Universidade FUMEC
Faculdade de Ciências Empresariais
Programa de Pós-Graduação em Sistemas de Informação e Gestão do Conhecimento

Identifying Code Smells with Machine Learning Techniques

Frederico Caram Luiz

Belo Horizonte
2018

Frederico Caram Luiz

Identifying Code Smells with Machine Learning Techniques

MSc thesis presented to the Programa de Pós-Graduação em Sistemas de Informação e Gestão do Conhecimento of FUMEC University, as partial fulfillment of the requirements for the Master's degree in Information Systems and Knowledge Management. Research track: Technology and Information Systems.

Supervisor: Prof. Dr. Fernando Silva Parreiras

Belo Horizonte

2018

Dados Internacionais de Catalogação na Publicação (CIP)

L953i Luiz, Frederico Caram, 1985 -

Identifying Code Smells with Machine Learning Techniques / Frederico Caram Luiz. – Belo Horizonte, 2018.

101 f. : il. ; 29,7 cm

Orientador: Fernando Silva Parreiras

Dissertação (Mestrado em Sistemas de Informação e Gestão do Conhecimento), Universidade FUMEC, Faculdade de Ciências Empresariais, Belo Horizonte, 2018.

1. Aprendizado do computador. 2. Identificação - Brasil. 3. Computação - Brasil. I. Título. II. Parreiras, Fernando Silva. III. Universidade FUMEC, Faculdade de Ciências Empresariais.

CDU: 681.3

Ficha catalográfica elaborada pela Biblioteca Universitária-FUMEC

UNIVERSIDADE
FUMEC

Dissertação intitulada "**Identifying Code Smells with Machine Learning Techniques**" de autoria de Frederico Caram Luiz, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Fernando Silva Parreiras – Universidade FUMEC
(Orientador)

Prof. Dr. Luiz Cláudio Gomes Maia – Universidade FUMEC
(Examinador Interno)

Prof. Dr. Fernando Hadad Zaidan – IETEC
(Examinador Externo)

Moisés de Matos Botelho, Me. – PRODEMGE
(Consultor *Ad Hoc*)

Prof. Dr. Fernando Silva Parreiras
Coordenador do Programa de Pós-Graduação em Sistemas de Informação e Gestão do Conhecimento da Universidade FUMEC

Belo Horizonte, 27 de fevereiro de 2018.

Abstract

Context: Code smells are an accepted approach to identify design flaws in the source code. Many studies regarding their automatic identification were developed, ranging from hard threshold metrics based and rule based to machine learning techniques. But there is still a lack of empirical benchmarks to define when they should be used.

Objective: This study aims at the development of an mapping study of machine learning techniques and code smells found in literature and of an experiment based on the state-of-art machine learning techniques identified and applied in a standardized dataset that reflects a real project scenario in order to create a benchmark for future work.

Method: A mapping study was used to identify the techniques used for each smell and an empirical experiment based on the previously identified techniques.

Results: From the studied smells, the technique used for Long Method was the one that performed closer to the original experiments, but yet 34% worst than it, while most were outperformed by more than 50% and some performed even 86% below the original experiments. The imbalanced dataset techniques that were used, also performed worst than it, but still were able to bring improvement, ranging from 1% that was the case of Long Methods and Feature Envy to more than 100% in the case of Shotgun Surgery and Parallel Inheritance.

Conclusions: The replicated techniques results diverged from the original experiment. But techniques for imbalanced data were able improve the existing techniques under these circumstances. Ensemble models presented the best performance for relationships between methods and classes, while the Boosting techniques for the ones related to the structure of classes and methods. For future experiments, we suggest future works to further extend the database by adding other smells kind and techniques to it to create a broader benchmarking.

keyword: Code smells, identification, machine learning, positive-unlabeled learning.

Resumo

Contexto: Code smells são uma abordagem bem aceita para a identificação de problemas de design do código. Muitos estudos envolvendo sua identificação automática já foram desenvolvidos, abrangendo desde técnicas baseadas em limites rígidos até as baseadas em machine learning. Mas ainda faltam evidências empricas para definir quais são melhores para cada cenário.

Objetivo: Este estudo busca desenvolver um estudo de mapeamento de técnicas de machine learning para a identificação de code smells na literatura e um experimento baseado nas técnicas estado da arte identificadas a partir dele e aplicadas em um banco de dados padronizado, refletindo um cenário próximo ao real com o objetivo de criar uma referência para pesquisas futuras.

Método: Um estudo de mapeamento para identificação das técnicas utilizadas para code smell e um experimento empirico baseado nas técnincas identificadas préviamente.

Resultados: Dos code smell estudados a técnica used para Long Methods foi a que performou mais próxima ao experimento original, mas ainda assim 34% pior, enquanto a maioria das outras performou 50

Conclusões: Os resultados apresentaram variação em relação aos experimentos originais. Mas as técnicas para bases desbalanceadas conseguiram apresentar melhora em relação à elas. Os modelos de Ensemble apresentaram a melhor performance em code smells que envolvem o relacionamento entrem classes e métodos, enquanto as técnicas de Boosting foram melhores os mais estruturais. Para futuros experimentos, sugerimos estender o database para incluir mais tipos de code smells e testar novas técnicas para criar um benchmarking mais amplo.

keyword: Code smells, identification, machine learning, positive-unlabeled learning.

Acknowledgements

In first place I would like to thank my wife Desiree for her love and constant support and patience during this long period. I would also like to thank my parents and my sister, whose love and guidance are with me in whatever I pursue. They are the my role models. A special thanks goes to my supervisor Dr. Fernando Parreiras for all the guidance he provided, his patience, passion and enthusiasm. Finally a thanks for my fellow researchers from LAIS lab, in special thanks to Amadeu, Bruno and Daniel for the help in the literature review and in the results.

List of Figures

List of Tables

List of abbreviations and acronyms

ACDI	Alternative Classes with Different Interfaces
CI	Confidence Interval
COM	Comments
DAC	Data Clumps
DC	Data Class
DCP	Divergent Change
DUC	Duplicated Code
FD	Functional Decomposition
FE	Feature Envy
GC	God Class
II	Inappropriate Intimacy
IJSEKE	International Journal of Software Engineering and Knowledge Engineering
ILC	Incomplete Library Class
LAZ	Lazy Class
LC	Large Class
LM	Long Method
LPL	Long Parameter List
MC	Message Chains
ML	Machine Learning
MM	Middle Man
OOD	Object-Oriented Design
PIH	Parallel Inheritance Hierarchies
PO	Primitive Obsession

RB	Refused Bequest
RQ	Research Question
SG	Speculative Generality
SS	Shotgun Surgery
SW	Switch Statements
SC	Spaghetti Code
SLR	Systematic Literature Review
TF	Temporary Field

Contents

1 Introduction

It is estimated that, in 2016, there are 3.4 connected devices for each person (Global Web Index, 2016) and that the software industry moved aroung 3.8 billion dollars last year (Criteo, 2015). But part of this money is wasted due to software defects, it has been reported that above 75% of the total software cost is used for maintenance activities (Bennett and Rajlich, 2000; Liu et al., 2012). A factor that is important for that, is the quality of the written code, since it affects the readability of the code, and affects its maintainability (Aggarwal et al., 2002). In addition, it is shown that software maintainers spend around 60% of their time in understanding the code (Abran and Nguyenkim, 1993). But even in cautiously designed systems, the quality of the source code tends to degrade as the project evolves, since a system's original design is rarely prepared for every new requirement and the changes need to be made quickly by different people without properly adjusting the system's structure (Seng et al., 2006). The developers also tends to focus on the addition of the new functions and bug fixes rather than improving software maintainability (Tufano et al., 2015). Software engineers also overlook it when it is seemly complex and when it seems not to be critical to maintain the longevity of the software (Murphy-hill et al., 2012). This behavior leads to an increase in the software complexity and the stacking of bad quality code.

A common way to avoid this degradation is to identify and fix those flaws as they appear, one of the main theories to identify them in object-oriented design is the detection of code-smells (Mens and Tourwé, 2004). Code smells provide heuristics for the identification of design flaws in the source code that make software harder to evolve, comprehend and maintain. Each code smell examines a specific kind of system elements (class, methods, etc...) that can be evaluated by its characteristics (Olbrich et al., 2009). One downside it that it is error-prone and time-consuming (Murphy-hill et al., 2012), it is also up to the programmers interpretation (Fowler and Beck, 1999), and their definition is not a consensus among developers (Bryton and Abreu, 2009; Fontana et al., 2016). In order to reduce this subjectivity, automated approaches based on the source code have been presented (Fontana et al., 2012; Fokaefs et al., 2007; Mantyla et al., 2004; Rasool and Arshad, 2015), but a relevant part of those approaches are based on code metrics. (Bryton et al., 2010; Counsell et al., 2010; Marinescu, 2004; Moha and Guéhéneuc, 2010; Rasool and Arshad, 2015). These techniques uses metrics and thresholds that are not consistent among them, leading to an increasing number of false positives, not representing real problem (Fontana et al., 2016).Since it does not consider information related to the context, domain, size and design of the system (Ferme et al., 2013).

For these reason, considering the subjective definition of code smells, it is required

that the technique can be aware and sensible to the particular context. For this we propose a machine learning (ML) approach to help the identification of code-smells by the developers. The results of these techniques will be compared against state-of-art rules based techniques to compare the performance of these technique for different situations and code smells.

1.1 Problem

Code smells can be identified by either manual or automated analysis of the source code (Moha and Guéhéneuc, 2010). The manual recognition of code smells on the source code by developers is a error prone, costly and time-consuming activity, since it depends on the developer's degree of experience and perception (Counsell et al., 2010). There are also evidences showing that eradication of code smells is not being achieved to a satisfactory level because usually the developers are not even aware of their presence (Yamashita and Moonen, 2013). Another hardship in the code smells identification is that it is subject to the developers interpretation, what one developer considers a code smell may not be one from the point of view of another developer (Bryton and Abreu, 2009; Fontana et al., 2016).

In order to reduce the subjectivity in the code smell identification, attempts were done to identify useful metrics which can give hints on the existence of design flaws (Bryton and Abreu, 2009). Another approach that can help reducing this subjectivity is the use of automatic tools to identify flawed code, it is possible to find diverse tools and techniques that can be used to identify different kind of code smells. About all of those tools convert the source code to an intermediate representation, then uses static analysis based on rules, metrics and thresholds (Fernandes et al., 2016; Rasool and Arshad, 2015). However, this kind of approach does not consider information related to the context, domain, size and design of the system (Ferme et al., 2013). Sometimes what is seen as a smell could be the best way to actually implement or design a (part of a) program. For example, automatically generated parsers are often Spaghetti Code, large classes with long methods. Only quality analysts can evaluate their impact in this context (Khomh et al., 2009b). This lack of context lead to false-positives, more than 50% of the automatically detected code is not related to architectural problems (Macia et al., 2012). In order to reduce this uncertainty related to the context, other techniques were proposed such as the ones that uses graph or social analysis techniques to identify the system structure, that use statistical methods to identify metrics thresholds and the ones that uses ML to identify code smell patterns (Rasool and Arshad, 2015).

The later has been attracting growing attention of the academic and corporate communities (Wen et al., 2012). The main advantage of this kind of techniques is that

it can find patterns that are hard to define using predefined or statical rules, being able to find patterns where even humans cannot see one (Kotsiantis, 2007). But, despite of its advantages, as a topic that was only recently applied to the code smell identification problem, it is still limited to only a set of code-smells and still lacks empirical evidences to support it's usage.

Given this scenario, the following question arises **"What is the efficiency of ML techniques for code smells identification?"**.

1.2 Objectives

In the recent times with the growing interest in ML techniques, works were developed applying it for code smells identification. But no work was done to review and compare the work that has been done in order to identify what code smells and techniques were explored and which ones still lacks research and development.

In this context, this work objective is to assess the ML techniques for code smells identification performance when compared to traditional models. Since ML can provide new and performing ways of finding code-smells, with more flexibility than heuristics and metrics based approaches and is a field that has been receiving successive attention lately (Fontana et al., 2016). It can also help the software companies to reduce rework and improve quality and reliability, and also helping the software engineers to improve the productivity.

The specific objectives that decomposes the main objective are:

- **Obj. 1:** Identify code smells identification techniques that uses ML Techniques.
- **Obj. 2:** Define a baseline where future studies on the subject can compare their results against.

1.3 Motivation

Improving software maintainability and software quality can save money for both software development companies, teams and their clients (Sjoberg et al., 2013). Quality can also be a competitive advantage, which affects the company survivability in the long run. There is continuous work being done in order to improve software maintainability, but the ML branch is still under development and have plenty room for improvements when compared with static methods. There is also a lack of empirical data to support the development of new research regarding ML techniques for code smell identification (Fontana et al., 2016). Further developing empirical support can help understanding the machine

learning techniques flaws and take actions in order to enhance its performance, providing us with data about their benefits and gaps.

1.4 Adherence to FUMEC's Graduate Program in Information Systems and Knowledge Management

FUMEC's graduate program in Information Systems and Knowledge Management is focused on academic knowledge, scientific development and applied research on the Information Systems and Knowledge Management. The program is organized into two main streams: Technology and Information Systems and Information and Knowledge Management. Multidisciplinary approach is a key concept on FUMEC's graduate program.

This research proposes the application of ML techniques for code smell identification, in order to improve software reliability. ML can also be used for a handful of uses, that ranges from biology, medicine, finance, business, logistics, information retrieval and other applications, it is conceived to allow machines to mimic the human thinking process (Ekbia, 2010). By mimicking the human thinking process, ML enables computers help in the decision making process, fitting it into the track of application of statistical and computational models for decision making.

Code smells are considered a good indicator of design flaws and it is important for the improvement of software quality. This quality improvement is an important aspect of software engineering, which is frequently receiving attention of empirical studies. Given this scenario, the study proposal focus is under the Information Systems in compliance with FUMEC's graduate program. The multidisciplinary aspect arise from the business application of this study which helps the improvement of the quality provided by software development companies and teams.

1.5 Document Structure

This proposal was structured in 3 chapters. Chapter 1 was the introduction of the document. Targeting at the first objective chapter 2 presents a Mapping Study about ML techniques for code smell identification. Chapter 3 describe the methodological procedures that will be taken for this experiment. Chapter 4 describes an experiment based on the state-of-art techniques identified in the mapping study, reproduced in a standardized dataset aiming to accomplish the second objective. Finally chapter 5 brings the conclusion of the study and a guidance for future experiments.

2 Mapping Study

2.1 Introduction

One of the most costly operations involving software development is maintenance. It has been reported that above 75% of the total software cost is used for maintenance activities Bennett and Rajlich (2000); Liu et al. (2012). An important factor about it is the quality of the written code, as it affects the readability, and hence the maintainability of the code Aggarwal et al. (2002), given that software maintainers spend around 60% of their time understanding the code they are working at Abran and Nguyenkim (1993). Even in cautiously designed systems, the quality of the source code tends to degrade as the project evolves, provided that the original design of a system is rarely prepared for every new requirement and for quick changes made by different people without properly adjusting the system structure Seng et al. (2006). Another factor is that developers also tend to focus on the addition of the new functions and bug fixes rather than improving software maintainability Tufano et al. (2015). They also tend to overlook the code when it is apparently complex and when it seems not to be critical to maintain the longevity of the software Murphy-hill et al. (2012), one way to better the software maintenance is avoid code smells.

Code smells are code snippets with design problems, their presence in the code difficult the software maintenance and affects the quality of software. When a code smell is detected it is suggest to do refactoring, to remove the code smells in code there are refactoring to each one them. The refactoring improve the code quality but not change the behave of system Chatzigeorgiou and Manakos (2014); Fowler and Beck (1999). Code smells provide heuristics for the identification of design flaws in the source code that makes software harder to evolve, comprehend and maintain. Each code smell examines a specific kind of system elements (classes, methods, and so on) that can be evaluated by its characteristics Olbrich et al. (2009). As the software evolves the number of code smells increase with the time Chatzigeorgiou and Manakos (2010, 2014).

The code smells provide guidelines to identify undesirable behaviour and common coding mistakes, but they still depend on the interpretation of programmers Fowler and Beck (1999), since there can be different interpretations according to the each scenario and they may also be considered critical or not Taibi et al. (2017). The definition of what is and what is not a code smell in a given context may not be a consensus among developers working in the same application Bryton and Abreu (2009); Fontana et al. (2016); Hozano et al. (2017), making their identification an error-prone and time-consuming task considering the size of commercial applications Murphy-hill et al. (2012).

In order to reduce this subjective interpretation, automated approaches based on the source code were presented in previous works Fontana et al. (2012); Fokaefs et al. (2007); Mantyla et al. (2004); Rasool and Arshad (2015), but a relevant part of those approaches are based on code metrics. Bryton et al. (2010); Counsell et al. (2010); Marinescu (2004); Moha and Guéhéneuc (2010); Rasool and Arshad (2015). These techniques use metrics and thresholds that are not consistent among them, leading to a growing number of false positives, not representing the real problem Fontana et al. (2016) since it does not consider information related to the context, domain, size and design of the system Ferme et al. (2013). In this scenario, machine learning techniques can be used to capture this subjectivity. They are techniques utilized for a wide range of applications such as: risk management Cowell et al. (2007), medicine Akay (2009), biology Kell (2005), financial markets Doostmohammadi et al. (2017), among others Fenton and Neil (2007); Li (2017). And they can also be used for the identification of code smells in source code, providing more flexibility in comparison to the current metrics-based approaches Kotsiantis (2007). The tools to identify code smells based on metrics not analyze the past version of code, so it is not allowed to understand the context of code smell, once some code smells can be removed naturally in it evolution while others require more effort to resolve with refactoring Chatzigeorgiou and Manakos (2014).

Studies regarding the application of machine learning techniques are increasing, but each one uses different models and techniques to achieve this task Rasool and Arshad (2015). Even with studies appearing it gets harder to find out the ones that perform better for each code smell. In other words, it is not yet known the real performance about the machine learning approaches applied to identify code smells. Aiming to solve this problem this study has as objective clarify the understand of the methods and practices most frequently adopted in literature when applying machine learning for code smells identification. Thus, this paper answer the following research questions:

- **RQ1**: Which code smells are addressed by papers using machine learning techniques for code smells detection?
- **RQ2**: Which machine learning techniques are used to detect code smells?
- **RQ3**: Which machine learning techniques are the most used for each code smell?
- **RQ4**: Which machine learning techniques perform better for each code smell?

To answer these questions, we developed a mapping study on machine learning techniques and code smells found in literature, covering papers from the introduction of anti-patterns in 1999 by Fowler and Beck (1999) and including papers published up to December 2016. The design flaws were categorized under the definition of the works defined by Fowler and Beck (1999) and Brown et al. (1998).

The study resulted in the classification of 26 papers out of 53 researched papers, separated and categorized by design flaws and applied techniques. We found that regarding F-measure: Association Rules techniques obtained better results on Speculative Generality, Divergent Change, Large Class and Long Method smells. Random Forest technique had better results on Large Class and Long Parameter List. On the other hand, Decision Tree is not good enough to identify Message Chains. While Naive Bayes Classifier presented the worst overall performance among the studied practices on Middle Man, Long Parameter List and Shotgun Surgery smells.

This paper was organized according to the following structure: Section 2 provides a background research about code smells, refactoring and machine learning techniques; Section 3 presents the related works of the area; Section 4 addresses the methodology used in this work; Section 5 displays the results of the study; Section 6 discusses the results; Section 7 presents the threats posed to the validity of the study and finally Section 8 shows the conclusion and provides suggestions for future work.

2.2 Background

2.2.1 Code smells

Code smells are symptoms which can but not necessarily have to point to an actual issue. Therefore, they are not patterns to be avoided, but signals that require a more thorough examination Walter and Alkhaeir (2016). We follow a short description of each smell proposed by Fowler and Beck (1999):

- **Alternative Classes with Different Interfaces**: a case in which a class can operate with alternative classes but the interface of these classes is different.
- **Comments**: misuse of comments to compensate poor code structure.
- **Data Class**: a class that contains data but does not contain logic.
- **Data Clumps**: data items that usually appear together.
- **Divergent Change**: when a class needs to be changed every time another class is changed.
- **Duplicate Code**: code that does the same thing as another piece of code.
- **Feature Envy**: a method that is more interested in other properties of the classes than in the ones from its own class.
- **Inappropriate Intimacy**: when two classes are tightly coupled.

- **Incomplete Library Class**: when the software uses an incomplete library.
- **Large Class**: a class that tries to do a load of things, having plenty of instance variables or methods.
- **Lazy class**: a class that is not doing enough and should be removed.
- **Long Method**: a method that is long, so it is hard to understand, change or extend.
- **Long Parameter List**: a parameter that is long and difficult to represent.
- **Message Chains**: a chain of calls from one object to another, without adding any new behaviour.
- **Middle Man**: when a class delegates a great deal of its behaviour to another class.
- **Parallel Inheritance Hierarchies**: a situation where two parallel class hierarchies exist and are related.
- **Primitive Obsession**: represents the usage of primitives instead of small classes, making it less meaningful and reusable.
- **Refused Bequest**: a child class does fully support its parent implementation.
- **Shotgun Surgery**: when a class changes requires a broadcast changing of other classes.
- **Speculative Generality**: when unnecessary code is created anticipating future changes on software.
- **Switch Statements**: usage of type codes or run-time class type detection instead of polymorphism.
- **Temporary Field**: the class has a variable which is only used in specific situations.

Mantyla et al. Mantyla et al. (2003) categorized code smells into 8 categories as shown below:

- **The Bloaters:** represent something that has grown so large that it cannot be effectively handled. This category covers the following smells: Long-Method; Large Class; Primitive Obsession; Long Parameter List; and Data Clumps.
- **The Object-Orientation Abusers:** represent code that does not exploit the possibilities of Object-Oriented Design. The following smells are included in this category: Switch Statements; Temporary Fields; Refused Bequest; Alternative Classes with Different Interfaces; Parallel Inheritance Hierarchies.

- **The Change Preventers:** smells that prevent or hinder the changing or further development of the system. This category is composed by: Shotgun Surgery and Divergent Change.
- **The Dispensables:** represent something that is unnecessary and should be removed from the code. Represented by the smells: Lazy class; Data class; Duplicated Code and Speculative Generality.
- **The Encapsulators:** smells that deal with data communication or encapsulation, including Message Chains and Middle Man.
- **The Couplers:** smells that increase the coupling of the system, being composed of Feature Envy and Inappropriate Intimacy.
- **Others:** smells that do not fit in any of the previous categories and are not comparable, such as: Incomplete Library Class and Comments.

Code smells can also be considered as symptom of a design level flaw, also known as anti-patterns Moha and Guéhéneuc (2010). The anti-patterns concept was introduced by Brown et al. (1998) defining it as a literary form that describes a recurrent solution to a problem that generates decidedly negative consequences. The studies also use approaches to identify the code anti-patterns instead of code smells, since they describe more generic flaws, in this study we use three of them:

- **The Blob**: corresponds to a large controller class that depends on data stored in surrounding data classes. A large class declares fields and methods with a low cohesion. A controller class monopolizes the processing done by a system, takes the main decisions, and directs the processing of other classes.
- **The Functional Decomposition**: consists of a main class in which inheritance and polymorphism are scarcely used, associated with small classes, which declare a great deal of private fields and implement sparse methods.
- **The Spaghetti Code**: classes with no structure, declaring long methods with no parameters, and utilizing global variables for processing. Names of classes and methods may suggest procedural programming.

2.2.2 Machine Learning

Machine learning techniques can be categorized in three ways: supervised, unsupervised and semi-supervised. If instances are given with known labels (the human annotated correct output) then the learning is called supervised, otherwise, when the instances are unlabeled, it is unsupervised learning Jain et al. (1999). There is also a hybrid

approach, which is the semi-supervised learning that uses both labeled and unlabeled data to perform an otherwise supervised learning or unsupervised learning task Zhu (2011).

