UNIVERSITÉ DE MONTRÉAL

DEEP LEARNING STRUCTURAL AND HISTORICAL FEATURES FOR ANTI-PATTERNS DETECTION

ANTOINE BARBEZ
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DEEP LEARNING STRUCTURAL AND HISTORICAL FEATURES FOR ANTI-PATTERNS DETECTION

présenté par: BARBEZ Antoine
en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de:

Mme BOUCHENEB Hanifa, Doctorat, présidente
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et codirecteur de recherche
M. MERLO Ettore, Ph. D., membre

DEDICATION

To my beloved parents who always supported me,
and to kiki who was always there for me...

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to Prof. Foutse Khomh and Prof. Yann-Gaël Guéhéneuc for being my supervisor and co-supervisor respectively at Polytechnique Montréal and Concordia University.

Thank you Foutse Khomh for your guidance and for always pushing me in the right direction.

Thank you Yann-Gaël Guéhéneuc for your time, your advises and your constant kindness during these two years.

I would also like to thank the examining committee for taking the time to read and review this thesis.

I would also like to express my thanks to all the members of the SWAT and Ptidej teams, it has been a pleasure working with you. Particularly, I would like to thank Rodrigo Morales, Manel Abdellatif, Cristiano Politowski, Rubén Saborido Infantes, Satnam Singh, Vaibhav Jain, Prabhdeep Singh and Loïc Fontaine for helping me building my oracle.

Finally, I would like to thank my friend Akram for his precious advices.

RÉSUMÉ

Les systèmes logiciels sont devenus une des composantes principales de tous les secteurs d'activité. Dans la course au rendement économique, les développeurs sont susceptibles d'implémenter des solutions non optimales aux problèmes qui leur sont posés. On nomme ainsi anti-patrons ou "design smells" ces mauvais choix de conception introduits par manque de temps et–ou d'expérience. Si ces derniers n'ont pas forcément d'impact à l'exécution, de nombreuses études ont mis en lumière leur influence négative sur la maintenabilité des systèmes.

De nombreuses approches de détection automatique des anti-patrons ont été proposées. Pour la plupart, ces approches reposent sur l'analyse statique du code, mais il a été montré que les anti-pattrons sont aussi détectables par une analyse des données historiques des systèmes. Cependant, aucune d'entre elles ne semble clairement se distinguer des autres, et chaque approche identifie des ensembles d'occurrences différents, en particulier quand celles-ci reposent sur des sources d'information complémentaires (i.e., structurelles vs. historiques).

Plusieurs approches basées sur l'apprentissage automatique ont tenté d'adresser ce problème. Toutefois, ces approches semblent faire face à des limitations qui leur sont intrinsèques. D'une part, inférer des caractéristiques de haut niveau sur les systèmes à partir de données brutes nécessite des modèles d'une grande complexité. D'autre part, l'entrainement de tels modèles requière un nombre conséquent d'exemples d'apprentissage, qui sont fastidieux à produire et existent en nombre très limité.

Ce travail tire profit des méthodes d'apprentissage automatique pour répondre aux limitations évoquées précédemment. Dans un premier temps, nous proposons une méthode ensembliste permettant d'agréger plusieurs outils de détection. Nous montrons qu'une telle méthode atteint des performances nettement supérieures à celles des outils ainsi agrégés et permet de générer des instances d'apprentissage pour des modèles plus complexes à partir d'un nombre raisonnable d'exemples. Ensuite, nous proposons un modèle d'apprentissage profond pour la détection des anti-patrons. Ce modèle est basé sur l'analyse de l'évolution des métriques logicielles. Plus précisément, nous calculons les valeurs de certaines métriques pour chaque révision du système étudié, et, entrainons un réseau de neurones convolutif à y détecter les anti-patrons à partir de ces données. Nous montrons qu'en s'appuyant ainsi sur les aspects structurels et historiques des systèmes, notre modèle surpasse les approches existantes.

Nos approches ont été expérimentées dans le cadre de la détection de deux anti-patrons populaires : God Class et Feature Envy, et leurs performances comparées avec celles de l'état de l'art.

ABSTRACT

Software systems are constantly modified, whether to be adapted or to be fixed. Due to the exigence of economic performances, these modifications are sometimes performed in a hurry and developers often implement sub optimal solutions that decrease the quality of the code. In this context, the term "anti-pattern" have been introduced to represent such "bad" solutions to recurring design problems.

A variety of approaches have been proposed to identify the occurrences of anti-patterns in source code. Most of them rely on structural aspects of software systems but some alternative solutions exist. It has been shown that anti-patterns are also detectable through an analysis of historical information, i.e., by analyzing how code components evolve with one another over time. However, none of these approaches can claim high performances for any anti-pattern and for any system. Furthermore different approaches identify different sets of occurrences, especially when based on orthogonal sources of information (structural vs. historical).

Several machine-learning based approaches have been proposed to address this issue. However these approaches failed to surpass conventional detection techniques. On the one hand, learning high level features from raw data requires complex models such as deep neural-networks. On the other hand, training such complex models requires substantial amounts of manually-produced training data, which is hardly available and time consuming to produce for anti-patterns.

In this work, we address these issues by taking advantage of machine-learning techniques. First we propose a machine-learning based ensemble method to efficiently aggregate various anti-patterns detection tools. We show that (1) such approach clearly enhances the performances of the so aggregated tools and; (2) our method produces reliable training instances for more complex anti-pattern detection models from a reasonable number of training examples. Second we propose a deep-learning based approach to detect anti-patterns by analyzing how source code metrics evolve over time. To do so, we retrieve code metrics values for each revision of the system under investigation by mining its version control system. This information is then provided as input to a convolutional neural network to perform final prediction. The results of our experiments indicate that our model significantly outperforms existing approaches.

We experiment our approaches for the detection of two widely known anti-patterns: God Class and Feature Envy and compare their performances with those of state-of-the-art.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE OF CONTENTS vii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ABBREVIATIONS AND NOTATIONS xii

CHAPTER 1 INTRODUCTION 1
1.1 Context and Motivations 1
1.2 Research Objectives 2
1.3 Research Contributions 3
1.4 Thesis Structure 4

CHAPTER 2 BACKGROUND ON SUPERVISED LEARNING 5
2.1 Classification 5
2.2 Models 6
2.2.1 Logistic Regression 6
2.2.2 Multi-layer Perceptron 8
2.2.3 Convolutional Neural Network 10
2.3 Optimization 12
2.3.1 Loss Function 12
2.3.2 Gradient Descent 12
2.3.3 Hyper-parameters Calibration 13

CHAPTER 3 LITERATURE REVIEW 15
3.1 Definitions 15
3.1.1 God Class 15

3.1.2 Feature Envy . . . 15
3.2 Detection Techniques . . . 17
3.2.1 God Class . . . 17
3.2.2 Feature Envy . . . 20
3.3 Empirical Studies on Anti-patterns . . . 23
3.3.1 Impact of Anti-patterns on Software Quality . . . 23
3.3.2 Evolution and Presence of Anti-patterns . . . 24

CHAPTER 4 STUDY BACKGROUND . . . 26
4.1 Studied Systems . . . 26
4.2 Building a Reliable Oracle . . . 27
4.3 Evaluation Metrics . . . 28
4.4 Training . . . 29
4.4.1 Custom Loss Function . . . 29
4.4.2 Optimization . . . 30
4.4.3 Regularization . . . 31
4.4.4 Ensemble Learning . . . 32

CHAPTER 5 A MACHINE-LEARNING BASED ENSEMBLE METHOD FOR ANTI-PATTERNS DETECTION . . . 33
5.1 Introduction . . . 33
5.2 SMart Aggregation of Anti-pattern Detectors . . . 35
5.2.1 Baseline . . . 35
5.2.2 Overview . . . 36
5.2.3 Input . . . 37
5.3 Evaluation of the Detection Perfomances . . . 40
5.3.1 Study Design . . . 40
5.3.2 Parameters Calibration . . . 41
5.3.3 Analysis of the Results . . . 42
5.4 Evaluation of the Ability to Label Training Instances . . . 44
5.4.1 Study Design . . . 44
5.4.2 Parameters Calibration . . . 45
5.4.3 Analysis of the Results . . . 45
5.5 Conclusion and Future Work . . . 46

CHAPTER 6 DEEP-LEARNING ANTI-PATTERNS DETECTION FROM CODE METRICS HISTORY . . . 48

6.1 Introduction 48
6.2 CAME: Convolutional Analysis of Metrics Evolution 50
6.2.1 Overview 50
6.2.2 Input 51
6.2.3 Architecture 52
6.3 Experiments 53
6.3.1 Study Design 53
6.3.2 Hyper-parameters Calibration 54
6.3.3 Analysis of the Results 55
6.4 Conclusion and Future Work 56

CHAPTER 7 THREATS TO VALIDITY 58
7.1 Construct Validity 58
7.2 Internal Validity 58
7.3 External Validity 59

CHAPTER 8 CONCLUSION 60
8.1 Synthesis 60
8.2 Future Work 62

RÉFÉRENCES 63

LIST OF TABLES

Table 4.1 Characteristics of the Studied Systems 26
Table 4.2 Characteristics of the Oracle 28
Table 4.3 Confusion Matrix for Anti-patterns Detection 28
Table 5.1 Hyper-parameters Calibration of SMAD 41
Table 5.2 Hyper-parameters Calibration of the Competitive Tools 42
Table 5.3 Performances Evaluation of SMAD for God Class detection 42
Table 5.4 Performances Evaluation of SMAD for Feature Envy detection 42
Table 5.5 Characteristics of the Systems used to Generate Training Instances . 45
Table 5.6 Hyper-parameters Calibration of the Subject Model 45
Table 5.7 Performances of the Subject Model Trained on Injected vs Generated Smells 46
Table 6.1 Hyper-parameters Calibration of CAME 54
Table 6.2 Hyper-parameters Calibration of the Concurrent MLP 55
Table 6.3 Performances Evaluation of CAME for God Class detection 55

LIST OF FIGURES

Figure 2.1	Graph Representation of Logistic Regression	7
Figure 2.2	Graph Representation of Multi-class Logistic Regression	7
Figure 2.3	Graph Representation of an Artificial Neuron	8
Figure 2.4	Examples of Activation Functions	8
Figure 2.5	Architecture of a MLP model	9
Figure 2.6	Convolution of a 4×5 tensor by a 1×2 filter	10
Figure 2.7	2×2 max-pooling of a 4×4 feature map	11
Figure 2.8	Architecture of a CNN model	11
Figure 2.9	Gradient Descent Algorithm	13
Figure 3.1	Feature Envy example	16
Figure 3.2	Move Method Refactoring solution	16
Figure 3.3	Lanza and Marinescu (2007) detection rule for God Class.	18
Figure 3.4	Moha et al. (2010) Rule Card for God Class detection. (Hexagons are anti-patterns, gray ovals are code smells, and white ovals are properties).	18
Figure 3.5	Lanza and Marinescu (2007) detection rule for Feature Envy.	21
Figure 3.6	Liu et al. (2018) architecture for Feature Envy detection.	23
Figure 4.1	Comparison of feeding approaches: (a) gradient descent, (b) mini-batch SGD, (c) imbalanced-batch SGD. Colors represent instances belonging to same systems.	31
Figure 5.1	Overview of SMAD detection process	37
Figure 6.1	Workflow of CAME	50
Figure 6.2	Architecture of CAME's CNN	52

LIST OF ABBREVIATIONS AND NOTATIONS

Abbreviations

ANN	Artificial Neural Network
API	Application Programming Interface
ATFD	Access To Foreign Data
AST	Abstract Syntax Tree
BDTEX	Bayesian Detection Expert
CAME	Convolutional Analysis of Metrics Evolution
CNN	Convolutional Neural Network
CSV	Comma Separated Values
DECOR	DEtection and CORrection of Design Flaws
FDP	Foreign Data Providers
HIST	Historical Information for Smell deTection
LAA	Locality of Attribute Accesses
LCOM	Lack of Cohesion in Methods
LOC	Lines Of Code
MLP	Multi-layer Perceptron
NAD	Number of Attributes Declared
NMD	Number of Methods Declared
PADL	Pattern and Abstract-level Description Language
relu	rectified linear unit
SGD	Stochastic Gradient Descent
SHA	Secure Hash Algorithm
SMAD	SMart Aggregation of Anti-pattern Detectors
tanh	hyperbolic tangent
TCC	Tight Class Cohesion
UML	Unified Modeling Language
WMC	Weighted Method Count

Notations

K	number of classes
N	number of instances
m	number of attributes
c	class
x	attribute, $x \in \mathbb{R}$
$\mathbf{x}$	input vector (i.e., instance), $\mathbf{x} = [x_1, x_2, ..., x_m]$
y	label for a single class, $y \in [0, 1]$
$\mathbf{y}$	label for a multi-class problem, $\mathbf{y} = [y_1, y_2, ..., y_K], \quad \exists! k \quad \mid \quad y_k = 1$
$\mathcal{D}$	training set, $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$
w	weight of a neuronal connection, $w \in \mathbb{R}$
$\mathbf{w}$	weights vector, $\mathbf{w} = [w_0, w_1, ..., w_m]$
$\mathbf{W}$	weights matrix, $\mathbf{W} = [\mathbf{w}_1, \mathbf{w}_2, ..., \mathbf{w}_{M_l}]$
$\boldsymbol{\theta}$	set of weights, $\boldsymbol{\theta} = \{\mathbf{W}_l\}_{l=1}^{L+1}$
$\mathbf{f}$	output vector, $\mathbf{f} = [f_1, f_2, ..., f_k] = [P_{\boldsymbol{\theta}}(c_1\|\mathbf{x}_1), P_{\boldsymbol{\theta}}(c_2\|\mathbf{x}_2), ..., P_{\boldsymbol{\theta}}(c_K\|\mathbf{x}_K)]$
$p(.)$	probability distribution
$P(.)$	probability
L	number of hidden layers
M	hidden layer size
$L(.)$	loss function
η	learning rate
λ	L_2 regularization hyper-parameter
P_{keep}	probability to keep a neuron in dropout
$[\![.]\!]$	boolean variant of the Kroenecker delta, $[\![True]\!] = 1$ & $[\![False]\!] = 0$

CHAPTER 1 INTRODUCTION

1.1 Context and Motivations

Software systems are becoming one of the key components of every industry. Thus, maintaining software quality at a high level while constantly innovating is a major issue for every company. Software quality is impacted by the patterns applied and followed by developers. Among these patterns, design patterns and anti-patterns have been shown in the literature to impact the maintenance and evolution of systems.

Design patterns (Gamma et al. (1994)) and anti-patterns (Brown et al. (1998)) have been introduced to encode the "good" and "bad" design practices of experienced software developers. Design patterns describe solutions to common recurring design problems and promote flexibility and reusability. Design anti-patterns present common recurring design problems, i.e., "bad" solutions that decrease some quality characteristics, and suggest "good" alternative solutions.

Anti-patterns are typically introduced in the source code of systems when developers implement suboptimal solutions to their problems due for example to a lack of knowledge and–or time constraints. For instance, the God Class anti-pattern refers to the situation in which a class centralizes most of the system intelligence and implements a high number of responsibilities. Such classes appear when developers always assign new functionalities to the same class, thus breaking the principle of single responsibility.

There have been many empirical studies aiming to understand the effect of design anti-patterns on software systems. These works have highlighted their negative impact on software comprehension (Abbes et al. (2011)), fault-proneness (Khomh et al. (2012)) and maintainability (Yamashita and Moonen (2013)).

In response to these works, several strategies have been proposed to identify the occurrences of anti-patterns in software systems. Most of these works describe anti-patterns using structural metrics (e.g., cyclomatic complexity or lines of code) and attempt to identify bad motifs in models of the source code by defining thresholds to apply to the value of these metrics. For example, Moha et al. (2010) proposed a domain-specific language to describe and generate detection algorithms for anti-patterns using structural and lexical metrics. Alternative solutions have also been proposed to detect anti-patterns using others aspects of software systems. For instance, Palomba et al. (2013, 2015a) have shown that number of anti-patterns impact how source code entities, i.e., classes and methods, evolve with one another when

changes are applied to the systems. Therefore, they proposed a set of rules designed to identify occurrences of anti-patterns from co-changes occurring between source code entities.

Even though these approaches have shown acceptable detection performances, they still exhibit large numbers of false positives and misses, and none of them seem to truly stand out among others. Besides, a low agreement can be observed between different approaches (Fontana et al. (2012)) because each of them is based on a different definition of anti-patterns. Thus, each approach can identify occurrences that cannot be detected by others, especially when they rely on orthogonal sources of information (Palomba et al. (2013)).

Recently, machine-learning models have been shown efficient in a variety of domains. Specifically, deep neural-networks have completely redefined the fields of speech recognition (Graves et al. (2013)), image processing (Krizhevsky et al. (2012)) or sentiment analysis (dos Santos and Gatti (2014)). This success stands on their ability to extract "deep features" i.e., high level characteristics, from complex data. Several machine-learning based approaches have been proposed to detect anti-patterns. For example, Maiga et al. (2012a,b) proposed the use of support vector machines (SVM) for the detection of God Class, Functional Decomposition, Spaghetti code, and Swiss Army Knife, while Liu et al. (2018) proposed a deep-learning model to detect Feature Envy. However, these approaches failed to surpass clearly conventional detection techniques.

We identify two main reasons to the limitations faced by machine-learning models in detecting anti-patterns. First, as shown by Palomba et al. (2018), software systems are usually affected by a small proportion of anti-patterns (<1%). Consequently, the repartition of labels (i.e., *Affected* or *Healthy*) within the data is highly imbalanced, which have been shown to compromise the performances of machine-learning models (He and Garcia (2008)). Second, training complex models such as deep neural-networks requires substantial amounts of training data, i.e., manually-validated examples of *Affected* and *Healthy* components, which is hardly available and time consuming to produce for anti-patterns.

