Soft Computing (2021) 25:14171–14181
https://doi.org/10.1007/s00500-021-06254-7

DATA ANALYTICS AND MACHINE LEARNING

Cross-project smell-based defect prediction

Bruno Sotto-Mayor[1] · **Meir Kalech**[1]

Accepted: 9 September 2021 / Published online: 4 October 2021
© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2021

Abstract
Defect prediction is a technique introduced to optimize the testing phase of the software development pipeline by predicting which components in the software may contain defects. Its methodology trains a classifier with data regarding a set of features measured on each component from the target software project to predict whether the component may be defective or not. However, suppose the defective information is not available in the training set. In that case, we need to rely on an alternate approach that uses the training set of external projects to train the classifier. This approached is called cross-project defect prediction. Bad code smells are a category of features that have been previously explored in defect prediction and have been shown to be a good predictor of defects. Code smells are patterns of poor development in the code and indicate flaws in its design and implementation. Although they have been previously studied in the context of defect prediction, they have not been studied as features for cross-project defect prediction. In our experiment, we train defect prediction models for 100 projects to evaluate the predictive performance of the bad code smells. We implemented four cross-project approaches known in the literature and compared the performance of 37 smells with 56 code metrics, commonly used for defect prediction. The results show that the cross-project defect prediction models trained with code smells significantly improved 6.50% on the ROC AUC compared against the code metrics.

Keywords Cross-project defect prediction · Defect prediction · Code smell · Mining software repositories · Software quality · Software engineering

1 Introduction

In the software development process, assuring software quality is a crucial requirement to accommodate today's expectations of software delivery. Within the proposed techniques to optimize the delivery time of software while maintaining its quality, defect prediction was shown to be a promising approach, considering its application to test case prioritization and selection (Paterson et al. 2019). Defect prediction uses the historical information of software, such as the source code in its previous versions and the defective information through the reported bugs, to make predictions about the location of bugs in the code. This is made using classification algorithms, which, when given a training set, learn to label each software component as defective or not. This is called within-project defect prediction.

✉ Bruno Sotto-Mayor
machadob@post.bgu.ac.il

Meir Kalech
kalech@bgu.ac.il

[1] Ben-Gurion University of the Negev, Beer-Sheva, Israel

Beyond the traditional application of defect prediction, which assumes natural access to the past information of the software, in a practical industrial scenario, this information may not be available. It is common for companies not to maintain a clear historical data record of defects or not contain sufficient data from previous versions of a project (Kitchenham et al. 2007). In addition, historical data are not available for new projects. This limits the creation of a training set capable of building a classifier fit to predict which components in the software may be defective. Therefore, research accommodated this problem by introducing a variant approach to defect prediction, called *cross-project defect prediction*, that builds the training set from external sources to the project (Goel et al. 2017).

Several studies were proposed, introducing new approaches for cross-project defect prediction. They leverage the training set from other projects, for instance, by either homogenizing the respective training set or target set (Watanabe et al. 2008), or selecting the training instances closest to the target project (Turhan et al. 2009). Moreover, the categories of metrics used to build the cross-project models were based

on source code metrics, commonly studied in the classical problem of within-project defect prediction (Herbold et al. 2018).

A particular category of code metrics are *bad code smells*, which are patterns of bad code design and implementation that increase the technical debt of software; thus, they lead to defects and high-cost maintenance tasks (Suryanarayana et al. 2015). Although there is a considerable study on the application of cross-project defect prediction for code metrics, the respective evaluation for bad code smells has not been deeply explored. Therefore, **the goal of this study is to evaluate the impact of bad code smells for cross-project defect prediction**, compared with the code metrics already studied in the literature in this context.

For evaluation, we implemented four cross-project defect prediction approaches and built cross-project models using those approaches on five classifiers. Then, we trained them using bad code smells and code metrics as the features extracted from five versions of 100 open-source projects written in Java. All projects were used as target projects, as well as training projects for the other respective projects. We examine whether the performance of both sets of features—bad code smells and code metrics—when used together to build the models improves the performance of the cross-project defect prediction.

We conclude that cross-project defect prediction models trained with code smells outperform the models trained with code metrics, with an average improvement of 6.50% on the ROC AUC. Therefore, by evaluating the improvement of each individual project, we observed that in 76% of the projects, there was an improvement for the code smells compared with code metrics. Moreover, we observed that the combination of both sets of features does not produce higher predictive models than the models trained with only code smells.

The rest of the paper is outlined as follows. In Sect. 2, we discuss the related work. In Sect. 3, we introduce the problem definition concerning cross-project defect prediction, and bad code smells. In Sect. 4, we describe the evaluation methodology to evaluate the impact of the bad code smells. Lastly, in Sect. 5, we present and discuss the results from this study.

2 Related work

This section discusses the related work for within and cross-project defect prediction and its application using bad code smells as features.

Defect prediction is an active research topic in software engineering. It is commonly used to select and prioritize test cases to optimize the software testing phase while maintaining its quality. As shown by Paterson et al. (2019), to produce good results when the test cases are prioritized using within-project defect prediction, i.e., the defect prediction model is trained using resources from the same project.

Over the years, several models and implementations were introduced for defect prediction research. In a broad view, Arpitha Kotte (2021) published a very recent study with a comprehensive survey of the different models proposed for defect prediction.

Several studies were published for within-project defect prediction that thoroughly evaluate the impact of different metrics as predictors for defects classification (Li et al. 2018); for instance, code and process metrics. Code metrics are measured directly from the source code, e.g., size or complexity, and process metrics are extracted from the historical information of the software repositories (Radjenović et al. 2013).