Supervised methods Kotsiantis (2007) are the leading approach used for code smells identification Fernandes et al. (2016). The following supervised approaches were identified by Kotsiantis (2007) in his literature review:

- **Decision Trees:** Decision Trees are trees that classify instances by sorting them based on feature values. Each node in a Decision Tree represents a feature in an instance to be classified, and each branch represents a value that the node can assume. Instances are classified starting at the root node and sorted based on their feature values.
- **Learning Set of Rules:** Decision Trees can be translated into a set of rules by creating a separate rule for each path from the root to a leaf in the tree. However, rules can also be induced from training data using a variety of rule-based algorithms.
- **Single layered perceptrons:** Uses a single layer of weights to define a linearly separable binary classification.
- **Multi layered perceptrons (Artificial Neural Network):** Created to solve non linear classification problems that cannot be solved by a single layer. A multi-layer neural network consists of large number of units (neurons) joined together in a pattern of connections.
- **Radial Basis Function (RBF) network:** An RBF network is a three-layer feed-back network, in which each hidden unit implements a radial activation function and each output unit implements a weighted sum of hidden units outputs.
- **Naive Bayes:** Naive Bayesian networks (NB) are simple Bayesian networks which are composed of graphs with only one unobserved node and a chain of children observed nodes, with an assumption of state independence between child nodes and their parent. The Naive Bayes is based on estimating the probabilities of the unobserved node, based on the observed ones.
- **Bayesian networks:** A Bayesian Network (BN) is a graph based model that establishes a probability relationship among a set of known variables. The Bayesian network structure is a graph containing nodes linked with its features. The features must be conditionally independent from their non-descendants in relation to its parents.
- **Instance Based learning:** Instance-based learning algorithms are statistically based and lazy-learning algorithms, as they delay the induction or generalization process until classification is performed.

- **Support Vector Machines (SVM):** SVMs are based on the notion of a "margin" in either side of a hyperplane separating two features. Its optimizing objective is to increase the margin and create the largest distance between features in the hyperplane. The complexity is unaffected by the number of features. So SVMs are suited to deal with learning tasks where the number of features is large with respect to the number of training instances.

Unsupervised learning is composed mainly of Clustering Techniques. The clustering objective is to develop an automatic algorithm that discovers the natural groupings in the unlabeled data Jain et al. (1999). Clustering algorithms can be broadly divided into two groups: hierarchical and partitional. Hierarchical clustering algorithms recursively find nested clusters, while the partitional clustering find the clusters simultaneously. In semi-supervised clustering, instead of specifying the class labels, constraints are specified, as a weaker way of encoding the labeled data. Semi-supervised learning can be applied in place supervised learning, using unlabeled data for training Jain (2010). Semi-supervised learning has been applied to natural language processing (word sense disambiguation, document categorization, named entity classification, sentiment analysis, machine translation), computer vision (object recognition, image segmentation), bio-informatics (protein function prediction), and cognitive psychology Zhu (2011), and also to address code smell identification problems Fernandes et al. (2016).

Other algorithms can be used to identify code smells are Genetic Algorithms (GA). The GAs are algorithms try to imitate natural selection based on biological mechanisms Newman (2006). They working with concept of genome and three operators: Reproduction, crossover and mutation, proving robust search in complex spaces, which they find the optimal solution Goldberg and Holland (1988); Newman (2006); Lee et al. (2011).

2.3 Related work

Rasool and Arshad (2015) elaborated a literature review to identify state-of-art tools used for mining code smells based on the source code of different software projects. It identified that earlier code smell detection techniques were focused on static source code analysis methods for detecting code smells, afterwards it shifted to a combination of static and dynamic source code analysis methods. Recently, search-based code smell detection techniques obtained attention, applying code metrics and machine learning methods for the identification. A large number of code smell detection techniques used source code metrics for the detection, computing metrics from source code or from third-party tools, usually aiming at Feature Envy, Data Class, Large Class, Long Method, and Long Parameter List code smells, while alternative classes with different interfaces and Incomplete Library Class were not covered. In regard to the tools, many apply different object-oriented

source code metrics to detect a large number of code smells. They presented high language dependency for the detection of code smells, given that 38 out of the 46 tools are focused on Java language. The same pattern repeated regarding the experiments. The performance of the tools varied depending on the studied smell, the same happened in the use of different tools for the same smell. They claim those differences to be due to the selection of different metrics and thresholds by the different techniques and also criticize the unavailability of standard benchmark systems for comparing results of code smells detection tools. The dependence on code metrics also made it harder to reuse the techniques to detect different code smells, the accuracy commonly relied on the selection of threshold values and may be deceptive. 30% of the tools spotlight the accuracy, that is, precision and recall, of their technique or tool. The study provided an insight on the evolution of the techniques used for code smell and the current state of the tools and techniques, identifying research gaps and opportunities, exposing that the machine learning techniques have been generating growing interest through the researches. Our study aims to cover the machine learning techniques, to understand how they are being used and which techniques fit and perform better for each code smell. Different from the given study, that focuses on general techniques and tools that were used for code smells identification, we did a mapping study focusing on the machine learning techniques that were applied to code smells and their performance.

Fernandes et al. (2016) performed a systematic review to identify and document all tools reported and used in the literature for bad smell detection. From the 22 bad smells defined by Fowler and Beck (1999), they identified tools to detect 20 of them, the exceptions are Alternative Classes with Different Interfaces and Incomplete Library Class. While Duplicated Code and Large Class were the most targeted smells, more than 40% of the tools target at least one of these bad smells. In addition, there were also tools to identify 41 smells defined by other authors. From the analyzed tools, 30 of them are plug-ins, 30 are standalone applications and 4 of them are available as both. They found a concentration of proposed tools for three languages: Java, C, and C++. But only one out of the top 10 used languages were not covered by any tool. The authors developed a comparison among the 4 selected tools: inFusion; JDeodorant; PMD and JSpirit, and 2 code smells: Large Class and Long Method, selected because they are the most covered smells, for Duplicated Code, it was hard to quantify its results. Each tool was tested agains the projects: JUnit, to check agreement between the tools, and MobileMedia for agreement, recall and precision. Regarding recall, PMD and JSpIRIT provided the highest results achieving 50% and 67% of recall respectively. When analyzing precision, inFusion and PMD had the highest values achieving 100% of precision. This study contributed for the automatic code smell detection, by cataloguing the tools used for code smells detection and also the situations in which they can be used. However, this study differs from ours, since it focuses on tools for code smells detection and not on the techniques, it

provides little information about the used techniques, as some tools do not provide that information, whereas our study focuses on the machine learning techniques, excluding studies that rely only in predefined or user-defined metrics.

Rattan et al. (2013) developed a systematic literature review aiming at identifying and classifying the existing literature about clone detection, clone management, semantic clone detection and model based clone detection techniques. In order to do so, 213 studies were reviewed, where 100 were found to be research studies of software clone detection. The studies generally used an intermediate code representation, with different granularity, Abstract Syntax Trees (ASTs) or Parse trees, source code or text and Regularized tokens are the most frequently used. Regarding the matching technique: metrics/feature vectors clustering, Suffix Tree based token by token, substring/subtree model comparison and Dynamic programming were the most common approaches. The authors concluded that clones definition are still unclear and that there is a lack of empirical studies regarding the harmfulness of clones. They also concluded that it can be used as principled re-engineering technique and be beneficial as it is not easy to refactor all the clones due to cost/risk associated with refactoring. It is suggested that instead of removing clones, we should have proper clone management facilities. This study is similar to ours for it also focuses on a code-smell identification and the techniques used for that, but it differs from our study since it focuses only on one code smell and uses a broader scope for the techniques and it also contemplates any automated identification task and not only machine learning techniques.

Al Dallal (2015) reports a systematic literature review that identifies the state-of-the-art techniques regarding the identification of refactoring opportunities, assessing and discussing the collected and reported findings. The studies considered 22 refactoring opportunities; 20 of them are among the 72 activities identified by Fowler and Beck (1999) [23], and two were proposed by others. It identified that Quality metrics-oriented, Precondition-oriented and Clustering-oriented were the preponderant techniques. For the empirical evaluation, intuition-based techniques were the frequently used, followed by Quality based and Mutation based evaluations. The studies in general used Java open source projects for their empirical experiments. The author found that the studies focuses more on Move Method, Extract Class, and Extract Method than in the other refactoring activities, justifying their importance for the industry, while 72.2% of the refactoring activities proposed by Fowler were not considered by any study. It was also found that most of the studies lack empirical data to support their techniques. This study is related to ours, since the techniques used for refactoring and the techniques for the code smells identification share common features, such as metrics and techniques aiming at their identification, but it is different of ours in its nature, as we focus on identifying the code smells while it focuses on identifying refactorings opportunities.

2.4 Research Method

This study performed a mapping study on the usage of machine learning techniques for code smells identification. Mapping studies are based on a clear search strategy, that ensures the rigor, completeness and a reproducible process, focusing on the identification, evaluation and interpretation of the available research that is relevant to a particular question Kitchenham et al. (2010).

The process adopted for mapping study was based on the work of Kitchenham et al. Kitchenham et al. (2015) that includes three phases: planning, conduction and documentation. The following subsections describe the steps taken in the planning and conduction phases. The documentation phase is addressed in the sections 2.5 and 2.6.

2.4.1 Planning

We developed a protocol according to the guidelines provided by Kitchenham et al. Kitchenham et al. (2015) in order to ensure that the research was executed in a planned way and not driven by the expectations of the researcher. In the protocol, the following items were documented as part of the planning for the study: research objectives; research questions; search strategy; study selection, quality assessment of the studies; data extraction; and data synthesis and aggregation.

2.4.2 Research Questions

This research focuses on the identification machine learning techniques used for code smells detection and in order to accomplish that the following questions must be answered:

- **RQ1: Which code smells are addressed by papers using machine learning techniques for code smells detection?** RQ1 focuses on the most common code smells detected by studies utilizing machine learning techniques. The answer to this research question can help identify code smells that have not been explored in existing research and those that can be opportunities for future work.

- **RQ2: Which machine learning techniques are used to detect code smells?** RQ2 aims in the identification of machine learning techniques commonly used to detect code smells in the literature. This information can be useful to determine the most popular machine learning techniques used in the code smells domain and also to generate a list of potential machine learning techniques that were not explored for code smells detection.

- **RQ3: Which machine learning techniques are the most used for each code smell?** RQ3 is concerned with the relationship between the machine learning techniques and the code smells detected by these techniques. The outcome of this question can provide insights on the association between machine learning techniques and types of code smells. It could also lead to the identification of gaps in research pointing to potential machine learning techniques that can be used for similar code smells but have not been used for this intent.

- **RQ4: Which machine learning techniques perform better for each code smell?** RQ4 compares the performance of different machine learning techniques for code smell detection. The answer to this question can help researchers define which machine learning techniques to use when studying different code smells.

2.4.3 Search Strategy

In order to find relevant papers to achieve the goals of this study, we conducted searches based on the recommendations by Kitchenham et al. Kitchenham et al. (2015). In the first step, we used personal experience to define a list of journals and conferences, summarized in table 1, used to establish the quasi-gold standard (QGS). The QGS for this mapping study is composed of 21 articles. The following digital libraries were used in the automated searches: ACM, IEEE, Science Direct and Wiley.

Table 1 – List of conferences and journals used in the manual search

Conference/Journal Title
- Empirical Software Engineering
- Expert Systems with Applications
- International Conference on Software Engineering (ICSE)
- IEEE Transactions on Software Engineering
- International Conference on Frontiers in Intelligent Computing: Theory and Applications
- International Conference on Software Maintenance (ICSM)
- International Symposium on Software Reliability Engineering (ISSRE)
- Journal of Software Maintenance and Evolution
- Journal of Software: Evolution and Process
- Journal of Systems and Software
- Knowledge-Based Systems

The definition of the search string followed these steps:

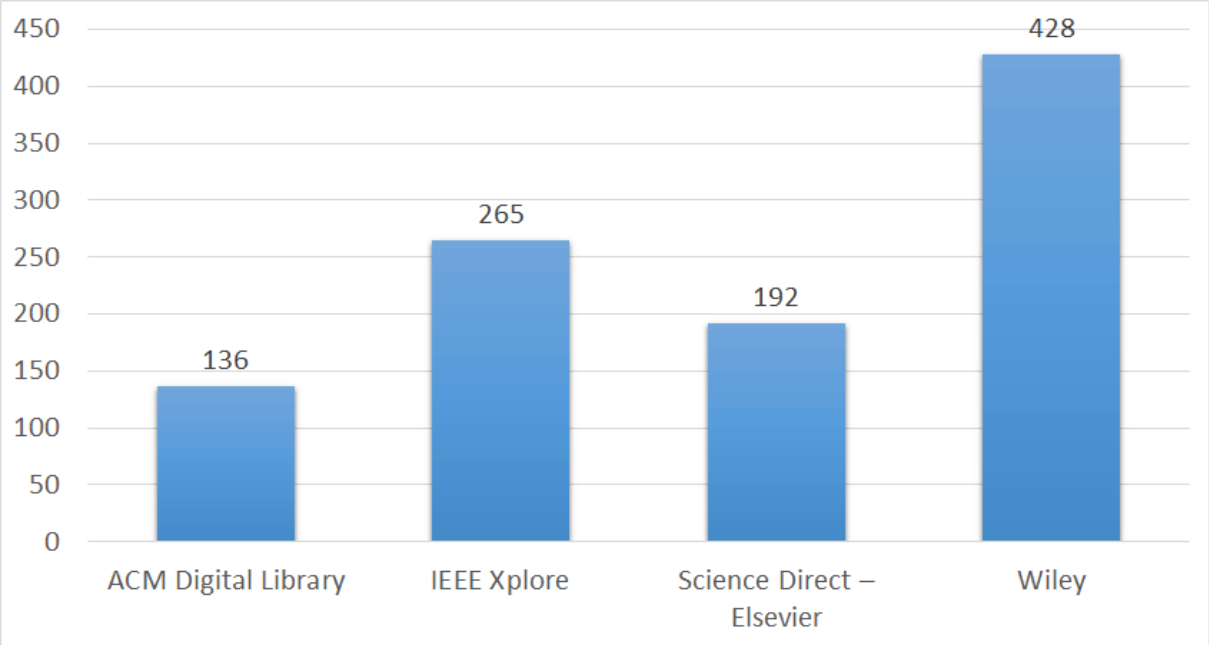
1. Derived major terms for machine learning and code smells from research articles.

2. Broke major terms down into smaller terms the machine learning related terms were based on Wen et al. (2012).

3. Identified alternative spellings or synonyms for the smaller terms.
4. Checked the search terms in known relevant papers.
5. Used Boolean operator OR combined alternative spellings and synonyms. Used Boolean operator AND linked machine learning and code smells terms.

This is the search string used for papers retrieval in automated searches: "("code smell" OR "bad smell") Fowler and Beck (1999) AND ("learning" OR "data mining" OR "artificial intelligence" OR "pattern recognition" OR "case based reasoning" OR "decision tree" OR "regression tree" OR "classification tree" OR "neural net" OR "genetic programming" OR "genetic algorithm" OR "Bayesian belief network" OR "Bayesian net" OR "association rule" OR "support vector machine" OR "support vector regression")"Wen et al. (2012). The automatic search returned 1021 papers distributed among the digital libraries as summarized in Figure 1.

Figure 1 – Distribution of papers found in automated search by digital library



The search performance was measured using the 'quasi-sensitivity' which is the recall of the search. The result had a sensitivity of 85% higher than the threshold of 70% to 80% proposed by Zhang et al. Zhang et al. (2011a). Based on this result, we proceeded to the next phase of the mapping study.

2.4.4 Studies Selection

The selection criteria targets at filtering relevant studies to the research questions Kitchenham et al. (2015). We defined three inclusion and three exclusion criteria.

Inclusion criteria:

- Papers from computer science, information system or software development fields.
- Papers describing the use of machine learning techniques applied to code smells detection.
- The studies should be peer-reviewed.
- The studies should be published in English.

Exclusion criteria:

- Remove duplicates.
- Remove studies in which the code smells detection is done manually.
- Remove works that were not completed.
- Remove non-empirical studies.

The selection process consisted of five stages as shown bellow and it was administered by three of the authors. Each step was peer-reviewed by graduated students, and the results were compared and discussed in order to reach a consensus.

1. The databases search returns 1021 papers
2. After the exclusion of duplicate paper 683 papers left
3. Evaluated the type of review, fieldof the research and publication language 286 papers left
4. Removed studies unrelated to machine learning techniques for code smells identification and also non-empirical studies 155 papers left
5. Removed papers based on titile and abstract 53 papers left

In the first stage of the selection process, we removed 338 duplicated papers and moved 683 papers to stage 2. Stage 2 evaluated the type of review, field of the research and publication language. This stage selected 286 articles to move to the next part of the process. The studies were then filtered by title in stage 3, removing studies unrelated to the usage of machine learning techniques for code smells identification and also non-empirical studies resulting in 155 articles. In stage four, we applied the same criteria as in stage three but the filtering was based in the abstract of the studies. Stage 4 selected 53 papers to continue in the review process going through quality assessment and classification.

2.4.5 Quality Assessment

In this mapping study we used the quality assessment to have a better insight about the selected papers to exclude papers for the final data extraction and analysis. Our intention was to use the detailed quality information about the paper during analysis and also to identify which percentage of the papers would have comparable results.

The quality assessment questions used in our study were based on the quality checklist defined by Dybå and Dingsøyr Dybå and Dingsøyr (2008):

- **QA1**: Is the paper based on research (or is it merely a lessons learned report based on expert opinion)?
- **QA2**: Are the aims of the research defined explicitly?
- **QA3**: Is there experimental design appropriate and justifiable?
- **QA4**: Is the proposed estimation method comparable with other methods?
- **QA5**: Are the findings of study stated and supported by the reported results?
- **QA6**: Does the study add value to the academy or to the practice communities?

The quality of the papers was assessed by three of the researchers during data extraction. All of them performed the quality assessment checklist for each of the papers. In order to reach a consensus a Delphi method Dalkey and Helmer (1963) was used. Initially, each participant received a spreadsheet containing the selected papers and the quality assessment questions, which they had to fill with either yes or no. In each round a paper was selected and the researchers answers to each of the quality questions was compared, in case of disagreement the participants had to justify and discuss their answers, then another round would be executed. This process was repeated until a consensus was achieved.

2.4.6 Data Extraction and Classification

The data extraction process was conducted by three of the authors. Each author read the papers and provided values for the items available in the classification form. The data was reconciled using discussion and moderation to achieve an agreement between the reviewers.

The classifications used in this phase were Fowler and Beck (1999); Rasool and Arshad (2015); Fernandes et al. (2016):

- Data source: The projects used for the technique assessment;

- Language: The language of the used project;
- Metrics: Which metrics were taken into consideration;
- Baseline: The baseline against which the technique performance was compared;
- Addressed Code Smells: The code smells addressed by the paper.

2.5 Results

In this section we present the results of the systematic mapping. Firstly, we show an overview of the selected papers and then the answers to the research questions and their findings.

2.5.1 Overview

In this study we identified 26 papers that used machine learning techniques for code smell identification. They were published between 1999 to 2016 and they used experimentation as methodology. 65% of the papers were published in conferences and the other 37% were published in journals. 5 publications were represented by more than one publication venue: Annual Conference on Genetic and Evolutionary Computation; International Conference on Software Engineering; Journal of Systems and Software; Expert Systems with Applications; Journal of Software: Evolution and Process. These publications represented almost half of the selected papers as shown in Table 2.

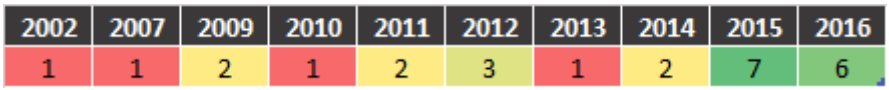
Table 2 – Papers by publication

Publication	# of studies
Annual Conference on Genetic and Evolutionary Computation	3
International Conference on Software Engineering	3
Journal of Systems and Software	3
Expert Systems with Applications	2
Journal of Software: Evolution and Process	2
Others	13

Through Figure 2 it is possible to see a trend in the publications over the years. Figure 2 shows a greater interest in studies identifying code smell using machine learning techniques in 2015 and 2016 than between 2002 and 2014. The number of researches in the last two years overcame the numbers of the previous years. It is worth remembering that the code smell term was used by Kent Beck in the late 1990's and the research about code smell was focused on other techniques as metrics. Detection of code smells based in machine learning techniques has not been extensively explored Fontana et al. (2013).

All papers selected in this study identified code smells analyzing projects developed with Java language. Out of 26 papers, 23 used open source projects, while 3 used private

Figure 2 – Number of Papers by year



data source. We identified 88 open source projects as dataset used in research, the dataset more frequently used was the Xerces, followed by JHotDraw, Eclipse Core and ArgoUML. Among the 88 datasets, 62 of them were used once. As detailed by the Table 3 below:

Table 3 – Open source projects adopted

Software	# Papers	Software	# Papers
Xerces	6	Apache Commons Logging	1
JHotDraw	5	JDI-Ford	1
Eclipse Core	5	Apache Derby	1
ArgoUML	4	JFreeChart	1
JFreeChart	3	Apache James Mime4j	1
GanttProject	3	JRDF	1
Apache Cassandra	2	Apache Tomcat	1
Rhino	2	Maven	1
Qualitas Corpus	2	ApacheAnt	1
GanttProject	2	Pixelitor	1
jEdit	1	ApacheAnt	1
Ant	1	sapphire	1
Log4J	1	Android API (framework-opt-telephony)	1
And Engine	1	XWorks	1
JabRef	1	BCEL	1
Apache Commons Codec	1	Jboss	1
Android API (tool-base)	1	Closure Compiler	1
Apache Commons IO	1	JDK	1
nebula.widgets.nattable	1	dltk.core	1
Apache Commons Lang	1	Android API (sdk)	1
Xerces	1	Android API (frameworks-base)	1
JGraphx	1	Aardvark	1
egit	1	platform.resources	1
JHotDraw	1	Gitblit	1
FindBugs	1	Ant-Apache	1
jUnit	1	Google Guava	1

FreeMind	1	ANTLR	1
Lucene	1	graphiti	1
GanttAzureus	1	Xom	1
Mongo DB	1	Guava	1
Android API (frameworks-support)	1	Apache Ant	1
Nutch	1	Hibernate	1

2.5.2 Which code smells are addressed by papers using machine learning techniques for code smells detection?

In this section, in addition to Fowler and Beck (1999) smells and Brown et al. (1998) anti-patterns we also the included a classification called "others", meant to capture smells defined by other authors and code-flaws not related to a specific smell, since the authors aim at code metrics optimization instead of a specific smell.

Comparing the studied code-related design flaws using machine learning, the ones with higher occurrence were Feature Envy smell and BLOBs, both studied by 5 papers, followed by Long Methods that showed up in 4 papers. While Comments , Primitive Obsession, Refused Bequest, Alternative Classes with Different Interfaces and Incomplete Library Class were not addressed by any of the studied papers. The distribution of the smells is displayed in Figure 3. Except for the Duplicated Code, which is one of the most studied smells, the other smells are coherent with the ones identified by Zhang et al. (2011b). The category Others, composed mainly of optimization in the code metrics, appeared 8 times, using alternatives to the traditional code smells.

When grouping by the classification defined by Mantyla et al. (2003) it is possible to observe that the main focus regarding the addressed type of smell are the Bloaters representing 35% of the studied smells. As detailed on Figure 4.

When analyzing the smells studied together, one of the strongest co-occurrences is between BLOB, Spaghetti Code and Functional Decomposition. One of the main reasons is that they use the anti-patterns definition by Brown et al. (1998), while the others are defined by Fowler and Beck (1999). Long Parameter List, Large Class and Long class also presented a high correlation, a possible explanation is that they share the same type, what makes it easier to apply the same kind of algorithm to all of them. The co-occurrence graph is shown in details in figure 5.