1.2 Research Objectives

This master's thesis focuses on the detection of design anti-patterns by leveraging both structural and historical information. In particular, we propose two novel machine-learning detection approaches and we implement them for two widely known anti-patterns: God Class and Feature Envy. This work also addresses the issues evoked in the previous section as follows: (1) we created an oracle reporting the occurrences of the studied anti-patterns in eight open-source Java projects and; (2) we propose a training procedure designed to address

the imbalanced data problem. In a first part, we propose a machine-learning based ensemble method to efficiently aggregate various anti-patterns detection tools based on different strategies and–or sources of information. We show that (1) such approach significantly enhances the performances of the so aggregated tools and; (2) our method produces reliable training instances for more complex anti-pattern detection models from a reasonable number of training examples. In a second part, we propose a deep-learning based approach to detect anti-patterns by analyzing how source code metrics evolve over time. To do so, we retrieve code metrics values for each revision of the system under investigation by mining its version control system. This information is then provided as input to a convolutional neural network to perform final prediction. The results of our experiments indicate that our model significantly outperforms state-of-the-art detection tools. Thus, this work answers the following research questions:

- How can the imbalanced data problem be addressed for anti-patterns detection?
- How can we leverage existing detection tools to automatically label training instances for machine-learning anti-patterns detection models?
- How can we leverage machine-learning techniques to detect anti-patterns using both structural and historical informations?
- How does such approaches compare with state-of-the-art?

1.3 Research Contributions

This thesis investigates the use of machine-learning techniques to detect anti-patterns from both structural and historical sources of information. With the results of our experiments, we make the following contributions:

1. An oracle reporting the occurrences of God Class and Feature Envy in eight Java software systems.
2. A procedure to train feed-forward neural-networks to detect anti-patterns.
3. A machine-learning based ensemble method to efficiently aggregate existing anti-patterns detection approaches.
4. A procedure to automatically generate training data for machine-learning anti-patterns detection models.

5. A deep-learning model that rely on code metrics evolution to detect anti-patterns.
6. An implementation of our approaches for the detection of God Class and Feature Envy and a comparison of their performances with state-of-the-art.

1.4 Thesis Structure

This thesis is organized as follows. Chapter 2 presents a background on neural-networks and provides necessary information to understand the models used in this work. Chapter 3 overviews the related literature on design anti-patterns. Chapter 4 presents the background common to our studies as well as the building of our oracle and our training procedure. Chapter 5 presents our first study on the aggregation of various detection approaches while Chapter 6 presents our deep-learning model for anti-patterns detection from code metrics history. Finally, Chapter 7 discusses the threats that could affect the validity of our studies and Chapter 8 concludes and discusses future work.

CHAPTER 2 BACKGROUND ON SUPERVISED LEARNING

2.1 Classification

Classification is the task of arranging the elements of a set into different classes $c_1, c_2, ..., c_K$. Formally, we dispose of a set of N samples x_i, $i \in \{1, 2, ..., N\}$ and we want to assign a label $y_i \in \{c_1, c_2, ..., c_K\}$ to each sample. To perform this classification, each sample x_i is characterized by a set of m numerical attributes (i.e., features) $x_{ij} \in \mathbb{R}$, $j \in \{1, 2, ..., m\}$ and the values of this set of attributes for a given sample is commonly referred to as *instance*. Then, the task of classification consists in finding a function f that predicts the label of an instance from its input attributes, which can be expressed as:

$$y_i = f(x_{i1}, x_{i2}, ..., x_{im}) \tag{2.1}$$

Real world problems are rarely deterministic and are often described using probabilities. Thus, the problem of classification is usually addressed by predicting a probability for each class and taking the one with the maximum value. In this context, a probabilistic classification model outputs the labels conditional probability distribution for a given element:

$$p(y_i|x_{i1}, x_{i2}, ..., x_{im}) = f(x_{i1}, x_{i2}, ..., x_{im}) \tag{2.2}$$

In a typical scenario, the set $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ of the instances and their associated labels is called the *training set*. We want to learn the conditional distribution given in Equation 2.2 as a parametric function of the attributes. Our goal is to find the parameters of this function that lead to the best description of the probability distribution among instances and labels in the training set. Then, when a new unobserved instance has to be classified, the so "trained" model is used to predict which label is the most probable given its attributes.

Anti-patterns detection can be seen as a binary classification problem, where entities (classes or methods) of a software system need to be classified between two labels: *Affected* or *Healthy*. In the following sections, we first describe the models used to perform classification before discussing the optimization of such models, commonly known as *training*.

2.2 Models

This section describes the machine-learning models used in the context of our studies to perform binary classification.

2.2.1 Logistic Regression

Logistic regression is a statistical technique for predicting the probability distribution for a single class c_k. Hence, the variable of interest here is the boolean variable $y_i = c_k$ which takes the value 1 if the input instance belongs to c_k and 0 for other classes. Logistic regression relies on the fundamental hypothesis that the log-odds of this variable can be expressed as a linear function of the attributes:

$$\log\left(\frac{P(1|x_{i1}, x_{i2}, ..., x_{im})}{1 - P(1|x_{i1}, x_{i2}, ..., x_{im})}\right) = w_0 + w_1 x_{i1} + w_2 x_{i2} + ... + w_m x_{im} \tag{2.3}$$

which can be expressed in vector form as:

$$\log\left(\frac{P(1|\mathbf{x}_i)}{1 - P(1|\mathbf{x}_i)}\right) = \mathbf{x}_i^\top \mathbf{w} \tag{2.4}$$

with the left-hand side of Equation 2.4, the log-odds (or logit transformation) for $y_i = c_k$, $\mathbf{x}_i = [1, x_{i1}, ..., x_{im}]^\top$ the input vector of the i^{th} instance and $\mathbf{w} = [w_0, w_1, ..., w_m] \in \mathbb{R}^m$ the vector of *weights*. The advantage of such expression is that the logit transformation maps the input probability, which is bounded in $[0, 1]$, into a real value. Thus, we can solve Equation 2.4 for $P(1|\mathbf{x}_i)$ to obtain the desired form for a binary classification expressed in Equation 2.2:

$$P(1|\mathbf{x}_i) = \frac{1}{1 + \exp(-\mathbf{x}_i^\top \mathbf{w})} = sigmoid(\mathbf{x}_i^\top \mathbf{w}) \tag{2.5}$$

Figure 2.1 illustrates the process of logistic regression in a graphical form.

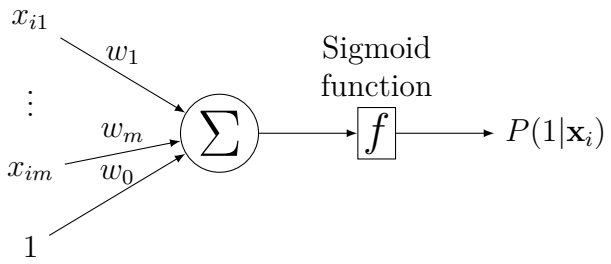

Figure 2.1 Graph Representation of Logistic Regression

Consider now that we want to express the conditional probability distribution of the variable y_i among the different classes, i.e., the K probabilities that $\mathbf{x}_i$ belongs to each class c_k. In this context, a *multi-class logistic regression* generalizes Equation 2.5 as follows:

$$P(y_i = c_k|\mathbf{x}_i) = \frac{\exp(\mathbf{x}_i^\top \mathbf{w}_k)}{\sum_{j=1}^{K} \exp(\mathbf{x}_i^\top \mathbf{w}_j)} = softmax(\mathbf{x}_i^\top \mathbf{w}_k) \tag{2.6}$$

With $\mathbf{w}_j = [w_{j0}, w_{j1}, ..., w_{jm}]$ the weights associated with the j^{th} class. Figure 2.2 illustrates this process in a graphical form. Note that for the sake of readability, we did not represent the weights associated to each connection.

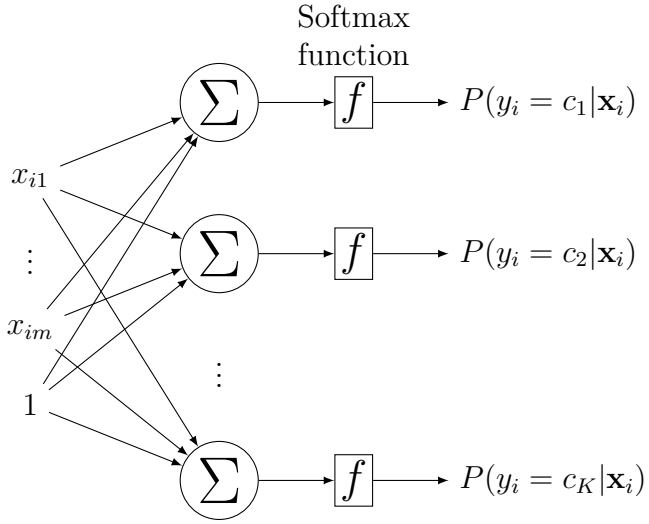

Figure 2.2 Graph Representation of Multi-class Logistic Regression

It is interesting to remark that there are two ways of performing binary classification, i.e., $y_i \in \{0, 1\}$. The first one consists in the operation illustrated in Figure 2.1 where the output is one single real value: $P(1|\mathbf{x}_i)$ using the *sigmoid* function. The second one considers binary classification as a two-class classification problem which uses the *softmax* function to output a vector of two values: $[P(0|\mathbf{x}_i), P(1|\mathbf{x}_i)]$.

2.2.2 Multi-layer Perceptron

A multi-layer perceptron (MLP) is a type of feed-forward artificial neural-network (ANN). ANNs, are a family of probabilistic models that can be seen as a mathematical abstraction of the biological nervous system. An ANN is composed of elementary units called *"neurons"*. As shown in Figure 2.3, an artificial neuron takes as input a set of numerical values. These input values are then summed and passed as input to an *"activation function"* that returns the output of the neuron.

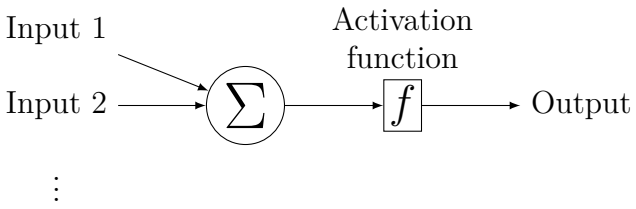

Figure 2.3 Graph Representation of an Artificial Neuron

There exists a variety of activation functions. As shown in Figure 2.4, most of them are step-like functions which reminds us of the behavior of a biological neuron, i.e., the neuron outputs a positive value only if the sum of the inputs is greater than a given threshold.

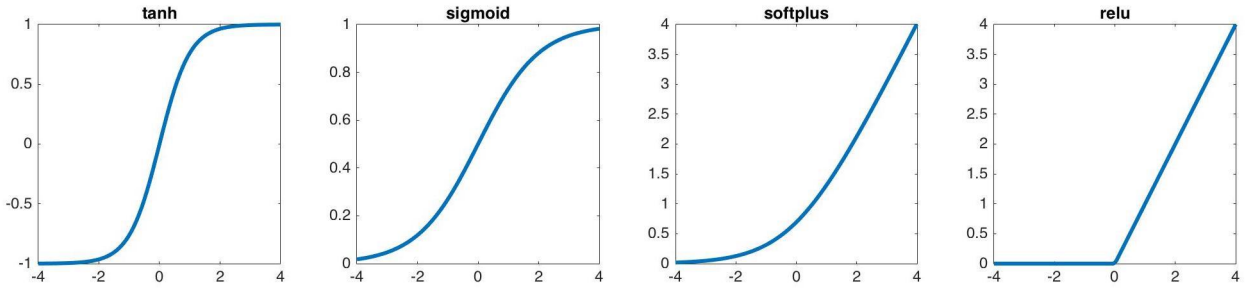

Figure 2.4 Examples of Activation Functions

The architecture of a MLP model is organized as follows:

1. Neurons are organized in successive fully-connected layers (or dense layers), i.e., the output of a neuron is connected to all the neurons of the next layer and there exists no connections between neurons of a same layer.

2. Connections between neurons are weighted. Thus the input of a given neuron is a linear function of the outputs of all the neurons of the previous layer.

3. The *input layer* (i.e., the first layer) represents the attributes of the data used to perform classification.

4. The *output layer* (i.e., the last layer) represents the conditional probabilities for each label predicted by the model.

5. The *hidden layers* (i.e., the layers in between) have no direct connections with the environment.

Let us illustrate such architecture with the MLP presented in Figure 2.5. This model takes as input m attributes and is composed of L hidden layers. It outputs a vector of K probabilities. We note M_l, $l \in \{1, 2, ..., L\}$ the size (i.e., number of neurons) of the l^{th} hidden layer. Activation functions are represented inside the neurons. Here, hidden neurons have a *relu* activation function and the output neuron a *softmax*, thus outputting a probability.

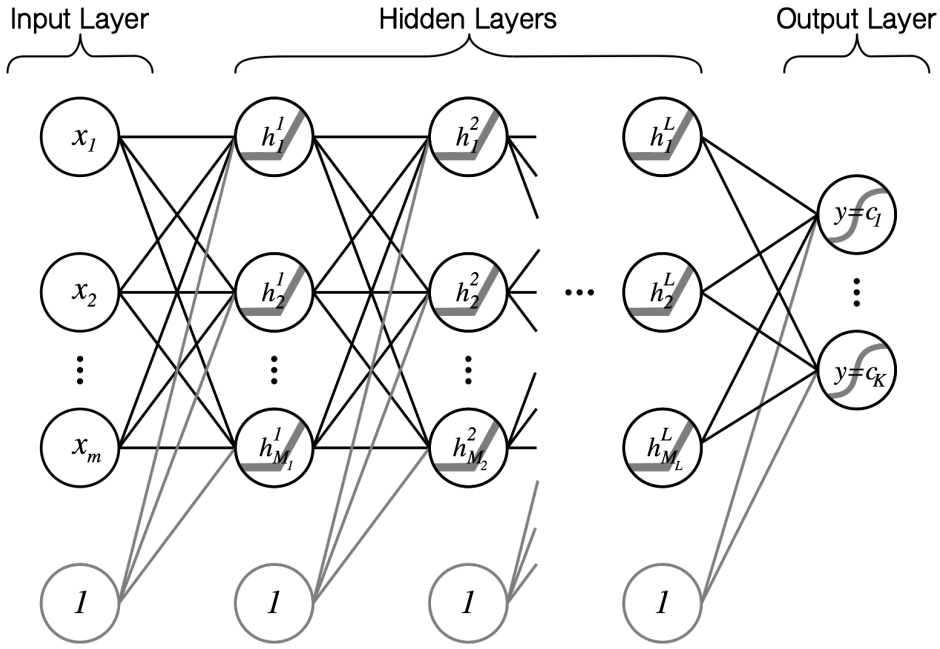

Figure 2.5 Architecture of a MLP model

Note that input and hidden layers have an additional constant node called the *bias*. This node allows each neuron to receive as input a linear function of the output of the neurons of the previous layer. One can also remark that logistic regression is in fact equivalent to a MLP with no hidden layers.

2.2.3 Convolutional Neural Network

Convolutional Neural Networks are a special kind of feed-forward ANNs. These networks have proved to be extremely efficient to process multi-dimensional inputs such as images. Indeed, these networks happen to have a similar architecture than that of the human and animal visual cortex (Hubel and Wiesel (1962)). CNNs have originally been proposed by LeCun et al. (1998). However, their great potential for image processing have only been recognized by the community after the deep CNN proposed by Krizhevsky et al. (2012) achieved breakthrough results at the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC). CNNs are characterized by the use of successive so called *convolution layers* directly after the input layer. Then, the output of these convolution layers is usually fully-connected to a MLP model (i.e., dense layers).

Convolution Layer

A convolution layer takes as input a multi-dimensional array of numbers (typically an image) called a *tensor* and returns several filtered versions (called *feature maps*) of this input. Therefore, a convolution layer contains several fixed-size filters and outputs the filter's response at each spatial location of the input. A convolution filter can be seen as an artificial neuron that takes as input a fixed-size portion of the input tensor and pass the weighted sum of its inputs to an activation function. Figure 2.6 illustrate the process of filtering of the input also called *convolution*. In this example, a 4×5 input tensor is filtered by a 1×2 convolution filter with a *relu* activation function, which produces a 4×4 output.

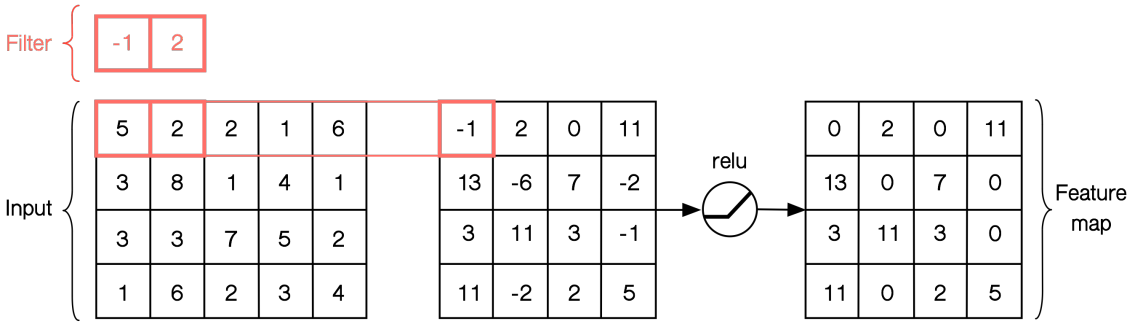

Figure 2.6 Convolution of a 4×5 tensor by a 1×2 filter

Pooling Layer

Once the input has been filtered by several filters, feature maps are often aggregated across a small spatial region to reduce their dimensionality. This process called *pooling* is usually done using the average or maximum value. Figure 2.7 illustrate a 2×2 max-pooling operation across the feature map presented in Figure 2.6.

