ELSEVIER

The Journal of SYSTEMS and SOFTWARE

REPD: Source code defect prediction as anomaly detection

ARTICLE INFO

Article history:
Received 13 September 2019
Received in revised form 27 April 2020
Accepted 11 May 2020
Available online 15 May 2020

Keywords:
Defect prediction
Anomaly detection
REPD
Program analysis

ABSTRACT

In this paper, we present a novel approach for within-project source code defect prediction. Since defect prediction datasets are typically imbalanced, and there are few defective examples, we treat defect prediction as anomaly detection. We present our **R**econstruction **E**rror **P**robability **D**istribution (REPD) model which can handle point and collective anomalies. We compare it on five different traditional code feature datasets against five models: Gaussian Naive Bayes, logistic regression, k-nearest-neighbors, decision tree, and Hybrid SMOTE-Ensemble. In addition, REPD is compared on 24 semantic features datasets against previously mentioned models. In order to compare the performance of competing models, we utilize F1-score measure. By using statistical means, we show that our model produces significantly better results, improving F1-score up to 7.12%. Additionally, REPD's robustness to dataset imbalance is analyzed by creating defect undersampled and non-defect oversampled datasets.

1. Introduction

Software defects, colloquially called *bugs*, can cause massive problems leading to great financial loss, and in some cases even critical safety issues. Traditional ways of combating software defects, such as testing and code review, require large amounts of time and specialized teams to be performed thoroughly. In order to reduce the needed resources, defect prediction tries to automate defect discovery and thus minimize operational costs, while improving software quality. Defect prediction is often treated as a supervised binary classification problem. It often suffers from the common problem of imbalanced datasets, as can be seen from Tan et al. (2015), Bennin et al. (2018) and Yu et al. (2017). In practical circumstances, datasets often contain fewer defective instances then non-defective instances.

From our coding experience, we know defects often occur because of a lack of understanding of the code worked on, changing and conflicting requirements, and interruptions to the coding process. All these reasons can be considered anomalies to the healthy coding process. Thus, the resulting defects could be thought of as anomalies inside healthy code. Seeing defects as anomalies easily explains why defect prediction datasets often suffer from the dataset imbalance problem. Anomalies are less frequent than normal examples. Motivated to investigate this approach and to combat the dataset imbalance problem, we decided to treat source code defect prediction as an anomaly detection problem in which defects represent anomalies. Anomalies can be divided into three types: *point anomalies*, single instances which are too different from the general population to be considered normal; *contextual anomalies*, which are context specific, and common in time-series data; and *collective anomalies*, a set of data instances which can collectively be considered anomalies.

In the present work, we develop a supervised anomaly detection/classification model, which we call ***R**econstruction **E**rror **P**robability **D**istribution* (REPD) which can handle point and collective anomalies. It cannot handle contextual anomalies, i.e. data instances which are seen as anomalies only in a certain context, because the model does not have a sense of context. To assess the effectiveness of the developed model in practice, we collected several commonly used traditional code feature datasets and prepared several sets of semantic code features. We compare our model to five classifiers: Gaussian Naive Bayes, logistic regression, k-nearest-neighbors, decision tree, and Hybrid SMOTE-Ensemble. The first four are standard well known classifiers, and Hybrid SMOTE-Ensemble is a state-of-the-art defect prediction classifier presented in Alsawalqah et al. (2017). These classifiers were chosen because of their frequent use (Menzies et al., 2007; Denaro, 2000; Khoshgoftaar et al., 2005; Khoshgoftaar and Seliya, 2002b; Kim et al., 2008; Giger et al., 2012; Koru and Liu, 2005; Wang et al., 2016; Tan et al., 2015) and their easily accessible or reproducible implementations. Using statistical analysis we compare mean performance samples of our model to other models and quantify the significance of the mean performance difference,

☆ This research has been partly supported by the European Regional Development Fund under the grant KK.01.1.1.01.0009 (DATACROSS) and under the grant KK.01.2.1.01.0111 (OperOSS). The authors acknowledge the support of the Croatian Science Foundation for this research through the Reliable Composite Applications Based on Web Services (IP-01-2018-6423) research project. The Titan X Pascal used for this research was donated by the NVIDIA Corporation.

* Corresponding author.
E-mail addresses: petar.afric@fer.hr (P. Afric), lucija.sikic@fer.hr (L. Sikic), adrian.kurdija@fer.hr (A.S. Kurdija), marin.silic@fer.hr (M. Silic).

https://doi.org/10.1016/j.jss.2020.110641

showing that our model has superior performance of a large significance. We demonstrate the improvement of F1-score by up to 7.12% when compared to competing approaches. Finally, we test the model robustness when it comes to handling imbalanced datasets.

The rest of this paper is organized as follows. Section 2 gives a more detailed introduction to the problem of source code defect prediction and an overview of related work. In Section 3 we describe our model. In Section 4, we present the datasets we used for evaluation, the evaluation methodology, and our evaluation results. Section 5 outlines threats to the validity of the presented work. Finally, in Section 6 we summarize our achievements and give guidelines for future work.

2. Background and related work

Software defect prediction is a field of study which tries to identify causality between software features and defective software. More precisely, the aim is to develop the capability of classifying code as defective or non-defective, given a set of features describing the code. This prediction can be done at different levels: at *change level* (Kim et al., 2008; Hassan, 2009; Kim et al., 2011; Prechelt and Pepper, 2014; Kim et al., 2006), at *method level* (Giger et al., 2012; Koru and Liu, 2005), at *file level* (Meneely et al., 2008; Bettenburg et al., 2012; Kim et al., 2007; Moser et al., 2008; Ostrand et al., 2005; Zhang, 2009), or at *component level* (Menzies et al., 2010; Graves et al., 2000; Nagappan et al., 2006; Cheung et al., 2008; Zhang et al., 2007). Defect prediction can also be divided into within-project prediction and cross-project prediction. In cross-project prediction, the classifier is trained on project *A* and then applied to detect defects in project *B*. In within-project prediction, the aim is to train a classifier on a set of data derived from a project and use the classifier on the same project in order to predict defects. Within-project prediction can further be divided into inner-version and cross-version defect prediction. When using inner-version defect prediction, the classifier is trained and used on the same version of the same project. When using cross-version defect prediction, the classifier is trained on an older version of the project, and then used to predict defects in newer project versions.

Datasets for training defect prediction are acquired by mining code repositories, version control systems, and bug tracking systems. Commits are correlated with bug fixes, the code edited by the bug fix is identified, and its introduction point is identified by searching the version control system. States of the code changed by the bug fix are labeled as defective, while all others are labeled as non-defective.

Many different kinds of features have been proposed for the purpose of defect prediction. Most of the features are manually designed: Halsted features (Halstead, 1977) which are based on operator and operand occurrences; McCabe features (McCabe, 1976) which are based of dependencies; CK features (Chidamber and Kemerer, 1994) based on functions and inheritance counts; MOOD features (Harrison et al., 1998) based on polymorphism factors, coupling factors, etc.; change features (Jiang et al., 2013); and object-oriented features (e Abreu and Rogério, 1994). We refer to these manually designed features as classical features. A more detailed presentation of these features is given in the Appendix. There is some interesting work in automatically learning semantic features for defect prediction (Wang et al., 2016).

Given how active and attractive the field of defect prediction is, there have been decades of work done in it. Many researchers have proposed different approaches for defect prediction. The most common approaches include using some supervised model in order to learn defect prediction (Graves et al., 2000; Hassan, 2009; Hall et al., 2012; Jiang et al., 2013; Khoshgoftaar and Seliya, 2002a; Khoshgoftaar et al., 2000; Lessmann et al., 2008; Jing et al., 2014). There have also been attempts at using unsupervised models for defect prediction (Yang et al., 2016; Fu and Menzies, 2017). However, we have come across only one paper which deals with defect prediction as an anomaly detection problem (Neela et al., 2017). Kamrun et al. approach this problem as an anomaly detection problem with similar intuition and motivation as us, but their approach is completely different. It performs feature selection as suggested by Menzies et al. (2007), and then models non-defective instances using a Univariate (Koh, 2014) or a Multivariate Gaussian distribution (Flury, 1997). The test instances are then classified as defective if they are too far from the calculated distribution. Their results are quite promising and, in addition to our own, contribute to the idea that treating defect prediction as anomaly detection is a fruitful idea. To the best of our knowledge, we are the first to propose a model such as REPD, which is described in detail in the following section.

3. Model description

In this section, we present the proposed REPD model for source code defect prediction. In order to detect defects in software artifacts, we employ the autoencoder neural network architecture which demonstrated quite effective results in anomaly detection tasks (Sakurada and Yairi, 2014). The autoencoder is a feed forward neural network where input and output layers have the same size. The network learns to reconstruct the input as closely as possible while performing data compression in the hidden layers. Compression is achieved by making the hidden layers smaller that the visible layers, thus forcing the network to choose data which best describes the input. The REPD model assumes that the autoencoder, which is trained to reconstruct features of non-defective examples, will manifest difference in reconstruction error for defective and non-defective examples. In order to quantify the difference in reconstruction error, we estimate the probability density function separately for non-defective and defective software artifacts. The reconstruction error of an unseen example will have a certain probability of belonging to the negative reconstruction error distribution, and a certain probability of belonging to the positive reconstruction error distribution. Based on those probabilities, we classify the examples as non-defective or defective. The decision to use an autoencoder was made because of its property to learn encoding and decoding from the presented data, thus allowing the data to speak for itself. To make it clear, the outputs of the autoencoder are not new features, but a form of mid-result acquired during data processing.

The high-level overview of the model is depicted in Fig. 1. As can be seen in the figure, interaction with the model consists of three phases: *training phase, distribution determination phase,* and *usage phase.* In the training phase, we train the autoencoder on negative examples, so it can reconstruct features of non-defective software artifacts. During the distribution determination phase, we use both negative and positive examples to estimate the probability density function of the reconstruction error, separately for negative and positive examples. Finally, in the usage phase, we use the created model to classify the unseen examples. Each of the phases is described in more details in the following subsections.

