li et al., 2020; Sun et al., 2021; Zhang et al., 2017). Some researchers also aimed to utilize data from other companies to build their defect prediction models (a.k.a., cross-project defect prediction) (Jin, 2021; Wu et al., 2017).

Researchers working on software defect prediction aim to help developers in improving software quality and testing efficiency by building machine learning models to predict defect-prone units timely (Lessmann et al., 2008). While such models provide some benefits in some contexts, they also have the following drawbacks (Kamei et al., 2012): (1) developers should explore defect-prone units to identify defects; (2) a developer should be assigned to a defect identification task; (3) developers may forget the details about the changes they made before. To overcome these drawbacks, researchers proposed to predict the code changes that may lead to defects, a.k.a., just-in-time defect prediction (JITDP) (Kamei et al., 2012; Kim et al., 2008; Mockus & Weiss, 2000). Change-level predictions are expected to help the developer, who made a code change, in identifying defects by only reviewing the change at an early stage without forgetting details (Kamei et al., 2012). Recently, many models have been developed to address JITDP and some of them focused on deep learning (DL) algorithms (Yang et al., 2015; Zeng et al., 2021; Zhao et al., 2021a, b, c).

To build a defect prediction model, a dataset including data about the changes (e.g., lines of codes added/modified and the number of modified files) and associated defect data (e.g., defective module and non-defective module) is required. As in many available datasets (Mahmood et al., 2015; Wang et al., 2021), defect data for Android applications are imbalanced (Zhao et al., 2021c). In other words, the number of changes leading to defects is much fewer than those not leading to any defect. Imbalanced datasets cause learning difficulty (Wang & Yao, 2013), and prediction performance decrease (Mahmood et al., 2015).

Sampling methods are the dominant approaches to tackle imbalanced learning problems (He & Ma, 2013).

Since recently DL algorithms have achieved remarkable results in many different application domains, there is a tendency among software engineering researchers to apply them for all kinds of relevant problems (Giray, 2021), often without considering the scale of the available datasets and applicability of the algorithm in the context. In this research, our objective is to evaluate the performance of shallow learning algorithms (i.e., traditional machine learning algorithms) and the effect of sampling methods on defect prediction models and also compare these results with the performance of DL algorithms. If acceptable performance can be achieved with traditional algorithms, we can conclude that there is no need to deal with DL algorithms because they require extra computing power, training time, and human effort to find the optimal model. Therefore, we evaluate three high-performance machine learning algorithms and eight sampling methods in this study. We report our findings on the comparison of the prediction performance of three machine learning algorithms (i.e., MLP, TabNet, and XGBoost) using eight sampling methods (i.e., ROS, RUS, SMOTE, SMOTEN, SMOTESVM, SMOTET, BSMOTE, and ADASYN). Experiments were performed on 12 publicly available datasets built based on Android apps (Catolino et al., 2019).

The contributions of this study are four-fold:

- We demonstrated that DL algorithms using sampling methods perform significantly worse than the decision tree-based ensemble method.
- We proposed a new XGBoost-based prediction model using the StandardScaler normalization and SVMSMOTE data balancing approaches and evaluated its performance on 12 publicly available Android datasets.
- The XGBoost-based model is 116 times faster than the MLP method and has a 32% higher MCC (Matthews correlation coefficient) than the baseline method. This study showed that DL-based models are not always the answer for building highly accurate prediction models and XGBoost-based models can even provide better performance in terms of computational time complexity and performance.
- Our experiments are reproducible and improvable, as our code is publicly available at https://github.com/rvdinter/JIT-defect-prediction-Android-apps.

The rest of the paper is organized as follows: Section 2 introduces the related work. Section 3 explains the research methodology. Section 4 presents the results. Section 5 discusses our findings and reports the threats to validity. Section 6 concludes the paper.

2 Related work

Kamei et al. (2012) proposed predicting defects by analyzing changes instead of files or packages. They conducted a large-scale study involving six open-source and five commercial non-mobile applications to predict defects just-in-time.

Scandariato and Walden (2012) focused on developing a vulnerability model specific to Android applications. Their prediction model is based on object-oriented metrics, like the number of superclasses, depth of inheritance tree, cumulative Halstead bugs, and Halstead volume, to name a few. Kaur et al. obtained better prediction performance for mobile applications by using process metrics (i.e., number of lines added/deleted,

number of developers, number of revisions) instead of code metrics (Kaur et al., 2015, 2016). Malhotra (2016) compared the prediction performance of 18 machine learning algorithms using object-oriented metrics as the feature set. Ricky et al. (2016) built a prediction model for the games developed for Android and Windows Phone using software metrics.

Catolino et al. (2019) analyzed the relevant metrics useful for predicting defects just-in-time in mobile applications. They applied information gain for feature selection with a threshold of 0.1, as suggested by previous studies (Catolino et al., 2018; Quinlan, 1986). For coping with imbalanced dataset problem, they applied Synthetic Minority Over-sampling Technique (SMOTE), proposed by Chawla et al. (2002). They identified the following six metrics that contributed to defect prediction with an information gain value exceeding 0.1: number of unique changes to modified files (nuc), number of lines added (la), number of lines deleted (ld), number of modified files (nf), number of modified directories (nd), and number of developers working on the file (ndev). Zhao et al. (2021b) proposed an imbalanced DL (IDL) methodology for JIT defect prediction in Android applications through applying a cost-sensitive cross entropy loss function to a deep neural network. They compared their model against the sampling-based imbalanced learning methods (ROS, RUS, SMOT, SMOTEN, SVMSMOT, SMOTT, SMOTB, and ADASYN) by conducting experiments on a benchmark dataset involving 12 Android applications and found out that their model performed better than the other imbalanced learning methods in terms of Matthews correlation coefficient performance indicator.

Bennin et al. (2017) state the drawbacks of sampling approaches: (1) the possibility of generating erroneous or duplicated data instances, (2) the tendency to generate less diverse data points within the minority class. To overcome these drawbacks, they propose an over-sampling approach called MAHAKIL. Their approach outperformed four other over-sampling approaches (ROS, SMOTE, Borderline-SMOTE, and ADASYN) using five classification models (C4.5, NNET, KNN, RF, SVM) on 20 imbalanced datasets consisting of non-mobile applications. Tantithamthavorn et al. (2018) investigated the impact of four resampling methods (over-sampling, under-sampling, SMOTE, and ROSE) by building defect prediction models based on seven classification techniques (random forest, logistic regression, Naive Bayes, neural network (AVNNet), C5.0 Boosting (C5.0), extreme gradient boosting (xGBTree), and gradient boosting method) and 101 publicly available datasets. Bennin et al. (2019) conducted an experiment on six sampling methods (SMOTE, Borderline-SMOTE, Safe-level SMOTE, ADASYN, random over-sampling, random under-sampling), five prediction models (KNN, SVM, C4.5, RF, and NNET), and 20 open-source projects.

Researchers have been utilizing DL for defect prediction, more densely as of 2019 (Giray et al., 2023). Two recent surveys report the increasing use of DL in JITDP (Zhao et al., 2023) and specifically for mobile apps (Jorayeva et al., 2022a). Jorayeva et al. (2022b) investigated the performance of DL and data balancing approaches on cross-project defect prediction for mobile apps. Cheng et al. (2022) proposed a method for cross-project JITDP in the context of Android mobile apps. Huang et al. (2023) used multi-task learning and deep neural network to alleviate limited labeled data problem JITDP on mobile apps.

In this study, we replicated the experiments conducted by Zhao et al. (2021b). Also, we investigated the performance of sampling methods when used with a base deep neural network (MLP), a neural network designed for tabular data (TabNet), and a traditional machine learning algorithm designed for tabular data (XGBoost) in JIT defect prediction for Android applications. We compare the results of our algorithms against the IDL methodology by Zhao et al. (2021b). To the best of our knowledge, this is the first study that evaluates the relative performance of shallow learning algorithms against DL algorithms in the case of JITDP.