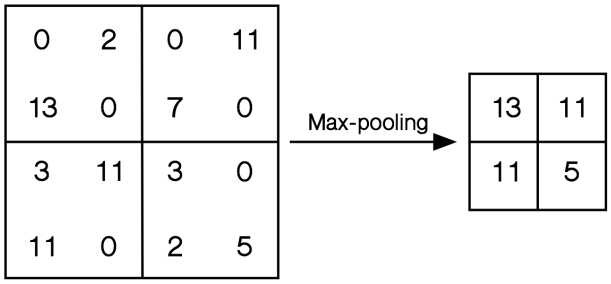

Figure 2.7 2×2 max-pooling of a 4×4 feature map

Thus, in a CNN, the input tensor is aggregated into several smaller tensors through multiple successive convolution + pooling operations. These output tensors are then flattened and concatenated to feed a MLP which performs the final prediction. Figure 2.8 overviews the whole process with a CNN composed of two convolution + pooling layers, one dense layer and three output neurons.

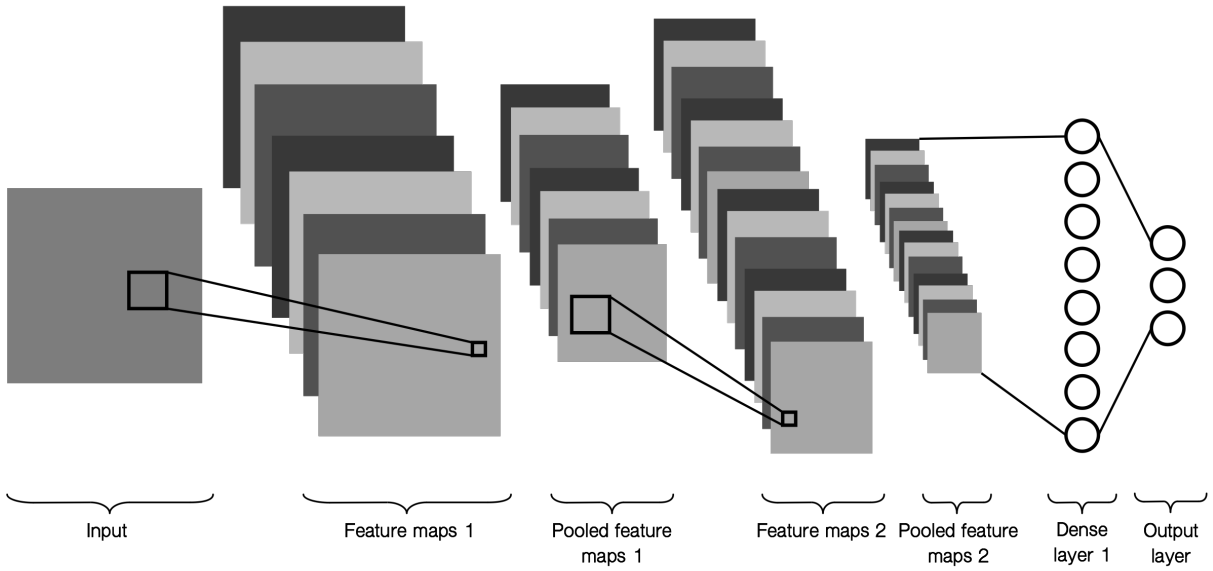

Figure 2.8 Architecture of a CNN model

2.3 Optimization

The previous section focuses on the structure of the neural-networks used for classification. In this section, we address the problem of training such models, i.e., finding the set of weights that minimizes the error achieved by a model on a set of input–output examples. In the remainder of this section, we consider having a *training set* $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ of instances and their known associated labels. We note $\boldsymbol{\theta} = \{\mathbf{W}_l\}_{l=1}^{L+1}$ the set of weights of the model we want to train.

2.3.1 Loss Function

We have seen how neural networks map an input vector $\mathbf{x}_i$ of real values into an output vector $\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta})$ of probabilities. Training a neural-network model, is the process of finding an optimal set of weights $\boldsymbol{\theta}^*$ that minimizes the error (or *loss*) obtained by the model on the examples of $\mathcal{D}$. Thus, one must define a loss function $L(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}), \mathbf{y}_i)$ that measures how "bad" is the model's prediction for a given example $(\mathbf{x}_i, \mathbf{y}_i)$. Then, once the loss function is defined, we want to minimize the mean loss performed by the model over every example of the training set, which is called the *empirical risk*:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}} \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}), \mathbf{y}_i) \tag{2.7}$$

There exists a variety of loss functions. The choice of the right loss to use for a given classification problem depends on the nature of this problem and how we want our model to behave. Equations 2.8 and 2.9 presents two commonly used loss functions, the *cross entropy* and the *squared error*:

$$cross\_entropy = -\sum_{k=1}^{K} [y_{ik} \log(f_k(\mathbf{x}_i, \boldsymbol{\theta})) + (1 - y_{ik}) \log(1 - f_k(\mathbf{x}_i, \boldsymbol{\theta}))] \tag{2.8}$$

$$squared\_error = \sum_{k=1}^{K} (f_k(\mathbf{x}_i, \boldsymbol{\theta}) - y_{ik})^2 \tag{2.9}$$

2.3.2 Gradient Descent

Gradient descent, is a procedure that allows to minimize the empirical risk by incrementally updating the model weights. At each *epoch*, i.e., pass over the whole training set, the weights are updated according to the gradient of the empirical risk computed with respect to the model weights. This process is repeated until the computed value of the empirical risk has

converged. Figure 2.9 shows the algorithm of gradient descent in pseudocode. The hyper-parameter η is called the *learning rate*. It represents the length of the step by which the weights are updated in the opposite direction to that of the gradient.

$\boldsymbol{\theta} = \boldsymbol{\theta_0}$; // weights initialization
while *converged == FALSE* **do**
  $\mathbf{g} = \frac{\partial}{\partial \boldsymbol{\theta}}[\frac{1}{N}\sum_{i=1}^{N} L(\mathbf{f}(\mathbf{x}_i, \boldsymbol{\theta}), \mathbf{y}_i)]$;
  $\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \mathbf{g}$;
end

Figure 2.9 Gradient Descent Algorithm

In gradient descent, weights are only updated after all training examples have been fed through the model. However, weights can also be updated according to the gradient of the loss function computed for each training example, which is called stochastic gradient descent (SGD). Another variant of gradient descent called mini-batch stochastic gradient descent consists in splitting the training set in equal size subsets called *mini-batch*. Then, at each epoch, the training set is split into new mini-batches and the weights are updated according to the gradient of the empirical risk computed for each mini-batch.

2.3.3 Hyper-parameters Calibration

In the previous subsections, we have seen how to find an optimal set of weights $\boldsymbol{\theta}^*$ by incrementally updating $\boldsymbol{\theta}$ according to the gradient of the loss. However, before training a neural-network, one must also find the optimal set of hyper-parameters for the model. For example, the number of layers in the network (L), the size of each layer (M_l) as well as the learning rate (η) are common hyper-parameters that must be assessed before training.

To do so, a common approach consists in keeping a portion of the training set (usually 30%) called the *validation set* to monitor the performances achieved by the model with different sets of hyper-parameters. Hence, for each set of hyper-parameters to test, the model is trained on the new training set (70%) and the performances are tested on the validation set (30%). Once the optimal set of hyper-parameters has been decided, it is a common practice to perform the final training of the model on the whole set (100%) to maximize the number of training examples. However, in this approach the validation set is selected randomly which can lead to a wrong choice of hyper-parameters if the 30% sample selected is not representative of the data. An alternative strategy is called the *k-fold cross validation*. It consists in splitting the training set into k equal size partitions called *folds*. Afterwards, each

set of hyper-parameters is tested k times by leaving one fold out for testing and keeping the remaining $k-1$ for training. Finally, the performance value retained for the tested hyper-parameters is computed by taking the mean across the k generated values. Although k-fold cross validation usually leads to a better hyper-parameters calibration, it can be tedious to execute in practice, especially for deep-learning models that can take days to train.

The values of the hyper-parameters to test are usually selected using a random search, which simply consists in randomly selecting the value for each hyper-parameter inside a predefined range. This technique has proved to be more efficient than grid search (Bergstra and Bengio (2012)).

CHAPTER 3 LITERATURE REVIEW

During the past decade, the use of machine-learning has allowed great improvements in a variety of domains, and of course, the field of anti-patterns detection has not been immune from it. In this chapter, we first define the two anti-patterns considered in this thesis, then, we present a literature review of (1) conventional techniques used to detect these anti-patterns as well as their machine-learning counterparts and; (2) empirical studies conducted on the impact of anti-patterns on software systems.

3.1 Definitions

3.1.1 God Class

A God Class or Blob, is a class that tends to centralize most of the system's intelligence, and implements a high number of responsibilities. It is characterized by the presence of a large number of attributes, methods and dependencies with data classes (i.e., classes only used to store data in the form of attributes that can be accessed via getters and setters). Thus, assigning much of the work to a single class, delegating only minor operations to other small classes causes a negative impact on program comprehension (Abbes et al. (2011)) and reusability. The alternative refactoring operation commonly applied to remove this anti-pattern is called Extract Class Refactoring and consists in splitting the affected God Class into several more cohesive smaller classes (Fowler (1999)).

3.1.2 Feature Envy

A method that is more interested in the data of another class (the envied class) than that of the class it is actually in. This anti-pattern represents a symptom of the method's misplacement, and is characterized by a lot of access to foreign attributes and methods. The main consequences are an increase of coupling and a reduction of cohesion, because the affected method often implements responsibilities more related to the envied class with respect to the methods of its own class. This anti-pattern is commonly removed using Move Method Refactoring, which consists in moving all or parts of the affected method to the envied class (Fowler (1999)). Let us consider the situation where a class `UseRectangle` needs to compute the area of an instance of a class `Rectangle`. In the implementation presented in Figure 3.1, the class `UseRectangle` implements a method `getArea(Rectangle)` to compute the desired area from the public attributes provided by the class rectangle instead of asking the object

to do the computation himself. This is a clear case where the method `getArea(Rectangle)` envies the class `Rectangle`. As shown in Figure 3.2 this issue can be addressed by moving the envious method to the envied class thus keeping the attributes `width` and `height` private.

```
import Rectangle;

class UseRectangle {
   private int area;

   public UseRectangle(Rectangle r) {
      this.area = getArea(r);
   }

   private int getArea(Rectangle r) {
      int width = r.width;
      int height = r.height;
      return width*height;
   }
    ...
}
```

```
class Rectangle {
   public int width;
   public int height;

   public Rectangle(int w, int h) {
      this.width = w;
      this.height = h;
   }
}
```

Figure 3.1 Feature Envy example

```
import Rectangle;

class UseRectangle {
   private int area;

   public UseRectangle(Rectangle r) {
      this.area = r.getArea();
   }
    ...
}
```

```
class Rectangle {
   private int width;
   private int height;

   public Rectangle(int w, int h) {
      this.width = w;
      this.height = h;
   }

   public getArea() {
      return width*height;
   }
}
```

Figure 3.2 Move Method Refactoring solution

3.2 Detection Techniques

The idea of anti-patterns or design smells has first been introduced by Webster (1995) to capture the pitfalls of object oriented development. Since then, number of books have been written to define new anti-patterns, sharpen the definition of existing ones, and propose alternative refactoring solutions. Among these books, Fowler (1999) wrote a taxonomy of 22 design and code smells and discussed that such smells are indicators of design or implementation issues to be addressed by refactorings. Lanza and Marinescu (2007) provided a metric oriented approach to characterize, evaluate and improve the design of object oriented systems. Suryanarayana (2014) described 25 structural design smells contributing to technical debt in software projects. Based on the definitions provided by these books, several automatic detection approaches have been proposed in the literature to detect instances of anti-patterns in source code. Later, as in many research fields, machine learning techniques have been used to overcome the performance issues encountered by the previous approaches. In the following, we present the detection approaches that have been proposed in literature to identify the occurrences of the two anti-patterns considered in this thesis.

3.2.1 God Class

Heuristic Based Detection Approaches

The first attempts to detect components affected by anti-patterns in general, and God Classes in particular have focused on the definition of rule-based approaches which use some metrics to capture deviations from good object oriented design principles. First, Marinescu (2004) presented *detection strategy*, a metric-based mechanism for analyzing source code models and detect design fragments using a quantifiable expression of a rule. They illustrate their methodology step by step by defining the detection strategy for God Class. Later, Lanza and Marinescu (2007) formulated their detection strategies for 11 design and code smells by designing a set of metrics for each smell along with thresholds. These metrics are combined with their respective thresholds to create the final detection rules for each anti-pattern. As described in Figure 3.3, God Class occurrences are detected using a set of three metrics, namely ATFD (Access To Foreign Data), WMC (Weighted Method Count), and TCC (Tight Class Cohesion). These heuristics have then been implemented to create anti-pattern detection tools like InCode (Marinescu et al. (2010)).

Similar to the approach described above, Moha et al. (2010) performed a systematic analysis of the definitions of code and design smells in the literature and proposed templates and a grammar to encode these smells and generate detection algorithms automatically. Based

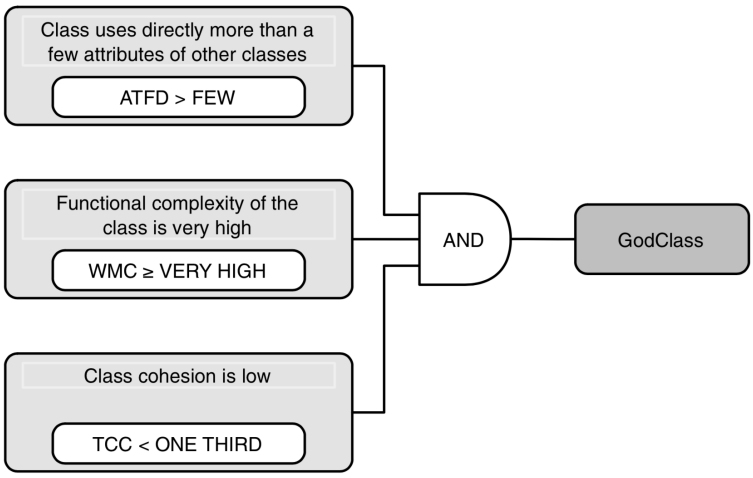

Figure 3.3 Lanza and Marinescu (2007) detection rule for God Class.

on such analysis, they proposed the detection tool DECOR (DEtection and CORrection of Design Flaws), implemented for four design anti-patterns: God Class, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells. Their detection approach takes the form of a "Rule Card" which encodes the formal definition of anti-patterns and code smells. As described in Figure 3.4 the identification of classes affected by God Class is based on both structural and lexical information.

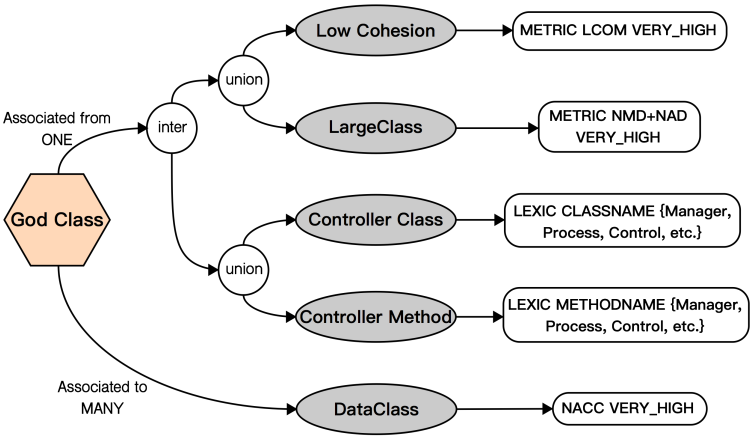

Figure 3.4 Moha et al. (2010) Rule Card for God Class detection. (Hexagons are anti-patterns, gray ovals are code smells, and white ovals are properties).

Other approaches rely on the identification of refactoring opportunities to detect anti-patterns. Based on this consideration, instances of a given anti-pattern can be detected in a system by looking at the opportunities to apply the corresponding refactoring operation. In this context, Fokaefs et al. (2012) proposed an approach to detect God Classes in a system by suggesting a set of Extract Class Refactoring operations. This set of refactoring opportu-

nities is generated in two main steps. First, they identify cohesive clusters of entities (i.e., attributes and methods) in each class of the system, that could then be extracted as separate classes. To do so, the Jaccard distance is computed among each class members (i.e., entities). The Jaccard distance between two entities e_i and e_j measures the dissimilarity between their respective *"entity sets"* S_i and S_j and is computed as follows:

$$dist(e_i, e_j) = 1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|} \tag{3.1}$$

For a method, the *"entity set"* contains the entities accessed by the method, and for an attribute, it contains the methods accessing the attribute. Then, cohesive groups of entities are identified using a hierarchical agglomerative algorithm on the information previously generated. In the second step, the potential classes to be extracted are filtered using a set of rules, to ensure that the behavior of the original program is preserved. Later, this approach has been implemented as an Eclipse plug-in called *JDeodorant* (Fokaefs et al. (2011)).

The approaches described above are solely based on structural information to predict whether an entity is affected of not by an anti-pattern. However, anti-patterns can also impact how source code entities change together over time. Based on such considerations, Palomba et al. (2013, 2015a) proposed HIST (Historical Information for Smell deTection), an approach to detect anti-patterns occurrences in systems using historical information derived from version control systems (e.g., Git, SVN). They applied their approach to the detection of five anti-patterns: Divergent Change, Shotgun Surgery, Parallel Inheritance, God Class and Feature Envy. The detection process followed by HIST consists of two steps. First, historical information is extracted from versioning systems using a component called *change history extractor* which outputs the sequence of changes applied to source code entities (i.e., classes or methods) through the history of the system. Second, a set of rules is applied to this so produced sequence to identify occurrences of anti-patterns. In this context, God Classes are identified as: "*classes modified (in any way) in more than α% of commits involving at least another class*", with a value of α set to 8% after parameter calibration.

