Extremely Imbalanced Smell-based Defect Prediction

Bruno Miguel Sotto-Mayor de Castro Machado

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Prof. João Fernando Peixoto Ferreira

September 2019
Acknowledgments

I would like to express my sincere gratitude to Professor Rui Abreu, my master thesis supervisor, for his patient guidance, useful critiques and for always being helpful and available. I could not have imagined having a better advisor and mentor.

I am particularly grateful for the assistance given by Daniel Correia for helping me review and improve the thesis.

I would also like to extend my gratitude to my fellow colleagues from our research group for their valuable support and insightful exchange of ideas.

Finally, I wish to thank my family for their support and encouragement.
Abstract

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

Keywords: Software, Defect Prediction, Code Smells, Data Imbalance
Resumo

Um dos maiores requisitos para uma entrega rápida de software em continuous integration/continuous delivery é a capacidade para encontrar bugs de forma eficiente. Por essa razão, muitas soluções foram introduzidas na literatura. Por exemplo, métodos de defect prediction baseados em bad code smells extraídos de cada versão de software. No entanto, estes métodos não consideram o problema em que pode existir uma percentagem extremamente superior de módulos não defeituosos, comparado a módulos defeituosos. Desta forma, nesta tese, introduzimos um novo modelo com o algoritmo de autoencoder que usa design smells e implementation smells para detetar módulos defeituosos portanto, treinamos cinco autoencoders com arquiteturas diferentes. Adicionalmente, para avaliarmos a nossa abordagem, nós comparamos cada modelo de autoencoder com a mesma arquitetura, treinados com traditional object-oriented metrics e a junção de ambas as features. A nossa análise não demonstrou resultados promissores, na medida em que, a utilização de smells e a junção das features não forneceu uma melhoria comparativamente à utilização de metrics. Todavia, nós introduzimos um ponto de partida para smell-based defect prediction no contexto de dataset imbalance e introduzimos uma baseline para trabalho futuro.

**Palavras-chave:** Software, Defect Prediction, Code Smells, Data Imbalance
Contents

1 Introduction ...................................................... 1
  1.1 Motivation and Problem Statement .......................... 2
  1.2 Research Questions ........................................ 3
  1.3 Contributions ................................................ 4
  1.4 Thesis Outline .............................................. 5

2 Background and Literature Review ................................. 6
  2.1 Background ................................................... 7
    2.1.1 Defect Prediction ..................................... 7
    2.1.2 Bad Code Smells ..................................... 8
    2.1.3 Imbalanced Data .................................... 9
    2.1.4 Autoencoders ........................................ 10
  2.2 Literature Review ........................................... 13
    2.2.1 Defect Prediction ..................................... 14
      2.2.1.A Defect Prediction Metrics ...................... 14
      2.2.1.B Defect Prediction Software Archives .......... 16
      2.2.1.C Defect prediction algorithms ................ 16
      2.2.1.D Evaluation measures for defect prediction ... 17
    2.2.2 Data Imbalance ....................................... 18
    2.2.3 Smell-based Defect Prediction ....................... 19
      2.2.3.A Toward a Smell-aware Bug Prediction Model (2019) 19
      2.2.3.C Defect Prediction with Bad Smells in Code (2016) 21
      2.2.3.D Software Analytics in Practice: A Defect Prediction Model Using Code Smells (2016) 22

3 Smell-based Defect Prediction Approach ............................ 23
  3.1 Smell-based Defect Prediction Approach .................. 25
    3.1.1 Feature Extraction .................................... 26
3.1.2 Dataset Processing .................................................. 27
3.1.3 Model Training ..................................................... 29
3.1.4 Evaluation approach .............................................. 32
   3.1.4.A Confusion matrix ........................................... 32
   3.1.4.B Receiver Operating Characteristic Curve .................. 33
   3.1.4.C Precision-Recall-Gain Curve ............................... 36
3.2 Smell-based Defect Prediction Methodology ...................... 39
   3.2.1 Methodology for feature extraction ......................... 39
      3.2.1.A Interacting with repositories from Defects4J ........ 40
      3.2.1.B Interacting with repositories from BugsDotJar ....... 41
      3.2.1.C Extracting features with Designite .................... 42
      3.2.1.D Capturing defect information from each module ...... 43
   3.2.2 Methodology for data processing ............................ 44
   3.2.3 Methodology for model training ............................. 46
   3.2.4 Methodology for model evaluation .......................... 47

4 Results ........................................................................... 49
   4.1 Methodologies ....................................................... 51
      4.1.1 Receiver Operating Characteristic Methodology ........ 51
      4.1.2 Precision-Recall-Gain Methodology ....................... 55
   4.2 Overall analysis ..................................................... 60

5 Discussion ..................................................................... 62
   5.1 Results interpretation ............................................... 63
   5.2 Answer to research questions ..................................... 64
   5.3 Results discussion ................................................... 65
   5.4 Contributions ........................................................ 67
   5.5 Future Work ........................................................... 68

6 Conclusion .................................................................... 69
   A Defect Prediction Tables ........................................... 81
   B Literature’s Appendix ................................................. 83
   C Extracted Metrics’ Preview ......................................... 87
   D Listings of Code Snippets ............................................ 90
# List of Figures

1.1 CI/CD pipeline [7] ......................................................... 2

2.1 Software defect prediction process [1] ................................ 8

2.2 Structure of a biological neuron. [16] ................................. 10

2.3 Structure of an artificial neuron. [16] ................................. 11

2.4 Architecture of a basic auto-encoder. [16] .......................... 12

2.5 Graphics representing each of the activation functions. ........... 13

3.1 Approach steps overview. ............................................. 25

3.2 Feature extraction phase. ............................................. 26

3.3 Data processing phase. .............................................. 28

3.4 Model Training phase. ............................................... 30

3.5 Overview of our models decision, based on threshold and error values. ................................. 32

3.6 Model Evaluation phase. ............................................ 33

3.7 Confusion Matrix ..................................................... 33

3.8 A basic ROC graph showing five discrete classifiers. [77] ........... 34

3.9 The ROC "curve" created by thresholding a test set. The graph shows the corresponding ROC curve with each point labeled by the threshold that produces it, each represented with the respective classification score. [77] ................................. 36

3.10 (left) Conventional PR curve with hyperbolic F1 isometrics (dotted lines) and the baseline performance by the always-positive classifier (solid hyperbole). (right) Precision-Recall-Gain curve with minor diagonal as baseline, parallel F1 isometrics and a convex Pareto front. [78] ......................................................... 39

3.11 Architecture of Defects4J. [44] ....................................... 41

3.12 BugsDotJar representation on GitHub. [45] .......................... 42
4.1 Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the first autoencoder architecture .......................................................... 52

4.2 Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the second autoencoder architecture .......................................................... 53

4.3 Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the third autoencoder architecture .......................................................... 54

4.4 Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fourth autoencoder architecture .......................................................... 54

4.5 Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fifth autoencoder architecture .......................................................... 55

4.6 Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the first autoencoder architecture .......................................................... 56

4.7 Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the second autoencoder architecture .......................................................... 57

4.8 Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the third autoencoder architecture .......................................................... 58

4.9 Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fourth autoencoder architecture .......................................................... 59

4.10 Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fifth autoencoder architecture .......................................................... 59

4.11 Comparing the ROC areas under the curve from all the models trained with five different architectures using the three datasets — smells-only, metrics-only and metrics-and-smells .......................................................... 61

4.12 Comparing the PRG areas under the curve from all the models trained with five different architectures using the three datasets — smells-only, metrics-only and metrics-and-smells .......................................................... 61

B.1 Commonly used performance evaluation measures [1] .......................................................... 86
List of Tables

2.1 Confusion matrix. [1] .......................... 18

3.1 Set of traditional object-oriented metrics categories we extracted from modules. .... 27

A.1 Name and description of the design smells used in this study organized by principle. . . 82
A.2 Name and description of the implementation smells used in this study. .................. 82

B.1 Summary of the publicly available datasets for software defect prediction. [1] ......... 84
B.2 Comparison of the defect prediction methods using machine learning techniques. [1] .. 85

C.1 Preview of traditional object oriented metrics extracted from project camel version 100
using Designite .................................................. 88
C.2 Preview of design smells detected from project camel version 100 using Designite ...... 89
C.3 Preview of implementation smells detected from project camel version 100 using Designite 89

List of Algorithms

3.1 Defects4J project versions iteration .................................................. 41
3.2 BugsDotJar project version iteration .................................................. 42
3.3 Extracting features with Designite .................................................. 43
3.4 Capturing defect information with defects4j ............................................ 44
3.5 Capturing defect information with bugsdotjar ........................................... 44
3.6 Generating metrics dataset ............................................ 45
3.7 Generating Smells dataset ............................................. 46
Listings

D.1 Code sample for standard scaling applied to three metrics ........................................ 90
D.2 Code sample for stratification .................................................................................. 90
D.3 Removal of defective instances .............................................................................. 91
D.4 Architecture implementation of a two hidden layer architecture with ReLU activation function ............................................................... 91
D.5 Architecture implementation of a four hidden layer architecture ....................... 91
D.6 Architecture implementation of a six hidden layer architecture ......................... 92
D.7 Making prediction with testing dataset features, calculating reconstruction error and return 
   error and testing output ......................................................................................... 92
D.8 Generating ROC Curve and calculating the area under the curve ......................... 92
D.9 Generating PRG Curve and calculating the area under the curve ......................... 93
1

Introduction

"If it smells bad, change it."

Kent Beck,

Contents

1.1 Motivation and Problem Statement ........................................ 2
1.2 Research Questions .......................................................... 3
1.3 Contributions ........................................................................ 4
1.4 Thesis Outline ...................................................................... 5
This chapter provides an introduction to our study. Thus, we provide the motivation behind this research, we define the problem and the research questions, we provide the contributions introduced by this research, and we provide an overview for the subsequent chapters of this thesis.

Therefore, the remainder of this chapter is organized as follows. Section 1.1 establishes the motivation for this study. Thus defining the problem we focused on, and describing the approach that contributes to the solution. Section 1.2 describes the research questions we considered in this study. In addition, it introduces the evaluation approach to answer the research questions. Section 1.3 provides the contributions introduced by this study. Section 1.4 concludes the chapter with an outline of the thesis.

1.1 Motivation and Problem Statement

One of the main requirements for a high-speed delivery of software is the ability to find bugs in an effortless and resource-less way. As a matter of fact, one of the main adversities in continuous integration/continuous delivery happens at the testing phase. Since, this phase may become really time consuming and be a bottleneck for the pipeline. For this reason, the Defect Prediction field has gain relevance in recent years. And, has became one of the most active research areas in software engineering. In particular, a promising family of defect prediction approaches based on the history of software has received much more attention in the past few years [1]. Additionally, in the software engineering field, there is a topic concerning bad code smells. It focuses on the relationship between the detection of code smells and the presence of defects in modules [2]. Accordingly, a recent field has emerged, that combines both of these areas. It explores the use of code smells as features for the prediction of defects in software [3–6].

Figure 1.1: CI/CD pipeline [7]
In the field of smell-based defect prediction, several studies suggested multiple approaches to predict defects using smells. Moreover, the majority introduced classification models whose use of smells improved the model’s performance. However, these studies do not consider a scenario with an extremely imbalanced dataset. For example, if the dataset had one million instances for the non-defective class and only one hundred for the defective class. Thus, creating an extreme uneven ratio. In such scenarios, if it used the same approaches as in the ones used with even rations, or slight imbalanced datasets, then, the model would not achieve its optimal performance [8]. Furthermore, if the standard evaluation approaches that are used for performance measurement (such as the ROC curve) are also used in these scenarios, then it would produce misleading scores for the model performance. [9]

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

1.2 Research Questions

In this study, we are proposing a new smell-based defect prediction model. This model was trained with an autoencoder algorithm because of the imbalance dataset problem. Additionally, we will compare our model with an equivalent defect prediction model trained with traditional object oriented metrics, alone and combined with smells. With this in mind, next we describe the research questions.

Research Question 1: Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with design and implementation smells or with traditional object-oriented features?

Research Question 2: Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with the combination of both design and implementation smells, and with traditional object-oriented features or when it is trained with only one single of those features?
Research Question 3: When training a smell-based defect prediction model with an autoencoder algorithm, what category of architecture provides the best model performance?

To evaluate our study, we created five different architectures of autoencoders. The variable properties between architectures were the number of layers and the activation functions used in each layer. Furthermore, we trained each of those autoencoders with three different datasets. A dataset with only design and implementation smells, a dataset with only traditional object-oriented metrics and a dataset with the combination of both features. Then, we extracted measures regarding the performance from each model. And we compared those measures to draw conclusions regarding the research questions. Whether the autoencoder architecture functions best for a smell-based defect model or a model trained with traditional object-oriented metrics. Then, whether the autoencoder trained with the combination of both metrics, performs better than either one of the previous models. Lastly, what type of autoencoder architecture performs best among the five smell-based defect prediction models. Concerning, the number of layers, from shallow to deep, and the activation functions applied in each layer.

In the end, we compared the performance of an autoencoder architecture with smells and object-oriented metrics trained with an imbalanced dataset. For the purpose of proving whether using smells as features provides enough information for an autoencoder to make accurate predictions, compared against traditional object-oriented metrics when trained with imbalanced data. Above all, we did not compare our results against models produced in the literature. Since, the datasets we used have an extreme data imbalance and, those models were trained with algorithms that do not work well with this scenario.

1.3 Contributions

Considering the goal of this study, this thesis did not show promising results. Because, in general, the models trained with traditional object-oriented metrics outperformed the models trained with design and implementation smells. In particular, the smells-only models showed the same level of performance as a random classifier.

Nevertheless, from this research we were able to draw conclusions about the best autoencoder architecture to fit the smell-based defect prediction model trained with an extremely imbalanced dataset. Thus, the most promising architecture was a deep layer architecture. In detail, the four layer hidden architecture with ReLU activation functions in all hidden layers. However, the interval of performances between models was very slim. Therefore, the architecture of the autoencoder does not appear to have great relevance to the performance of the smell-based model.
Furthermore, we can create a baseline for smell-based defect prediction models. In particular, we can provide the performance results from our autoencoder-model trained with smells. In addition, future studies can use this results as a baseline for comparisons against new smell-based defect prediction models, built with extremely imbalanced datasets.

Additionally, this study defines a starting point in smell-based defect prediction in the context of extremely imbalanced data and, although we did not achieve promising results, there are several routes to explore to build a more accurate model. Therefore, we suggest other possible approaches for future work. Such as, doing a finer-grained approach. Thus considering only relevant smells or calculating the value intensity on the detected smells [3]. Moreover, we propose building a model focused on the programmer. Thus, having a model trained from different projects, but with the same style of coding.

1.4 Thesis Outline

Chapter 2 provides the background information required for this study. Furthermore, it also reviews the literature in the bad code smells, defect prediction and smell-based defect prediction fields. The purpose of this chapter is to construct a workable base of knowledge and survey the latest contributions in each field. In particular, we want to analyze the latest solutions in smell-based defect prediction.

Chapter 3 outlines the research approach and methodology. Thus, we describe the implementation procedures to generate the datasets, train the autoencoders, and evaluate the defect prediction models.

