Empirical Study about Class Change Proneness Prediction using Software Metrics and Code Smells

Antonio Diogo Forte Martins, Cristiano Melo, José Maria Monteiro and Javam de Castro Machado
Federal University of Ceara, Fortaleza, Ceara, Brazil

Keywords: Machine Learning, Change-proneness Prediction, Software Quality.

Abstract: During the lifecycle of software, maintenance has been considered one of the most complex and costly phases in terms of resources and costs. In addition, software evolves in response to the needs and demands of the ever-changing world and thus becomes increasingly complex. In this scenario, an approach that has been widely used to rationalize resources and costs during the evolution of object-oriented software is to predict change-prone classes. A change-prone class may indicate a part of poor quality of software that needs to be refactored. Recently, some strategies for predicting change-prone classes, which are based on the use of software metrics and code smells, have been proposed. In this paper, we present an empirical study on the performance of 8 machine learning techniques used to predict classes prone to change. Three different training scenarios were investigated: object-oriented metrics, code smells, and object-oriented metrics and code smells combined. To perform the experiments, we built a data set containing eight object-oriented metrics and 32 types of code smells, which were extracted from the source code of a web application that was developed between 2013 and 2018 over eight releases. The machine learning algorithms that presented the best results were: RF, LGBM, and LR. The training scenario that presented the best results was the combination of code smells and object-oriented metrics.

1 INTRODUCTION

During the development and lifespan of software, maintenance is considered one of the most arduous and expensive tasks (Koru and Liu, 2007). Software systems evolve in response to the needs and requirements of a dynamic and ever-changing world. Hence, a change may occur due to bugs, new features, code refactoring, or adoption of new technologies. Throughout its evolution, software becomes larger and more complex (Koru and Liu, 2007). Thus, manage and control changes is one of the most concerns of the software development industry. During a software system evolution, it is impractical for the development team to focus equally on all parts of the system (Elish and Al-Rahman Al-Khiaty, 2013).

The quality of software may decrease over time due to different factors, such as aging, inconsistent design, and inadequate requirements design. In software engineering, the study about code smells is recent and is gaining importance to analyze different aspects of software quality, because they may point out problems related to structure, efficiency, maintainability, and readability of code (Fowler, 2018). Anyhow, even if a code smell does not represent a bug directly, because they are not technically incorrect and do not interfere in the code execution, they should not be ignored, because they may compromise the software quality and then lead to larger problems (Singh and Chopra, 2013).

In this context, a change prone class can be defined as a class that probably will suffer alterations for the next software release, representing a part of low quality of the system. Therewith, the capability of performing a prediction of a class change proneness may be handy to guide the software development team, because, with this information, they can better allocate resources, allowing project managers to focus their efforts and attention on these classes during the evolution process of a software system (Elish et al., 2015).

Although many works make use of machine learning techniques to predict class change proneness, no practical guide had been proposed to assist software engineers in using these techniques properly and be able to extract correct conclusions from the results. In (Melo. et al., 2019), the authors, recognizing the urgency of methodology standardization for the task

Martins, A., Melo, C., Monteiro, J. and Machado, J.
Empirical Study about Class Change Proneness Prediction using Software Metrics and Code Smells.
DOI: 10.5220/0009410601400147
In *Proceedings of the 22nd International Conference on Enterprise Information Systems (ICEIS 2020) - Volume 1*, pages 140-147
ISBN: 978-989-758-423-7

of class change proneness prediction, propose a comprehensive practical guide to help make predictions methodologically correct. The practical guide consists of a list of minimal activities, from proper data set construction to perform predictions, that must be executed to optimize the performance of the class change proneness classifier.

The objective of the following paper is, using the methodology based on the practical guide proposed in (Melo. et al., 2019), build an appropriate data set to perform class change proneness prediction with 8 object-oriented metrics and 32 types of code smells, empirically examine the performance of machine learning techniques in the task of class change proneness prediction, and evaluate the techniques in three scenarios of training: only object-oriented metrics, only code smells, and a combination of object-oriented metrics with code smells.

