Extremely Imbalanced Smell-based Defect Prediction

Bruno Castro Machado
bruno.castro.machado@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal
October 2019

Abstract

In continuous integration/continuous delivery, one of the main requirements for high-speed delivery of software is to find bugs efficiently. For this reason, multiple solutions were introduced in the literature. For instance, defect prediction approaches based on bad code smells detected in modules from each version of the software. Nevertheless, these approaches do not consider the problem where there may exist an extremely higher percentage of non-defective modules compared to defective modules. Given that, each version of the software may only have a small number of defects. As a result, in this thesis, we introduce a new model with an autoencoder algorithm that uses design and implementation smells to detect defective modules. Therefore, we trained five autoencoders with distinct architectures. Additionally, for evaluation, we compared each model against autoencoders with the same architecture, trained with traditional object-oriented metrics and the combination of both. Our analysis did not show promising results, as the use of only smells and the combination of features did not provide an improvement compared with the use of metrics. However, we introduce a starting point for smell-based defect prediction in the context of dataset imbalance. Furthermore, we introduce a baseline for future work.

Keywords: Software, Defect Prediction, Code Smells

1. Introduction

One of the main requirements for a high-speed delivery of software is the ability to find bugs in an effortless and resource-less way. As a matter of fact, one of the main adversities in continuous integration/continuous delivery happens at the testing phase. As, this phase may become really time consuming and be a bottleneck for the pipeline. For this reason, the Defect Prediction field has gain relevance in recent years. And, has became one of the most active research areas in software engineering. In particular, a promising family of defect prediction approaches based on the history of software has received much more attention in the past few years [14]. Additionally, in the software engineering field, there is a topic concerning bad code smells. It focuses on the relationship between the detection of code smells and the presence of defects in modules [13]. Accordingly, a recent field has emerged, that combines both of these areas. It explores the use of code smells as features for the prediction of defects in software [17, 21, 10, 27].

In the field of smell-based defect prediction, several studies suggested multiple approaches to predict defects using smells. Moreover, the majority introduced classification models whose use of smells improved the model’s performance. However, these studies do not consider a scenario with an extremely imbalanced dataset. For example, the scenario where the dataset has one million instances for the non-defective class and only one hundred for the defective class. Thus, creating an extreme uneven ratio. In such scenarios, if it used the same approaches as in the ones used with even rations, or slight imbalanced datasets. Then, the models would not achieve their optimal performance [20]. Furthermore, if the standard evaluation approaches that are used for performance measurement (such as the ROC curve) are also used in this scenarios, then it produces misleading scores for the model performance. [23]

Considering that problem, in this study we introduce a new smell-based defect prediction model prepared for the imbalanced dataset problem. Therefore, we focused on the application of an outlier detection approach to train the model to detect defective modules. Specifically, we decided to use an autoencoder algorithm. This model learns smells from non-defective modules. And, it becomes fit to reconstruct non-defective input with the minimum loss. Given that, the outliers are detected from the features that are reconstructed with the highest errors compared against the
original features. In other words, smells that are badly reconstructed by the model should indicate that the module is defective. In the end, to create classification, it is used a threshold value on the reconstruction value to ascertain whether the module is defective or not.

With this in mind, for this study we considered three research questions.

RQ1 Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with design and implementation smells or with traditional object-oriented features?

RQ2 Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with the combination of both design and implementation smells, and with traditional object-oriented features or when it is trained with only one single of those features?

RQ3 When training a smell-based defect prediction model with an autoencoder algorithm, what category of architecture provides the best model performance?

In the end, this study did not show promising results. Since, in general, the models trained with traditional object-oriented metrics outperformed the models trained with design and implementation smells. Nevertheless, we were able draw conclusions about the impact of the architecture on the models performance. Where, the deeper architectures created models with a slightly higher accuracy than the shallower architectures. However, in our opinion, we do not consider that the architecture choice is an impactful decision for the model performance.

We organized this paper as follows. Section 2 provides the background and related work, introducing the concepts within defect prediction, bad code smells, imbalanced data and autoencoders. Section 3 discusses the methodology we implemented in the research. Section 4 presents and discusses the results. Section 5 presents the future work. And Section 6 presents the conclusion for this study.

