Exploring Designite for Smell-Based Defect Prediction

Bruno Sotto-Mayor[1] and **Amir Mishali**[1] and **Meir Kalech**[1] and **Rui Abreu** [2]
[1]Ben-Gurion University of the Negev, Beer-Sheva, Israel
e-mail: {machadob, amirelm}@post.bgu.ac.il
e-mail: kalechs@bgu.ac.il
[2]Faculty of Engineering of University of Porto, Portugal
e-mail: rui@computer.org

Abstract

Defect prediction is commonly used to reduce the effort from the testing phase of the software development. A promising strategy is to use machine learning techniques to predict which components of the software may be defective. Features are key factors to the success of the prediction, and thus extracting significant features can improve the model's accuracy. In particular, bad code smells are a category of those features that have been shown to greatly improve the prediction performance. However, Designite, a state of the art framework for bad code smells, has not been studied in the context of defect prediction. As such, in this paper we study the performance of defect prediction models by training multiple classifiers for 97 real projects, using as features the Designite and the traditional smells from the literature, and the combination of both. Moreover, we apply feature selection, to explore alternative combinations of the smells. We conclude that the models trained with both the Designite smells and the smells from the literature performed the best, with an improvement of 5% for the AUC score. Consequently, Designite smells are a good addition to the smells commonly studied in the literature for defect prediction.

1 Introduction

In recent years, there has been a higher demand for fast software delivery while maintaining a good product quality. Therefore, the trend of research has been focused on finding approaches to more effectively allocate resources on the software development pipeline. Since testing is one of the most time consuming phases, the attention has geared toward finding novel approaches to minimise the testing time, without degrading the software quality. In particular, one of those approaches is based on the application of defect prediction tasks that use prediction models to predict which software components are defective, thus facilitating the scheduling of resources to those components.

Commonly studied in the literature, defect prediction algorithms use the historical information of software, such as its previous versions (through version control tools) and its reported bugs (through issue tracking tools), as the source of predictors for the classification of defects. In particular, they extract a set of features from each component of a software repository and assign a target label to whether the component as defects. This creates the training dataset, which is used as input to a machine learning algorithm that will output a classification model. This will be use to predict which components in the following versions may be defective or not. In the end, these models have been shown to produce classifiers with a great predictive performance (Lessmann *et al.*, 2008; Nagappan and Ball, 2005).

A key factor to the success of the defect prediction models are the features extracted from the software repositories. One of those features, that has been studied for defect prediction is the set of bad code smells. They are patterns in the source code that indicate deeper underlying issues in the system (Fowler, 2019). Due to their high correlation with the presence of defects, they have been shown to be a good predictor candidate to build defect prediction classifiers (Piotrowski and Madeyski, 2020). Moreover, their static and relatively fast acquisition is a good motivator for their usage, and they also provide additional information for defect prediction when used in combination with other metrics.

Previous research on smell-based defect prediction studied the impact of different smells proposed in the literature. These include the works published by Martin Fowler (Fowler, 2019) and Brown (Brown, 1998), which introduce well known conceptual models for refactoring and bad code smells. In recent years, several papers were published, exploring the impact of those smells in defect prediction. Most of them, proved the correlation between defects and smells, and a positive impact on the use of smells in defect prediction (Piotrowski and Madeyski, 2020).

Although there is intensive research on the application of bad code smells in defect prediction, there are frameworks of smells that have not been explored in this context. In particular, the Design Code Smells proposed by Ganesh et al. (Ganesh *et al.*, 2013), a comprehensive catalog of 31 structural design smells, conceptualised with the four fundamental object-oriented design principles. Consequently, the authors developed a tool called Designite that implements 17 of those features (Sharma, 2018). In this paper, we empirically study the impact of Designite smells in defect prediction and we compare our results with traditional smells used in the literature.

Given the research goals, we mined 97 Apache repositories and extracted the Designite smells, as well as 20 other traditional smells used in the literature. In addition, we created datasets for each category of features and their combinations, thus we applied feature selection, trained several classifiers with those datasets, and selected and optimised the best models. In the end, the models trained with both Designite and the traditional smells performed the best, with a 5% gain of the AUC compared with a classifier trained with only the traditional metrics with a significance value of 0.05.