Machine-learning Based Detection Approaches

The approaches described above detect God Classes among other classes of a system using manually-defined heuristics, while a number of machine-learning based approaches have been proposed in the past decade. First, Kreimer (2005) proposed the use of decision trees to identify occurrences of God Class and Long Method. Their model relies on the number of fields, number of methods, and number of statements as decision criteria for God Class

detection and have been evaluated on two small systems (IYC and WEKA). This observation has been confirmed 10 years later by Amorim et al. (2015) who extended this approach to 12 anti-patterns.

Khomh et al. (2009a, 2011) presented BDTEX (Bayesian Detection Expert), a metric based approach to build Bayesian Belief Networks from the definitions of anti-patterns. This approach has been validated on three different anti-patterns (God Class, Functional Decomposition, and Spaghetti Code) and provides a probability that a given entity is affected instead of a boolean value like other approaches. Following, Vaucher et al. (2009) relied on Bayesian Belief Networks to track the evolution of the"godliness" of a class and thus, distinguishing real God Classes from those that are so by design.

Later, Maiga et al. (2012a,b) introduced SVMDetect, an approach based on Support Vector Machines to detect four well known anti-patterns: God Class, Functional Decomposition, Spaghetti code, and Swiss Army Knife. The input vector fed into their classifier for God Class detection is composed of 60 structural metrics computed from the PADL meta-model (Guéhéneuc (2005)).

Fontana et al. (2016) performed the largest experiment on the effectiveness of machine learning algorithms for smell detection. They conducted a study where 16 different machine learning algorithms were implemented (along with their boosting variant) for the detection of four smells (Data Class, God Class, Feature Envy, and Long Method) on 74 software systems belonging to the `Qualitas Corpus` dataset (Tempero et al. (2010)). The experiments have been conducted using a set of independent metrics related to class, method, package and project level as input information and the datasets used for training and evaluation have been filtered using an under-sampling technique (i.e., instances have been removed from the original dataset) to avoid the poor performances commonly reported from machine learning models on imbalanced datasets. Their study concluded that the algorithm that performed the best for God Class detection was the J48 decision tree algorithm with an F-measure of 99%. However, Di Nucci et al. (2018) replicated their study and highlighted many limitations. In particular, the way the datasets used in this study have been constructed is strongly discussed and the performances achieved after replication were far from those originally reported.

3.2.2 Feature Envy

Heuristic Based Detection Approaches

As for other anti-patterns, the first approaches proposed to detect Feature Envy are based on manyally-defined heuristics that rely on some metrics. First, Lanza and Marinescu (2007)

proposed the *detection strategy* illustrated in Figure 3.5 which rely on: (1) the number of Accesses To Foreign Data (ATFD) made by a method; (2) the Locality of Attribute Accesses (LAA), i.e., ratio between the number of accesses to attributes that belongs to the envied class vs. the enclosing class and; (3) the number of Foreign Data Providers (FDP) i.e., the number of classes accessed in the body of a method. One must remark that this approach focuses only on predicting whether or not a method is involved in the Feature Envy anti-pattern but does not provide any information about the envied class.

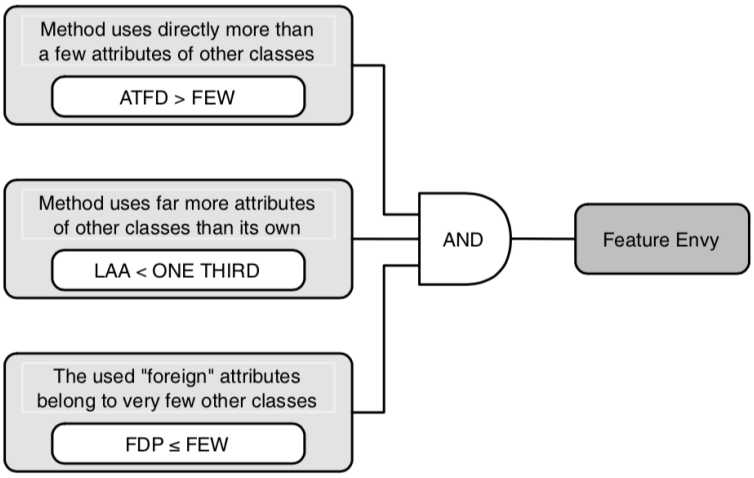

Figure 3.5 Lanza and Marinescu (2007) detection rule for Feature Envy.

Similarly to this work, Nongpong (2015) proposed the *Feature Envy Factor*, a metric for automatic Feature Envy detection. This approach relies on counting the number of calls made on a given object by the method under investigation, in order to produce a metric assess from zero to one how good is the Feature Envy candidate. The *Feature Envy Factor* between an object *obj* and a method *mtd* is computed as follows:

$$FEF(obj, mtd) = w(m/n) + (1 - w)(1 - x^m) \tag{3.2}$$

Where m is the number of calls on the object *obj*; n is the total number of calls on any objects defined or visible by the method *mtd*; w and x are real values in the range $[0, 1]$.

It is also possible to detect occurrences of Feature Envy by looking at the opportunities to apply the corresponding refactoring operation. Methods that can potentially be moved to another class under certain conditions are presented to the software engineer as potentially affected components. In this context, Tsantalis and Chatzigeorgiou (2009) proposed an approach for automatic suggestions of Move Method Refactoring. First, for each method m in the system, a set of candidate target classes T is created by examining the entities that are accessed in the body of m. Second, T is sorted according to two criteria: (1) the number of

entities that m accesses from each target class of T in descending order and; (2) the Jaccard distance from m to each target class in ascending order if m accesses an equal number of entities from two or more classes. In this context, the Jaccard distance between an entity e and a class C is computed as follows:

$$dist(e, C) = 1 - \frac{|S_e \cap S_C|}{|S_e \cup S_C|} \quad \text{where} \quad S_c = \bigcup_{e \in C} \{e\} \tag{3.3}$$

With S_e the entity set of a method defined in Equation 3.1. Third, T is filtered under the condition that m must modify at least one data structure in the target class. Fourth, they suggest to move m to the first target class in T that satisfies a set of preconditions related to compilation, behavior, and quality. This algorithm is implemented in the Eclipse plug-in *JDeodorant* (Fokaefs et al. (2007)).

Similarly to God Class, Palomba et al. (2013, 2015a) proposed to detect Feature Envy using historical information. First, the sequence of co-changed methods is extracted from version control systems using the *Change History Extractor*. Then, the detection rule for HIST rely on the conjecture that "*a method affected by feature envy changes more often with the envied class than with the class it is actually in*". Thus, Feature Envy methods are identified as those involved in commits with methods of another class of the system β % more than in commits with methods of their class. The value of β being set to 80% after parameter calibration.

Machine-learning Based Detection Approaches

The first attempt to detect Feature Envy using machine-learning techniques has been proposed by Fontana et al. (2016) during their large-scale study. In this context, the J48 decision tree algorithm outperformed other classifiers in detecting Feature Envy with an F-measure of 97%. Again, these results have been challenged by Di Nucci et al. (2018).

More recently, Liu et al. (2018) proposed a deep learning based approach to detect Feature Envy. Their approach relies on both structural and lexical information. On one side, the names of the method, the enclosing class (i.e., where the method is implemented) and the envied class are fed into convolutional layers. On the other side, the distance proposed by Tsantalis and Chatzigeorgiou (2009) is computed for both the enclosing class ($dist(m, ec)$) and the target class ($dist(m, tc)$), and values are fed into other convolutional layers. Then the output of both sides is fed into fully-connected layers to perform final decision. To train and evaluate their model, they use an approach similar to Moghadam and Cinneide (2012) where labeled samples are automatically generated from open-source applications by the injection of affected methods. These methods assumed to be correctly placed in the original systems

are extracted and moved into random classes to produce artificial Feature Envy instances (i.e., misplaced methods). Figure 3.6 overviews the proposed approach.

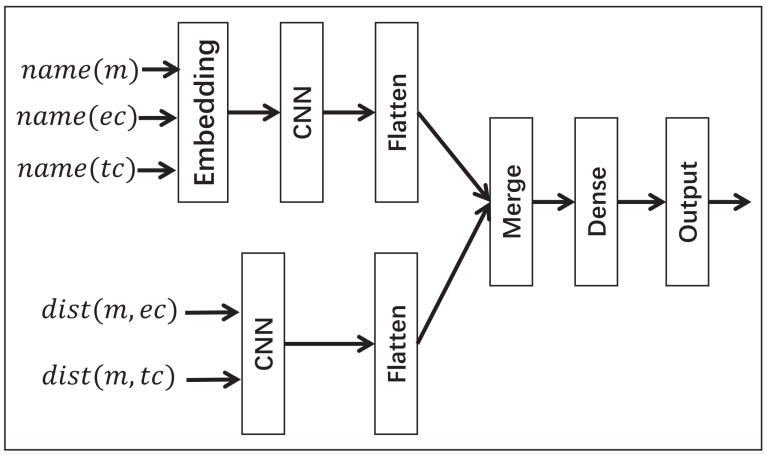

Figure 3.6 Liu et al. (2018) architecture for Feature Envy detection.

3.3 Empirical Studies on Anti-patterns

This section reports the empirical studies that have been conducted on anti-patterns. First we present studies aiming to understand the impact of anti-patterns on software quality. Second, we overview the studies conducted to understand how anti-patterns appear and evolve over time.

3.3.1 Impact of Anti-patterns on Software Quality

First, Deligiannis et al. (2004) performed a controlled experiment to understand the impact of God Class on design quality. Their results show that the presence of God Classes in a system negatively impacts the maintainability of the source code. Furthermore they concluded that it considerably impacts the way developers apply the inheritance mechanism. Yamashita and Moonen (2012) investigated the relation between specific anti-patterns and a variety of maintenance characteristics such as effort, change size and simplicity. They identify which anti-patterns can be used as indicator for maintainability assessments based on: (1) expert-based maintainability assessments of four Java systems and; (2) observations and interviews with professional developers who were asked to maintain these systems during a period of time. For the anti-patterns considered in this thesis, their results show that God Class affects the *Simplicity* and the *Use Of Components* and that Feature Envy affects the *Logic Spread*. Later Yamashita and Moonen (2013) performed a similar study aiming at understanding the

interactions between co-located anti-patterns and the impact of such interactions on software maintenance.

Abbes et al. (2011) conducted an empirical study with the aim of understanding the impact of two anti-patterns, namely God Class and Spaghetti Code on program comprehension. In this study, subjects were asked to perform basic tasks related to program comprehension on systems affected or not by the investigated anti-patterns. The results of their study show: (1) an increase in subjects' time and effort and a decrease of their percentage of correct answers in systems affected by God Class; (2) no significant correlation between program comprehension and the presence of Spaghetti Code and; (3) a strong difference between subjects' efforts, times, and percentages of correct answers on systems affected by both anti-patterns.

Khomh et al. (2009b, 2012) conducted a large scale empirical study investigating the relation between the presence of anti-patterns and the classes change- and fault-proneness. They investigate 13 anti-patterns in 54 releases of four software systems and analyze the changes and fault-fixing operations applied to the classes of these systems. Their results indicate clearly that classes participating in anti-patterns are more change- and fault-prone than classes not affected by any anti-pattern. Later, Palomba et al. (2018) confirmed the above findings by performing a similar experiment on a larger number of systems.

3.3.2 Evolution and Presence of Anti-patterns

First, Olbrich et al. (2009) studied the impact of anti-patterns on the change behavior of code components. Specifically, they analyzed the historical data over several years of development, of two large scale software systems and compared the change frequency and size of components affected by God Class and Shotgun Surgery with those of healthy components. With the results of their study, the authors confirmed that affected components exhibit different change behaviors. They also identified different phases in the life of software systems where the number of anti-patterns increase and decrease. Similarly, Vaucher et al. (2009) studied the "life cycle" of God Class occurrences in two open-source systems with the aim of understanding when they arise and how they evolve. With the results of their study, the authors were able to develop prevention mechanisms to predict whether changes applied to the system are likely to introduce new anti-patterns.

On the same line, Chatzigeorgiou and Manakos (2010) tracked the evolution of three anti-patterns: Long Method, Feature Envy and State Checking in the history of two open-source systems, showing that: (1) anti-patterns persist after being introduced; (2) most of the time, anti-patterns are introduced when the component they affect is added to the system and; (3) few occurrences are willingly removed through refactoring operations.

Tufano et al. (2015) performed the largest experiment on the presence of anti-patterns through the history of systems. Specifically, they mined the history of 200 software projects to understand when and why (i.e., under what circumstances) anti-patterns appear. First, their results confirm the observation made by Chatzigeorgiou and Manakos (2010) that most of instances are introduced when the file is added to the system. Second, they show that anti-patterns are also often introduced the last month before deadlines by experienced developers.

Finally, Palomba et al. (2018) assessed during their large scale study, the diffuseness, i.e., the percentage of affected code components of 13 anti-patterns in 30 open-source systems. They concluded that most of the anti-patterns are quite diffused, especially the ones characterized by their size or complexity. However they also identified few other anti-patterns such as Feature Envy that are less diffused.

CHAPTER 4 STUDY BACKGROUND

This chapter presents the background common to the studies detailed in the remainder of this thesis. First, we present and discuss the choice of the eight software systems considered in these studies. Second, we describe the oracle we created to conduct our experiments, which reports the occurrences of God Class and Feature Envy in the studied systems. Third, we overview the metrics used for evaluation. Finally, we discuss the considerations adopted to train machine-learning models on the task of anti-patterns detection.

4.1 Studied Systems

The context of our studies consists of eight open-source Java software systems belonging to various ecosystems. Two systems belong to the Android APIs[1]: Android Opt Telephony and Android Support. Four systems belong to the Apache Foundation[2]: Apache Ant, Apache Tomcat, Apache Lucene, and Apache Xerces. Finally, one free UML design software: ArgoUML[3] and one text editor: Jedit[4] available under GNU General Public License[5]. As further discussed in Section 4.2, this choice is motivated by the preliminary manual-detection of God Classes performed in prior studies on these systems (Moha et al. (2010); Palomba et al. (2013)). Without loss of generalizability, we chose to analyze only the directories that implement the core features of the systems and to ignore test directories. Table 4.1 reports for each system, the Git identification (SHA) of the considered snapshot, its "age" (i.e., number of commit) and its size (i.e., number of class).

Table 4.1 Characteristics of the Studied Systems

System name	Snapshot	Directory	#Commit	#Class
Android Opt Telephony	c241cad	src/java/	98	192
Android Support	38fc0cf	v4/	195	109
Apache Ant	e7734de	src/main/	6397	694
Apache Tomcat	398ca7ee	java/org/	3289	925
Apache Lucene	39f6dc1	src/java/	429	155
Apache Xerces	c986230	src/	3453	512
ArgoUML	6edc166	src_new/	5559	1230
Jedit	e343491	./	1181	423

[1]https://android.googlesource.com/
[2]https://www.apache.org/
[3]http://argouml.tigris.org/
[4]http://www.jedit.org/
[5]https://www.gnu.org/

4.2 Building a Reliable Oracle

To train and evaluate the performances of our models, we needed an oracle reporting the occurrences of the studied anti-patterns in the considered systems snapshots. We found no such large dataset in the literature. One existing crowd-sourcing dataset, Landfill created by Palomba et al. (2015b) included manually-produced anti-pattern instances but we found many erroneously-tagged instances, which discouraged and prevented its use in our work.

For God Class, we found two sets of manually-detected occurrences in open-source Java systems, respectively from DECOR (Moha et al. (2010)) and HIST (Palomba et al. (2013)) replication packages. Thus, we created our oracle from these occurrences under two constraints: (1) the full history of the system must be available and (2) the occurrences reported must be relevant. After filtering, over the 15 systems available in these replication packages, we retained eight to construct our oracle.

For Feature Envy, most of the approaches proposed in the literature are evaluated on artificial examples, i.e., assuming methods are correctly placed in the original systems, they are extracted and moved into random classes to produce Feature Envy occurrences (i.e., misplaced methods) (Moghadam and Cinneide (2012); Sales et al. (2013); Liu et al. (2018)). However, our approach relies on the history of code components. Therefore, such artificial anti-patterns are not usable because they have been willingly introduced in the considered systems' snapshot. Thus, we had to build manually our own oracle.

First, we formed a set of 779 candidate Feature Envy instances over the eight subject systems by merging the output of three detection tools (HIST, InCode, and JDeodorant), adjusting their detection thresholds to produce a number of candidate per system proportional to the systems sizes. Second, three different groups of people manually checked each candidate of this set: (1) the author of this thesis, (2) nine M.Sc. and Ph.D. students, and (3) two software engineers. We gave them access to the source code of the enclosing classes (where the methods were defined) and the potential envied classes. After analyzing each candidate, we asked respondents to report their confidence in the range [*strongly_approve*, *weakly_approve*, *weakly_disapprove*, *strongly_disapprove*]. To avoid any bias, none of the respondent was aware of the origin of each candidate. We made the final decision using a weighted vote over the reported answers. First we assigned the following weights to each confidence level:

strongly_approve	$\rightarrow$	1.00	*weakly_disapprove*	$\rightarrow$	0.33
weakly_approve	$\rightarrow$	0.66	*strongly_disapprove*	$\rightarrow$	0.00

Then, an instance is considered as a Feature Envy if the mean weight of the three answers reported for this instance is greater than 0.5.

Table 4.2 reports, for each system, the number of God Classes, the number of produced candidate Feature Envy instances, and the number of Feature Envy instances retained after manual-validation.