2. Background and Related Work

2.1. Defect Prediction

Defect Prediction is a sub-field of the mining software repositories (MSR) major area. This field provides a conventional methodology to do research in that field. Henceforth, Hemmati, H. et al. [9] describes this process as being composed by four phases. First, it is the acquisition and preparation of data. It consists of the extraction of raw data from different types of development artifact repositories. For example, source code, SCM, bug tracking issues, and mailing lists. In addition, it includes the modeling of information and the creation of connections between repositories. Second, it is an appropriate mining/learning technique to the prepared data. Such as clustering, classification, prediction, regression, or other machine learning algorithm. Third, it is applied the analysis of data. Thus producing the research conclusions. Last, the data and tools used in the previous study are shared. For the purpose of external validation and replication studies.

We have defined the general methodology for mining software repositories. Now, we will focus on defining the process of building a defect prediction model. Firstly, we extract data from software archives. Such as version control systems (eg. SVN, CVS, Git), which provide source code and commit messages, issue tracking systems (eg. Bugzilla, jira) that contains defect information, and so on. The extracted data instances are considered as modules. Each module may be a method, a class, a source code file, a package or a code change, depending on the chosen granularity. Secondly, we create data instances with the metrics (also features) extracted from each module. These metrics are defect prediction features. And can be composed by the complexity of software (eg. lines of code) and by the development process (eg. code changes). Then, to each set of features, representing a module, we attach a label. This label classifies the model as defective or non-defective. Thirdly, we split the data instances into train and testing datasets. Moreover, we train a machine learning algorithm the training dataset. Lastly, we use the testing instances to evaluate the performance of the trained model [14]. This process is illustrated in figure 1.

2.2. Bad Code Smells

A bad code smell is a pattern of malpracticed software development. To put it differently, smells
are symptoms of poor decisions in the creation of software. These decisions comprise the whole life-cycle of the system development. Thus, for each phase, there are an unique set of poor choices that can be applied. For example, the design and testing stage, each, have a set of unique bad decisions that can be committed. For this reason, from each step of the life-cycle, there are a set of smells that can be detected. The accumulation of smells over a significant number of revisions, contributes to the accumulation of technical debt [7]. Technical debt is the aggregated cost of additional project maintenance and rework. As a result, the higher the debt and accumulation of bad decisions, the higher the price to be paid. Over an extreme scenario, the technical debt will reach a level where it will be impossible to pay the debt. Consequently, leading to a scenario called technical bankruptcy. Causing the project to be discontinued.

For the remainder of this study we will only consider design and implementation smells. Design smells are a set of bad practices concerning the design stage of the software development. Moreover, they are characterized by the principles they violate. These principles are comprised by abstraction, encapsulation, modularization and hierarchy principles. Their definition and corresponding set of smells are described in table 1. Implementation smells are the set of malpractices concerning the implementation phase. We define each implementation smell in table 2 [29].

### 2.3. Imbalanced Datasets

The problem of imbalanced data consists of an unequal distribution of classes in classification problems [1]. For illustration, consider a scenario of a 2-class (binary) classification problem with 1000 instances. In particular, assume that 990 instances are labelled with Class-1 (majority class) and the remaining 10 with Class-2 (minority class). Moreover, the goal is to predict the Class-2 events. However, most common classification algorithms work poorly for such scenarios. The reason is that classifiers tend to be biased towards the majority class. Hence, not performing well for the minority class. With this in mind, the classifier will measure a very high accuracy of 99%. But it will be a misleading measurement. Since the accuracy is only reflecting the underlying class distribution.

Imbalanced data is a recurrent problem in real life applications. For instance, the application of fault diagnosis, anomaly detection, e-mail folding and face recognition, among others [8]. Furthermore, defect prediction also has an imbalance problem. Since, usually only a small percentage of modules, in each version of the software repository, is defective. Thus, leading to an uneven ratio of class percentages.

We want to build a prediction model, considering the data imbalance. Thus we can take advantage of the imbalance of data. In detail, we can teach the model to detect the minority class as an outlier. In general, this approach is called outlier detection. And, is a commonly used solution to train a classifier with data imbalance. On the whole, outlier detection is a data cleaning task. Commonly, used to explore the set of normal objects in data. Nonetheless, it has been gaining more prominence as a potential data discovery task for anomaly detection. Or, in our context, defect detection [28].