A different set of metrics that has also been thoroughly studied in the literature is the set of bad code smells. Piotrowski and Madeyski (2020) did a systematic literature review of 27 papers from 2006 to 2019 that analyze the relationship between smells and defects and their impact on defect prediction. In the end, they identify an overall positive correlation between code smells and defects; hence, bad code smells are a good indicator of defect prediction.

However, these studies only focus on models for **within-project defect prediction**. When we remove the assumption that the project contains a clear historical record of defect data or enough information from its previous versions, we do not have enough information to build models with enough capability to perform defect prediction (Kitchenham et al. 2007).

Therefore, a new class of defect prediction models was created that uses information from other projects to complement the missing information of the target project. Porto et al. (2019) perform a systematic review, and experimental comparison of 31 CPDP methods of cross-project defect prediction, and Herbold et al. (2018) propose a benchmark for cross-project defect prediction, thus replicating 24 approaches and experimentally comparing their performance. They observed that the standardization approaches performed significantly well, particularly the ones proposed by Cruz and Ochimizu (2009) and Watanabe et al. (2008). Moreover, Turhan et al. (2009) propose an approach that also reports good performances; it builds the training set by selecting the instances from the cross-projects that are closer to each instance of the target project.

Due to their good performance, these were the approaches we applied to evaluate the impact of bad code smells as predictors for cross-project defect prediction. Furthermore, we also considered one approach proposed by Guo et al. (2016), called *best-of-breed*, that uses the training set of the project with higher F2-score classification when evaluated with the target project. We also considered the approach from Bal

(2018), that introduces an extreme learning approach to train the cross-project defect prediction model.

Beyond the use of code metrics for cross-project defect prediction, only one project studied the application of bad code smells as predictors for classification. Taba et al. (2013) proposed metrics based on code smells to study the effect of code smells on the density of defects in files and study the impact of those metrics on traditional defect prediction models. This study was the only one we identified that uses code smells as predictors for cross-project defect prediction. They used a straightforward approach of training the model with the other cross-projects and then testing with the target project. Although Taba et al. (2013) evaluate the performance of smells, it is limited to evaluating only four metrics based on the frequency and entropy of smells, which, by no means, equates to the metrics used in our study. Moreover, we apply, in addition to the approach used by the authors, three cross-project approaches, thus providing additional robustness for the results from the cross-project defect prediction evaluation.

3 Problem definition and methodology

In this section, we introduce the application of bad code smells as predictors for cross-project defect prediction. We start by formally defining defect prediction and its process, from the extraction and construction of the data sets to the training and evaluation of the defect prediction classifiers (Sect. 3.1). Then, we extend the definition to accommodate the cross-project use case; thus, in Sect. 3.2 we define the problem that motivates the cross-project defect prediction approach and the approaches we applied in our study. Last, we describe the bad code smells and their contribution to the cross-project defect prediction (Sect. 3.3).

3.1 Defect prediction

The goal of defect prediction is to predict the location of defects in the succeeding version of a specific project. Viewed formally, given a software repository with n versions $V = \{v_1, ..., v_n\}$, each composed by a discrete number of components defined in our study by every file f_j in a particular version v_k; the classification problem is to determine whether the state of that particular component f_j is defective or not. This state is described as a label assigned to each instance of the data set. Moreover, it is dependent on the features extracted from the source code, which are used to train the classifier. Thereupon, the selection of the features is a crucial step for achieving good performance while predicting the target state (Moser et al. 2008).

Product and process metrics are the categories of metrics most widely studied in defect prediction; thus, they have generally shown positive results (Li et al. 2018). The product metrics measure the design and behavior of the current state of the software, for example, the CK metrics (Chidamber and Kemerer 1994) and McCabe's cyclomatic complexity (McCabe 1976). The process metrics measure the historical information stored in the software repositories, using both version control systems and issue tracking systems. For example, the churn metrics (Nagappan and Ball 2005) and entropy metrics (Hassan 2009).

Table 1 Cross-project defect prediction approaches applied in this study

Approach	
Standard	Taba et al. (2013)
KNN	Turhan et al. (2009)
ELM	Bal (2018)
Best Of Breed	Guo et al. (2016)

The usual approach for defect prediction classification is the application of supervised machine learning algorithms on the data. It starts with the generation of data sets accrued from the features extracted in each component of the versions in the software repository, additionally attached with the label describing whether the component is defective. Then, the data set is processed. It deals with missing values, scales abnormalities, and splits the data into training and testing sets. The training set is used as input to a learning algorithm that outputs a classification model capable of predicting whether a new unlabeled instance is defective. The testing set is used to measure the model's performance by comparing the predicted classification with the true classification to evaluate the output model. Altogether, the goal of the classification model is to define a mapping between the feature and target label.

3.2 Cross-project defect prediction

From the definition of defect prediction, the dependent variable, i.e., the target label we aim to predict, has a critical role in the learning process. To such a degree that learning is only possible if the defective data are available. However, in a practical scenario, it is not guaranteed that the historical defective data are correctly maintained, and it is available from the project history (Kitchenham et al. 2007). A solution is to build the training set from features extracted from external projects with known defect information to handle this adversity. This approach is called cross-project defect prediction, and it solves the lack of historical defective data. Nevertheless, it introduces heterogeneity on the data, leading to a decrease in the efficiency of the defect prediction models (Zimmermann et al. 2009). However, we accounted for this in our methodology and we selected four approaches that

were reported to produce good results (Herbold et al. 2018; Porto et al. 2019; Bal 2018; Taba et al. 2013). In Table 1, we list the approaches applied in this study.