Figure 3 – Number of papers by Code Smell

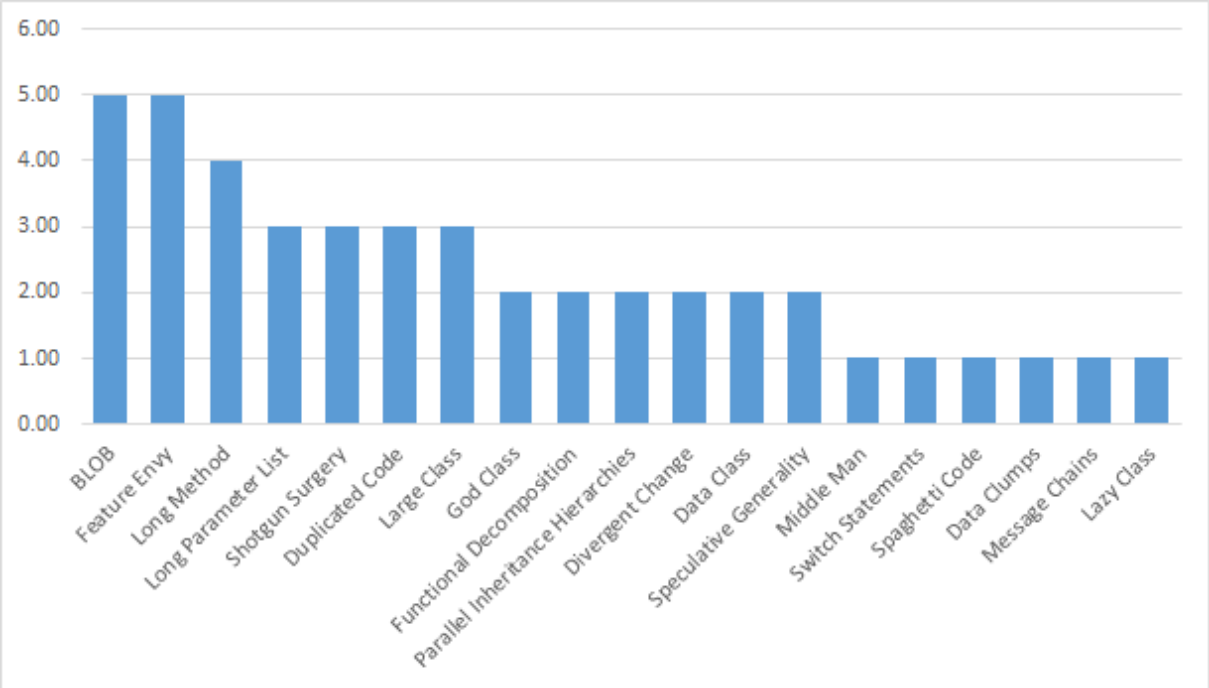
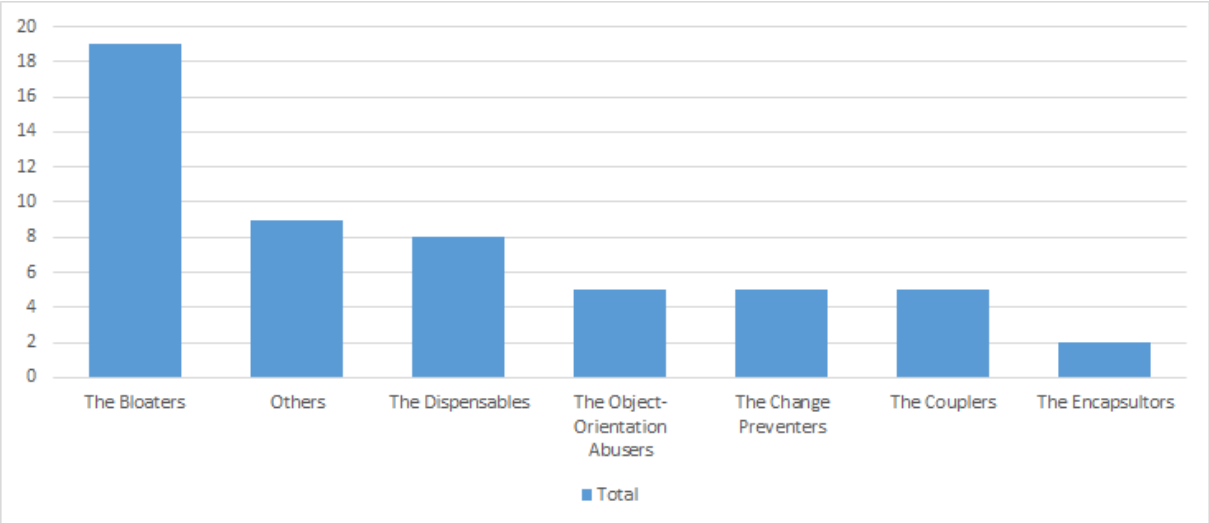


Figure 4 – Number of papers by Code Smell Type



2.5.3 Which machine learning techniques are used to detect code smells?

The leading technique in the analyzed papers was the Genetic Algorithm which appeared 8 times. This technique is used in search-based techniques and focuses on optimizing one or more metrics by mutating and enhancing the code. It was followed by Naive Bayes Classifiers that appeared 4 times. Whereas approaches such as Linear discriminant analysis; Decision Tree; Support; Vector Machine; Directed Acyclic Graph; and Text-Based; showed up once. The distribution can be viewed in Figure 6.

Regarding the kind of technique used, the supervised techniques were the majority, being used in 32 tests (88%); Semi-supervised and unsupervised techniques appeared

Figure 5 – Graph representing which smells were assessed in the same paper

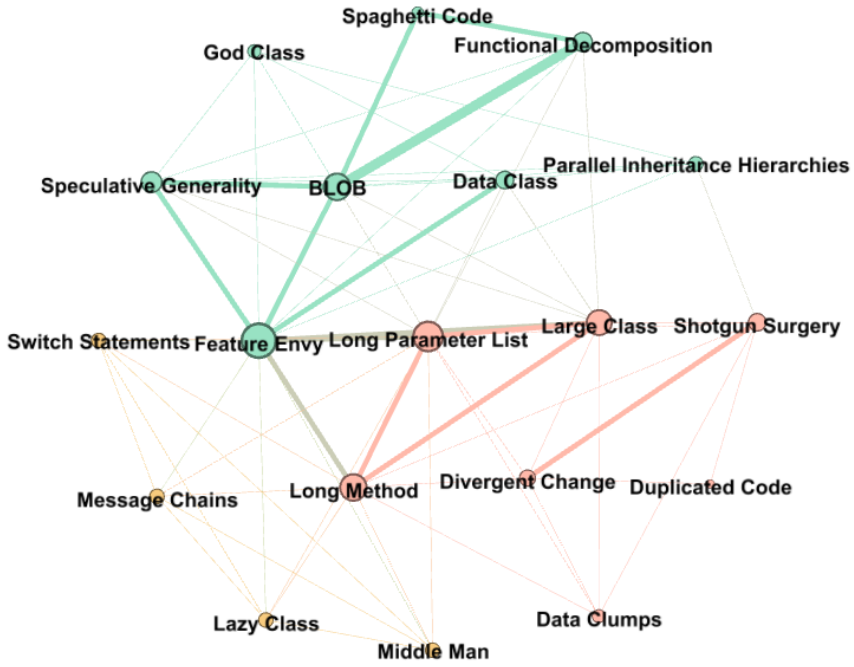
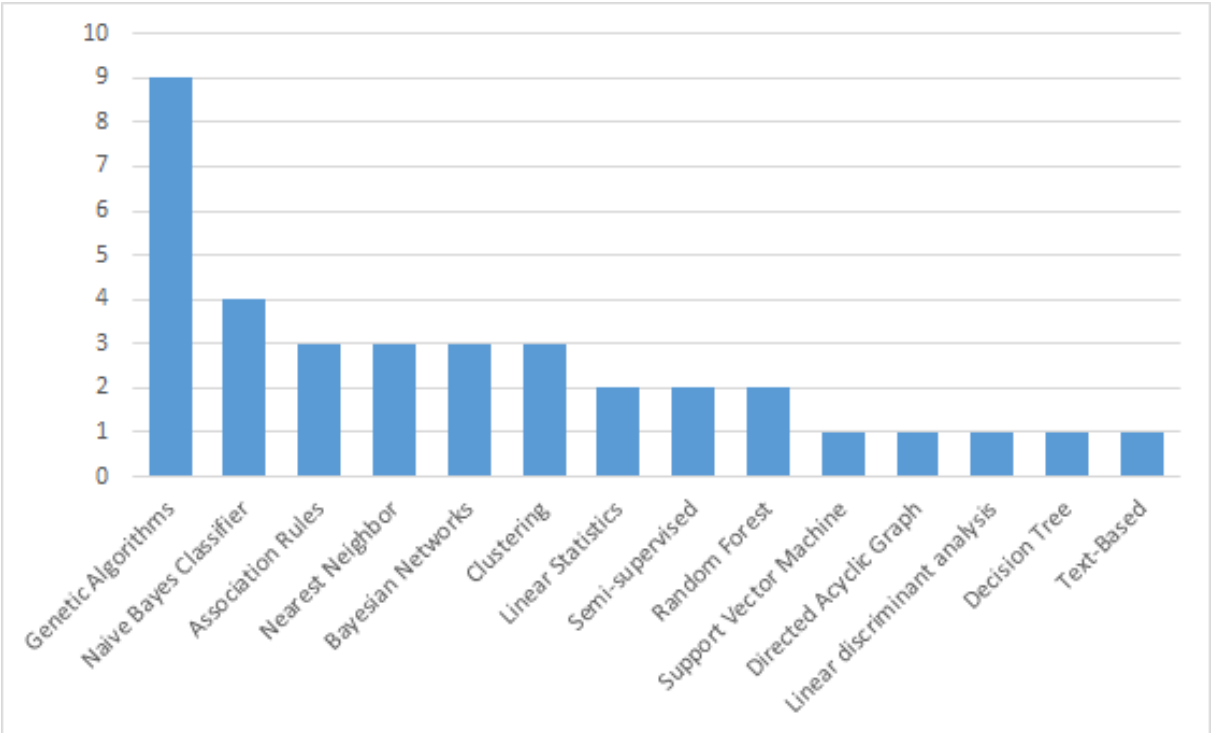


Figure 6 – Number of Papers by Machine Learning Technique



6% each. The results were expected since supervised tests are the most often used in researches Kotsiantis (2007).

2.5.4 Which machine learning techniques are the most used for each code smell?

The smells addressed by the techniques were coherent with the smells that appeared in the related papers Fernandes et al. (2016), showing a high level of redundancy. The smell focused by most of the techniques was Feature Envy, which was aimed by 9 techniques (64%). Followed by Long Method with 8 techniques (57%) and Long Parameter List with 7 (50%). From the smells targeted by the studied papers, the ones covered by less techniques were Speculative Generality, Spaghetti Code, Data Class, God Class, Parallel Inheritance and Divergent Changes with 2 techniques each (14%).

From a machine learning technique perspective, the Association Rules technique was the technique with the highest usage, covering 13 smells (59% of them), followed by Linear Statistics with 11 smells (50%), Naive Bayes and Random Forest with 9 smells (43%). While Text-Based and Linear Discriminant Analysis with 1 smell (5%) are in the lower half.

When comparing the relationship between the code smell and the techniques, there was a relationship between the following techniques and the respective code smells: Association Rules for Divergent Change, God Class, Data Clumps, Parallel Inheritance, Shotgun Surgery and Speculative Generality; Bayesian Networks for Speculative Generality; Clustering for God Class and Parallel Inheritance Hierarchies; Genetic Algorithms for Spaghetti Code; Linear Statistics for Data Clumps and Semi Supervised for Speculative Generality. The relation between smells and techniques can be visualized in Figure 7. When analyzing the smell categories, we found out the Association Rules technique with a relevant focus on the Change Preventers type of smell, as the usage of the technique focus on the relationship between methods and classes. The other smell types did not show any relevant relationship.

2.5.5 Which machine learning techniques performs better for each code smell?

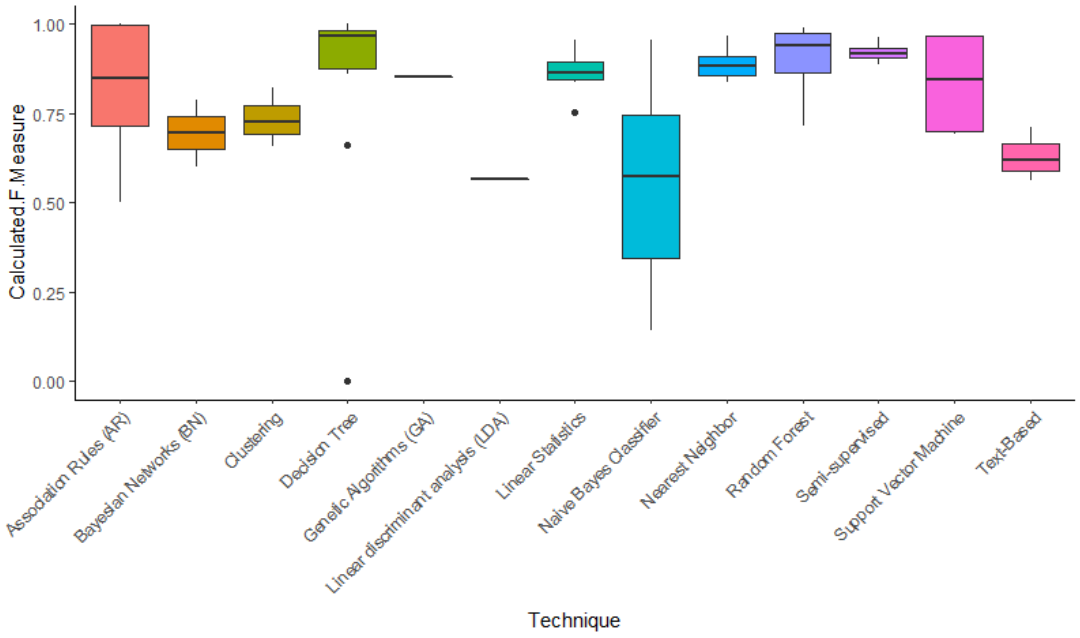
One hardship we found when comparing the performance of the techniques is the lack of standardized data. 14 out of the 26 papers provided performance information. From those, 13 provided precision data, 12 provided recall values and 6 provided us with F-measures. We used the recall and precision data provided by the other to calculate their f-measures as well as to have a comparison measure. In order to have comparable parameters, we also selected the papers that used the same baseline as the majority of the articles, in this case a manual annotation of the code smells.

Figure 7 – Relationship between techniques and code smells

Machine Learning Techiniques	Code Smells																		
	BLOB	Data Class	Data Clumps	Divergent Change	Duplicated Code	Feature Envy	Functional Decomposition	God Class	Large Class	Lazy Class	Long Method	Long Parameter List	Message Chains	Middle Man	Parallel Inheritance Hierarchies	Shotgun Surgery	Spaghetti Code	Speculative Generality	Switch Statements
Association Rules	17%	25%	50%	67%	25%	8%	0%	50%	17%	0%	8%	10%	0%	0%	50%	50%	0%	50%	0%
Bayesian Networks	33%	0%	0%	0%	0%	0%	33%	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%
Clustering	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%	0%	0%	0%	0%	50%	25%	0%	0%	0%
Decision Tree	0%	0%	0%	0%	0%	8%	0%	0%	0%	14%	8%	10%	14%	14%	0%	0%	0%	0%	14%
Genetic Algorithms	33%	0%	0%	0%	25%	0%	33%	0%	0%	0%	0%	0%	0%	0%	0%	0%	50%	0%	0%
Linear discriminant analysis	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Linear Statistics	0%	0%	50%	33%	0%	8%	0%	0%	17%	14%	15%	20%	14%	14%	0%	25%	0%	0%	14%
Naive Bayes Classifier	0%	25%	0%	0%	0%	23%	0%	0%	17%	29%	23%	20%	29%	29%	0%	0%	0%	0%	29%
Nearest Neighbor	0%	0%	0%	0%	25%	15%	0%	0%	0%	29%	15%	20%	29%	29%	0%	0%	0%	0%	29%
Random Forest	0%	25%	0%	0%	0%	15%	0%	0%	17%	14%	15%	10%	14%	14%	0%	0%	0%	0%	14%
Semi-supervised	17%	0%	0%	0%	25%	8%	33%	0%	17%	0%	0%	10%	0%	0%	0%	0%	0%	50%	0%
Support Vector Machine	0%	25%	0%	0%	0%	8%	0%	0%	17%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%
Text-Based	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%	0%	0%	0%	0%

In terms of f-measure the best average performance was provided by Decision Tree, followed by Random Forest, Semi-supervised and Nearest Neighbor techniques. While Text-Based, Linear Discriminant Analysis and Naive Bayes presented the worst performance overall between the studied practices, as demonstrated by Figure 8.

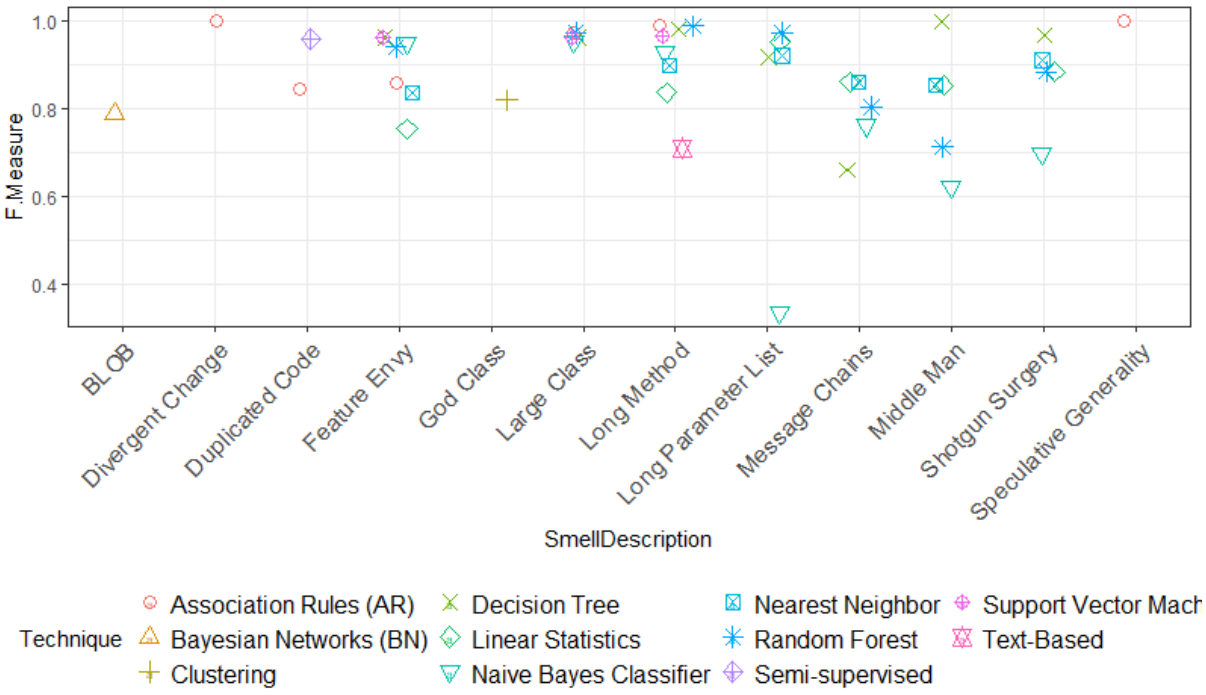
Figure 8 – Machine Learning techniques F-measure Box-plot



When comparing the results by f-measure as demonstrated by Figure 9, it is possible to notice that the Association Rules technique performed above the others for Di-

vergent Changes and Speculative Generality smells, the ones covered by this technique. But it performs poorly for Duplicated Code and Feature Envy, when compared to the other techniques. Decision Tree was also the best performing technique for Middle Man and Shotgun Surgery smells, while Random Forest had an outstanding performance for Long parameter list and Semi-supervised techniques for Duplicated Code. Naive Bayes on the other side, performed poorly for Long Parameter List, Middle Man and Shotgun Surgery. In regard to the other smells, there was no outstanding technique.

Figure 9 – F-Measure technique by code smell



When analyzing the techniques by precision, the Linear Discriminant Analysis technique presented the best average performance, followed respectively to its performance by Association Rules, Semi-supervised and Decision Tree. In this aspect the worst performing techniques were the Bayes-based techniques: Naive Bayes Classifier and Bayesian Networks. The results can be visualized in Figure 10.

Comparing the techniques by precision, Association Rules, also demonstrated an outstanding performance on Divergent Changes and Speculative Generality, where it is the only technique used. Semi-supervised techniques had an outstanding performance for Duplicated Code, Random Forest for FE and Decision Tree for Lazy Class and Middle Man also worth mentioning. We had the Bayesian Networks and Naive Bayes Classifiers performing poorly than the other techniques on the smells. Those observations are displayed in Figure 11.

When compared by recall, the techniques, in general, performed above 80%. The best performing techniques under these perspectives were: Bayesian Networks, Decision

Figure 10 – Machine Learning techniques Precision Box-plot

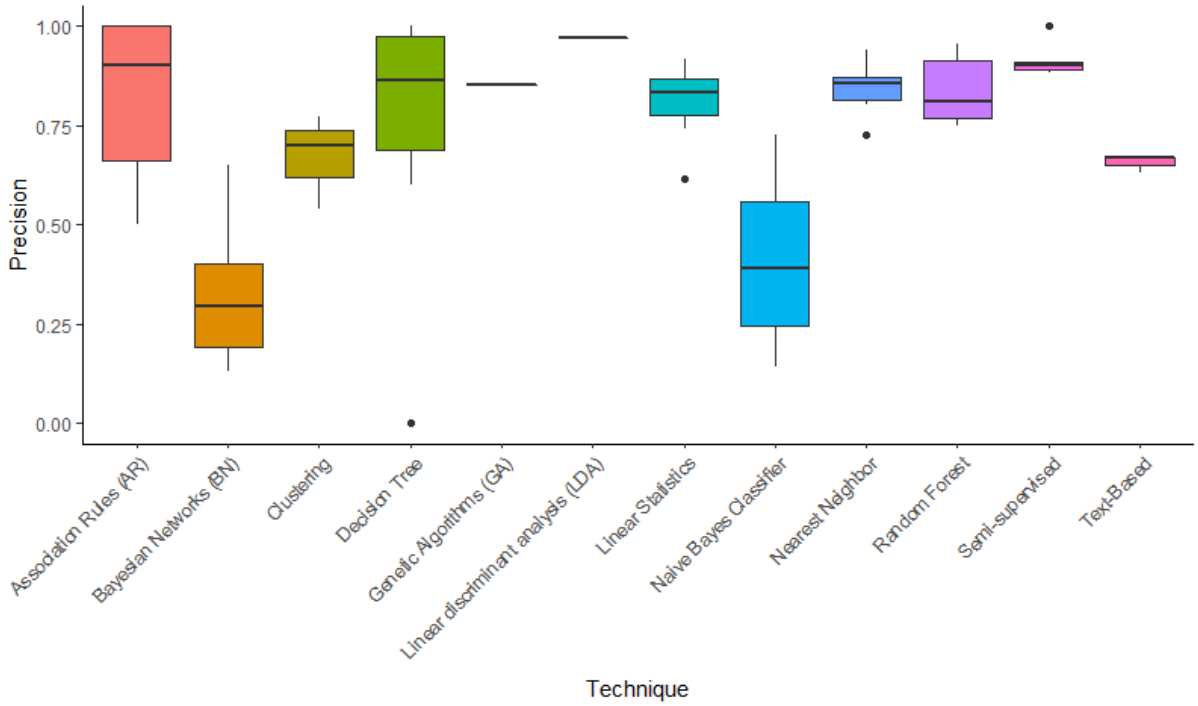
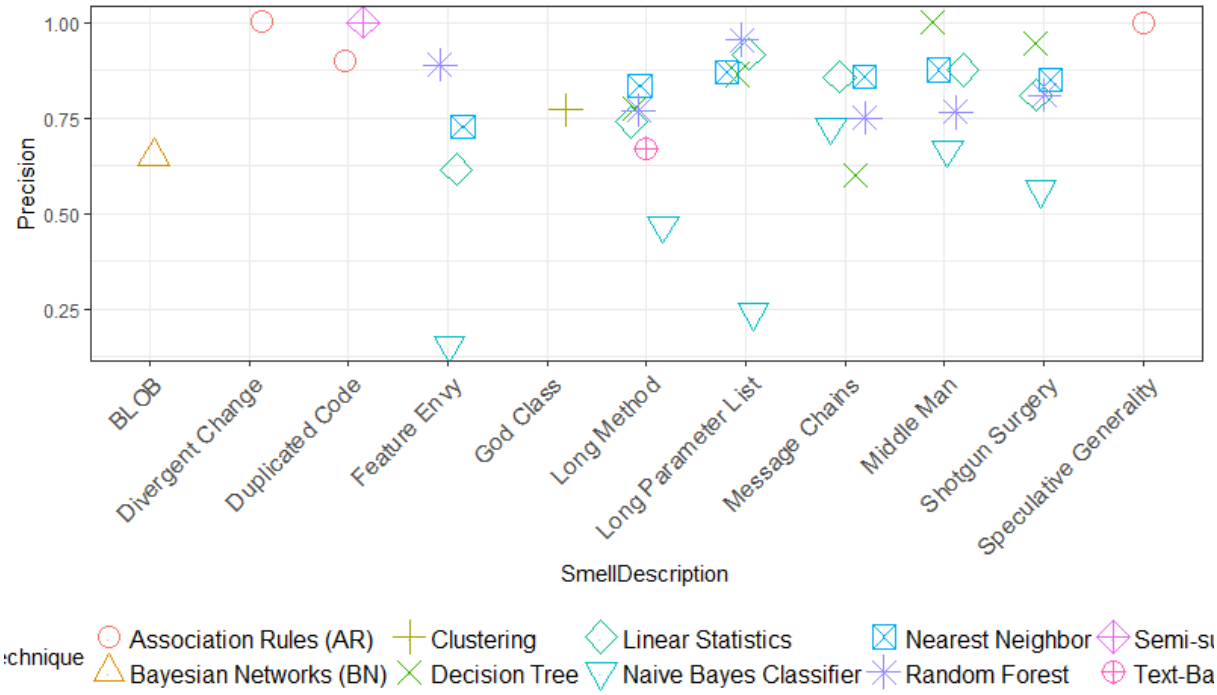


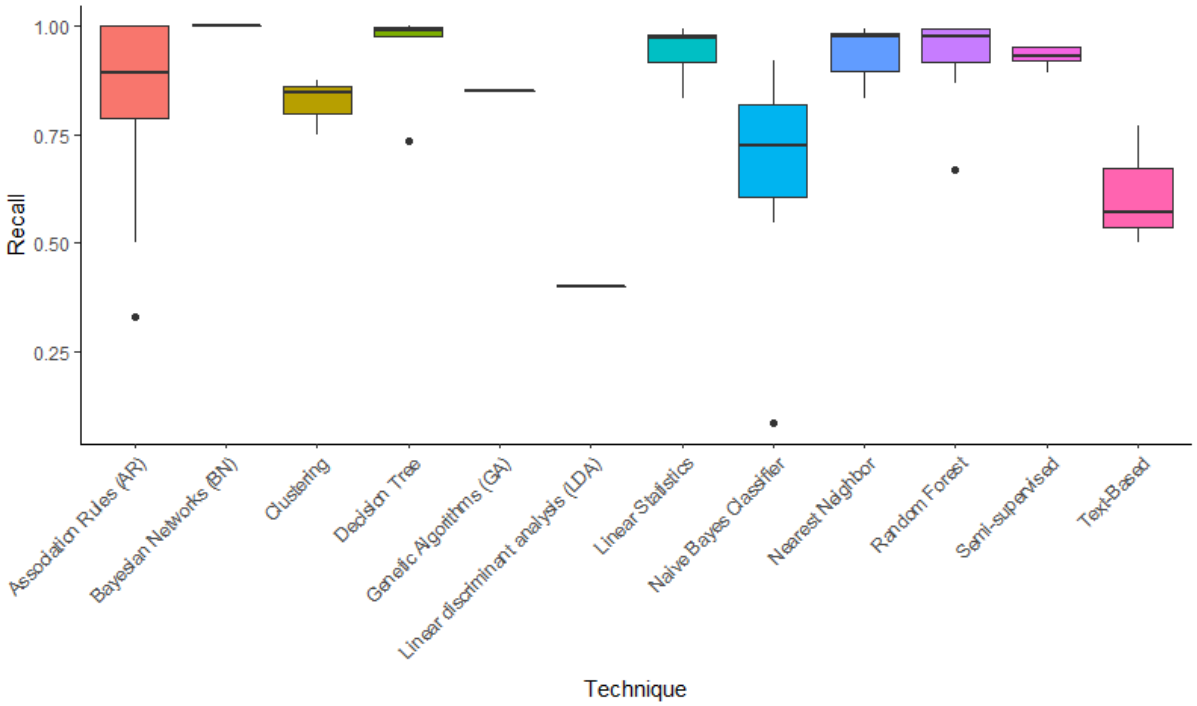
Figure 11 – Precision technique by code smell



Tree, Random Forest, Nearest Neighbor, Linear statistics, Semi-supervised, Genetic Algorithm and Clustering. While the worst performance came from Naive Bayes classifier, Text-Based and Linear Discriminant Analysis, as displayed in Figure 12.