Chapter 4 presents the results from the performance evaluations. Specifically, it compares the results between the three datasets. The smell-only, the metrics-only and the metrics-and-smells datasets. Additionally, it presents the comparison between the five autoencoder architectures.

Chapter 5 presents the results for the three research questions. Thus, it interprets and discusses the obtained results. Furthermore, it introduces the contributions given by this study. And, discusses the future work for smell-based defect prediction focused on imbalanced datasets.
Background and Literature Review

“If I have seen further than others, it is by standing upon the shoulders of giants.”

Isaac Newton,

Contents

2.1 Background ............................................. 7
2.2 Literature Review ...................................... 13
In this chapter, we build a workable framework of knowledge, thus we introduce concepts of defect prediction, bad code smells, imbalanced datasets, and autoencoders. Furthermore, we survey the latest contributions in the literature, by analyzing the latest solutions in defect prediction. Specifically, the solutions regarding each step of its process. Moreover, we describe the latest contributions on data imbalance and, we review the literature in smell-based defect prediction.

Therefore, the remainder of this chapter is organized into two parts. Section 2.1 describes the required background information for this study. Thus we provide the required understanding for this study. Additionally, Section 2.2 reviews the literature related to our study. Hence, it analyses the latest approaches in defect prediction, data-imbalance and smell-based defect prediction.

## 2.1 Background

Our goal is to explore a new model approach for smell-based defect prediction. Moreover, this model was built to fit the imbalance data scenario. Thus our decision to rely on autoencoders, an algorithm appropriate for outlier detection. Given these points, in the following sections we will cover the required background to understand the scope of this study.

First, we will describe the defect prediction process. In essence, required to build the defect prediction model in Section 2.1.1. Second, in Section 2.1.2, we will introduce code smells and we will describe the smells that we will be considering. Third we will introduce the imbalanced data problem. Coupled with, the outlier detection task in Section 2.1.3. Lastly, we will describe the autoencoder implementation in Section 2.1.4.

### 2.1.1 Defect Prediction

Defect Prediction is a sub-field of the mining software repositories (MSR) major area. This field provides a conventional methodology to do research in that field. Henceforth, Hemmati, H. et al. [10] describes this process as being composed by four phases. First, it is the acquisition and preparation of data. It consists of the extraction of raw data from different types of development artifact repositories. For example, source code, SCM, bug tracking issues, and mailing lists. In addition, it includes the modeling of information and the creation of connections between repositories. Second, it is an appropriate mining/learning technique to the prepared data. Such as clustering, classification, prediction, regression, or other machine learning algorithm. Third, it is applied the analysis of data, thus producing the research conclusions. Last, the used data and tools used in the previous study are shared. For the purpose of external validation and replication studies.
We have defined the general methodology for mining software repositories. Now, we will focus on defining the process of building a defect prediction model. Firstly, we extract data from software archives, including version control systems (eg. SVN, CVS, Git), which provide source code and commit messages, issue tracking systems (eg. Bugzilla, jira) that contains defect information, and so on. The extracted data instances are considered as modules. Each module may be a method, a class, a source code file, a package or a code change, depending on the chosen granularity. Secondly, we create data instances with the metrics (also features) extracted from each module. These metrics are defect prediction features and can be composed by the complexity of software (eg. lines of code), and by the development process (eg. code changes). Then, to each set of features, representing a module, we attach a label. This label classifies the model as defective or non-defective. Thirdly, we split the data instances into train and testing datasets. Then, we train a machine learning algorithm using the training dataset. Lastly, we use the testing instances to evaluate the performance of the trained model [1]. This process is illustrated in figure 2.1.

![Software defect prediction process](image)

Figure 2.1: Software defect prediction process [1]

2.1.2 Bad Code Smells

A bad code smell is a pattern of malpracticed software development. To put it differently, smells are symptoms of poor decisions in the creation of software. These decisions comprise the whole life-cycle of the system development. Thus, for each phase, there are an unique set of poor choices that can be applied. For example, the design and testing stage, each, have a set of unique bad decisions that can be committed. For this reason, from each step of the life-cycle, there are a set of smells that can be detected.
The accumulation of smells over a significant number of revisions, contributes to the accumulation of technical debt [11]. Technical debt is the aggregated cost of additional project maintenance and rework. As a result, the higher the debt and accumulation of bad decisions, the higher the price to be paid. Over an extreme scenario, the technical debt will reach a level where it will be impossible to pay the debt. Consequently, leading to a scenario called technical bankruptcy and causing the project to be discontinued.

For the remainder of this study we will only consider design and implementation smells. Design smells are a set of bad practices concerning the design stage of the software development and they are characterized by the principles they violate. These principles are comprised by abstraction, encapsulation, modularization and hierarchy principles. Their definition and corresponding set of smells are described in table A.1. Implementation smells are the set of malpractices concerning the implementation phase. We define each implementation smell in table A.2 [12].

### 2.1.3 Imbalanced Data

The problem of imbalanced data consists of an unequal distribution of classes in classification problems [13]. For illustration, consider a scenario of a 2-class (binary) classification problem with 1000 instances. In particular, assume that 990 instances are labelled with Class-1 (majority class) and the remaining 10 with Class-2 (minority class). The goal is to predict the Class-2 events. However, most common classification algorithms work poorly for such scenarios because the classifiers tend to be biased towards the majority class. Hence, not performing well for the minority class. With this in mind, the classifier will measure a very high accuracy of 99%, but it will be a misleading measurement since the accuracy is only reflecting the underlying class distribution.

Imbalanced data is a recurrent problem in real life applications. For instance, the application of fault diagnosis, anomaly detection, e-mail foldering and face recognition, among others [14]. Furthermore, defect prediction also has an imbalance problem since only a small percentage of modules, in each version of the software repository, is defective. Thus, leading to an uneven ratio of class percentages.

We want to build a prediction model, considering the data imbalance. Thus we can take advantage of the imbalance of data. In particular, we can teach the model to detect the minority class as an outlier. In general, this approach is called outlier detection and it is a commonly used solution to train a classifier with data imbalance. On the whole, outlier detection is a data cleaning task used to explore the set of normal objects in data. Nonetheless, it has been gaining more prominence as a potential data discovery task for anomaly detection, or in our context, defect detection [15].
2.1.4 Autoencoders

For this study we have to consider a prediction algorithm that works well with outlier detection. Therefore, we considered the autoencoder [16] which is an unsupervised artificial neural network whose purpose is to generate a solid internal representation of the input data.

An artificial neural network is a set of interconnected networks of processing units. These processing units (also artificial neurons) mimic the biological neurons found in the brain. Each biological neuron receives input signals from its neighboring neurons into its dendrites. Then, the signals are passed on to the cell body, where they are summed into a single signal. Again, the resulting signal is passed on to the terminal of the neuron. If the signal’s charge is higher than a specified threshold, then the signal is released to the neighboring dendrites.

![Figure 2.2: Structure of a biological neuron. [16]](image)

The artificial neurons receive an input signal from the neighboring neurons, where each signal is a data instance concerning a single feature. Then, all the signals reach the neuron and are summed up. During the summation, each signal is assigned a weight and added a bias. The weight represents the strength of the connection between neurons and the bias is an offset that forces the activation of the neuron if all inputs are 0. Additionally, to introduce non-linearity, we can apply a non-linear activation function which scales the signal domain to a different interval. In the end, we pass on the resulting signal to the neighboring neurons.
In its simplest form, an autoencoder is composed by three layers. The input layer, the hidden layer and the output layer. In another perspective, it is divided in two stages. The encoder and the decoder stage. The encoder stage occurs between the input layer and the hidden layer. And, its goal is to reduce the dimension of the input features to generate a meaningful representation of the data. Then, the decoder stage occurs from the hidden layer to the output layer, and its goal is to reconstruct the encoded features to the original input features dimension. In the end, it outputs a meaningful reconstruction of the input features.

To clarify, consider the architecture in Figure 2.4. It illustrates the representation of a simple autoencoder. In detail, the input layer consists of the input vector $x = [x_1 \ x_2 \ x_3 \ldots \ x_6]^T \in \mathbb{R}^{6 \times 1}$. Whereas the hidden layer has a smaller dimension, being composed by the encoded features, and it is defined as $h = [h_1 \ h_2 \ h_3 \ h_4]^T \in \mathbb{R}^{4 \times 1}$. Lastly, the output layer has the same dimension as the input layer. Being represented by $y = x = [y_1 \ y_2 \ y_3 \ldots \ y_6] \in \mathbb{R}^{6 \times 1}$. It has the same dimension because it minimizes the error from the reconstruction and returns a more meaningful, equivalent representation of the input features. Altogether, each layer can be generally represented by the following equations. Equation 2.1a illustrates the input layer. Equation 2.1b illustrates the hidden layer. And, Equation 2.1c illustrates the output layer.

\[
\begin{align*}
    x &= [x_1 \ x_2 \ x_3 \ldots \ x_6]^T \in \mathbb{R}^{6 \times 1} \\
    h &= [h_1 \ h_2 \ h_3 \ldots \ h_4]^T \in \mathbb{R}^{4 \times 1} \\
    y &= [y_1 \ y_2 \ y_3 \ldots \ y_6]^T \in \mathbb{R}^{6 \times 1}
\end{align*}
\]
In each layer, all signals are combined into one. Particularly, every signal is assigned a weight from an initialized weight matrix $W \in \mathbb{R}^{d \times n}$, and it is added a bias $b = [b_1 \ b_2 \ b_3 ... b_d]^T \in \mathbb{R}^{d \times 1}$. Then, to the result of the sum from all the signals, it is applied an activation function $f_1$. So that, in the end, the neuron can produce an output signal $h$. Therefore, from a set of input features $x = [x_1 \ x_2 \ x_3 ... x_n]^T \in \mathbb{R}^{n \times 1}$, it is applied the expression defined in Equation 2.2.

$$h = f_1(Wx + b) \quad (2.2)$$

The activation function is used to introduce non-linearity to a neural network. Since, in the real world, not all phenomena follow a linear pattern. Therefore, the application of an activation function transforms the domain of the signal based on a certain constraint. For example, shrinking the signal to only being ranged from 0 to 1. As such, in our study, we considered four activation functions. First, we considered the linear activation function where the function output is its input. Second, we considered the sigmoid activation function which produces a continuous range from 0 to 1. Moreover, in Equation 2.3a we represent the expression that transforms the input. Third, we considered the tanh activation function and outputs values between $-1$ and $1$. In Equation 2.3b there is the expression that transforms the input. Last, we considered the ReLU activation function, which outputs 0 if the input is negative and the input value if it is positive. Its expression is defined in 2.3c. In Figure 2.5, we represent the graph for each function.

$$y = \frac{1}{1 + e^{-z}} \quad (2.3a)$$

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (2.3b)$$
\[ y = \max(0, w^T x + b) \]  

(2.3c)

Figure 2.5: Graphics representing each of the activation functions.

2.2 Literature Review

This chapter reviews the literature to understand the current contributions related to our study. Therefore, we will survey the metrics, datasets, algorithms, and evaluation procedures used in the literature for defect prediction, to have a deeper understanding of the latest implementations in the defect prediction process. Furthermore, we will review the studies discussing the data imbalance problem and analyze the state-of-the-art approaches to smell-based defect prediction.
Henceforth, in Section 2.2.1 we will focus on the defect prediction process. Then, in Section 2.2.2 we will analyze studies on data imbalance. And, in Section 2.2.3 we will review smell-based defect prediction approaches.

### 2.2.1 Defect Prediction

To build a defect prediction model, we have to train the model using a set of training instances. However, first, we have to generate those data instances. In general, we would extract them from software archives. Including version control systems (eg. SVN, CVS, GIT), which provide source code and commit messages, as well as, issue tracking systems (eg. Bugzilla, Jira) that contain some defect information. Furthermore, the extracted data instances can be represented by a method, a class, a source code file, a package or a code change, depending on the prediction granularity. Consequently, the training instances are created using metrics. These are defect prediction features that are extracted from the software archive and can represent the complexity of software (eg. lines of code) or the development process (eg. code changes). Additionally, the training instances have labels annexed to them. Therefore, each data instance is labeled as defective or non-defective, depending upon whether or not the module contains a defect. Forthwith, a defect prediction model is built by training a machine learning algorithm with the training instances. Henceforth, it becomes trained to classify whether a test instance is defective or not. In the end, we apply evaluation methods to determine the model’s performance [1].

From the previously described defect prediction process, next, we survey several approaches on those steps. Therefore, Section 2.2.1.A focuses on the extraction of metrics. Section 2.2.1.B describes the software archives that are most commonly used in defect prediction. Section 2.2.1.C present the machine learning algorithms used for defect prediction. Lastly, Section 2.2.1.D describes the most common performance evaluation metrics.

### 2.2.1.A Defect Prediction Metrics

When extracted from the archives, the defect prediction metrics are divided into two broad classes. There are the product/code metrics, that are static and measure properties of the code (e.g. size and complexity), and there are the process metrics, which focus on the historical information of the system (e.g. number of changes, number of developers) [1, 17].

The product metrics are retrieved from the source code. Furthermore, they measure the properties from the source code to assess the complexity of the code. Therefore, it follows an assumption that a
higher complexity implies a higher tendency for the module to be defective. For instance, one of the most commonly used and representative size metrics is the Lines of Code (LOC) metric.

Additionally, Halstead, M. [18] developed a metric where it re-frames a program as a sequence of operators and their associated operands. Then, it analyses the complexity through the count of those components. Moreover, McCabe, T. [19] associated the complexity of the software to the structure of the program. Thus, he introduced several cyclomatic metrics using a control flow graph of the program. These set of metrics were categorized by Radjenović, D. et al. [20] as traditional metrics.

Consequently, with the popularization of object-oriented programming, new code metrics for object-oriented languages were introduced. For instance, Chindamber and Kemerer metrics (CK) [20,21] are the most representative metrics for object-oriented programs. For example, two metrics from CK suite are the number of children (NOC) and the depth of inheritance tree (DIT). To explain, the first is the number of subclasses that are going to inherit the methods of the parent class, and the second is the measure of how many ancestors classes can potentially affect the class. From there, many researchers proposed other object-oriented metrics focused on the volume and quantity of source code [21]. These include: the count of variables, methods, classes and so on.

The other style of metrics are process metrics. These focus on the history information extracted from the software archives. To put it differently, the metrics are represented by the changes that occur over time. Therefore, they quantify several elements of the software development process. These include, changes of source code, the number of code changes, developer information, and others.