The rest of the paper is structured as follows. In Section 2, we discuss the related work. In Section 3 we define the problem definition regarding defect prediction and the code smells. In Section 4 we describe the evaluation methodology for the defect prediction classifiers. Lastly, in Section 5, we present and discuss the results from this study.

2 Related Work

Defect prediction is one of the most actively researched areas in software engineering. Its goal is to efficiently produce a list of defective-prone software components, which will allow developers to more effectively allocate their time and resources to those components (Paterson *et al.*, 2019).

The application of bad code smells in defect prediction has been studied in the literature. Recently, *Piotrowski et al.* (Piotrowski and Madeyski, 2020) performed a systematic literature review of 27 papers from 2006 to 2019, to analyse the relationship between smells and defects, as well as, to evaluate the performance of code smells in defect prediction using machine learning techniques. They confirmed that there is a positive correlation between code smells, hence they are a good indicator of defects and they influence positively the performance of defect prediction models.

From those studies, *Ma et al.* (Ma *et al.*, 2016) researched the possibility of improving defect prediction results by using code smells. They also evaluated whether they could use the defect prediction's results to prioritize code smells refactoring. They considered 8 smells, detected from a single version of four projects using the *DECOR* smell detection tool (Moha *et al.*, 2010). They found that results from the code smells detection improved the recall of fault prediction in all projects by $9\% \sim 16\%$.

Taba et al. (Taba *et al.*, 2013) proposed a set of antipattern metrics derived from the history of code smells within the version history of the software. They evaluated defect prediction models trained with 13 smells taken from multiple versions of Eclipse and ArgoUML (i.e, AntiSingleton, Blob, ClassDataShouldBePrivate, ComplexClass, LargeClass LazyClass, LongParameterList, LongMethod, MessageChain, RefusedParentBequest, SpaghettiCode, SwissArmyKnife and SpeculativeGenerality). In the end, they found that those smells produce higher density of bugs, compared against traditional metrics.

Palomba et al. (Palomba *et al.*, 2019; 2016), studied the effectiveness of the measure of severity of code smells (i.e., code smell intensity) as a predictor in defect prediction. They evaluated the predictive power of the intensity index of 6 code smells (i.e., GodClass, DataClass, BrainMethod, ShotgunSurgery, DispersedCoupling and MessageChains) by adding it to existing prediction models and comparing them against baseline metrics. In addition, they did an empirical comparison between their model and the antipattern metrics model suggested by *Taba et al.* (Taba *et al.*, 2013), to which they concluded that the intensity index always positively contributes to the state-of-the-art prediction models.

Although there are several studies that have explored the impact of code smells in defect prediction, there is a framework of code smells developed by *Ganesh et al.*(Ganesh *et al.*, 2013), based on the design principles of object-oriented programming, which has not been studied in the context of defect prediction. Furthermore, *Sharma* developed a tool called *Designite* (Sharma, 2018) that implements a subset of 17 of the proposed smells. In this paper, we aim to understand what contribution can the Designite smells introduce to the code smells already studied for defect prediction.

3 Problem Definition Methodology

In this section we define the application of bad code smells in defect prediction. We start by formally defining defect prediction and its process, from the extraction and construction of the datasets to the training and evaluation of the defect prediction classifiers (Section 3.1). Then, we describe the bad code smells, and their contribution to the defect prediction (Section 3.2).

3.1 Defect Prediction

The defect prediction goal is to predict which components on the next version of the software have defects. Formally, a software repository is usually composed by a sequence of n versions $V = \{v_1, ..., v_n\}$, consequently composed by a discrete number of components. In our study, we assume each component to be a file, where $v_i = \{f_1, ..., f_k\}$. As such, each instance of the mined datasets represents a particular file f_j in a particular version v_k of the software repository.

Given a particular software component, the classification problem of defect prediction is to determine what is the state of the said component - defective, or not. This state is described as the label attached to each instance of dataset, and it is dependent on the features used to train the classifier, which are extracted from the source code of each one of those instances. Consequently, one of the key aspects to achieving a good performance while predicting the target state, is the choice of the features (Moser *et al.*, 2008). Product metrics and process metrics are the most widely explored categories, vastly studied in the earlier times of defect prediction and they have generally showed positive results (Li *et al.*, 2018). While the product metrics describe the design and behaviour of the current state of the software (eg. CK (Chidamber and Kemerer, 1994) and McCabe's cyclomatic complexity (McCabe, 1976)), the process metrics extract the features from historical information stored in software repositories such as version control systems and issue tracking systems(eg. churn (Nagappan and Ball, 2005) and entropy metrics (Hassan, 2009)).