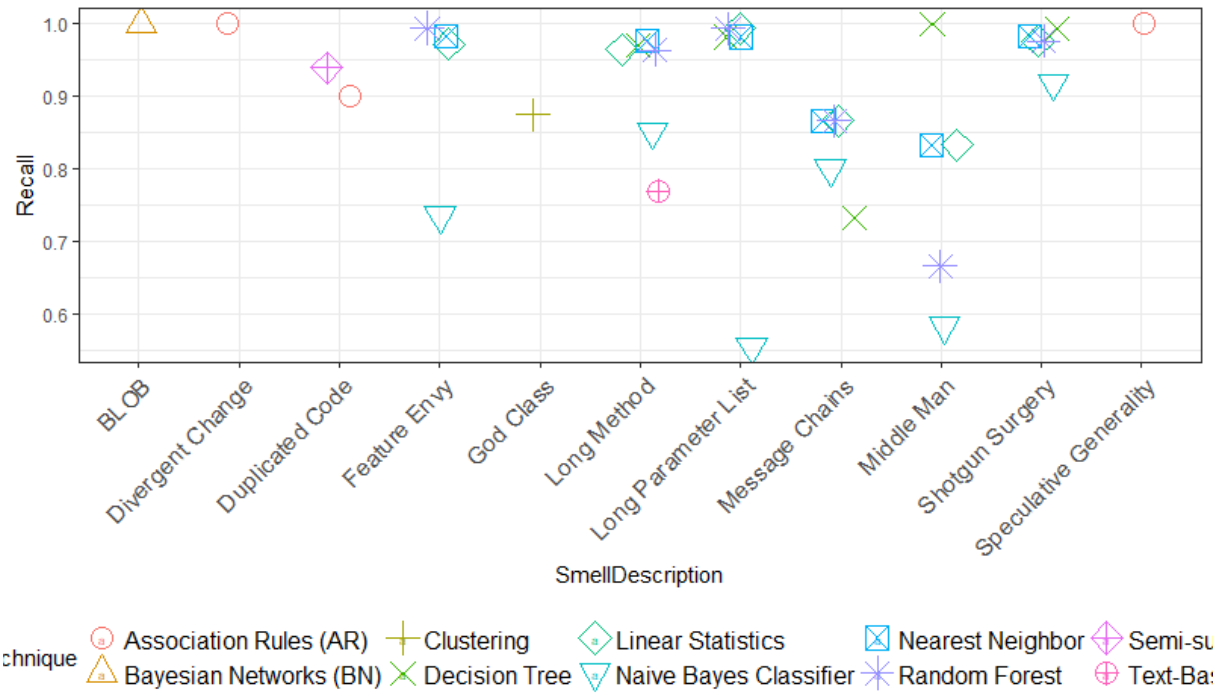
We assessed the technique by smells under a recall perspective as demonstrated

Figure 12 – Machine Learning techniques recall Box-plot



in Figure 13. As occurred in the previous perspective, we have the Naive Bayes classifier performing worst in general. Other techniques that displayed a bad performance were Random Forest for Middle Man, Decision Tree for Message Chain and Text-Based technique for Long Method.

Figure 13 – Recall technique by code smell



2.6 Discussion

This research tried to comprehend the patterns regarding machine learning applied for code smells identification. The study covered the papers in the period of 1999 to 2016, although no paper was published on the matter for about 2 years after the publication of the code smells by Fowler and Beck (1999) and it has been an active research topic for the last 2 years, as shown on Table 2.

By studying the smells it was possible to assert that contrary to Fowler and Beck (1999); Rasool and Arshad (2015); Fernandes et al. (2016); Rattan et al. (2013) that showed the Duplicated Code as the leading smell, the ones using machine learning showed more concern about Feature Envy, BLOB and Long Methods, the latter was also covered by code smell detecting tools as highlighted by Rasool and Arshad (2015). There is increasingly focus on search based techniques, based on genetic algorithm, coinciding with the results found by Rasool and Arshad (2015).

Regarding the techniques, although techniques such as Naive Bayes and Nearest Neighbors have been used more often than the others due to their simplicity, on the overall we did not find any killer technique receiving more attention, corroborating with the study by Fernandes et al. (2016), which found a high variance in the result when comparing the same technique for different code smells. However, when comparing the techniques for each code smell, we found that the following techniques are more used for the following specific smells: Association Rules for Divergent Change, God Class, Data Clumps, Parallel Inheritance, Shotgun Surgery and Speculative Generality; Bayesian Networks for Speculative Generality; Clustering for God Class and Parallel Inheritance Hierarchies; Genetic Algorithms are more used for Spaghetti Code; Linear Statistics for Data Clumps and Semi-supervised for Speculative Generality. We also found that although the machine learning techniques were used for 18 out of the 22 Fowler and Beck (1999) smells, it still covers less smells than the existing smell detecting tools, that as stated by Rasool and Arshad (2015) and Fernandes et al. (2016) on their reviews covered 20 out of the 22 smells.

Comparing the performances we found that on average Decision Tree, followed by Random Forest, had the best performance, agreeing with the experiment by Fontana et al. (2016) which found Decision Tree and Random Forest related algorithms to outperform others on smell identification tasks. Semi-supervised and Nearest Neighbor classifiers also slightly outperformed the remaining techniques, while Text-Based, Linear Discriminant Analysis and Naive Bayes presented the worst performance overall between the studied practices, going against the findings of Fontana et al. (2016); Soltanifar et al. (2016) that found the Bayes approaches performing well for class-related smells. There were also techniques that performed better for specific smells such as Association Rules for Divergent Changes and Speculative Generality, Decision Tree for Lazy class, Middle Man

and Shotgun Surgery smells, Random Forest for Long parameter list and Semi-Supervised techniques for Duplicated Code. When comparing the same techniques, used for the same smells, there is a disparity caused by the different metrics and features selection for the different techniques, given that the performance of the algorithm can only be as good as its input Chakraborty and Joseph (2017). Some techniques provided metrics on the technique level, but did not give information on smell level and were excluded from comparison. In this work we found papers using search-based approach as Genetic Algorithms to improve the automation of refactoring, for this propose they find code smells. For instance, despite the Genetic Algorithm is the technique that more appeared in the researches, but none of papers referenced their results separated by each code smell, difficultly comparisons.

This study also found that the papers lack comparable results, using the same data and performance metrics, a recurring problem in a significant number of studies so far Rattan et al. (2013); Al Dallal (2015); Rasool and Arshad (2015); Fernandes et al. (2016). They also aim at the same smells, showing a high redundancy between the different techniques, the same happened when comparing tools as identified by Rasool and Arshad (2015); Fernandes et al. (2016). This factors turn the comparison of performance metrics between techniques a harder and inaccurate task, making the results less reliable and the studies harder to reproduce.

2.7 Threats to validity

We have selected the search terms according to the research questions, taking into consideration the defined acceptance criteria and have used them to retrieve the relevant studies in the four electronic databases. Although some relevant studies may not use the terms related to the research questions in their titles, abstracts or keywords. We also left out broader terms such as refactoring and anti-patterns on purpose, in order to reduce the noise during the research stage, since terms can be referred to other fields out of code smells, leading to unrelated papers. As a result, we may have the high risk of leaving out these studies in the automatic search process. In order to mitigate this risk we have defined a selection criteria that strictly complies with the research questions to prevent the desired studies from being mistakenly excluded. In addition to that, the decision regarding study selection was made through double confirmation, taking separate selections by graduated researchers and a disagreement resolution for the divergent selections. However, relevant studies may have been missed. If such studies do exist, we believe that the number of them is reduced.

Another threat to validity is that the papers use different projects to assess the results, but even the studies that use the same project use different annotations to train the data, decreasing the reliability of the comparisons of performance. This threat is increased

by publication bias as the researches tend to release only positive results, avoiding the negative ones, and also tend to show that their results outperform the others. In order to mitigate this threat we have registered the baseline that each study used, avoiding the comparison of studies developed on different projects with different baselines.

2.8 Conclusions

This study reviewed 26 papers covering machine learning techniques for code smells identification. For each of these smells we evaluated the studied smells, the techniques and how they relate to each other and the performance of each of these techniques for the code smells.

We found out that the techniques perform close to each other, but Decision Tree, Random Forest, Semi-supervised and Nearest Neighbor techniques had better performance overall, besides the fact that they also tend to be heterogeneous covering smells from different types, due to this fact, the techniques tend to have a high redundancy, without specializing in a single smell. Exceptions for those findings were the Bayes approaches that performed worst than the others in general. The Association Rules and Decision Tree algorithms displayed better usage for smells that involve the relationship between different methods, classes and structures.

We also faced problems regarding the lack of comparability of the studies, since the studies use annotations done by their own personal instead of using a common base, they also did not publish the results using the same metrics, making it harder to compare and assess their performance, reducing the reliability of the study and performance assertions.

3 Methodology

Based on the results of the Mapping Study this study proposes the reproduction of the state-of-art Machine Learning approaches for code smell identification in a standard database. The methods were selected from the Mapping Study based on their performance, where the best performance approaches for each code smell was selected.

In order to accomplish this we used an empirical experimental setup as it is meant to find the result that best perform according to the goal by controlling one or more variables (Easterbrook et al., 2007). The results will be assessed by a quantitative approach, since mathematics and statistical methods were used to test and explain the relationship between the variables (Creswell, 2013). From the objective point of view, the study is exploratory, since this kind of study focuses on the description of the phenomenon being studied (Kothari, 2004) and this study tries to describe the current stage in the usage of ML techniques applied for code smell. The relationship between the study objectives and the used methodology can be seen in table 4.

Table 4 – Objectives X Methods

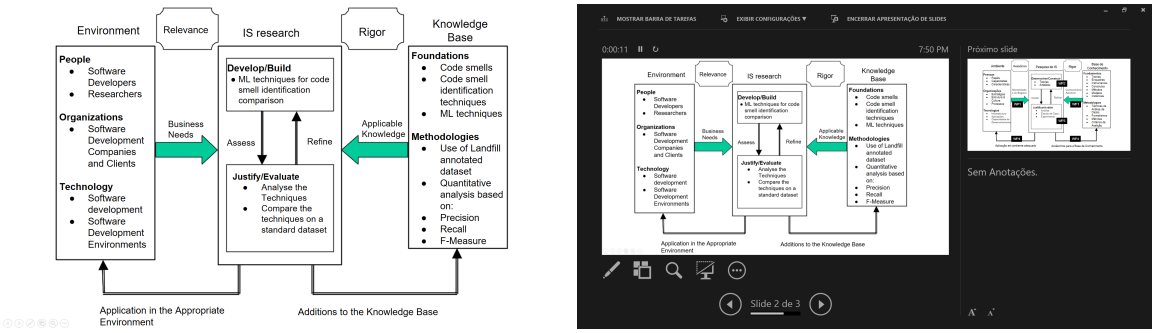
Objective	**Method**
Discover code smells identification techniques that uses ML Techniques	SLR regarding ML techniques for code smells identification
Define a baseline where future studies on the subject can compare their results against.	Empirical experiment comparing ML techniques identified in the literature in a standardized dataset.

3.1 Methods

In order to compare the performance of the machine learning techniques, an empirical experimental was developed in a controlled environment, using the same database, that was split in: training and testing set, these sets will be the same for the different models. So that the only variable that will be changed during the experiment will be the machine learning techniques. Adjustments, such as re-engineering and features extraction were done according to the approaches identified in the literature for the diverse techniques. The research setup can be visualized in the Figure 14

In the following subsections we will detail about the used methods and techniques.

Figure 14 – Research Setup



3.1.1 Empirical Experiment

(Fittkau, 2011). Due to the previously stated fact, for a bulk of the process, tools and techniques in the field, the question "under which circumstances is it better than another" lacks an answer (Juristo and Moreno, 2001). And the same is true for the ML techniques applied for code smells detection, in order to fill this gap we propose a empirical experiment to compare ML techniques for code smell identification in a highly imbalanced dataset.

The main feature that is common to the experiments is varying something with the intention to discover what happens to something else (Basili, 2007). In this study the techniques and code smells variables will vary, while keeping the source code variables state. With this, we intend to figure out the impact that the techniques can cause in the code smell identification performance. Since the units will not be assigned randomly, the experiment type will be a quasi-experiment.

3.2 Used dataset

In order to compare the ML techniques for code smell identification, a common code should be used to guarantee a controlled and reproducible environment. It is also important for future replicability for the code base license to be opened for any researcher, and it also have to keep track of previous version, since a change in the version can affect the replicability. For that end the Landfill Data Set will be used, since it is a collection of software systems intended to be used for empirical studies of code smells, providing a resource that supports reproducible studies of software.

The corpus is composed of 2 datasets, each is composed of 20 open source systems, with eight annotated code smells, but since two of them are test smells which are out of the scope of this experiment we will use the six remaining: Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, Long Method and Feature Envy (Palomba et al., 2015b). Since the datasets only contains the positive annotations (the classes, methods or relationships which has smells), we will work with it in a positive / unlabeled setup,

where the unlabeled part of the set will be the code that the parts of the code that is not annotated as positive. The descriptive statistics for the data in the dataset can be found in table 5.

Table 5 – Basic descriptive stats from the smells

Smell	Projects	Smells By Project	Unlabeled By Project	Ratio By Project	Deviation	% Deviation
Large Class	36	6.86	989.22	0.0120	0.0218	182%
Long Method	19	21.58	753.26	0.0270	0.0361	134%
Feature Envy	27	4.67	2,659.96	0.0059	0.0089	151%
Parallel Inheritance	7	2.71	645.00	0.0173	0.0306	177%
Divergent Change	9	1.11	1,203.78	0.0024	0.0024	102%
Shotgun Surgery	5	2.80	1,498.80	0.0036	0.0056	156%

3.2.1 Benchmark Techniques

To check the performance of the ML techniques we will apply and measure it in a standardized dataset, that can be used by future studies. Based on our Mapping Study results, we selected the best performing techniques for each code smell, but since we are relying on the Landfill Dataset to run the experiments, we will use only the smells that it supports, as can be seem on table 6.

Smell	Techniques
Divergent Change	Association Rules
Feature Envy	Decision Tree, Random Forest and Naive Bayes
Large Class	Decision Tree, Random Forest and Naive Bayes
Long Method	Decision Tree, Random Forest and Naive Bayes
Parallel Inheritance	Association Rules
Shotgun Surgery	Association Rules

Table 6 – Selected techniques by smell

3.2.2 Results Comparison

It is important to establish a clear comparison and analysis criteria for code smell detection tools results. Since replication play an important role for establishing benchmark systems (Shull et al., 2008). It is a hard task finding common tools that performed experiments on common systems for extracting common smells, as the techniques performed experiments on different systems and present their results in different formats. In general, the publications about tools present the number of detected smells, and one cannot exactly view which class, method, or artifact of source code is a cause of a particular code smell (Rasool and Arshad, 2015).

Furthermore, even though the precision, recall and f-measure are seldom calculated, it was the leading metric identified in the Mapping Study presented in the previous chapter. These are also used statistical methods commonly used in quantitative studies, particularly when imbalanced positive/negative ratios are present, providing an appropriate way to quantify and assess the reliability of a gold standard in these studies (Hripcsak and Rothschild, 2005).

3.3 Tools

Since the Landfill Data Set that will be used as the Sample Data and there is an abundance of studies based on the Java language, this will be used as the base language for this study. The Java Programming Language platform provides a portable, interpreted, high-performance, simple, object-oriented programming language and supporting run-time environment while keeping a simple and friendly syntax (Gosling and McGilton, 1995). For the ML techniques replication and execution we will use the Python language, since it is currently the most used language for ML related tasks, it has a huge community supporting and constantly updating the frameworks and packages. From these we used the sklearn framework for the development of the ML models, we also used pandas and numpy for data handling and manipulation, for the under/oversampling technique imblearn framework was used. We also used the XGBoost, CatBoost and LightGBM python packages.

4 Results

4.1 Introduction

Code smells, also known as code bad smells, are "a surface indication that usually corresponds to a deeper problem in the system" Fownler (2016). Introduced by Fowler and Beck (1999) in 1999 where the author conceptualize each of them and also provide some guidance on refactoring them, since then their impact on software maintainability and flexibility were targeted by many studies Mens and Tourwé (2004); Olbrich et al. (2009). But even though their concepts are clearly defined, their identification is still subjective to the developer's interpretation Fowler and Beck (1999). Studies found that even among experienced developers working in the same application the existence of a given code smell may not be a consensus Bryton and Abreu (2009); Fontana et al. (2016); Hozano et al. (2017). The manual detection of code smells is time consuming, non-repeatable and does not scale so it could benefit from the usage of automated approaches for code smell identification Marinescu (2004).Many other studies on the automated identification were proposed Fontana et al. (2012); Fokaefs et al. (2007); Mantyla et al. (2004); Rasool and Arshad (2015), but even though they make it easier to identify code smells, they still fail to bring context, domain, size and design of the system to the identification Ferme et al. (2013). Given this context, machine learning based techniques can bring some more flexibility Kotsiantis (2007).

There are tools Fernandes et al. (2016) and techniques Rasool and Arshad (2015) for the code smell detection in literature, helping the developer to identify potential flaws that he or another developer may have missed initially. Even though the usage of machine learning based techniques is recent and consequently has less studied and tested techniques, it is growing steadily. But we still lack comparable results that allows us to identify which one should be used for each smell, this is aggravated by the subjectivity of the code smells so that even for the same system the smells selected by the developers are very likely to differ from one another Palomba et al. (2015b). Another common issue is that most of the experiments rely on a Positive/Negative set that usually contains a high smell ratio Maneerat and Muenchaisri (2011); Fontana et al. (2013); Fu and Shen (2015); Palomba et al. (2015a); Fontana et al. (2016), which doesn't reflect a realistic ration and may mislead the results.

In order to address these problems, this study aims to developing the state-of-art machine learning techniques in a standardized dataset that reflects a real project scenario in order to create a benchmark for future work. For that end, we developed an experiment based on the machine learning techniques for code smells identification that

better performed in literature. For this we will use a public dataset named landfill Palomba et al. (2015b) composed of 2 databases that sums 1770 annotated smells spread across 8 different types of smells. The usage of an open and standardized data also contributes to the replication of the study and consequent comparison of the used methods. Since it only provides positive examples of annotated smells, this represents the machine learning problem as a positive-unlabeled problem, this setup represents more accurately the code smell identification problem, since it has a more realistic smell ratio. We used it as a unlabeled instead of a negative set because it is not feasible to manually annotate the whole code and it is unlikely that the code contains no further smells.

The study resulted in the development of a reproducible and open source algorithm, implementing the state-of-art technique identified in the literature, as well as some technique that were proven to work better under this experiment setup. We found that Boosting and Ensemble models proved to work better for this experiment than the ones identified on literature, but none of the technique were able to achieve results as good performing as the obtained as the original papers.

This paper was organized according to the following structure: Section 2 presents works related to this project; Section 3 provides a background about the code smells and machine learning techniques necessary for this experiment; Section 4 addresses the methodology used in this work; Section 5 displays the results of the study; Section 6 discusses the results; Section 7 presents the threats posed to the validity of the study and finally Section 8 shows the conclusion and provides suggestions for future work.

4.2 Related work

On their original proposition Fowler and Beck (1999) provides 22 heuristics for the identification and refactoring of code smells, which would become the most common conceptualization of software design that may hinder the development and maintenance of the system Marinescu (2004). The code smells identification should be subjective to the developer's interpretation Fowler and Beck (1999). But Marinescu (2004) counter arguments that manually identifying the code smells is a time consuming, non-repeatable and non scalable task, proposing that automated identification may improve the task based on code metrics. Independently on the detection approach used to identify the code smells the human interpretation is still necessary Fontana et al. (2016), since code smells are hints for a design problem but do not necessarily represent a design problem Fownler (2016). So the manual identification is still the best approach, as it eliminates uncertainties of the process, however, due to its human-centric aspect, it is time-consuming and error-prone, so it should be helped by automated approaches Counsell et al. (2010).

Yamashita and Moonen (2013) argues yet that the automated approaches are

important since it is common for the developers to be unaware of their presence in the code. Shatnawi and Li (2008) shows a relationship between code metrics and qualitative metrics such as code smell and class error tendencies. Khomh et al. (2009a) also states that code smells have a negative impact on classes maintainability aspects. Jaafar et al. (2016) demonstrates that classes related to anti-patterns exhibit more defects than those which are correlated to design patterns.

Even though there is empirical support for code-based metrics Ferme et al. (2013) found that these approaches rely on fixed thresholds to determine the presence or absence of code smells, but those thresholds are usually empirical and unreliable, hurting the performance of the algorithms. Machine learning can be used to reduce this uncertainty, but are dependant on large amounts of human-annotated data and their quality Rasool and Arshad (2015). But in practice, until this moment it is unclear which presents the best performance, but what is certain about the Machine Learning approach that it reduces the cognitive load required from the engineers, since it does not require them to define the roles and thresholds Fontana et al. (2016).