3 Research methodology

In the following, we describe the steps of the method. First of all, we describe the adopted datasets necessary for JIT prediction. This is followed by the models that are used for JIT prediction. Since we are dealing with unbalanced datasets, we will also elaborate on the sampling-based imbalanced learning methods. Finally, we describe the adopted methods and the flowchart for the experiments.

3.1 Dataset

We have used 12 publicly available benchmark datasets from Android apps built by Catolino et al. (2019). Catolino et al. (2019) also provide an in-depth description of each of these apps. Table 1 provides an overview of the 12 Android apps that have been used to evaluate the algorithms. The table shows for each app the lines of code (#LOC), total number of commit instances (#TC), total number of defective instances (#DC), total number of clean instances (#CC), and the ratio of defective instances (%DR). An instance is deemed as defective when the committed instance introduces the defect; otherwise, it is deemed as clean. The ratio of defective instances varies greatly between 14 and 40%. Also, the scale of the apps varies, as the code lines of these apps are between 9506 and 275,637. The number of samples in the dataset is equal to the total number of commit instances. This means that for Turner, there are only 164 samples to learn from. Catolino et al. (2019) analyzed several features that could be of interest for classifying defective commit instances. They identified six features that could be categorized into three scopes: history, size, and diffusion (Zhao et al., 2021b).

Table 2 provides a brief description of the six features deemed most informative for JIT defect prediction for Android apps.

Table 1 Metadata of the 12 Android apps

Project	#LOC	#TC	#DC	#CC	%DR
Firewall	77,243	1025	414	611	40.4%
Alfresco	152,047	1004	214	790	21.3%
Sync	275,637	209	62	147	30.0%
Wallpaper	35,917	588	94	494	16.0%
Keyboard	114,784	2971	819	2152	27.6%
Apg	151,204	3780	1304	2476	34.5%
Secure	98,768	2579	853	1726	33.1%
Facebook	103,802	548	180	368	32.8%
Kiwix	32,598	1373	350	1023	25.5%
Cloud	115,169	3700	830	2870	22.4%
Turner	30,943	164	23	141	14.0%
Reddit	9506	222	60	162	27.0%

Table 2 Description of six features for JIT defect prediction

Scope	Feature	Description
History	NUC	Number of unique changes to modified files
	NDEV	Number of developers working on the files
Size	LD	Lines of code deleted
	LA	Lines of code added
Diffusion	NF	Number of modified files
	ND	Number of modified directories

3.2 Models

In this study, we evaluate three models: a vanilla deep neural network, a neural network designed for tabular data, and a traditional machine learning algorithm designed for tabular data. We focus our models on tabular data, as the datasets by Catolino et al. (2019) are of tabular form.

3.2.1 Multilayer perceptron

We make use of a multilayer perceptron model based on the studies from Zhao et al. (2021b) and Xu et al. (2019). The multilayer perceptron (MLP) is a class of feedforward artificial neural networks (ANN). It makes use of three network layer types: (1) the input layer, (2) the hidden layer, and (3) the output layer. The first layer is the input layer, which uses as many units as the number of features in the dataset. The last layer is the output layer, which outputs the requested result. In our case, as shown in Fig. 1, the input layer is of size 6, and the output layer is of size 1, as the model classifies whether the commit does (i.e., the output is 1) or does not (i.e., the output is 0) contain a defect using a Sigmoid activation function. There can be multiple hidden layers in an MLP. Also, the number of units in a hidden layer can vary. If more than one hidden layer is used in an ANN, nowadays, the model is called deep neural network (DNN). Our MLP makes use of 2 hidden layers with 10 units leveraging the ReLu activation function. For the hyperparameters, we apply the RMSProp optimization algorithm with a batch size of 16 and 10,000 iterations. We use an adapted learning rate of 0.001 without weight decay to keep the model as simple as possible. We did not apply early stopping for regularization.

3.2.2 TabNet

A major downside to MLP and other DL models is that their predictions cannot be explained. However, TabNet attempts to break this assumption (Arık & Pfister, 2020). This DL algorithm attempts to combine the best of both worlds: the performance of DL algorithms while being explainable like decision tree-based classifiers. Being developed by Google Cloud AI, it is already widely rolled out in Google Cloud Platform (Arık & Le, 2020). As Arık and Pfister (2020) describe: "TabNet uses sequential attention to choose which features to reason from at each decision step, enabling interpretability and

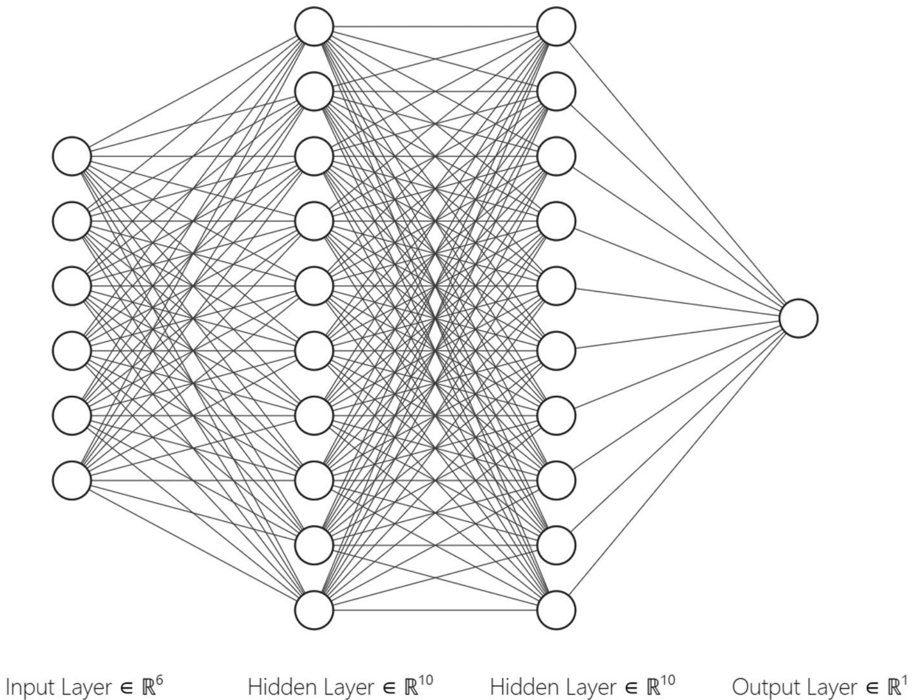

Fig. 1 The multilayer perceptron model architecture adapted from Zhao et al. (2021b) and Xu et al. (2019)

more efficient learning as the learning capacity is used for the most salient features" (Arık & Pfister, 2020).

We optimized TabNet's model hyperparameters using sklearn's RandomizedSearchCV on the Reddit dataset, as the TabNet model contains many hyperparameters, which makes GridSearchCV very resource-intensive. In this search, we used mostly default hyperparameters, while applying a clip value of 1, and a learning rate of 2e-3. Additionally, we used a maximum of 150 epochs, and a patience of 20 epochs with no improvement in the loss after which the model stops training, which also required the use of a validation set, which is a 10% partition of the training set.

3.2.3 XGBoost

To solve class imbalance challenges, one could use sampling methods, ensemble-based algorithms, or cost-based methods (Zhao et al., 2021b). XGBoost, or eXtreme Gradient Boosting, is a decision tree-based ensemble method developed by Chen et al. (2015). It is based on gradient boosting machines, such as AdaBoost, which has been used often to compare classifiers in this domain (Catolino et al., 2019; Zhao et al., 2021a, b, c). A gradient boosting machine uses a loss function, many weak learners, and an additive model to add weak learners to minimize the loss (Brownlee, 2019). XGBoost's advantage over other gradient boosting machines is its speed and performance. As Brownlee (2019) notes, XGBoost has been the go-to model for tabular machine learning challenges on Kaggle for years.

We used GridSearchCV to find XGBoost's optimal hyperparameters on the Reddit dataset. We search to find the optimal maximum depth, number of estimators, and learning rate. This resulted in a maximum depth of 3100 estimators, and a learning rate of 0.1.