The typical approach for classification is the application of supervised machine learning algorithms on the data. This process begins with the generation of the datasets, in particular, by extracting the set of predictors from the components of each version of the software repository and attaching the corresponding defective information. For instance, in our study, the predictors are the Designite smells and the traditional smells used in the literature. Hereafter, the dataset is processed, accounting for missing values and scaling abnormalities in the data, and it is split into a training and a testing dataset. The training dataset is used as input to a learning algorithm, which outputs a classification model that predicts for a new un-labeled instance whether it is defective or not. After the creation of the classifier, the testing dataset is used to evaluate the model's performance by comparing the predicted classification against the actual classification. On the whole, the goal of the classification model is to define a mapping between the features and the target label.

To extract the target label from the software repositories, i.e. the defective information, we rely on the Jira issue tracking system and the Git version control system. Jira records all reported bugs and tracks changes in their status, associating each defect with an unique issue ID. Git tracks every modification done in the source files, to which it is a common practice for developers to associate the corresponding issue ID in the commit message. Therefore, we map the defective information of each file, in each commit, by associating the issue ID from both the commit message that fixed the defect and the reported defect in the issue tracking system. In the end, we label each file in the version as defective when there is a defect fixing commit that modifies the file, thus we assume that a file is fully defective if it was just modified in a fixing commit.

In the context of defect prediction, code smells have been shown to be both positively correlated with software defects and have been shown to positively influence the performance of defect prediction models used as features. (Piotrowski and Madeyski, 2020). As such, in the next section we introduce the topic of bad code smells.

3.2 Bad Code Smells

Bad Code Smells are defined as patterns in the source code that imply deeper problems in the system (Fowler, 2019). Their method of detection is based on the violations of fundamental design principles, that negatively impact design quality. As such, they imply weaknesses in the design, which, although not technically incorrect, may lead to a slower development and increase the risk of defect production. Consequently, they can contribute to the accumulation of technical debt, which can lead to a technical bankruptcy, rendering the project unmaintainable, thus having to be abandoned in the end (Suryanarayana *et al.*, 2015; Tufano *et al.*, 2015).

Traditional Code Smells In the literature, the most common smells are those proposed by *Fowler et al.* (Fowler, 2019) and *Brown* (Brown, 1998). *Fowler et al.* (Fowler, 2019) was the one to introduce the notion of code smells. Their main purpose was to determine when should the developer do a specific code refactoring. As such, they defined 22 representations of code smells, each associated to a set of refactoring methods. For example, the *Shotgun Surgery* smell occurs when, during a change, the developer needs to do several small changes on different classes. This makes them harder to find, thus making it easier to miss an important change. Furthermore, Brown proposes a list of negative patterns that cause development roadblocks and categorised them for the different roles of software development: management, architectural, and development. Brown's main motivation is to accurately describe commonly occurring situations, their consequences and solutions, related to three perspectives. As an example, the Swiss Army Knife is a management perspective anti-pattern that describes the overdesign of interfaces. It results in objects with numerous methods that attempt to anticipate every possible need, thus leading to designs that are harder to comprehend, use and debug.

However, since the smells defined by Fowler and Williams are only conceptual descriptions of the patterns, other works make effort to define formal methods of detection based on their description. The main idea is to define rules using code metrics, define associations between them, and set thresholds. Moreover, because the thresholds are qualitatively defined (e.g. few), a common approach is to get the distribution of values and assign quantiles to each qualitative threshold. As a consequence of the definition of rules, several tools have been developed to the extraction of smells. In our study, we used the Organic Project[1] which is based on Bavota et al. rules (Bavota *et al.*, 2015). We also used DECOR rules to detect smells and adapted it to work with Java files (Moha *et al.*, 2010). In the end, we extracted 20 smells to represent the traditional smells used in the literature.