One of the first works using Machine Learning techniques were developed by Kreimer (2005) which uses Decision Trees algorithms to identify Large Classes and Long Methods smells, Khomh et al. (2009b), Kosker et al. (2009) and Khomh et al. (2011) relies on Bayesian approaches to identify anti-patterns, but while the first one uses a Bayesian Network approach, the second one uses a Naive Bayes and the last one uses "Goal, Question and Metric" in combination with the Bayesian approach. Fontana et al. (2013) uses the following WEKA algorithms: Support Vector Machines (SMO, LibSVM), Decision Trees (J48), Random Forest, Naıve Bayes and JRip, for the identification of Long Method, Large Class and Feature Envy extracted from projects from Qualitas Corpus, this work is further evolved by Fontana et al. (2016) where it uses 16 different WEKA algorithms and also adds variations of these algorithms with boosting techniques. Fu and Shen (2015) and Palomba et al. (2015a) uses Association Rules approaches based on the code repository history, while the first focuses on Speculative Generality and Divergent Change smells the second focuses on Divergent Change, Feature Envy, Parallel Inheritance and Shotgun Surgery.

All these works contributed for the discussion and for the development of this topic, but a mapping study that was conducted previously and that is still under review under International Journal of Software Engineering and Knowledge Engineering (IJSEKE) one of the hardships found in the code smells mapping study is that there is a lack of comparable results, each study uses a different dataset with different settings, what reduces the reliability of the results. Other factor, is that only a small part of the code, the labeled part, is used for training and testing the model, the bigger part that is unlabeled is simply discarded. Even though it is a common practice, it may be subject

to a selection bias, since developers may labels easier detectable smells. Our work differs from them by tackling these two detected gaps, we use Landfill Palomba et al. (2015b), an open dataset that contains 8 annotated different kinds of smells and can be easily reused by another experiments and both the labeled and the unlabeled code will be used for the model development.

4.3 Background

The Landfill dataset contains smells annotations for 8 different types of smell Palomba et al. (2015b), but since this work is focused on objected-oriented smells we discard two test-related smells: Eager Test and General Fixture. So we will focus on the remaining smells: Divergent Change, Feature Envy, Large Class, Long Method, Parallel Inheritance and Shotgun Surgery and the machine learning techniques proposed for each of them.

4.3.1 Code smells

4.3.1.1 Code Smells definition

Code smells represents design choices that may lead to a future degradation on maintainability, understand-ability and changeability of a given part of the code Fowler and Beck (1999), but even though smells are hints that the code may present a design problem it does not necessarily indicates one Fownler (2016). Below the definition of each of the selected code smells are described, we will use Mantyla et al. (2003) taxonomy to classify each of them, but will also classify them by the code entity they affect and the quality aspects they affect as defined by Marinescu (2005):

Divergent Change

The divergent change smells represent classes that change together whenever another class changes, classes which have this smell usually demonstrates a high coupling, hurting the low coupling high cohesion design guideline. It is classified as a change preventer class and affects inter-class interactions.

Feature Envy

A method that is more interested in other properties of the classes than in the ones from its own class. This kind of smell affects the coupling, cohesion and encapsulation design aspects of the system, representing a problem in the abstract design of the system. It is classified as coupler smell and affects method/property entities.

Large Class

A class that tries to do a load of things, having plenty of instance variables or methods. It is similar to the Blob/God class anti-pattern Brown et al. (1998). A

class with this smell tends to present coupling, cohesion and complexity problem, affecting the maintainability and understand-ability of the class. It is classified as a bloatter smell and affects class entities.

Long Method

A method that is so long that it is hard to understand, change or extend. It also increases the complexibility of the system. It is classified as a bloatter smell that affects method level entities.

Parallel Inheritance Hierarchies

A situation where two parallel class hierarchies exist and are related. It's quality aspects were not defined on Marinescu (2005), but by it's nature it affects the abstraction design of the system. It is classified as an object-orientation abuser and affects inter-class inheritance relationships.

Shotgun Surgery

The shotgun surgery smell exists when changing a given class requires a consequent change on other classes that depends on it, classes with this smell usually has a low cohesion, hurting the low coupling high cohesion best practice. It is classified as a change preventer class and affects inter-class interactions.

4.3.2 Machine Learning

Machine learning techniques can be categorized in three ways: supervised, unsupervised and semi-supervised, all of them take features as input, these features used may be categorized as continuous, categorical or binary, depending on their nature Kotsiantis (2007). If instances are given with known labels (the human annotated correct output) then the learning is called supervised, otherwise, when the instances are unlabeled, it is unsupervised learning Jain et al. (1999). There is also a hybrid approach, which is the semi-supervised learning that uses both labeled and unlabeled data to perform an otherwise supervised learning or unsupervised learning task Zhu and Goldberg (2009). This work will focus on the supervised techniques, since we use labeled smells data and most of the experiments use this approach Fernandes et al. (2016); Kotsiantis (2007). During this section we will give a brief explanation of the techniques used on the experiment.

- **Association Rules:** Association rules is a rules-based technique that aims at identifying relationship between variables that exists in the dataset.
- **Decision Trees:** Decision Trees classify instances by sorting them based on feature values and splitting them into branches, each branch represents the value thresholds the contained nodes can assume and each node represents a feature. Instances are

classified starting at the root node and then sorted based on the features value and .

- **Random Forest:** Random Forest is a tree-based ensemble technique, which uses a bagging of trees models, built using only a subset of the features, the average of those trees is taken to calculate for the features prediction.
- **Logistic Regression:** Fits a sigmoid function in a linear regression model for binary classification, the function determines the classification probability of each instance and based on a probability threshold determines if it is classified as positive or negative.
- **Neural Networks:** A multi-layer neural network consists of large number of units (neurons) joined together in a pattern of connections, each connection has an weight that is established by the model, and each node contains an activation function that determines the node value. It usually has one output node for each class that are usually defined by a sigmoid function.
- **Naive Bayes:** Naive Bayes uses Bayesian Statistics to establish the probability between one unobserved node and a chain of children observed nodes. It assumes an independent relationship between child nodes and their parent.
- **Nearest Neighbors:** Nearest Neighbors, also known as k-nearest neighbors is a lazy-learning algorithms classifying algorithm that classifies the items based on their position and distance in a hyper-plane.
- **Boosting Techniques:** Boosting techniques are classifiers which uses the combination of a set weak learners to build a stronger one, with lower-bias and variance. Those classifiers usually perform well under scenarios where there isn't enough data to train a more complex classifier such as a deep neural network. Some of the most prominent boosting techniques currently are XGBoost Chen and Guestrin (2016), LightGBM Ke et al. (2017) and CatBoost Dorogush et al. (2017) which were recently developed and are known to be used by the winners of multiple Kaggle contexts.

4.4 Experiment Setup

The study performed an empirical experiment on the usage of machine learning techniques for code smells identification. Empirical Experiments uses empirical studies to build and produce a theory or a model and demonstrate that it is usable from a practical perspective Zelkowitz and Wallace (1998). In order to compare the performance of the machine learning techniques, an empirical experimental will be developed. The experiment will be conducted in a controlled environment, using the same database, that will be split

in: training and testing set, these sets will be the same for the different models. So that the only variable that will be changed during the experiment will be the will be the machine learning techniques. Adjustments, such as feature-engineering and extraction may have to be done in the sets to be usable with the diverse techniques, but they shall be done in a way that do not affect the outcome of the project.

4.4.1 Experiment Design

To help in the definition of the scope we defined a goal, according to the baselines defined by Wohlin et al. (2012): Analyze the Machine Learning Techniques for Code Smells identification for the purpose of evaluation with respect to their effectiveness and efficiency from the point of view of a researcher in the context of a standardized environment that can be replicated and compared by future experiments.

4.4.2 The smells dataset

For the purpose of replicability, we used a public dataset named landfill, since it constitutes the largest collection of manually validated smells publicly available as of today Palomba et al. (2015b). It is composed of 2 databases, each contains around 20 different projects with annotated smells, summing up to 1770 annotated smells spread across 8 different types. Since this work is focused on objected-oriented smells we discard two test-related smells: Eager Test and General Fixture. What leaves 6 smells remaining: Divergent Change, Feature Envy, Large Class, Long Method, Parallel Inheritance and Shotgun Surgery.

We downloaded the project snapshot for each project contained in the Dataset and used the Metrics Reloaded Leijdekkers (2017) Idea IntelliJ plugin to download the Metrics required for each technique, for the metrics that were not supported by the plugin, we used the tool used in the given experiment. For the Association Rules smells, we used the change history provided by the same authors of the Landfill Dataset that used them on the history mining study Palomba et al. (2015a), and extracted the Association Rules from them. As the experiment was setup to be a Positive-Unlabeled problem, we merged the whole metrics data with the annotated smell, the ones that matched were classified as Positive, while the remaining metrics were classified as Unlabeled. It is important to call-out that we weren't able to match all the corpus smells with it's metrics counterpart, the main reason for that was that some annotations did not follow the same name convention, there were also some duplicated annotations for the same smell, which were removed to leave just unique annotations.

4.4.3 Code Smells detection strategy

A comprehensive number of strategies for the detection of code smells on the source code is present on the literature, as the focus of this work is on Machine Learning techniques, we will focus on the best strategies found on literature using Machine Learning techniques for each of the covered smells.

Divergent Change and Shotgun Surgery

The top performing article for this smell uses the change history in the code repository to identify how often the classes changes together with others. An association rules (Apriori) technique was used to identify the support, confidence and lift in the classes that presents changes on the same commits as described on the experiment done by Palomba et al. (2015a).

Feature Envy and Long Method

The article which presented the best performance in the literature Fontana et al. (2013) uses the following IPlasma Metrics Marinescu et al. (2005): Size metrics – measure the size of the analyzed entity; Complexity metrics – measure the complexity of the analyzed entity; Coupling metrics – measure the data coupling between entities; Cohesion metrics – measure the cohesion of classes and Fluid tools metrics Nongpong (2012): Depth of Inheritance Tree (DIT); Number of Children (NOC); Weighted Methods per Class (WMC); Afferent Coupling (Ca); Efferent Coupling (Ce); Lack of Cohesion in Methods (LCOM*);McCabe's Cyclomatic Complexit.

Large Class

The dominant article on this subject in the literature Fontana et al. (2013) also uses IPlasma Metrics such as: Size metrics – measure the size of the analyzed entity; Complexity metrics – measure the complexity of the analyzed entity ; Coupling metrics – measure the data coupling between entities; Cohesion metrics – measure the cohesion of classes but adds some PMD Rules Pmd (2013).

Parallel Inheritance Hierarchies

Similar to the Divergent Change and Shotgun Surgery detection, this smell is also detected using the code repository change history and uses association rules (Apriori) to identify the support, confidence and lift. But instead of checking for classes that presents changes in the same commits it checks for super-classes that changes together when a subclass is added. For the other techniques, we kept the association rules variables (confidence, support and lift) as features, since they provide valuable information, but instead of using a hard threshold we fit them to the model.

4.4.4 Evaluated models

Based on the smells supported by the Dataset we selected the best performing technique for each of them based on the literature review submitted ot IJSEKE, the smells and the respective associated technique can be seem on the table 7.

Smell	Techniques	Reference
Divergent Change	Association Rules	Palomba et al. (2015a)
Feature Envy	Decision Tree, Random Forest and Naive Bayes	Fontana et al. (2013)
Large Class	Decision Tree, Random Forest and Naive Bayes	Fontana et al. (2013)
Long Method	Decision Tree, Random Forest and Naive Bayes	Fontana et al. (2013)
Parallel Inheritance	Association Rules	Palomba et al. (2015a)
Shotgun Surgery	Association Rules	Palomba et al. (2015a)

Table 7 – Best performing technique for each smell

For each of the smells we also developed models that were recommended by the literature for Positive-Unlabeled problems, such as: One class classifiers (Such as One class SVM or Local Outlier Detection), Boosting (Random Forest, XGBBoost, LightGBM, CatBoost) and Ensemble (multiple ML techniques with a soft voting classifier) Khan and Madden (2014), a weight adjustment proposed by Elkan and Noto (2008) and a under/oversampling technique. Since Random Forest was recommended by the article and is also a Boosting model, it fits into two categories, so when it appears as recommended by the Article we will classify it as Article recommendation instead of as a boosting model.

4.4.5 Assessing the models

For the models assessing we used the precision, recall and f-measure metrics Powers (2011), which are commonly used for classifications with unbalanced positive/negative rate. Once the unlabeled sample contains both positive and negative examples, we used the default contingency table for the labeled data, while we used an adapted contingency table for the unlabeled data as proposed by Claesen et al. (2015) as shown below where β is the estimated number of smells in the unlabeled data, fpr stands for the false positive rate, r for recall and U for the unlabeled samples:

- **True positive (Unlabeled)**: $\beta * U * r$
- **True negative (Unlabeled)**: $(1 - \beta) * U * (1 - fpr)$
- **False positive (Unlabeled)**: $(1 - \beta) * U * fpr$
- **False negative (Unlabeled)**: $\beta * U * (1 - r)$

4.4.6 Research questions

In this subsection we describe how each of the research questions were addressed by the experiment

- **How does the baseline models perform on the selected dataset?** To answer this question, the original experiments were replicated in our dataset, but instead of using them for a Positive/Negative learning problem, we used them on a Positive/Unlabeled problem. We also tested them using some imbalanced data treatment, such as the combination of oversampling and under-sampling techniques and the class weighting adjustment proposed by Elkan and Noto (2008). We took the precision, recall and f-measure with the contingency table proposed by Claesen et al. (2015) also considering a confidence-interval for the smell-proportion.

- **How the techniques recommended for positive/unlabeled settings perform when compared to the recommended techniques?** For this question algorithms recommended for positive/unlabeled learning problems were applied for each smell, using the same setup and measurement as used for the question above. We then compared the results with the best performing code smell identification techniques.

4.5 Results

In this Section we present the results of the empirical experiment. Firstly, we will demonstrate some descriptive statistics of the used dataset, after that we will show the result of the literature selected models and finally we will show the results of techniques recommended for Positive-Unlabeled learning and compare with the baseline results.

4.5.1 Overview

To understand how the smells are distributed the dataset, we present a descriptive statistics on table 8. It is possible to notice in this table that the dataset is highly unbalanced, the smells represents less than 1% of the set for most of the smells. The smell ration also variate a lot from project to projects, as demonstrated by the Deviation, the % Deviation for all the smells is above 100%. This difference may occur due to the differences in the quality of the smell, but also due to the project size or the developers knowledge and experience in the given project Marinescu (2004).

In order to calculate a more precise f-measure considering the unknown values in the unlabeled set we have to define the Beta value. It is important to have a well defined β to avoid misleading results Claesen et al. (2015). Since we don't have any prior empirical

Table 8 – Basic descriptive stats from the smells

Smell	Projects	Smells By Project	Unlabeled By Project	Ratio By Project	Deviation	% Deviation
Large Class	36	6.86	989.22	0.0120	0.0218	182%
Long Method	19	21.58	753.26	0.0270	0.0361	134%
Feature Envy	27	4.67	2,659.96	0.0059	0.0089	151%
Parallel Inheritance	7	2.71	645.00	0.0173	0.0306	177%
Divergent Change	9	1.11	1,203.78	0.0024	0.0024	102%
Shotgun Surgery	5	2.80	1,498.80	0.0036	0.0056	156%

study which quantifies the average percent of smell sin the population, we will use the number of labeled and unlabeled features for each kind of smells, and the consequent average smell percent considering the percentage of each project and use it as β, as well as its 95% confidence interval that was used to calculate the f-measure CI. It is important to make clear that the ratio is just an approximation since the ratio of smells in a given projects can greatly vary from project to project as shown in 8. The ratio used for each kind of smell can be found on the table 9.

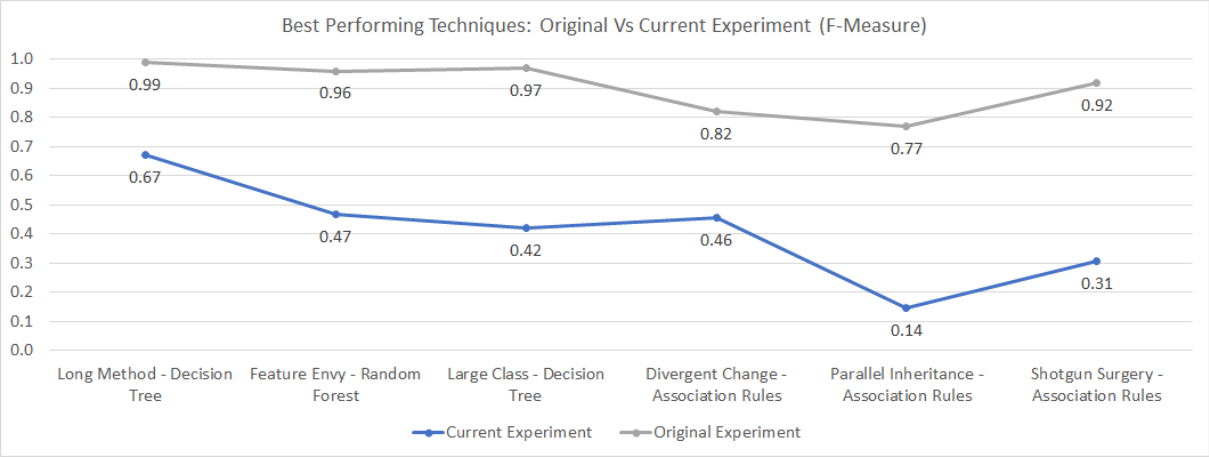
Table 9 – Smell ratio

Smell	Positive	Unlabeled	Ratio	Projects Average Ratio (β)	Confidence Interval
Long Method	410	14312	0.0278	0.0270	[0.0092 , 0.0449]
Feature Envy	126	71819	0.0018	0.0023	[0.0059 , 0.0095]
Large Class	247	35612	0.0069	0.0045	[0.0120 , 0.0195]
Shotgun Surgery	14	7494	0.0019	0.0036	[0.0000 , 0.0113]
Divergent Change	10	10834	0.0009	0.0024	[0.0004 , 0.0043]
Parallel Inheritance	19	4515	0.0042	0.0173	[0.0000 , 0.0479]

4.5.2 How does the baseline models perform on the selected dataset?

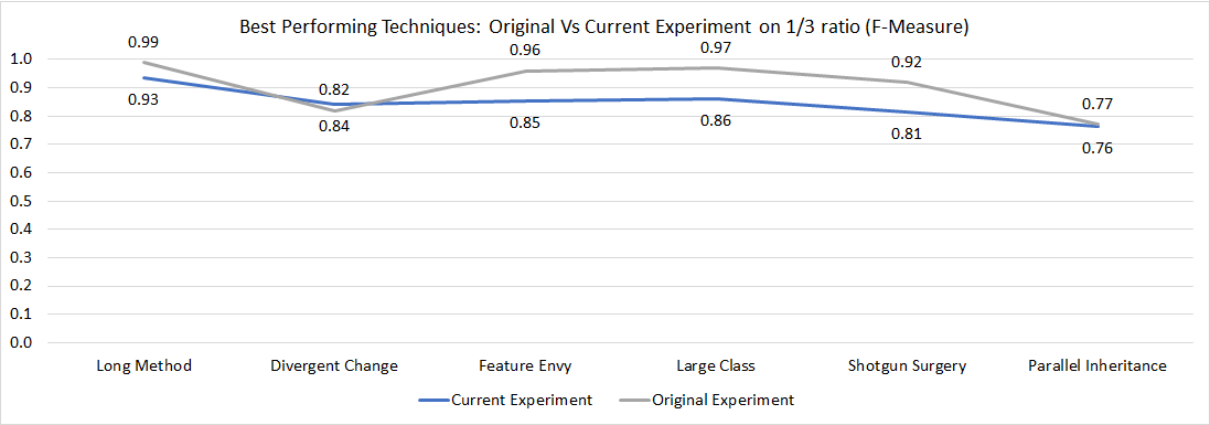
We replicated the best performing studies found in the literature for each smell in the dataset we prepared for this experiment, the results of the replicated results were then compared to the results reported on the original experiment as can be seem on figure 15. There is a noticeable difference between the original results and the replicated techniques for every smell. The model which performed the best in our replicated experiment was the Decision Tree used for the Long Method detection, but it still performed 34% worst than the original experiment, while most were outperformed by more than 50% and some performed even 86% below the original experiments.

Figure 15 – Original x Current Experiment (F-Measure)



To help understand how much the performance difference is impacted by the divergent smell ratios, we performed an experiment using a Near Miss undersampling technique to emulate a 1/3 smell ratio in the dataset, the results can be seem on figure 16. Even though some smells still perform worst then the baseline, it is possible to notice that the results are more similar to it, the biggest variation occurred for shotgun surgery with 11%.

Figure 16 – Original x Current Experiment (F-Measure) with 1/3 smell ratio



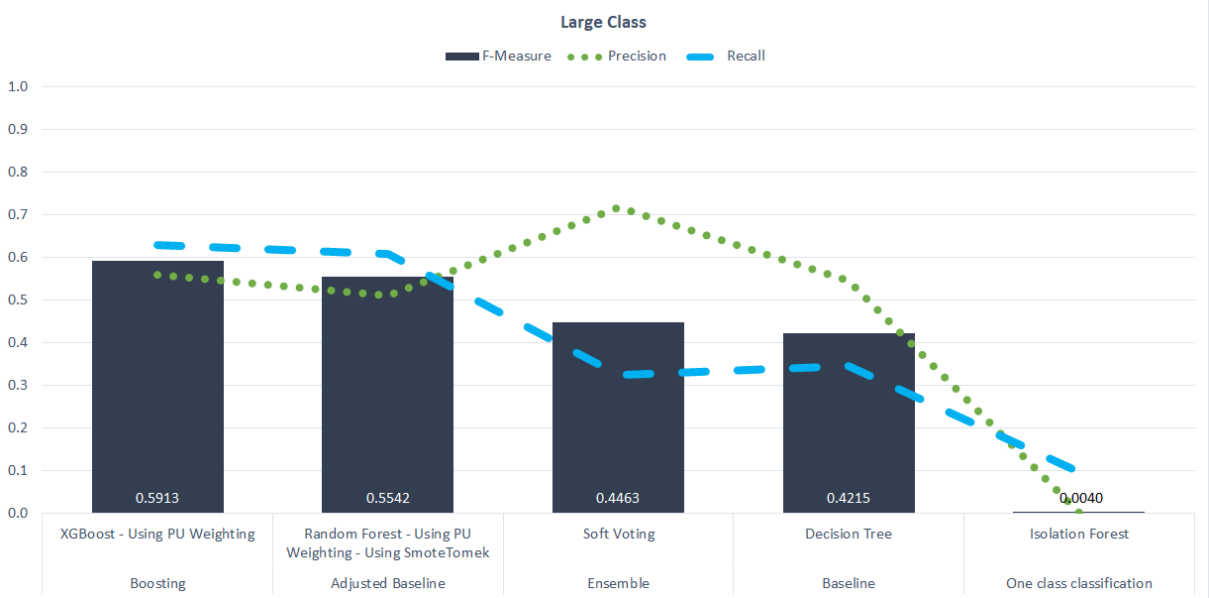
4.5.3 How the techniques recommended for positive/unlabeled settings perform when compared to the recommended techniques?

For each of the studied smells we tested a set of approaches for highly unbalanced data: One class classifiers (Such as One class SVM or Local Outlier Detection), Boosting (Random Forest, XGBBoost, LightGBM, CatBoost) and Ensemble (multiple ML techniques with a soft voting classifier). We also tried them with and without under/oversampling combination technique and the Positive/Unlabeled weight adjustment.

For the charts below we selected only the best performing techniques of each category, the whole experiment result can be found in Appendix B.