In the literature, several models were introduced that combine defect prediction with the detection of bad code smells. However, these models were not created under an environment where the datasets have extremely imbalanced ratios. Thus, they were built using supervised classification algorithms. And, they considered the receiver operating characteristic curve and the f-measure to evaluate the models [17, 21, 10, 27].

### 2.4. Autoencoders

For this study we have to consider a prediction algorithm that works well with outlier detection. Therefore, we considered the autoencoder [18]. An autoencoder is an unsupervised artificial neural network. And, its purpose is to generate a solid internal representation of the input data. An artificial neural network is a set of interconnected networks of processing units. These processing units (also artificial neurons) mimic the biological neurons found in the brain. Each biological neuron receives input signals from its neighboring neurons into its dendrites. Then, the signals are passed on to the cell body. Where, they are summed into a single signal. Again, the resulting signal is passed on to the terminal of the neuron. If the signal’s charge is higher than a specified threshold, then the signal is released to the neighboring dendrites.

A meaningful category of unsupervised artificial networks that generates a solid internal representation of the input data is one of the autoencoders. The simplest structure of an autoencoder consists of three layers: the input layer, the hidden layer, and the output layer. An autoencoder is divided into two stages, the encoder, and the decoder. The encoder stage occurs from the input layer to...
the hidden layer and has the goal of generating a meaningful representation of the input at the hidden layers. The decoder stage takes the produced representation and reconstructs it to the original dimension from the hidden layer to the output layer. Between layers there are a set of transitions that propagate signals. Each one has a correspondent weight and bias associated with it. Thus, if we represent the weights as the matrix $W \in \mathbb{R}^{d \times n}$ and the bias as $b = [b_1, b_2, b_3, \ldots, b_d]^T \in \mathbb{R}^{d \times 1}$, after retrieving a signal we can express the receiving unit $h_1$ where $f_1$ is the activation function at that layer.

$$h = f_1(Wx + b)$$  \hspace{1cm} (1)

The purpose of the activation function is to identify non-linear complex functional mappings between the data, thus introducing non-linear properties on the network. There is an extensive array of activation functions used for neural units. Their use varies based on the problem context and the properties of the neural network.

The sigmoid activation function produces a continuous range from 0 to 1. This characteristic makes the sigmoid function ideal to be used to output probability related to a given class for binary classification. While the tanh activation function outputs values between -1 and 1. The ReLU activation function output is equal to the net input if the received input is higher than zero, and it is zero if the overall input is lesser than or equal to zero.

### 3. Methodology

In this chapter we describe the approach and methodology that we implemented to apply smell-based defect prediction. We will organize this chapter into four sections, each defining a step in the procedure. First, in chapter 3.1, we describe the feature extraction phase. Then, in chapter 3.2 we introduce the data processing phase. In chapter 3.3 we describe the methodology for training the models. And in chapter 3.4 we discuss the methods to evaluate the models. In figure 2 we illustrate the different phases in our approach.

### 3.1. Feature extraction

The main goal of this phase is to extract three types of features from all versions of open source repositories. Therefore, we iterated through all the defective versions of a set of repositories, and for each module, we extracted a group of traditional object-oriented metrics, design smells and implementation smells. And we recorded the information regarding whether the module is defective or not, to be used for the output of our datasets.

We used two different repository databases with a
### Implementation Smells

<table>
<thead>
<tr>
<th>Implementation Smell From Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Function Call From Constructor</td>
<td>Constructor calls an abstract method</td>
</tr>
<tr>
<td>Complex Conditional</td>
<td>Conditional statement is complex</td>
</tr>
<tr>
<td>Complex Method</td>
<td>Method has high cyclomatic complexity</td>
</tr>
<tr>
<td>Empty catch clause</td>
<td>Catch block of an exception is empty</td>
</tr>
<tr>
<td>Long Identifier</td>
<td>Identifier has an excessive length</td>
</tr>
<tr>
<td>Long Method</td>
<td>Method is too long to be understood</td>
</tr>
<tr>
<td>Long Parameter List</td>
<td>Method accepts a long list of parameters</td>
</tr>
<tr>
<td>Long Statement</td>
<td>Statement has an excessive length</td>
</tr>
<tr>
<td>Magic Number</td>
<td>Unexplained number is used in an expression</td>
</tr>
<tr>
<td>Missing default</td>
<td>Switch statement does not contain a default case</td>
</tr>
</tbody>
</table>

Table 2: Name and description of the implementation smells used in this study.

set of multiple real bugs: Defects4J [12] and BugsDotJar [22]. Then, when iterating each module, we extracted the features we required using Designite [26] and we captured the information regarding the modules being defective or not, based on the interface from the software framework.