2 RELATED WORK

In (Kaur et al., 2016), the authors compare the performance of object-oriented software metrics and code smells when used as features to train machine learning algorithms to perform class change proneness prediction in an imbalanced data set. The authors used eight machine learning algorithms e to balance the data set they used SMOTE for over-sampling and RUS for under-sampling. The authors chose AUC and F-score metrics to evaluate the performance of the classifiers. According to their experiments, when trained with code smells, the classifiers achieve better performance than when trained with object-oriented software metrics. The authors conducted experiments with both imbalanced and balanced data, and classifiers trained with code smells performed better in both scenarios.

In (Catolino et al., 2019), the authors investigate the effect of adding a new metric, called the intensity index of a code smell, to three sets of metrics used to perform the task of class change proneness prediction found in the literature. The value assigned to this index is equivalent to how much the metric evaluated to detect a code smell exceeds the limit set by the rule, normalized between 0 and 10. The authors use a classifier that uses the LR algorithm and train it separately with each set of metrics. After taking these results as a baseline, they train once again by adding the intensity index as a feature and evaluate its effect. The chosen performance metrics were AUC, F-score, sensitivity, and specificity. After analysis, the authors state that performance in class change proneness prediction is statistically better after adding the intensity index to the features. In the end, they merge all metric sets and intensity index into one single data set and retrain the classifier obtaining the best prediction result.

In (Melo. et al., 2019), the authors propose a practical guideline to support an object-oriented change-prone class prediction. The practical guide consists of good practices that must be adopted to properly perform class change proneness prediction since the authors identified several possibly misleading results in the literature for not performing activities considered crucial by them. The authors present in detail all the activities that must be performed from the data set design to the class change proneness prediction and, in the end, carry out a case study replicating the activities step by step. The authors conclude that the practical guide can be used as a standard minimum activity list to develop class change proneness classifiers optimally. The case study performed to validate the method is based on an extremely imbalanced data set extracted from commercial software containing eight object-oriented metrics proposed by (Chidamber and Kemerer, 1994) and (McCabe, 1976).

3 METHODOLOGY

The methodology of this work was based on the practical guide proposed in (Melo. et al., 2019). The practical guide is designed to support class change proneness prediction. It contains a comprehensive step-by-step that covers all the steps necessary to build a useful data set and the correct way to make predictions.

3.1 Phase 1: Data Set Design

The first phase of the practical guide focuses on the data set design that will be used to perform class change proneness predictions. This phase consists of the following steps: Choose Independent Variable, Choose Dependent Variable, and Collect Metrics.

3.1.1 Choose Independent Variable

We chose the independent variables, also known as predictors or features, taking into consideration the metrics found in the literature. The papers that study class change proneness prediction use the metrics proposed by (Chidamber and Kemerer, 1994) and (McCabe, 1976), which are metrics capable of quantifying structural aspects of a class in object-oriented abstraction.

We decided to use the following object-oriented metrics: Class Between Object (CBO), Cyclomatic Complexity (CC), Depth of Inheritance Tree (DIT),

Lines Of Code (LOC), Number Of Children (NOC) and Weighted Methods per Class (WMC). In addition, we decided to use code smells as metrics, too, because they are also a feature of a class.

3.1.2 Choose Dependent Variable

Also known as the label, we chose to be the dependent variable the change proneness of a class. A class is said change prone if it has changed from one release to another, the parameter used to know if this change occurred is by checking the value of LOC in the two releases (Lu et al., 2012). If there has been a change, label 1 is assigned. If not, label 0 is assigned to the class under analysis.

3.1.3 Collect Metrics

We use in this paper a data set generated from the source code of a web application that was developed between 2013 and 2018. We analyzed the class change proneness of 8 releases of the application. This application is a collection of modules that manage the needs of a company concerning its internal processes, such as product return control and product quality management.