3.1. Training phase

The first step of the training phase is creating feature representations from the source code. The REPD model requires numeric features, but imposes no restrictions on them. An autoencoder is trained to reconstruct non-defective examples. It consists of two

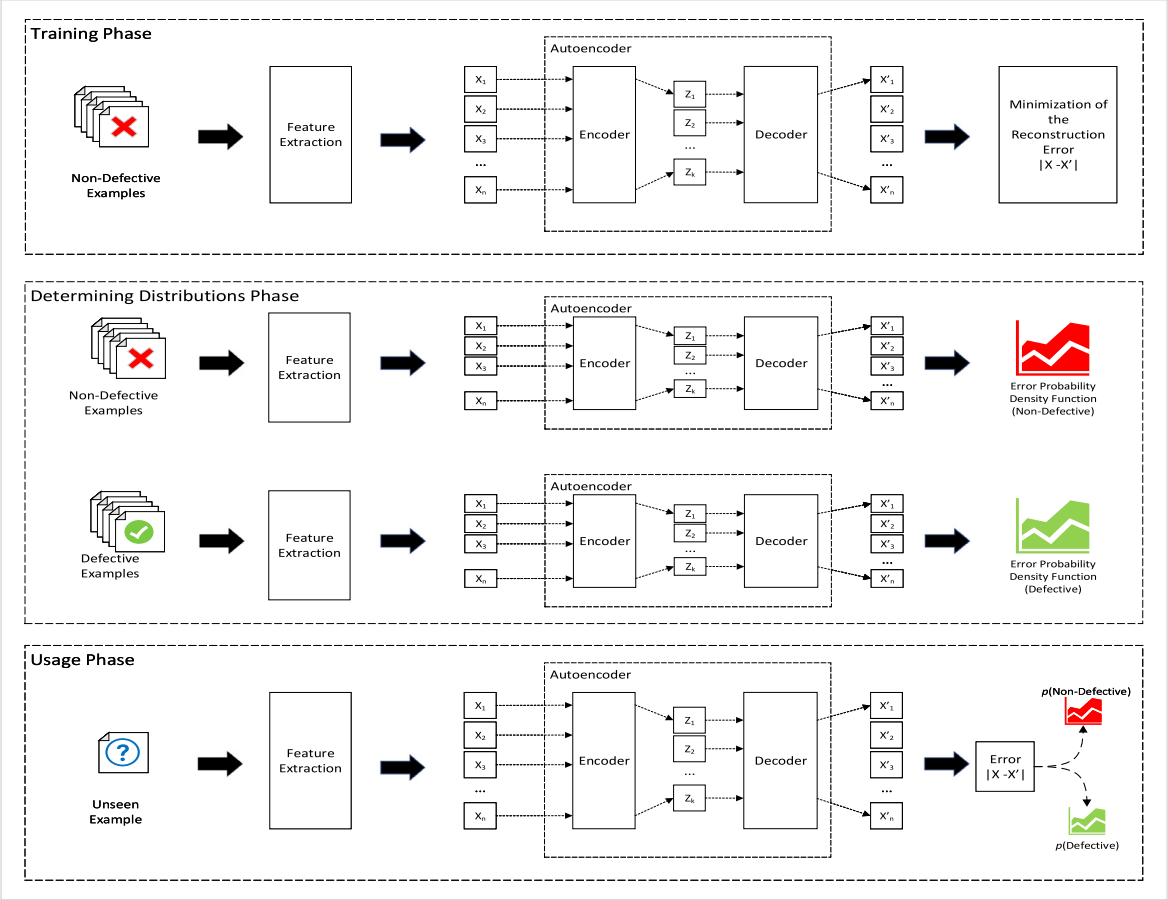


Fig. 1. Training the REPD model.

layers, the visible layer and the hidden layer. The visible layer is the size of the input examples, while the hidden layer is half the size of the visible layer. The autoencoder first encodes an example by mapping it from the visible layer to the hidden layer, and then decodes it by mapping the hidden layer representation back to the visible layer. In this way a reconstruction is obtained. The aim is to make this reconstruction as similar as possible to the original example.

More precisely, an autoencoder consists of the *encoder* and the *decoder*, which are (respectively) a transition $\phi : \mathcal{X} \to \mathcal{F}$ from the input space $\mathcal{X}$ to the feature space $\mathcal{F}$, and $\psi : \mathcal{F} \to \mathcal{X}$ the other way around. They are trained to minimize the reconstruction error:

$$\phi, \psi = \arg\min_{\phi,\psi} \|X - (\psi(\phi(X)))\|^2. \tag{1}$$

In our case,

$$\phi(\mathbf{x}) = \mathbf{z} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}), \tag{2}$$

$$\psi(\mathbf{z}) = \mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}'), \tag{3}$$

where $\mathbf{W}$, $\mathbf{W}'$ are weight matrices and $\mathbf{b}$, $\mathbf{b}'$ are bias vectors. These parameters are learned to minimize the squared loss:

$$\begin{aligned} \mathcal{L}(\mathbf{x}, \mathbf{x}') &= \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma'(\mathbf{W}'\mathbf{z} + \mathbf{b}')\|^2 \\ &= \|\mathbf{x} - \sigma'(\mathbf{W}'(\sigma(\mathbf{W}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2. \end{aligned} \tag{4}$$

In this way, a root mean square error (RMSE) between the original input and the reconstruction is calculated. This RMSE is referred to as the reconstruction error. The autoencoder is trained by minimizing reconstruction error using the backpropagation algorithm with the Adam optimizer (Kingma and Ba, 2014), while for weights initialization Xavier initializer (Glorot and Bengio, 2010) is used. After training, the autoencoder is used to calculate the reconstruction errors of defective and non-defective instances. The simple two-layer architecture was chosen because of its simplicity and speed. There was no thorough optimization of the autoencoder architecture.

3.2. Distribution determination phase

In the distribution determination phase, probability distributions of reconstruction errors for defective and non-defective examples are determined. The same procedure is performed for reconstruction errors of defective and non-defective examples. An assumption is made that the errors come from one of the following distributions: normal (Singh, 1998), lognorm (Hart, 1990), exponentiated Weibull continuous random variable (Pal et al., 2003), pareto (Arnold, 2015), gamma (Dubey, 1970), or beta (Dubey, 1970). Each of the distributions is fitted to the errors. Then a Kolmogorov–Smirnov test (Dodge, 2008) is used to determine which of the distributions best describes the reconstruction errors.

More precisely, each of the aforementioned probability distributions is fitted to the observed errors by a maximum likelihood estimation (MLE) method (using the *SciPy* library Virtanen et al., 2020). MLE is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most

probable. When such a distribution $F(x)$ is found, a Kolmogorov–Smirnov (KS) test is used to measure the actual fit. The KS test first computes the empirical distribution function:

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} I_{[-\infty,x]}(X_i), \tag{5}$$

where $X_1, \ldots, X_n$ are the observed errors and $I_{[-\infty,x]}(X_i) = 1$ if $X_i \leq x$, and 0 otherwise. It is then compared to the distribution F by the Kolmogorov–Smirnov statistic (the smaller the better):

$$D_n = \sup_x |F_n(x) - F(x)|. \tag{6}$$

The probability distribution with the smallest value of D_n is chosen as the best fit to the reconstruction errors.

After performing this procedure, two distributions are obtained: one describing defective reconstruction errors, and the other one describing non-defective reconstruction errors. The pseudocode of the distribution determination phase is shown in Algorithm 1.

Algorithm 1 REPD model: determining distribution

Input data
Data instances: X_defective, X_non_defective
Data labels: y
Data coding model: autoencoder

Results
Non defective error distribution: non_def_dist
Defective error distribution: def_dist

Procedure
```
non_def_errors =
        autoencoder.reconstruction_error(X_non_defective)
def_errors =
        autoencoder.reconstruction_error(X_defective)
non_def_distributions = Ø
def_distributions = Ø
for distribution ← potential_distributions do
  non_def_distributions = non_def_distributions ∪
        distribution.new_instance().fit(non_def_errors)
  def_distributions = def_distributions ∪
        distribution.new_instance().fit(def_errors)
end for
non_def_dist = KS_test_get_best(non_def_distributions)
def_dist = KS_test_get_best(def_distributions)
```

3.3. Usage phase

Int the usage phase, a trained model is used to predict if the unseen instances are defective or not. For an unseen instances, their reconstruction error is calculated using the autoencoder. The reconstruction error is then tested against the two distributions obtained in the previous phase and the probability of belonging to each distribution is estimated as the value of the corresponding probability density function $PDF(error)$. The new instance is classified as defective if its reconstruction error has a higher probability of belonging to the defective reconstruction errors distribution than to the non-defective one:

$$PDF_{defective}(error) > PDF_{non-defective}(error). \tag{7}$$

Otherwise, it is classified as non-defective. The usage pseudocode is shown in Algorithm 2.

Algorithm 2 REPD model: usage pseudocode

Input data
Unclassified instance: X

Results
Instance class: non_defective *or* defective

Procedure
```
p_nd = non_def_dist.probability_density_function(X)
p_d = def_dist.probability_density_function(X)
if p_nd > p_d then
  return non_defective
else
  return defective
end if
```

Table 1
Information about traditional datasets.

Dataset	Language	Dataset size	Defective count	Defective share
CM1	C	498	49	10.86%
JM1	C	10 885	2106	22.5%
KC1	C++	2109	326	25.99%
KC2	C++	522	105	28.0%
PC1	C	1109	77	7.34%

4. Evaluation

In this section, we present datasets used for evaluation, our evaluation methodology, and the achieved evaluation results. In order to encourage other researchers to further validate our model, we have made the implementation open source and available at Afric et al. (2019b).

In Section 4.1, we describe the datasets used in our evaluation. Section 4.2 presents the evaluation methodology and the results of the competing approaches by using traditional code features datasets, while Section 4.3 describes the evaluation methodology and the results for the competing approaches by using semantic code features datasets. Finally, Section 4.4 assesses the robustness of the competing approaches against the imbalance dataset problem.

4.1. Datasets used for evaluation

In our experiments, we used an amount of five datasets which all have 21 features and are based on Halstead and McCabe features such as operator, operation, and dependency count. These datasets were created by mining modules of NASA products. Namely, we used CM1, JM1, KC1, KC2, and PC1 datasets available from the PROMISE dataset repository (Sayyad Shirabad and Menzies, 2005). Basic information about these traditional datasets is presented in Table 1.