### 3.2. Data processing

From the information generated in the previous phase, we produced and processed three datasets. One with traditional object oriented metrics, one with design and implementation smells and one with the combination of both metrics and smells. Therefore, we iterated through the information class by class and retrieved for each module (which is an instance in the dataset) the appropriate features for all the datasets. Furthermore, we also included the output information for each module and assigned the value 1 if the module is defective and the value 0 otherwise. Consequently, we applied a set of operations to prepare the datasets for training. We standardized the features to balance the different magnitudes among values. We applied stratification to partition the data into subpopulations (stratum), in order to divide and shuffle the dataset into a testing and training dataset, while maintaining the proportions of samples for each class. And we removed the defective instances from the training dataset to train our autoencoders to recognize the defective modules as the outliers.

To apply the operations, we used a set of python's libraries such as the pandas python library [11] [15] to manipulate the data, generate the datasets and remove defective modules from the training dataset. We also used the scikit-learn python library [11] [19] to apply the operations of standardization and stratification.

### 3.3. Model training

In this section, we discuss the description and the decisions regarding the type and architecture of the autoencoder for each model we have decided to train, considering both the input and the output involved. Thus, we considered two variants of five different model architectures that fit the extremely imbalanced environment.

3.3. Model training

Currently, there is not enough research directing us toward an optimal architecture and there is not a general guideline to help us decide on the best parameters. Therefore, we applied an empirical observation, comparing five distinct models against three different datasets. We defined the distinction between the architectures based on the following criteria. The number of layers, where we applied three shallow networks (when it has two or less hidden layers) and two deep networks (when it has more than two hidden layers). The activation functions in each layer. And, the dimensions in each layer. As such, all the shallow layers have two hidden layers architectures. Therefore, the first has two hidden sigmoid functions. The second has two hidden tanh functions. And the third has two hidden ReLU functions. The two deep autoencoders have a different number of layers with the same activation functions. Thus, both of them used the ReLU activation functions. The first architecture has four hidden layers and the second architecture has six hidden layers. Furthermore, the output layers we considered are the sigmoid activation function for the autoencoders trained with only smells, with values ranging between 0 and 1. Also, we applied a ReLU activation function for the other two datasets, as the metric values need to be positive. The criteria to the dimension of the layers, was to to divide by two and, then, multiply by two, thus creating encoding and decoding layers. To measure the loss during training we used the mean square error (MSE) loss function,
which is the average squared difference between the estimated values and what is estimated. We decided to apply this function due to its property to assign heavy weights to the outliers, which is desirable as our end goal is to detect defective modules (outliers). [3]

In the end, we trained three models for each of five different architectures. Each set of five models using a distinct dataset. Therefore, in practice, to define and train our models, we used the keras python library [4].

3.4. Model evaluation
To analyze each model about its performance regarding its dataset and architecture, we relied on the different outcomes of the confusion matrix and applied two procedures, the receiver operating characteristic curve [5] and the precision-recall-gain curve [6]. In addition, to compare the performance of the different models, we used the area under the curve (AUC) metric on both curves.

The confusion matrix illustrates the relation between the predicted values and the actual output. Therefore, as illustrated in figure 3, we have four possible outcomes. True Positive, if the instance is defective (d) and we predicted defective (d). True Negative if it predicted, instead, non-defective (n). False Positive if it was predicted non-defective (n) and was actually non-defective. And False Negative if was, instead, predicted defective (d).