3.3 Sampling-based imbalanced learning methods

In general, imbalanced learning methods are based on sampling, ensembles, or cost functions. We evaluate our models described in the previous section against eight sampling methods to analyze whether the models can gain significant performance apart from hyperparameter tuning. Table 3 lists the eight imbalanced learning methods we evaluated. RUS is the only under-sampling method, SMOTET is a combination of over- and under-sampling methods, while all other methods use an over-sampling method.

3.4 Metric

In the JITDP, AUC, F-measure, and MCC metrics are often reported to synthesize the results of a study. However, Ng (2017) describes that adding multiple metrics in a paper can confuse which metric is most important. Therefore, he recommends using an all-encompassing metric. Previous studies have proven that the MCC is the most appropriate metric for JIT defect prediction (Song et al., 2018; Yao & Shepperd, 2020).

The MCC, or Matthews correlation coefficient, is a metric used for binary and multiclass classification. It is designed for imbalanced datasets, such as software defect prediction. MCC is derived from the Pearson correlation coefficient and takes all terms from the confusion matrix. MCC's formula is expressed as

$$\frac{TP\times TN-FP\times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}\# \tag{1}$$

where TP, TN, FP, and FN denote true positive, true negative, false positive, and false negative, respectively. The MCC is a correlation coefficient value between − 1 and + 1. This statistic is also known as the phi coefficient. As such, a coefficient of + 1 is a perfect prediction, 0 is an average random prediction, and a coefficient of − 1 is an inverse prediction.

Table 3 Imbalanced learning methods

Abbreviation	Description
ROS	Random over-sampling
RUS	Random under-sampling
SMOTE	Synthetic Minority Over-sampling Technique
SMOTEN	Synthetic Minority Over-sampling Technique for Nominal
SMOTESVM	Over-sampling using SVMSMOTE
SMOTET	Over-sampling using SMOTE and cleaning using Tomek links
BSMOTE	Over-sampling using Borderline-SMOTE
ADASYN	Oversample using Adaptive Synthetic (ADASYN) algorithm

3.5 Experiments

As in the experiments by Zhao et al. (2021b), we evaluate a set of model and sampler configurations. To do so, we have generated nested iterations. Figure 2 shows the detailed visualization of the procedure.

First, we iterate over the models to evaluate: MLP, TabNet, and XGBoost. Then, we iterate over the eight sampling methods. For each of these model-sampler configurations, we perform twofold cross-validation, which we repeat 25 times (i.e., N*M cross-validation, $N=2$, $M=25$). If the repeated cross-validation is finished, we move to the next sampler until all samplers have been evaluated against one model. Then, we move to the next model and repeat the process.

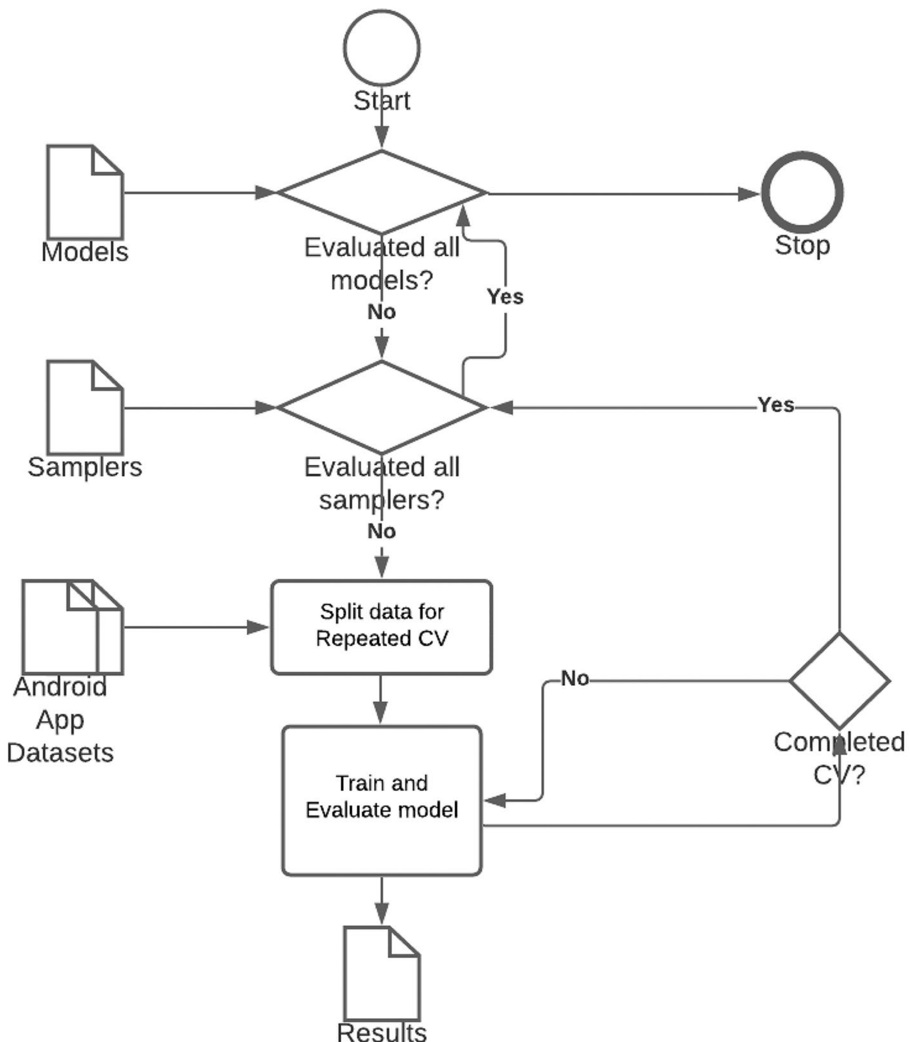

Fig. 2 Visualization of the experiment

4 Results

In this section, we evaluate the results from our models and each of the sampling-based learning methods. We included the results from Zhao et al. (2021b) in Table 4 as a baseline for comparison. The results show the mean and the corresponding standard deviation of each sampling-based method for the twelve datasets. The last row shows the average for the sampling-based method for all datasets. The highest mean values for Table 4 are shown in bold. Table 4 shows that the IDL method gained the highest score on 8 out of 12 datasets. Additionally, the IDL method gained the highest score on average.

Tables 5, 6 and 7 show the results from our study. As with Table 4, the highest means are in bold. Additionally, if the highest mean value is higher than the highest mean value from the baseline in Table 4, we added an asterisk. Table 5 shows the MCC results for the MLP model. The MLP outperforms the IDL method on 7 out of 12 datasets. Even though our MLP is based on the MLP from Zhao et al. (2021b), hyperparameter optimization resulted in large performance gains. Additionally, the average result for the MLP model with SVMSMOTE sampling-based method was higher than the baseline IDL method. Table 6 shows the average MCC results for the TabNet model. As its hyperparameters have been optimized for the Reddit dataset, we see that the TabNet model gained the highest performance on that dataset. Unfortunately, the hyperparameter settings are not dataset-independent, as the average MCC results for the other datasets and sampling-based methods are not outperforming the baseline. Table 7 shows the results for the XGBoost model with the eight sampling-based methods. We see that the MCC results for this method are higher on average than the baseline and DL models. Additionally, SVMSMOTE outperformed the baseline's highest average values on 5 out of 12 datasets, and SMOTET achieved the highest average MCC of all methods.

We performed a Scott-Knott ESD (SKESD) test to statistically verify which sampling method performs best per model. Figures 3, 4 and 5 show the SKESD results for the MLP, TabNet, and XGBoost models respectively. For the MLP and XGBoost, SVMSMOTE has been ranked highest overall, while it has been ranked third for the TabNet model. TabNet, however, has ranked SMOTET as the overall best sampling method. SMOTET is ranked second for the XGBoost model and sixth for the MLP model. Overall, SVMSMOTE has been ranked as the best sampling-based method among the models. The SVMSMOTE method for XGBoost has improved the MCC results by 32% over the IDL baseline method. Additionally, we performed a SKESD test for the machine learning methods per dataset for all sampling methods. In Fig. 6, MLP and TabNet have been ranked closely together, while XGBoost has been ranked first for eleven out of twelve datasets.