Designite Code Smells The smells we are studying were proposed by *Ganesh et. al* (Ganesh *et al.*, 2013). They published a comprehensive catalog of 31 design smells, classified based on the violation of one of the four fundamental OO design principles. Their main motivation was the lack of common ground for the definition of smells, since while some treated a smell as a problem itself, others considered it to be an indicative of a deeper problem. Therefore, their goal was to create a framework that defines a taxonomy for all documented smells and to organize them under the perspective of the violation of a design principle. They base those violations on the fundamental design principles from the four major elements of the "object model" defined by *Booch et al.* (Booch and Booch, 2007): abstraction, encapsulation, modularity, and hierarchy. With this, *Sharma* developed Designite (Sharma, 2018), a code assessment tool written in Java that detects 17 of the design smells mentioned above.

In summary, we used bad code smells as features for defect prediction. We considered 20 smells commonly used in the literature as our baseline, extracted using the Organic and DECOR rules. We, also, considered 17 smells from Designite [2] to which we are studying whether it increases the defect prediction performance. Therefore, for each instance of the training dataset, we ended up with 37 boolean features, each describing whether it was detected or not in the particular file. Take notice that since we used different tools to extract the smells, we merged the results from the different tools by the name of the file, thus, for missing files (instances), we assumed that none of smells were detected, so we defined the respective instance smells values to *False*.

4 Evaluation Methodology

Our research goal is to study the impact of Designite smells in defect prediction. Therefore, we designed our study to empirically compare the performance of defect prediction classifiers trained with the Designite smells against the smells traditionally used in the literature. As such, we evaluate the different models trained with both categories of smells individually, and then with the combination of the different features by applying feature selection techniques. With this in mind, we defined the following research questions.

> **RQ .1**: *Do defect prediction models trained with Designite code smells outperform those trained with traditional code smells?*

[1]https://github.com/opus-research/organic.git

[2]The list with all the smells used in this study and their description will be available after the paper's publication.

RQ .2: *Do Designite code smells contribute to the performance of models trained with traditional code smells?*

RQ .3: *Which combinations of code smells and metrics outperform the defect prediction models?*

We divide this section following the same approach, commonly used in defect prediction studies. As such, we start by collecting the data from repositories, which includes smells, metrics and defects information. Then, we apply feature selection whose purpose is to examine which features influence the best the defect prediction. Next, we train classification models to predict defects based on several algorithms and optimise them with hyper-parameterization. Last, we cross-validate the models and evaluate them using different classification metrics. Each step is represented by the following subsections.

4.1 Dataset Construction

The first step of our approach is to collect the data and to generate the datasets required for training and testing of the classifiers. Therefore, we start by iterating the versions of each project and extract the defective information, i.e. whether each file has defects or not, as well as the target features.

Then, we preprocess the datasets for training, in particular, we handle the missing information, standardise the data and deal with data imbalance. In the end, we split the data into training and testing dataset, and split the training dataset to account for the validation process.

Data Collection We collected the data from 97 projects[3], in particular we selected 5 versions from each project and applied the feature extraction approach on a file granularity. The criteria to select the versions is based on the ratio of defects in each file, as such, we opt for versions with ratios between 10% and 30% of defects, since it composes a good representation of defects that reduces the class imbalance, that is produced by the low number of defects, and it is not an outlier, for instance, a version that was created just to fix issues. The versions were selected from 97 Apache projects written in Java, and they represent different populations in regard to authors, number of contributors, number of commits and release times. In the end, we collected features from a high number of versions, representing a big diversity of projects and a good representation of defects for each version.

Extracting Defects After selecting the versions, the following step is to analyse the source code and collect the features under study and target variable. As a first step we identified which files are defective and which are not. In our study, we assume that a file to be defective needs to have at least one defect in it. We used the Jira tracker from Apache to collect the defects information, thus mapping a boolean value for each file asserting whether it is defective or not. Using the Jira tracker is used in the previous step to calculate the ratios in the version selection and, then used in this step to define the target variable in the classification for each file for all versions.

Extracting Designite Smells We extracted the main feature under study, by running Designite on the source code of each version. It produces a mapping for each class where, for every smell, it describes whether it was detected or not. Therefore, we abstracted the class to file granularity and assumed that if it was detected in a class, it was detected in the file.