Over the past few years, multiple process metrics have been proposed. Therefore, they have been classified into five groups. First, it is the group of metrics based on code-change. These are composed by the relative code change churn. For example, the number of lines that were added, deleted and changed [22]. Also, composed by change, for example, the number of revisions on a file [23]. Then, by change entropy [24], by Code Metrics churn. And, lastly, composed by code entropy [25]. Second, it is the group of metrics based on developer information. These are the number of developers that access individual code units [26]. Then, the fragmentation of developer contributions using developer-module networks [27]. The developer network and social network [28]. The authorship and ownership [29,30]. The relation between developer focus and ownership [31]. The personalized change classification based on defect patterns of the developer [32]. And, lastly, the developer micro interaction history [33]. Third, is the group of metrics based on dependency analysis. These are composed by dependency graphs [34]. By socio-technical networks [35]. By change couplings [36]. By citation influence topic [37].
And, composed by the change genealogy [38]. Fourth, it is a group of metrics based on project team organization. These are related to organizational structure and organizational volatility [39–41]. Last, it is the group that comprises the other metrics. These are popularity [42], for instance, measuring the amount of discussions generated in e-mail archives related to particular source code components and anti-patterns [17] (also bad code smells), the features used in our model.

2.2.1.B Defect Prediction Software Archives

One of the requirements to extract features for the generation of training and testing datasets is the selection of an appropriate software archive. Therefore, next, we describe a set of software archives used in the literature for defect prediction.

Li, Z. et al. [1] presents a list of datasets that where publicly shared by academic researchers and companies. Thus, it summaries, for each dataset, the number of software projects, the number and types of metrics used and the granularity of each data instance (Table B.1).

Another great source to select proper datasets is GitHub. Since, it is one of the most used sources of software artifacts. Containing over 100 million git repositories. However, before considering GitHub as a suitable artifact, it is necessary to consider the perils that may be attached with it [43].

For defect prediction, it is also necessary to get information regarding the defects in each module of the repository. Therefore, Defects4J [44] and BugsDotJar [45] are two examples of databases with real faults that can be reproduced. Furthermore, they also provide interfaces that allow access to the source code of each version for all projects.

2.2.1.C Defect prediction algorithms

The next step too build a defect prediction model is to train a machine learning algorithm. Therefore, we have to choose the best algorithm to fit our data and context. In general, we have to decide what type of learning is the most appropriate for our problem. Hence, Shalev-Shwartz, S. and Ben-David, S. [46] describes four parameters to which learning paradigms can be classified.

First, learning ranges from supervised to unsupervised techniques. Therefore, supervised learning is described when the training instances possess enough information of the actual output, to create a model with enough performance to make good predictions. On the other hand, in unsupervised learning, the training instances have the same information as the testing instances. Thus, we can not rely on the actual output to create an accurate model.
Second, learning is defined by the type of role the model will play. An active learner actively interacts with the environment during the training time. Thus, performing queries and experiments. On the contrary, a passive learner reads the information without changing or manipulating it.

Third, learning is defined by the helpfulness of the teacher. Therefore, the learning process can be supported by a helpful teacher. Since, it provides the most useful information to help the learner achieve its learning goals. On the contrary, learning can occur without the help of a teacher. Thus, the model learns by observing random phenomena of the environment. At the same time, learning can also be influenced by an adversarial teacher. Hence, it follows the principle that if the model can learn against an opposing teacher, then it is able to success interacting with any odd teacher.

Last, learning is defined by whether the model can incrementally learn from a stream of incoming data. Therefore, a model is trained with batch learning when it can only be trained offline with all the available data. Hence, it can not be incrementally trained. On the other hand, with online learning, the model is incrementally fed, with either individual data instances or with small groups called mini-batches.

There are several algorithms that have been proposed in the literature. In particular, the majority of defect prediction models consist of models trained with supervised learning. In the end, Li, Z. et al. [1] summarized a list of relevant state-of-the-art algorithms (Table B.2).

In our solution, we have to consider the imbalance dataset problem. Therefore, we can not rely on the supervised learning style to train our model. Instead, we will focus on the unsupervised batch learning. Where we will train the model to reconstruct with the minimum loss the input features that are not defective. So that, we are able to identify the defective features (the outliers). Since, they will reconstruct poorly.

### 2.2.1.D Evaluation measures for defect prediction

To measure the performance of a defect prediction model, the usual method is to extract information from a confusion matrix. Therefore, as shown in Table 2.1, a confusion matrix summarizes how the model predicts the defect categories against the actual classification.
Table 2.1: Confusion matrix. [1]

<table>
<thead>
<tr>
<th></th>
<th>Predicted Defective</th>
<th>Predicted Non-Defective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Defective</td>
<td>TP</td>
<td>FN</td>
</tr>
<tr>
<td>Actual Non-Defective</td>
<td>FP</td>
<td>TN</td>
</tr>
</tbody>
</table>

From the application of the confusion matrix are returned four results. First, is the number of defective instances that are predicted as defective (also True Positive - TP). Second, is the number of defective instances that are predicted as non-defective (also False Negative - FN). Third, is the number of non-defective instances that are predicted as defective (also False Positive - FP). Last, is the number of non-defective instances that are predicted as non defective (also True Negative - TN).

From the values in the confusion matrix, there are several measures used for performance evaluation. For example, the precision describes how good a model is at predicting the positive class. And, the recall describes how good the model is at predicting the positive class when the actual outcome is positive. In Figure B.1 there is a list of the most common performance evaluation metrics [1].

2.2.2 Data Imbalance

The data imbalance problem is also known as the class imbalance problem. Thus, it typically occurs in classification scenarios, when there are more instances of a certain class compared to others. Chawla V., N., et al. [47] observed that the data imbalance problem made its appearance in the machine learning/data mining research community three decades ago. However, its relevance has grown, as more researchers realized that their datasets were imbalanced and that this imbalance caused supoptimal classification.

Chawla V., N., et al. [47] surveyed the developments in the area of class imbalances. They described the contributions to the three subareas of the class imbalance problem. Sampling, feature selection and one class learning.

First, they present the literature for sampling. In general, it focuses on defining the correct distribution for a learning algorithm. For instance, it has been proved that the natural occurring distribution is not the optimal distribution [48]. Additionally, random undersampling can possibly remove specific input examples. Likewise, random oversampling can lead to overfitting. Therefore, Batista [49] compares the multiple strategies for sampling. These combine over and undersampling techniques. Such as, SMOTE+Tomek or SMOTE+ENN. And, they are applicable to datasets that are highly imbalanced.
Moreover, other contributions were introduced to the literature, which are focused on boost procedures. For instance, the DataBoost and SMOTEBoost [50].

Second, they describe the contributions for feature selection. It focuses on selecting features that lead to a higher separability between the classes. Furthermore, the majority of the contributions in feature selection are focused on the domain of text classification and web categorization [51, 52]. However, Zheng and Srihari [53] suggest that the existing feature extraction measures, used for feature selection, are not appropriate for dataset imbalance. Thus, they propose a new feature selection framework. In detail, it separates features for positive and negative classes, and explicitly combines them.

Last, they present the contributions for one-class learning. They describe it as a recognition-based approach. To put it differently, the model is created using only the examples form the target class. Moreover, they measure the similarity score between the query object and the target class. Consequently, they impose a threshold for the similarity value to create classification [54]. This approach is particularly useful for extremely imbalanced datasets with a high dimensional noisy feature space [8]. In the literature, two categories of one-class learners were introduced. The application of SVMs [55, 56]. And, the application of autoencoders [54, 57]. When comparing both approaches, they have been found to be competitive to one-another [8].

2.2.3 Smell-based Defect Prediction

Smell-based defect prediction, is the application of bad code smells as the metrics in defect prediction to predict faults in software. In this section, we present related work concerning this topic. We focus on the detected smells and the tools used for their extraction, the prediction models used and the evaluation procedures and results.

2.2.3.A Toward a Smell-aware Bug Prediction Model (2019)

In the paper Toward a Smell-aware Bug Prediction Model [3], the authors explore the effectiveness of code smell intensity in defect prediction. They measure the severity of code smells, by adding this metric to existing bug prediction models and comparing the new model against baseline models. The code smell intensity is an intensity index that ranges from 1 to 10.

They extracted the smells from the PROMISE dataset which is a database for software repositories [58]. To detect smells, the authors used JCodeOdor [59] because it detects all the smells they considered in the empirical study and it computes the value intensity on the detected code smells. Which detects six types of code smells, that intersect with a few of the smells considered in this thesis.
To create the prediction models they experimented with several supervised classifiers, namely the Multilayer Perceptron [80], the AdTree [61], the Naive Bayes [62], the Logistic Regression [63], Decision Table Majority [64] and Simple Logistic [65], that output a probability of a module being defective.

To evaluate their research, they compared the models trained with the intensity index against a baseline model trained without the intensity index. To validate their performance, they applied a 10-fold cross-validation strategy, that randomly partitions the original set of data into 10 equal sized subsets, where one is retained as that set while the remaining are used as the training sets. Then, the cross-validation is repeated 10 times. Furthermore, to verify the contribution of the intensity index, they used the precision and recall metrics and applied the F-measure, also known as the harmonic mean of precision and recall and considered the area under the curve that quantifies the overall ability of a prediction model, where the closer to 1 the higher the model’s performance. They also computed the Brier score, that measures the distance between probabilities predicted by a model and the actual outcome.

In the end, they concluded that adding the intensity index as a predictor of defective components increases the performance of the baseline bug models, having observed a case where the prediction accuracy increased up to 47%, when compared with the baseline model not using the intensity metric.


Reshi, Junaid Ali and Singh, Satwinder, in Predicting Software Defects through SVM: An Empirical Approach [4], explore a new approach of defect prediction by using the machine learning algorithm, support vector machines, through the usage of code smells.

They collect smells from several versions the Eclipse source code repository. To collect the code smells, they used the iPlasma tool [66] that provides the means to extract four class level code smells and three method level code smells. Before training, they used the data pre-processing technique WrapperSubsetEval [67] with Evolutionary search to clean and prepare the data for the creation of smell prediction models.

Then, they created models for each version of the software, where each model performance measures were obtained and stored for each run of the model. They used each prediction model to predict defects in the succeeding versions of Eclipse.

To validate the performance of their model, they used Weka’s [67] 10-fold cross validation. After building the models, they used the precision and recall metrics, they calculated the F-measure and applied the
ROC Curve, that summarizes the performance of a binary classifier, by plotting the relation between the true positive rate and the false positive rate at various threshold values.

In the end, despite the performance of the SVM based defect prediction models not showing significant values, the authors concluded that the role of smells in predicting the defects of a software is significant.

2.2.3.C Defect Prediction with Bad Smells in Code (2016)

In the paper Defect Prediction with Bad Smells in Code [6], the author wants to verify how metrics based on bad code smells impact defect prediction in an industrial software development project. To perform this study, they used a critical industry system used in Volvo Group vehicle factories called PROSIT+, which was developed using .NET technology.

To evaluate the impact of smells, they compared using a set metrics which they extracted using SourceMonitor: lines of code, the methods per class, the percentage of comments, the maximum block depth and the average block depth.

They extracted code smells using the Microsoft CodeAnalysis tool, which contains two hundred rules, triggering eleven warnings concerning the design, globalization, interoperability, maintainability, mobility, naming, performance, portability, reliability, security and usage.

To decide on the best prediction mechanism to use, they tested a combination of different classifiers, feature selection and balance algorithms against two datasets, with and without code bad smells. For the classifier, they compared the Naive Bayes, the Random Forest and the PNN. For the feature selection they tried with none, with elimination and with simulated annealing. They tried with and without the Smote algorithm to balance classes with and without defects. To evaluate on the best model they used the F-measure and the best configuration was the dataset with code bad smells that used the Random Forest classifier with the SMOTE algorithm and reversed elimination feature selection mechanism.

After deciding on the prediction model, to verify whether code smells-based metrics can be valuable to defect prediction, they divided all available code into 20 smaller sub-modules with an equivalent size. Each with a total of 700 records after SMOTE oversampling. Then, for each sub-module they collected metrics and smells and generated 20 datasets of SourceMonitor metrics only, 20 datasets of CodeAnalysis metrics only and 20 datasets of combined metrics.
For evaluation, they collected the accuracy and the Cohen's kappa measures for overall results and the F-measure and Recall for defect-prone classes. In the end, the best achieved results had the bad smells metrics included, with an f-measure of 0.9713.

However, the difference between that result and the one with the bad smell included is 0.0059 lower. Therefore, the authors suggest that the use of code smells metrics in software prediction is irrelevant in that industrial environment because of the negligibly small impact of code bad smells based metrics.

2.2.3.D Software Analytics in Practice: A Defect Prediction Model Using Code Smells (2016)

In Software Analytics in Practice: A Defect Prediction Model Using Code Smells [6], Soltanifar, B. et al. study the effectiveness of code smell metrics on prediction models compared against the use of churn metrics and the combination of both. Churn is the measure of the amount of code change taking place within a software unit over time. Therefore, it is the information after it is applied a change. They extracted the Lines of Code Added, the Lines of Code Removed, the Version Count, the Number of Unique Developers and the Age of files that were edited in each code commit.

They extracted their datasets from two industrial projects. The first is a project for a large wireless telecom operator company, it was developed by five programmers for five years, it is composed by 747 unique source files and was written in Java. The second project is a document management component of a web content management application written in PHP and Javascript, it was developed by five programmers during three years and includes 447 unique source code files.

They used a tool called PMD to extract 12 different types of code smells from the first project. For the second project they used a php code sniffer and jshint to extract smells. The types of code smells they extracted were basic, unnecessary, strict exceptions, string and string buffer, clone implementation, naming, J2EE, optimization, braces, code size, empty code, security code, code style, code size class, performance category and security.

They built two different models using Naive Bayers and Logistic Regression algorithms and considered five performance measures. The recall, the precision, the balance, the f-measure and the receiver operating characteristic curve (ROC).

In end, they concluded that using code smells brings a significant impact on the performance of defect prediction models, and the two machine learning algorithms, logistic regression and Naive Bayes with code smells metrics, outperform the ones only trained with churn metrics.
Smell-based Defect Prediction

Approach

“Science is beautiful when it makes simple explanations of phenomena or connections between different observations. Examples include the double helix in biology and the fundamental equations of physics.”

Stephen Hawking.

Contents

| Section                                                        | Page |
|                                                               |      |
| 3.1 Smell-based Defect Prediction Approach                   | 25   |
| 3.2 Smell-based Defect Prediction Methodology                | 39   |
The purpose of this chapter is to discuss the approach we followed to build and evaluate the model and to describe its implementation. Therefore, Section 3.1 describes the approach of our study and Section 3.2 describes the methodology we implemented.

### 3.1 Smell-based Defect Prediction Approach

In this section, we describe the general approach we decided to follow. To answer the research questions we are exploring in this study. Therefore, as illustrated in Figure 3.1, the approach is composed by four steps. First, we apply the feature extraction. Where, from all the versions from a set of repositories we extract a batch of object-oriented metrics, design and implementation smells which, are then turned into datasets. Second, we apply multiple operations to process the dataset. These operations, prepare the data for training and testing. Third, we focus on training the autoencoders. Thus, we make several decisions regarding the architecture and the parameters used for training. Last, we evaluate the different models. Hence, we measure a set of metrics to compare the models.

In short, Section 3.1.1 describes the extraction of features. Section 3.1.2 describes the processing of the datasets for training. Section 3.1.3 describes the training process. Lastly, Section 3.1.4 evaluates the models.