The receiver operating characteristic curve measures the relationship between the true positive rate (TPR), which is how many of the modules we predicted defective were actually defective and the false positive rate (FPR), that describes how many of the modules we predicted defective were actually not defective. The curve summarizes the trade-offs between the benefits (true positives) and the costs (false positives) for each possible threshold score, that defines when a prediction is defective or not. Thus, the curve is conceptualized by aligning all the points generated by each threshold. Considering a list of score points defining each threshold that ranges from $-\infty$ to $+\infty$, the minimum possible threshold produces the point (0, 0) and the maximum, the point (1, 1). Therefore, to analyze the curve, the performance is better as the curve bows closer the point (1, 0) or (0, 1), but is worse as the curve falls closer to $y = x$ line, as it provides the least amount of information. We can also use the area under the curve that ranges from 0 to 1 and the values closer to 0.5 summarize a set of lesser skilled models compared against models with scores near 0 and 1.

To generate the receiver operating characteristic curve and calculate its area, we calculated the score values, which are the loss values that represent the distance between the reconstructed features and the input features. And, we, used the function provided by scikit-learn — sklearn.metrics.roc_curve — to get the true positive rate and the false positive rate for each threshold and used the sklearn.metrics.auc function to get the area under the curve.

Under an imbalanced dataset scenario, is has been proved that using a ROC curve for evaluation, leads to a deceptive interpretation of the model’s true performance [23]. As such, we also considered the precision-recall-gain curve to measure the model’s performance. Since, it only evaluates the fraction of true positives among the positive predictions. The PRG curve is an extension from the the precision-recall curve, introduced by Flach, P. and Kull, M. [6], as they identified a set problems and lack of properties in the precision-recall curve compared with the ROC curve. Therefore, they introduced a variation to that method, as it applies a min-max harmonic scale to the calculation of the precision and the recall. When analyzing the precision-recall-gain curve, the performance of the model is better as the curve bows closer to the point (1, 1) or (0, 0), being considered a perfect model when it has precG and recG of 1. If the curve is closer to the baseline, which is the minor diagonal that goes from (1,0) to (0,1) then the model has a worse performance. We can also consider the area under the curve that ranges from 0 to 1 and the value closer to 0.5 summarizes a set of lesser skilled model.

To generate the precision-recall-gain curve and calculate the area under the curve, we used the library implemented by Flach, P. and Kull, M. [6], that uses the score values and the true values to generate the curve and to calculate the area under the curve.
4. Results & discussion

Illustrated in figure 4, it is the relationship between areas under the curve when applied the receiver operating characteristic evaluation between architectures for each dataset type. And, the differences in areas under the curve between datasets for each architecture.

We observed that the area of the models trained with the smells dataset averaged to 0.51. This shows that the predictive accuracy of all the models trained with a smell-only dataset reveal the same performance as a random classifier. Thus, it shows that the use of smells is not providing relevant information for defect prediction. This implies that using only smells is not enough to achieve an accurate prediction. And, that there may not exist a correlation between smells and defects. Additionally, all the models trained with traditional object oriented metrics showed a higher accuracy prediction compared against all the other datasets. For instance, while the area for the smells-only models ranged from 0.50 to 0.52, the metrics-only models had a minimum area of 0.58 and a maximum area, for the deeper models, of 0.71. This implies that we can extract better information for defect prediction from traditional object-oriented metrics than with design and implementation smells. When considering the models trained with the combination of both metrics and smells, the predictive accuracy was better with models trained with only smells, compared against models trained with metrics-only. We analyzed that the models trained with the metrics dataset, except for the two-tanh-hidden-layers architecture whose area is worse by 0.01, outperformed the metrics-and-smells models. This means that using smells does not provide extra skillfulness to the models when used with traditional object-oriented metrics, making the performance worse. In addition, combining metrics to smells improved the predictive accuracy of the models, except for the two-ReLU-hidden-layers architecture whose area became worse with a difference of 0.10.

Illustrated in figure 5 it is the relationships between the precision-recall-gain areas under the curve between all models architectures trained with all datasets. Considering the results displayed in that figure, the metrics-only models showed an overall nearly perfect performance, averaging 0.97. Compared against the models trained with the combination of metrics and smells, when including the smells as features, it lead to a decrease in area for all the models, most significantly on the model with the two-ReLU-hidden-layers architecture, with a difference in area of 0.33. On the other hand, the models trained with the smells-only dataset showed negative results. The authors of the PRG methodology, briefly described that negative areas are indicators of models that are worse than a random classification model. Since all smells-only models displayed negative results, we assume that their performance was worse compared with models trained by the other datasets.
Answer to RQ1 After analyzing the different models, we concluded that a model with traditional object-oriented metrics outperformed a model trained with a dataset with only smells. Looking into the receiver operating characteristic curve results, the smells-only model showed results near the baseline, with an average area of 0.51 with a standard deviation of 0.0071, revealing the equivalent performance of a random classifier. On the other hand, the model trained with only traditional object-oriented metrics, had an average area of 0.655 with a standard deviation of 0.058, which is still close to the baseline value but with a better skill than the smells-only model.