Lastly, we share our observations of the time consumption of the MLP, TabNet, and XGBoost algorithms. The TabNet model utilized the GPU, as it has been optimized for GPU processing. The MLP and XGBoost utilized a CPU. The experiments have been run on the Kaggle Kernels free cloud computing service. Figure 7 shows the time consumption of each of the machine learning algorithms for the full experiment. We see that the MLP took 1860 min (31 h) to complete the experiment. The TabNet algorithm took 414 min (6.9 h), while XGBoost took just 16 min to complete the experiment. This means that XGBoost completed the experiment over 116 times faster than the MLP algorithm and over 25 times faster than the TabNet algorithm.

Table 4 Baseline results, adapted from Zhao et al. (2021b)

Project	ROS	RUS	SMOTE	SMOTEN	SVMSMOTE	SMOTET	BSMOTE	ADASYN	IDL
Firewall	0.162 ± (0.153)	0.146 ± (0.154)	0.166 ± (0.149)	0.154 ± (0.160)	0.157 ± (0.156)	0.160 ± (0.154)	0.156 ± (0.146)	0.159 ± (0.146)	**0.249 ± (0.057)**
Alfresco	0.347 ± (0.126)	0.327 ± (0.151)	0.340 ± (0.147)	0.313 ± (0.141)	0.334 ± (0.122)	0.345 ± (0.141)	0.326 ± (0.145)	0.315 ± (0.165)	**0.383 ± (0.053)**
Sync	0.402 ± (0.089)	0.390 ± (0.090)	0.357 ± (0.125)	0.353 ± (0.083)	0.383 ± (0.106)	0.352 ± (0.107)	0.339 ± (0.137)	0.339 ± (0.142)	**0.410 ± (0.058)**
Wallpaper	0.221 ± (0.055)	0.210 ± (0.067)	**0.232 ± (0.050)**	0.218 ± (0.051)	0.221 ± (0.059)	0.226 ± (0.065)	0.223 ± (0.066)	0.223 ± (0.067)	0.157 ± (0.057)
Keyboard	0.212 ± (0.127)	0.213 ± (0.123)	0.199 ± (0.120)	0.269 ± (0.070)	0.233 ± (0.123)	0.220 ± (0.115)	0.181 ± (0.122)	0.173 ± (0.123)	**0.288 ± (0.025)**
Apg	0.215 ± (0.123)	0.210 ± (0.122)	0.234 ± (0.121)	**0.235 ± (0.128)**	0.171 ± (0.146)	0.223 ± (0.120)	0.206 ± (0.119)	0.214 ± (0.114)	0.205 ± (0.092)
Secure	0.229 ± (0.162)	0.221 ± (0.157)	0.230 ± (0.155)	0.207 ± (0.154)	0.163 ± (0.160)	0.226 ± (0.154)	0.224 ± (0.155)	0.221 ± (0.154)	**0.275 ± (0.073)**
Facebook	0.220 ± (0.199)	0.226 ± (0.203)	0.218 ± (0.203)	0.210 ± (0.193)	0.219 ± (0.204)	0.223 ± (0.196)	0.217 ± (0.198)	0.220 ± (0.201)	**0.282 ± (0.098)**
Kiwix	0.258 ± (0.119)	0.258 ± (0.116)	0.231 ± (0.129)	0.213 ± (0.139)	0.234 ± (0.120)	0.220 ± (0.132)	0.232 ± (0.123)	0.233 ± (0.125)	**0.311 ± (0.021)**
Cloud	0.336 ± (0.073)	0.337 ± (0.082)	0.299 ± (0.125)	0.292 ± (0.129)	0.278 ± (0.128)	0.294 ± (0.135)	0.251 ± (0.189)	0.265 ± (0.157)	**0.385 ± (0.018)**
Turner	0.257 ± (0.141)	0.259 ± (0.152)	0.254 ± (0.165)	0.274 ± (0.128)	**0.329 ± (0.111)**	0.280 ± (0.124)	0.307 ± (0.114)	0.268 ± (0.150)	0.303 ± (0.143)
Reddit	0.523 ± (0.118)	0.531 ± (0.097)	0.521 ± (0.116)	0.525 ± (0.090)	**0.542 ± (0.080)**	0.521 ± (0.099)	0.511 ± (0.118)	0.505 ± (0.121)	0.438 ± (0.053)
Average	0.282 ± (0.098)	0.277 ± (0.100)	0.273 ± (0.092)	0.272 ± (0.092)	0.272 ± (0.106)	0.274 ± (0.092)	0.264 ± (0.092)	0.261 ± (0.089)	**0.307 ± (0.081)**

Highest mean values for each dataset are shown in bold

Table 5 Average MCC results of the MLP model and sampling-based methods

Project	ROS	RUS	SMOTE	SMOTEN	SVMSMOTE	SMOTET	BSMOTE	ADASYN
Firewall	0.354 ± (0.034)	0.35 ± (0.036)	0.36 ± (0.035)	0.33 ± (0.041)	**0.361 ± (0.037)***	0.352 ± (0.03)	0.344 ± (0.04)	0.341 ± (0.042)
Alfresco	0.264 ± (0.062)	0.35 ± (0.039)	0.236 ± (0.066)	0.206 ± (0.07)	0.328 ± (0.06)	**0.386 ± (0.037)***	0.216 ± (0.067)	0.273 ± (0.064)
Sync	0.283 ± (0.094)	0.278 ± (0.108)	0.281 ± (0.091)	0.273 ± (0.101)	**0.286 ± (0.086)**	0.274 ± (0.097)	0.272 ± (0.087)	0.272 ± (0.105)
Wallpaper	0.144 ± (0.085)	0.139 ± (0.058)	0.138 ± (0.075)	0.107 ± (0.062)	0.154 ± (0.064)	0.14 ± (0.076)	**0.147 ± (0.064)**	0.146 ± (0.073)
Keyboard	0.254 ± (0.028)	**0.331 ± (0.023)***	0.249 ± (0.029)	0.261 ± (0.027)	0.281 ± (0.032)	0.233 ± (0.115)	0.245 ± (0.024)	0.16 ± (0.113)
Apg	**0.377 ± (0.026)***	0.225 ± (0.051)	0.376 ± (0.022)	0.349 ± (0.029)	0.368 ± (0.027)	0.281 ± (0.087)	0.365 ± (0.037)	0.329 ± (0.058)
Secure	**0.371 ± (0.033)***	0.359 ± (0.036)	0.371 ± (0.035)	0.339 ± (0.03)	0.37 ± (0.032)	0.227 ± (0.11)	0.368 ± (0.034)	0.257 ± (0.133)
Facebook	0.43 ± (0.051)	0.416 ± (0.054)	0.434 ± (0.033)	0.378 ± (0.056)	0.434 ± (0.05)	**0.436 ± (0.046)***	0.429 ± (0.045)	0.428 ± (0.045)
Kiwix	0.301 ± (0.041)	**0.309 ± (0.037)**	0.305 ± (0.036)	0.225 ± (0.057)	0.308 ± (0.037)	0.294 ± (0.057)	0.288 ± (0.046)	0.305 ± (0.037)
Cloud	0.394 ± (0.023)	**0.407 ± (0.028)***	0.4 ± (0.025)	0.383 ± (0.024)	0.399 ± (0.032)	0.239 ± (0.109)	0.403 ± (0.026)	0.303 ± (0.116)
Turner	0.238 ± (0.134)	0.238 ± (0.116)	0.287 ± (0.115)	0.226 ± (0.122)	0.273 ± (0.13)	**0.296 ± (0.119)**	0.272 ± (0.117)	0.257 ± (0.137)
Reddit	0.439 ± (0.089)	0.446 ± (0.087)	0.447 ± (0.095)	0.412 ± (0.09)	**0.467 ± (0.077)**	0.449 ± (0.075)	0.432 ± (0.078)	0.445 ± (0.076)
Average	0.321 ± (0.058)	0.321 ± (0.056)	0.324 ± (0.055)	0.291 ± (0.059)	**0.336 ± (0.055)***	0.301 ± (0.08)	0.315 ± (0.055)	0.293 ± (0.083)