![Figure 3.1: Approach steps overview.](image-url)
3.1.1 Feature Extraction

For the first step in our approach, the goal is to extract three types of features from all the software versions of open-source repositories. Therefore, we iterated through every single buggy version on a set of repositories and, for each module, we extracted a group of traditional object-oriented metrics, design smells and implementation smells. Moreover, we associated to each module the granularity of class. Additionally, we recorded the information regarding whether the module is defective or not. Because, we will use that information for the output of our datasets. Next, we describe the metrics and smells considered during the extraction procedure.

![Feature extraction phase](image)

**Figure 3.2**: Feature extraction phase.

**Metrics** From each module, we extracted a set of eleven traditional object-oriented metrics. In particular, we extracted the following metrics. The number of lines of code in the class (LOC), the number of fields (NOF), the number of public fields (NOPF), the number of methods (NOM), the number of public methods (NOPM), the weighted methods per class (WMC), which is the sum of the complexity of each method in the class, the number of children (NC), which is the number of sub-classes subordinated to the module in the class hierarchy, the depth in inheritance tree (DIT), defined as the number of ancestors in direct lineage from the module, the lack of cohesion in methods (LCOM) which measures the degree to which methods and fields within a class are related to one another and, lastly, we extracted the fan-in and the fan-out metrics. These are the number of modules that call our module and the number of modules that our module calls [68,69]. These metrics are enumerated in table 3.1.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Object-Oriented Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Lines Of Code</td>
</tr>
<tr>
<td>NOF</td>
<td>Number of Fields</td>
</tr>
<tr>
<td>NOPF</td>
<td>Number of Methods</td>
</tr>
<tr>
<td>NOM</td>
<td>Number of Methods</td>
</tr>
<tr>
<td>NOPM</td>
<td>Number of Public Methods</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted Methods per Class</td>
</tr>
<tr>
<td>NC</td>
<td>Number of Children</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of Inheritance Tree</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of Cohesion in Methods</td>
</tr>
<tr>
<td>FANIN</td>
<td>Fan-in</td>
</tr>
<tr>
<td>FANOUT</td>
<td>Fan-out</td>
</tr>
</tbody>
</table>

Table 3.1: Set of traditional object-oriented metrics categories we extracted from modules.

**Design Smells**  From each module, we detected the design smells that may exist in that particular instance. In general, design smells use patterns in the design that imply the violation of fundamental principles and negatively impacts design quality. Therefore, there are four principles that may be disregarded. The abstraction, the encapsulation, the modularization and the hierarchy principle. In the end, considering all principles, we were able to detect a total of 17 design smells, from each module. In table A.1, we describe the design smells we considered.

**Implementation Smells**  Additionally, we detected implementation smells from each module. In general, implementation smells are bad code smells that represent poorly implemented patterns that negatively affect the system maintainability. Therefore, in our context, we considered 10 implementation smells. Hence, we describe them in Table A.2.

### 3.1.2 Dataset Processing

The goal of this stage is to generate the final dataset from the data received in the previous step. This data consists of the metrics, design, and implementation smells from each module, and the corresponding information regarding it being defective (1) or not (0). Then, we use these information to generate three datasets. A dataset with only metrics, a dataset with only smells and a dataset with a combination of all features. Consequently, we prepared the generated datasets for the training and evaluation phases. Therefore, we applied a set of transformations to make the datasets adequate for those operations. In general, we applied three transformations to the dataset. First, we change the data to the
standard-normal form. Second, we use stratification to split the data. Thus, creating a small portion for
the testing dataset and defining the rest as the training dataset. Last, we removed the defective modules
from the training dataset. So that the autoencoder is trained with only the non-defective modules. Thus,
learning to detect the defective module as the outliers.

![Diagram of data processing phase.](image)

**Figure 3.3:** Data processing phase.

**Dataset generation** From the extracted data, we want to generate three different datasets. Thus, each
instance in the dataset is associated to a specific module and its values depend on the type of dataset.
Whether it consists of only metrics, only smells or the combination of both. Additionally, for each feature
we are expecting different types of values. For the datasets with metrics as features, we are expecting
to have positive discrete values. Except, for the lack of cohesion metric, that has positive and negative
continuous values. For the datasets with smells, we are expecting twenty-nine features with binary
values. Therefore, 1 if it was detected in the module and 0 otherwise. Then, regarding the output value,
all the datasets have a binary value. If the module is defective it gets the value of 1, otherwise gets 0.

**Feature scaling - Standardization** In the features values, higher magnitude values outweigh the lower
magnitude values. Therefore, to solve this issue several feature scaling methods were introduced to
balance those magnitudes [70]. We decided to apply standardization, or z-score normalization. In
detail, it re-scales the features so that they’ll have the properties of a standard normal distribution. This
means that, the mean becomes 0 and the standard deviation becomes 1 (μ = 0, σ = 1). In short,
it removes the mean and scales the features to unit variance. Therefore, equation 3.1.2 describes the
expression to calculate the standard score z for a sample x, with a mean μ and a standard deviation
σ [71]. In the end, the main reason behind the application of standardization is that the standardized
features facilitate the optimization of algorithms used in neural networks [72, 73].
\[ z = \frac{x - \mu}{\sigma} \]  

(3.1)

**Dividing dataset - Stratified sampling**  To split the datasets into training and testing data, we decided to apply a stratification sampling approach. The goal of this method is to partition the data into subpopulations (called stratum). Therefore, to divide and shuffle the dataset into a testing and training dataset. In particular, while maintaining the proportions of samples for each class. The reason behind this decision was due to the data imbalance in our dataset. Since we have a significant amount of non-defective modules compared against the defective modules. And, so that we can avoid the possible scenario where the testing dataset only has non-defective modules.

**Removing Non-Defective instances**  When training the autoencoder, we will teach it how to detect the defective modules as the outliers. Therefore, we will build a network that minimizes the reconstruction loss for the non-defective modules prediction so that we can distinguish the defective modules. Since, they will have a significantly higher loss score when reconstructing the features. With this in mind, the last step in the data processing stage is to remove the defective modules from the training dataset. Ultimately, to generate a model that only learned to reconstruct effectively non-defective modules.

### 3.1.3 Model Training

The purpose of this section is to discuss the training phase of the model. Therefore, we will discuss the description and decisions about the architecture for each model. In particular, we will consider how does the input and the output influenced the architectural decisions. Thus, we will discuss the two variants of the five different model architectures based on the datasets provided for training.

In our study, the datasets we are considering are extremely imbalanced. Therefore, we have an extremely small percentage of modules that are defective. Thus, using supervised learning classification is not enough to accurately predict defective modules. With this in mind, our solution will apply the outlier detection strategy. In particular, we will use unsupervised learning to train the model to detect defective modules as outliers in the data [74], to put it differently, the goal of outlier detection is to identify data objects that do not fit well within the general data distribution. Thus, in our context, we would want to create a model that only fits well with the general population (the non-defective modules). With this, the outliers are the instances that do not fit well. As such, they would be considered the defective modules.
In our solution, we decided to use autoencoders to predict defective modules. Hence, we trained the model using non-defective modules to learn how to reconstruct them with minimum loss. Consequently, our goal is to create a model that produces a low reconstruction error when given a non-defective module and a high reconstruction error when given a defective module. With this, to create classification, we defined a threshold for the reconstruction error that ascertains whether a given module is defective or not.

The architecture of an autoencoder is defined by a set of criteria that range from the number of layers in the autoencoder to the activation function in each layer. Currently, there is not enough research directing us toward an optimal architecture to choose from. As such, there is not a general guideline to help us decide on the best parameters. Therefore, we applied an empirical observation, comparing five distinct models against three different datasets. In particular, we defined the distinction between architectures based on the number of layers and the activation function in each layer, and the dimension in each layer [75]. With this in mind, we used autoencoders with shallow and deep architectures. And, considered the linear, sigmoid, ReLU, and tanh activation functions. Also, we followed a specific rule for the layer’s dimensions.

**Criteria 1: Number of layers and activation functions** For the shallow layers, we considered two hidden layers architectures with the activation function as the change factor. We created an autoencoder with two hidden sigmoid functions, another with two hidden tanh functions and one with two hidden ReLU functions. For the two deep autoencoders, the change factor between both was the number of layers. Thus, both of them used the ReLU activation functions, with the first architecture having four hidden layers and the second architecture having six hidden layers. Regarding the output layers, as the
autoencoders require the output domain to be the same as the input domain, we decided to use the sigmoid activation function on the autoencoders trained with the smells-only dataset, making the values range between 0 and 1. For the other two datasets, we applied a ReLU activation function, due to the metric values of the input layer not being within a specific range and only having positive values.

Criteria 2: Layers dimensions Another criterion to consider, while defining an autoencoder architecture is the number of features each layer has. The general rule we applied for the shallow networks was, from the input layer, to divide the number of features by two in the following layer. Then, to divide by two again in the second hidden layer. And the last layer, the output layer, to go back to the same number of features as the first layer. For the first deep network architecture, from the input layer, to divide the dimension of features by two for the first hidden layer. Then, to divide by two for the second hidden layer. And, to divide by two, again, for the third hidden layer (the mid-layer). From there, to multiply by two for the fourth hidden layer. And, the output layer to have the same dimension as the input layer. For the second deep network, from the input layer, for the first three hidden layers we divide each by two. Then, for the fourth hidden layer (the mid-layer) we define the dimension as 1. Then, for the last two hidden layers, we multiply each by two. And, the output layer had the same dimension as the input layer.

Loss Function To measure the loss, we decided to apply the mean square error (MSE) loss function. The mean square error of an estimator is the average squared difference between the estimated values and what is estimated (equation 3.1.3). The result from this calculation is always going to be non-negative and the values that are closer to zero will be the most accurate. In our context, having an output value closer to zero means that the corresponding input value reconstruction is nearly perfect. One of the main reasons behind the decision to choose the MSE function was due to the heavy weight it assigns to the outliers [76]. Since, the result squares each term, giving a greater emphasis on large errors compared against small errors. Thus, this function fits well within our context because our end goal is to detect the outliers in our model.

\[
MSE = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2
\]  

(3.2)

To summarize, figure 3.5 illustrates the overview of the process just described. Given a set of input features as instances, we apply the reconstruction using the model. Then, from the reconstruction process, we apply the MSE to get the reconstruction error. Therefore, we classify the instances by comparing the error with the threshold. Thus, labeling them as defective or non-defective.
3.1.4 Evaluation approach

In this section, we analyse each model regarding the used dataset and the autoencoder architecture. Therefore, we applied two evaluation procedures that rely on different outcomes of the confusion matrix. These procedures are the receiver operating characteristic curve [77] and the precision-recall-gain curve [78]. Furthermore, we considered the area under the curve from both curves.

With this in mind, this section is outlined as follows. Section 3.1.4.A characterizes the confusion matrix and its outcomes. Section 3.1.4.B describes the application of the receiver operating characteristic curve and the calculation of the area for this curve. And, Section 3.1.4.C describes the application of the precision-recall-gain curve and the calculation of the area under the curve.

3.1.4.A Confusion matrix

In our problem scenario, the datasets we are considering for training have a binary class output label. Therefore, between the predicted values and the actual output values, there are four possible outcomes. These include, true positives, true negatives, false positives, and false negatives. Under those circumstances, if the instance is defective and it was predicted as defective then it is a true positive outcome. On the other hand, if it was instead predicted as non-defective then it is a true negative outcome. If the instance is non-defective and it was predicted as non-defective then it is a false positive outcome. Otherwise, if it was predicted defective, then it is a false negative outcome. With this, given a classifier
and a set of instances from the testing dataset, we can build a confusion-matrix, as illustrated in figure 3.7. Thus, it represents the distribution of the different outcomes regarding the predicted values that are defective (\(\hat{d}\)) and non-defective (\(n\)), and the true values that are defective (\(d\)) and non-defective (\(n\)) [77].

![Confusion Matrix](image)

**Figure 3.7:** Confusion Matrix

### 3.1.4.B Receiver Operating Characteristic Curve

The receiver operating characteristic (ROC) is a graphical plot that describes the diagnostic ability of a binary classifier, as its discrimination threshold varies. Furthermore, we have to calculate the true positive rate (TPR, also hit rate) and the false positive rate (FPR, also false alarm rate). The first measures, from a set of defective modules, how many of the modules that we predicted defective were actually defective. The second measures, from all the non-defective modules, how many of the
modules that we predicted defective were actually non-defective. In short, both of these expressions are described in Equations 3.3 and 3.4.

\[ tp \ rate = \frac{Positives \ correctly \ classified}{Total \ positives} = \frac{TP}{d} = \frac{TP}{TP + FN} \]  

(3.3)  

\[ fp \ rate = \frac{Negatives \ incorrectly \ classified}{Total \ negatives} = \frac{FP}{n} = \frac{FP}{FP + TN} \]  

(3.4)  

Considering the graphical representation, the ROC Curves are two-dimensional graphs with the TPR plotted on the y-axis and with the FPR plotted on the x-axis. As a result, this graph summarizes the trade-offs between the benefits (true positives) and the costs (false positives). As an illustration, consider figure 3.8, where are described five different discrete classifiers with distinct TPR and FPR values.

![Figure 3.8: A basic ROC graph showing five discrete classifiers. [77]](image)

When analyzing the extremes of the graph. The point (0, 0) represents a "conservative" strategy. In this strategy, the model does not commit any false positive mistakes but also does not detect any defective module (true positive). On the other extreme, the point (1, 1) represents a "liberal" strategy. In this strategy, the model unconditionally predicts all of the modules as defective. For example, in figure 3.8, model A is more "conservative" than B.
Furthermore, model D is represented by the point (0, 1). This point describes the perfect model. Therefore, it predicts all the defective modules correctly. And, it does not predict the modules that are non-defective as defective.

The diagonal line $y = x$ represents the strategy of randomly guessing a class. Thus, if the model predicts a fraction of $k$ modules as being defective, due to randomness, then the same fraction of $k$ defective modules will be predicted defective (true positives). Similarly, the same fraction $k$ of non-defective modules will be predicted defective (false positives). For example, the classifier C, on point $(0.7, 0.7)$, describes a random model. As it randomly guesses positive classes 70% of the time.

Moreover, the models that are represented in the graph on the lower right triangle appear to perform worse than the random guessing. For instance, as described by classifier E. However, the negative performance can be inverted if we reverse every prediction on every single decision of the classifier. Therefore, any point of the lower part of the graph can be negated to generate a point in the upper part. With this, we determine that only the points closest to the $y = x$ line provide the least amount of information concerning the model performance. On the other hand, the classifiers described as being closest to the points (0, 1) and (1, 0) provide the most information [79].

Multiple classifiers, including decision trees or rule sets are built to generate a single class decision. Such as whether the module is defective or non-defective. This type of classification produces a single confusion matrix that defines a single ROC Point (as the ones in figure 3.8). Other classifiers, for example Naive Bayes or neural networks, which fit within our implementation, produce an instance probability or score, which describes the degree to which an instance is a member of a class. In our context, we would reconstruct the input feature from the test dataset and use the mean squared error value between the input and the predicted feature as the score to measure how close a module is from being defective, as illustrated in figure 3.5. This score can be used within each threshold to produce a discrete classifier: defective if above the threshold level and non-defective if below.