The trivial approach to cross-project defect prediction is the direct application of the cross-project (i.e., external project) data set as the training set for the target project (i.e., the project that is missing the defective information). Therefore, it is a simple variation of the process described in the Defect Prediction definition (Sect. 3.1), where the data set generation is accrued from an external project. We designated this approach the *Standard Approach*. Another approach is the *Best-Of-Breed Approach*. Instead of being manually attributed to a specific cross-project, it applies a majority voting on a set of candidate cross-projects, selecting the project with the highest F2-score (Guo et al. 2016). This approach aims to identify the projects with the highest similarity to the target project. Then, Turhan et al. (2009) proposed the *KNN Approach* which, also from a set of candidate cross-projects, selects the instances from all projects that are most similar to each instance from the data set of the target project. This approach uses the K-nearest neighbors algorithm to identify the K closest cross-project instances to each instance of the target project. Last, the *ELM Approach* is a variant of the *Standard Approach*, where instead of using common classifiers used in defect prediction, it uses extreme learning machines (Bal 2018). These are feed-forward networks reported to produce good generalization performances and learn thousands of times faster than back-propagation networks (Huang et al. 2006).

3.3 Bad code smells

Bad code smells are patterns in the code that indicate underlying problems in the design and implementation of a system (Fowler and Beck 1999). The process of code smells detection is based on the violation of fundamental design principles that undermine the quality of a system. Therefore, they draw out weaknesses in the system's design and implementation to which, although not technically incorrect, may cause a slower development and increase the likeliness of introducing defects. Consequently, it leads to an accumulation of technical debt, which may cause a technical bankruptcy of the project, rendering it unmaintainable; therefore, having to be abandoned at the end (Suryanarayana et al. 2015).

The set of code smells commonly studied as predictors for defect prediction are the ones proposed by Fowler and Beck (1999) and Brown (1998). As identified in the literature review published by Piotrowski and Madeyski (2020), the authors collect published studies relating bad code smells as predictors for defect prediction and draw conclusions to the correlation between code smells and defects. Most of the reviewed studies featured the code smells proposed by the two previously cited sources. Fowler and Beck (1999) introduce the notion of code smells and define them as the trigger to the application of code refactoring. One example is the *Shotgun Surgery* smell which occurs when a single change leads to several minor changes on different components. This hinders the software development and maintenance tasks, as the locations of the new changes are hard to keep track of. Moreover, Brown (1998) define a list of anti-patterns that are the source of development roadblocks and categorize them for the different roles of software development: management, architectural, and development. For example, the *Swiss Army Knife* is a management anti-pattern that describes the over-design of interfaces. It is detected when objects with numerous methods attempt to anticipate every possible need, thus causing the construction of designs that are hard to comprehend, use, and debug.

Beyond these code smells another set of smells proposed by Suryanarayana et al. (2015) tackle code issues in the perspective of four fundamental design principles of object-oriented programming introduced by Booch and Booch (2007): abstraction, encapsulation, modularity, and hierarchy. These smells were formulated from the generalization of smells proposed in the literature. In particular, to a framework that follows the violation of those design principles. One example of these smells is *Deficient Encapsulation*. It occurs when the accessibility of one or more members of an abstraction is more permissive than required, for instance, a class that sets its fields as public.

In our study, we evaluate the impact of bad code smells for cross-project defect prediction. We used 37 code smells, from the three different-sources: Fowler and Beck (1999), Brown (1998), and Suryanarayana et al. (2015). Moreover, to evaluate the performance of the cross-project models, we compared the smell-based models with cross-project defect prediction models trained with product metrics which from now on we will designate as *code metrics*. Henceforth, we used 56 code metrics. In Table 2, we list the bad code smells evaluated in this study, and in Table 3 we list the code metrics we used to compare our cross-project models.

4 Evaluation methodology

Our research goal is to evaluate the impact of bad code smells as predictors for cross-project defect prediction. We set up our study to empirically compare cross-project defect prediction classifiers' performance with bad code smells against those trained with code metrics, commonly used in cross-project defect prediction research (Herbold et al. 2018). Therefore, we compare the performance of the bad code smells to the performance of code metrics. We reason to whether the combination of both features' sets improves the

Table 2 Listing of the bad code smells utilized in this study

Code smells	
Imperative abstraction	Multifaceted abstraction
Unnecessary abstraction	Unutilized abstraction
Deficient encapsulation	Unexploited encapsulation
Broken modularization	Cyclic-dependent modul.
Insufficient modularization	Hub-like modularization
Broken hierarchy	Cyclic hierarchy
Deep Hierarchy	Missing hierarchy
Multi-path hierarchy	Rebellious hierarchy
Wide hierarchy	God class
Class data should be private	Complex class
Lazy class	Refused bequest
Spaghetti code	Speculative generality
Data class	Brain class
Large class	Swiss army knife
Anti singleton	Feature envy
Long method	Long parameter list
Message chain	Dispersed coupling
Intensive coupling	Shotgun surgery
Brain method	

Table 3 Listing of the product code metrics utilized in this study

Code metrics	
# Of fields	# Of public fields
# Of methods	# Of public methods
# Of children	Depth of inheritance
LOC class	LOC method
LCOM	Fan-In
Fan-out	Total # Of operators
# Of distinct operators	Length
Vocabulary	Volume
Difficulty	Effort
NCSS for this file	Nested if else depth
Boolean expr. complexity	Cyclomatic complexity
NCSS Method	NCSS Class
N Path complexity	# of throws
# of executable statement	Method length
File length	# Of methods
# Of public methods	RFC
CBO	CDAC
Returns	# Of variables
# Of parameters	# Of Loops
# Of comparisons	# Of Try Catch
# Of parenthesized expressions	# Of string literals
# Of #s	# Of assignments
# Of math operations	Max # Of nested blocks
# Of anonymous classes	# Of inner classes
# Of lambdas	# Of unique words
# Of modifiers	# Of log statements

cross-project models' performance. With this in mind, we defined the following research questions.