Regarding the Large Class smell, the best performing technique was the XGBoost, it outperformed the recommended technique by 0.17 (40%), it presented a better recall than the other method without losing too much on the precision. The usage of under/oversampling technique also improved the recommended technique by 0.13 (31%), it had a small loss on the precision, but it was compensated by a huge gain on the recall, while the Ensemble model only provided a small gain of 0.02 (5%), with a much better precision than the baseline, but a slightly worst recall. The isolation forest classifier performed below the baseline by 0.25 (59%), with an extreme precision but a very low recall. The chart with the Large Class results can be seem on figure 17

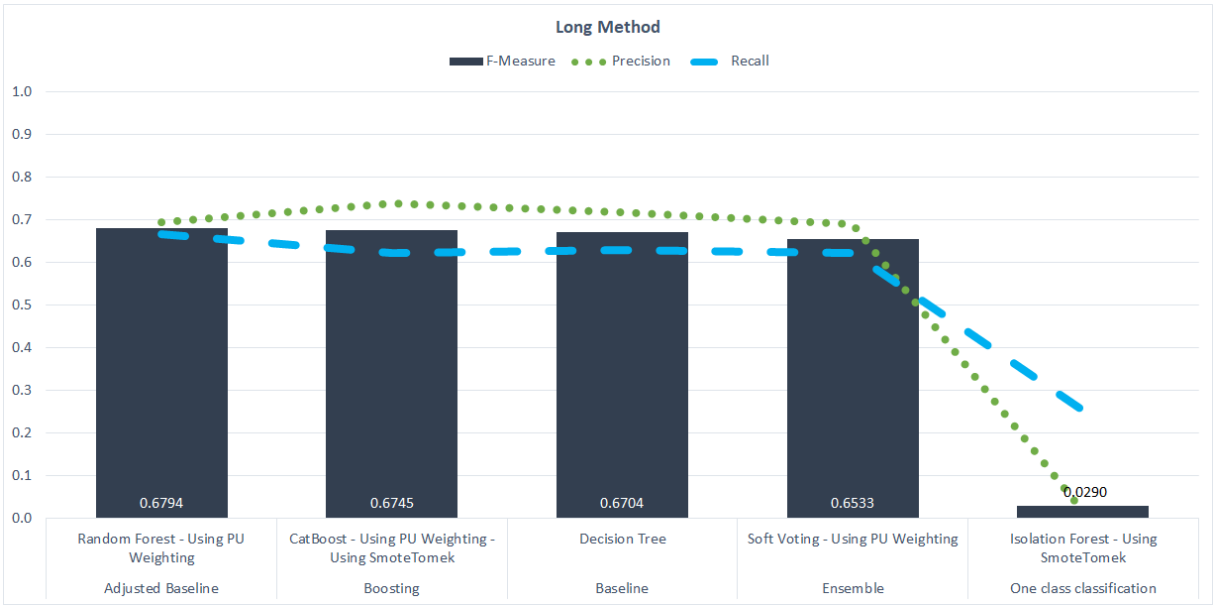
Figure 17 – Large class results



The long method smell presented a very small variance on the results for most of the techniques. The baseline technique adjusted with a PU weighting was the best performing one with 0.009 (1.3%) above the baseline, it presented a very similar precision and recall balance. It was followed by the CatBoost, which performed only 0.004 (0.6%) above the baseline, it had a higher precision than the baseline but was penalized on the recall. The ensemble model in other hand presented a slightly worst result, performing 0.015 below the baseline (2.2%), with almost the same recall but a worst precision. Again, the worst model goes for the Isolation Forest, which performed 0.26(39%) below the baseline, with an extreme precision but a small recall. As shown in the figure 18

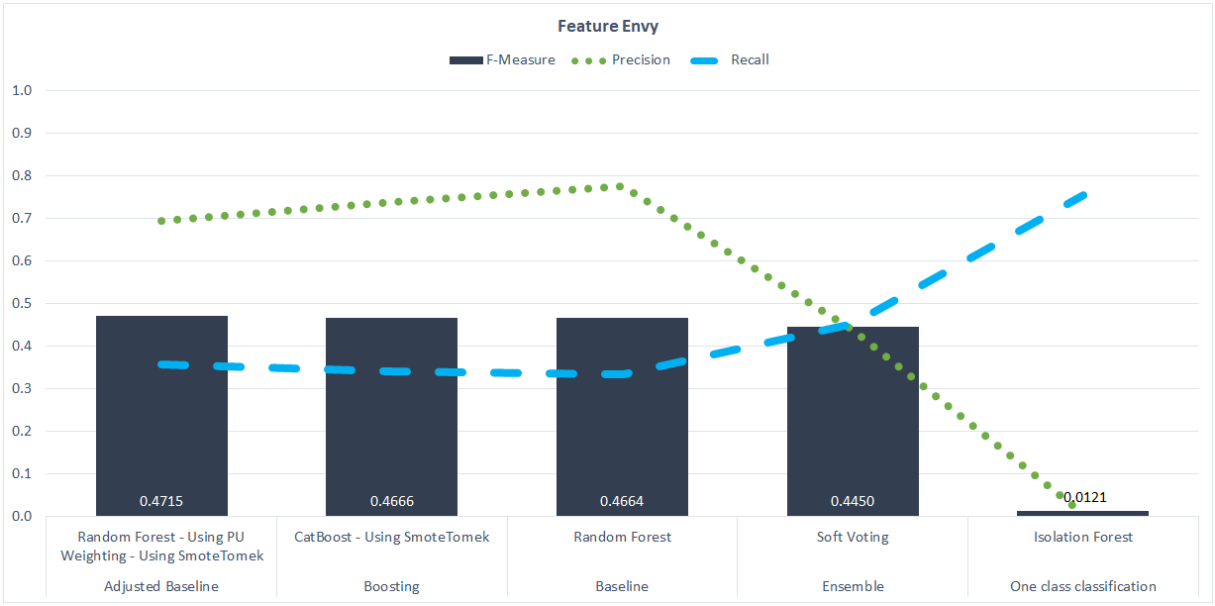
The feature envy also had just a small variance when compared to the Article recommended technique, the best performing technique was the random forest recommended by the baseline but using PU weighting adjustments and over/under sampling techniques which performed better than the baseline by 0.005 (1%) with a better recall

Figure 18 – Long method results



but worst precision than it, the second best performance, which was CatBoost performed just slightly better than the baseline, with a better recall and worst precision as well it score 0.0002(0,045%) below it. The Ensemble model didn't perform well, scoring 0.021 (4%) below the baseline.

Figure 19 – Feature Envy results

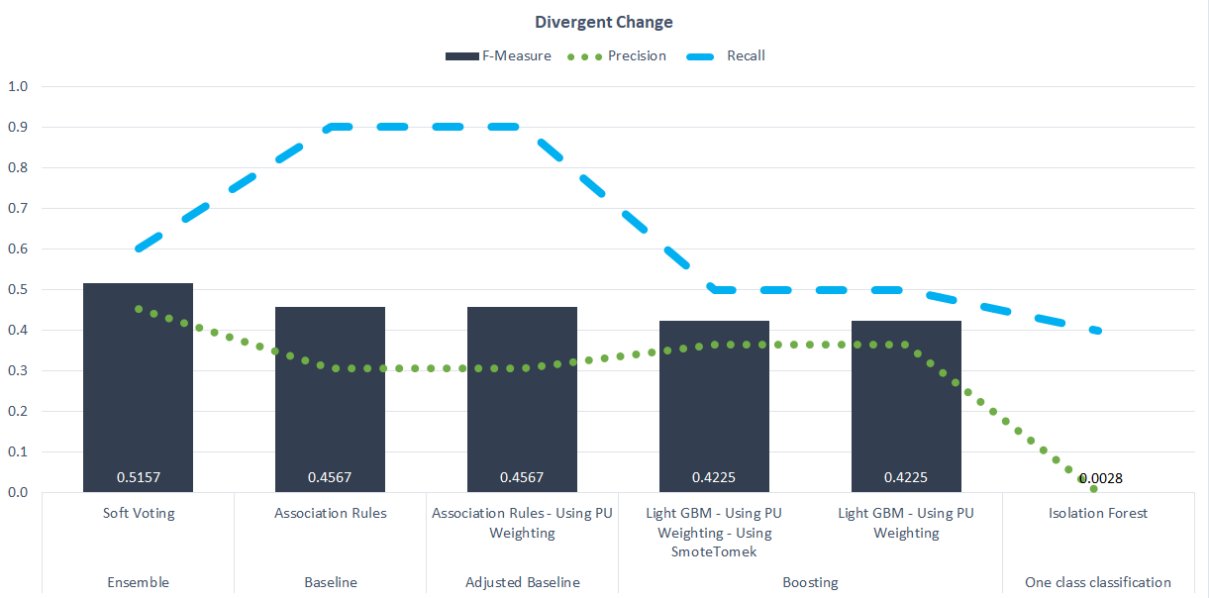


For the next three smells: Divergent Change, Shotgun Surgery and Parallel Inheritance, given that in the original experiment they use a hard threshold to define the smell, for the calculation of the adjusted model with PU weighting and under/oversampling techniques we used these technique for recalculating the threshold.

For divergent change smell the Ensemble model demonstrated the best perfor-

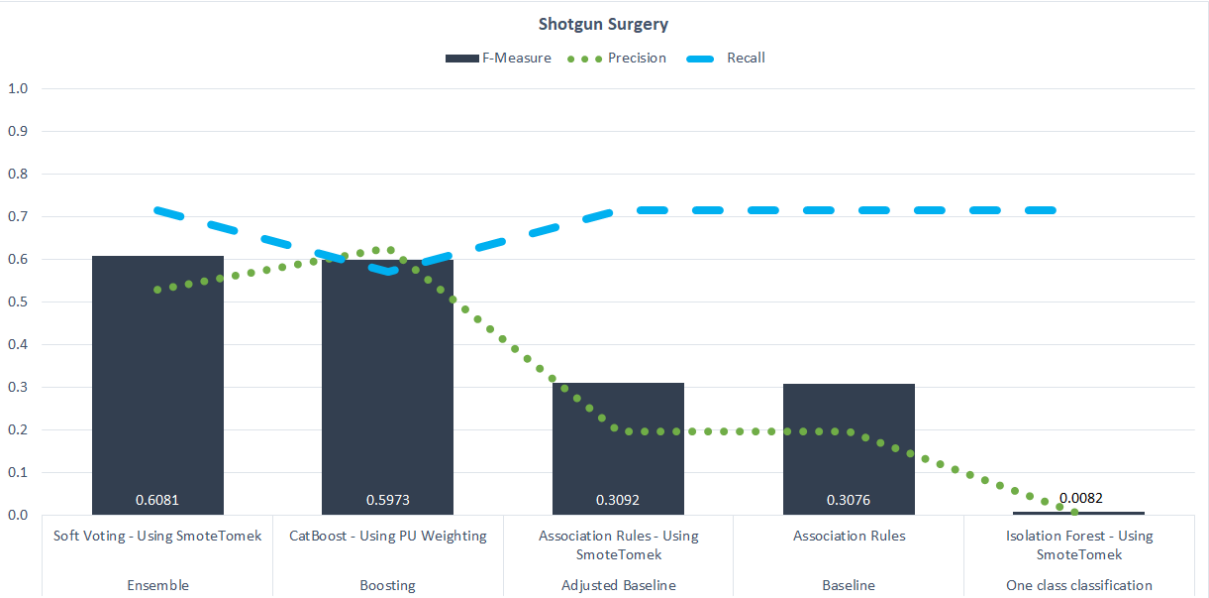
mance with a score 0.06 (13%) above the Baseline. It presented a better precision, but a smaller recall. The adjusted model did not have any effect on the results. While the Boosting technique performed 0.034 (7%) worst with a slightly better precision but a much worst recall.

Figure 20 – Divergent Change results



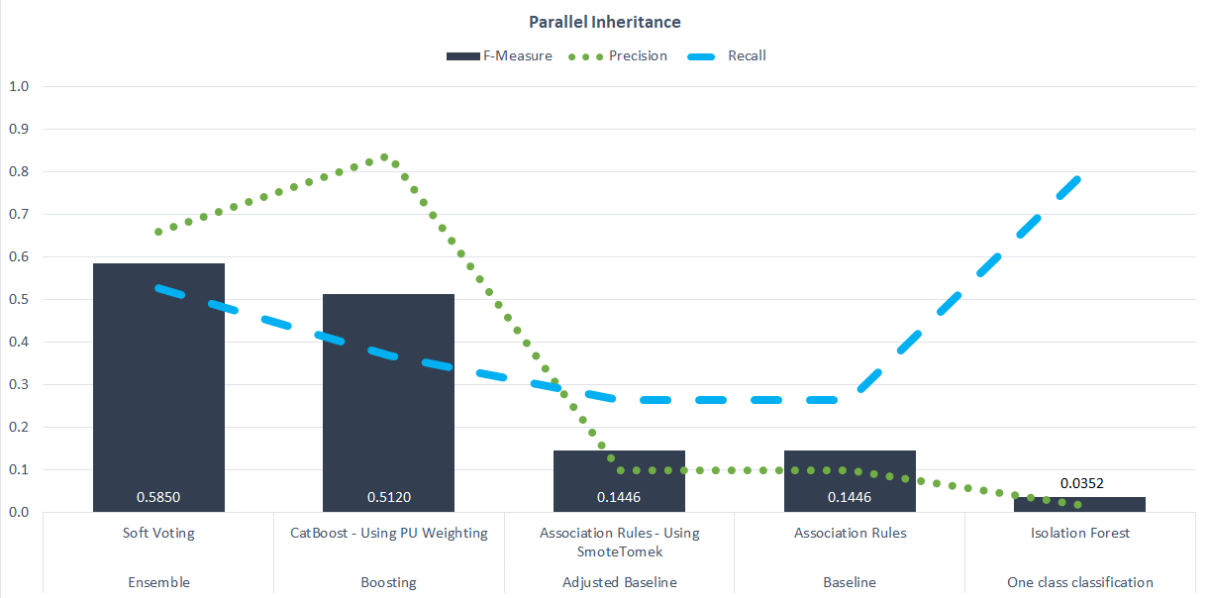
The shotgun surgery best performing model was also the Ensemble model, which score 0.3 (100%) above the baseline, with a loss on recall but a huge gain on precision. It was followed by CatBoost, with a gain of 0.29 (48.5%) with a worst recall but a much better precision than the baseline. The adjusted model only performed slightly better than the original model, both had a big recall but a small precision.

Figure 21 – Shotgun Surgery results



When it comes to Parallel Inheritance the best performing technique was also the Ensemble model, which outperformed the original technique by 0.44 (304%), presenting a better precision and recall. It was followed by the CatBoost, which performed 0.37 (255%) better than the baseline, it had a huge precision, but a recall smaller than the Ensemble. On this smell, the Adjusted Article also did not demonstrated any improvement when compared to the baseline.

Figure 22 – Parallel Inheritance results



For all the smells the One Class Classification Techniques performed poorly, even though it achieved a high recall it wasn't able to obtain a relevant precision, since it searches for outliers it is probable that the algorithms could not find a clear boundary between the smells and the unlabeled set, since the difference in the metrics is subtle, bringing the results to a lower score.

The Positive/Unlabeled weighting adjustment helped improving the score in 1/2 of the highest scoring techniques, while the under/oversampling technique helped it in 1/3 of the top techniques. They also were able to improve the baseline techniques performance in 4/6 of the techniques, on the other 2 it performed the same. From these 4 that achieved a better performance, 2 used a combination of the weight adjustment and under/oversampling, 1 used only the weight adjustment and 1 used only the under/oversampling.

4.6 Discussion

This studied tried to create a replicable experiment recreating the best performing machine learning techniques learning applied for code smells identification found in the

literature machine. The study also added some techniques that are recommended for highly imbalanced datasets, that characterizes the setup of this experiment.

All the replicated techniques performed worst on this experiment than on the original experiment they were performed. We believe that the main reason for that is the different ratio of smells in the code. While the original experiments used Positive/Negative annotated code in a ration that varied between 1/3 of smells and 1/2 Fontana et al. (2013); Palomba et al. (2015a), ours used a Positive/Unlabeled approach, drastically reducing the rate which varied between 0.3% and 2% depending on the smell. This setup makes it harder to achieve a high precision without hurting the recall and vice-versa, while the harmonic mean (F-Measure) penalizes lower values for any of the metrics challenging the models to achieve a good balance between both of them.

Even though none of the models performed as well on the replicated experiment as they did on the original experiment, the Decision Tree used for the Long Method was the technique that achieved the highest result among the replicated techniques. One possible reason for that is that the features that characterizes Long Method smells are cleared defined them the features used for the other smells. The higher smell ratio may also help it's performance, but we believe that this factor doe not have as much influence as the features, given that the Parallel Inheritance smell has the second bigger ratio but was one of the worst performing. We also found that the smells which relied heavily on metrics performed better on the replicated experiment than the ones which relied on the change history, this may happen due to the hard thresholds used on the Association Rules extracted for the change history, which made the original experiments less flexible.

The original models also took advantage of the weight adjustment and under/oversampling techniques, 4 out of the 6 replicated experiments achieved betters results by using these techniques. The smells that took the most advantage on it were the ones that used more code metrics, that achieved improvements ranging from 1% improvement on Long Method to 31% for Large Class. From the models that used the change history and association rules only Shotgun Surgery had any improvement by using it, but with only a tiny contribution.

The model which performed the best for most of the smells was the Soft Voting Ensemble model, it performed particularly well for the relationship based smells: Divergent Change, Shotgun Surgery and Parallel Inheritance, that relied more in the change history and association rules parameters. They were also the smells with the least amount of annotated smell. The random forest that was recommended as baseline for Feature Envy and Long Method, and was improved with weight adjustment for both of them and under/oversampling for the first, also were the best performers for 2 smells. Since Random Forest can also be considered a boosting technique, we can conclude that the Boosting techniques also were the top performers for 3 out of the 6 smells, mainly those that rely

heavily on metrics.

Most of the best performing technique also found the weight adjustments and SmoteTomek techniques to be useful for improving their performance. Only two of the best performing techniques didn't use it and both of them were Ensemble models. The one class classification techniques were not able to reach any significant performance for any of the smells, even though it was able to reach high recall rates, it failed to reach a good recall. Since it relies heavily on outlier detection, it's poor performance demonstrates that the boundaries between what is and what is not a smell can't be clearly defined only by the used features, it may also rely on some adjustable probability.

4.7 Threats to validity

We have selected the best performing techniques based on a Mapping Study, but since the studied techniques used different datasets and even the studies that use the same project use different annotations to train the data, decreasing the reliability of the comparisons of performance. This threat is increased by publication bias as the researches tend to release only positive results, avoiding the negative ones, and also tend to show that their results outperform the others.

Another possible threat is that some snapshots used by the dataset does not exist anymore, in order to reduce this risk we used snapshots taken around the same time of the ones proposed in the original dataset and manually re-validate it. There were are also some annotations that failed to merge with our dataset due to non-standard nomenclature, what can reduce the annotation size and consequently impact the performance.

Finally, we could not find any empirical study to define the distribution of smells per project, hence the β value used to build the contingency table had to be build based on the dataset distribution per project. In order to mitigate this risk we also developed a confidence interval for the beta value and calculated the resulting values based on this confidence interval, which can be found in Appendix B.

4.8 Conclusions

This study replicated the best performing machine learning techniques for code smell identification to create a bench-marking in a standardized annotated dataset for the evaluation and posterior comparison of the techniques. The code smell distribution differed from the original experiment, the rates used by the original setup varied between 1/3 and 1/2 smells per non smells, ours vary between 0.3% to 2%, we kept the ratio found in the annotated dataset, since it is closer real life scenarios.

We found the techniques did not perform as well as they did on the original experiment. But the code metrics related smells were able to take advantage of weight adjustments and under/oversampling to improve their performance. Using techniques that are best fit for this kind of imbalanced data can greatly improve the existing techniques under these circumstances. Among the studied smells we found that Ensemble models were best fit for the smells that were based on relationships between methods and classes, while the Boosting techniques performed better for smells that are related to the structure of classes and methods.

We believe that the smells identification techniques could benefit from using a realistic ratio of smell since it would reduce the potential number of false positives, improving the developer's trust in the techniques and reducing the amount of work needed to track the existing smells. Future experiments can also benefit from the developed experiment to run benchmarks where they can check their techniques performance against proven techniques without needing manual labor to annotate smells.

Although we were able to improve the results of the replicated techniques the overall results still could not reach a satisfactory results. Future experiments could focus on the improvement of the results for the existing setup. The used dataset, even thought is the most complete that could be found in literature, still lack many of the 22 smells defined by Fowler and Beck (1999), future experiments could extend this study by improving the Palomba et al. (2015b) work and adding new smells annotations to it, and using these new smells to create new bench-markings.

5 Conclusion

During this study, we mapped the best performing techniques that we could find in literature. Even though many of the mapped studies reported the results in terms of precision, recall and f-measure, they are still hard to compare given that they do not use the same dataset, and when they do they do not use the same annotated smells. So we replicated them on a standardized dataset aiming to develop a ML techniques for code smell identification golden standard. Since we found that the existing techniques considered an unreal smell ratio in the projects we tried to incorporate this element into our model by using a Positive (the landfill annotated smells) / Unlabeled (the rest of the code) setup.

Using a smell ratio of 1/3 that is similar to the one used in the original datasets, we were able to perform close to the original results, with a maximum variance of 12% for the replicated techniques. But when the same techniques were replicated using the whole code, they performed worst, some performed even 300% below the original results. We used some techniques recommended for highly imbalanced datasets in order to improve the performance on this scenario, these techniques brought significant improvement over the original techniques. Regarding the studied smells we found that Ensemble models were best fit for the smells that were based on relationships between methods and classes, while the Boosting techniques performed better for smells that are related to the structure of classes and methods.

We believe that the results from this work can help the evolution of the subject by providing a way baseline where future works can be compared against. We also believe that the smells identification techniques could greatly benefit from using a more realistic ratio of smell since it would reduce the potential number of false positives, improving the developer's trust in the techniques and reducing the amount of work needed to track the existing smells.

Even though we were able to improve the results of the replicated techniques under the highly imbalanced setup the results still have plenty room for improvement, hence future experiments could focus on the improvement of the results for this setup. The used dataset, even thought is the most complete in literature, still lack many of the 22 smells defined by Fowler and Beck (1999), future experiments could extend this study by improving the Palomba et al. (2015b) work and improving the annotations and creating new benchmarks.

Bibliography

Abran, A. and Nguyenkim, H. (1993). Measurement of the maintenance process from a demand based perspective. *Journal of Software Maintenance: Research and Practice*, 5(2):63–90.

Aggarwal, K. K., Singh, Y., and Chhabra, J. K. (2002). An integrated measure of software maintainability. In *Reliability and Maintainability Symposium, 2002. Proceedings. Annual*, pages 235–241, Seattle, WA, USA. IEEE.

Akay, F. M. (2009). Support vector machines combined with feature selection for breast cancer diagnosis. *Expert Systems with Applications*, 36(2):3240–3247.

Al Dallal, J. (2015). Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and Software Technology*, 58:231–249.

Basili, V. R. (2007). The role of controlled experiments in software engineering research. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, pages 33–37. Springer.

Bavota, G., De Lucia, A., and Oliveto, R. (2011). Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414.

Bennett, K. H. and Rajlich, V. T. V. T. (2000). Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, volume 225, pages 73 – 87. ACM.

Bernardi, M. L., Cimitile, M., and Di Lucca, G. (2016). Mining static and dynamic crosscutting concerns: a role-based approach. *Journal of Software: Evolution and Process*, 28(5):306–339.

Brown, W. J., Malveau, R. C., Mowbray, T. J., and Wiley, J. (1998). *AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis*, volume 3. John Wiley & Sons, Inc.

Bryton, S. and Abreu, F. B. (2009). Strengthening refactoring: Towards software evolution with quantitative and experimental grounds. In *4th International Conference on Software Engineering Advances, ICSEA 2009*, pages 570–575, Porto, Portugal. IEEE.

Bryton, S., Brito E Abreu, F., and Monteiro, M. (2010). Reducing subjectivity in code smells detection: Experimenting with the Long Method. In *Proceedings - 7th Inter-*

national Conference on the Quality of Information and Communications Technology, QUATIC 2010, 7, pages 337–342, Porto, Portugal. IEEE.

Chakraborty, C. and Joseph, A. (2017). Machine learning at central banks. Bank of England working papers 674, Bank of England.

Chatzigeorgiou, A. and Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 106–115.

Chatzigeorgiou, A. and Manakos, A. (2014). Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, 10(1):3–18.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794. ACM.

Claesen, M., Davis, J., De Smet, F., and De Moor, B. (2015). Assessing binary classifiers using only positive and unlabeled data. *ArXiv e-prints*, 1.

Counsell, S., Hierons, R. M., Hamza, H., Black, S., and Durrand, M. (2010). Is a strategy for code smell assessment long overdue? In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pages 32–38, Cape Town, South Africa. ACM.

Cowell, R. G., Verrall, R. J., and Yoon, Y. K. (2007). Modelling Operational Risk With Bayesian Networks. *The Journal of Risk and Insurance*, 74(4):795–827.

Creswell, J. W. (2013). *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

Criteo (2015). E-Commerce Industry Outlook 2015.

Dalkey, N. and Helmer, O. (1963). An experimental application of the delphi method to the use of experts. *Management science*, 9(3):458–467.

Doostmohammadi, A., Amjady, N., and Zareipour, H. (2017). Day-ahead Financial Loss/Gain Modeling and Prediction for a Generation Company. *IEEE Transactions on Power Systems*, PP(99):1–1.

Dorogush, A. V., Ershov, V., and Gulin, A. (2017). Catboost: gradient boosting with categorical features support.

Dybå, T. and Dingsøyr, T. (2008). Strength of evidence in systematic reviews in software engineering. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 178–187. ACM.