As each threshold generates a different point in the ROC space, when aligning them all together, we can conceptually visualize a curve whose thresholds range from $-\infty$ to $+\infty$. For example, figure 3.9 illustrates the ROC curve. It considers a list of score points, where each point defines a threshold. Therefore, the threshold of $-\infty$ produces the point (0, 0) and the threshold $+\infty$ the point (1, 1). As a result, lowering this threshold leads to a migration from a more "conservative" area of the graph to a more "liberal" area.
Figure 3.9: The ROC "curve" created by thresholding a test set. The graph shows the corresponding ROC curve with each point labeled by the threshold that produces it, each represented with the respective classification score. [77]

To evaluate the model's performance using with the ROC methodology, we can use the area under the curve metrics. It ranges from 0 to 1. And, the values closer to 0.5 summarize a set of lesser skilled models compared with models with scores near 0 and 1. This metric is defined in Equation 3.5.

\[
ROCAUC = \int_0^1 tpr \, d fpr
\]  

(3.5)

3.1.4.C Precision-Recall-Gain Curve

In this study, we are considering datasets that are extremely imbalanced. Therefore, the non-defective modules that were extracted from the repositories are higher than the number of defective modules. With this in mind, it has been proved that this kind of imbalance in data, when evaluated using a ROC curve, leads to a deceptive interpretation of the plot regarding the model's performance. As a result, an alternate procedure would be to use the precision-recall curve to evaluate the model's performance. Since, it evaluates the fraction of true positives among the positive predictions [9].
Given that, from all the modules that were predicted as defective, precision is the fraction of those modules that were actually defective. On the other hand, recall is defined as the fraction from the defective modules that were predicted defective. The precision is expressed by Equation 3.6 and recall is expressed by Equation 3.7 [80].

\[
\text{precision} = \frac{\text{Positives correctly classified}}{\text{Total predicted positives}} = \frac{TP}{TP + FP} \tag{3.6}
\]

\[
\text{recall} = \text{tp rate} = \frac{\text{Positives correctly classified}}{\text{Total positives}} = \frac{TP}{d} = \frac{TP}{TP + FN} \tag{3.7}
\]

Nevertheless, the application of the precision-recall curve lacks many properties that are included in the ROC curve. Thus, it does not have an universal baseline, due to its baseline being a horizontal line that is dependent on the class distribution. Also, it has no linear interpolation [81]. And, it has a non-convex Pareto front, wherein the absence of linear interpolation the set of non-dominated operating points do not form a convex curve. Besides, the curve is not straightforward to interpret by visual inspection. Then, its area under the curve is uninterpretable [82]. Thus, it does not provide any meaningful interpretation. Beyond, the geometric interpretation of the expected precision, when uniformly varying the recall. Furthermore the plot has unachievable areas at the lower right side.

As a result, Flach, P. and Kull, M. [78] proposed a solution to these issues by plotting the Precision-Recall curve in a different coordinate system. In general, they apply a min-max harmonic scale, to rescale the precision and the recall. To best understand the motivation behind this rescaling, we have to consider the weighted harmonic mean defined in equation 3.8.

\[
F_\beta \triangleq \frac{1}{1+\beta^2/\text{precision} + \beta^2/\text{recall}} = \frac{(1 - \beta^2)TP}{(1 + \beta^2)TP + FP + \beta^2FN} \tag{3.8}
\]

A random classifier that predicts defective modules with probability \( p \), and with the proportion of positive classes \( \pi \), has the \( F_\beta \) score of \( (1 + \beta^2)p\pi/(p + \beta^2\pi) \). This score increases along with the value of \( p \), that ranges from 0 to 1, until it reaches the maximum value for \( p = 1 \). Which is the always positive classifier (it is equivalent to the point \( (1, 1) \) in the ROC Curve). Therefore, a characteristic that distinguishes the precision-recall analysis from the receiver operating characteristic accuracy, which
gives the same weight to instances correctly predicted regardless of their class, is that the baseline to beat is the always-positive classifier rather than the random classifier. This baseline is set by the values \(\text{precision} = \pi\) and \(\text{recall} = 1\). Thus any model with the values of \(\text{precision} < \pi\) or \(\text{recall} < \pi\) loses against this baseline. Thus it makes sense to only consider the values of precision and recall to be between \([\pi, 1]\), which leads to the rescaling of the calculated precision and recall values.

To rescale any real-valued variable \(x \in [\text{min}, \text{max}]\), it is only necessary to map \(x \mapsto \frac{x-\text{min}}{\text{max}-\text{min}}\). Besides, due to not being appropriate to scale linearly, they applied, instead, the harmonic scale. The general scaling expression is defined in equation \(3.9\).

\[
\frac{1/x - 1/\text{min}}{1/\text{max} - 1/\text{min}} = \frac{\text{max} \cdot (x - \text{min})}{(\text{max} - \text{min}) \cdot x} \tag{3.9}
\]

With \(\text{max} = 1\) and \(\text{min} = \pi\), we applied the previous expression to define both precision-gain (also precG) and recall-gain (also recallG). Precision-gain is defined in equation \(3.10a\). And, recall-gain is defined in equation \(3.10b\).

\[
\text{precG} = \frac{\text{prec} - \pi}{(1 - \pi)\text{prec}} = 1 - \frac{\pi}{1 - \pi} \frac{FP}{TP} \tag{3.10a}
\]

\[
\text{recG} = \frac{\text{rec} - \pi}{(1 - \pi)\text{rec}} = 1 - \frac{\pi}{1 - \pi} \frac{FN}{TP} \tag{3.10b}
\]

In the precision-recall-gain, the always-positive classifier is described by a precision-gain of 1 and a recall-gain of 0. Therefore, it is plotted in the lower right end corner of the precision-recall-gain space, regardless of which class distribution was used. The baseline is described by the minor diagonal that goes from \((1, 0)\) to \((0, 1)\). These characteristics are illustrated in the right figure of \(3.10\), which is the plot of a precision and recall gain curve. Compared against the precision-recall curve that is illustrated in the left figure of \(3.10\).

To evaluate the model’s performance and compare against other models, we can apply the area under the Precision-Recall-Gain curve. Its values range from 0 to 1 and summarize the skill of the model. This metric is defined in equation \(3.11\).
\[ AUPRG = \int_0^1 \text{precG} \, d\text{recG} \]  

(3.11)

3.2 Smell-based Defect Prediction Methodology

In this section, we describe the implementation regarding the approach we followed in this study. Therefore, we will describe the tools to extract features from the software archives, how did we generate and process the datasets, and the libraries and implementation to train and evaluate each model. Moreover, the implemented tools can be found in https://github.com/Bruno81930/extreme-imbalanced-sbdf and the generated datasets can be found in https://doi.org/10.5281/zenodo.3380274.

With this in mind, Section 3.2.1 describes the methodology to extract features from the software archives. Section 3.2.2 describes the methodology to prepare the data for training. Section 3.2.3 describes the methodology to build the model. And, Section 3.2.4 describes the methodology to evaluate each model.

3.2.1 Methodology for feature extraction

From this process, the end-goal is to extract the data to produce the datasets. Thus we extracted four properties from the software archives. First, we extracted traditional object-oriented metrics. Second,
we extracted design smells. Third, we extracted implementation smells. And, lastly, we extracted whether each module is defective or not. Therefore, we iterated through a set of versions from different projects and, for each module, we collected all the required features and the defect information.

This section is outlined as follows. In Section 3.2.1.A and 3.2.1.B we describe the software archives we extract features from. Thus we describe two different repository databases with a set of multiple real bugs. First Defects4J [44] and, second, BugsDotJar [45]. Then, in Section 3.2.1.C we describe the process of features extraction. As well as Designite [83], the tool to extract features. Lastly, in Section 3.2.1.D we present the procedure to retrieve the output information from modules. To whether the module is defective or not.

3.2.1.A Interacting with repositories from Defects4J

Defects4J is a database and a framework that provides 357 real bugs, created to support testing research [44]. Additionally, Defects4J's architecture is composed by three layers. The bottom layer contains both the real faults metadata database and the "version control" mechanism, to which all the faults database are connected to. The version control component, instead of containing the faulty and fixed program versions and tests, has a list of references that point to each project version control system. Therefore, because the repositories’ version control systems are heterogeneous, it was introduced a second layer that abstracts the database. This layer provides an abstraction of version control systems and an abstraction of build systems for the different projects. Lastly, the topmost layer has the test execution framework. Its main goal is to reduce the researchers’ effort of having to (re-)implement common test-related tasks. Hence, this layer provides an API that offers the ability to monitor test execution, manipulate test suites, generate tests, do mutation analysis, and code coverage analysis. Furthermore, it also contains version control-oriented commands, such as an info command that provides the summary of a project or a bug, and a checkout command that provides access to the source code of a particular project version.

To get access to each module in every version (to extract all the features and the output), we iterated through every defective software version and retrieved the source code for every revision. The procedure to get the source code from all the repositories is described in algorithm 3.1. The algorithm is summarized as follows. For every project in the Defects4J database, we start by using the info command. Then, we use regular expressions to filter out the number of bugs the project has. And, to get access to each source code project, we iterate through every bug, using the checkout command, using as arguments the name of the project and the version of the bug(project version). In the end, we obtained access to the source code from each version of all the projects in Defects4J.
**Algorithm 3.1:** Defects4J project versions iteration

```java
foreach projectName ∈ Defects4JDatabase do
    projectInformation ← defects4j info -p projectName;
    numberOfBugs ← extract number of bugs from projectInformation;
    for versionID ← 0 to numberOfBugs do
        defects4j checkout -p projectName -v versionID;
```

### 3.2.1.B Interacting with repositories from BugsDotJar

BugsDotJar is a large-scale dataset that provides 1158 bugs and patches collected from 8 large popular open-source projects. Since, it was built to assist research in automated debugging, patching and testing of Java programs [45]. Moreover, BugsDotJar is available as a repository on Github. Where, every directory is a specific project repository. Therefore, to access a specific buggy version of a project, users must check out to a specific branch using the corresponding BugsDotJar ID. A Buggy version, also defined as bug instance, is composed by four artifacts. First, it is composed by the buggy version of source code. Second, by the bug report. Third, by the bug reproducing test-suite. Last, by the developer’s patch to fix the bug. Additionally, because there was not enough space to keep all the snapshots of the buggy versions in the same archive, the authors decided to design a tree data structure for BugsDotJar. And, stored it in a GitHub repository with links referencing to the original subject repositories. As such, inside each project, the authors leveraged several Git features to create flexibility and transparency in managing the buggy versions. For illustration, consider figure 3.12, where each project contains a branch with $n$ commits, ranging from $C_1$ to $C_n$. Furthermore, all the buggy instances (with the four artifacts) are branches reconstructed from a developer’s commit that fixed a bug ($C_1$). Hence, connecting to the buggy instance ($B_1$) that is a branch identified by a particular BugsDotJar ID (for instance, BugsDotJar-Bug-1).
With this in mind, to produce the training instances, we used the git command line API to iterate through all the buggy branches from all the repositories. Thus, gaining access to the source code. In general, the procedure to obtain the source code from all the repositories in the BugsDotJar database is described in algorithm 3.2. The algorithm is summarized as follows. First, we iterate through all the projects (sub-directories) in the root folder. Then, we use the command `git branch` to access all the branches in the current project (each branch is a version). Consequently, we filter the branches to only get the buggy versions of the project. And, we iterate through those versions to check out each source code project, using the `git checkout` command.

**Algorithm 3.2: BugsDotJar project version iteration**

\[
\text{foreach } \text{projectName } \in \text{BugsDotJarDatabase} \text{ do} \\
\text{allBranches } \leftarrow \text{git branch -a;} \\
\text{buggyVersions } \leftarrow \text{extract the buggy version from all the branches;} \\
\text{foreach } \text{versionName } \in \text{buggyVersions} \text{ do} \\
\text{git checkout versionName}
\]

### 3.2.1.C Extracting features with Designite

After gaining access to each version from the software archives, the following step was to extract all of the required features for the datasets. As such, we extracted a set of traditional object-oriented metrics, design smells and implementation smells. Therefore, we used Designite [83] to extract all of
those features. For an overview of all the extracted metrics, the traditional object-oriented metrics are represented in table 3.1. And, the design and implementation smells are displayed in tables A.1 and A.2.

Designite is a software design quality assessment tool. It detects a comprehensive set of design, and implementation smells, and provides mechanisms such as detailed metrics analysis. Therefore, to extract all the features, we ran Designite and provided, as arguments, the input and output directories. For the input, we provided the primary source directory path, from the version we are extracting the features. And, for the output, we provided a directory, created hold the list of metrics, design smells and implementation smells for that particular version. Consequently, Designite iterated recursively, through all the source files (.java files) and extracted all the metrics and detected all the smells. As the primary source code’s root directory may change regarding the repository, we analyzed the primary directories regarding each type of repositories. For instance, for the bugsDotJar repositories, we considered the main/ folder as the primary folder. And, for the Defects4J repositories, within the src/ folder, we filtered out the following set of directories: {test/, tests/, javadoc/, resources/, example/, examples/, gen/, experimental/ and swt’}. Furthermore, for the output, we provided as an argument the output directory, to which Designite generated three files. One for the design smells, one for the implementation smells and one for the traditional object-oriented metrics.

In algorithm 3.3, we describe the feature extraction procedure. And, in tables C.1, C.2 and C.3 we preview a sample of the output data for the metrics, design, and implementation, respectively.

---

**Algorithm 3.3: Extracting features with Designite**

**Input:** Project version source code path and output path  
**Output:** Object-oriented metrics, design and implementation smells initialization;  
if Project from BugsDotJar then  
| sourceDirectory ← /main;  
else if Project from Defects4J then  
| sourceDirectory ← /src;  
| sourceDirectory ← filter sub-directories [test/, tests/, javadoc/, resources/, example/, examples/, gen/, experimental/ and swt’];  
| designite -i sourceDirectory -o outputPath;

---

3.2.1.D Capturing defect information from each module

In general, the procedure to get the information about whether a module is defective or not, consists of accessing the bug information from each version. And, record the name of the module (class) that contains the defect.
For Defects4J repositories, to extract the defective information we ran the command info for each version. It outputs a report with information about a particular software version, thus it includes the defective module. Therefore, to collect the defective module, we filtered the name of the defective class. In algorithm 3.4, we describe this procedure.

**Algorithm 3.4: Capturing defect information with defects4j**

**Input:** The project name and current version id

**Output:** The name of the defective class

```
versionReport ← defects4j info -p projectName -i versionId;
defectiveClass ← filter the defective class name from the versionReport;
```

For the Bugsdotjar repository, within every version’s folder there is a directory called `.bugs-dot-jar/`. This directory has a file, called `developer-patch.diff`, that contains the piece of code that was fixed in the current version. Therefore, to store the name of the defective class, we extracted the name of the class that was fixed in that version. In algorithm 3.5, we describe this procedure.