Extracting Traditional Smells We extracted the smells commonly used in the literature for smell-based defect prediction, by using Organic[4] to analyse the existence of code smells in the source code. This tool is an eclipse plugin that detects 11 types of smells based on the rules implemented by Bavota et al. (Bavota *et al.*, 2015), consequently we adapted and extended it to be able to detect all the smells mentioned in the previous section. In the end, we were able to detect 20 traditional smells both from class and method granularity, therefore we inferred that if a method or a class detects a smell, then the whole file detects it.

Feature Selection In addition to comparing the defect prediction classifiers trained by each individual feature category, we aim to explore the impact of the combination of Designite code smells with the traditional code smells. Therefore, we applied the previously defined approach on the generation of datasets to the combination of the different clusters of features. Hence, we trained classifiers with both Designite smells and the traditional smells and applied feature selection to study whether it improves the defect prediction performance and to study which feature selection methods selects the subset of features that produce the most accurate defect prediction classifiers. Consequently, we applied the following three feature selection methods based on univariate statistical tests, while selecting both 20 and 50 percentile of features: the chi-square test; the ANOVA F-value; and the mutual information. In addition, we also applied recursive feature elimination with 5 fold cross validation and based on the F1 Score.

Dataset Preprocessing From the generated datasets, we applied preprocessing operations to prepare and split the data to create the defect prediction classifiers. One of the first required operations is to handle missing values, as such, we dropped rows with missing values. Then, for the numerical metrics, due to the high value ranges, we applied the min max scaler to scale and translate each feature to a range between 0 and 1. Since there is a higher ratio of non-defective files than defective files, which is a common occurrence in defect prediction, we applied the SMOTE oversample technique to synthetically increase the ratio of defective instances in the datasets. Lastly, we split the data into a training and a testing dataset, in particular we allocated the first four versions of the project to training and the last version to testing.

Training Classifiers With the datasets generated from the collection step we trained the defect prediction classifiers. Because we are looking to assess the feature sensitivity of Designite smells, we experimented with a large set of classifiers, and we applied hyper-parameterization with cross validation to validate and select the classifiers and configurations with the best performances. Then we selected the ten highest performance classifier configurations for each dataset and train them, thus creating defect prediction classifiers for each dataset.

Since the goal of our study is to evaluate the impact of Designite smells, we trained several classifiers targeting different categories of classification. The rationale is to focus on the sensitivity of the features thus we explore a large num-

[3]The list with the information from the 97 projects used in this study will be available from the paper's publication

[4]https://github.com/opus-research/organic.git

ber of classifiers with different parameter configurations and select the highest prediction scores. As such, we classify with three statistical classifiers: Linear discriminant analysis, quadratic discriminant analysis, and the logistic regression; and with a bernoulli naive bayes classifier; k-nearest neighbours classifier; decision tree and a random forest classifiers; support vector machine; and neural network: a multilayer perceptron. For the practical application of these classifiers, we used the scikit learning tool, which provides support for each one of them.

We first applied a grid search with 10 fold cross validation on all the classifiers to find the best classifiers for the target dataset and to optimize the parameters for higher performance. We used F1 Score as the optimisation score, since it is the most fitted for binary targets by its definition. To do so, we used scikit learn's GridSearchCV tool, which makes an exhaustive search over the specified parameters for each classifier we analysed. After obtaining the score for all the combinations of parameter for each classifier, we selected the ten configurations with the highest F1 Score for each dataset. This way, we can reduce the number of classifiers to be trained to the evaluation and comparison step, while having a broad cover of different classifiers.

4.2 Data Analysis and Metrics

Having created the classifiers for each of the datasets, the next step is to evaluate each trained model from each configuration. Therefore, we calculated different metrics commonly studied in defect prediction and compared them to analyse the features combinations we are evaluating. Moreover, since we took into account the ten optimal classifier configurations in the previous step, to analyse the results, we consider the best score from each dataset and we summarise their scores by their average. The evaluation metrics we calculated to measure the performance of each classifier are: precision, recall, F1 Score, AUC-ROC and the Brier Score. We discuss their rationale in the rest of the section.