The object-oriented metrics of the application is available in a public GitHub repository (Melo. et al., 2019)[1]. The metrics were extracted from the source code using a Visual Studio plug-in called NDepends (NDpends, 2018). NDepends is a static analysis tool for C# code. In addition to other features, this tool has a specific one called CQLinq that allows the user to recover attributes and metrics by writing queries. The authors wrote and executed the CQLinq queries to precisely extract all the metrics available in their data set. The final data set is available as a csv file containing the metrics of each class in each release.

We performed the code smells extraction using the Designite tool (Sharma, 2016). This tool detects code smells in C# code and other relevant metrics, for instance, code smells density and number of classes in a project. We found 32 types of code smells after analyzing all the classes from all eight releases.

When the Designite tool analyzes a project, in our case, one release of the application, it generates object-oriented metrics and code smells reports. The code smells report is divided into three levels of granularity: Architecture (namespace), Design (class), Implementation (method). These reports are exported as a csv file being one file for each level. We developed Python scripts to organize and join all the files in a single csv per release.

[1]https://github.com/cristmelo/PracticalGuide

Although there is a difference in granularity between the code smells types, the tool always associates the code smell occurrence to a class, for instance, if an Architecture code smell is identified, the tool is capable of point which classes inside the namespace are responsible for this occurrence. Therefore the classes in question will account for one occurrence of this code smell. We used the same occurrence extraction logic for the Implementation code smells, since the tool tell us in which class the method that has a code smell belong.

After we finish the two data sets, we had to join the two to build the complete data set with the object-oriented metrics and code smells for each class. Creating isolated small data sets for each release facilitated the construction of the final set, as we ensured that if a class appears in more than one release, the object-oriented metrics and code smells were of the same release for a given class. In the end, we concatenated the eight isolated releases data sets forming a final set with all class of all releases with their respective features containing a total of 11576 classes.

3.2 Phase 2: Apply Class Change Proneness Prediction

The second phase of the practical guide focuses on developing class change proneness prediction models. We treated the class change proneness prediction problem as a classification problem and we used algorithms that work via supervised learning. This phase of the practical guide also indicates what the best performance metrics are, how to present results, and how to ensure the reproducibility of experiments.

3.2.1 Statistical Analysis

First, we performed a statistical analysis to extract relevant information about how object-oriented software metrics and code smells behave in the data set built at the end of Phase 1. Table 1 shows the descriptive statistical data for object-orientated software metrics.

The information in Table 1 is essential for understanding the domain of variables, whether they are discrete or continuous variables, and notions of their distributions. We did not include descriptive statistical data about code smells as they are mostly binary data. In the paper repository, it is possible to visualize the data in detail.

After analyzing the data distribution, we could highlight the large number of zero values that the metrics have. In the case of the code smells, the fact that there are many zeros is that most of these code smells are already avoided by the use of development tools

Table 1: Descriptive Statistics.

Metric	Min.	Max	Avg.	Median	Std. Dev.	Kurtosis	Skewness
CBO	0	162	5.73	3	9.57	32.4	4.43
CC	0	488	12.27	7	22.15	103.37	7.99
DIT	0	7	0.73	0	1.60	6.85	2.63
LCOM	0	1	0.14	0	0.27	0.82	1.55
LOC	0	1369	25.40	12	66.49	122.32	9.44
NOC	0	189	0.44	0	5.19	409.06	18.57
RFC	0	413	7.05	1	18.96	79.11	7.31
WMC	0	56	1.23	0	3.24	77.39	7.26

and good programming practices (de Almeida Filho et al., 2019).

3.2.2 Normalization and Outlier Detection

As stated in Section 3.2.1, the features of the constructed data set have very different domains. For instance, the metric LOC has values between 0 and 1369, while LCOM varies between 0 and 1. When faced with such situations where metrics are at different scales, it is interesting to normalize the data so that the models to be trained are not biased because of this difference.

In this paper, we normalized all the features, object-oriented software metrics and code smells, using the min-max normalization technique. When using this technique, all features will have their minimum value set to 0 and their maximum value set to 1.