**Algorithm 3.5: Capturing defect information with bugsdotjar**

**Input:** Already in the version directory

**Output:** The name of the defective class

```
versionReport ← read ".bugs-dot-jar/developer-patch.diff" file;
defectiveClass ← extract the defective class name from the versionReport;
```

### 3.2.2 Methodology for data processing

In this step of the approach, from the collected metrics, the goal is to produce three datasets, a metrics-only, a smells-only and a combination of metrics and smells. Therefore, we iterated through every version and retrieved, for each module (which is an instance in the dataset), the corresponding features based on the dataset. Furthermore, we also included the output information for each module and assigned the value 1 if the module is defective and the value 0 otherwise.

As such, this section is outlined as follows. In Section 3.2.2 we describe the generation of the datasets from the metrics. Additionally, after producing the datasets, we applied a set of operations to prepare the datasets for training. So, we standardized all the features, stratified the datasets and removed the non-defective instances from the data. Therefore, in Section 3.2.2 we describe the standardization procedure. In Section 3.2.2 we describe the stratification procedure. And, in Section 3.2.2 we describe the procedure to remove non-defective instances.
Datasets generation  To generate the datasets we iterated through all of the lines from the metrics files. Then, for each module name, we collected the correspondent features for each specific dataset. For the metrics-only dataset, we extracted each traditional object oriented metric value. And, for the cells with more than one value, we applied the arithmetic mean. For the smells-only dataset, we collected the design and implementation smells that were detected in each module. However, because each class can have different combinations of features, we forced each instance in the dataset to consider all the smells. So that, for the ones that were detected, we assigned the value of 1 and to the others, the value of 0. Moreover, we also appended, to each module in the datasets, the value of 1 or 0 based on whether the class is defective or not. In detail, we describe both metrics-only and smells-only dataset generation in algorithms 3.6 and 3.7. Additionally, we also created a third dataset with the combination of both metrics and smells. Thus we called it metrics-and-smells dataset.

**Algorithm 3.6: Generating metrics dataset**

```
Input: Extracted metrics and defect information from a project version using Designite  
Output: Metrics dataset  
dataset = {};  
for each line in extracted metrics do  
    moduleName ← line["Package Name"] + line["Type Name"];  
    if ∃X ∈ [NOF’, FANOUT] where |line[X]| > 1 then  
        line[X] = AVG(line[X]);  
        dataset[moduleName] ← line[NOF’:FANOUT];  
    if moduleName is defective then  
        dataset[moduleName] += 1;  
    else  
        dataset[moduleName] += 0;
```

Traditional object oriented metrics rescaling: Standardization  To apply standardization to the metrics and metrics-and-smells datasets, we used the Scikit-learn Python library \[84,85\]. In particular, we used the function `sklearn.preprocessing.StandardScaler`. Before applying the standard scaler, we reshaped the data to conform to the 1-D data input requirement. To demonstrate, in the following sample of code (listing D.1), we present an example for three metrics of the Python code we used to apply standard scaling.

Training and testing dataset generation: Stratification  For stratification, we also used a scikit-learn function called `sklearn.model_selection.StratifiedShuffleSplit`. It receives as input the number of splits (which is the number of re-shuffling and splitting iterations), the test size and the random state (which is the seed used for the random number generator). For instance, we considered one split, the test size being 20% of the original dataset and the seed of 42 to the random number generator. As an illustration, in listing D.2, we present the sample of code we implemented.
**Algorithm 3.7:** Generating Smells dataset

**Input:** Extracted design and implementation smells and defect information from a project version using Designite

**Output:** Metrics dataset

```plaintext
dataset = {}

for Design And Implementation Smells do
    foreach line ∈ extracted smells do
        moduleName ← line[‘Package Name’] + line[‘Type Name’];
        if moduleName ∉ dataset then
            initialize X values in dataset[moduleName] as 0, where X is the number of design and implementation smells;
            foreach detectedSmell ∈ line do
                dataset[moduleName][defectedSmell] ← 1;
        
        foreach moduleName in dataset.keys do
            if moduleName is defective then
                dataset[moduleName] += 1;
            else
                dataset[moduleName] += 0;
```

**Non-defective modules elimination** To remove the defective modules from the training datasets, we used the Pandas’ Python library [84, 86]. As it provides capabilities to remove the rows with the output value of 0. This is described in the sample of code presented in listing D.3.

### 3.2.3 Methodology for model training

After generating the datasets, the following step is to train the models. Therefore, we trained three models for each of five different architectures. In particular, we used distinct autoencoder architectures. And, we implemented them using the Keras Python library [87]. In detail, we used this library to define and train each model.

In general, we used the same hyperparameters in all the architectures. For instance, we used the same number of epochs. Which is the number of complete iterations through the training dataset. And we used the same batch size. Which is the number of samples that were propagated through the network. We understand that the best course of action would be to experiment with a range of several values. However, we did not have the time resources to conduct such experiment. Therefore we chose to execute 100 epochs for a batch size of 128.

The first three architectures have the same number of layers (two hidden layers) and the same dimension in each layer: \( X \rightarrow (X/2 \rightarrow X/4) \rightarrow X \), with \( X \) being the number of input features. The variations between them are the activation functions used in each hidden layer. Therefore, there is
one architecture with only the sigmoid activation function, one with the ReLU activation function and another with the tanh activation function. Considering this variation, in Listing D.4, we describe the implementation of the two hidden layers ReLU activation functions, where, to get the others architecture it is only necessary to replace the parameter in "activation = 'relu'" from relu to sigmoid or tanh.

The fourth architecture autoencoder has four hidden layers and the dimensions in the layers are, respectively, \(X \rightarrow (X/2 \rightarrow X/4 \rightarrow X/8 \rightarrow X) \rightarrow X\). Thus, we describe the implementation in Listing D.5.

The fifth architecture autoencoder has six hidden layers and the dimensions in the layers are, \(X \rightarrow (X/2 \rightarrow X/4 \rightarrow X/8 \rightarrow 1 \rightarrow X/8 \rightarrow X/4) \rightarrow X\). Thus, we describe the implementation in Listing D.6.

Furthermore, in each architecture, there is a small variance as a consequence of the dataset that is used. As such, for the smells-only dataset, as it only has a binary input, we used the sigmoid activation function for the output layer. On the other hand, for the other datasets we used the ReLU function, which only outputs positive values.

### 3.2.4 Methodology for model evaluation

To evaluate the model, our approach consists of three phases. First, the calculation of the reconstruction error, which measures the loss from the reconstruction. Second, the calculation of the receiver operating characteristic and its area under the curve metric. And, third, the calculation of the precision-recall-gain curve and the corresponding area under the curve.

**Calculating the reconstruction error** After training the autoencoder, the first step to evaluate our model is to reconstruct the input features of the testing dataset. And, then, to calculate its reconstruction error. In other words, to calculate the loss value that represents the distance between the reconstructed features and the input features. In general, the expression we used to calculate the loss is the same we used for training. Therefore we used the mean squared error. With this in mind, to calculate this function we use Pandas and Numpy (np) [84, 88] Python libraries to calculate the mean square error on each instance. This implementation is represented in listing D.7.

**Calculating the Receiver Operating Characteristic curve and AUC** With the previously calculated reconstruction error and the testing dataset output values, we can provide these values as parameters
to a receiver operating characteristic function provided by scikit-learn, called sklearn.metrics.roc_curve. Thus, it returns three lists as the output, the true-positive-rate, the false-positive-rate and the thresholds. With these values, we can plot a ROC curve with the false-positive-rate in the x-axis and the true-positive-rate in the y-axis. And, using the sklearn.metrics.auc, we can also get the area under the curve for the ROC curve. In listing D.8 we describe this implementation.

Calculating the Precision-Recall-Gain curve and AUC To calculate the precision-recall-gain, Flach, Peter and Kull, Meelis [78] provide a set of functions that generate the Precision-recall curve and calculate the area under the curve (AUCPRG). Therefore, the first function receives as arguments the previously calculated reconstruction error and the true values from the classification. Then, the function returns a dictionary with the precision-gain and recall-gain. The second function receives the values produced by the first function and calculates the correspondent area under the curve. The implementation is described in listing D.9.
Results

“Torture the data, and it will confess to anything.”

Ronald Coase,

Contents

4.1 Methodologies .......................................................... 51
4.2 Overall analysis .......................................................... 60
The purpose of this goal is to present the results from the models evaluation. In particular, the comparison between the curves and the areas. Therefore, we will compare the curves and areas under the curve for the three datasets, for each architecture using both Receiver Operating Characteristic and the Precision-Recall-Gain methodology. In the end, we will do a global comparison of all the architectures for each methodology and a comparison of all the areas.

As such, this chapter is outlined as follows. Section 4.1 describes the comparisons between datasets for each individual architecture, applied for the two methodologies. Moreover, Section 4.2 presents the overall comparisons between architectures and between methodologies.

### 4.1 Methodologies

In this section, we compare the results from each architecture for both methodologies. Thus, we display two graphs for each architecture. The first shows the curve regarding the methodology applied to the three datasets and includes the the area under the curve for each dataset. The second shows the differences in area between the three datasets. Therefore, it translates the differences in performances between models trained with a specified architecture based on the dataset used. With this in mind, this section is outlined as follows. Section 4.1.1 shows the results for the five architectures evaluated with the Receiver Operating Characteristic methodology. And, Section 4.1.2 shows the results for the architectures evaluated with the Precision-Recall-Gain methodology.

#### 4.1.1 Receiver Operating Characteristic Methodology

The Receiver Operating Characteristic curve (ROC Curve) can be interpreted as follows. The baseline curve is a straight line defined as $y = x$ that goes from the point $(0,0)$ to the point $(1,1)$. This curve represents the performance of a random classification model with an area under the curve of 0.50. Therefore, a model with an area superior to 0.50 with its curve bowing toward the point $(0, 1)$ shows a greater accuracy than a random model. Hence, a model closer to the point $(0, 1)$ represents a perfect model with an optimal performance and with an area of 1.0. On the other hand, if the model has a curve that bows toward the point $(1, 0)$ and has an area inferior to 0.50 the model shows an inferior performance. However, the binary output can be reversed. Thus reversing the performance values from the model. For instance an area inferior to 0.50, becomes higher than the random baseline.

As a result, in the following figures, the graph represented as (a) presents the curves for the three datasets. First, a dataset with only traditional object-oriented metrics (Metrics-only). Second, a dataset with only design and implementation smells. And, third, a dataset with the combination of traditional
object-oriented metrics with design and implementation smells. Additionally, we include, for each
dataset, the area under the curve (AUC), described in the lower right box inside of the graph.

The bar plot represented as (b) summarizes the difference between areas from the different datasets.
Therefore, it provides a perspective of the distances between the different combinations of two datasets.
For example, for the plot (b) in Figure 4.1, there is a difference of −0.17 between the metrics-and-smells
and the metrics-only. Therefore, the dataset with only metrics has a better performance than the datasets
with the combination of both metrics and smells. Additionally, it shows that has the same distance in
performances between the models trained with only smells and metrics.

**Architecture 1: Two sigmoid function hidden layers architecture**  The results for the first architec-
ture evaluated with the receiver operating characteristic methodology are illustrated in Figure 4.1. The
model trained with the metrics-only reveals the best accuracy compared against the smells-only and
metrics-and-smells model with a difference in area of 0.17. Therefore, the model trained with a smells
dataset has an area under the curve of 0.683.

![ROC curves and areas under the curve](image1)

**Figure 4.1:** Applying ROC methodology to compare the ROC curve and the distance between areas for the three
different datasets — metrics-only, smells-only and metrics-and-smells — for the first autoencoder ar-
chitecture

**Architecture 2: Two ReLU function hidden layers architecture**  The results for the second archi-
tecture evaluated with the receiver operating characteristic methodology are illustrated in Figure 4.2.

![Distances of areas under the curve between datasets](image2)
The model trained with the metrics-and-smells dataset reveals the best performance when inverting the results — $1 - 0.4025 = 0.595$ — followed by the metrics-only and the smells-only. Therefore, with this change, the model trained with the metrics and smells dataset shows a difference of 0.015 and, with the smells-only model, a difference of 0.093.

![ROC curves and areas under the curve](image1)

**Figure 4.2:** Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the second autoencoder architecture

**Architecture 3: Two tanh function hidden layers architecture** The results for the third architecture evaluated with the receiver operating characteristic methodology are illustrated in Figure 4.3. The model trained with the metrics-and-smells reveals the best performance with an area under the curve of 0.623, followed by the metrics-only, with a difference of 0.01. And, then, followed by the smells-only model with a difference of 0.11.

**Architecture 4: Four ReLU function hidden layers architecture** The results for the fourth architecture evaluated with the receiver operating characteristic methodology are illustrated in Figure 4.4. The model trained with the metrics-only dataset reveals the best performance, with an area under the curve of 0.709. It is, then, followed by the metrics-and-smells dataset, with a difference of 0.16. And, followed by the smells-only model with a difference of 0.19.
**Figure 4.3:** Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the third autoencoder architecture

**Figure 4.4:** Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fourth autoencoder architecture
Architecture 5: Six ReLU function hidden layers architecture  The results for the fifth architecture evaluated with the receiver operating characteristic methodology are illustrated in Figure 4.5. The model trained with the metrics-only dataset reveals the best performance with an area under the curve of 0.702, followed by the metrics-and-smells dataset, with a difference of 0.07, followed by the smells-only model, with a difference of 0.19.

![ROC curves and areas under the curve](image)

![Distances of areas under the curve between datasets](image)

Figure 4.5: Applying ROC methodology to compare the ROC curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fifth autoencoder architecture

### 4.1.2 Precision-Recall-Gain Methodology

The Precision-Recall-Gain curve (PRG Curve) can be interpreted as follows. The baseline curve is a straight line defined as \( y = -x + 1 \) that goes from the point \((0,1)\) to the point \((1,0)\). This curve represents the performance of a always-positive classification model with an area under the curve of 0.50. Therefore, a model with an area superior to 0.50 with its curve bowing toward the point \((1, 1)\) shows a greater accuracy than the always-positive model. Hence, a model closer to the point \((1, 1)\) represents a perfect model with an optimal performance and with an area of 1.0. On the other hand, if the model has a curve that bows toward the point \((1, 0)\) and has an area inferior to 0.50 the model shows an inferior performance. There is a possibility of appearing negative areas under the curve. The authors of this methodology do not provide much information about this problem, however they claim that those models perform worse than random.
As a result, in the following figures, the graph represented as (a) presents the curves for the three datasets. First, a dataset with only traditional object-oriented metrics (Metrics-only). Second, a dataset with only design and implementation smells. And, third, a dataset with the combination of traditional object-oriented metrics with design and implementation smells. Additionally, we include, for each dataset, the area under the curve (AUC), described in the lower right box inside of the graph.

The bar plot represented as (b) has the same configuration as in the Receiver operation characteristic methodology. Therefore, it summarizes the difference between areas from the models trained with three different datasets.

**Architecture 1: Two sigmoid function hidden layers architecture**  The results for the first architecture evaluated with the precision-recall-gain methodology are illustrated in Figure 4.6. The model trained with the metrics-only dataset revealed the best skill. Thus, it showed an area of 0.972. Then, it was followed by the metrics-and-smells, with a difference of 0.02. And, the smells-only model obtained an area that was smaller than zero. Therefore, we assumed that the model trained with only smells performed worse than with the other datasets.