To further test the validity and performance of our model, we have downloaded the traditional defect prediction datasets for the following open source projects: ant $v1.5$, ant $v1.6$, camel $v1.2$, camel $v1.4$, log4j $v1.1$, log4j $v1.2$, poi $v2.0$, and poi $v2.5$ (Tantithamthavorn, 2019a). We then downloaded the source code of those projects and their specific versions. For each of the classes in the defect prediction datasets, we created a lexical token representation, while limiting ourselves to lexical tokens representing keywords. Individual tokens were represented by integers, and to ensure the same length of all token representations, zeros were appended to the shorter ones. We used three different neural networks to create three different semantic feature based datasets for each of the original datasets. We used a deep belief

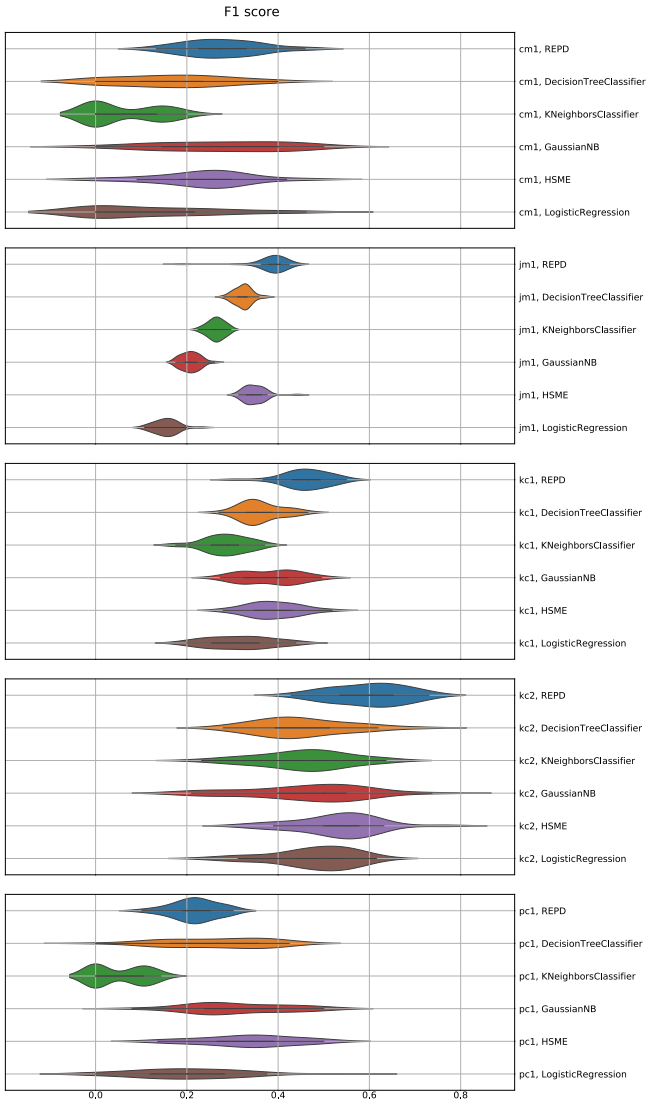


Fig. 2. F1-scores achieved by different models on different datasets.

network, a deep autoencoder, and a convolutional autoencoder, encoding each of the sequences to have 100 features. In this way we have created 24 semantic feature based datasets for further evaluation of our model. To make sure that the results are not influenced by unfortunate feature encoding, it was actually done 30 times with different random seeds for each dataset and each feature type, resulting in 720 datasets. The choice of 30, as in several other cases during our analysis, was made following the general rule of thumb that it would allow a statistically adequate observation to employ the benefits of the central limit theorem. However, the performances of the trained models are presented as the average performance per dataset, per feature type.

All used open-source projects are written in Java. Since the previously mentioned projects are written in C and C++, this adds an extra layer of confidence that our approach is efficient in practice. Basic information about semantic datasets based on these modules is presented in Table 2.

We decided to use different datasets for semantic features because they come from a different programming language, which brings further validation to our model.

Table 2
Information about semantic datasets.

Dataset	Dataset size	Defective count	Defective share
ant $v1.5$	292	32	10.96%
ant $v1.6$	350	92	26.29%
camel $v1.2$	595	216	36.30%
camel $v1.4$	846	145	17.14%
log4j $v1.1$	104	37	35.58%
log4j $v1.2$	194	186	95.88%
poi $v2.0$	309	37	11.97%
poi $v2.5$	380	248	62.26%

4.2. Model performance evaluation for traditional code features datasets

In this section we describe the methodology used to evaluate the competing approaches on traditional code features datasets followed by the obtained evaluation results. To evaluate the performance of our model on traditional code features datasets, we compare it to five models: Gaussian Naive Bayes, logistic regression, k-nearest-neighbors, decision tree, and Hybrid SMOTE-Ensemble. Training and evaluation of our model and each of the alternative models was done 100 times. As the main performance indicator we use F1-score. On those datasets where our REPD model has the highest average F1-score, the following statistical analysis is performed to compare our model to an alternative model:

(1) We run a normality test (Madansky, 1988) on the 100 performance samples of our model and on the 100 performance samples of the alternative model. The null hypothesis is that the samples come from a normal distribution. If we do not manage to reject the null hypothesis, we act as if it is proven. We use a p-value threshold of 0.001. In this instance, we make a liberal choice of a small p-value in order to relax our criteria for a normal distribution. The reasoning comes from the central limit theorem, which states that a distribution of sample means approximates a normal distribution as the sample size becomes larger. Therefore, knowing that our data should approach a normal distribution as the sample size grows, instead of generating a huge sample size, we save the time by relaxing criteria for a normal distribution.

(2) If both the samples of our model and of the alternative model have passed the normality test, we use a t-test (Cleophas and Zwinderman, 2011) to show there is a difference between the two distributions. The null hypothesis is that the two samples come from the distributions with the same average. Our aim is to reject the null hypothesis and thus show there is a difference in the average performance of the models. We use a p-value threshold of 0.05. This is a standard p-value used in statistical analysis.

(3) If we have managed to show that there is a difference in the average performance of our model and the alternative model by rejecting the null hypothesis of the t-test, we attempt to quantify that difference. This is done using Choen's d-test (Hill and Thompson, 2005). Cohen's d-value is an effect size used to indicate the standardized difference between two means. We say the effect size is:

- trivial if $d\text{-}value < 0.2$,
- small if $0.2 \leq d\text{-}value < 0.5$,
- moderate if $0.5 \leq d\text{-}value < 0.8$,
- large if $0.8 \leq d\text{-}value$.

To avoid confusion, statistical analysis was not performed on datasets where our model performs worse, because then (as our

Table 3
REPD comparison on CM1 dataset.

Model	Is normal	Normal-test p-value	Different mean values: t-test	t-test p-value	d-value	Effect interpretation
GaussianNB	Yes	0.2904	No	9.5810e−01		
LogisticRegression	Yes	0.1327	Yes	6.4000e−06	1.2817	Large
KNeighborsClassifier	No	0.0000				
DecisionTreeClassifier	Yes	0.4691	Yes	1.0882e−04	1.0725	Large
HSME	Yes	0.4470	No	9.5736e−02		

Table 4
REPD comparison on JM1 dataset.

Model	Is normal	Normal-test p-value	Different mean values: t-test	t-test p-value	d-value	Effect interpretation
GaussianNB	Yes	0.6869	Yes	2.7569e−28	5.3013	Large
LogisticRegression	Yes	0.1304	Yes	8.1836e−33	6.4732	Large
KNeighborsClassifier	Yes	0.8308	Yes	1.3613e−20	3.6776	Large
DecisionTreeClassifier	Yes	0.6142	Yes	1.6015e−09	1.8477	Large
HSME	No	0.0000				

Table 5
REPD comparison on KC1 dataset.

Model	Is normal	Normal-test p-value	Different mean values: t-test	t-test p-value	d-value	Effect interpretation
GaussianNB	Yes	0.1638	Yes	2.4800e−06	1.3489	Large
LogisticRegression	Yes	0.3268	Yes	5.3244e−13	2.3860	Large
KNeighborsClassifier	Yes	0.5832	Yes	1.3882e−19	3.4918	Large
DecisionTreeClassifier	Yes	0.4055	Yes	7.7310e−11	2.0500	Large
HSME	Yes	0.8862	Yes	3.8844e−06	1.3172	Large

Table 6
REPD comparison on KC2 dataset.

Model	Is normal	Normal-test p-value	Different mean values: t-test	t-test p-value	d-value	Effect interpretation
GaussianNB	Yes	0.5634	Yes	7.8509e−06	1.2671	Large
LogisticRegression	Yes	0.1290	Yes	1.0428e−06	1.4095	Large
KNeighborsClassifier	Yes	0.8052	Yes	9.2545e−08	1.5755	Large
DecisionTreeClassifier	Yes	0.3227	Yes	8.1248e−08	1.5843	Large
HSME	Yes	0.4648	Yes	3.0167e−03	0.7995	Moderate

model is not the worst) some effect size interpretations would indicate a significant improving difference and some would indicate a significant impairing difference.

Fig. 2 depicts the evaluation results obtained by the aforementioned process. More specifically, Fig. 2 shows F1-scores obtained on the traditional code features datasets. By a quick visual inspection of the presented results, we see the competitive performance of the REPD model. It has the highest average F1-score on 4/5 datasets.

On those datasets, statistical analysis of the presented results is laid out in the following tables which are divided by datasets:

- CM1 dataset statistical results are shown in Table 3.
- JM1 dataset statistical results are shown in Table 4.
- KC1 dataset statistical results are shown in Table 5.
- KC2 dataset statistical results are shown in Table 6.

In these tables, the first column (*Model*) specifies the model which is being compared to the REPD model. The second column (*Is normal*) specifies the result of the normality test which is used to check if the performance measurements of the model have a normal distribution. The third column (*normal-test p-value*) specifies the *p*-value of the normal test. If this value is below the threshold, the following columns are empty. The fourth column (*Different mean values: t-test*) specifies whether the performance mean values of the model under comparison and REPD are different. The fifth column (*t-test p-value*) specifies the *p*-value of the t-test. The sixth column (*d-value*) specifies the effect size value of the Cohen's d-test. The last column (*effect interpretation*) specifies whether the difference between values is significant and to which extent.