About outliers detection and removal, in this paper, because the data distribution shows that there are many zeros and the averages have low values, we decided not to remove the outliers.

3.2.3 Feature Selection

First, we performed feature selection only on the object-oriented software metrics. The selected features are the same proposed by the practical guide (Melo. et al., 2019), because they are the same metrics from the same analyzed data set. The eight metrics were submitted to five feature selection techniques: Chi-Square, One-R, Information Gain, Symmetrical Uncertainty, and Correlation Analysis.

In (Melo. et al., 2019), the authors defined a criterion to choose the features from the results of the four first techniques. They ordered the results and the best-evaluated metric received 8 points and the worst 1 point. The metrics with the total number of points higher than half of the maximum were selected. Five metrics had the total number of points inside the limits, but CBO and RFC have Pearson's correlation coefficient of 0.87, in other words, they are strongly correlated, then the chosen metric was CBO because it had a higher total number of points than RFC. Lastly, the authors selected four metrics: CBO, WMC, CC, and LCOM.

In the case of feature selection for code smells, we performed an analysis using PCA (Abdi and Williams, 2010). We applied the technique on the group of code smells of the same level, i.e., the 32 features of code smells found were reduced to only three features. However, this approach worse results than using the 32 features. After analysis, we selected all 32 features as necessary, because each one represents particular and peculiar characteristics of the source code.

In the end, the features selected to be used in the prediction model were the 32 found code smells and the object-oriented software metrics CBO, WMC, CC, and LCOM.

3.2.4 Data Balancing Techniques

First, we need to analyze the change prone labels proportion to check if it will be necessary to use any data balancing technique. As we stated before if a class has label 1 it is because this class is change prone. If it is not, it will receive the label 0. Our data set is extremely imbalanced, 11263 classes have the label 0, while, only 313 have label 1, i.e., only 2.7% of the analyzed classes are change prone.

Using an imbalanced data set to train classification algorithms can lead to misclassification as the classifier may be biased and not correctly classify instances of the minority label (Melo. et al., 2019). In real machine learning problems, most data sets are imbalanced and the label of interest for classification is usually minority one.

The practical guide suggests using data under-sampling and over-sampling techniques to solve the imbalance problem. Under-sampling techniques: Random UnderSampler (RUS), Edited Nearest Neighbours (ENN) (Wilson, 1972), and Tomek's

Link (TL) (Tomek, 1976). Over-sampling techniques: Synthetic Minority Over Sampling Technique (SMOTE) (Chawla et al., 2002), Adaptive Synthetic Sampling (ADASYN) (He et al., 2008), and Random OverSampler (ROS). To assist in performing this step of the practical guide, we used the Python library Imbalanced-Learn (Lemaître et al., 2017). It is essential to emphasize that balancing techniques should be applied only to the training set.

3.2.5 Cross-Validation

There are many Cross-Validation techniques, but the most used, and the one we applied in this paper is the k-fold Cross-Validation. This technique consists of split the data set in k equal parts. With these subsets, the model is trained in k different rounds, in each round, one of the subsets is previously separated as the test set, and the rest is used for training. At the end of the k rounds, we calculate the average value of the results to evaluate the performance of the model. In our paper, we use ten as value for k, as the practical guideline suggests (Melo. et al., 2019). To ensure that the data is split into subsets keeping the labels proportion, we used the scikit-learn function StratifiedKFold.

3.2.6 Tuning the Prediction Model

In this paper, we used the following machine learning algorithms: Logistic Regression (LR), Support Vector Machine (SVM), Decision Tree (DT), Random Forest (RF), K-Nearest Neighbours (KNN), Light Gradient Boost Machine (LGBM), and eXtreme Gradient Boost Machine (XGB).

The scikit-learn Python library implements all the used algorithms, except LGBM and XGB that have their library. All these algorithms have hyperparameters, which, if properly adjusted and well selected, can improve the algorithm results (Melo. et al., 2019).