Table 4.2 Characteristics of the Oracle

System name	#God_Class	#Candidate_FE	#Feature_Envy
Android Opt Telephony	11	62	18
Android Support	4	21	2
Apache Ant	7	110	25
Apache Tomcat	5	173	57
Apache Lucene	4	42	4
ArgoUML	22	144	24
Jedit	5	98	22
Xerces	15	129	37
Total	**73**	**779**	**189**

4.3 Evaluation Metrics

To compare the performances achieved by different approaches on the studied systems, we consider each approach as a binary classifier able to perform a boolean prediction on each entity of the system. Thus, we evaluate their performances using the following confusion matrix:

Table 4.3 Confusion Matrix for Anti-patterns Detection

		predicted		***total***
		1	0	
true	1	A	B	n_{pos}
	0	C	D	n_{neg}
total		m_{pos}	m_{neg}	n

With (A) the number of true positives, (B) the number of misses, (C) the number of false alarms and (D) the number of true negatives. Then, based on this matrix, we compute the widely adopted *precision* and *recall* metrics:

$$precision = \frac{A}{A+C} \qquad (4.1) \qquad\qquad recall = \frac{A}{A+B} \qquad (4.2)$$

We also compute the F-measure (i.e., the harmonic mean of precision and recall) to obtain a single aggregated metric:

$$F\text{-}measure = 2 \times \frac{precision \times recall}{precision + recall} = 2 \times \frac{A}{n_{pos} + m_{pos}} \qquad (4.3)$$

4.4 Training

This section discusses the considerations adopted to train neural-networks on the task of anti-patterns detection. We consider training a multi-layer feed-forward neural-network to perform a boolean prediction on each entity of the training systems. First, the training set contains N training systems which can be expressed as:

$$\mathcal{D} = \{S_i\}_{i=1}^{N}, \quad with \quad S_i = \{(\mathbf{x}_{ij} y_{ij})\}_{j=1}^{n_i} \tag{4.4}$$

With S_i the i^{th} training system, $\mathbf{x}_{ij}$ the input vector corresponding to the j^{th} entity of this system, $y_{ij} \in \{0, 1\}$ the true label for this entity and n_i the size (i.e., number of entities) of S_i. Second, we refer to the output of the neural network corresponding to the positive label, i.e., the predicted probability that an entity is affected as: $P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij})$.

4.4.1 Custom Loss Function

Software systems are usually affected by a small proportion of anti-patterns ($< 1\%$) (Palomba et al. (2018)). As a consequence, the repartition of labels within a dataset composed of software system entities is highly imbalanced. Such imbalanced dataset compromises the performances of models optimized using conventional loss functions (He and Garcia (2008)). Indeed, the conventional *binary_cross_entropy* (cf. Equation 2.8) loss function maximizes the expected accuracy on a given dataset ,i.e., the proportion of instances correctly labeled. In the context of anti-patterns, the use of this loss function lead to useless models that assign the majority label to all input instances, thus maximizing the overall accuracy ($> 99\%$) during training. To overcome this issue, we must define a loss function that reflects our training objective (i.e., maximizing the F-measure achieved over the training systems).

Let us formulate our training objective as finding the set of parameters $\boldsymbol{\theta}^*$ that maximizes the mean F-measure achieved over the training system, which can be expressed as:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} F_m(\boldsymbol{\theta}, S_i) \tag{4.5}$$

Which is equivalent to the minimization of the empirical risk expressed in Equation 2.7 with a loss: $L = -F_m$. However, to solve this problem through gradient descent, we need our loss to be a continuous and differentiable function of the weights $\boldsymbol{\theta}$. As defined in Equation 4.3, the F-measure does not meet this criterion, which prevents its direct use to define our loss function. Indeed, computing the number of true positives (A) and positives (m_{pos}) requires

counting elements from the probability outputed by the model, which necessarily involves discontinuous operators like the Kronecker operator $[\![]\!]$ (cf. Equation 5.1):

$$A(\boldsymbol{\theta}, S_i) = \sum_{\substack{j=1 \\ y_{ij}=+1}}^{n_i} [\![P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) > 0.5]\!] \tag{4.6}$$

$$m_{pos}(\boldsymbol{\theta}, S_i) = \sum_{j=1}^{n_i} [\![P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) > 0.5]\!] \tag{4.7}$$

Consequently, we use the differentiable approximation of the F-measure provided by Jansche (2005), which simply consists in considering:

$$[\![P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) > 0.5]\!] \approx P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) \tag{4.8}$$

Thus, the approximated F-measure can be expressed as:

$$\tilde{F}_m(\boldsymbol{\theta}, S) = 2 \times \frac{\tilde{A}(\boldsymbol{\theta}, S)}{n_{pos} + \tilde{m}_{pos}(\boldsymbol{\theta}, S)} \tag{4.9}$$

Where:

$$\tilde{A}(\boldsymbol{\theta}, S_i) = \sum_{\substack{j=1 \\ y_{ij}=+1}}^{n_i} P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) \tag{4.10}$$

$$\tilde{m}_{pos}(\boldsymbol{\theta}, S_i) = \sum_{j=1}^{n_i} P_{\boldsymbol{\theta}}(1|\mathbf{x}_{ij}) \tag{4.11}$$

Finally, we define our loss function as follows:

$$L = -\tilde{F}_m(\boldsymbol{\theta}, S) \tag{4.12}$$

4.4.2 Optimization

When performing optimization through gradient descent, model parameters are usually updated according to the gradient of the loss computed on the whole training set or on equal-size subsets of the training set (mini-batch based SGD). To prevent our model to overfit on large systems (that contain more instances than others), we performed weights updates on variable size batches that contain instances of the same system. Hence, each training system have the same impact on optimization, regardless of their sizes. In fact, this approach is equivalent to

a stochastic gradient descent, considering each training system as a single instance. Fig. 4.1 overviews our approach in comparison to gradient descent and mini-batch SGD.

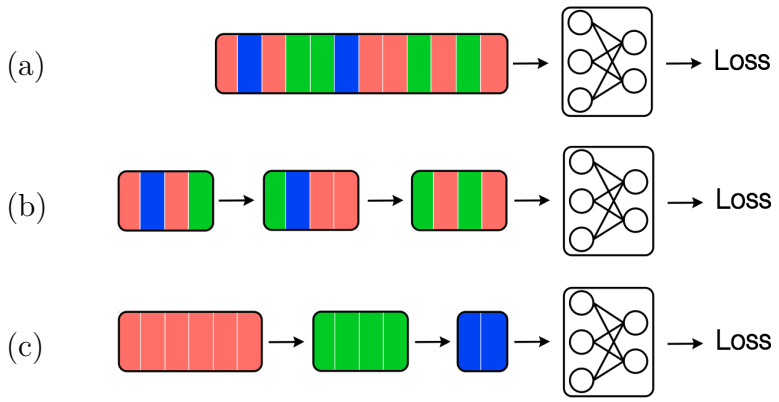

Figure 4.1 Comparison of feeding approaches: (a) gradient descent, (b) mini-batch SGD, (c) imbalanced-batch SGD. Colors represent instances belonging to same systems.

4.4.3 Regularization

Regularization is a way to prevent over-fitting. We used two widely-adopted regularization techniques: L_2 regularization and dropout.

L_2 Regularization

L_2 regularization consists in adding a term to the loss function to encourage the weights to be small (Witten et al. (2016)). This term is proportional to the sum of the Euclidean norm of the weight matrices, i.e., $\|\mathbf{W}\|_2 = \sqrt{\mathbf{W}^\top \mathbf{W}}$, also called L_2-norm. Thus, the L_2 regularization term added to the loss function can be expressed as:

$$L_2 = \lambda \sum_{l=1}^{L+1} \|\mathbf{W}_l\|_2 \tag{4.13}$$

With $\lambda \in \mathbb{R}$ an hyper-parameter adjusted during cross-validation.

Dropout

Dropout consists in dropping randomly out units, i.e., temporarily removing nodes of the network along with their connections during training. Thus, at each step, each node has

a probability $(1 - P_{keep})$ to be removed from the network. Dropout is equivalent to combining exponentially many architectures with shared parameters and has proved to prevent successfully over-fitting (Srivastava et al. (2014)).

4.4.4 Ensemble Learning

Ensemble learning is a common practice to improve the final performances of probabilistic models as well as to reduce the output variability from one training to another (Dietterich (2000)). The key idea is to train separately several randomly-initialized models and, thus, construct a set of classifiers. Then, when a new instance must be classified, the final prediction is computed from the output of each classifier. In the context of this study, we used the widely-adopted Bayesian averaging heuristic to compute the final prediction. Thus, after training M randomly-initialized models, the final predicted probability that an entity is affected given its corresponding feature vector $\mathbf{x}$ can be expressed as:

$$P_{ensemble}(1|\mathbf{x}) = \frac{\sum_{i=1}^{M} P_{\boldsymbol{\theta}_i}(1|\mathbf{x})}{M} \tag{4.14}$$

With $\boldsymbol{\theta}_i$, the set of weights of the i^{th} model.

CHAPTER 5 A MACHINE-LEARNING BASED ENSEMBLE METHOD FOR ANTI-PATTERNS DETECTION

5.1 Introduction

Fowler (1999) defined design smells as symptoms of poor solutions to recurring design problems. These symptoms, also called anti-patterns, are typically introduced in object-oriented systems when developers implement suboptimal design solutions due to lack of knowledge and–or time constraints. For example, the God Class anti-pattern refers to the situation in which a class grows rapidly by the addition of new functionalities, when developers break the principle of single responsibility. Prior empirical studies highlighted the negative impact of anti-patterns on a variety of quality characteristics, such as program comprehension (Abbes et al. (2011)), maintainability (Yamashita and Moonen (2013)), and correctness (increase of fault-proneness) (Khomh et al. (2012)). Thus, it is of major importance to identify their occurrences in software systems and apply refactoring operations to remove them.

Several approaches have been proposed to detect the occurrences of anti-patterns in systems. Most of these approaches attempt to identify bad motifs in models of source code using manually-defined heuristics that rely on some metrics (e.g., cyclomatic complexity). For example, Moha et al. (2010) proposed a domain-specific language to describe and generate detection algorithms for anti-patterns using structural and lexical metrics, while Palomba et al. (2013, 2015a) proposed a rule-based approach to detect anti-patterns from change history information.

Even though these approaches have shown acceptable performances, none of them can claim high accuracy on any systems and for any anti-patterns. Besides, each approach relies on its own definitions of anti-patterns and only focuses on specific aspects of systems. Thus, tools based on different detection strategies identify different sets of occurrences and have a low agreement (Fontana et al. (2012)), especially when these strategies rely on orthogonal sources of information (Palomba et al. (2013)).

Recently, machine-learning models have been shown efficient in a variety of domains, such as speech recognition (Graves et al. (2013)) or image processing (Krizhevsky et al. (2012)). Several machine-learning based approaches have been proposed to detect anti-patterns. However, these approaches failed to surpass clearly previous approaches. On the one hand, learning high-level features of systems requires complex machine-learning models, such as deep-neural-networks. On the other hand, these complex models require substantial amounts of

manually-produced training data, which is hardly available and time consuming to produce for anti-patterns.

Consequently, we propose SMAD (SMart Aggregation of Anti-pattern Detectors), a machine-learning based ensemble method to efficiently aggregate various anti-pattern detection tools. For each tool to be aggregated, we identify a set of core metrics, i.e., metrics that reflect the internal detection process. We then use the core metrics as input features of a simple neural-architecture. We identify three major advantages of our approach: (1) it combines detection tools more efficiently than conventional voting techniques; (2) it allows using a simple architecture by considering a low number of high-level features (i.e., core-metrics). Thus, the number of manually-produced examples needed to train our model is relatively low; and, (3) the occurrences detected by our approach could be used as training or pre-training instances for more complex models.

We implemented the proposed ensemble method to detect two well known anti-patterns: God Class and Feature Envy. To train and evaluate our model, we created an oracle containing instances of the studied anti-patterns in eight Java systems. We used instances from five of the eight systems to train the proposed model and the remaining instances for evaluation. We compared the performances achieved by SMAD with those obtained by (1) the tools aggregated through our approach and (2) the baseline voting ensemble technique. Thus, we can answer the following two research questions:

(RQ1) Does SMAD outperform state-of-the-art detection tools?
Our approach significantly improves the state-of-the-art. Compared to the tool that performed best, the average F-measure improves from 38% to 66% for God class and from 52% to 70% for Feature Envy.

(RQ2) Does SMAD outperform voting ensemble technique?
Our results indicate that our approach outperforms the voting technique by 35% for God class and 27% for Feature Envy in term of F-measure.

Finally, we assessed the possibility of using SMAD to produce training instances for deep-learning based anti-pattern detection models. We compared the performances achieved by the model proposed by Liu et al. (2018) trained on a dataset of artificial anti-patterns instances with those of the same model trained using instances identified by SMAD, thus answering the following research question:

(RQ3) To what extent can SMAD be used to label training instances for deep-learning anti-pattern detection models?

> Our results show that the model proposed by Liu et al. (2018) achieves better performances (+13%) when trained on instances identified by SMAD.

Thus, we make the following contributions: (1) a manually-produced oracle reporting the occurrences of God Class and Feature Envy in eight Java software systems; (2) a machine learning-based ensemble method to aggregate efficiently existing anti-pattern detection tools; and (3) a process for the automatic generation of training data for machine learning-based anti-pattern detection models.

The remainder of this chapter is organized as follows. Section 5.2 presents our approach SMAD. Sections 5.3 presents the study aiming at answering the first two research questions, while Section 5.4 presents the second study, thus answering the third research question. Finally, Section 5.5 concludes with future work.

5.2 SMart Aggregation of Anti-pattern Detectors

In this section, we present our machine-learning based ensemble method to aggregate efficiently various anti-pattern detection tools.

5.2.1 Baseline

Let us consider D detection tools $d_1, d_2, ..., d_D$ performing a boolean prediction over the entities of a software system based on some internal detection rule. We refer to as $d_i(e) \in \{True, False\}$ the boolean prediction of the i^{th} detection tool on an entity e.

We want to combine these tools to maximize the F-measure of the so-produced "merged" prediction over the entities of the studied system. The baseline approach consists in aggregating these tools using a voting policy over their predictions. We can define the function that outputs the "voted" prediction on a given entity e as:

$$V(e) = (\sum_{i=1}^{D} [\![d_i(e)]\!] \geq k) \quad with \quad [\![x]\!] = \begin{cases} 1 \text{ if } x\text{=True} \\ 0 \text{ if } x\text{=False} \end{cases} \tag{5.1}$$

where $k \in \{1, 2, ..., D\}$ is the policy, i.e., minimal number of positive agreements beyond which an entity receive the label True.

We identify two major limitations to this simple approach. First, every detection tool has the same weight on the final result while we argue that a tool with poor performances should have less weight in the vote than the one with better performances. Second, this approach

ignore the confidence of each tool in its prediction. Let us suppose that an entity is very close to be positively labeled by a tool, then we argue that the prediction of this tool should positively influence the voting on this entity more than predictions that are far from the tool detection threshold.

5.2.2 Overview

The key idea behind SMAD is to combine various detection tools by computing their core-metrics for each input instance and use these metrics to feed a machine-learning based classifier. First, for each anti-pattern considered in this study, we selected three state-of-the-art detection tools. These tools respectively rely on:

- *Rule Cards:* Affected entities are identified using a combination of source-code metrics designed to reflect the formal definition of the anti-patterns. For this category, we selected DECOR (Moha et al. (2010)) for God Class and InCode (Marinescu et al. (2010)) for Feature Envy detection.
- *Historical Information:* Affected entities are identified via an analysis of change history information derived from versioning systems. For this category, we used HIST (Palomba et al. (2013, 2015a)) for both God Class and Feature Envy detection.
- *Refactoring Opportunities:* Anti-patterns are detected by identifying the opportunities to apply their corresponding refactoring operations. For this category, we used the refactoring operations Extract Class (Fokaefs et al. (2012)) and Move Method (Tsantalis and Chatzigeorgiou (2009)) provided by JDeodorant, respectively for God Class and Feature Envy detection.

Selecting tools that are based on different strategies allows us to expect a low degree of agreement between them and thus, maximize the expected performances of our approach.

Then, we selected the core-metrics, i.e., metrics that reflect best the internal decision process of each tool, as input features for our model. Finally, we performed the classification through a logistic regression. Our model is a fully-connected neural-network (i.e., a MLP) composed of *tanh* hidden layers connected to a *softmax* output layer. Fig. 5.1 overviews our approach.

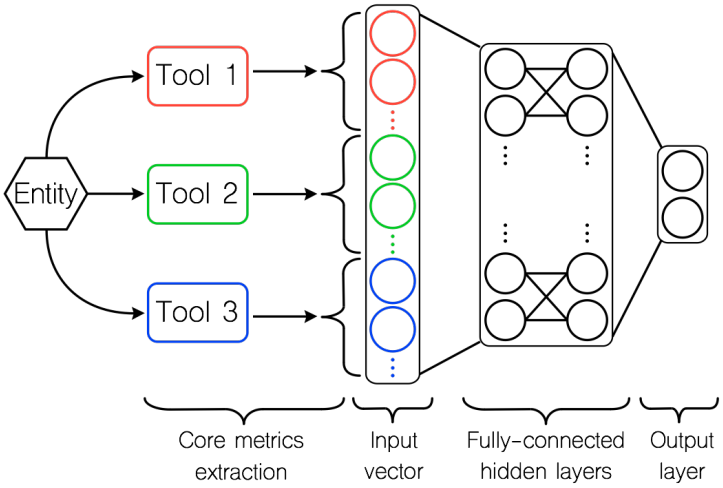

Figure 5.1 Overview of SMAD detection process

5.2.3 Input

Metrics for God Class Detection

For God Class detection, we extract **six** core-metrics from the three detection tools considered in this study. These metrics are computed for each class of a system.

DECOR: The internal detection rule relies on the definition of four code and design smells and can be expressed as: "(is associated to many *DataClass*) AND is a (*ControllerClass* OR *LargeClass* OR *LowCohesionClass*)". These smells are defined using structural and lexical metrics along with some thresholds: (1) *DataClass* relies on the number of accessors, (2) *ControllerClass* relies on lexical properties, (3) *LargeClass* relies on the sum of the NMD and NAD metrics (Number of Methods Declared + Number of Attributes Declared), and (4) *LowCohesionClass* relies on the LCOM metric (Lack of Cohesion in Methods) (Briand et al. (1998)). Thus, we extracted four core metrics from DECOR internal detection rule for God Class:

- Number of associated *DataClass*
- $[\![$*ControllerClass*$]\!]$
- nmd_nad
- lcom

with nmd_nad and lcom being the uppercase metrics divided by their respective threshold.