We can see from the presented results and analysis that our model, in a majority of cases, has the best average performance and that its improvement over other models is statistically of large significance. F1-score improvement ranges from 0.16% to 7.12% depending on the dataset. This increase in the F1-score mainly comes from Recall improvement. REPD's Recall on these datasets ranges from 0.4838 to 0.6971. This means that, from a practical point of view, REPD is a very effective early defect detection system, identifying, in most cases, more than half of newly introduced errors. We believe the reason it underperforms on the PC1 dataset is that in this dataset there is no fundamental difference between defective and non-defective examples. Since this is a key premise for our model, violating it will cause the model to underperform.

Fig. 3 depicts the recall and Fig. 4 depicts the precision metric obtained by the evaluation process. These figures show that the proposed model has an inclination towards defect identification even at the expense of precision.

4.3. Model performance evaluation for semantic feature datasets

In order to further evaluate the performance and validity of our model, we also evaluate it on previously described semantic feature based datasets. Again, we use the same models for comparison.

As previously mentioned, for each dataset and each feature encoding (deep belief network, deep autoencoder, and convolutional autoencoder) 30 datasets were created, resulting in a total of 720 datasets. For each of the created datasets, we trained and evaluated each model 30 times, which adds up to 21 600 evaluations per model or 108 000 evaluations in total. When analyzing and comparing performance, we group the results by dataset, feature encoding, and of course, model. Again, we used the F1-score as the main performance indicator.

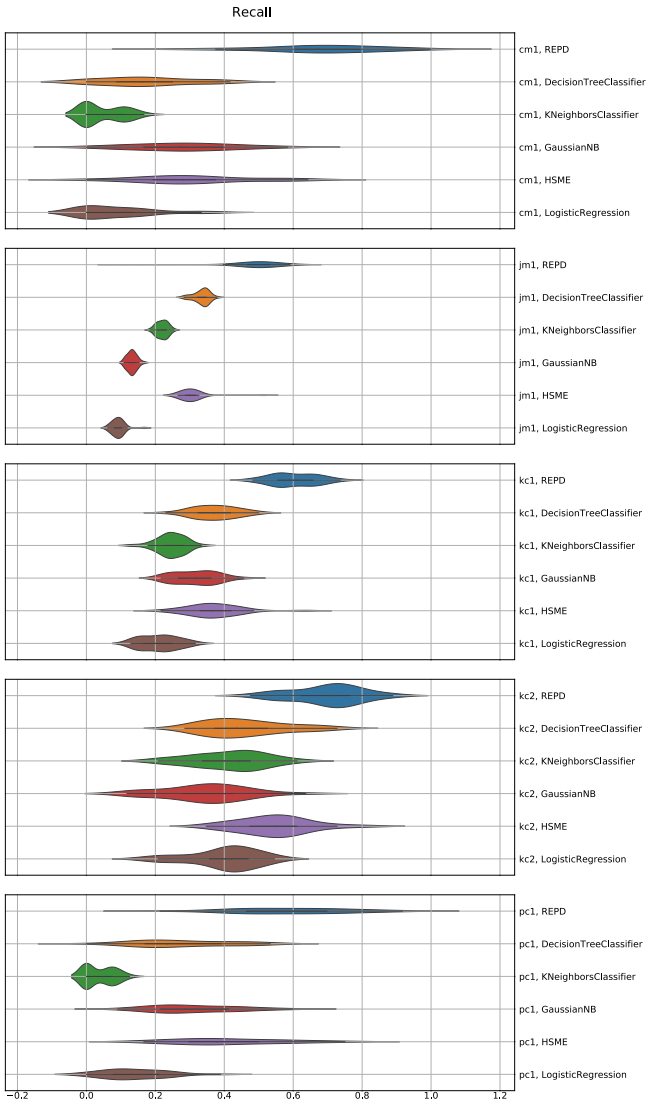


Fig. 3. Recall scores achieved by different models on different datasets.

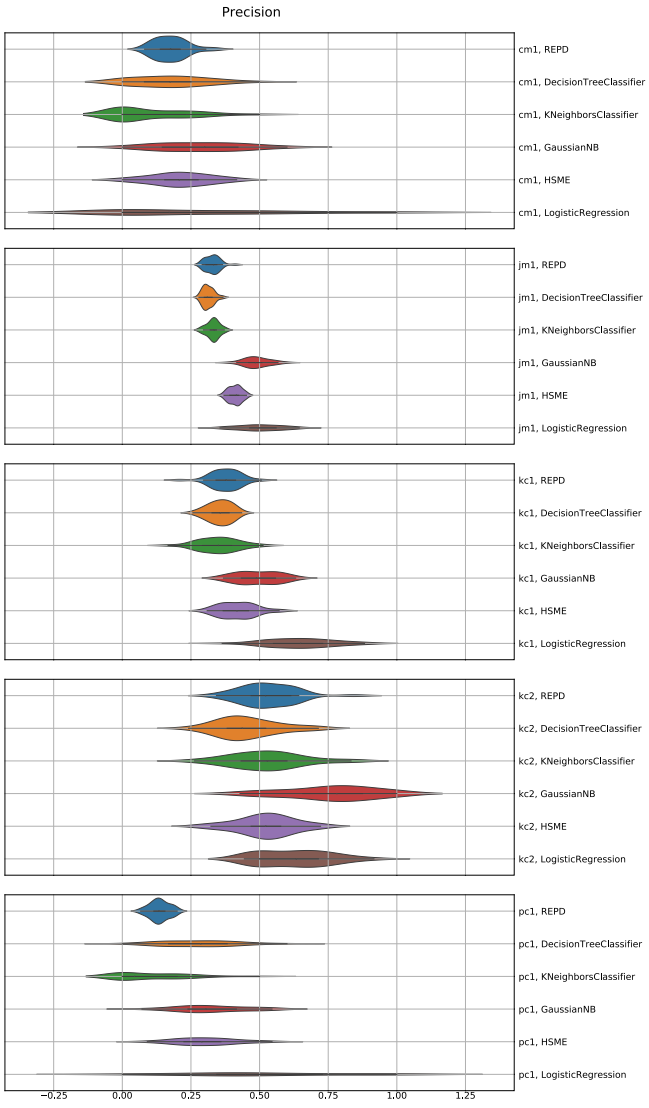


Fig. 4. Precision scores achieved by different models on different datasets.

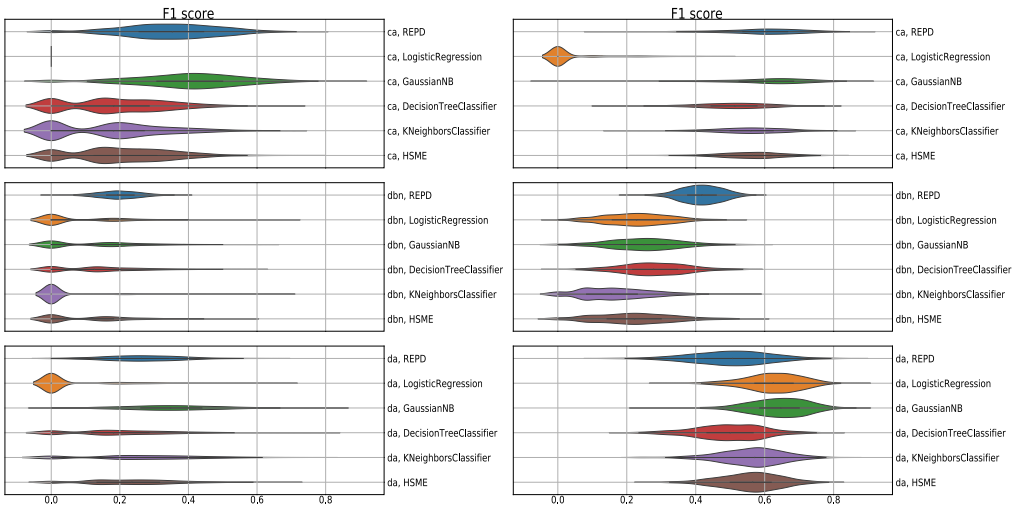
The evaluation results obtained in the described process are presented in Figs. 5–8. In Figs. 5–8 we used abbreviations: *dbn* - deep belief network, *da* - deep autoencoder, *ca* - convolution autoencoder.

Fig. 5 shows F1-scores, Recall and Precision obtained on the semantic datasets based on *ant* $v1.5$ and $v1.6$. When comparing the mean performance by F1-scores values on ant $v1.5$, we see that REPD is the second best model on the convolution autoencoder and deep autoencoder created features, and the best model on the deep belief network created features. When comparing the mean performance by F1-scores values on ant $v1.6$, we see that REPD is the best model on the convolution autoencoder and deep belief network created features. On the deep autoencoder created features it does not perform as well.

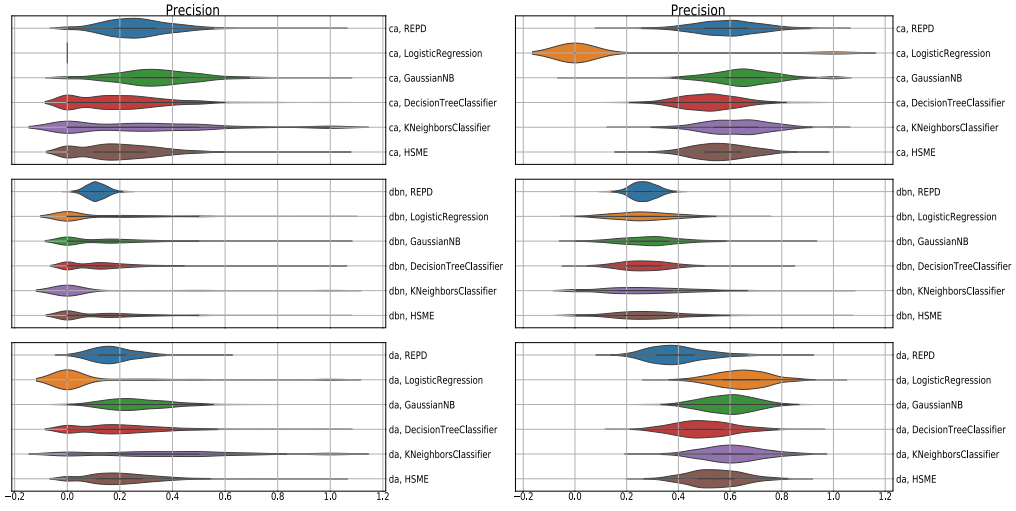
Fig. 6 shows F1-scores, Recall and Precision obtained on the semantic datasets based on *camel* $v1.2$ and $v1.4$. When comparing the mean performance by F1-scores values on camel $v1.2$, we see that REPD is the best performing model on the deep autoencoder and deep belief network created features, and the third best model on the convolutional autoencoder created features. When comparing the mean performance by F1-scores values on camel $v1.4$, we see that REPD is the best performing model on the convolutional autoencoder and deep belief network created features, and the second best model on the deep autoencoder created features.