In this paper, we used the Grid Search as the method of search and evaluation of the hyperparameters. We chose the vectors arbitrarily, as previously mentioned, the model tuning process is a trial and error work.

Since we use Grid Search, it is necessary to use the nested Cross-Validation technique to estimate the model generalization with the chosen hyperparameters, as suggests the practical guide (Melo. et al., 2019).

3.2.7 Selection of Performance Metrics

For classification problems the most commonly used performance metrics are: accuracy, precision, sensibility, specificity, *F-score* and AUC. The confusion matrix also gives a notion of how the model performed the classification.

In the case of this paper, where the data set is imbalanced, the practical guide (Melo. et al., 2019) recommends using either F-score or AUC, as they take into account the correct classification of the minority class. We decided to use AUC as the main performance metric, but the other metrics were also collected.

3.2.8 Ensure the Reproducibility

The practical guide(Melo. et al., 2019) suggests that the authors of scientific papers ensure that their work can be reproduced and used as a basis for future work. To this end, the data set created at the end of Phase 1, statistical analysis for code smells and object-oriented software metrics, Jupyter Notebooks with the experiments already executed and the results of the evaluation of the trained models are organized and available in a repository on GitHub[2].

4 RESULTS AND DISCUSSION

At the experimentation step, we defined three training scenarios for the machine learning algorithms. The first trains the algorithms using only object-oriented software metrics. The second one train the algorithms using only code smells. Lastly, the third scenario uses both object-oriented software metrics and code smells as features. In all three scenarios, all the experiments were performed using the same steps described in Section 3.2.

Tables 2, 3, 4 show the 10 best results after the execution of the nested cross-validation process for each algorithm using the data set imbalanced, subsampled, and over-sampled having as features the ones defined for the scenario 1, 2, and 3 respectively. The values presented are the average value of AUC after the 10 rounds of nested cross-validation and also the standard deviation values for the execution.

Table 2: Results for nested Cross-Validation - Only object-oriented software metrics - 10 best classifiers.

Algorithm	Balancing Tech.	AUC (%)	Std. Dev. (%)
RF	RUS	73.3	2.8
LGBM	ROS	72.7	3.3
LGBM	RUS	72.5	4.2
LGBM	SMOTE	71.7	2.3
LR	SMOTE	71.5	3.3
LR	ROS	71.4	3.2
XGB	RUS	71.3	3.5
LR	ADASYN	71.2	3.2
LGBM	ADASYN	71.1	3.3
KNN	RUS	71	3.9

4.1 Only Object-oriented Software Metrics

We can observe that classifiers using the LGBM algorithm figure 4 times between the 10 best-evaluated classifiers using the three over-sampling techniques and the under-sampling technique RUS. The classifier with the better average value of AUC found in our experiments was RF trained with data under-sampled by RUS. Classifiers based on the LR algorithm trained with data over-sampled with the three techniques are also in the 10 best. It is interesting to highlight that the most straightforward algorithm, KNN, figure as the tenth best classifier when trained with data under-sampled by RUS.

Classifiers trained with over-sampled data figured more in the 10 best classifiers than the ones trained with under-sampled data in scenario 1. Furthermore, out of three under-sampling techniques, only RUS achieved good results.

4.2 Only Code Smells

From the results, we can observe that classifiers using LR with the three over-sampling techniques and the under-sampling technique RUS are present in the 10 best-evaluated classifiers, being the classifier that uses LR trained with data over-sampled by ROS the best in scenario 2.

The classifiers that use LGBM and over-sampled data are also present among the best 10 classifiers. The classifier using RF trained with data under-sampled by RUS is also figuring in the best classifiers for this scenario. The classifiers that use XGB trained both with data under-sampled by RUS and over-sampled by ROS are among the best for scenario 2.