RQ.1: *Do bad code smells outperform code metrics for cross-project defect prediction?*
RQ.2: *Does combining bad code smells and code metrics improve the performance of cross-project defect prediction?*

The remainder of this section is organized following the approach to generate the data sets, build the models and evaluate them. We start by measuring the code smells and the metrics, and we gather the defective information from the source code, thus constructing the data sets. Then, we process the data sets and use them for training the classifiers on each approach and building the cross-project models. Last, we evaluate the models, comparing the sets of each category of features. Figure 1 shows an overview of the methodology applied in this study.

4.1 Data sets construction

The goal of the first step of our approach is to obtain the data sets required for the classification. It is composed of a data collection step, followed by the extraction of the features and the defects. Lastly, it pre-processes the data sets for training, thus accounting for missing information, data inconsistencies, and data imbalance.

4.1.1 Data collection

We considered 100 Apache projects written in Java whose repository have *Git* as the version control system and *Jira* as the issue tracking system. Moreover, for each project, we manually selected five versions whose percentage of defects fall within 10–30% and contain the highest amount of components, i.e., files. Since it composes a good representation of defects, the ratio is high enough that reduces the class imbalance and is low enough not to select outlier versions (e.g., a version that was created only to fix issues).

4.1.2 Features extraction

In this step, we go over each file of the selected versions and extract the features into the respective sets: **code smells**, **code metrics**, and **code smells + metrics**. The first one contains the 37 bad code smells that we considered for this study. We collected the design smells proposed by Suryanarayana et al. (2015) using Designite (Sharma 2018). In addition, the Fowler and Beck (1999) and Brown (1998) smells were

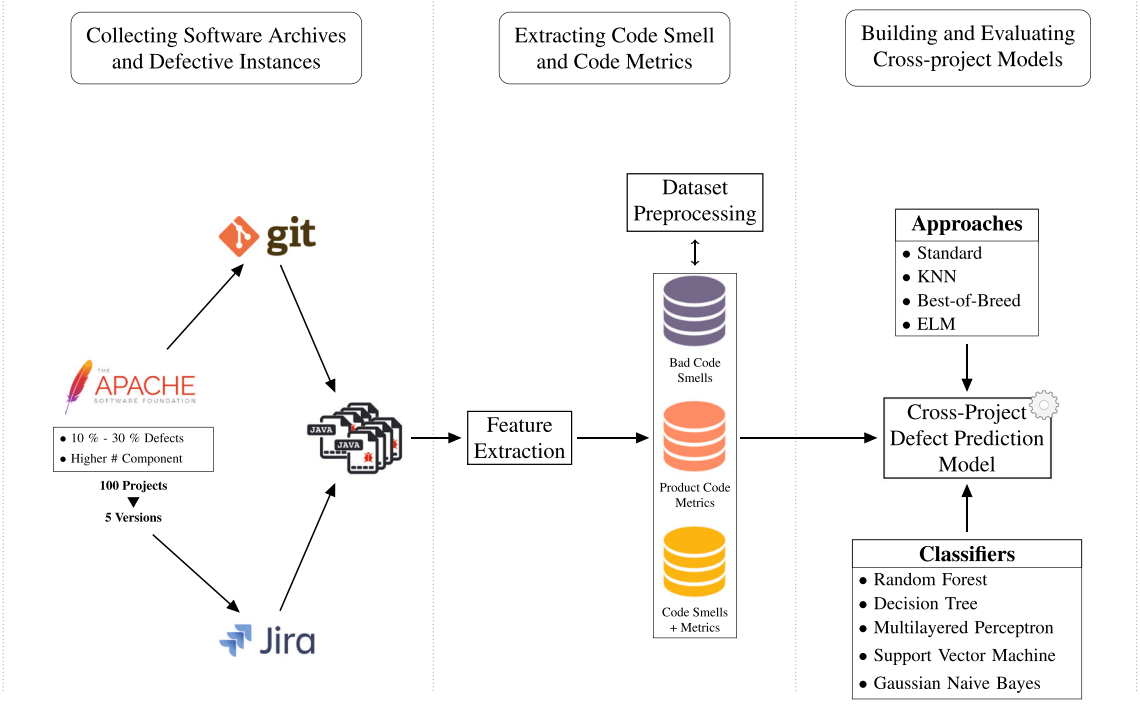

Fig. 1 Overview of the methodology applied in this study

extracted using a variant of the Organic tool (Cedrim and Sousa 2018), extended by the authors of this manuscript. The second set contains 56 code metrics, which we use to compare the smells. We extract metrics using the Designite metrics extractor functionality and Checkstyle (Ivanov et al. 2021). Moreover, we used a tool developed by the authors of this manuscript to collect the CK (Chidamber and Kemerer 1994), Mood (Brito e Abreu and Carapuça 1994), and Halstead (Halstead 1977) metrics from the source code. The last set contains the combination of all features, consisting of 93 features.

4.1.3 Defects extraction

The following step is to attach a target label to each instance of the created data sets. In particular, to set whether each extracted file is defective or not. Therefore, to extract the defects, we applied a variant of the SZZ algorithm (Borg et al. 2019) that accounts for the vulnerabilities in the algorithm (Herbold et al. 2018). We collected the issues assigned as closed bug reports for the selected versions from the Jira Issue Tracker of each project. Then, we applied a mapping between the collected issues and the commits in that specific version, thus connecting them by the issue id. Since Jira contains a unique ID of the format <PROJECT>−<NUMBER>, the matching of the id in the title and commit message becomes less ambiguous than the id format of the Bugzilla (only a number), which is a vulnerability reported by (Herbold et al. 2018). Then, we associated each issue with the changes that fixed it and, by pinpointing the files involved in the changing commit, we collected the defective files. Another reported issue was the absence of filtering for the files involved in the change and consequently were responsible for the defect. Therefore, we filtered the files in each commit to only account for Java files that were not tests.