Highest mean values for each dataset are shown in bold. If it is higher than the baseline in Table 4, we added an asterisk

Table 6 Average MCC results of the TabNet model and sampling-based methods

Project	ROS	RUS	SMOTE	SMOTEN	SVMSMOTE	SMOTET	BSMOTE	ADASYN
Firewall	0.21 ± (0.074)	0.2 ± (0.098)	0.201 ± (0.077)	0.197 ± (0.088)	0.214 ± (0.087)	0.186 ± (0.09)	0.21 ± (0.08)	**0.223 ± (0.073)**
Alfresco	0.342 ± (0.057)	0.334 ± (0.05)	0.328 ± (0.071)	**0.345 ± (0.061)**	0.326 ± (0.053)	0.343 ± (0.058)	0.343 ± (0.048)	0.335 ± (0.053)
Sync	0.353 ± (0.093)	0.352 ± (0.098)	0.354 ± (0.093)	0.321 ± (0.108)	0.359 ± (0.098)	**0.359 ± (0.094)**	0.351 ± (0.105)	0.345 ± (0.097)
Wallpaper	0.132 ± (0.055)	**0.15 ± (0.063)**	0.133 ± (0.059)	0.079 ± (0.084)	0.129 ± (0.049)	0.135 ± (0.046)	0.142 ± (0.048)	0.14 ± (0.051)
Keyboard	0.262 ± (0.038)	0.23 ± (0.037)	0.272 ± (0.038)	**0.275 ± (0.047)**	0.271 ± (0.04)	0.27 ± (0.038)	0.264 ± (0.041)	0.258 ± (0.041)
Apg	0.299 ± (0.032)	0.278 ± (0.044)	0.302 ± (0.034)	0.291 ± (0.041)	0.297 ± (0.04)	**0.306 ± (0.036)***	0.283 ± (0.036)	0.302 ± (0.04)
Secure	0.261 ± (0.046)	0.24 ± (0.05)	0.261 ± (0.039)	0.233 ± (0.048)	0.257 ± (0.043)	**0.271 ± (0.043)**	0.246 ± (0.039)	0.249 ± (0.049)
Facebook	0.186 ± (0.117)	0.137 ± (0.127)	0.158 ± (0.111)	0.158 ± (0.1)	0.186 ± (0.102)	**0.189 ± (0.12)**	0.175 ± (0.121)	0.16 ± (0.113)
Kiwix	0.218 ± (0.065)	**0.228 ± (0.064)**	0.211 ± (0.075)	0.214 ± (0.073)	0.219 ± (0.063)	0.215 ± (0.059)	0.218 ± (0.063)	0.216 ± (0.065)
Cloud	0.278 ± (0.046)	0.189 ± (0.066)	0.287 ± (0.049)	0.291 ± (0.041)	**0.295 ± (0.047)**	0.281 ± (0.052)	0.249 ± (0.057)	0.27 ± (0.06)
Turner	0.116 ± (0.148)	**0.121 ± (0.144)**	0.096 ± (0.162)	0.036 ± (0.108)	0.119 ± (0.154)	0.104 ± (0.132)	0.117 ± (0.135)	0.105 ± (0.119)
Reddit	0.543 ± (0.079)	**0.55 ± (0.095)***	0.537 ± (0.062)	0.528 ± (0.095)	0.535 ± (0.081)	0.54 ± (0.075)	0.52 ± (0.088)	0.519 ± (0.073)
Average	0.267 ± (0.071)	0.251 ± (0.078)	0.262 ± (0.072)	0.247 ± (0.075)	0.267 ± (0.071)	**0.267 ± (0.07)**	0.26 ± (0.072)	0.26 ± (0.07)

Highest mean values for each dataset are shown in bold. If it is higher than the baseline in Table 4, we added an asterisk

Table 7 Average MCC results of the XGBoost model and sampling-based methods

Project	ROS	RUS	SMOTE	SMOTEN	SVMSMOTE	SMOTET	BSMOTE	ADASYN
Firewall	0.386 ± (0.035)	0.373 ± (0.036)	**0.386 ± (0.034)***	0.348 ± (0.036)	0.383 ± (0.036)	0.385 ± (0.034)	0.378 ± (0.034)	0.379 ± (0.036)
Alfresco	0.396 ± (0.046)	0.355 ± (0.036)	0.409 ± (0.041)	0.37 ± (0.043)	**0.415 ± (0.037)***	0.403 ± (0.039)	0.396 ± (0.042)	0.401 ± (0.04)
Sync	0.368 ± (0.076)	0.363 ± (0.074)	0.375 ± (0.083)	0.327 ± (0.08)	0.387 ± (0.07)	**0.39 ± (0.084)**	0.369 ± (0.077)	0.366 ± (0.087)
Wallpaper	0.185 ± (0.067)	0.186 ± (0.051)	0.206 ± (0.068)	0.102 ± (0.067)	0.191 ± (0.061)	0.211 ± (0.065)	**0.212 ± (0.062)**	0.205 ± (0.051)
Keyboard	0.424 ± (0.024)	0.414 ± (0.021)	0.425 ± (0.021)	0.363 ± (0.022)	**0.43 ± (0.022)***	0.429 ± (0.023)	0.423 ± (0.023)	0.416 ± (0.023)
Apg	0.469 ± (0.018)	0.461 ± (0.016)	0.471 ± (0.014)	0.448 ± (0.018)	**0.473 ± (0.016)***	0.472 ± (0.015)	0.464 ± (0.015)	0.465 ± (0.015)
Secure	0.456 ± (0.02)	0.444 ± (0.022)	0.466 ± (0.022)	0.427 ± (0.023)	0.467 ± (0.02)	**0.467 ± (0.018)***	0.46 ± (0.018)	0.463 ± (0.017)
Facebook	0.429 ± (0.035)	0.425 ± (0.041)	**0.456 ± (0.043)***	0.398 ± (0.048)	0.451 ± (0.039)	0.45 ± (0.044)	0.442 ± (0.039)	0.446 ± (0.044)
Kiwix	0.378 ± (0.027)	0.361 ± (0.027)	0.383 ± (0.026)	0.315 ± (0.037)	**0.397 ± (0.028)***	0.384 ± (0.028)	0.375 ± (0.031)	0.38 ± (0.03)
Cloud	0.484 ± (0.019)	0.47 ± (0.016)	0.49 ± (0.018)	0.44 ± (0.021)	**0.494 ± (0.017)***	0.487 ± (0.021)	0.481 ± (0.018)	0.479 ± (0.019)
Turner	0.244 ± (0.15)	0.294 ± (0.094)	0.321 ± (0.137)	0.141 ± (0.155)	0.285 ± (0.152)	0.323 ± (0.143)	0.31 ± (0.142)	**0.339 ± (0.131)***
Reddit	0.468 ± (0.073)	**0.503 ± (0.064)**	0.484 ± (0.062)	0.434 ± (0.089)	0.489 ± (0.063)	0.492 ± (0.068)	0.48 ± (0.058)	0.487 ± (0.068)
Average	0.391 ± (0.049)	0.388 ± (0.042)	0.406 ± (0.047)	0.343 ± (0.053)	0.405 ± (0.047)	**0.408 ± (0.048)**	0.399 ± (0.046)	0.402 ± (0.047)

Highest mean values for each dataset are shown in bold. If it is higher than the baseline in Table 4, we added an asterisk