Easterbrook, S. M., Singer, J., Storey, M., and Damian, D. (2007). Selecting empirical methods for Software Engineering research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer.

Eichberg, M., Hermann, B., Mezini, M., and Glanz, L. (2015). Hidden Truths in Dead Software Paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 474–484, New York, NY, USA. ACM.

Ekbia, H. R. (2010). Fifty Years of Research in Artificial Intelligence. *Annual review of information science and technology*, 44(1):201–242.

Elkan, C. and Noto, K. (2008). Learning classifiers from only positive and unlabeled data. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 213–220. ACM.

Fenton, N. and Neil, M. (2007). Managing risk in the modern world. *Application of Bayesian Networks*, 1(1):1–28.

Ferme, V., Marino, A., and Fontana, F. A. (2013). Is it a Real Code Smell to be Removed or not? In *International Workshop on Refactoring & Testing (RefTest) 2013*, Wien, Austria.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, pages 1–12, Limerick, Ireland. ACM.

Fittkau, F. (2011). Controlled experiments in software engineering. Working paper, Kiel University, Kiel, Germany.

Fokaefs, M., Tsantalis, N., and Chatzigeorgiou, A. (2007). Jdeodorant: Identification and removal of feature envy bad smells. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 519–520, Paris, France. IEEE.

Fokaefs, M., Tsantalis, N., Stroulia, E., and Chatzigeorgiou, A. (2012). Identification and application of Extract Class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260.

Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2):1–5.

Fontana, F. A., Ferme, V., Zanoni, M., and Yamashita, A. (2015). Automatic metric thresholds derivation for code smell detection. In *International Workshop on Emerging Trends in Software Metrics, WETSoM*, volume 2015-August, pages 44–53, Piscataway, NJ, USA. IEEE Press.

Fontana, F. A., Zanoni, M., Marino, A., and Mäntylä, M. V. (2013). Code smell detection: Towards a machine learning-based approach. In *IEEE International Conference on Software Maintenance, ICSM*, pages 396–399, Eindhoven, The Netherlands.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Fownler, M. (2016). Codesmell. https://martinfowler.com/bliki/CodeSmell.html. (Accessed on 01/14/2018).

Fu, S. and Shen, B. (2015). Code Bad Smell Detection through Evolutionary Data Mining. In *International Symposium on Empirical Software Engineering and Measurement*, pages 41–49.

Ghannem, A., El Boussaidi, G., and Kessentini, M. (2014). Model refactoring using examples: A search-based approach. *Journal of Software: Evolution and Process*, 26(7):692–713.

Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., and Daniel, R. (2007). Using concept analysis to detect co-change patterns. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, volume 6, page 89, New York, NY, USA. ACM.

Global Web Index (2016). Digital consumers own 3.64 connected devices. Global Web Index.

Goldberg, D. E. and Holland, J. H. (1988). Genetic algorithms and machine learning. *Machine Learning*, 3(2):95–99.

Gosling, J. and McGilton, H. (1995). The Java language environment: a white paper. *Language*, 1:86.

Griffith, I., Wahl, S., and Izurieta, C. (2011). Evolution of legacy system comprehensibility through automated refactoring. In *Proceedings of the International Workshop on Machine Learning Technologies in Software Engineering (MALETS)*, pages 35–42, New York, NY, USA. ACM.

Hozano, M., Garcia, A., Fonseca, B., and Costa, E. (2017). Are you smelling it? investigating how similar developers detect code smells. *Information and Software Technology*.

Hripcsak, G. and Rothschild, A. S. (2005). Agreement, the f-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298.

Jaafar, F., Guéhéneuc, Y.-G., Hamel, S., Khomh, F., and Zulkernine, M. (2016). Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults. *Empirical Software Engineering*, 21(3):896–931.

Jain, A. (2010). Data clustering: 50 years beyond K-meansstar, open. *Pattern Recognition Letters*, 31(8):651–666.

Jain, A. K., Murty, M. N., and Flynn, P. J. (1999). Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323.

Juristo, N. and Moreno, A. M. (2001). Basics of Software Engineering Experimentation. *Analysis*, 5/6:420.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. (2017). Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157.

Kell, D. B. (2005). Metabolomics, machine learning and modelling: towards an understanding of the language of cells. *Biochemical Society transactions*, 33(Pt 3):520–524.

Khan, S. S. and Madden, M. G. (2014). One-class classification: taxonomy of study and review of techniques. *The Knowledge Engineering Review*, 29(3):345–374.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009a). An exploratory study of the impact of code smells on software change-proneness. In *Working Conference on Reverse Engineering, 2009. WCRE'09.*, 16, pages 75–84, Lille, France. IEEE.

Khomh, F., Vaucher, S., Gueheneuc, Y. G., and Sahraoui, H. (2009b). A Bayesian Approach for the Detection of Code and Design Smells. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 305–314, Jeju, Korea.

Khomh, F., Vaucher, S., Guéhéneuc, Y. G., and Sahraoui, H. (2011). BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572.

Kitchenham, B., Pretorius, R., Budgen, D., Brereton, O. P., Turner, M., Niazi, M., and Linkman, S. (2010). Systematic literature reviews in software engineering-A tertiary study. *Information and Software Technology*, 52(8):792–805.

Kitchenham, B. A., Budgen, D., and Brereton, P. (2015). Evidence-based software engineering and systematic reviews.

Kosker, Y., Turhan, B., and Bener, A. (2009). An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003.

Kothari, C. R. (2004). *Research methodology: Methods and techniques*. New Age International.

Kotsiantis, S. B. (2007). Supervised Machine Learning : A Review of Classification Techniques. *Informatica, An International Journal of Computing and Informatics*, 3176(31):249–268.

Kreimer, J. (2005). Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science*, 141(4):117–136.

Lee, S., Bae, G., Chae, H. S., Bae, D. H., and Kwon, Y. R. (2011). Automated scheduling for clone-based refactoring using a competent GA. *Software - Practice and Experience*, 41(5):521–550.

Lee, S. J., Lo, L. H., Chen, Y. C., and Shen, S. M. (2016). Co-changing code volume prediction through association rule mining and linear regression model. *Expert Systems with Applications*, 45:185–194.

Leijdekkers, B. (2017). Metricsreloaded.

Li, Y. (2017). Deep Reinforcement Learning: An Overview. *arXiv preprint arXiv:1701.07274*.

Liu, H., Ma, Z., Shao, W., and Niu, Z. (2012). Schedule of bad smell detection and resolution: A new way to save effort. *IEEE Transactions on Software Engineering*, 38(1):220–235.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012). Are automatically-detected code anomalies relevant to architectural modularity? In *11th annual international conference on Aspect-oriented Software Development (AOSD '12)*, page 167, Potsdam, Germany. ACM.

Maneerat, N. and Muenchaisri, P. (2011). Bad-smell prediction from software design model using machine learning techniques. In *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 331–336.

Mantyla, M., Vanhanen, J., and Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, OCTOBER:381–384.

Mantyla, M., Vanhanen, J., and Lassenius, C. (2004). Bad smells: humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 399–408, Chicago, Illinois, USA. IEEE.

Marinescu, C., Marinescu, R., Mihancea, P. F., and Wettel, R. (2005). iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*. Citeseer.

Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In *IEEE International Conference on Software Maintenance, ICSM*, pages 350–359, Chicago, Illinois, USA. IEEE.

Marinescu, R. (2005). Measurement and quality in object-oriented design. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 701–704. IEEE.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M. (2014a). High dimensional search-based software engineering. In *Proceedings of the 2014 conference on Genetic and evolutionary computation - GECCO '14*, pages 1263–1270, New York, NY, USA. ACM.

Mkaouer, M. W., Kessentini, M., Bechikh, S., Ó'Cinnéide, M., and Deb, K. (2014b). Software refactoring under uncertainty. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion - GECCO Comp '14*, pages 187–188, New York, NY, USA. ACM.

Moha, N. and Guéhéneuc, Y. (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Murphy-hill, E., Parnin, C., and Black, A. P. (2012). How We Refactor , and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):55–57.

Newman, D. R. (2006). *The use of linkage learning in genetic algorithms*. September.

Nongpong, K. (2012). *Integrating" code smells" detection with refactoring tool support*. PhD thesis, The University of Wisconsin-Milwaukee.

Olbrich, S., Cruzes, D. S., Basili, V., and Zazworka, N. (2009). The Evolution and Impact of Code Smells : A Case Study of Two Open Source Systems What are code smells ? In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement*, April, pages 390–400, Orlando, FL, USA. IEEE Computer Society.

Oliveto, R., Gethers, M., Bavota, G., Poshyvanyk, D., and De Lucia, A. (2011). Identifying method friendships to remove the feature envy bad smell. In *Proceeding of the 33rd*

international conference on Software engineering - ICSE '11, page 820, New York, NY, USA. ACM.

Palomba and Fabio (2015). Textual analysis for code smell detection. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 769–771, Piscataway, NJ, USA. IEEE Press.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Poshyvanyk, D., and De Lucia, A. (2015a). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489.

Palomba, F., Nucci, D. D., Tufano, M., Bavota, G., Oliveto, R., Poshyvanyk, D., and Lucia, A. D. (2015b). Landfill : an Open Dataset of Code Smells with Public Evaluation. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 482–485, Florence, Italy. IEEE.

Pmd (2013). Pmd.

Powers, D. M. (2011). Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, 2.

Rasool, G. and Arshad, Z. (2015). A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895.

Rattan, D., Bhatia, R., and Singh, M. (2013). Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199.

Sahin, D., Kessentini, M., Bechikh, S., and Deb, K. (2014). Code-Smell Detection as a Bilevel Problem. *ACM Transactions on Software Engineering and Methodology*, 24(1):1–44.

Seng, O., Stammel, J., and Burkhart, D. (2006). Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 1909–1916, Seattle, WA, USA. ACM.

Shatnawi, R. and Li, W. (2008). The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of Systems and Software*, 81(11):1868–1882.

Shull, F. J., Carver, J. C., Vegas, S., and Juristo, N. (2008). The role of replications in Empirical Software Engineering. *Empirical Software Engineering*, 13(2):211–218.

Sjoberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A., and Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156.

Soltanifar, B., Akbarinasaji, S., Caglayan, B., Bener, A. B., Filiz, A., and Kramer, B. M. (2016). Software Analytics in Practice. In *Proceedings of the 20th International Database Engineering & Applications Symposium on - IDEAS '16*, pages 148–155, New York, NY, USA. ACM.

Taibi, D., Janes, A., and Lenarduzzi, V. (2017). How developers perceive smells in source code: A replicated study. *Information and Software Technology*, 92(Supplement C):223 – 235.

Tufano, M., Palomba, F., Bavota, G., Olivetox, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 403–414, Florence, Italy. IEEE Press.

Walter, B. and Alkhaeir, T. (2016). The relationship between design patterns and code smells: An exploratory study. *Information and Software Technology*, 74:127–142.

Wang, H., Kessentini, M., Grosky, W., and Meddeb, H. (2015). On the use of time series and search based software engineering for refactoring recommendation. In *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems - MEDES '15*, volume 7, pages 35–42, New York, NY, USA. ACM.

Wen, J., Li, S., Lin, Z., Hu, Y., and Huang, C. (2012). Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology*, 54(1):41–59.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yamashita, A. and Moonen, L. (2013). Do developers care about code smells? An exploratory survey. In *Proceedings - Working Conference on Reverse Engineering, WCRE*, volume 13, pages 242–251, Koblenz, Germany.

Zelkowitz, M. V. and Wallace, D. R. (1998). Experimental models for validating technology. *Computer*, 31(5):23–31.

Zhang, H., Babar, M. A., and Tell, P. (2011a). Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637.

Zhang, M., Hall, T., and Baddoo, N. (2011b). Code Bad Smells: A review of current knowledge. *Journal of Software Maintenance and Evolution*, 23(3):179–202.

Zhu, X. (2011). Semi-supervised learning. In *Encyclopedia of Machine Learning*, pages 892–897. Springer.

Zhu, X. and Goldberg, A. B. (2009). Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130.

Zibran, M. F. and Roy, C. K. (2012). IDE-based Real-time Focused Search for Near-miss Clones. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1235–1242, New York, NY, USA. ACM.

Appendix

APPENDIX A – SLR Results and Articles

Table 10 – F-Measure summary per smell and technique: Ordered by the median f-measure

Smell	Technique	Min	Median	Max
Middle Man	Decision Tree	1.00	1.00	1.00
	Linear Statistics	0.85	0.85	0.85
	Nearest Neighbor	0.85	0.85	0.85
	Random Forest	0.71	0.71	0.71
	Naive Bayes Classifier	0.14	0.38	0.62
Speculative Generality	Association Rules (AR)	0.86	1.00	1.00
Divergent Change	Association Rules (AR)	0.55	0.85	1.00
Long Method	Association Rules (AR)	0.99	0.99	0.99
	Random Forest	0.86	0.92	0.99
	Decision Tree	0.86	0.92	0.98
	Support Vector Machine	0.69	0.83	0.97
	Naive Bayes Classifier	0.54	0.60	0.93
	Nearest Neighbor	0.89	0.90	0.90
	Linear Statistics	0.84	0.84	0.84
	Text-Based	0.56	0.62	0.71
Large Class	Random Forest	0.97	0.97	0.97
	Association Rules (AR)	0.97	0.97	0.97
	Decision Tree	0.96	0.96	0.96
	Naive Bayes Classifier	0.96	0.96	0.96
	Support Vector Machine	0.73	0.85	0.96
Long Parameter List	Random Forest	0.97	0.97	0.97
	Linear Statistics	0.95	0.95	0.95
	Nearest Neighbor	0.92	0.92	0.92
	Decision Tree	0.92	0.92	0.92
	Naive Bayes Classifier	0.31	0.32	0.33
Shotgun Surgery	Decision Tree	0.97	0.97	0.97
	Nearest Neighbor	0.89	0.90	0.91
	Linear Statistics	0.89	0.89	0.89
	Random Forest	0.89	0.89	0.89
	Naive Bayes Classifier	0.52	0.61	0.70
Feature Envy	Decision Tree	0.96	0.96	0.96
	Support Vector Machine	0.69	0.83	0.96
	Naive Bayes Classifier	0.24	0.26	0.95
	Random Forest	0.94	0.94	0.94

	Association Rules (AR)	0.86	0.86	0.86
	Nearest Neighbor	0.84	0.84	0.84
	Linear Statistics	0.75	0.75	0.75
Duplicated Code	Semi-supervised	0.96	0.96	0.96
	Association Rules (AR)	0.72	0.85	0.85
Message Chains	Linear Statistics	0.86	0.86	0.86
	Nearest Neighbor	0.86	0.86	0.86
	Random Forest	0.80	0.80	0.80
	Naive Bayes Classifier	0.67	0.71	0.76
	Decision Tree	0.66	0.66	0.66
God Class	Clustering	0.66	0.72	0.82
BLOB	Bayesian Networks (BN)	0.60	0.69	0.79

Table 11 – Precision and recall summary per smell and technique: Ordered by the median precision

Technique	Smell Description	Precision			Recall		
		Min	Mean	Max	Min	Mean	Max
Decision Tree	Middle Man	1.00	1.00	1.00	1.00	1.00	1.00
Decision Tree	Shotgun Surgery	0.94	0.94	0.94	0.99	0.99	0.99
Decision Tree	Long Parameter List	0.86	0.86	0.86	0.98	0.98	0.98
Decision Tree	Long Method	0.77	0.77	0.77	0.97	0.97	0.97
Decision Tree	Message Chains	0.60	0.60	0.60	0.73	0.73	0.73
Semi-supervised	Duplicated Code	1.00	1.00	1.00	0.94	0.94	0.94
Association Rules (AR)	Speculative Generality	0.75	0.94	1.00	1.00	1.00	1.00
Association Rules (AR)	Duplicated Code	0.75	0.85	0.90	0.60	0.79	0.90
Association Rules (AR)	Divergent Change	0.50	0.80	1.00	0.50	0.76	1.00
Random Forest	Long Parameter List	0.95	0.95	0.95	0.99	0.99	0.99
Random Forest	Feature Envy	0.89	0.89	0.89	0.99	0.99	0.99
Random Forest	Shotgun Surgery	0.81	0.81	0.81	0.98	0.98	0.98
Random Forest	Long Method	0.77	0.77	0.77	0.96	0.96	0.96
Random Forest	Middle Man	0.76	0.76	0.76	0.67	0.67	0.67
Random Forest	Message Chains	0.75	0.75	0.75	0.87	0.87	0.87
Linear Statistics	Long Parameter List	0.92	0.92	0.92	0.99	0.99	0.99
Linear Statistics	Middle Man	0.88	0.88	0.88	0.83	0.83	0.83
Linear Statistics	Message Chains	0.86	0.86	0.86	0.87	0.87	0.87

Linear Statistics	Shotgun Surgery	0.81	0.81	0.81	0.98	0.98	0.98
Linear Statistics	Long Method	0.74	0.74	0.74	0.96	0.96	0.96
Linear Statistics	Feature Envy	0.62	0.62	0.62	0.97	0.97	0.97
Nearest Neighbor	Middle Man	0.88	0.88	0.88	0.83	0.83	0.83
Nearest Neighbor	Long Parameter List	0.87	0.87	0.87	0.98	0.98	0.98
Nearest Neighbor	Message Chains	0.86	0.86	0.86	0.87	0.87	0.87
Nearest Neighbor	Shotgun Surgery	0.81	0.83	0.85	0.98	0.98	0.98
Nearest Neighbor	Long Method	0.83	0.83	0.83	0.98	0.98	0.98
Nearest Neighbor	Feature Envy	0.73	0.73	0.73	0.98	0.98	0.98
Clustering	God Class	0.54	0.67	0.77	0.75	0.82	0.88
Naive Bayes Classifier	Message Chains	0.67	0.70	0.73	0.67	0.73	0.80
Naive Bayes Classifier	Middle Man	0.54	0.60	0.67	0.08	0.33	0.58
Naive Bayes Classifier	Shotgun Surgery	0.38	0.47	0.56	0.83	0.88	0.92
Naive Bayes Classifier	Long Method	0.40	0.43	0.47	0.82	0.84	0.85
Naive Bayes Classifier	Long Parameter List	0.21	0.23	0.24	0.54	0.55	0.56
Naive Bayes Classifier	Feature Envy	0.14	0.15	0.16	0.73	0.73	0.74
Text-Based	Long Method	0.63	0.66	0.67	0.50	0.61	0.77
Bayesian Networks (BN)	BLOB	0.43	0.54	0.65	1.00	1.00	1.00

Table 12 – Articles selected for SLR

Article	Author
A Bayesian Approach for the Detection of Code and Design Smells	Khomh et al. (2009b)
An expert system for determining candidate software classes for refactoring	Kosker et al. (2009)
Automated scheduling for clone-based refactoring using a competent GA	Lee et al. (2011)
Automatic Metric Thresholds Derivation for Code Smell Detection	Fontana et al. (2015)

Bad-smell prediction from software design model using machine learning techniques	Maneerat and Muenchaisri (2011)
BDTEX: A GQM-based Bayesian approach for the detection of anti-patterns	Khomh et al. (2011)
Co-changing code volume prediction through association rule mining and linear regression model	Lee et al. (2016)
Code Bad Smell Detection through Evolutionary Data Mining	Fu and Shen (2015)
Code Smell Detection: Towards a Machine Learning-Based Approach	Fontana et al. (2013)
Code-Smell Detection As a Bilevel Problem	Sahin et al. (2014)
Evolution of Legacy System Comprehensibility Through Automated Refactoring	Griffith et al. (2011)
Hidden Truths in Dead Software Paths	Eichberg et al. (2015)
High Dimensional Search-based Software Engineering	Mkaouer et al. (2014a)
IDE-based Real-time Focused Search for Near-miss Clones	Zibran and Roy (2012)
Identification and application of Extract Class refactorings in object-oriented systems	Fokaefs et al. (2012)

Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures	Bavota et al. (2011)
Identifying Method Friendships to Remove the Feature Envy Bad Smell	Oliveto et al. (2011)
Mining static and dynamic crosscutting concerns: a role-based approach	Bernardi et al. (2016)
Mining Version Histories for Detecting Code Smells	Palomba et al. (2015a)
Model refactoring using examples: a search-based approach	Ghannem et al. (2014)
On the Use of Time Series and Search Based Software Engineering for Refactoring Recommendation	Wang et al. (2015)
Search-based Determination of Refactorings for Improving the Class Structure of Object-oriented Systems	Seng et al. (2006)
Software Analytics in Practice: A Defect Prediction Model Using Code Smells	Soltanifar et al. (2016)
Software Refactoring Under Uncertainty: A Robust Multi-objective Approach	Mkaouer et al. (2014b)
Textual Analysis for Code Smell Detection	Palomba and Fabio (2015)
Using Concept Analysis to Detect Co-change Patterns	Gîrba et al. (2007)

APPENDIX B – Experiment Results

Table 13 – Experiment Results and Confidence Interval (lowerbound(LB), mean and upperbound(UB))

Smell	Category	Model	F-Measure			Precision			Recall		
			CI (LB)	Mean	CI (UB)	CI (LB)	Mean	CI (UB)	CI (LB)	Mean	CI (UB)
Blob	Boosting	XGBoost - Using PU Weighting	0.5119	0.5913	0.6339	0.4322	0.5590	0.6404	0.6275	0.6275	0.6275
Blob	Boosting	CatBoost - Using PU Weighting	0.5215	0.5834	0.6151	0.4954	0.6204	0.6967	0.5506	0.5506	0.5506
Blob	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.4970	0.5772	0.6206	0.4169	0.5434	0.6258	0.6154	0.6154	0.6154
Blob	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.5050	0.5636	0.5934	0.4925	0.6176	0.6942	0.5182	0.5182	0.5182
Blob	Adjusted Baseline	Random Forest - Using PU Weighting - Using SmoteTomek	0.4707	0.5542	0.6003	0.3843	0.5096	0.5935	0.6073	0.6073	0.6073
Blob	Adjusted Baseline	Random Forest - Using PU Weighting	0.4551	0.5378	0.5838	0.3716	0.4961	0.5805	0.5870	0.5870	0.5870
Blob	Adjusted Baseline	Decision Tree - Using PU Weighting	0.4153	0.4558	0.4759	0.4829	0.6086	0.6860	0.3644	0.3644	0.3644
Blob	Ensemble	Soft Voting	0.4217	0.4463	0.4579	0.6042	0.7176	0.7813	0.3239	0.3239	0.3239
Blob	Ensemble	Soft Voting - Using SmoteTomek	0.4140	0.4381	0.4494	0.6009	0.7148	0.7789	0.3158	0.3158	0.3158
Blob	Baseline	Decision Tree	0.3772	0.4215	0.4442	0.4174	0.5439	0.6263	0.3441	0.3441	0.3441
Blob	Boosting	CatBoost - Using SmoteTomek	0.3966	0.4117	0.4186	0.6837	0.7825	0.8349	0.2794	0.2794	0.2794
Blob	Boosting	CatBoost	0.3953	0.4108	0.4180	0.6756	0.7761	0.8297	0.2794	0.2794	0.2794
Blob	Ensemble	Soft Voting - Using PU Weighting	0.2981	0.4080	0.4858	0.1809	0.2688	0.3407	0.8462	0.8462	0.8462
Blob	Adjusted Baseline	Decision Tree - Using PU Weighting - Using SmoteTomek	0.3675	0.4080	0.4286	0.4246	0.5512	0.6332	0.3239	0.3239	0.3239
Blob	Adjusted Baseline	Decision Tree - Using SmoteTomek	0.3639	0.4079	0.4304	0.4026	0.5287	0.6119	0.3320	0.3320	0.3320
Blob	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.2863	0.3950	0.4729	0.1719	0.2569	0.3270	0.8543	0.8543	0.8543
Blob	Boosting	Light GBM	0.3481	0.3584	0.3630	0.7081	0.8016	0.8502	0.2308	0.2308	0.2308
Blob	Boosting	Light GBM - Using SmoteTomek	0.3481	0.3584	0.3630	0.7081	0.8016	0.8502	0.2308	0.2308	0.2308