![Figure 4.6](image.png)

(a) PRG curves and areas under the curve  
(b) Distances of areas under the curve between datasets

**Figure 4.6:** Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the first autoencoder architecture
Architecture 2: Two ReLU function hidden layers architecture The results for the second architecture evaluated with the precision-recall-gain methodology are illustrated in Figure 4.7. The model trained with the metrics-only dataset revealed the best skill. Thus, it showed an area of 0.97. Then, it was followed by the metrics-and-smells, with a difference of 0.33. And, the smells-only model obtained an area that was smaller than zero. Therefore, we assumed that the model trained with only smells performed worse than with the other datasets.

![Figure 4.7](image.png)

(a) PRG curves and areas under the curve  
(b) Distances of areas under the curve between datasets

**Figure 4.7:** Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the second autoencoder architecture

Architecture 3: Two tanh function hidden layers architecture The results for the third architecture evaluated with the precision-recall-gain methodology are illustrated in Figure 4.8. The model trained with the metrics-only dataset revealed the best skill. Thus, it showed an area of 0.971. Then, it was followed by the metrics-and-smells, with a difference of 0.07. And, the smells-only model obtained an area that was smaller than zero. Therefore, we assumed that the model trained with only smells performed worse than with the other datasets.

Architecture 4: Four ReLU function hidden layers architecture The results for the fourth architecture evaluated with the precision-recall-gain methodology are illustrated in Figure 4.9. The model trained with the metrics-only dataset revealed the best skill. Thus, it showed an area of 0.972. Then, it was followed by the metrics-and-smells, with a difference of 0.16. And, the smells-only model obtained an area
that was smaller than zero. Therefore, we assumed that the model trained with only smells performed worse than with the other datasets.

**Architecture 5: Six ReLU function hidden layers architecture** The results for the fifth architecture evaluated with the precision-recall-gain methodology are illustrated in Figure 4.10. The model trained with the metrics-only dataset revealed the best skill. Thus, it showed an area of 0.97. Then, it was followed by the metrics-and-smells, with a difference of 0.931. And, the smells-only model obtained an area that was smaller than zero. Therefore, we assumed that the model trained with only smells performed worse than with the other datasets.
Figure 4.9: Applying PRG methodology to compare the PRG curve and the distance between areas for the three different datasets — metrics-only, smells-only and metrics-and-smells — for the fourth autoencoder architecture.
4.2 Overall analysis

In this section, we make an overall comparison of the areas for all the architectures. Thus, we present two bar plots for each methodology. The first plot compares the area under the curve for each dataset between architectures. Hence, providing information related to the order of performance of the autoencoders for each dataset. Additionally, the second plot compares the absolute distances between datasets for the architectures. Thus, providing a perspective on how far apart is each dataset comparison between architectures.

**Receiver Operating Characteristic (ROC) Curve** In figure 4.11 we present the two plots showing the application of the receiver operating characteristic methodology. From their analysis, we can observe that all models trained with the smells-only dataset have an area under the curve very close to 0.50. Thus, they can be described as models with the worse performance. Nevertheless, the metrics-only model showed the highest AUC, with values ranging from 0.58 to 0.71. There was an exception, to the model trained with the third architecture (two tanh hidden layers architecture), where the metrics-and-smells revealed a small improvement over the metrics-only model of 0.01. The metrics-and-smells showed slightly smaller areas compared with the metrics-only model, with the highest difference being 0.18 for the second architecture (two ReLU function hidden layer). As such, the metrics-only autoencoder showed the highest skill, while the metrics-and-smells performed as the third best. We can also note that the areas of the smells-only and the metrics-only models have a similar distribution, as the second model is the most skilled compared against the fourth model, which is the least.

**Precision-Recall-Gain (PRG) Curve** In figure 4.12 we present the two plots showing the application of the receiver operating characteristic methodology. From their analysis, we can observe that the area of the smells-only models is negative which, based on the authors of the PRG methodology, makes the model performance worse than random classifiers. We can also describe that the metrics-only models showed the best skills with all models having 0.97 of area. Thus, meaning that the architecture does not have a significant relevance on the model's performance. However, the metrics-and-smells model showed the worst skill in the fourth architecture, with an area of 0.64 compared against the best skilled architecture, of 0.95. Thus showing a difference of 0.33. On the other hand, the most skilled metrics-and-smells model has a difference from the metrics-only of 0.02.
(a) Areas under the curve between the models trained with the five architectures using the three datasets

(b) Differences in areas under the curve between the three datasets for the five architectures

**Figure 4.11:** Comparing the ROC areas under the curve from all the models trained with five different architectures using the three datasets — smells-only, metrics-only and metrics-and-smells

(a) Areas under the curve between the models trained with the five architectures using the three datasets

(b) Distances of areas under the curve between the three datasets for the five architectures

**Figure 4.12:** Comparing the PRG areas under the curve from all the models trained with five different architectures using the three datasets — smells-only, metrics-only and metrics-and-smells

61
Discussion

"I have not failed. I've just found 10,000 ways that won't work."

Thomas A. Edison,

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Results interpretation</td>
<td>63</td>
</tr>
<tr>
<td>5.2 Answer to research questions</td>
<td>64</td>
</tr>
<tr>
<td>5.3 Results discussion</td>
<td>65</td>
</tr>
<tr>
<td>5.4 Contributions</td>
<td>67</td>
</tr>
<tr>
<td>5.5 Future Work</td>
<td>68</td>
</tr>
</tbody>
</table>
The results described in the previous section appear to be non-promising, based on the goal we defined for this study. As such, in this section, we will analyze and interpret the obtained results and, then, we will provide arguments to justify such results. Therefore, we will provide reasons to why certain obstacles appeared during the implementation and evaluation procedures of the study. Furthermore, considering the analyzed results, we will answer the research questions and we will discuss the contributions introduced by our research. Lastly, we will present possible routes for future work.

With this in mind, we outlined this chapter as follows. Section 5.1 provides interpretations for the results, considering the context of our study. Section 5.2 answers the research question based on the retrieved results. Section 5.3 discusses the results and provides arguments for its outcomes. Section 5.4 discusses the contributions given by this study. And, Section 5.5 introduces possible routes for future work.

5.1 Results interpretation

We observed that the area under the curve evaluated using the receiver operating characteristic methodology, for the models trained with the smells dataset averaged to 0.51, with the curve coinciding with the bounds of the baseline (i.e. the diagonal). This shows that the predictive accuracy of all the models trained with a smell-only dataset is the same as the performance of a random classification model. Therefore, it shows that these models have the worst performance. In the end, these could suggest implications, such as the use of design and implementation smells not having enough information to create models with an accurate prediction, or that there may not exist a correlation between the smells we detected and defects in the archives. Additionally, the repositories we used may have also been too mature to consider its smells. In other words, the detected smells in a version may point to a high probability of defects that may have already been fixed on a previous version.

Furthermore, all the models trained with the metrics-only dataset showed a higher accuracy prediction compared against the correspondent models trained with the smells dataset. While the area for the receiver operating characteristic for the smells-only models ranged from 0.50 to 0.52, the metrics-only models had a minimum area of 0.58 and a maximum area, for the deeper models, of 0.71. This implies that we can extract better information for defect prediction from traditional object-oriented metrics than with design and implementation smells.
Additionally, when considering models trained with the combination of both metrics and smells, the predictive accuracy was better than models trained with only smells. However, generally worst compared against models trained with only metrics. In particular, we analyzed that the models trained with the metrics dataset, except for the two-tanh-hidden-layers architecture whose area is worse by 0.01, outperformed the metrics-and-smells models. This means that using smells does not provide extra skillfulness to the models when used combined with traditional object-oriented metrics. In fact, making the performance worse. On the other hand, combining traditional object-oriented metrics on top of smells, improved the predictive accuracy of the models, except for the two-ReLU-hidden-layers architecture whose area became worse with a difference of 0.10.

Considering the precision-recall-gain evaluation, the metrics-only models showed an overall, nearly perfect performance, averaging an area of 0.97. Additionally, compared against the models trained with the combination of metrics and smells, when including the smells as features, it lead to a decrease in area for all the models, most significantly on the model with the two-ReLU-hidden-layers architecture, with a difference in area of 0.33.

The results produced by the precision-recall-gain evaluation for the smells-only models were negative values. These may have resulted from incorrect calculations because of the predicted low values, which lead to low reconstruction errors. For instance, after analyzing the reconstruction errors, we identified values that were very close to zero, going as far as ten to the power of minus ten ($10^{-10}$). To calculate the curve and areas, we used the library provided by the authors of the methodology. Therefore, the negative areas may have been the product of calculation errors in the program because of the significantly low reconstruction loss values. However, the authors of this evaluation metric associated the negative values to models with worst performances than a random model. Thus the model trained with only smells could have a worse performance than a random model.

## 5.2 Answer to research questions

**Research Question 1**: Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with design and implementation smells or with traditional object-oriented features?

After analyzing the different models, we concluded that a model with traditional object-oriented metrics outperformed a model trained with only design and implementation smells under an extremely imbalanced scenario. Looking into the receiver operating characteristic curve results, the smells-only model showed results near the baseline, with an average area of 0.51 with a standard deviation of
0.0071, revealing the equivalent performance of a random classifier. On the other hand, the model trained with only traditional object-oriented metrics, had an average area of 0.656 with a standard deviation of 0.058, which is still close to the baseline value but with a better skill than the smells-only model.

**Research Question 2:** Does training an autoencoder as a defect prediction model with an imbalanced dataset produces a better performance with the combination of both design and implementation smells, and with traditional object-oriented features or when it is trained with only one single of those features?

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

**Research Question 3:** When training a smell-based defect prediction model with an autoencoder algorithm, what category of architecture provides the best model performance?

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

### 5.3 Results discussion

Next, we will provide arguments for the less promising results we obtained. Thus, we will discuss possible reasons why the performance of models trained with only smells was near the skill of a random classifier and why models trained with traditional object-oriented-metrics and the combination of all features performed better than the smells-only models.
First, while inspecting the code, we noticed that for all the projects, most fixes did not apply code refactoring to clean the existing smells. This means that, at a certain point, a particular model detected a set of smells and correctly predicted its defects, and fixed them, while leaving the smells uncleaned. This could have produced a model that kept predicting defects that may already have been detected and fixed. To deal with this, we could have considered training our model with other features, related to the number of times the project had previously made a fix, or related to how many changes a certain module had suffered.

Second, as we used Designite to extract smells from each module, it constrained us to the use of a limited set of design and implementation smells. Therefore, from the structural class of design smells, we only considered seventeen from twenty-five smells listed in [12]. There are also implementation smells that we did not consider as features for our prediction models [89]. We also did not consider architecture smells [90], test smells, nor performance smells [91]. With this, by having narrowed down the categories of smells to a smaller domain, the models may have not gotten the whole necessary picture for the model to predict defects with accuracy.

Third, we applied a coarser-grained approach for training, where we created models using the detected smells equally and extracted the smells from a large range of different repositories. Therefore, we could have followed a finer-grained approach and categorized the different projects based on the style of development of the programmer or the team, or on the type of project. And, then, trained particular models for each category. We could also have determined which types of smells are most relevant in predicting defects for each type of repository. Thus, having set higher weights to those smells during training. Combining the previous suggestions, another alternative would have been to focus only on each single repository and develop a custom-made model that has a specific architecture and whose smells have different weights based on how relevant each one is to the defect prediction.

Fourth, while further analyzing the repositories, we discovered that the frameworks that provided the repositories and the defect information from each version had faults in them. Therefore these could have suggested defects in modules that were not defective. Additionally, this may have lead to poor training and inaccurate evaluation of the models’ skill. Furthermore, if the models trained correctly and predicted accurately, they might have also identified real faults in the repositories that the frameworks did not recognize.

Finally, while training the autoencoders, we were not able to test all possible combinations of parameters to achieve the best level of training because of the lack of resources. With more time and a better
5.4 Contributions

In this section, we present the main contributions introduced by our thesis. We will discuss how does our thesis define a starting point for research on smell-based defect prediction with deep learning and how it sets an initial baseline for future defect prediction models.

In this thesis, we made a broader observation of the application of bad code smells in the subject of defect prediction. Although the results we acquired appear to be less promising than we expected, there are still several routes that we can explore. Since, our goal was to study the overall performance when using a small subset of smells over a large cluster of repositories and, from the results, it is clear that a more general approach is not enough to understand the patterns between smells and defects. Thus, not enough to train models that can accurately make precision for possible bugs in projects. From here, there are several paths that can be taken to further explore the relation between smell detection and defect prediction. For instance, we can do a custom selection of smells for a particular project. Or, we could refined the autoencoder architectures.

Moreover, we were able to draw conclusions on the best type of autoencoder algorithm to build a model focused on extremely imbalanced smell-based defect prediction. Therefore, we inferred that a model train with an deep layer autoencoder architecture showed a better performance. In particular, when trained with a four hidden layers with ReLU activation functions. However, the difference in results between all architectures was not particularly significant. Because there was only a small difference in performance between all models when trained with different layered architectures.

From the evaluation results we got, we can define a defect prediction starting baseline. When evaluating future models that are trained with smells under extreme imbalanced datasets. Thus, the performance results we get from both the Receiver Operating Characteristic and the Precision-Recall-Gain can be used as the initial baseline for performance comparison. In the end, we are providing an evaluation guideline for future smell-based defect prediction models and setting the starting base for a ranking system for models created for this category.
5.5 Future Work

We will introduce a set of paths and ideas that can be researched for future work. We will start by discussing the proceeding steps in the smell-based defect prediction research, then we will discuss the practical applications of defect prediction to test case prioritization.

We applied a coarser-grained approach to find defects using smells. Further on, we could apply a finer-grained approach. For a specific category of software projects or only a specific project, we could identify which smells have a higher relevance in predicting the defective modules. We could attribute different weights to smells that are more suggestive of defects, based on the project or the programmer. We could also lower the level of granularity of a module, from class-level to method-level. This would allow us to locate the defects in the software with a much higher accuracy. Furthermore, instead of only identifying whether a module is defective, we could calculate the probability of a module being defective.

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

In regression testing, one of the main issues is the increasing number of software tests that need to be executed for each new version added to the project. This leads to a testing phase, for each new introduced version of the software, that takes a significant amount of time. So, a set of solutions were proposed to fix these issues. One of the most promising methods was test case prioritization, where the test cases are reordered giving higher precedence to the test cases that most likely detect defective modules. In the end, test case prioritization would benefit from smell-based defect prediction by assigning higher priorities to the test cases that cover modules with higher percentages of predicted defects.
6

Conclusion

“Wisdom is knowing what to do next; virtue is doing it.”

David Starr Jordan,
In general, this study is motivated by the need to decrease the time resources spent on the testing phase of the software development life cycle. Since, being able to achieve fast-delivered high quality software is one of the expected requirements for highly successful software products. Therefore, we considered defect prediction, a relevant research topic that focuses on this challenge. In particular, a defect prediction approach based on bad code smells because extracting smells from software archives is inexpensive and fast, and they provide relevant information to identify defects. With this in mind, there is also a knowledge and research gap regarding the application of smell-based defect prediction in the context of extreme data imbalance, which can be an occurring scenario in features extracted for defect prediction. Therefore, our study focuses on introducing a new model for smell-based defect prediction that is trained with an extremely imbalanced dataset.