4.1.4 Data sets pre-processing

The pre-processing step resolves the missing values in the data sets, handles the values inconsistencies and the data imbalance. Therefore, we start by removing the missing values from the data sets. Then, we standardize the range of values of the data sets by applying a min-max scaling to values between 0 and 1000. The goal was to have a consistent range, broad enough to facilitate the calculations for the k-neighbors approach. Last, since there is a higher ratio of non-defective files, which is a common occurrence in defect prediction, we applied the Synthetic Minority Oversampling Technique (SMOTE) to increase the ratio of defective instances in the training sets (Chawla et al. 2002).

4.2 Cross-project defect prediction model training

Following the generation of the data sets, we trained the defect prediction models based on the Cross-Project approaches we refer to in Sect. 3.2. For all the approaches,

except the Extreme Learning Approach, we build the models by training five classifiers commonly used in defect prediction. We used the sci-kit learn tool (Pedregosa et al. 2011) and set the parameters as the default ones for each classifier. The classifiers are the following.

- Random Forest
- Support Vector Machine
- Multilayered Perceptron
- Decision Tree
- Gaussian Naive Bayes

Moreover, for the ELM approach, we used the extreme learning machine classifier from the extended sci-kit learn library (McGinnis 2015).

The **Standard Approach** is the trivial approach for Cross-Project defect prediction. The approach is used in the single study that evaluates specific code smells in Cross-Project defect prediction (Taba et al. 2013). We trained the model for a specific target project using the data set of one project from the remaining 99 projects. Then, we tested the model using the data set of the target project. In total, we evaluated the performance for the 100 target projects. Therefore, using the Standard Approach, we evaluated 49,500 models (100 target projects $\times$ 99 train projects $\times$ 5 classifiers).

The **Best-of-Breed Approach** applies a majority voting to identify the best overall performing defect prediction model. Considering a particular cross-project, the algorithm trains 99 models using the remaining projects. For each model, it then tests with the data sets of the 98 other projects and evaluates the F2-score of each test. In the end, for each of the 99 models, it calculates the average of the 98 evaluations; then, it selects the model with the higher average F2-score.

The **KNN Approach** builds a training set from the instances closest to each instance of the data set of the cross-project. We iterate through all instances of the cross-project's data set and, to optimize the search, we first apply a Mini Batch K-Means clustering algorithm to reduce the search space of the complete set of instances from the 99 projects. Then, within the cluster closest to the target instance, we calculate the euclidean distance between each vector of those instances and the vector of the target instance. Thus, we select the k instances with the smallest distance. In our scenario, we selected the ten closest instances from each target instance; therefore, if the cross-project has 100 instances, the training set will have 1000 (100 $\times$ 10) instances.

The **ELM Approach** uses feed-forward neural networks to apply several machine learning tasks, including classification. We trained each cross-project model with the General ELM classifier from the sklearn-extensions library (McGinnis 2015). We used a feed-forward neural network with 100 hidden layers calculating the radial-based function with a width of 0.1.

4.3 Data analysis and metrics

To evaluate the cross-project models and compare the impact of the different features, we calculated metrics commonly used in defect prediction. The evaluation metrics we calculated to measure the performance of each classifier are ROC AUC, F1-Score, and PR AUC. These were recommended as evaluation metrics by Rathore and Kumar (2019), in particular the AUC, to which they recommend as the primary indicator for evaluation. We discuss their rationale in this section.

We calculated the Area Under the Curve (AUC) of the Receiver Operating Characteristic curve (ROC) and the Precision-Recall curve (PR). AUC summarizes the ability of the classifier to discriminate between defective and non-defective classes. As such, the closer the value is to 1, the higher the classifier's skill of discerning the classes affected or not by the defect. Nevertheless, a score closer to 0.5 describes a classifier with lower accuracy, thus having a classification ability closer to a random classifier.

The ROC curve is based on the relationship between the true positive rate (TPR) and the false positive rate (FPR). True positive rate, also known as sensitivity, measures the proportion of components predicted defective that were correctly identified. The false-positive rate measures the proportion of non-defective components that were incorrectly labeled as defective over the total number of actual non-defective components. Both equations are represented as follows:

$$TPR = \frac{TP}{TP + FN} \qquad FPR = \frac{FP}{FP + TN} \tag{1}$$

The PR curve is based on the relationship between the precision and the recall. Precision and recall are two widely used metrics in defect prediction. They measure the relationships between specific parameters in the confusion matrix:

$$precision = \frac{TP}{TP + FP} \qquad recall = \frac{TP}{TP + TN} \tag{2}$$

In both computations, 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. FN is the number of non-defective classes that the classifier incorrectly predicts as defective. FP is the number of classes where the classifier fails to predict defective classes by declaring defective classes as non-defective.