Fig. 7 shows F1-scores, Recall and Precision obtained on the semantic datasets based on *log4j* $v1.1$ and $v1.2$. When comparing the mean performance by F1-scores values on log4j $v1.1$, we see that REPD is the best performing model on the deep belief network created features and the second best model on the convolutional autoencoder created features. However, it is the worst model on the deep autoencoder created features. When comparing the mean performance by F1-scores values on log4j $v1.2$, we see that REPD is the best model when using the deep belief network created features and the worst performing model. We attribute the bad performance of REPD to the very small amount of non-defective examples, not giving enough information to the dimensionality reduction part of REPD to learn from.

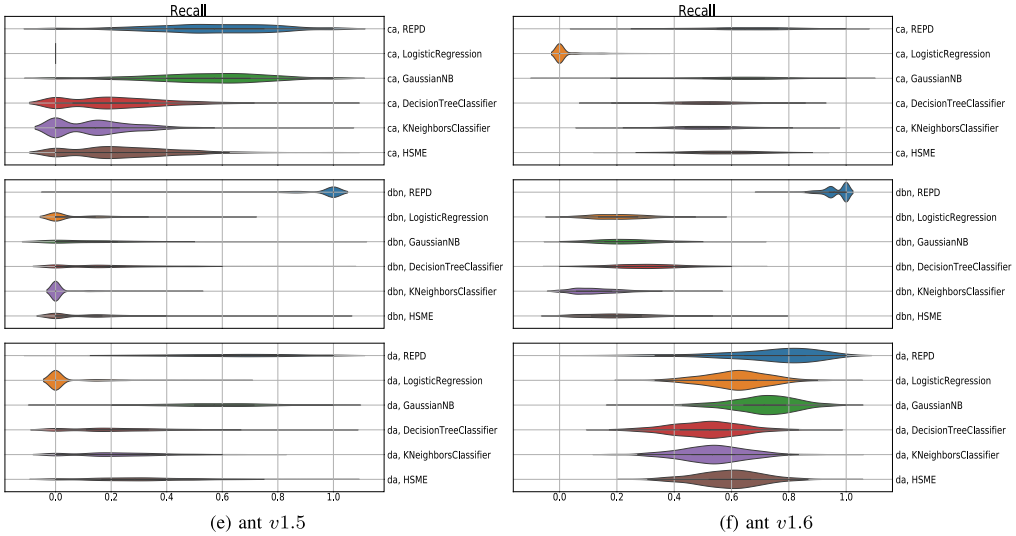
Fig. 8 shows F1-scores, Recall and Precision obtained on the semantic datasets based on *poi* $v2.0$ and $v2.5$. When comparing the mean performance by F1-scores values on $v2.0$, we see that REPD is the best performing model on the deep belief network created features, and the second best performing model on the convolutional autoencoder and deep autoencoder created features. When comparing the mean performance by F1-scores values on $v2.5$, we see that REPD is the best performing model on deep belief



(a) ant $v1.5$ (b) ant $v1.6$



(c) ant $v1.5$ (d) ant $v1.6$

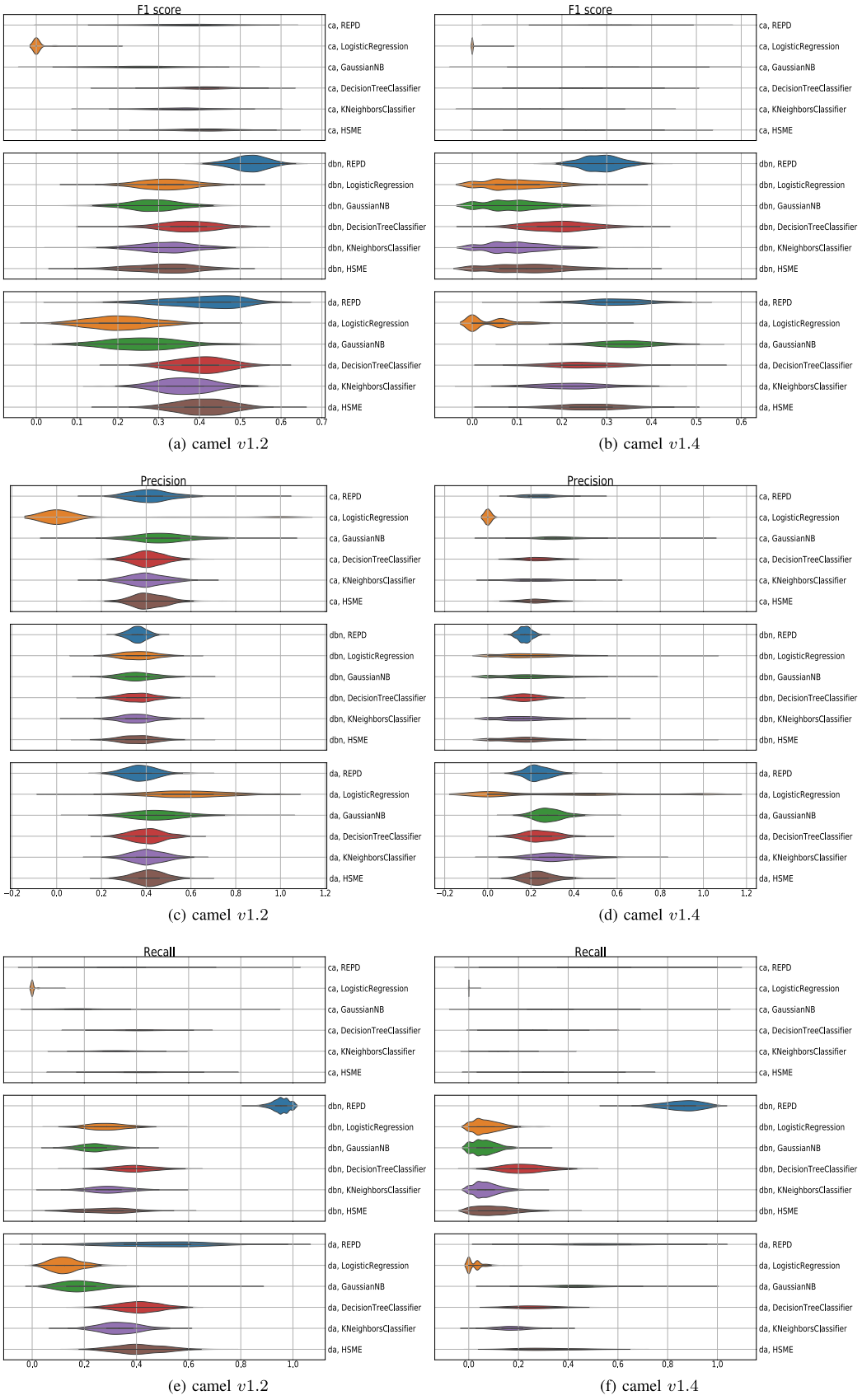


(e) ant $v1.5$ (f) ant $v1.6$

Fig. 5. Results achieved by different models on ant datasets for different feature types.

network created features, but on others its performance is bad when compared to other models. We attribute this to the small amount of non-defective examples.

When all of the results are taken into account, it turns out that REPD has the best performance on 11 out of 24 datasets and the second best performance on 6 of the remaining 12 datasets. These results show that REPD is not a “silver bullet”, but it is a high



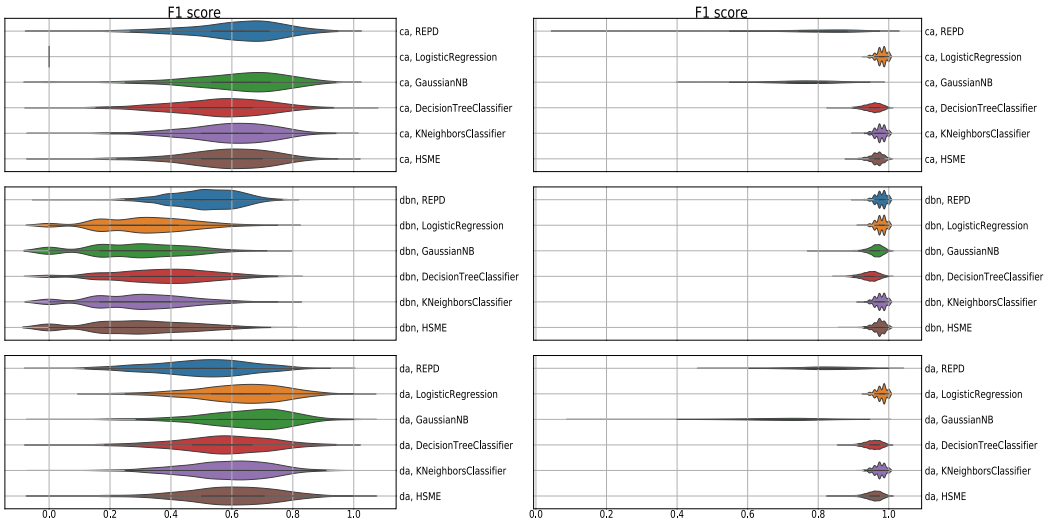
(a) camel $v1.2$ (b) camel $v1.4$

(c) camel $v1.2$ (d) camel $v1.4$

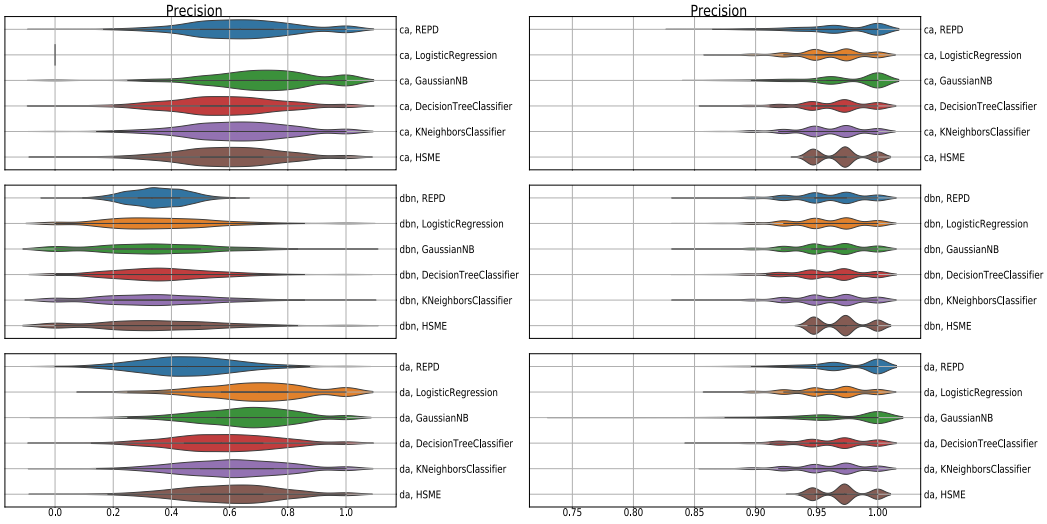
(e) camel $v1.2$ (f) camel $v1.4$

Fig. 6. Results achieved by different models on camel datasets for different feature types.