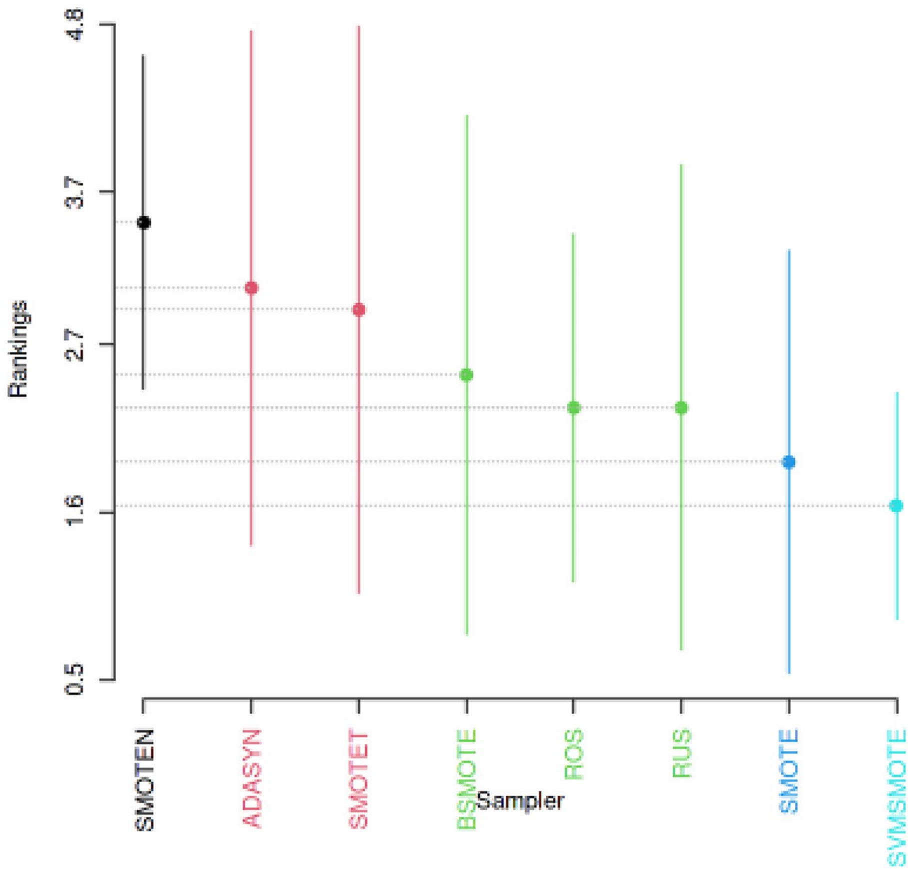

Fig. 3 SKESD test for sampling-based methods for MLP

5 Discussion

This study presents a comparison between a baseline study, a slightly adapted deep neural network, a state-of-the-art DL method for tabular data, and a decision trees-based ensemble method. Our results show that the XGBoost decision tree-based ensemble method is the fastest and statistically highest ranked method. XGBoost in combination with the SVMSMOTE over-sampling method shows an improvement over the IDL baseline method of 32% of the MCC results. Furthermore, our XGBoost method is 99% faster than the MLP method. If the time consumption of the baseline method can be assumed to be like the MLP method, we would also cut the time cost of the IDL method by 99%.

Our results also show that the differences between various sampling methods are minimal in comparison to the results between various machine learning algorithms. SMOTESVM and SMOTET have shown to be good sampling methods, however, to increase the performance of a machine learning algorithm.

This study is subject to threats to validity that can be classified as construct, internal, and external.

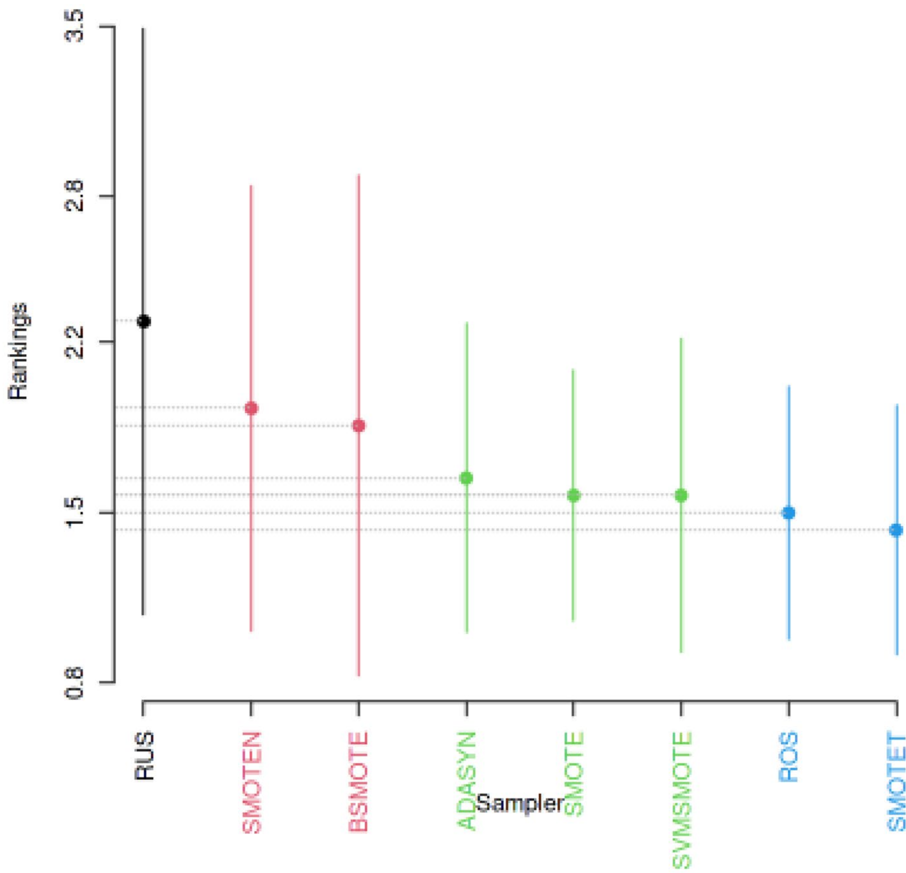

Fig. 4 SKESD test for sampling-based methods for TabNet

Construct validity As Ng (2017) described, having multiple-number evaluation metrics (e.g., precision and recall) makes it challenging to compare algorithms. Having single-number evaluation metrics allows us to sort all models according to their performance. An all-encompassing single-number evaluation metric for the JIT defect prediction domain is the Matthews correlation coefficient. The MCC is appropriate for imbalanced datasets. Additionally, to verify the statistical validity of our results, we apply a state-of-the-art statistic test method. The Scott-Knott effect size difference has been designed for JIT defect prediction. It was used for "significant difference" analysis.

Internal validity In our work, we carefully implemented the sampling methods and machine learning algorithms using the DMLC XGBoost, DreamQuark TabNet, and PyTorch library. The optimal parameters have been obtained using RandomSearchCV for the TabNet algorithm and GridSearchCV for the XGBoost library. The MLP was based on previous studies (Zhao et al., 2021b, c); however, more optimal settings could be found through hyperparameter optimization algorithms. For comparative methods, we implemented third-party libraries with default parameters.