Moreover, we calculated the F1-Score. It is the harmonic mean of both precision and recall, defined as follows:

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{3}$$

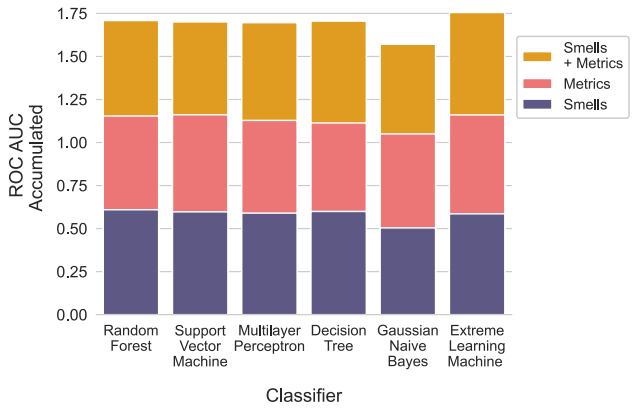

Fig. 2 Comparison of the ROC AUC score between classifiers, for the accumulated score of the three Data Sets

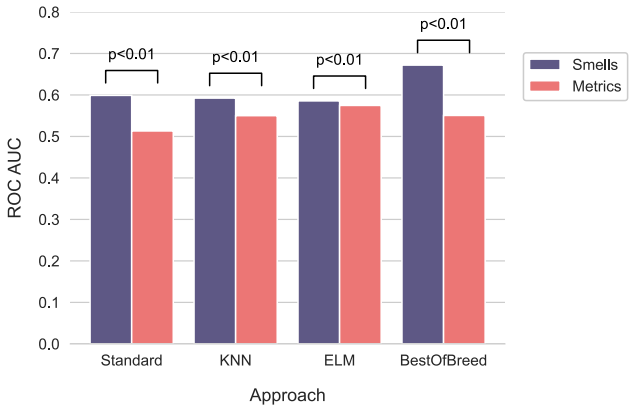

Fig. 3 Comparison of the ROC-AUC score between the code smells set and the code metrics set, for each cross-project defect prediction approach

5 Results

To study the impact of cross-project defect prediction using code smells, we present and discuss the results obtained according to each research question in this section.

In general, as observed in Fig. 2, the results were consistent among all classifiers. Therefore, to show the results and discuss the research questions, we selected and used the Decision Tree as the representative classifier.

The first goal of our research is to evaluate the impact of code smells against code metrics as features for cross-project defect prediction. Therefore, we measured the difference in performance between the models trained with code smells for different Cross-Project approaches. Figure 3 displays the difference in performance between the two categories of features for each of the cross-project defect prediction approach and the statistical significance for each approach. We observe a significant improvement of the models when using code smells compared to the code metrics. In particular, there was an average improvement of 6.50% for all the approaches, with a significance level of $p < 0.01$.

Despite the results obtained for the ROC AUC, the performance measured by the F1-Score and the PR AUC were generally not significant. However, the results obtained for the Standard Approach were shown to be significant ($\alpha < 0.01$). They display an increase in performance for the model trained with smells, compared with metrics (1.6% for the F1-Score and 0.9% for the PR AUC).

Since the results display an overall improvement of the cross-project models trained with code smells against traditional code metrics, it would be interesting to observe the percentage of projects where there was an improvement and visualize the individual improvement of each project for the ROC AUC. We observed that, within the 100 projects considered in this experiment, 73% showed an improvement. Therefore, in Fig. 4 we display the individual improvement of each project, represented by a point on both axes of the

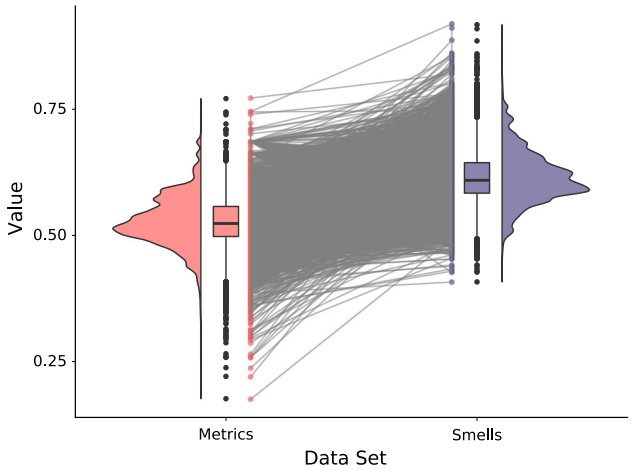

Fig. 4 Improvement for each project where there is an improvement from *code smells* to *code metrics*

Data Set, connected by a line. Moreover, we include half of a violin plot for each data set, as it shows the scores' distribution and the box plot, which describes the groups of the scores in the data considering their quartiles.

The second goal of our research is to evaluate whether both code smells and code metrics can be used in combination as features for cross-project defect prediction. Therefore, we compared the performance of models trained with features against the code smells set.

In Fig. 5, we display the ROC AUC of both sets for each approach, including the respective statistical significance. We observe that both features' categories, when used in combination, do not build models with higher performance than those trained with smells. Moreover, we compared the performance of the models trained with the combination of the features with the models trained with only metrics. We observed an improvement of 2% in the AUC ROC; however, the results were not statistically significant. Furthermore, the F1-Score

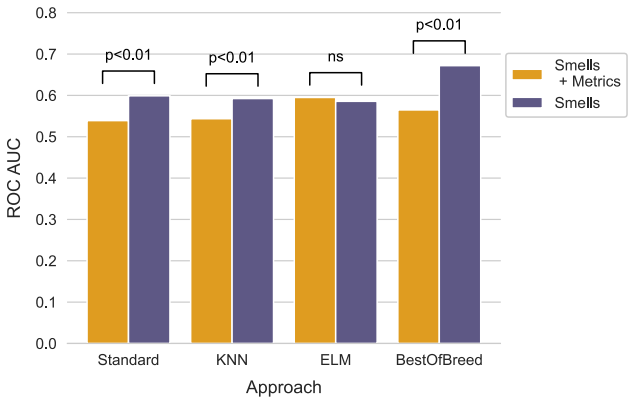

Fig. 5 Comparison of the ROC AUC score between the combination of code smells + code metrics and the code smells set, for each cross-project defect prediction approach

and the PR AUC scores were not statistically significant; therefore, we did not consider their results conclusively.

In the end, we observed the following results concerning the application of bad code smells to cross-project defect prediction.