Blob	Boosting	XGBoost	0.3345	0.3418	0.3450	0.7586	0.8395	0.8803	0.2146	0.2146	0.2146
Blob	Boosting	XGBoost - Using SmoteTomek	0.3345	0.3418	0.3450	0.7586	0.8395	0.8803	0.2146	0.2146	0.2146
Blob	Adjusted Baseline	Random Forest - Using SmoteTomek	0.3268	0.3349	0.3385	0.7296	0.8179	0.8632	0.2105	0.2105	0.2105
Blob	Adjusted Baseline	Naïve Bayes - Using SmoteTomek	0.2106	0.3062	0.3814	0.1186	0.1831	0.2395	0.9352	0.9352	0.9352
Blob	Baseline	Naïve Bayes	0.2106	0.3062	0.3814	0.1186	0.1831	0.2395	0.9352	0.9352	0.9352
Blob	Adjusted Baseline	Naïve Bayes - Using PU Weighting	0.2104	0.3062	0.3816	0.1183	0.1826	0.2389	0.9474	0.9474	0.9474
Blob	Baseline	Random Forest	0.2986	0.3046	0.3072	0.7534	0.8357	0.8773	0.1862	0.1862	0.1862
Blob	Adjusted Baseline	Naïve Bayes - Using PU Weighting - Using SmoteTomek	0.2072	0.3020	0.3768	0.1164	0.1799	0.2356	0.9393	0.9393	0.9393
Blob	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.0472	0.0762	0.1037	0.0242	0.0397	0.0549	0.9312	0.9312	0.9312
Blob	Boosting	Light GBM - Using PU Weighting	0.0264	0.0432	0.0596	0.0134	0.0221	0.0308	0.9474	0.9474	0.9474
Blob	One class classification	Isolation Forest	0.0024	0.0040	0.0056	0.0012	0.0021	0.0029	0.0972	0.0972	0.0972
Blob	One class classification	Isolation Forest - Using SmoteTomek	0.0023	0.0039	0.0054	0.0012	0.0020	0.0028	0.0931	0.0931	0.0931
Blob	One class classification	One Class SVM	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Blob	One class classification	One Class SVM - Using SmoteTomek	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Divergent Change	Ensemble	Soft Voting	0.3525	0.5157	0.5841	0.2495	0.4522	0.5690	0.6000	0.6000	0.6000
Divergent Change	Ensemble	Soft Voting - Using SmoteTomek	0.3455	0.5096	0.5791	0.2425	0.4428	0.5597	0.6000	0.6000	0.6000
Divergent Change	Ensemble	Soft Voting - Using PU Weighting	0.3455	0.5096	0.5791	0.2425	0.4428	0.5597	0.6000	0.6000	0.6000
Divergent Change	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.3259	0.4920	0.5649	0.2237	0.4170	0.5336	0.6000	0.6000	0.6000
Divergent Change	Adjusted Baseline	Association Rules - Using PU Weighting	0.2584	0.4567	0.5667	0.1509	0.3060	0.4136	0.9000	0.9000	0.9000
Divergent Change	Baseline	Association Rules	0.2584	0.4567	0.5667	0.1509	0.3060	0.4136	0.9000	0.9000	0.9000
Divergent Change	Adjusted Baseline	Association Rules - Using PU Weighting - Using SmoteTomek	0.2533	0.4503	0.5605	0.1474	0.3003	0.4070	0.9000	0.9000	0.9000

Divergent Change	Boosting	Light GBM - Using PU Weighting	0.2739	0.4225	0.4897	0.1886	0.3658	0.4799	0.5000	0.5000	0.5000
Divergent Change	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.2739	0.4225	0.4897	0.1886	0.3658	0.4799	0.5000	0.5000	0.5000
Divergent Change	Adjusted Baseline	Association Rules - Using SmoteTomek	0.2354	0.4196	0.5231	0.1380	0.2844	0.3886	0.8000	0.8000	0.8000
Divergent Change	Boosting	CatBoost	0.3643	0.4167	0.4325	0.4637	0.6821	0.7744	0.3000	0.3000	0.3000
Divergent Change	Boosting	CatBoost - Using SmoteTomek	0.3643	0.4167	0.4325	0.4637	0.6821	0.7744	0.3000	0.3000	0.3000
Divergent Change	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.3643	0.4167	0.4325	0.4637	0.6821	0.7744	0.3000	0.3000	0.3000
Divergent Change	Boosting	CatBoost - Using PU Weighting	0.3234	0.3938	0.4167	0.3508	0.5729	0.6820	0.3000	0.3000	0.3000
Divergent Change	Boosting	Light GBM	0.2552	0.3811	0.4355	0.1874	0.3640	0.4779	0.4000	0.4000	0.4000
Divergent Change	Boosting	Light GBM - Using SmoteTomek	0.2552	0.3811	0.4355	0.1874	0.3640	0.4779	0.4000	0.4000	0.4000
Divergent Change	Boosting	XGBoost - Using SmoteTomek	0.2725	0.3607	0.3929	0.2495	0.4522	0.5690	0.3000	0.3000	0.3000
Divergent Change	Boosting	XGBoost	0.2641	0.3547	0.3884	0.2359	0.4339	0.5507	0.3000	0.3000	0.3000
Divergent Change	Boosting	XGBoost - Using PU Weighting	0.2641	0.3547	0.3884	0.2359	0.4339	0.5507	0.3000	0.3000	0.3000
Divergent Change	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.2563	0.3490	0.3841	0.2237	0.4170	0.5336	0.3000	0.3000	0.3000
Divergent Change	One class classification	Isolation Forest	0.0011	0.0028	0.0044	0.0006	0.0014	0.0022	0.4000	0.4000	0.4000
Divergent Change	One class classification	Isolation Forest - Using SmoteTomek	0.0011	0.0028	0.0044	0.0006	0.0014	0.0022	0.4000	0.4000	0.4000
Divergent Change	One class classification	One Class SVM	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Divergent Change	One class classification	One Class SVM - Using SmoteTomek	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Feature Envy	Adjusted Baseline	Random Forest - Using PU Weighting - Using SmoteTomek	0.4314	0.4715	0.4878	0.5446	0.6935	0.7694	0.3571	0.3571	0.3571
Feature Envy	Boosting	CatBoost - Using SmoteTomek	0.4344	0.4666	0.4794	0.5974	0.7373	0.8054	0.3413	0.3413	0.3413
Feature Envy	Baseline	Random Forest	0.4400	0.4664	0.4767	0.6469	0.7761	0.8364	0.3333	0.3333	0.3333
Feature Envy	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.4334	0.4660	0.4790	0.5939	0.7344	0.8031	0.3413	0.3413	0.3413

Feature Envy	Boosting	CatBoost	0.4325	0.4654	0.4786	0.5903	0.7316	0.8008	0.3413	0.3413	0.3413
Feature Envy	Adjusted Baseline	Random Forest - Using SmoteTomek	0.4327	0.4584	0.4684	0.6458	0.7752	0.8357	0.3254	0.3254	0.3254
Feature Envy	Ensemble	Soft Voting	0.3547	0.4450	0.4901	0.2917	0.4379	0.5346	0.4524	0.4524	0.4524
Feature Envy	Adjusted Baseline	Decision Tree - Using PU Weighting - Using SmoteTomek	0.3748	0.4440	0.4757	0.3616	0.5173	0.6124	0.3889	0.3889	0.3889
Feature Envy	Ensemble	Soft Voting - Using SmoteTomek	0.3495	0.4407	0.4865	0.2848	0.4296	0.5262	0.4524	0.4524	0.4524
Feature Envy	Boosting	CatBoost - Using PU Weighting	0.4094	0.4407	0.4531	0.5763	0.7201	0.7914	0.3175	0.3175	0.3175
Feature Envy	Adjusted Baseline	Random Forest - Using PU Weighting	0.4016	0.4393	0.4547	0.5243	0.6759	0.7546	0.3254	0.3254	0.3254
Feature Envy	Adjusted Baseline	Decision Tree - Using SmoteTomek	0.3575	0.4262	0.4580	0.3432	0.4970	0.5931	0.3730	0.3730	0.3730
Feature Envy	Baseline	Decision Tree	0.3483	0.4168	0.4487	0.3330	0.4857	0.5821	0.3651	0.3651	0.3651
Feature Envy	Adjusted Baseline	Decision Tree - Using PU Weighting	0.3478	0.4164	0.4484	0.3320	0.4846	0.5809	0.3651	0.3651	0.3651
Feature Envy	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.3010	0.4046	0.4621	0.2168	0.3436	0.4356	0.4921	0.4921	0.4921
Feature Envy	Ensemble	Soft Voting - Using PU Weighting	0.2864	0.3868	0.4429	0.2063	0.3295	0.4202	0.4683	0.4683	0.4683
Feature Envy	Boosting	XGBoost	0.1751	0.2735	0.3431	0.1029	0.1782	0.2424	0.5873	0.5873	0.5873
Feature Envy	Boosting	XGBoost - Using SmoteTomek	0.1681	0.2629	0.3300	0.0991	0.1722	0.2347	0.5556	0.5556	0.5556
Feature Envy	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.1505	0.2424	0.3110	0.0855	0.1503	0.2068	0.6270	0.6270	0.6270
Feature Envy	Boosting	Light GBM - Using SmoteTomek	0.1480	0.2379	0.3048	0.0845	0.1487	0.2048	0.5952	0.5952	0.5952
Feature Envy	Boosting	Light GBM	0.1477	0.2378	0.3049	0.0842	0.1481	0.2040	0.6032	0.6032	0.6032
Feature Envy	Boosting	XGBoost - Using PU Weighting	0.1464	0.2359	0.3029	0.0833	0.1467	0.2022	0.6032	0.6032	0.6032
Feature Envy	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.0984	0.1673	0.2240	0.0532	0.0962	0.1356	0.6429	0.6429	0.6429
Feature Envy	Boosting	Light GBM - Using PU Weighting	0.0972	0.1658	0.2224	0.0525	0.0948	0.1338	0.6587	0.6587	0.6587
Feature Envy	Baseline	Naïve Bayes	0.0964	0.1647	0.2212	0.0520	0.0939	0.1326	0.6667	0.6667	0.6667
Feature Envy	Adjusted Baseline	Naïve Bayes - Using SmoteTomek	0.0951	0.1626	0.2187	0.0512	0.0926	0.1308	0.6667	0.6667	0.6667

Feature Envy	Adjusted Baseline	Naïve Bayes - Using PU Weighting - Using SmoteTomek	0.0772	0.1347	0.1844	0.0409	0.0746	0.1062	0.6984	0.6984	0.6984
Feature Envy	Adjusted Baseline	Naïve Bayes - Using PU Weighting	0.0705	0.1238	0.1704	0.0371	0.0679	0.0970	0.6984	0.6984	0.6984
Feature Envy	One class classification	Isolation Forest	0.0064	0.0121	0.0177	0.0032	0.0061	0.0090	0.7540	0.7540	0.7540
Feature Envy	One class classification	Isolation Forest - Using SmoteTomek	0.0064	0.0121	0.0177	0.0032	0.0061	0.0090	0.7540	0.7540	0.7540
Feature Envy	One class classification	One Class SVM	0.0024	0.0046	0.0067	0.0012	0.0023	0.0034	0.1905	0.1905	0.1905
Feature Envy	One class classification	One Class SVM - Using SmoteTomek	0.0024	0.0046	0.0067	0.0012	0.0023	0.0034	0.1905	0.1905	0.1905
Long Method	Adjusted Baseline	Random Forest - Using PU Weighting	0.6320	0.6794	0.7066	0.6014	0.6935	0.7527	0.6659	0.6659	0.6659
Long Method	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.6362	0.6745	0.6960	0.6511	0.7367	0.7902	0.6220	0.6220	0.6220
Long Method	Boosting	CatBoost - Using PU Weighting	0.6356	0.6740	0.6957	0.6498	0.7357	0.7892	0.6220	0.6220	0.6220
Long Method	Baseline	Decision Tree	0.6289	0.6704	0.6940	0.6286	0.7174	0.7735	0.6293	0.6293	0.6293
Long Method	Adjusted Baseline	Decision Tree - Using PU Weighting	0.6228	0.6639	0.6873	0.6261	0.7152	0.7717	0.6195	0.6195	0.6195
Long Method	Adjusted Baseline	Decision Tree - Using SmoteTomek	0.6205	0.6617	0.6851	0.6240	0.7134	0.7701	0.6171	0.6171	0.6171
Long Method	Boosting	XGBoost - Using PU Weighting	0.5933	0.6599	0.7004	0.4944	0.5946	0.6637	0.7415	0.7415	0.7415
Long Method	Ensemble	Soft Voting - Using PU Weighting	0.6083	0.6533	0.6791	0.5952	0.6880	0.7479	0.6220	0.6220	0.6220
Long Method	Adjusted Baseline	Decision Tree - Using PU Weighting - Using SmoteTomek	0.6114	0.6520	0.6751	0.6207	0.7105	0.7676	0.6024	0.6024	0.6024
Long Method	Adjusted Baseline	Random Forest - Using PU Weighting - Using SmoteTomek	0.5770	0.6471	0.6901	0.4703	0.5711	0.6418	0.7463	0.7463	0.7463
Long Method	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.5846	0.6451	0.6813	0.5097	0.6092	0.6772	0.6854	0.6854	0.6854
Long Method	Boosting	CatBoost - Using SmoteTomek	0.6211	0.6363	0.6443	0.8133	0.8672	0.8978	0.5024	0.5024	0.5024

Long Method	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.5610	0.6332	0.6781	0.4502	0.5512	0.6230	0.7439	0.7439	0.7439
Long Method	Adjusted Baseline	Random Forest - Using SmoteTomek	0.6199	0.6314	0.6375	0.8500	0.8947	0.9196	0.4878	0.4878	0.4878
Long Method	Boosting	CatBoost	0.6170	0.6307	0.6380	0.8252	0.8762	0.9050	0.4927	0.4927	0.4927
Long Method	Baseline	Random Forest	0.5955	0.6090	0.6162	0.8177	0.8706	0.9005	0.4683	0.4683	0.4683
Long Method	Ensemble	Soft Voting	0.5564	0.5792	0.5917	0.7017	0.7791	0.8260	0.4610	0.4610	0.4610
Long Method	Ensemble	Soft Voting - Using SmoteTomek	0.5561	0.5784	0.5905	0.7065	0.7831	0.8293	0.4585	0.4585	0.4585
Long Method	Boosting	Light GBM - Using PU Weighting	0.4828	0.5697	0.6277	0.3452	0.4416	0.5155	0.8024	0.8024	0.8024
Long Method	Adjusted Baseline	Naïve Bayes - Using PU Weighting	0.4801	0.5434	0.5830	0.4069	0.5071	0.5806	0.5854	0.5854	0.5854
Long Method	Boosting	Light GBM	0.5261	0.5426	0.5515	0.7428	0.8124	0.8536	0.4073	0.4073	0.4073
Long Method	Boosting	Light GBM - Using SmoteTomek	0.5261	0.5426	0.5515	0.7428	0.8124	0.8536	0.4073	0.4073	0.4073
Long Method	Adjusted Baseline	Naïve Bayes - Using PU Weighting - Using SmoteTomek	0.4790	0.5425	0.5822	0.4053	0.5054	0.5790	0.5854	0.5854	0.5854
Long Method	Adjusted Baseline	Naïve Bayes - Using SmoteTomek	0.4789	0.5407	0.5790	0.4113	0.5116	0.5850	0.5732	0.5732	0.5732
Long Method	Baseline	Naïve Bayes	0.4789	0.5407	0.5790	0.4113	0.5116	0.5850	0.5732	0.5732	0.5732
Long Method	Boosting	XGBoost	0.5230	0.5344	0.5405	0.8029	0.8593	0.8915	0.3878	0.3878	0.3878
Long Method	Boosting	XGBoost - Using SmoteTomek	0.5230	0.5344	0.5405	0.8029	0.8593	0.8915	0.3878	0.3878	0.3878
Long Method	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.2009	0.2724	0.3334	0.1131	0.1606	0.2047	0.8976	0.8976	0.8976
Long Method	One class classification	Isolation Forest - Using SmoteTomek	0.0198	0.0290	0.0381	0.0103	0.0154	0.0206	0.2512	0.2512	0.2512
Long Method	One class classification	Isolation Forest	0.0185	0.0271	0.0356	0.0096	0.0144	0.0193	0.2341	0.2341	0.2341
Long Method	One class classification	One Class SVM	0.0074	0.0107	0.0139	0.0040	0.0060	0.0080	0.0537	0.0537	0.0537
Long Method	One class classification	One Class SVM - Using SmoteTomek	0.0074	0.0107	0.0139	0.0040	0.0060	0.0080	0.0537	0.0537	0.0537
Parallel Inheritance	Ensemble	Soft Voting	0.3571	0.5850	0.6436	0.2703	0.6583	0.8280	0.5263	0.5263	0.5263

Parallel Inheritance	Boosting	CatBoost - Using PU Weighting	0.4242	0.5120	0.5275	0.5000	0.8387	0.9286	0.3684	0.3684	0.3684
Parallel Inheritance	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.2075	0.4931	0.6137	0.1264	0.4295	0.6530	0.5789	0.5789	0.5789
Parallel Inheritance	Ensemble	Soft Voting - Using SmoteTomek	0.2857	0.4912	0.5474	0.2162	0.5893	0.7820	0.4211	0.4211	0.4211
Parallel Inheritance	Ensemble	Soft Voting - Using PU Weighting	0.1961	0.4647	0.5778	0.1205	0.4161	0.6404	0.5263	0.5263	0.5263
Parallel Inheritance	Boosting	CatBoost	0.4000	0.4622	0.4727	0.5455	0.8619	0.9398	0.3158	0.3158	0.3158
Parallel Inheritance	Boosting	CatBoost - Using SmoteTomek	0.3636	0.4522	0.4685	0.4286	0.7960	0.9070	0.3158	0.3158	0.3158
Parallel Inheritance	Boosting	XGBoost - Using PU Weighting	0.2264	0.3950	0.4419	0.1765	0.5271	0.7358	0.3158	0.3158	0.3158
Parallel Inheritance	Boosting	Light GBM	0.1842	0.3931	0.4691	0.1228	0.4214	0.6454	0.3684	0.3684	0.3684
Parallel Inheritance	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.2222	0.3925	0.4407	0.1714	0.5183	0.7290	0.3158	0.3158	0.3158
Parallel Inheritance	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.1404	0.3659	0.4749	0.0842	0.3235	0.5445	0.4211	0.4211	0.4211
Parallel Inheritance	Boosting	Light GBM - Using SmoteTomek	0.1429	0.3302	0.4062	0.0923	0.3460	0.5693	0.3158	0.3158	0.3158
Parallel Inheritance	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.2667	0.3286	0.3399	0.3636	0.7483	0.8814	0.2105	0.2105	0.2105
Parallel Inheritance	Boosting	Light GBM - Using PU Weighting	0.1217	0.3247	0.4262	0.0729	0.2903	0.5056	0.3684	0.3684	0.3684
Parallel Inheritance	Boosting	XGBoost - Using SmoteTomek	0.1538	0.2800	0.3171	0.1212	0.4177	0.6420	0.2105	0.2105	0.2105
Parallel Inheritance	Boosting	XGBoost	0.1509	0.2781	0.3161	0.1176	0.4095	0.6341	0.2105	0.2105	0.2105
Parallel Inheritance	Adjusted Baseline	Association Rules - Using SmoteTomek	0.0386	0.1446	0.2377	0.0208	0.0996	0.2167	0.2632	0.2632	0.2632
Parallel Inheritance	Baseline	Association Rules	0.0386	0.1446	0.2377	0.0208	0.0996	0.2167	0.2632	0.2632	0.2632

Parallel Inheritance	Adjusted Baseline	Association Rules - Using PU Weighting - Using SmoteTomek	0.0119	0.0582	0.1307	0.0060	0.0305	0.0729	0.6316	0.6316	0.6316
Parallel Inheritance	Adjusted Baseline	Association Rules - Using PU Weighting	0.0115	0.0563	0.1269	0.0058	0.0295	0.0706	0.6316	0.6316	0.6316
Parallel Inheritance	One class classification	Isolation Forest	0.0070	0.0352	0.0831	0.0035	0.0180	0.0439	0.7895	0.7895	0.7895
Parallel Inheritance	One class classification	Isolation Forest - Using SmoteTomek	0.0070	0.0352	0.0829	0.0035	0.0180	0.0438	0.7895	0.7895	0.7895
Parallel Inheritance	One class classification	One Class SVM	0.0015	0.0074	0.0176	0.0007	0.0039	0.0096	0.1053	0.1053	0.1053
Parallel Inheritance	One class classification	One Class SVM - Using SmoteTomek	0.0015	0.0074	0.0176	0.0007	0.0039	0.0096	0.1053	0.1053	0.1053
Shotgun Surgery	Ensemble	Soft Voting - Using SmoteTomek	0.4000	0.6081	0.7235	0.2778	0.5294	0.7330	0.7143	0.7143	0.7143
Shotgun Surgery	Boosting	CatBoost - Using PU Weighting	0.4444	0.5973	0.6678	0.3636	0.6257	0.8031	0.5714	0.5714	0.5714
Shotgun Surgery	Ensemble	Soft Voting - Using PU Weighting	0.3846	0.5957	0.7163	0.2632	0.5109	0.7183	0.7143	0.7143	0.7143
Shotgun Surgery	Boosting	Light GBM	0.3673	0.5644	0.6756	0.2571	0.5031	0.7119	0.6429	0.6429	0.6429
Shotgun Surgery	Boosting	CatBoost - Using SmoteTomek	0.4242	0.5577	0.6173	0.3684	0.6305	0.8064	0.5000	0.5000	0.5000
Shotgun Surgery	Boosting	XGBoost	0.3810	0.5548	0.6451	0.2857	0.5392	0.7407	0.5714	0.5714	0.5714
Shotgun Surgery	Boosting	XGBoost - Using SmoteTomek	0.3721	0.5483	0.6415	0.2759	0.5270	0.7312	0.5714	0.5714	0.5714
Shotgun Surgery	Boosting	XGBoost - Using PU Weighting	0.3721	0.5483	0.6415	0.2759	0.5270	0.7312	0.5714	0.5714	0.5714
Shotgun Surgery	Boosting	XGBoost - Using PU Weighting - Using SmoteTomek	0.3636	0.5420	0.6379	0.2667	0.5154	0.7219	0.5714	0.5714	0.5714
Shotgun Surgery	Ensemble	Soft Voting - Using PU Weighting - Using SmoteTomek	0.3396	0.5412	0.6617	0.2308	0.4674	0.6817	0.6429	0.6429	0.6429
Shotgun Surgery	Ensemble	Soft Voting	0.3333	0.5357	0.6583	0.2250	0.4592	0.6746	0.6429	0.6429	0.6429
Shotgun Surgery	Boosting	Light GBM - Using SmoteTomek	0.3404	0.5238	0.6274	0.2424	0.4835	0.6956	0.5714	0.5714	0.5714
Shotgun Surgery	Boosting	Light GBM - Using PU Weighting	0.3051	0.5098	0.6419	0.2000	0.4224	0.6409	0.6429	0.6429	0.6429
Shotgun Surgery	Boosting	CatBoost	0.3636	0.4909	0.5499	0.3158	0.5745	0.7672	0.4286	0.4286	0.4286
Shotgun Surgery	Boosting	CatBoost - Using PU Weighting - Using SmoteTomek	0.3636	0.4909	0.5499	0.3158	0.5745	0.7672	0.4286	0.4286	0.4286

Shotgun Surgery	Boosting	Light GBM - Using PU Weighting - Using SmoteTomek	0.2800	0.4529	0.5586	0.1944	0.4138	0.6328	0.5000	0.5000	0.5000
Shotgun Surgery	Adjusted Baseline	Association Rules - Using SmoteTomek	0.1399	0.3092	0.4918	0.0775	0.1973	0.3750	0.7143	0.7143	0.7143
Shotgun Surgery	Baseline	Association Rules	0.1389	0.3076	0.4901	0.0769	0.1960	0.3730	0.7143	0.7143	0.7143
Shotgun Surgery	Adjusted Baseline	Association Rules - Using PU Weighting - Using SmoteTomek	0.1342	0.2997	0.4818	0.0741	0.1896	0.3635	0.7143	0.7143	0.7143
Shotgun Surgery	Adjusted Baseline	Association Rules - Using PU Weighting	0.1233	0.2767	0.4474	0.0682	0.1763	0.3431	0.6429	0.6429	0.6429
Shotgun Surgery	One class classification	Isolation Forest - Using SmoteTomek	0.0028	0.0082	0.0197	0.0014	0.0041	0.0100	0.7143	0.7143	0.7143
Shotgun Surgery	One class classification	Isolation Forest	0.0028	0.0082	0.0197	0.0014	0.0041	0.0100	0.7143	0.7143	0.7143
Shotgun Surgery	One class classification	One Class SVM	0.0004	0.0012	0.0030	0.0002	0.0006	0.0015	0.0714	0.0714	0.0714
Shotgun Surgery	One class classification	One Class SVM - Using SmoteTomek	0.0004	0.0012	0.0030	0.0002	0.0006	0.0015	0.0714	0.0714	0.0714