[2]https://github.com/diogofm/TCC-ChangeProneness Prediction

4.3 Code Smells and Object-oriented Software Metrics

In scenario 3, the classifier trained with the RF algorithm and using the RUS under-sampling technique was again the best-evaluated classifier. The classifiers trained with the LGBM and LR algorithms using over-sampling techniques are among the 10 best-evaluated classifiers models, highlighting the over-sampling technique ROS that made the algorithms have better performance in our experiments. The algorithm DT figures for the first time among the best classifiers when trained with data over-sampled by ROS.

4.4 Discussion

As expected, the classifiers trained with imbalanced data did not perform well in our experiments, because the algorithms would be exposed to a few observations of the minority class and would not be able to classify well. Similarly, the TL and ENN under-sampling techniques did not achieve satisfactory results, because these techniques are distance-based, depending on the data set, they do not balance correctly, and this fact could be verified in our experiments because when these techniques were used the data set remained highly imbalanced.

Scenario 2 classifiers, trained only with code smells, achieved the worst performance among the three scenarios. No classifier reached an average value of AUC after the 10 rounds of nested cross-validation greater than 70%, which is the value, generally, considered to be an acceptable performance threshold for a classifier.

The classifiers trained in scenarios 1 and 3 showed the best performance in predicting class change proneness. Object-oriented software metrics when used to train the classifiers, scenario 1, allowed sim-

Table 3: Results for nested Cross-Validation - Only code smells metrics - 10 best classifiers.

Algorithm	Balancing Tech.	AUC (%)	Std. Dev. (%)
LR	ROS	68.8	3.6
LR	ADASYN	68.7	4.1
LGBM	ROS	68.7	4.4
LR	SMOTE	68.6	4.5
RF	RUS	68.6	4.3
XGB	ROS	67.9	3.8
LR	RUS	67.9	4.2
LGBM	SMOTE	67.3	4.5
XGB	RUS	67.2	4.4
LGBM	ADASYN	67.1	3

Table 4: Results for nested Cross-Validation - Object-oriented software metrics and code smells metrics - 10 best classifiers.

Algorithm	Balancing Tech.	AUC (%)	Std. Dev. (%)
RF	RUS	73.5	3.7
LGBM	ROS	73.4	3.1
LGBM	RUS	72.8	2.9
LR	ROS	72.5	4.2
LR	ADASYN	71.5	3.5
LR	SMOTE	71.1	4.1
LGBM	ADASYN	70.8	3.4
LGBM	SMOTE	70.6	3.7
DT	ROS	70.6	2.8
LR	RUS	70.3	4.1

ple classifiers, such as KNN and LR, to achieve good average value of AUC after the 10 rounds of nested cross-validation.

The addition of the code smells to the object-oriented software metrics improved the performance of some classifiers. For instance, the best classifier in scenario 1 remained the best in scenario 3, the RF algorithm trained with data under-sampled by RUS. It had a 0.2% improvement in the average value of AUC. It is also essential to highlight the performance improvement of the classifier that uses the DT algorithm trained with data over-sampled by ROS since when trained only with object-oriented metrics, the classifier did not even figure among the 10 best, the performance increase was of 9.5%.

The fact that the performance of the classifiers trained in scenario 2 was below acceptable threshold conflicts with the results found by (Kaur et al., 2016), where the authors state that code smells are better predictors than object-oriented software metrics. Our experiments showed that the code smells collected for this particular software and detected by the Designite tool were poor predictors, but when combined with the object-oriented software metrics, they were able to achieve a significant performance increase.

5 CONCLUSION

In this paper, we presented an empirical study about the performance of 8 machine learning techniques while performing the task of class change proneness prediction. We have proposed three training scenarios using a data set containing eight object-oriented software metrics and 32 types of code smells extracted from the source code of a web application that had its development between 2013 and 2018, having eight releases following the activities of the practical guide proposed by (Melo. et al., 2019).