Precision and recall are two widely used metrics in defect prediction. They measure the relationships between specific parameters in the confusion matrix:

$$precision = \frac{TP}{TP+FP} \qquad recall = \frac{TP}{TP+TN} \tag{1}$$

where, TP is the number of classes containing defects that were correctly predicted as defective; TN describes the number of non-defective classes that were predicted as defective; and FP is the number of classes where the classifier fails to predict defective classes, by declaring non-defective classes as defective. In addition to these scores, we calculated the F1 Score which is the harmonic mean of both precision and recall, defined as follows:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{2}$$

Moreover, we determine the Area Under the Curve (AUC) of the Receiver Operating Characteristic curve. AUC summarises the ability of the classifier to discriminate between defective and non-defective classes. As such, the closer it is to 1, the higher the skill of the classifier to discern the classes affected or not by the defect. Nevertheless, a score closer to 0.5 describes a classifier with a lower accuracy, thus having a classification ability closer to a random classifier.

Furthermore, we evaluated the classifiers using the Brier Score. This score measures the distance between the probabilities predicted by a model and the actual outcome. Accordingly, the Brier Score is defined as follows:

$$\frac{1}{N}\sum_{i=1}^{N}(p_c - o_c) \tag{3}$$

where p_c is the predicted probability by the classifier and o_c is the actual outcome for the class c. Therefore the distance will be 0 if the class is not defective and 1 otherwise; N is the total number of classes in the dataset. In the end, a low Brier Score represents a good classifier performance, while a high score represents a low performance.

5 Results

In this section, we discuss the obtained results focusing on the research questions we initially defined. As such, we first analyse the results of the models trained with each category of smells, and then we analyse the results from the combination of all smells, including the subsets of smells.

To address **RQ.1** we evaluated whether the performance of our models trained with Designite code smells would outperform those trained with the traditional code smells. As such we calculated the arithmetic mean and the standard deviation scores of all projects considering the best classifier configurations.

In Figure 2 are illustrated the arithmetic means for all the scores representing from the comparison between Designite models and the Traditional models. We can observe that, although, there is a slight improvement of its precision, we can see that on the overall, the Traditional smells outperform the Designite smells. Moreover, we are able to see that the performances from Designite are low, for instance the AUC measure is very close to the score of the random model.

Regarding **RQ.2**, we studied whether the performance of the models trained with both smells sets will perform better than any of the single categories. Since we saw that the traditional smells outperform the models trained with Designite smells, we will consider those as the baseline to compare the combination of smells. Moreover, we will also verify whether there is a subset of the combination of smells resulting from the feature selection that can outperform the models trained with all the smells.

Figure 2 also describes the arithmetic mean of the score for the combination of smells. Overall, we can verify that there is an increase in the performance of the combination of both smells compared against the Traditional smells. This is more prominent in the AUC metric, where we can see an improvement of 5%. In particular, this difference is better represented in the distribution presentation in Figure 1. The figure illustrates a violin plot and a box plot for the comparison of the different scores: AUC ROC, the Brier Score, and the F1 Measure, between the models trained with both Designite and the Traditional smells, and those only trained with Traditional Smells. The symmetric lines delineating the violin plot represent the rotated probability density of the score on all projects, i.e the score density distribution for all projects. Moreover, the box plot describes a standardized summary of the score for all projects through the minimum, the maximum, the sample median, and the first and third quartiles. In particular, the middle line denotes the range from the minimum score to the maximum score, excluding