- Bad code smells outperform the performance of the code metrics in all the evaluated approaches, with an average difference of the ROC AUC score of 6.50% ($\alpha < 0.01$).
- Considering each project improvement, 73% of the projects registered an improvement on the ROC AUC score when trained with code smells compared against the code metrics.
- When training the cross-project models with both code smells + metrics, there was not an improvement in performance compared against the code smells alone.

6 Threats to validity

For our study, we identified the following threats to validity:

In our study, we constrained the comparison of code smells to the code metrics. We do not compare the performance of defect prediction to other extensions of metrics such as process metrics. However, our study's scope and research goal only target the comparison of code smells with code metrics for cross-project defect prediction.

All of the used projects are Java open-source Apache projects. This is a threat to the generalization of our results. However, several defect prediction studies have used projects from Apache as the software archive Hosseini et al. (2019). Moreover, projects from Apache were also integrated into the Promise data set Jureczko and Madeyski (2010). In addition, the use of Jira as the issue tracking system is also a threat to validity toward the results' generalization. However, it is coupled to the Apache project management.

Although the use of a specific programming language could be considered a threat, the code smells and product metrics studied in this paper are general to object-oriented programming. Therefore, given an equivalent implementation of the features' extraction for other object-oriented programming languages, the conclusions from these study are valid.

The authors of this study developed the implementations of the cross-project defect prediction approaches by following the methodologies proposed by the authors of each approach. Although unlikely, the approaches' implementation may contain defects. However, this manuscript includes the entire source code and data sets used in our study; therefore, we provide them open for external validation. Moreover, this applies equally to the authors' tools to extract the code smells and the defects.

To extract the design code smells, we used the tool provided by the authors. This could be a threat to validity since we assume the reliability of the tool.

7 Conclusion

In this study, we evaluated the impact of bad code smells on cross-project defect prediction. Accordingly, we applied four approaches of cross-project defect prediction on 100 projects; thus trained with three data sets, one with code smells, one with code metrics and the last with both code smells + metrics. In the end, we found that the cross-project defect prediction models trained with bad code smells performed the best compared against code metrics, with an average improvement of 6.50% for the ROC AUC. Moreover, we observed an improvement on 76% of projects when comparing the differences individually for each project. About the combination of both code smells + metrics to train the cross-project models, we observed that the code smells alone still perform better than the combination of both categories of features.

From these results, future work concerns a deeper analysis of the impact of each code smell and an application of other cross-project approaches. Moreover, we want to expand from defects to vulnerabilities, thus analyzing the impact of code smells on cross-project vulnerability prediction.

Author Contributions Conceptualization: BSM, MK; Funding acquisition: MK; Investigation: BSM, MK; Methodology: BSM, MK; Supervision: MK; Visualization: BSM; Writing—original draft: BSM; Writing—review & editing: BSM, MK.

Funding This work was supported by the Cyber Security Research Center at the Ben-Gurion University of the Negev.

Availability of data and material The data sets generated during and analyzed during the current study are available in the public data repository https://zenodo.org/record/4697491.

Declarations

Conflict of interest The authors declare that they have no known competing interests or personal relationships that could have influenced the work reported in this paper.

Code availability The software developed during the current study is available from the public repository at the website of https://github.com/Bruno81930/smells.

References

Bal PR (2018) Cross project software defect prediction using extreme learning machine: an ensemble based study. In: Proceedings of the 13th international conference on software technologies, SCITEPRESS - Science and Technology Publications, Porto, Portugal, pp 354–361, https://doi.org/10.5220/0006886503540361, http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006886503540361

Booch G, Booch G (eds) (2007) Object-oriented analysis and design with applications, 3rd edn. The Addison-Wesley object technology series, Addison-Wesley, Upper Saddle River, NJ, p oCLC: ocm80020116

Borg M, Svensson O, Berg K, Hansson D (2019) SZZ unleashed: an open implementation of the SZZ algorithm - featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In: Proceedings of the 3rd ACM SIGSOFT international workshop on machine learning techniques for software quality evaluation - MaLTeSQuE 2019, ACM Press, Tallinn, Estonia, pp 7–12. https://doi.org/10.1145/3340482.3342742, http://dl.acm.org/citation.cfm?doid=3340482.3342742

Brito e Abreu F, Carapuça R, (1994) In: Zenodo McLean, VA, USA, DOI, (eds) Object-Oriented Software Engineering: Measuring And Controlling The Development Process. In: 4th International. publisher: Zenodo, p https://doi.org/10.5281/ZENODO.1217609,

Brown WJ (ed) (1998) AntiPatterns: refactoring software, architectures, and projects in crisis. Wiley, New York

Cedrim D, Sousa L (2018) opus-research/organic. https://github.com/opus-research/organic

Chawla NV, Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: synthetic minority over-sampling technique. J Artif Intel Res 16:321–357. https://doi.org/10.1613/jair.953

Chidamber S, Kemerer C (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493. https://doi.org/10.1109/32.295895

Cruz AEC, Ochimizu K (2009) Towards logistic regression models for predicting fault-prone code across software projects. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE, Lake Buena Vista, FL, USA, pp 460–463. https://doi.org/10.1109/ESEM.2009.5316002, http://ieeexplore.ieee.org/document/5316002/

Fowler M, Beck K (1999) Refactoring: improving the design of existing code. The Addison-Wesley object technology series, Addison-Wesley, Reading, MA

Goel L, Damodaran D, Khatri SK, Sharma M (2017) A literature review on cross project defect prediction. In: 2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON), IEEE, Mathura, pp 680–685, https://doi.org/10.1109/UPCON.2017.8251131, http://ieeexplore.ieee.org/document/8251131/