Answer to RQ2 When evaluating the models, nearly all of those trained with the metrics-only dataset, outperformed the models trained with metrics-and-smells. Considering the ROC curve, four of the metrics-only models outperformed the metrics-and-smells models, averaging a difference in area of 0.145 with a standard deviation of 0.051. In addition, when considering the PRG curve, all of the metrics-only models outperformed the metrics-and-smells models, with an average difference of 0.16 and a standard deviation of 0.115. Thus, training the model with smells and traditional object-oriented metrics dataset showed a slightly worst performance than when trained with only metrics. There was an exception when using the model with two hidden layers architecture with the tanh activation function, where the ROC curve evaluation showed a slightly better performance of 0.01 for the model trained with metrics-and-smells.

Answer to RQ3 When evaluating using the receiver operating characteristic, the autoencoder architecture that showed the best skill for the models trained with the metrics-only dataset and the smells-only datasets was the deep autoencoder with four hidden layers with ReLU activation functions, with the maximum area of 0.71 for the metrics-only and 0.52 for the smells-only dataset. However, for the models trained with the metrics-and-smells dataset, the autoencoder that fit the best was the deep autoencoder with six hidden layers with the ReLU activation function, where it showed an area of 0.64.

From these results, we were able to make a broader observation of the application of bad code smells in the subject of defect prediction. And, despite being less promising than we expected, there are still several routes that we could be explored. For instance, considering our goal to study the overall performance when using smell-based defect prediction trained with extremely imbalanced datasets, the obtained results state that applying a more general approach may have not been enough enough to use smells to create an accurate defect prediction model. Thus, future work could be focused on more concentrated domains such as categories of projects, or on a
specific developer style. In the end, we introduce a starting point for future research on this problem concerning extremely imbalanced datasets.

Furthermore, from the evaluation results, we can set a defect prediction performance baseline. Therefore, when evaluating future models that are trained using extremely imbalanced smell-based datasets, they can use the performance results we obtained from both the Receiver Operating Characteristic and the Precision-Recall-Gain as a comparison baseline. In general, we are providing an evaluation guideline for future smell-based defect prediction models and setting a ranking system for approaches created for this problem.

5. Future Work
We applied a coarser-grained approach to find defects using smells. Further on, we could apply a finer-grained approach. For a specific category of software projects or only a specific project, we could identify which smells have a higher relevance in predicting the defective modules. Additionally, we could attribute different weights to smells that are more suggestive of defects, based on the project or the programmer. We could also lower the level of granularity of module, from class-level to method-level. And, we could calculate the probability of a module being defective.

The pattern of the most common smells detected in software and the weight each smell may have on existing defects should vary from programmer to programmer. This means that, instead of developing a model whose smell detection depends only on the project, we can build a model that focuses more on the programmer and its style of coding. So it only trains considering the pattern of bad code smells that lead to errors particular to the programmer.

6. Conclusions
To explore the effects of design and implementation smells detection on defect prediction on an extremely imbalanced data, we trained a set of five autoencoders with different architectures using three datasets: traditional object-oriented metrics, design and implementation smells, and the combination of all features. Therefore, we discovered that the models trained with the metrics-only dataset performed much better than models trained with the smells. Also, we concluded that combining both smells and metrics did not improve any model trained with the metrics-only dataset, but showed an improvement over the smells-only models. Additionally, from comparing the architectures we used in our models trained with smells and the metrics-only datasets, we concluded that the best models were the deep networks. Although, the small differences in performances mean that the architectural decision does not have a significant impact on the models accuracy. Succinctly, our research showed that when following a coarse-grained approach, using only smells did not show a greater level of improvement compared against traditional object-oriented metrics. Nevertheless, we set a starting point and introduced a baseline for future research on smell-based defect prediction that could be further explored with a finer-grained approach or through the use of a wider range of smells.

References