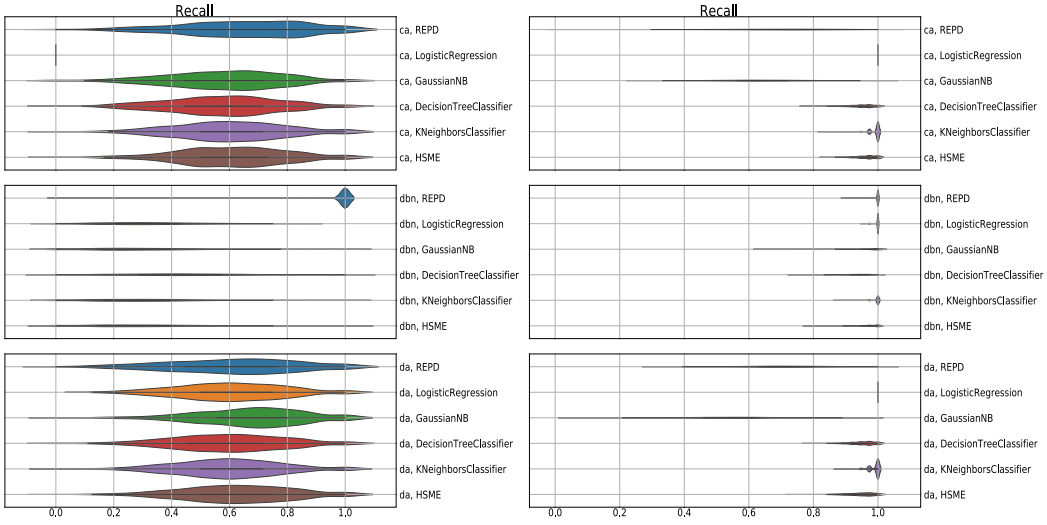
quality model for the area of defect prediction. Similarly to the experiments carried out on traditional datasets, we again see the high recall property of the proposed model, which makes it a very effective early defect detection system. These results show that



(a) log4j $v1.1$ (b) log4j $v1.2$

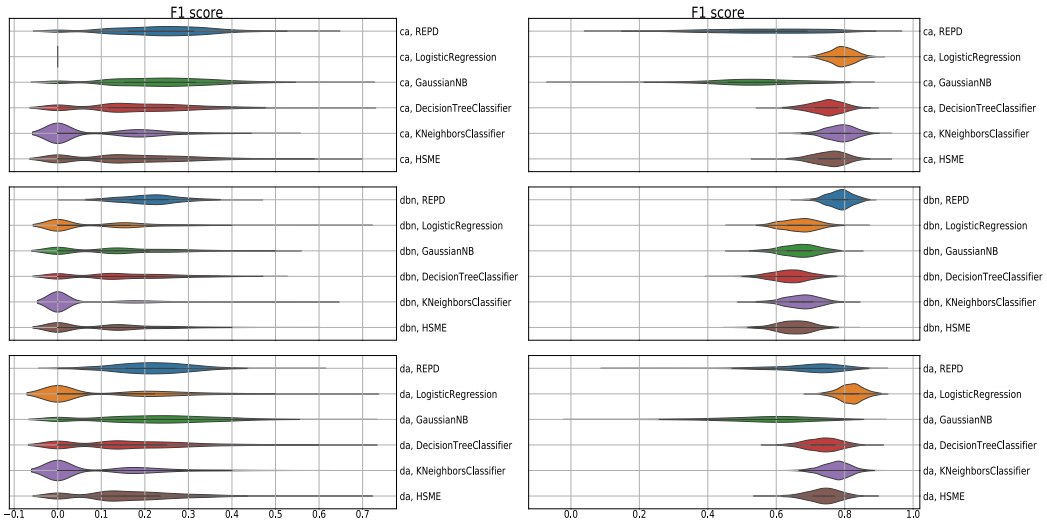


(c) log4j $v1.1$ (d) log4j $v1.2$

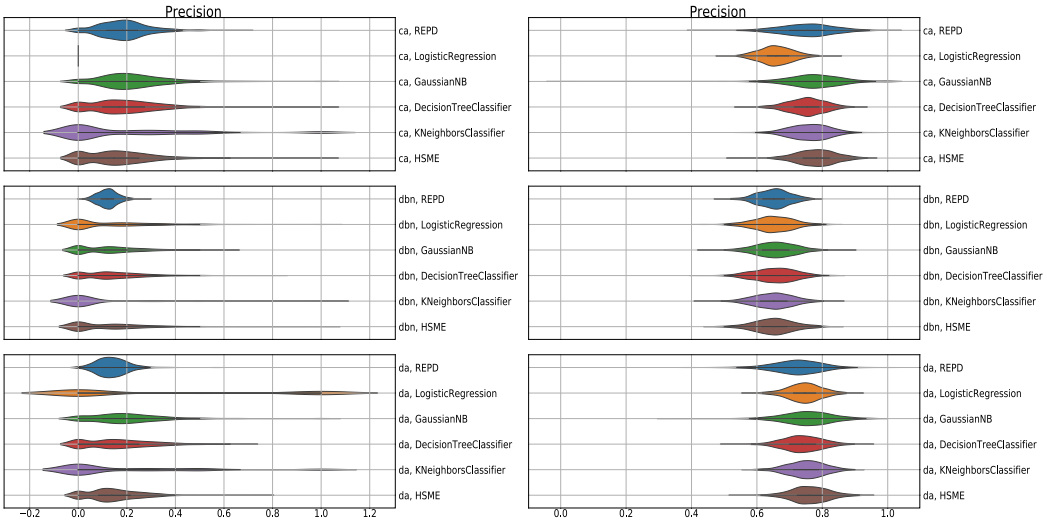


(e) log4j $v1.1$ (f) log4j $v1.2$

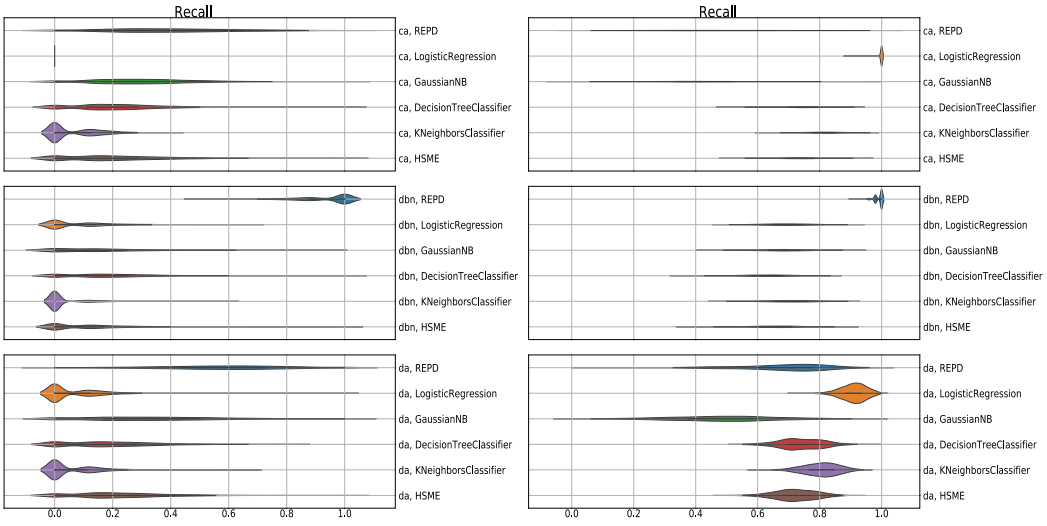
Fig. 7. Results achieved by different models on log4j datasets for different feature types.



(a) poi $v2.0$ (b) poi $v2.5$

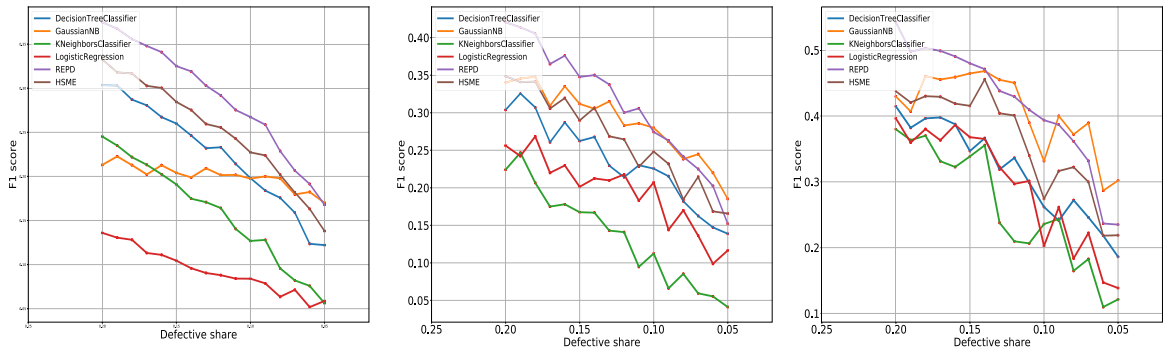


(c) poi $v2.0$ (d) poi $v2.5$



(e) poi $v2.0$ (f) poi $v2.5$

Fig. 8. Results achieved by different models on poi datasets for different feature types.