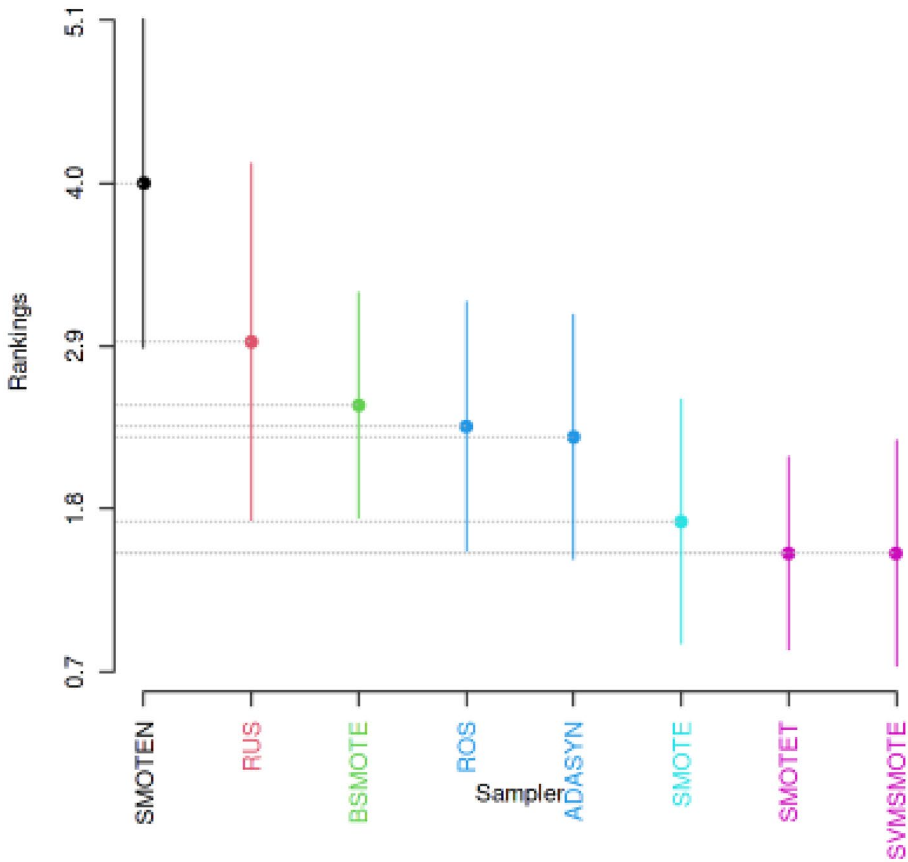

Fig. 5 SKESD test for sampling-based methods for XGBoost

External validity The datasets we applied our methods on are publicly available. The datasets are 12 Android apps developed in the Java programming language. As described by Zhao et al. (2021b), we have not evaluated whether the methods are suitable for Android apps developed in other languages (i.e., Kotlin) or IoS apps. Additionally, further studies must investigate whether our methods can be applied to other domains of JIT defect prediction. Our study also assumes the adoption of models with unbalanced datasets. If balanced datasets are adopted, other metrics, such as precision at recall, might be preferred. Furthermore, the TabNet model might perform significantly better with a balanced dataset. Additionally, as more and more data is collected, the use of DL methods will become increasingly relevant with increasing dataset sizes.

Fig. 6 SKESD test for XGBoost vs. DL models

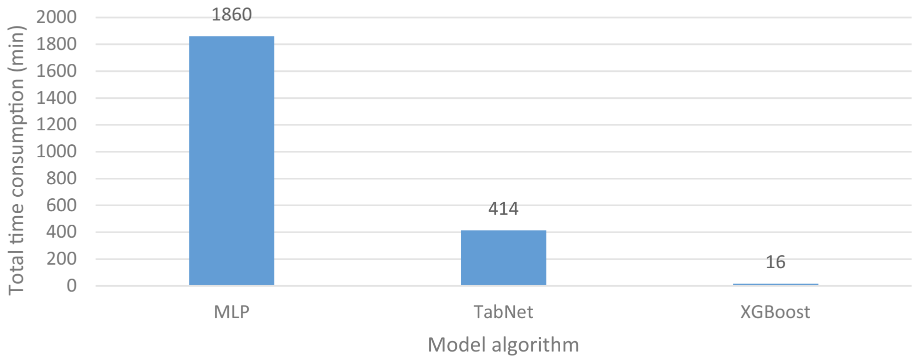

Fig. 7 The total time consumption of each machine learning algorithm for the full experiment

6 Conclusion and future work

In this study, we propose that deep neural networks are not always the optimal solution to a tabular dataset challenge. To test our hypothesis, we compare a baseline method, MLP, Tabular Network, and XGBoost. We also tested whether using eight different sampling-based methods would create significant improvements. To evaluate the effectiveness of our methodologies, we conducted experiments on 12 Android apps and used the Matthews correlation coefficient (MCC) and a Scott-Knott effect size difference statistical test.

Our results show that DL algorithms leveraging sampling methods perform significantly worse than a decision tree-based ensemble method. The XGBoost method is 116 times faster than the MLP method and has a 32% higher MCC than the baseline method. Our XGBoost pipeline takes the six input features, normalizes the features based on the training data using the StandardScaler method, and over-samples the training data using the SVMSMOTE algorithm to overcome the class imbalance challenges.

In our future work, we plan to adapt our methodology for different data sizes and different software applications. Recently, many deep learning algorithms have been developed (e.g., the transformer algorithm). These algorithms are combined in a different way for building numerous types of deep learning models. We plan to evaluate the effectiveness of these hybrid models in this problem and also investigate the efficiency of them. In addition, we will focus on the interpretable machine learning models, which are crucial in understanding the decision process of the models. Many feature engineering techniques that are available in machine learning and some of them will be also evaluated in the future work. Building deep learning models is time consuming and requires a lot of human effort; therefore, we will also focus on neural architecture search (NAS) and AutoML fields to minimize the efforts of building deep learning models. While shallow learning looks promising in this research, for different datasets, the case might be different, and therefore, several research dimensions are planned for the future.

Author contribution Conceptualization: Raymon van Dinter, Cagatay Catal; data curation: Raymon van Dinter; formal analysis: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan; investigation: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan; methodology: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan; project administration: Cagatay Catal, Bedir Tekinerdogan; resources: Raymon van Dinter, Bedir Tekinerdogan; supervision: Cagatay Catal, Bedir Tekinerdogan; validation: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan; writing—original draft: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan; writing—review and editing: Raymon van Dinter, Cagatay Catal, Gorkem Giray, Bedir Tekinerdogan.

Funding Open Access funding provided by the Qatar National Library.

Data availability Datasets are publicly available for researchers.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not

permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Alan, O., & Catal, C. (2011). Thresholds based outlier detection approach for mining class outliers: An empirical case study on software measurement datasets. *Expert Systems with Applications, 38*, 3440–3445.

Arık, S. Ö., & Le, L. T. (2020). TabNet on AI Platform: High-performance, Explainable Tabular Learning. https://cloud.google.com/blog/products/ai-machine-learning/ml-model-tabnet-is-easy-to-use-on-cloud-ai-platform/

Arık, S. Ö., & Pfister, T. (2021). Tabnet: attentive interpretable tabular learning. *AAAI Conference on Artificial Intelligence, 35*(8), 6679–6687.

Bennin, K. E., Keung, J., Phannachitta, P., Monden, A., & Mensah, S. (2017). Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. *IEEE Transactions on Software Engineering, 44*, 534–550.

Bennin, K. E., Keung, J. W., & Monden, A. (2019). On the relative value of data resampling approaches for software defect prediction. *Empirical Software Engineering, 24*, 602–636.

Brownlee, J. (2019). XGBoost with Python. Machine Learning Mastery.

Catal, C. (2014). A comparison of semi-supervised classification approaches for software defect prediction. *Journal of Intelligent Systems, 23*, 75–82.

Catal, C., & Diri, B. (2008). A fault prediction model with limited fault data to improve test process. *International Conference on Product Focused Software Process Improvement*. Springer, pp. 244–257.

Catal, C., & Diri, B. (2009). Unlabelled extra data do not always mean extra performance for semi-supervised fault prediction. *Expert Systems, 26*, 458–471.

Catal, C., Sevim, U., & Diri, B. (2010). Metrics-driven software quality prediction without prior fault data. *Electronic Engineering and Computing Technology*. Springer, pp. 189–199.

Catolino, G., Di Nucci, D., & Ferrucci, F. (2019). Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, pp. 99–110.

Catolino, G., Palomba, F., De Lucia, A., Ferrucci, F., & Zaidman, A. (2018). Enhancing change prediction models using developer-related factors. *Journal of Systems and Software, 143*, 14–28.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority oversampling technique. *Journal of Artificial Intelligence Research, 16*, 321–357.

Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., & Cho, H. (2015). Xgboost: Extreme gradient boosting. R package version 0.4–2.

Cheng, T., Zhao, K., Sun, S., Mateen, M., & Wen, J. (2022). Effort-aware cross-project just-in-time defect prediction framework for mobile apps. *Frontiers of Computer Science, 16*(6), 166207.

Giray, G. (2021). A software engineering perspective on engineering machine learning systems: State of the art and challenges. *Journal of Systems and Software, 180*, 111031.

Giray, G., Bennin, K. E., Köksal, Ö., Babur, Ö., & Tekinerdogan, B. (2023). On the use of deep learning in software defect prediction. *Journal of Systems and Software, 195*, 111537.

He, H., & Ma, Y. (2013). Imbalanced learning: Foundations, algorithms, and applications. Wiley-IEEE Press.