From the results of the experiments, we concluded that using our data set, when training in the scenario 3 that uses both object-oriented software metrics and code smells combined, the class change proneness prediction classifiers achieve better performance than when trained with object-oriented metrics and code smells separate. The classifier that uses RF as the machine learning technique trained with data under-sampled by the RUS technique was the best in our experiments achieving the average value of AUC after the 10 rounds of nested cross-validation of 73.5%, in scenario 3. Also, in scenario 3, LGBM has presented good results achieving the average value of AUC af-

ter the 10 rounds of nested cross-validation of 73.4% when trained with data over-sampled by ROS technique.

In summary, for the case of the data set constructed during the paper and used for the experiments, code smells alone are not good class change-proneness indicators as object-oriented software metrics. However, when combined, they can lead to good results, increasing the performance of machine learning algorithms. Furthermore, the best performing machine learning techniques, based on the average value of AUC after the 10 rounds of nested cross-validation and algorithm simplicity, were RF, LGBM, and LR.

As future work, there are several possibilities and different approaches to be investigated. From the point of view of the data set design, one approach would be adding new software metrics such as evolutionary (Elish and Al-Rahman Al-Khiaty, 2013) and intensity index of a code smell (Catolino et al., 2019). Moreover, for the class change proneness prediction, it is viable to test another machine learning and deep learning techniques and use more modern and elaborate methods to create synthetic training data.

ACKNOWLEDGEMENTS

This research was funded by LSBD/UFC.

REFERENCES

Abdi, H. and Williams, L. J. (2010). Principal component analysis. *WIREs Comput. Stat.*, 2(4):433–459.

Catolino, G., Palomba, F., Fontana, F. A., Lucia, A. D., Zaidman, A., and Ferrucci, F. (2019). Improving change prediction models with code smell-related information. *CoRR*, abs/1905.10889.

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

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

de Almeida Filho, F. G., Martins, A. D. F., Vinuto, T. d. S., Monteiro, J. M., de Sousa, I. P., de Castro Machado, J., and Rocha, L. S. (2019). Prevalence of bad smells in pl/sql projects. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 116–121, Piscataway, NJ, USA. IEEE Press.

Elish, M., Aljamaan, H., and Ahmad, I. (2015). Three empirical studies on predicting software maintainability using ensemble methods. *Soft Computing*, 19.

Elish, M. O. and Al-Rahman Al-Khiaty, M. (2013). A suite of metrics for quantifying historical changes to predict future change-prone classes in object-oriented software. *Journal of Software: Evolution and Process*, 25(5):407–437.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

He, H., Bai, Y., Garcia, E. A., and Li, S. (2008). Adasyn: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1322–1328.

Kaur, A., Kaur, K., and Jain, S. (2016). Predicting software change-proneness with code smells and class imbalance learning. In *2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 746–754.

Koru, A. G. and Liu, H. (2007). Identifying and characterizing change-prone classes in two large-scale open-source products. *Journal of Systems and Software*, 80(1):63 – 73.

Lemaître, G., Nogueira, F., and Aridas, C. K. (2017). Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning. *Journal of Machine Learning Research*, 18(17):1–5.

Lu, H., Zhou, Y., Xu, B., Leung, H., and Chen, L. (2012). The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineering*, 17(3).

McCabe, T. J. (1976). A complexity measure. *IEEE Transaction on Software Engineering*.

Melo., C. S., da Cruz., M. M. L., Martins., A. D. F., Matos., T., da Silva Monteiro Filho., J. M., and de Castro Machado., J. (2019). A practical guide to support change-proneness prediction. In *Proceedings of the 21st International Conference on Enterprise Information Systems - Volume 2: ICEIS,*, pages 269–276. INSTICC, SciTePress.

NDepends (2018). [Online; posted 09-November-2018].

Sharma, T. (2016). Designite - A Software Design Quality Assessment Tool.

Singh, G. and Chopra, V. (2013). A study of bad smells in code. *Int J Sci Emerg Technol Latest Trends*, 7(91):16–20.

Tomek, I. (1976). Two modifications of cnn. *IEEE Trans. Systems, Man and Cybernetics*, 6:769–772.

Wilson, D. L. (1972). Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-2(3):408–421.