(a) JM1 (b) KC1 (c) KC2

Fig. 9. F1-scores undersampling results.

the assumption of treating defect prediction as anomaly detection makes sense and leads to promising results.

4.4. Model robustness to dataset imbalance problem

In order to test how well REPD is suited for imbalanced datasets, we decided to carry out the following experiment:

(1) Use the JM1, KC1 and KC2 datasets which have more than 20% of defective examples. These datasets were chosen because they allow us to track the change in performance as the imbalance ratio is changed. CM1 and PC1 have a higher imbalance ratio and thus do not give us a lot of space for imbalance ratio change.
(2) Change the imbalance from 20% defective examples to 5% defective examples with a step of 1%. This is done in two different ways:
   - Undersample defective examples, thus reducing their share in the dataset.
   - Oversample non-defective examples, thus increasing their share in the dataset.
(3) Train all previously mentioned models on the dataset versions with a higher imbalance.
(4) Visualize and analyze the model performance as the imbalance is altered.

We again decided to use F1-score as the main performance indication. Please note that there is a difference concerning information between undersampling and oversampling. When undersampling is used, information is lost from the dataset, while when oversampling is used, there is no loss of information from the dataset.

Undersampling. is used in the first part of the experiment. Here we modify the dataset to contain from 20% to 5% of defective examples. For each percentage this is done 30 times by randomly selecting the needed number of defective examples. After the dataset modification, the selected models are trained and evaluated.

Fig. 9 shows how the performance of each model degrades as the datasets are altered. This is expected since we are reducing the amount of examples/information the models can learn from.

Oversampling. is used in the second part of the experiment. Here we modify the dataset to contain from 20% to 5% of defective examples. For each percentage this is done 30 times by randomly selecting and duplicating the needed number of non-defective examples. After the dataset modification, the selected models are trained and evaluated.

Fig. 10 shows the performance of each model as the datasets are altered. The performance of most models is stable as the JM1 dataset is altered. The only model whose performance noticeably suffers is k-nearest-neighbors, which is not surprising given the fact we are duplicating examples and thus changing neighborhoods. REPD has the best performance on all the alterations of the JM1 dataset. On the KC1 dataset k-nearest-neighbors and logistic regression are suffering from the changes in the dataset. For KNN, the explanation is the same as when using the JM1 dataset: by adding duplicates we are changing neighborhoods. For logistic regression, our suggestion is that it cannot discriminate between the classes and by adding duplicates we are changing the surface of its error function, thus leading to a suboptimal model. REPD has the best performance on all the alterations of the KC1 dataset. We see a lot of oscillation in the performances of the models on the KC2 dataset. Our suggestion is that this is due to a small dataset size leading to a small amount of information for a model to learn from. REPD has the best performance on all the alterations of the KC2 dataset.

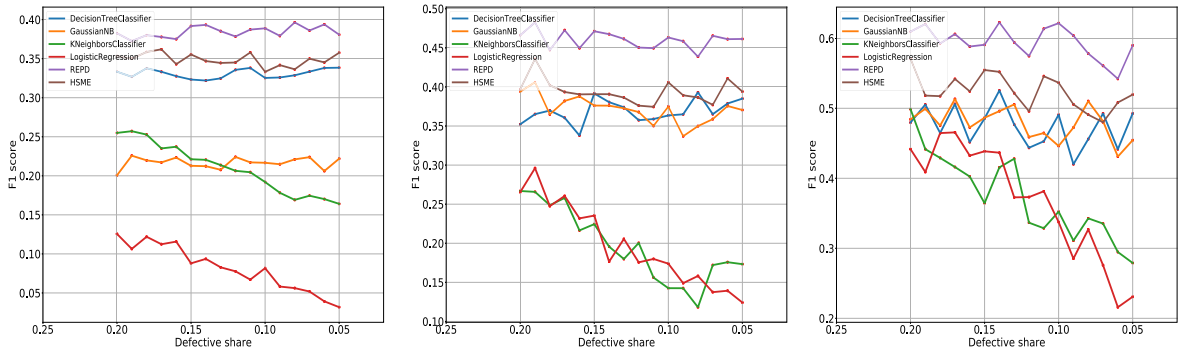
Analyzing. the results, we come to the conclusion that REPD is a superior model in performance, but lacks robustness to dataset imbalance. Namely, as the imbalance is increased, the performance of the model deteriorates and does not show stability.

5. Threats to validity

In this section, the following threats to validity of the presented work and its conclusions were identified:

Datasets used in this study come from the NASA metrics program and several open source projects. These datasets might not be representative enough and can thus give biased results. All of them come from programming languages of the C family, so the results might not translate to datasets with programming languages outside of C family. More importantly, since these datasets are created by automatically mining software repositories, they might contain noise and thus give a deceitful model performance. When mining software repositories, in order to identify defective code, commit history is examined. When a "bug fix" commit is identified, usually by the commit message containing the words *fix, fixed* or something similar, code before that commit is marked as defective and code after the commit is marked as non-defective. This process is naturally inclined towards producing false negatives. If a piece of code is defective, but has never been recognized as such, this process will mark it as non-defective and thus introduce noise into the dataset.

Experimental setup of our experiments is such that we used the default settings of alternative models. Since we did not do an exhaustive hyperparameter optimization of each of the models, it is possible that some of the models are underperforming. A more



(a) JM1 (b) KC1 (c) KC2

Fig. 10. F1-scores oversampling results.

robust model comparison would require performing a hyperparameter optimization for each of the models. In this study we did not perform such an exhaustive comparison due to budgeting and time constraints.

6. Conclusion

In this paper, we proposed treating the binary classification problem of source code defect prediction as an anomaly detection problem. We presented our REPD model which performs this task, evaluated its performance on traditional and semantic datasets, and investigated how it handles dataset imbalance. We have made the following contributions:

- Showed that anomaly detection is a valid approach to source code defect prediction.
- Introduced an anomaly detection based model and validated it on datasets using both traditional features and semantic features.
- Showed the high performance of the proposed model.
- Investigated its robustness to the dataset imbalance problem.

REPD is an applicable defect prediction model which exhibits improved defect prediction capabilities. We hope this paper to be the basis and inspiration for other researchers to consider treating defect prediction as anomaly detection.

In our future work, we will aim to further improve REPD, but also to try and investigate other alternative anomaly detection techniques which could prove beneficial for the field of source code defect prediction.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This research has been partly supported by the European Regional Development Fund under the grant KK.01.1.1.01.0009 (DATACROSS) and under the grant KK.01.2.1.01.0111 (OperOSS). The authors acknowledge the support of the Croatian Science Foundation for this research through the Reliable Composite Applications Based on Web Services (IP-01-2018-6423) research project. The Titan X Pascal used for this research was donated by the NVIDIA Corporation.

Appendix

Halsted features (Halstead, 1977) which are based on operator/operand counts and include:

- number of distinct operators η_1,
- number of distinct operands η_2,
- the total number of operators N_1,
- the total number of operands N_2,
- vocabulary $\eta = \eta_1 + \eta_2$,
- length $N = N_1 + N_2$,
- potential number of distinct operators η_1'. Counted as the number of function names and return statements.
- potential number of distinct operands η_2'. Counted as the number of arguments passed to functions.
- calculated length $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$,
- volume $V = N * \log_2 \eta$,
- difficulty $D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$ and
- effort $E = D \cdot V$.
- volume of minimal implementation $V^* = (2 + \eta_2') * log_2(2 + \eta_2')$
- program length V^*/N

McCabe features (McCabe, 1976) which are based of dependencies and include:

- Line count of code
- Cyclomatic complexity which is the number of linearly independent paths within a section of source code.
- the number of edges in a code graph E,
- the number of nodes in a code graph N and
- the number of connected components P.

CK features (Chidamber and Kemerer, 1994) based on functions and inheritance counts which include:

- number of methods defined in a class,
- number of methods in a class,
- number of remote methods directly called by methods of the class,
- number of remote methods called recursively through the entire call tree,
- depth of the inheritance tree,
- number of children meaning number of immediate subclasses of a class and
- number of classes to which a class is coupled to.

MOOD features (Harrison et al., 1998) based on polymorphism factors, coupling factors, etc. which include:

- number of visible methods,

- number of visible attributes,
- number of inherited methods,
- number of inherited attributes,
- total number of methods,
- total number of attributes,
- number of overrides,
- number of overloads,
- polymorphism factor (8),

$$PF = \frac{overrides}{\sum_{\forall class} new\ methods \cdot descendants} \tag{8}$$

- coupling C,
- maximal possible coupling MC and
- coupling factor $CF = C/MC$.

Change features (Jiang et al., 2013), which include:

- number of lines of code,
- number of lines of code added,
- number of lines of code deleted and
- number of lines of code edited.

Object oriented features (e Abreu and Rogério, 1994) include:

- Weighted Methods for Class which was proposed as the sum of all complexities of the methods in the class,
- Coupling between Objects which indicates the number of classes coupled to a particular class,
- Response For Class which is defined as the set of methods that can potentially be executed in response to a message received by an object of that class,
- Lack of Cohesion of Methods which is defined as the number of pairs of methods that share a reference to instance variables,
- Depth of Inheritance Tree which is defined as the number of classes that a particular class inherits from,
- Number of Children which is defined as the number of immediate subclasses of a class.

References

Afric, P., Sikic, L., Kurdija, A.S., Silic, M., 2019b. REPD Model-source code. https://github.com/pa1511/REPD, accessed: 18.11.2019.

Alsawalqah, H., Faris, H., Aljarah, I., Alnemer, L., Alhindawi, N., 2017. Hybrid smote-ensemble approach for software defect prediction. In: Silhavy, R., Silhavy, P., Prokopova, Z., Senkerik, R., Kominkova Oplatkova, Z. (Eds.), Software Engineering Trends and Techniques in Intelligent Systems. Springer International Publishing, Cham, pp. 355–366.

Arnold, B.C., 2015. Pareto distribution. In: Wiley StatsRef: Statistics Reference Online. American Cancer Society, pp. 1–10, [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118445112.stat01100.pub2.

Bennin, K.E., Keung, J., Phannachitta, P., Monden, A., Mensah, S., 2018. Mahakil: Diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction. IEEE Trans. Softw. Eng. 44 (6), 534–550.