Huang, Q., Li, Z., & Gu, Q. (2023). Multi-task deep neural networks for just-in-time software defect prediction on mobile apps. *Concurrency and Computation: Practice and Experience,* e7664.

Jin, C. (2021). Cross-project software defect prediction based on domain adaptation learning and optimization. *Expert Systems with Applications, 171*, 114637.

Jorayeva, M., Akbulut, A., Catal, C., & Mishra, A. (2022a). Machine learning-based software defect prediction for mobile applications: A systematic literature review. *Sensors, 22*(7), 2551.

Jorayeva, M., Akbulut, A., Catal, C., & Mishra, A. (2022b). Deep learning-based defect prediction for mobile applications. *Sensors, 22*(13), 4734.

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

Kaur, A., Kaur, K., & Kaur, H. (2015). An investigation of the accuracy of code and process metrics for defect prediction of mobile applications. *2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO)(Trends and Future Directions)*. IEEE, pp. 1–6.

Kaur, A., Kaur, K., & Kaur, H. (2016). Application of machine learning on process metrics for defect prediction in mobile application. *Information Systems Design and Intelligent Applications*. Springer, pp. 81–98.

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

Lessmann, S., Baesens, B., Mues, C., & Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering, 34*, 485–496.

Li, W., Zhang, W., Jia, X., & Huang, Z. (2020). Effort-aware semi-supervised just-in-time defect prediction. *Information and Software Technology, 126*, 106364.

Mahmood, Z., Bowes, D., Lane, P. C., & Hall, T. (2015). What is the impact of imbalance on software defect prediction performance? *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 1–4.

Malhotra, R. (2016). An empirical framework for defect prediction using machine learning techniques with Android software. *Applied Soft Computing, 49*, 1034–1050.

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

Ng, A. (2017). Machine learning yearning. https://info.deeplearning.ai/machine-learning-yearning-book

Quinlan, J. R. (1986). *Induction of Decision Trees. Machine Learning, 1*, 81–106.

Ricky, M. Y., Purnomo, F., & Yulianto, B. (2016). Mobile application software defect prediction. *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, pp. 307–313.

Scandariato, R., & Walden, J. (2012). Predicting vulnerable classes in an android application. *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, pp. 11–16.

Song, Q., Guo, Y., & Shepperd, M. (2018). A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering, 45*, 1253–1269.

Statista Research Department. (2021a). Average number of new Android app releases via Google Play per month from March 2019 to August 2021.

Statista Research Department. (2021b). Number of apps available in leading app stores as of 1st quarter 2021.

Sun, Y., Jing, X.-Y., Wu, F., Dong, X., Sun, Y., & Wang, R. (2021). Semi-supervised heterogeneous defect prediction with open-source projects on GitHub. *International Journal of Software Engineering and Knowledge Engineering, 31*, 889–916.

Tantithamthavorn, C., Hassan, A. E., & Matsumoto, K. (2018). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering, 46*, 1200–1219.

Wang, K., Liu, L., Yuan, C., & Wang, Z. (2021). Software defect prediction model based on LASSO–SVM. *Neural Computing and Applications, 33*, 8249–8259.

Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. *IEEE Transactions on Reliability, 62*, 434–443.

Wu, F., Jing, X.-Y., Dong, X., Cao, J., Xu, M., Zhang, H., Ying, S., & Xu, B. (2017). Cross-project and within-project semi-supervised software defect prediction problems study using a unified solution. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 195–197.

Xu, Z., Li, S., Xu, J., Liu, J., Luo, X., Zhang, Y., Zhang, T., Keung, J., & Tang, Y. (2019). LDFR: Learning deep feature representation for software defect prediction. *Journal of Systems and Software, 158*, 110402.

Yang, X., Lo, D., Xia, X., Zhang, Y., & Sun, J. (2015). Deep learning for just-in-time defect prediction. *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, pp. 17–26.

Yao, J., & Shepperd, M. (2020). Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters. *Proceedings of the Evaluation and Assessment in Software Engineering*, pp. 120–129.

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

Zhang, Z.-W., Jing, X.-Y., & Wang, T.-J. (2017). Label propagation based semi-supervised learning for software defect prediction. *Automated Software Engineering, 24*, 47–69.

Zhao, K., Liu, J., Xu, Z., Li, L., Yan, M., Yu, J., & Zhou, Y. (2021a). Predicting crash fault residence via simplified deep forest based on a reduced feature set. arXiv preprint arXiv:2104.01768

Zhao, K., Xu, Z., Yan, M., Tang, Y., Fan, M., & Catolino, G. (2021b). Just-in-time defect prediction for Android apps via imbalanced deep learning model. *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, pp. 1447–1454.

Zhao, K., Xu, Z., Zhang, T., Tang, Y., & Yan, M. (2021c). Simplified deep forest model based just-in-time defect prediction for Android mobile apps. *IEEE Transactions on Reliability*.

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

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Raymon van Dinter is a Software Engineer at Sioux Technologies in Apeldoorn, The Netherlands. As part of his work, he pursues PhD research at the Information Technologies group at Wageningen University. He obtained his BSc in Electrical Engineering at HAN University of Applied Sciences in 2019, and his MSc in Farm Technology at Wageningen University in 2021. He is experienced with Software Engineering and Design and Data Science. At Sioux Technologies, he worked on projects for various clients in Medical, Energy, and Semiconductor domains. His research interests include Software Engineering, Software Architecture, Predictive Maintenance, Machine Learning, and Deep Learning.

Cagatay Catal is a Full Professor at the Department of Computer Science & Engineering in Qatar University. He obtained his BSc and MSc degrees in Computer Engineering from Istanbul Technical University in 2002 and 2004, and the Ph.D. degree in Computer Engineering from Yildiz Technical University in Istanbul in 2008. He worked as a Full Professor at the Department of Computer Engineering in Bahcesehir University in Istanbul between 2020 and 2021. He worked for 2 years as a full-time faculty member at Wageningen University & Research (WUR) in the Netherlands before joining Bahcesehir University. Before WUR, he worked 6 years in the department of Computer Engineering at Istanbul Kultur University as an Associate Professor and the Head of the Department. He received the Associate Professor title in April 2014 from Inter-University Council of Turkey. Before joining the university, he worked 8 years at the Scientific and Technological Research Council of Turkey (TUBITAK), Information Technologies Institute as a Senior Researcher and Project Manager. His research interests include deep learning, machine learning, natural language processing, software engineering, software testing, and software architecture.

Görkem Giray is a software engineer, a researcher, and a part-time lecturer. He has been working in the industry for more than 20 years and pursuing an executive level position in recent years in a multinational company. In addition, he has been conducting research in software engineering and semantic web (knowledge graphs) and delivering software engineering courses at universities. Giray received a B.Sc. (1999) and a Ph.D. (2011) in computer engineering from Ege University. He holds an MBA degree (2001) from Koç University. More details can be found on his LinkedIn profile: https://www.linkedin.com/in/gorkemgiray/.

Bedir Tekinerdogan is currently a full professor and chair of the Information Technology group at Wageningen University in The Netherlands. He more than 25 years of experience in software/systems engineering and information technology and has authored more than 400 peer-reviewed scientific papers and 8 edited books. He has been involved in dozens of national and international research and consultancy projects with various large software companies as a principal researcher and leading software/system architect. His experience spans a broad range of domains, including consumer electronics, enterprise systems, automotive systems, critical infrastructures, cyber-physical systems, satellite systems, defense systems, production line systems, command and control systems, physical protection systems, radar systems, smart metering systems, energy systems, and precision farming. He takes a holistic, systemic, and multi disciplinary approach to solve real industrial problems and has ample experience in software and systems architecting, software and systems product line engineering, cyber-physical systems, model-driven software engineering, aspectoriented software engineering, global software development, systems engineering, system of systems engineering, data science, and artificial intelligence. In addition to his research, he is an active educator. He has developed and taught around 20 different academic courses and provided many software/systems engineering courses to more than 50 companies in The Netherlands, Germany, India, and Turkey.