From this perspective, the goal of our study is to propose a new defect prediction model, trained with design and implementation smells, considering an extremely imbalanced data. Thus, we decided to train the model using an autoencoder algorithm and evaluate the performance under a different range of autoencoder architectures. So, we compared five models trained with design and implementation smells with different architectures: three shallow models and two deeper models. To evaluate the effectiveness of the smell-based approach with an autoencoder, we compared each of the five models with two other models; one trained with traditional object-oriented metrics and the other trained with the combination of both metrics and smells. In particular, we used the receiver operating characteristic curve an the precision-recall-gain curve to measure the models performance.

In the end, we did not achieve promising results. Since, the models trained with traditional object-oriented metrics outperformed the models trained with design and implementation smells. Moreover, the models trained with smells showed equivalent performances to the random classification models and combining object-oriented metrics to the design and implementation smells improved the performance of the models. However, without the smells the models showed a greater performances. Furthermore, we also observed that the autoencoders with a deeper architectures had a higher accuracy than the shallow architectures. But, the observed distances between measures were not significant to provide relevance to the architecture decision.

As such, this study presented an introductory approach to solve the challenge of smell-based defect prediction trained in an extremely imbalanced data scenario. And, despite this approach not showing promising results, it is a starting point for future research in this problem-field. Moreover, we contributed with a baseline for future models trained in this scenario. Thus, it can be used to evaluate new approaches.
Accordingly, the application of a successful approach to the extremely imbalanced dataset for smell-based defect prediction, would bring the utmost benefit to other works in the literature. For example, test case prioritization, which has been a target of considerable research to reduce the amount of time taken to detect regressions in software [92]. Henceforth, it would improve the efficiency and effectiveness of test case prioritization by using an effective smell-based defect prediction model to be able to assign priorities to test cases that cover a higher percentage of defective modules.

For future work, we propose the application of a finer-grained approach to smell-based defect prediction, such as the application of a specific combination of smells and datasets. Moreover, we could also implement an intensity score [3] for the smells. So that, we could give priority to the most relevant smells. Additionally, we propose a change in paradigm, where we could train each model based on the programmer instead of being based on the project. Because, each developer has an unique style of development. Thus an unique pattern of bad code smells that could point toward defects in the code.
Bibliography


78


Defect Prediction Tables
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction Principle</td>
<td>Reduce duplication of information in a program</td>
</tr>
<tr>
<td>Imperative Abstraction</td>
<td>Abstraction does not support complementary or interrelated methods completely</td>
</tr>
<tr>
<td>Multifaceted Abstraction</td>
<td>Abstraction has more than one responsibility assigned to it</td>
</tr>
<tr>
<td>Unnecessary Abstraction</td>
<td>Abstraction that is not needed and it is introduced to the software design</td>
</tr>
<tr>
<td>Unutilized Abstraction</td>
<td>Abstraction is left unused (either not directly used or not reachable)</td>
</tr>
<tr>
<td>Encapsulation Principle</td>
<td>Bundling of data with the methods that operate on that data</td>
</tr>
<tr>
<td>Deficient Encapsulation</td>
<td>One or more members of an abstraction is more permissive than actually required</td>
</tr>
<tr>
<td>Unexploited Encapsulation</td>
<td>Client code uses explicit type checks instead of exploiting the types encapsulation</td>
</tr>
<tr>
<td>Modularization Principle</td>
<td>Systems should be built from cohesive, loosely coupled components (modules)</td>
</tr>
<tr>
<td>Broken Modularization</td>
<td>Data that should be in a single abstraction is dispersed across multiple abstractions</td>
</tr>
<tr>
<td>Insufficient Modularization</td>
<td>Abstraction that has not been completely decomposed</td>
</tr>
<tr>
<td>Cyclic-Dependent Modularization</td>
<td>Two or more abstractions are tightly coupled</td>
</tr>
<tr>
<td>Hub-like Modularization</td>
<td>Significant number of dependencies from one-to-many abstractions</td>
</tr>
<tr>
<td>Hierarchy Principle</td>
<td>Ranking or ordering of abstractions</td>
</tr>
<tr>
<td>Missing Hierarchy</td>
<td>Code segment uses conditional logic to explicitly manage variation</td>
</tr>
<tr>
<td>Wide Hierarchy</td>
<td>Hierarchy may be overly breadth</td>
</tr>
<tr>
<td>Deep Hierarchy</td>
<td>Inheritance hierarchy is unreasonably deep</td>
</tr>
<tr>
<td>Rebellious Hierarchy</td>
<td>Methods provided by a type are rejected by its subtype</td>
</tr>
<tr>
<td>Broken Hierarchy</td>
<td>Supertype and a subtype conceptually lacks an “IS-A” relationship</td>
</tr>
<tr>
<td>Multipath Hierarchy</td>
<td>Both direct and indirect inheritance between supertype and subtype,</td>
</tr>
<tr>
<td>Cyclic Hierarchy</td>
<td>Supertype depends on one or more of its subtypes</td>
</tr>
</tbody>
</table>

**Table A.1:** Name and description of the design smells used in this study organized by principle.

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

**Table A.2:** Name and description of the implementation smells used in this study.
Literature’s Appendix
<table>
<thead>
<tr>
<th>Website</th>
<th>Description</th>
<th>Number of Metric used</th>
<th>Number of Projects</th>
<th>Granularity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table B.1: Summary of the publicly available datasets for software defect prediction.
<table>
<thead>
<tr>
<th>Study</th>
<th>Category</th>
<th>Technique</th>
<th>Dataset</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>supervised</td>
<td></td>
<td>dictionary learning, cost-sensitive learning</td>
<td>NASA</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>collaborative representation</td>
<td>NASA</td>
<td>2014</td>
</tr>
<tr>
<td></td>
<td></td>
<td>multiple kernel ensemble learning, boosting</td>
<td>NASA</td>
<td>2016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deep learning</td>
<td>PROMISE</td>
<td>2016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>deep learning</td>
<td>JIT</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>transfer learning, boosting</td>
<td>PROMISE</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>transfer learning, GA, AdaBoost</td>
<td>PROMISE</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>multiobjective optimisation, GA</td>
<td>PROMISE</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>boosting, cost-sensitive learning, transfer learning</td>
<td>PROMISE</td>
<td>2017</td>
</tr>
<tr>
<td></td>
<td></td>
<td>decision tree, bagging, random forest</td>
<td>JIT</td>
<td>2017</td>
</tr>
<tr>
<td></td>
<td>semi-supervised</td>
<td>SemiBoost, graph learning, sparse representation</td>
<td>NASA</td>
<td>2016</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sparse representation, sparse graph, label propagation</td>
<td>NASSA</td>
<td>2017</td>
</tr>
<tr>
<td></td>
<td>unsupervised</td>
<td>sparse representation, cluster, feature selection</td>
<td>ReLink</td>
<td>2015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>spectral clustering</td>
<td>NASA, AEEEM, PROMISE</td>
<td>2016</td>
</tr>
</tbody>
</table>
**Figure B.1:** Commonly used performance evaluation measures [1]

<table>
<thead>
<tr>
<th>Measure</th>
<th>Defined as</th>
</tr>
</thead>
</table>
| Pd/recall/TP rate | \[
\frac{TP}{TP + FN}
\]                                                   |
| Pf/FP rate      | \[
\frac{FP}{FP + TN}
\]                                                   |
| precision       | \[
\frac{TP}{TP + FP}
\]                                                   |
| F-measure       | \[
\frac{2 \times Pd \times \text{precision}}{Pd + \text{precision}} = \frac{2 \times Pd \times \text{precision}}{2 \times Pd \times \text{precision} + FP + FN}
\] |
| G-measure       | \[
\frac{2 \times Pd \times (1 - Pf)}{Pd + (1 - Pf)}
\]                                                   |
| balance         | \[
1 - \frac{\sqrt{(0 - Pf)^2 + (1 - Pd)^2}}{\sqrt{2}}
\]                                                   |
| accuracy        | \[
\frac{TP + TN}{TP + FP + FN + TN}
\]                                                   |
| G-mean          | \[
\sqrt{Pd \times (1 - Pf)}
\]                                                   |
| MCC             | \[
\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP) \times (TP + FN) \times (TN + FP) \times (TN + FN)}}
\] |
| AUC             | the area under the ROC curve                                              |
| \( P_{opt} \)   | the area under effort-based cumulative lift charts, which compares a predicted model with an optimal model |
| AUCEC           | the area under cost-effectiveness curve                                   |
Extracted Metrics’ Preview
<table>
<thead>
<tr>
<th>Project Name</th>
<th>Package Name</th>
<th>Type Name</th>
<th>FANIN</th>
<th>FANOUT</th>
<th>NCM</th>
<th>LOC</th>
<th>NCOM</th>
<th>NOFP</th>
<th>NOPM</th>
<th>WMG</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>javax-collections-api</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>log4j-api</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>log4j-core</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>log4j-extras</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>log4j-util</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>log4j-xml</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.lang</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.lang3</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.math3</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.math4</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.parallel</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.proxy</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.regex</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.reflect</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.lang</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.lang3</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.math3</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.math4</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.parallel</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.proxy</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.regex</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.commons.reflect</td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Name</td>
<td>Package Name</td>
<td>Type Name</td>
<td>Code Smell</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------</td>
<td>----------------------</td>
<td>--------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>DotMojo</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>DotMojo</td>
<td>Deficient Encapsulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>IsolatedThreadGroup</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>DotMojo</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>DotMojo</td>
<td>Deficient Encapsulation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>IsolatedThreadGroup</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>ApiMethodAlias</td>
<td>Unutilized Abstraction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table C.2:** Preview of design smells detected from project camel version 100 using Designt

<table>
<thead>
<tr>
<th>Project Name</th>
<th>Package Name</th>
<th>Type Name</th>
<th>Method Name</th>
<th>Code Smell</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>EmbeddedMojo</td>
<td>createArguments</td>
<td>Magic Number</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>DotMojo</td>
<td>execute</td>
<td>Complex Method</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>execute</td>
<td>Complex Method</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>execute</td>
<td>Long Identifier</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>execute</td>
<td>Long Method</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>joinThread</td>
<td>Long Statement</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>terminateThreads</td>
<td>Complex Method</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>terminateThreads</td>
<td>Long Statement</td>
</tr>
<tr>
<td>main</td>
<td>org.apache.camel.maven</td>
<td>RunMojo</td>
<td>terminateThreads</td>
<td>Long Statement</td>
</tr>
</tbody>
</table>

**Table C.3:** Preview of implementation smells detected from project camel version 100 using Designt
Listings of Code Snippets

```python
1 def scale_data(self, dataset):
2     dataset['loc'] = StandardScaler().fit_transform(dataset['loc'].values.reshape(-1,1))
3     dataset['nof'] = StandardScaler().fit_transform(dataset['nof'].values.reshape(-1,1))
4     dataset['nopf'] = StandardScaler().fit_transform(dataset['nopf'].values.reshape(-1,1))
5     return dataset

Listing D.1: Code sample for standard scaling applied to three metrics

```python
1     split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
2     for train_index, test_index in split.split(dataset, dataset['target']):
3         strat_train_set = dataset.iloc[train_index]
4         strat_test_set = dataset.iloc[test_index]

Listing D.2: Code sample for stratification
```
Listing D.3: Removal of defective instances

```python
def train_autoencoder_1(self, input_dim, input_layer):
    encoding_dim = int(input_dim/2)
    hidden_dim = int(encoding_dim/2)
    input_layer = Input(shape=(input_dim, ))
    encoder = Dense(encoding_dim, activation='relu')(input_layer)
    hidden_layer = Dense(hidden_dim, activation='relu')(encoder)
    decoder = Dense(input_dim, activation='relu')(hidden_layer)
    return decoder
```

Listing D.4: Architecture implementation of a two hidden layer architecture with ReLU activation function

```python
def train_autoencoder_4(self, input_dim, input_layer):
    encoding_dim = int(input_dim/2)
    hidden_dim = int(encoding_dim/2)
    mid_dim = int(hidden_dim/2)
    input_layer = Input(shape=(input_dim, ))
    encoder = Dense(encoding_dim, activation="relu")(input_layer)
    hidden_layer = Dense(hidden_dim, activation="relu")(encoder)
    hidden_layer = Dense(mid_dim, activation="relu")(hidden_layer)
    hidden_layer = Dense(hidden_dim, activation="relu")(hidden_layer)
    decoder = Dense(input_dim, activation='relu')(hidden_layer)
    return decoder
```

Listing D.5: Architecture implementation of a four hidden layer architecture

```python
def train_autoencoder_4(self, input_dim, input_layer):
    encoding_dim = int(input_dim/2)
```
hidden_dim = int(encoding_dim/2)
mid_dim = int(hidden_dim/2)
unitary_dim = 1

encoder = Dense(encoding_dim, activation="relu")(input_layer)
hidden_layer = Dense(hidden_dim, activation="relu")(encoder)
hidden_layer = Dense(mid_dim, activation="relu")(hidden_layer)
hidden_layer = Dense(unitary_dim, activation="relu")(hidden_layer)
hidden_layer = Dense(mid_dim, activation="relu")(hidden_layer)
hidden_layer = Dense(hidden_dim, activation="relu")(hidden_layer)

Listing D.6: Architecture implementation of a six hidden layer architecture

def get_reconstruction_error(self, model):
test_x_predicted = model.predict(self._test_x)
mse = np.mean(np.power(self._test_x - test_x_predicted, 2), axis = 1)
error_df = pd.DataFrame({'Reconstruction_error':mse, 'True_values': self._test_y['target']})
return error_df

Listing D.7: Making prediction with testing dataset features, calculating reconstruction error and return error and testing output

def get_roc_and_auc(self, error_df):
false_pos_rate, true_pos_rate, thresholds = roc_curve(error_df.True_values,
error_df.Reconstruction_error)
i = np.arange(len(true_pos_rate))
roc_df = pd.DataFrame({'FPR': pd.Series(false_pos_rate, index=i),
'TPR': pd.Series(true_pos_rate, index=i),
'Threshold': pd.Series(thresholds, index=i)})
roc_auc = roc_auc_score(error_df.True_values, error_df.Reconstruction_error)
return roc_df, roc_auc

Listing D.8: Generating ROC Curve and calculating the area under the curve
```python
def get_prg_and_auc_df(self, error_df):
    prg_curve = prg.create_prg_curve(error_df.True_values, error_df.Reconstruction_error)
    prg_curve_df = pd.DataFrame.from_dict(prg_curve)
    i = np.arange(1)
    prg_auc = prg.calc_auc(prg_curve)
    prg_auc_df = pd.DataFrame({'AUC': pd.Series(prg_auc, index=i)})
    return prg_curve_df, prg_auc_df
```

**Listing D.9:** Generating PRG Curve and calculating the area under the curve