Guo J, Rahimi M, Cleland-Huang J, Rasin A, Hayes JH, Vierhauser M (2016) Cold-start software analytics. In: Proceedings of the 13th International Conference on Mining Software Repositories, ACM, Austin Texas, pp 142–153. https://doi.org/10.1145/2901739.2901740, https://dl.acm.org/doi/10.1145/2901739.2901740

Halstead MH (1977) Elements of software science. No. 2 in Operating and programming systems series, Elsevier, New York

Hassan AE (2009) Predicting faults using the complexity of code changes, In: 2009 IEEE 31st International Conference on Software Engineering, IEEE, Vancouver, BC, Canada, pp 78–88, https://doi.org/10.1109/ICSE.2009.5070510, http://ieeexplore.ieee.org/document/5070510/

Herbold S, Trautsch A, Grabowski J (2018) A comparative study to benchmark cross-project defect prediction approaches. IEEE Trans Softw Eng 44(9):811–833. https://doi.org/10.1109/TSE.2017.2724538

Hosseini S, Turhan B, Gunarathna D (2019) A systematic literature review and meta-analysis on cross project defect prediction. IEEE Trans Softw Eng 45(2):111–147. https://doi.org/10.1109/TSE.2017.2770124

Huang GB, Zhu QY, Siew CK (2006) Extreme learning machine: theory and applications. Neurocomputing 70(1–3):489–501. https://doi.org/10.1016/j.neucom.2005.12.126

Ivanov R, Veach R, Bludov P, Paikin A, Dubinin I, Selkin A, Lisetskii V, Burn O, Kordas M, Diachenko R, Izmailov B, Yaroslavtsev D, Sopov I, Kühne L, Giles R, Sukhodolsky O, Studman M, Schneeberger T (2021) checkstyle – Checkstyle 8.41.1. https://checkstyle.sourceforge.io/

Jureczko M, Madeyski L (2010) Towards identifying software project clusters with regard to defect prediction. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10, ACM Press, Timişoara, Romania

Kitchenham BA, Mendes E, Travassos GH (2007) Cross versus within-company cost estimation studies: a systematic review. IEEE Trans Softw Eng 33(5):316–329. https://doi.org/10.1109/TSE.2007.1001

Kotte A, Qyser D, Moiz AA (2021) A survey of different machine learning models for software defect testing. Eur J Mol Clin Med 7(9):3256–3268

Li Z, Jing XY, Zhu X (2018) Progress on approaches to software defect prediction. IET Softw 12(3):161–175. https://doi.org/10.1049/iet-sen.2017.0148

McCabe T (1976) A complexity measure. IEEE Trans Softw Eng SE 2(4):308–320. https://doi.org/10.1109/TSE.1976.233837

McGinnis W (2015) sklearn-extensions. https://github.com/wdm0006/sklearn-extensions

Moser R, Pedrycz W, Succi G (2008) Analysis of the reliability of a subset of change metrics for defect prediction. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08, ACM Press, Kaiserslautern, Germany, https://doi.org/10.1145/1414004.1414063, http://portal.acm.org/citation.cfm?doid=1414004.1414063

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on Software engineering - ICSE '05, ACM Press, St. Louis, MO, USA, p 284, https://doi.org/10.1145/1062455.1062514, http://portal.acm.org/citation.cfm?doid=1062455.1062514

Paterson D, Campos J, Abreu R, Kapfhammer GM, Fraser G, McMinn P (2019) 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), IEEE, Xi'an, China, pp 346–357, https://doi.org/10.1109/ICST.2019.00041, https://ieeexplore.ieee.org/document/8730206/

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E

(2011) Scikit-learn: machine learning in python. J Mach Learn Res 12:2825–2830

Piotrowski P, Madeyski L (2020) Software defect prediction using bad code smells: a systematic literature review. In: Poniszewska-Marańda A, Kryvinska N, Jarzbek S, Madeyski L (eds) Data-centric business and applications: towards software development (volume 4). Springer International Publishing, Cham, pp 77–99. https://doi.org/10.1007/978-3-030-34706-2

Porto F, Minku L, Mendes E, Simao A (2019) A systematic study of cross-project defect prediction with meta-learning. arXiv:1802.06025 [cs]

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Information and Software Technology 55(8):1397–1418. https://doi.org/10.1016/j.infsof.2013.02.009, https://linkinghub.elsevier.com/retrieve/pii/S0950584913000426

Rathore SS, Kumar S (2019) A study on software fault prediction techniques. Artif Intell Rev 51(2):255–327. https://doi.org/10.1007/s10462-017-9563-5

Sharma T (2018) DesigniteJava. https://doi.org/10.5281/zenodo.2566861

Suryanarayana G, Samarthyam G, Sharma T (2015) Refactoring for software design smells: managing technical debt. Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, Amsterdam, Boston

Taba SES, Khomh F, Zou Y, Hassan AE, Nagappan M (2013) Predicting Bugs Using Antipatterns, In: 2013 IEEE International Conference on Software Maintenance, IEEE, Eindhoven, Netherlands, pp 270–279, https://doi.org/10.1109/ICSM.2013.38, http://ieeexplore.ieee.org/document/6676898/

Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14(5):540–578. https://doi.org/10.1007/s10664-008-9103-7

Watanabe S, Kaiya H, Kaijiri K (2008) Adapting a fault prediction model to allow inter languagereuse. In: Proceedings of the 4th international workshop on Predictor models in software engineering - PROMISE '8, ACM Press, Leipzig, Germany, p 19, https://doi.org/10.1145/1370788.1370794, http://portal.acm.org/citation.cfm?doid=1370788.1370794

Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE '09, ACM Press, Amsterdam, The Netherlands, p 91, https://doi.org/10.1145/1595696.1595713, http://portal.acm.org/citation.cfm?doid=1595696.1595713

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