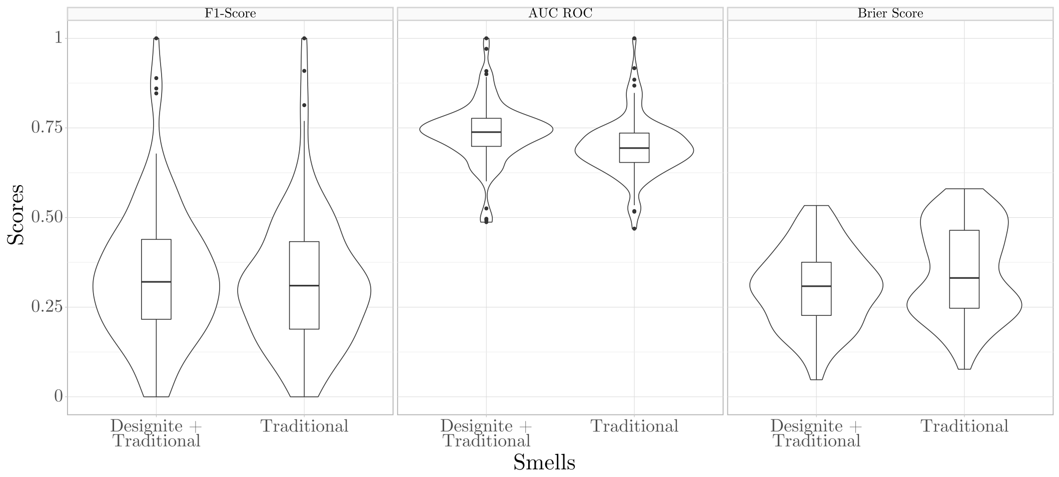

Figure 1: Distribution of the models trained with both **Designite and Traditional Smells** compared with only **Traditional Smells** for three scores: AUC ROC, Brier Score and F1 Measure

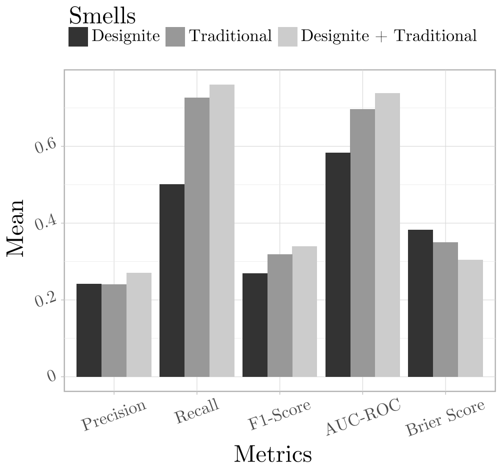

Figure 2: Score comparison between Designite Models, Traditional Models and their combination for the best classifier of each project.

the outliers which are represented as the dots. The upper and the lower limit of the box define the upper quartile $qn(0.75)$ and the lower quartile $qn(0.25)$ of the dataset. The middle line represents the median of the scores $qn(0.50)$ and describes the middle value of the dataset.

The precision and the recall also show better performance for the combined smells. We can verify that the recall is significantly larger than precision for both variables, with results close to 76% for the combined smells. The arithmetic mean of the F1 Scores is also better for the combined smells, although not as significant in the distribution (Fig. 1). Lastly, the Brier Score is 4.6% lower for the combination of smells, which means that it has a higher performance than the Traditional smells separately.

Regarding the distribution, the combined smells show a normal distribution, while in the Traditional smells distribution the results are somewhat divided between higher scores and lower scores. We can see that the lower scores in the Traditional smells actually perform better than the peak of the normal distribution of the combined smells. This could imply two possible clusters in the projects that could be further studied in future work. In the end, we can verify an improvement of the performance of the models when both Designite smells and the traditional smells are trained together.

Furthermore, we checked statistical significance with t-test and confirmed that for all the scores, the models trained with Designite and the Traditional smells combined, have higher performances than the models trained with only Traditional smells, at a significance level of $p < 0.05$.

In regards to the **RQ. 3**, we applied seven different methods to identify the subsets of features with the best performances. We observed that for the AUC ROC, feature selection produced a slight improvement of the models performances compared against the original models, with all features of Designite and Traditional code smells. The best selection is from the recursive elimination, with an improvement of 0.6%, then the ANOVA F-value with 20 and 50 percentile had the respective improvements of 0.4% and 0.3%, and lastly the mutual information with 50 percentile with 0.1% of improvement. Additionally, for the F1 Score, there was also just a slight improvement over all the combinations. The selection from mutual information with 50 percentiles increased the performance by 0.3%, followed by the recursive elimination and the chi-squared test with 20 percentiles, with an increased performance of 0.2%. For the Brier Score, there was an improvement of 3.6% when applied the chi-squared with 20 percentiles, followed by an improvement of 1.6% with the smells selected from the mutual information with 20 percentiles and an improvement of 2.9% with the ANOVA F-value with 20 percentiles. In the end, although there was an overall improvement when applying feature selection, it was not significant compared against using all smells.