HIST: God Classes are identified as: "classes modified (in any way) in more than α% of commits involving at least another class". Thus, we extracted one core-metric from HIST internal detection rule for God Class:

- Number of commits in which the considered class has been modified along with other classes.

JDeodorant: God Classes are classes from which a *concept* can be extracted, defined as: "a distinct entity or abstraction for which a single class provides a description and/or a set of attributes and methods that contribute together to the same task". We define our core-metric for JDeodorant as:

- Number of *concepts* that can be extracted from the considered class.

Metrics for Feature Envy Detection

A Feature Envy is characterized by two source code entities: a method (i.e., the envious method) and a class (i.e., the envied class). Thus, in a given system, the number of potential instances that must be investigated is equal to $n_m \times (n_c - 1)$ with n_c and n_m respectively the numbers of classes and methods in the system. To reduce this number, we filter the studied system at both class and method level, similarly to Tsantalis and Chatzigeorgiou (2009). First, we consider as potential envious methods only non-static and non-accessor methods. Then, for each of the remaining methods, we consider as potential envied classes only classes that are accessed in some way in the body of the method. We extract **seven** core-metrics from the three considered detection tools.

HIST: Feature Envy methods are identified as: "methods involved in commits with methods of another class of the system β% more than in commits with methods of their class". Thus, we extract one core-metric from HIST internal detection rule for Feature Envy:

- Ratio between the number of commits involving methods of the envied class and number of commits involving other methods of the enclosing class.

InCode: Methods are identified as being envious without information about the envied class. In this context, a method is declared affected if: (1) "it uses directly more than a few attributes of other classes" (ATFD > FEW), (2) "it uses far more attributes from other classes than its own" (LAA < ONE THIRD), and (3) "the used "foreign" attributes belong

to very few other classes" (FDP $\leq$ FEW). We redefined the first two metrics to express information about the envied class, which led us to three core-metrics:

- ATFD (Access To Foreign Data), i.e., number of attributes of the envied class accessed by the method.
- LAA (Locality of Attribute Accesses), i.e., ratio between the number of accesses to attributes that belongs to the envied class vs. the enclosing class.
- FDP (Foreign Data Providers), i.e., number of distinct foreign classes whose attributes are accessed by the method.

JDeodorant: For each method m in the system, a set of candidate target classes T is created and sorted based on: (1) the number of entities (methods or attributes) that m accesses from each class of T and (2) the Jaccard distance between m and each target class. Then, JDeodorant suggests to move m to the first target class that satisfies a set of preconditions related to compilation, behavior, and quality. We extracted three core-metrics, which can be expressed as:

- Ratio between the number of access to entities that belong to the envied class vs. enclosing class.
- Ratio between the Jaccard distances from the method to the envied class vs. enclosing class.
- Boolean value indicating whether the Move Method Refactoring operation has been proposed by JDeodorant or not.

System Metrics

For both God Class and Feature Envy detection, two additional system metrics are added to the input vector of our model:

- System size (i.e., number of classes)
- History length (i.e., number of commits)

Adding metrics related to the considered snapshot, positively leverage our model interpretation of the previous metrics. We confirmed this intuition when performing our experiments.

5.3 Evaluation of the Detection Perfomances

In this section, we address the evaluation of SMAD performances in detecting the two anti-patterns considered in this study. We answer the two following research questions:

- **RQ1:** *Does SMAD outperform state-of-the-art detection tools?*
- **RQ2:** *Does SMAD outperform voting ensemble technique?*

5.3.1 Study Design

The goal of this study is to evaluate SMAD on both God Class and Feature Envy and to compare SMAD to state-of-the-art detection tools as well as competitive ensemble techniques. The context of this study consists of the eight Java systems presented in Table 4.1. To answer both research questions, we selected three systems, i.e., Android Support, Apache Tomcat, and Jedit, to perform an evaluation. We selected these systems to increase the generalizability of our results. Indeed, they belong to different domains: telephony framework, service container, and text editor and their sizes and history lengths cover well the ranges of possible values as shown in Table 4.1. We used the remaining five systems to train our model and calibrate hyper-parameters.

To run the competitive tools on the evaluation systems (**RQ1**), we used their publicly-available implementations and replicated the approaches for which no implementation was available. Thus, we ran DECOR using the Ptidej API[1] and JDeodorant using its Eclipse plug-in[2]. We implemented the detection rules for HIST as described). InCode Eclipse plug-in is no longer available and we reimplemented its detection rule as described in its original paper (Lanza and Marinescu (2007)) to retrieve also information about the envied class, as explained in Section 5.2.3.

To assess the performances of competitive ensemble methods (**RQ2**), we implemented the voting technique described in Section 5.2.1 for the three possible values of k, i.e., policies as shown in Equation 5.1.

[1]https://github.com/ptidejteam/v5.2/
[2]https://marketplace.eclipse.org/content/jdeodorant/

5.3.2 Parameters Calibration

SMAD

To calibrate the hyper-parameters of our model, we performed a random search over 300 generations of five hyper-parameters: learning rate (η), λ, P_{keep}, number of hidden layers, and number of neurons per hidden layer. This technique has shown to be more efficient than grid search on similar multi-dimensional optimization problems (Bergstra and Bengio (2012)). We evaluated the performances achieved on each hyper-parameters' combination by carrying out a 5-fold cross-validation, i.e., leave-one-out, over the five systems contained in our training set: Android Opt Telephony, Apache Ant, Apache Lucene, ArgoUML, and Xerces. At each iteration, we trained five times our model on 100 epochs by leaving one system out to perform the evaluation while keeping the others for training. Table 5.1 reports, for each hyper-parameter, the range of values experimented as well as the value with the best result for both God Class and Feature Envy.

Finally, the models used for experiments have been trained on 400 epochs with an exponential learning rate decay of 0.7 every 100 epochs. As explained in Section 4.4.4 we computed the final prediction from the output of five randomly-initialized models.

Table 5.1 Hyper-parameters Calibration of SMAD

Hyper-parameter	Range	Best (GC)	Best (FE)
Learning Rate (η)	$10^{-[0.0;2.5]}$	8.26×10^{-2}	1.90×10^{-1}
L2-norm (λ)	$10^{-[0.0;2.5]}$	3.13×10^{-2}	1.97×10^{-1}
Dropout (P_{keep})	$[0.5; 1.0]$	0.5	1.0
Number of Hidden Layers	$[1; 3]$	2	2
Neurons per Layer	$[4; 140]$ then $[4; n]$	[34, 30]	[86, 44]

With n the size of the previous hidden layer.

Detection Tools

Although we followed rigorously the guidelines given by the authors of HIST and InCode when reimplementing these tools, some differences may remain between our respective implementations. Such differences could affect the optimal values of their parameters. Thus, we performed an additional parameter tuning for these tools by computing the mean F-measure achieved over the eight systems considered in this study. We retained values that led to the best performances in our experiments. Table 5.2 reports, for each investigated parameter, the range of values experimented and the value retained to conduct our experiments.

Table 5.2 Hyper-parameters Calibration of the Competitive Tools

Tool	Hyper-parameter(s)	Range	Best Value
HIST (FE)	α	From 0% to 300% by 10%	160%
HIST (GC)	β	From 0% to 20% by 0.5%	8%
InCode	(ATFD, LAA, FDP)	$[0; 7]^3$	(2, 3, 3)

5.3.3 Analysis of the Results

Table 5.3 reports the results of our experiments for God Class while Table 5.4 reports our results for Feature Envy. Our results report the performances on the three subject systems, in terms of precision, recall, and F-measure, achieved by: (1) the three detection tools used for aggregation; (2) the voting techniques; and, (3) SMAD. In addition, we report the mean values of the three performance metrics for each investigated approaches.

Table 5.3 Performances Evaluation of SMAD for God Class detection

Approaches	Apache Tomcat			JEdit			Android Platform Support			**Mean**		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
DECOR	67%	40%	50%	17%	60%	26%	0%	0%	0%	28%	33%	25%
HIST	0%	0%	0%	22%	40%	29%	100%	75%	86%	41%	38%	38%
JDeodorant	2%	60%	4%	5%	60%	9%	17%	25%	20%	8%	48%	11%
Vote (k=1)	3%	80%	6%	6%	80%	10%	38%	75%	50%	15%	**78%**	22%
Vote (k=2)	100%	20%	33%	13%	40%	20%	100%	25%	40%	71%	28%	31%
Vote (k=3)	0%	0%	0%	67%	40%	50%	0%	0%	0%	22%	13%	16%
SMAD	43%	60%	50%	80%	80%	80%	100%	50%	67%	**74%**	63%	**66%**

Table 5.4 Performances Evaluation of SMAD for Feature Envy detection

Approaches	Apache Tomcat			JEdit			Android Platform Support			**Mean**		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
HIST	9%	9%	9%	2%	5%	3%	0%	0%	0%	4%	4%	4%
InCode	52%	56%	54%	46%	59%	52%	50%	50%	50%	49%	55%	52%
JDeodorant	31%	42%	36%	44%	50%	47%	100%	50%	67%	59%	47%	50%
Vote (k=1)	30%	100%	46%	24%	100%	39%	29%	100%	44%	28%	**100%**	43%
Vote (k=2)	80%	7%	13%	75%	14%	23%	0%	0%	0%	52%	7%	12%
Vote (k=3)	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
SMAD	52%	51%	51%	69%	50%	58%	100%	100%	100%	**74%**	67%	**70%**

Does SMAD outperform state-of-the-art detection tools?

For God Class detection, SMAD shows a precision of 74% and a recall of 63% (F-measure of 66%) on average over the subject systems. Thus, the proposed ensemble method clearly

outperforms state-of-the-art detection tools. Specifically, the mean F-measure improves by 74% in comparison to the tool that performed the best (HIST with 38%). Considering the performances achieved on each system, SMAD shows an F-measure ranging between 50% and 80%, which confirms that SMAD performs well independently of the systems characteristics. On the contrary, each competitive tool shows poor performances on at least one system. However, the low performances achieved by JDeodorant (especially precision) can be due to this tool relying on a different definition of God Class than others. Indeed, affected entities are detected only if opportunities to split them are identified.

For Feature Envy detection, SMAD achieves on average a precision of 74% and a recall of 67% leading to an F-measure of 70%. We observe better performances in terms of F-measure achieved by the static code analysis tools (52% by InCode and 50% by JDeodorant) than for God Class detection and low results for HIST. These results show that SMAD outperforms state-of-the-art tools when detecting Feature Envy with a mean F-measure 35% higher than that of the tool that performed the best (InCode with 52%). However, when replicating HIST rules for Feature Envy detection, we used a different component[3] to extract changes at method level than that of the original approach because the original component is supposedly unavailable because of its license. We are aware that such difference could affect the reported performances.

SMAD significantly outperforms state-of-the-art detection tools on detecting God Class and Feature Envy. Furthermore, our results indicate that SMAD performs well independently of the systems characteristics.

Does SMAD outperform voting ensemble technique?

We report the results of our study for both God Class and Feature Envy, considering precision, recall, and F-measure independently in turn. In term of precision, our results indicate that SMAD outperforms the best voting policy ($k = 2$) with an average value of 74% to be compared to 71% and 52% for the voting technique, respectively on God Class and Feature Envy. In term of recall, unsurprisingly, the union voting policy ($k = 1$), i.e., union of the detected entities, achieves the highest performances (78% for God Class and 100% for Feature Envy). Finally, our results show that voting techniques are not a suitable ensemble method for anti-pattern detection in term of F-measure. Indeed, none of the policies seem to increase the mean F-measure of the aggregated tools. Furthermore, the policy (i.e., the value of k)

[3]http://www.incava.org/projects/diffj

that leads to the highest performances is not the same for both anti-patterns.

> *SMAD outperforms voting techniques in terms of precision and F-measure. Unsurprisingly, we observed that the union voting policy (k=1) lead to the highest recall. However, our results indicate that none of the voting policies are suitable to increase F-measure.*

5.4 Evaluation of the Ability to Label Training Instances

In this section, we evaluate SMAD on labeling entities to train complex machine-learning models. Thus, we answer the following research question:

- **RQ3:** *To what extent can SMAD be used to label training instances for deep-learning anti-pattern detection models?*

5.4.1 Study Design

This study evaluates the ability of SMAD to label training data for deep-learning anti-pattern detection models. We found no such architecture for the detection of God Class in the literature. Thus, we experiment this process only for the detection of Feature Envy on the Convolutional Neural Network (CNN) proposed by Liu et al. (2018). This study compares the performances achieved by: (1) the studied model trained on "*injected smells*", i.e., assuming methods are correctly placed in the original systems, they are moved into random classes to produce artificial Feature Envy occurrences and (2) the studied model trained on instances labeled by SMAD.

To produce labeled instances, we considered using the same systems selected in the original study. However, for some of these systems, historical information is not available through version-control systems, which prevents their use in our study. Thus, we selected eleven Java systems of different sizes and domains from the `Qualitas Corpus` (Tempero et al. (2010)). Table 5.5 overviews the characteristics of these systems. Then, we used the architecture of SMAD trained and evaluated in Section 5.3 to label the instances of these systems.

To compare both approaches, we use the implementation made available by the authors to run the original model and we implemented another version that allow the use of the custom loss function defined in Equation 4.12 for optimization as well as regularization to address the unbalanced labels produced by SMAD. We assess the performances of the two models on the same three systems used in our previous experiments.

Table 5.5 Characteristics of the Systems used to Generate Training Instances

System name	Snapshot	Directory	#Commit	#Class
Apache Derby	c30c7da	java/engine/	1338	1022
Apache Jena	dc0bfe6	jena-core/src/main/	403	686
Apache Jspwiki	a3b1041	src/	3993	330
Apache Log4j	7cf64b6	src/java/	734	313
Apache Velocity	23c979d	src/	1241	164
Javacc	1b23b61	src/	315	155
Jgraphx	25c9cfc	src/	117	177
Jgroups	2d2ee7d	src/	3138	276
Jhotdraw	58d8df3	jhotdraw7/src/main/	503	549
Mongodb	b67c0c4	src/main/	909	111
Pmd	6063aaf	pmd/src/main/	4656	815

5.4.2 Parameters Calibration

We calibrate the hyper-parameters of the subject model using a random search over 100 generations of: η and λ. We evaluate the performances achieved from each hyper-parameters combination by computing the mean F-measure achieved over the five systems used to train SMAD in Section 5.3. Thereby, we calibrate the model on manually-validated occurrences without using testing data. Table 5.6 reports for each hyper-parameter, the range explored and the value which led to the highest result.

Table 5.6 Hyper-parameters Calibration of the Subject Model

Hyper-parameter	Range	Best Value
Learning Rate (η)	$10^{-[0.0;4.0]}$	1.62×10^{-1}
L2-norm (λ)	$10^{-[0.0;4.0]}$	1.80×10^{-3}

When performing preliminary experiments, we observed that the model had difficulties to learn from both parts of its input (i.e., lexical and structural) together, thus performing better when trained using only lexical or structural informations. Consequently, we pretrain the model during 40 epochs using only the structural part of the input (i.e., the distances) before training it on 40 other epochs with the full input.

5.4.3 Analysis of the Results

Table 5.7 reports the performances achieved by the two CNNs proposed by Liu et al. (2018) on the three systems. The first version (referred as LIU_INJ) has been trained on "*injected smells*" while the second version (referred as LIU_GEN) has been trained on instances labeled by SMAD. We also report the mean values of the performance metrics on the systems.

Table 5.7 Performances of the Subject Model Trained on Injected vs Generated Smells

Models	Apache Tomcat			JEdit			Android Platform Support			**Mean**		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
LIU_INJ	2%	94%	3%	1%	88%	2%	1%	100%	2%	1%	**94%**	3%
LIU_GEN	17%	4%	7%	50%	35%	41%	0%	0%	0%	**22%**	13%	**16%**

To what extent can SMAD be used to label training instances for deep-learning anti-pattern detection models?

Our results show that the deep-learning model achieves a precision of 22% and a recall of 13% (F-measure of 16%) on average when trained on instances labeled by SMAD. These results confirm that instances labeled by SMAD are more suitable to train this model than *injected smells* for Feature Envy. Indeed, when trained on *injected smells*, the model achieves the highest recall but at the expense of its precision, which leads to a low F-measure of 3%.

Some factors could explain the difference between our results and those reported in the literature (Liu et al. (2018)). First, we evaluate a prediction as correct if and only if both the method AND the envied class are correct, while they evaluate their model on two tasks: (1) predicting if a method is associated with Feature Envy (without assessing the correctness of the proposed envied class); (2) recommending a destination (i.e., the envied class) only on the methods correctly detected in the previous step. Second, we use a different implementation to compute the distances between methods and classes.

The CNN proposed by Liu et al. (2018) achieves better performances when trained on instances labeled by SMAD than on "injected smells". This result confirms that SMAD generates reliable instances for deep-learning anti-pattern detection models.

5.5 Conclusion and Future Work

We proposed SMAD, a machine-learning based ensemble method to aggregate efficiently various anti-pattern detection tools based on their internal decision processes. To train and evaluate our approach, we built an oracle containing the occurrences of God Class and Feature Envy in eight open-source systems. Then, we evaluated SMAD on: (1) detecting occurrences of God Class and Feature Envy and (2) its ability to label training instances for deep-learning anti-pattern detection models. Key results of our experiments indicate that:

- SMAD significantly outperforms state-of-the-art detection tools on detecting God Class and Feature Envy and performs well independently of the systems characteristics.
- SMAD outperforms voting techniques in terms of precision and F-measure. Although the union voting policy ($k = 1$) leads to the highest recall, none of the voting policies increases F-measure.
- The CNN proposed by Liu et al. (2018) achieves better performances when trained on instances labeled by SMAD than on "injected smells", which confirms that SMAD generates reliable instances for deep-learning anti-pattern detection models.