Bettenburg, N., Nagappan, M., Hassan, A.E., 2012. Think locally, act globally: Improving defect and effort prediction models. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). pp. 60–69.

Cheung, L., Roshandel, R., Medvidovic, N., Golubchik, L., 2008. Early prediction of software component reliability. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 111–120.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Cleophas, T.J., Zwinderman, A.H., 2011. Bonferroni t-test. In: Statistical Analysis of Clinical Data on a Pocket Calculator: Statistics on a Pocket Calculator. Springer Netherlands, Dordrecht, pp. 41–42, [Online]. Available: https://doi.org/10.1007/978-94-007-1211-9_15.

Denaro, G., 2000. Estimating software fault-proneness for tuning testing activities. In: Proceedings - International Conference on Software Engineering. pp. 704–706. http://dx.doi.org/10.1109/ICSE.2000.870474.

Dodge, Y., 2008. Kolmogorov–Smirnov Test. In: The Concise Encyclopedia of Statistics. Springer New York, New York, NY, pp. 283–287, [Online]. Available: https://doi.org/10.1007/978-0-387-32833-1_214.

Dubey, S.D., 1970. Compound gamma, beta and f distributions. Metrika 16 (1), 27–31, [Online]. Available: https://doi.org/10.1007/BF02613934.

e Abreu, F.B., Rogério, C., 1994. Ndidate metrics for object-oriented software within a taxonomy framework. J. Syst. Softw. 26 (1), 87–96, Achieving quality in software. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016412129490099X.

Flury, B., 1997. The multivariate normal distribution. In: A First Course in Multivariate Statistics. Springer New York, New York, NY, pp. 171–207, [Online]. Available: https://doi.org/10.1007/978-1-4757-2765-4_3.

Fu, W., Menzies, T., 2017. Revisiting unsupervised learning for defect prediction. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, ACM, New York, NY, USA, pp. 72–83, [Online]. Available: http://doi.acm.org/10.1145/3106237.3106257.

Giger, E., D'Ambros, M., Pinzger, M., Gall, H.C., 2012. Method-level bug prediction. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. In: ESEM '12, ACM, New York, NY, USA, pp. 171–180, [Online]. Available: http://doi.acm.org/10.1145/2372251.2372285.

Glorot, X., Bengio, Y., 2010. Understanding the difficulty of training deep feedforward neural networks. J. Mach. Learn. Res. - Proc. Track 9, 249–256.

Graves, T.L., Karr, A.F., Marron, J.S., Siy, H., 2000. Predicting fault incidence using software change history. IEEE Trans. Softw. Eng. 26 (7), 653–661.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2012. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.

Halstead, M.H., 1977. Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA.

Harrison, R., Counsell, S.J., Nithi, R.V., 1998. An evaluation of the MOOD set of object-oriented software metrics. IEEE Trans. Softw. Eng. 24 (6), 491–496.

Hart, P.E., 1990. Lognormal distribution. In: Eatwell, J., Milgate, M., Newman, P. (Eds.), Econometrics. Palgrave Macmillan UK, London, pp. 145–147, [Online]. Available: https://doi.org/10.1007/978-1-349-20570-7_20.

Hassan, A.E., 2009. Predicting faults using the complexity of code changes. In: 2009 IEEE 31st International Conference on Software Engineering. pp. 78–88.

Hill, C.R., Thompson, B., 2005. Computing and interpreting effect sizes. In: Smart, J.C. (Ed.), Higher Education: Handbook of Theory and Research. Springer Netherlands, Dordrecht, pp. 175–196, [Online]. Available: https://doi.org/10.1007/1-4020-2456-8_5.

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

Jing, X.-Y., Ying, S., Zhang, Z.-W., Wu, S.-S., Liu, J., 2014. Dictionary learning based software defect prediction. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, ACM, New York, NY, USA, pp. 414–423, [Online]. Available: http://doi.acm.org/10.1145/2568225.2568320.

Khoshgoftaar, T.M., Seliya, N., 2002a. Software quality classification modeling using the SPRINT decision tree algorithm. In: 14th IEEE International Conference on Tools with Artificial Intelligence, 2002. (ICTAI 2002). Proceedings.. pp. 365–374.

Khoshgoftaar, T.M., Seliya, N., 2002b. Tree-based software quality estimation models for fault prediction. In: Proceedings Eighth IEEE Symposium on Software Metrics. pp. 203–214. http://dx.doi.org/10.1109/METRIC.2002.1011339.

Khoshgoftaar, T.M., Seliya, N., Gao, K., 2005. Assessment of a new three-group software quality classification technique: An empirical case study. Empir. Softw. Eng. 10 (2), 183–218, [Online]. Available: http://dx.doi.org/10.1007/s10664-004-6191-x.

Khoshgoftaar, T.M., Yuan, X., Allen, E.B., 2000. Balancing misclassification rates in classification-tree models of software quality. Empir. Softw. Eng. 5 (4), 313–330, [Online]. Available: https://doi.org/10.1023/A:1009896203228.

Kim, S., James Whitehead, E., Zhang, Y., 2008. Classifying software changes: Clean or buggy? Softw. Eng. IEEE Trans. 34, 181–196.

Kim, S., Zhang, H., Wu, R., Gong, L., 2011. Dealing with noise in defect prediction. In: 2011 33rd International Conference on Software Engineering (ICSE). pp. 481–490.

Kim, S., Zimmermann, T., Pan, K., Jr. Whitehead, E.J., 2006. Automatic identification of bug-introducing changes. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06). pp. 81–90.

Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A., 2007. Predicting faults from cached history. In: 29th International Conference on Software Engineering (ICSE'07). pp. 489–498.

Kingma, D., Ba, J., 2014. Adam: A method for stochastic optimization. In: International Conference on Learning Representations.

Koh, K., 2014. Univariate normal distribution. In: Michalos, A.C. (Ed.), Encyclopedia of Quality of Life and Well-Being Research. Springer Netherlands, Dordrecht, pp. 6817–6819, [Online]. Available: https://doi.org/10.1007/978-94-007-0753-5_3109.

Koru, A.G., Liu, H., 2005. An investigation of the effect of module size on defect prediction using static measures. SIGSOFT Softw. Eng. Notes 30 (4), 1–5. http://dx.doi.org/10.1145/1082983.1083172, [Online]. Available: http://doi.acm.org/10.1145/1082983.1083172.

Lessmann, S., Baesens, B., Mues, C., Pietsch, S., 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Trans. Softw. Eng. 34 (4), 485–496.

Madansky, A., 1988. Testing for normality. In: Prescriptions for Working Statisticians. Springer New York, New York, NY, pp. 14–55, [Online]. Available: https://doi.org/10.1007/978-1-4612-3794-5_2.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320.

Meneely, A., Williams, L.A., Snipes, W., Osborne, J.A., 2008. Predicting failures with developer networks and social network analysis. In: SIGSOFT FSE.

Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A., 2010. Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. 17 (4), 375–407, [Online]. Available: https://doi.org/10.1007/s10515-010-0069-5.

Moser, R., Pedrycz, W., Succi, G., 2008. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. pp. 181–190.

Nagappan, N., Ball, T., Zeller, A., 2006. Mining metrics to predict component failures. In: Proceedings of the 28th International Conference on Software Engineering. In: ICSE '06, ACM, New York, NY, USA, pp. 452–461, [Online]. Available: http://doi.acm.org/10.1145/1134285.1134349.

Neela, K.N., Ali, S.A., Ami, A.S., Gias, A.U., 2017. Modeling software defects as anomalies: A case study on promise repository. JSW 12, 759–772.

Ostrand, T.J., Weyuker, E.J., Bell, R.M., 2005. Predicting the location and number of faults in large software systems. IEEE Trans. Softw. Eng. 31 (4), 340–355.

Pal, M., Ali, M., Woo, J., 2003. Exponentiated Weibull distribution. Statistica 66, 139–147. http://dx.doi.org/10.6092/issn.1973-2201/493.

Prechelt, L., Pepper, A., 2014. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. Inf. Softw. Technol. 56 (10), 1377–1389, [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914001049.

Sakurada, M., Yairi, T., 2014. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In: Proceedings of the MLSDA 2014 2Nd Workshop on Machine Learning for Sensory Data Analysis. In: MLSDA'14, ACM, New York, NY, USA, pp. 4:4–4:11, [Online]. Available: http://doi.acm.org/10.1145/2689746.2689747.

Sayyad Shirabad, J., Menzies, T., 2005. The PROMISE Repository of Software Engineering Databases.. School of Information Technology and Engineering, University of Ottawa, Canada, [Online]. Available: http://promise.site.uottawa.ca/SERepository.

Singh, V.P., 1998. Normal distribution. In: Entropy-Based Parameter Estimation in Hydrology. Springer Netherlands, Dordrecht, pp. 56–67, [Online]. Available: https://doi.org/10.1007/978-94-017-1431-0_5.

Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data. In: Proceedings of the 37th International Conference on Software Engineering - Volume 2. In: ICSE '15, IEEE Press, Piscataway, NJ, USA, pp. 99–108, [Online]. Available: http://dl.acm.org/citation.cfm?id=2819009.2819026.

Tantithamthavorn, C., 2019a. https://github.com/klainfo/DefectData, accessed: 18.11.2019,

Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E.W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., Contributors, S..., 2020. Scipy 1.0: Fundamental algorithms for scientific computing in python. Nature Methods 17, 261–272. http://dx.doi.org/10.1038/s41592-019-0686-2.

Wang, S., Liu, T., Tan, L., 2016. Automatically learning semantic features for defect prediction. In: Proceedings of the 38th International Conference on Software Engineering. In: ICSE '16, ACM, New York, NY, USA, pp. 297–308, [Online]. Available: http://doi.acm.org/10.1145/2884781.2884804.

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., Xu, B., Leung, H., 2016. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, ACM, New York, NY, USA, pp. 157–168, [Online]. Available: http://doi.acm.org/10.1145/2950290.2950353.

Yu, X., Liu, J., Yang, Z., Jia, X., Ling, Q., Ye, S., 2017. Learning from imbalanced data for predicting the number of software defects. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). pp. 78–89.

Zhang, H., 2009. An investigation of the relationships between lines of code and defects. In: 2009 IEEE International Conference on Software Maintenance. pp. 274–283.

Zhang, H., Zhang, X., Gu, M., 2007. Predicting defective software components from code complexity measures. In: 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007). pp. 93–96.