In summary, the results from the study of the impact of Designite smells in defect prediction are the following:

- The models trained with both Designite and the Traditional smells outperform the models trained with only Traditional Smells on all scores, with a significant increase of the AUC of 5% ($p < 0.05$).
- Applying feature selection to the combination of both Designite and Traditional smells produce models with a slightly higher accuracy than models trained with all

the smells. For instance, the Brier Score had the most significant increase of $1.6\% \sim 3.6\%$

6 Conclusion

In this study we evaluated the code smells from Designite as a predictor for defect prediction. As such we compared with the models trained with smells taken from previous studies (traditional smells). We extracted the smells from 97 projects and used a broad range of classifier algorithms to train models with the Designite smells, the Traditional smells and the combination of both. In the end we evaluated and retrieved the scores from the best classifiers, and compared the performance of the different smells.

The results showed that Designite code smells are not good enough to be used for defect prediction on their own. However, when used in combination with other traditional smells used in the literature, they improve the performance of the defect prediction compared against each individual category. The current results suggest that for future work we should study the impact of the different categories of Designite code smells. In addition, we should study the factors that influenced the improved results for the combination of Designite and the traditional smells, against each one of the individual sets of smells.

References

(Bavota *et al.*, 2015) Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. An experimental investigation on the innate relationship between quality and refactoring. 2015.

(Booch and Booch, 2007) Grady Booch and Grady Booch. Object-oriented analysis and design with applications, 2007.

(Brown, 1998) William J. Brown. AntiPatterns: Refactoring software, architectures, and projects in crisis, 1998.

(Chidamber and Kemerer, 1994) S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. 20(6):476–493, 1994.

(Fowler, 2019) Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Signature Series. Addison-Wesley, second edition edition, 2019.

(Ganesh *et al.*, 2013) S.G. Ganesh, Tushar Sharma, and Girish Suryanarayana. Towards a Principle-based Classification of Structural Design Smells. 12(2):1:1, 2013.

(Hassan, 2009) Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *2009 IEEE 31st International Conference on Software Engineering*, pages 78–88. IEEE, 2009.

(Lessmann *et al.*, 2008) S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings. 34(4):485–496, 2008.

(Li *et al.*, 2018) Zhiqiang Li, Xiao-Yuan Jing, and Xiaoke Zhu. Progress on approaches to software defect prediction. 12(3):161–175, 2018.

(Ma *et al.*, 2016) Wanwangying Ma, Lin Chen, Yuming Zhou, and Baowen Xu. Do We Have a Chance to Fix Bugs When Refactoring Code Smells? In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 24–29. IEEE, 2016.

(McCabe, 1976) T.J. McCabe. A Complexity Measure. SE-2(4):308–320, 1976.

(Moha *et al.*, 2010) N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. 36(1):20–36, 2010.

(Moser *et al.*, 2008) Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 13th International Conference on Software Engineering - ICSE '08*, page 181. ACM Press, 2008.

(Nagappan and Ball, 2005) Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering - ICSE '05*, page 284. ACM Press, 2005.

(Palomba *et al.*, 2016) Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 244–255. IEEE, 2016.

(Palomba *et al.*, 2019) Fabio Palomba, Marco Zanoni, Francesca Arcelli Fontana, Andrea De Lucia, and Rocco Oliveto. Toward a Smell-Aware Bug Prediction Model. 45(2):194–218, 2019.

(Paterson *et al.*, 2019) David Paterson, Jose Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 346–357. IEEE, 2019.

(Piotrowski and Madeyski, 2020) Paweł Piotrowski and Lech Madeyski. Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review. In Aneta Poniszewska-Marańda, Natalia Kryvinska, Stanisław Jarząbek, and Lech Madeyski, editors, *Data-Centric Business and Applications*, volume 40 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 77–99. Springer International Publishing, 2020.

(Sharma, 2018) Tushar Sharma. DesigniteJava, 2018.

(Suryanarayana *et al.*, 2015) Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2015.

(Taba *et al.*, 2013) Seyyed Ehsan Salamati Taba, Foutse Khomh, Ying Zou, Ahmed E. Hassan, and Meiyappan Nagappan. Predicting Bugs Using Antipatterns. In *2013 IEEE International Conference on Software Maintenance*, pages 270–279. IEEE, 2013.

(Tufano *et al.*, 2015) Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 403–414. IEEE, 2015.