Future work includes a comparative study of the different machine-learning algorithms that could be used for aggregation. We also plan to extend our approach to the detection of other anti-patterns with a greater number of detection tools. Finally, we want to leverage deep learning techniques to "learn" core-metrics from raw data.

CHAPTER 6 DEEP-LEARNING ANTI-PATTERNS DETECTION FROM CODE METRICS HISTORY

6.1 Introduction

Anti-patterns have originally been defined by Fowler (1999) as symptoms of poor design choices. These anti-patterns are typically introduced in the source code of software systems when developers implement sub-optimal solutions to their daily tasks. Several empirical studies have highlighted the negative impact of anti-patterns on code quality and maintenance properties. For example, the Feature Envy anti-pattern, which happens when a method have been implemented in the wrong class, have been shown to violate the principles of high cohesion and low coupling (Palomba et al. (2014)).

A variety of approaches have been proposed to detect the occurrences of anti-patterns in source code (Moha et al. (2010); Tsantalis and Chatzigeorgiou (2009); Marinescu et al. (2010)). Most of them rely on the formal definition of anti-patterns and attempt to identify their occurrences in source code using structural metrics (e.g., Lines Of Code) along with empirically defined thresholds. However, anti-patterns can also be detected by an analysis of change history information (Palomba et al. (2013)). Indeed, the presence of anti-patterns in a system influence how source code entities evolve with one another over time. For example, Feature Envy can be detected by identifying methods that change more often with methods of another class than that of their own class.

Although structural and historical anti-patterns detection have shown acceptable performances, these approaches identify different sets of anti-patterns in a system and suffer from intrinsic limitations. On the one hand structural detection techniques rely only on one single version of software systems. On the other hand, the historical detection technique does not consider the structural properties of the changed entities.

Consequently, we propose CAME (Convolutional Analysis of code Metrics Evolution), a deep-learning based approach to detect anti-patterns by analyzing how source code metrics evolve over time. To do so, we retrieve code metrics values for each revision of the system under investigation by mining its version control system (e.g., Git, SVN). This information is then provided as input to a convolutional neural network to detect affected components. We implemented the proposed approach for the detection of God Class.

To the best of our knowledge, we are first to rely on code metrics evolution for anti-patterns detection. Furthermore, we are the first to apply deep-learning techniques to the detection of God Class.

We experiment the proposed approach on the manually-defined oracle presented in Section 4.2 reporting occurrences of the studied anti-pattern in eight Java software systems. First, we split this oracle into training (five systems) and test (three systems) sets. To increase the number of training examples, we use the ensemble method presented in Chapter 5 (SMAD) calibrated on the training set to label instances of eleven other systems selected in the `Qualitas Corpus` dataset (Tempero et al. (2010)). Finally, we compare the performances achieved by CAME on the test set with those obtained by: (1) an equivalent model ignoring source code metrics history and; (2) three state-of-the-art detection tools that rely on structural and historical information. With the results of our study, we aim to answer the following two research questions:

(RQ1) To what extent historical information about code metrics can improve detection performances?
Source code metrics evolution significantly improves the model for detecting occurrences of anti-patterns in software systems. With respect to a similar model which doesn't take into account metrics history in its prediction, CAME improves the average F-measure from 63% to 72% for God class.

(RQ2) Does CAME outperform state-of-the-art detection tools?
Our approach significantly improves the state-of-the-art. With respect to the tool that performed best, the average F-measure improves from 38% to 72% for God class.

Thus, we make the following contributions: (1) a deep-learning based model for anti-patterns detection; (2) a comparative study to understand the impact of code metrics evolution on anti-patterns detection; and (3) a comparison of our approach with state-of-the-art detection tools.

The remainder of this chapter is organized as follows. Section 6.2 presents our approach CAME. Sections 6.3 presents our study aiming at answering the two research questions. Finally, Section 6.4 concludes with future work.

6.2 CAME: Convolutional Analysis of Metrics Evolution

In this section, we present CAME (Convolutional Analysis of Metrics Evolution), our approach leverage code metrics history to detect anti-patterns. We first overview the main steps of our process, then we present the input of our model for the detection of God Class. Finally, we present in detail the CNN architecture of the model we use to perform classification.

6.2.1 Overview

Figure 6.1 overviews the main steps employed by CAME to detect affected entities (i.e., classes or methods) in a given software system.

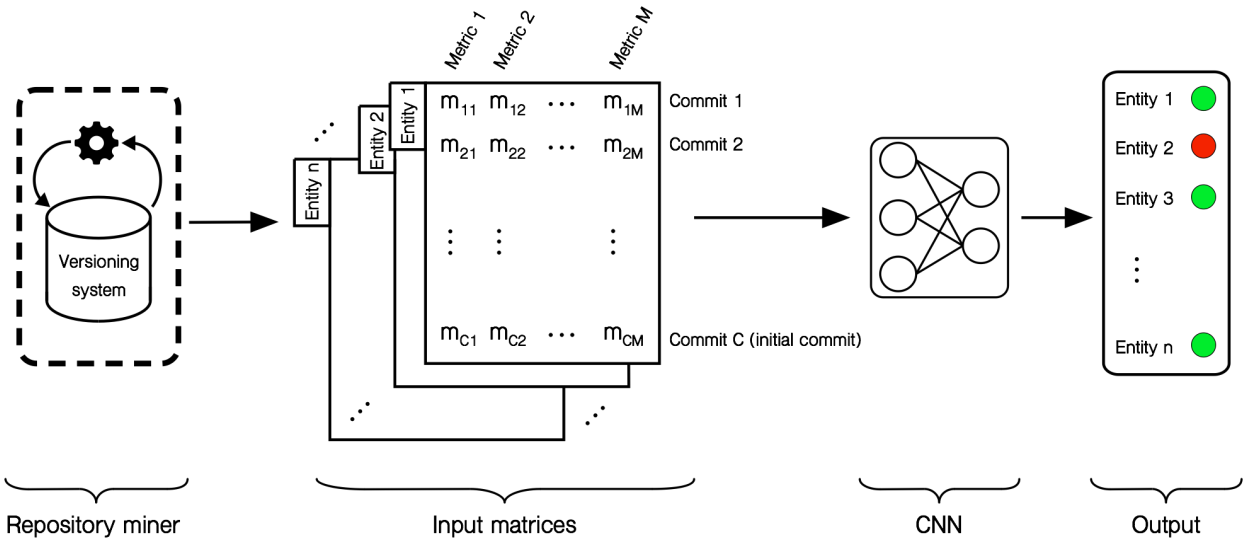

Figure 6.1 Workflow of CAME

Data Extraction

We designed a component called *repository miner* to automatically extract the necessary data by mining the version control system (e.g., Git, SVN) of software under investigation. The *repository miner* takes as input three arguments: (1) the URL of the system's repository; (2) the SHA ,i.e., the identification number, of the system's snapshot (i.e., version) we want to analyze and; (3) the sub-directories of interest ,i.e., those in which we want to detect affected components. Then, the *repository miner* downloads the repository and starts mining the system's history.

In the first step, the *repository miner* extract the name of all the entities implemented in the sub-directories of interest for the considered snapshot. Then, for each commit preceding the current snapshot, a set of object oriented and code metrics are computed for each entity. Note that we only consider commits for which at least one metric has been modified for at least one entity. Also, to reduce execution time, the *repository miner* keeps an internal representation of the system's entities which is updated at each new commit only for the the files that have been changed. By doing so, we can recalculate the new values of the metrics at each change applied to the system without reanalyzing all the Abstract Syntax Trees (ASTs). This process is repeated until we reach the first (i.e., initial) commit of the repository.

Finally, the output of the *repository miner* consists in several *.csv* files containing the names and metrics values of each entity for a given commit.

Classification

From the output of the *repository miner*, we compute the input matrix of our model for each entity. This matrix contains the values of the code metrics computed at each commit. Our model is a CNN which performs a binary classification over the entities of the system from the input matrix.

6.2.2 Input

Fog God Class detection, the entities to classify correspond to the classes of the systems. To decide whether or not a given class is affected by the God Class anti-pattern, we retrieve the history of **six** metrics.

First, a God Class is mainly characterized by its size. Hence, we compute for each class the widely known LOC metric which represents the number of lines of code which compose a class by ignoring blank lines and comments. Then, as proposed by Moha et al. (2010) we compute four other metrics used by DECOR: (1) NMD (Number of Methods Declared); (2) NAD (Number of Attributes Declared); (3) the LCOM5 metric (Lack of Cohesion in Methods) proposed by Briand et al. (1998) and; (4) the number of associated data classes. We consider as a data class, a class for which the ratio between the number of attributes declared vs. number of non accessor methods declared is greater than 8. Finally, we also compute the ATFD metric for a class proposed by Lanza and Marinescu (2007).

Comparatively to SMAD, we also compute two metrics related to the system's snapshot: (1) number of classes and; (2) history length. However, we do not retrieve the history of these metrics which are fed through the network along with the output of the convolutional layers.

Finally, to allow our model to receive a fixed size input matrix, we limit the length of the metrics history to 1000 commits. If the output of the *repository miner* contains more files, we process only the first 1000. However, if the number of outputted files is lower than 1000, we fill the rest of the input matrix with zeros.

6.2.3 Architecture

Figure 6.2 overviews the architecture of the convolutional neural network used by our approach to perform classification.

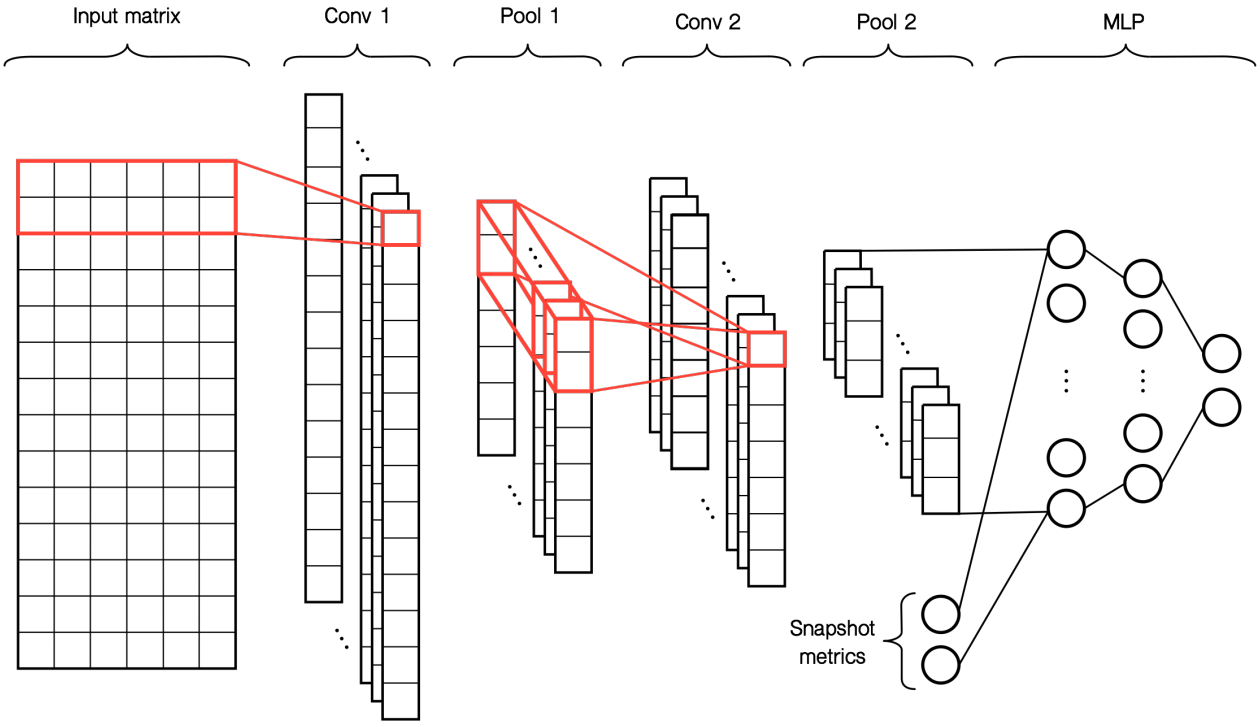

Figure 6.2 Architecture of CAME's CNN

The model contains two convolution + pooling layers fully-connected to several dense layers. The first convolution layer performs a 1D convolution with a filter size of 2. This characteristic allows the model to compare the values of metrics between two commits and infer that "something" has been changed. The other hyper-parameters , e.g., number of filters in conv 1, filter size in conv 2, etc., are adjusted at cross-validation time. As previously evoked, the two additional snapshot related metrics are fed through the network along with the output of the convolution layers.

6.3 Experiments

In this section, we evaluate CAME on the task of God Class detection. To avoid redundancy, we report the results for both research questions together:

- **RQ1:** *To what extent historical information about code metrics can improve detection performances?*
- **RQ2:** *Does CAME outperform state-of-the-art detection tools?*

6.3.1 Study Design

The goal of this study is to evaluate the performances of CAME in detecting the God Class anti-pattern and to compare CAME to state-of-the-art detection tools as well as to assess the importance of historical information in it's prediction. To train and evaluate the different approaches, we used our manually-built oracle which reports the occurrences of God Class in eight open-source Java projects (cf. Table 4.2). Our test set is composed of three systems: Android Support Apache Tomcat and Jedit while we keep the remaining five systems for training. To increase the size of the training set, we used the ensemble method SMAD presented in Chapter 5 to label instances of eleven other systems reported in table 5.5.

To assess the impact of code metrics history in CAME's performances (**RQ1**), we compare our approach with a similar machine-learning model that takes as input only one version of the considered metrics. As a competitive approach, we choose a MLP model which takes as input the same metrics than CAME (i.e., 6 structural metrics + 2 system metrics) computed for the considered system's snapshot. Indeed, what defers between CAME and a MLP model is the use of convolutional layers to process metrics history as a multi-dimensional array. To avoid any bias, we trained both models on the same systems and used the exact same method for hyper-parameters calibration.

To answer **RQ2**, we ran the competitive tools (DECOR, HIST, JDeodorant) on the three evaluation systems to compare their performances with those of CAME. As for previous experiments, we used the implementations made publicly available by the authors and replicated the approaches for which no public implementation was available.

6.3.2 Hyper-parameters Calibration

CAME

The set of optimal hyper-parameters for CAME have been found using a 4-fold cross validation over the eight hyper-parameters reported in Table 6.1. To do so, we split the training set into four folds of four systems. Hence, each set of hyper-parameters is tested four times by leaving one fold out for testing and keeping the remaining twelve systems for training. To avoid any bias, we made sure that each fold contains systems of various size, history length and domain. At each step, the new set of hyper-parameters is generated using a random search. Table 6.1 reports for each hyper-parameter, the ranges of values investigated and the value that led to the best result.

Table 6.1 Hyper-parameters Calibration of CAME

Hyper-parameter	Range	Best Value
Learning Rate (η)	$10^{-[0.5;3.0]}$	1.17×10^{-1}
L2-norm (λ)	$10^{-[0.5;3.0]}$	2.13×10^{-2}
Nb Filters Conv_1	$[10; 40]$	40
Size Pooling Conv_1	$\{5, 10, 20\}$	20
Nb Filters Conv_2	$[10; 20]$	20
Size Filters Conv_2	$[2; 5]$	3
Size Pooling Conv_2	$\{5, 10, 20\}$	20
Nb Dense Layers	$[1; 3]$	2
Size Dense Layers	$[4; 140]$ then $[4; n]$	[11, 8]

With n the size of the previous dense layer.

For evaluation, we trained CAME during 200 epochs by applying a learning rate decay of 0.7 every 50 epochs. Then, the final performances are computed using an ensemble learning technique from five randomly initialized models.

MLP

For the concurrent MLP model, the hyper-parameters tuning has been performed similarly to CAME. We used the same 4-fold cross validation to monitor the performances of the MLP using the hyper-parameters reported in Table 6.2. Finally, we trained the model during 400 epochs by applying a learning rate decay of 0.7 every 100 epochs and compute the final performances from an ensemble of five models.

Table 6.2 Hyper-parameters Calibration of the Concurrent MLP

Hyper-parameter	Range	Best Value
Learning Rate (η)	$10^{-[0.0;2.5]}$	2.44×10^{-2}
L2-norm (λ)	$10^{-[0.0;2.5]}$	6.38×10^{-2}
Dropout (P_{keep})	$\{0.5; 1.0\}$	1.0
Nb Dense Layers	$[1; 3]$	2
Size Dense Layers	$[4; 140]$ then $[4; n]$	[24, 22]

With n the size of the previous dense layer.

6.3.3 Analysis of the Results

Table 6.3 reports the detection performances for God Class achieved by CAME on the three test systems along with those of: (1) three state-of-the-art detection tools (i.e., DECOR, HIST and JDeodorant) and; (2) a MLP model that takes as input the same structural metrics than CAME. We also report an aggregate of the performances achieved on each system by computing the mean value.

Table 6.3 Performances Evaluation of CAME for God Class detection

Approaches	Apache Tomcat			JEdit			Android Platform Support			**Mean**		
	Precision	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*	*Precision*	*Recall*	*F-measure*
DECOR	67%	40%	50%	17%	60%	26%	0%	0%	0%	28%	33%	25%
HIST	0%	0%	0%	22%	40%	29%	100%	75%	86%	41%	38%	38%
JDeodorant	2%	60%	4%	5%	60%	9%	17%	25%	20%	8%	48%	11%
MLP	44%	80%	57%	75%	60%	67%	100%	50%	67%	**73%**	63%	63%
CAME	36%	80%	50%	80%	80%	80%	100%	75%	86%	72%	**78%**	**72%**

To what extent historical information about code metrics can improve detection performances?

For God Class detection, CAME shows an F-measure of 72% (precision of 72% and recall of 78%) on average over the three subject systems. Hence, the use of metrics historical information improves the detection performances by 14% in terms of F-measure. If we analyze the performances in detail, we see that in terms of precision, CAME achieves similar performances than the competitive approach with 72% for CAME against 73% for the MLP model. However, for recall we can see that using metrics history significantly improves the performance from 63% to 78% (23% improvement). Hence, CAME is able to identify occurrences of God Class that could not be detected without using historical information.

Using historical information as input of machine-learning anti-patterns detection models improves the performances in terms of F-measure and recall. Specifically, code metrics history does not have a significant impact on precision but reduces the number of misses

Does CAME outperform state-of-the-art detection tools?

Our results indicate that for God Class detection, CAME clearly outperforms state-of-the-art. Indeed, with respect to the approach that performed the best for God Class (i.e., HIST) the F-measure improves from 38% to 72% (89% improvement). Regarding precision and recall, the mean values improve respectively by 76% and 67% with respect to the best approaches (respectively HIST and JDeodorant). As mentioned in previous experiments, each competitive approach achieves poor performances on at least one software system which is not the case for CAME with an F-measure ranging from 50% to 86%.

For God Class CAME significantly outperforms state-of-the-art detection tools with an F-measure improvement of 89% with respect to the best approach. Furthermore, CAME achieves good performances on every tested systems

6.4 Conclusion and Future Work

In this chapter, we proposed CAME (Convolutional Analysis of Metrics Evolution) a deep-learning based approach for anti-patterns detection. Our approach first retrieves the values of source code metrics computed for each revision of the system under investigation by mining its version control system. Then this information is provided as input to a convolutional neural network which performs a binary classification over the entities of the system. This model has been trained using manually identified occurrences contained in five systems and instances of eleven other systems labeled using the ensemble method SMAD presented in chapter 5. We implemented our approach for the detection of God Class and evaluated its performances on three open source Java projects. Our results indicate that:

- Using historical information about code metrics for machine-learning based anti-patterns detection improves the performances in terms of F-measure. Particularly, our model is

able to identify more affected components than a similar model that rely on the same metrics computed solely for the investigated system's snapshot.

- CAME significantly outperforms state-of-the-art detection tools in detecting God Class and performs well independently of the systems characteristics.

Convolutional neural networks offer the possibility of visualizing their internal representation of what they have been trained to classify. This process is called feature visualization (Zeiler and Fergus (2014)). By doing so, one can visualize patterns in the input that influence the classification. Our future research agenda includes applying feature visualization techniques to CAME to understand the root causes of design anti-patterns.

CHAPTER 7 THREATS TO VALIDITY

In this chapter, we discuss the threats that could affect the validity of our studies.

7.1 Construct Validity

Threats to construct validity concern the relation between theory and observation. In our context, this could refer to the reliability of the oracle used to train and evaluate the different approaches investigated in this work. Instances of God Class extracted from HIST and DECOR replication packages have been filtered before being incorporated in our oracle. Furthermore, both papers have been awarded by the community, which confirms the quality of the processes conducted to produce these instances. For Feature Envy, we followed a strict blind procedure where each instance has been investigated by three different persons. However, we can not exclude the possibility of some missed occurrences or false positives. Another threat is related to the replication of some of the competitive approaches. We followed rigorously the guidelines provided by the respective authors, and as explained in Section 5.3.2 we performed an additional parameter tuning for each approach. However, some differences may remain between our respective implementations.

7.2 Internal Validity

Threats to internal validity concerns all the factors that could have impacted our results. In our context, this could refer to the training procedure presented in Section 4.4. Even though we compared the proposed procedure with conventional techniques while performing preliminary experiments. We did not report the results of our comparisons. Hence, a comparative study of the proposed procedure with conventional optimization approaches would be desirable. Also, we used such procedure along with other regularization techniques while training the model proposed by Liu et al. (2018). Note that these techniques are in fact part of the approach we propose and are necessary to train models on real imbalanced datasets. Another threat is related to choice of the architectures used in the approaches we propose (SMAD and CAME) for classification. We plan to investigate the use of different machine-learning algorithms to aggregate multiple anti-patterns detection tools for SMAD. For CAME, we plan to compare the current architecture with other deep-learning models such as Recurrent Neural Networks. Also, the size of the input matrix for CAME (1000 commits) was chosen arbitrarily. For future work, we plan to investigate the impact of this length on CAME's

performances. Finally, we choose a MLP model as a competitive approach to CAME in order to better assess the impact of metrics history in anti-patterns detection. Indeed, the MLP model is the closest architecture to CAME that could take a 1D vector as input without over-fitting. Furthermore, we used rigorously the same systems and procedure to train and perform hyper-parameters tuning on both models.

7.3 External Validity

Threats to external validity concern the generalizability of our findings. To reduce this threat, the software systems used for evaluation have been selected for their different domains, origins, sizes and history lengths. However, further evaluation of our models on a larger set of systems would be desirable.

CHAPTER 8 CONCLUSION

The impact of anti-patterns on software quality highlighted by number of empirical studies has motivated the development of various detection techniques. Although the proposed approaches have helped developers in identifying affected code components to be refactored, we identified a major limitation common to these works. Different detection techniques rely on different sources of information and thus, identify different sets of occurrences of anti-patterns. Hence, none of these works is able to truly address the problem of anti-patterns and to stand out among other approaches by identifying a significant proportion of the affected components. Recent trends have shown machine-learning models to be efficient in a variety of domains. Several machine-learning based approaches have been proposed to detect anti-patterns and address the issues encountered by previous approaches. However, these models have failed to clearly outperform conventional detection techniques. Indeed, Software systems are usually affected by a small proportion of anti-patterns. This characteristic lead to a strong imbalancement in anti-patterns datasets which compromises the performances of machine-learning models.

Consequently, we proposed two novel deep-learning based approaches for the detection of anti-patterns and implemented them for the detection of two widely known anti-patterns:God Class and Feature Envy. These models have in common to leverage both structural and historical sources of information to perform their predictions. To address the problem of imbalanced data, we designed a training procedure allowing to maximize the expected F-measure. Furthermore, we manually created an oracles reporting the occurrences of the studied anti-patterns in eight Java software systems to train and evaluate our approaches.

In the following, Section 8.1 synthesize our works while Section 8.2 discusses directions for future work.

8.1 Synthesis

To conduct our experiments, we created an oracle reporting the occurrences of God Class and Feature Envy in eight open source Java Projects. For God Class, we selected these occurrences from the replication packages of two works: Palomba et al. (2013) (HIST) and Moha et al. (2010) (DECOR). We also filtered the anti-patterns reported in these packages to make sure that the occurrences were reliable and that the full history of the systems were available. For Feature Envy, we created a set of potential occurrences by merging the detection

results of three state-of-the-art detection tools (HIST, InCode and JDeodorant), adjusting their detection thresholds to produce a number of candidate per system proportional to the systems' sizes. Then, each candidate have been manually analyzed by three different persons following a strict blind procedure.

To overcome the imbalancement problem in anti-patterns datasets, we designed a training procedure for feed-forward neural networks. This procedure presented in Section 4.4 consists in two main contributions. First, we designed a custom loss function to guide the optimization of our models. This loss function allows to minimize the expected mean F-measure achieved over the training systems (i.e., the empirical risk). Second, we proposed a feeding approach which is equivalent to a stochastic gradient descent considering each training system as a single instance. Furthermore, we proposed to use two regularization techniques, namely L_2 Regularization and Dropout to prevent our models from overfitting as well as an ensemble learning method to improve the quality of the performances reported.

Once the oracle and the training procedure have been defined, we proposed two novel deep-learning based detection techniques.

First, we proposed SMAD (SMart Aggregation of Anti-pattern Detectors), a machine-learning based ensemble method to efficiently aggregate existing detection tools on the basis of their internal detection rules. The workflow of SMAD is organized as follows. First, for each tool to be aggregated, we defined a set of *core metrics* ,i.e., metrics that reflect the internal decision process of each tool. Second, for each entity, these metrics are computed and fed into a dense feed-forward neural-network (i.e., a MLP) to perform classification. We implemented SMAD for the detection of God Class and Feature Envy and compared its performances with: (1) the tools aggregated through our approach and; (2) the baseline voting ensemble technique. Our results indicate that: (1) SMAD significantly improves the performances of the so aggregated tools and that it performs well independently of the systems characteristics and; (2) SMAD clearly outperforms the voting technique in terms of precision an F-measure. Furthermore, we observed that none of the voting policies are suitable to increase F-measure.

Another advantage of such ensemble method is its ability to label systems' instances for training other deep-learning anti-patterns detection models. Indeed, in a typical scenario, a training dataset is built by performing a manual validation over the occurrences detected by multiple approaches. Hence, SMAD can be used to automate this process and automatically label training instances from a reasonable number of manually-defined examples. Thus, we compared the performances achieved by two versions of the CNN proposed by Liu et al. (2018) for Feature Envy detection. The first version of this model have been trained by the authors on artificial Feature Envy occurrences while the second version have been trained

on instances labeled by SMAD. Our results show that the subject model achieves better performances when trained using instances labeled by our approach with respect to the same model trained on "injected smells" i.e., artificial occurrences.

Second, we proposed CAME (Convolutional Analysis of Mectrics Evolution), a deep-learning anti-patterns detection model which rely on source code metrics history. The workflow of CAME is organized as follows. First, we retrieve code metrics values for each revision of the system under investigation by mining its version control system. Second, this information, which has the shape of a 2D matrix, is provided as input to a convolutional neural network. We implemented CAME for the detection of God Class and compared its performances with those of: (1) a MLP model which relies on the same metrics than CAME without the historical part of the input and; (2) three state-of-the-art detection tools. Our results indicate that: (1) using code metrics historical information improves the performances of the models in terms of F-measure and recall; and (2) CAME significantly outperforms state-of-the-art detection tools. Furthermore we show that CAME achieves good performances independently of the systems characteristics.

8.2 Future Work

Our future research agenda manly focuses on the generalization of our findings. First, we plan to perform a comparative study of the training procedure proposed in section 4.4 with conventional optimization techniques. Second, we plan to extend our approaches to more anti-patterns and to experiment them on a larger set of systems.

For future research, we also plan to investigate the use of deep-learning visualization techniques (Zeiler and Fergus (2014)) on the architecture of CAME. We believe that such approach could help us identifying the root causes and characteristics of anti-patterns.

RÉFÉRENCES

M. Abbes, F. Khomh, Y.-G. Gueheneuc, et G. Antoniol, “An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension”, dans *Software maintenance and reengineering (CSMR), 2011 15th European conference on.* IEEE, 2011, pp. 181–190.

L. Amorim, E. Costa, N. Antunes, B. Fonseca, et M. Ribeiro, “Experience report: Evaluating the effectiveness of decision trees for detecting code smells”, dans *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on.* IEEE, 2015, pp. 261–269.

J. Bergstra et Y. Bengio, “Random search for hyper-parameter optimization”, *Journal of Machine Learning Research*, vol. 13, no. Feb, pp. 281–305, 2012.

L. C. Briand, J. W. Daly, et J. Wüst, “A unified framework for cohesion measurement in object-oriented systems”, *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, et T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} éd. John Wiley and Sons, March 1998.

A. Chatzigeorgiou et A. Manakos, “Investigating the evolution of bad smells in object-oriented code”, dans *International Conference on the Quality of Information and Communications Technology.* IEEE, 2010, pp. 106–115.

I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, et M. Shepperd, “A controlled experiment investigation of an object-oriented design heuristic for maintainability”, *Journal of Systems and Software*, vol. 72, no. 2, pp. 129–143, 2004.

D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, et A. De Lucia, “Detecting code smells using machine learning techniques: are we there yet?” dans *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2018, pp. 612–621.

T. G. Dietterich, “Ensemble methods in machine learning”, dans *International workshop on multiple classifier systems.* Springer, 2000, pp. 1–15.

C. dos Santos et M. Gatti, “Deep convolutional neural networks for sentiment analysis of short texts”, dans *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, 2014, pp. 69–78.

M. Fokaefs, N. Tsantalis, et A. Chatzigeorgiou, “Jdeodorant: Identification and removal of feature envy bad smells”, dans *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 519–520.

M. Fokaefs, N. Tsantalis, E. Stroulia, et A. Chatzigeorgiou, “Jdeodorant: identification and application of extract class refactorings”, dans *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1037–1039.

——, “Identification and application of extract class refactorings in object-oriented systems”, *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, 2012.

F. A. Fontana, P. Braione, et M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment.” *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

F. A. Fontana, M. V. Mäntylä, M. Zanoni, et A. Marino, “Comparing and experimenting machine learning techniques for code smell detection”, *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

M. Fowler, *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

E. Gamma, R. Helm, R. Johnson, et J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} éd. Addison-Wesley, 1994.

A. Graves, A.-r. Mohamed, et G. Hinton, “Speech recognition with deep recurrent neural networks”, dans *Acoustics, speech and signal processing (icassp), 2013 ieee international conference on*. IEEE, 2013, pp. 6645–6649.

Y.-G. Guéhéneuc, “Ptidej: Promoting patterns with patterns”, dans *Proceedings of the 1st ECOOP workshop on Building a System using Patterns. Springer-Verlag*, 2005.

H. He et E. A. Garcia, “Learning from imbalanced data”, *IEEE Transactions on Knowledge & Data Engineering*, no. 9, pp. 1263–1284, 2008.

D. H. Hubel et T. N. Wiesel, “Receptive fields, binocular interaction and functional architecture in the cat’s visual cortex”, *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

M. Jansche, “Maximum expected f-measure training of logistic regression models”, dans *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2005, pp. 692–699.

F. Khomh, M. Di Penta, et Y.-G. Gueheneuc, “An exploratory study of the impact of code smells on software change-proneness”, dans *Reverse Engineering, 2009. WCRE’09. 16th Working Conference on*. IEEE, 2009, pp. 75–84.

F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, et H. Sahraoui, “A bayesian approach for the detection of code and design smells”, dans *Quality Software, 2009. QSIC’09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

——, “Bdtex: A gqm-based bayesian approach for the detection of antipatterns”, *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, et G. Antoniol, “An exploratory study of the impact of antipatterns on class change-and fault-proneness”, *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

J. Kreimer, “Adaptive detection of design flaws”, *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

A. Krizhevsky, I. Sutskever, et G. E. Hinton, “Imagenet classification with deep convolutional neural networks”, dans *Advances in neural information processing systems*, 2012, pp. 1097–1105.

M. Lanza et R. Marinescu, *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer Science & Business Media, 2007.

Y. LeCun, L. Bottou, Y. Bengio, et P. Haffner, “Gradient-based learning applied to document recognition”, *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

H. Liu, Z. Xu, et Y. Zou, “Deep learning based feature envy detection”, dans *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 385–396.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, et E. Aimeur, “Smurf: A svm-based incremental anti-pattern detection approach”, dans *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 2012, pp. 466–475.

A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, et E. Aïmeur, “Support vector machines for anti-pattern detection”, dans *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 278–281.

R. Marinescu, “Detection strategies: Metrics-based rules for detecting design flaws”, dans *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*. IEEE, 2004, pp. 350–359.

R. Marinescu, G. Ganea, et I. Verebi, “Incode: Continuous quality assessment and improvement”, dans *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 274–275.

I. H. Moghadam et M. O. Cinneide, “Automated refactoring using design differencing”, dans *Software maintenance and reengineering (CSMR), 2012 16th European conference on*. IEEE, 2012, pp. 43–52.

N. Moha, Y. Guéhéneuc, D. Laurence, et L. M. Anne-Franccoise, “Decor: A method for the specification and detection of code and design smells”, *IEEE Transactions on Software Engineering (TSE)*, vol. 36, no. 1, pp. 20–36, 2010.

K. Nongpong, “Feature envy factor: A metric for automatic feature envy detection”, dans *Knowledge and Smart Technology (KST), 2015 7th International Conference on*. IEEE, 2015, pp. 7–12.

S. Olbrich, D. S. Cruzes, V. Basili, et N. Zazworka, “The evolution and impact of code smells: A case study of two open source systems”, dans *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. IEEE, 2009, pp. 390–400.

F. Palomba, G. Bavota, M. D. Penta, R. Oliverto, D. Poshyvanyk, et A. D. Lucia, “Mining version histories for detecting code smells”, *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, et D. Poshyvanyk, “Detecting bad smells in source code using change history information.” dans *ASE*, 2013, pp. 268–278.

F. Palomba, A. De Lucia, G. Bavota, et R. Oliveto, “Anti-pattern detection: Methods, challenges, and open issues”, dans *Advances in Computers*. Elsevier, 2014, vol. 95, pp. 201–238.

F. Palomba, D. D. Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, et A. De Lucia, “Landfill: An open dataset of code smells with public evaluation”, dans *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 482–485.

F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, et A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation”, *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

V. Sales, R. Terra, L. F. Miranda, et M. T. Valente, “Recommending move method refactorings using dependency sets”, dans *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 232–241.

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, et R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting”, *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

G. Suryanarayana, *Refactoring for Software Design Smells: Managing Technical Debt 1st Edition*. Morgan Kaufmann, 2014.

E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, et J. Noble, “The qualitas corpus: A curated collection of java code for empirical studies”, dans *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 336–345.

N. Tsantalis et A. Chatzigeorgiou, “Identification of move method refactoring opportunities”, *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, et D. Poshyvanyk, "When and why your code starts to smell bad", dans *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 2015, pp. 403–414.

S. Vaucher, F. Khomh, N. Moha, et Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes", dans *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on.* IEEE, 2009, pp. 145–154.

B. F. Webster, *Pitfalls of object-oriented development.* M & T, 1995.

I. H. Witten, E. Frank, M. A. Hall, et C. J. Pal, *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, 2016.

A. Yamashita et L. Moonen, "Do code smells reflect important maintainability aspects?" dans *2012 28th IEEE international conference on software maintenance (ICSM).* IEEE, 2012, pp. 306–315.

——, "Exploring the impact of inter-smell relations on software maintainability: An empirical study", dans *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 2013, pp. 682–691.

M. D. Zeiler et R. Fergus, "Visualizing and understanding convolutional networks", dans *European conference on computer vision.* Springer, 2014, pp. 818–833.