On the Detection of Community Smells Using Genetic Programming-based Ensemble Classifier Chain

Nuri Almarimi
ETS Montreal, University of Quebec
Montreal, QC, Canada
nuri.almarimi.1@ens.etsmtl.ca

Ali Ouni
ETS Montreal, University of Quebec
Montreal, QC, Canada
ali.ouni@etsmtl.ca

Moataz Chouchen
ETS Montreal, University of Quebec
Montreal, QC, Canada
moataz.chouchen.1@ens.etsmtl.ca

Islem Saidani
ETS Montreal, University of Quebec
Montreal, QC, Canada
islem.saidani.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
Rochester Institute of Technology
Rochester, NY, USA
mwmvse@rit.edu

ABSTRACT

Community smells are symptoms of organizational and social issues within the software development community that often increase the project costs and impact software quality. Recent studies have identified a variety of community smells and defined them as sub-optimal patterns connected to organizational-social structures in the software development community such as the lack of communication, coordination and collaboration. Recognizing the advantages of the early detection of potential community smells in a software project, we introduce a novel approach that learns from various community organizational and social practices to provide an automated support for detecting community smells. In particular, our approach learns from a set of interleaving organizational-social symptoms that characterize the existence of community smell instances in a software project. We build a multi-label learning model to detect 8 common types of community smells. We use the ensemble classifier chain (ECC) model that transforms multi-label problems into several single-label problems which are solved using genetic programming (GP) to find the optimal detection rules for each smell type. To evaluate the performance of our approach, we conducted an empirical study on a benchmark of 103 open source projects and 407 community smell instances. The statistical tests of our results show that our approach can detect the eight considered smell types with an average F-measure of 89% achieving a better performance compared to different state-of-the-art techniques. Furthermore, we found that the most influential factors that best characterize community smells include the social network density and closeness centrality as well as the standard deviation of the number of developers per time zone and per community.

CCS CONCEPTS

• **Software and its engineering → Software organization and properties**.

ICGSE '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7093-6/20/05…$15.00
https://doi.org/10.1145/3372787.3390439

KEYWORDS

Community smells, Social debt, Socio-technical factors, Multi-label learning, Genetic programming, Search-based software engineering

ACM Reference Format:
Nuri Almarimi, Ali Ouni, Moataz Chouchen, Islem Saidani, and Mohamed Wiem Mkaouer. 2020. On the Detection of Community Smells Using Genetic Programming-based Ensemble Classifier Chain. In *15th IEEE/ACM International Conference on Global Software Engineering (ICGSE '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372787.3390439

1 INTRODUCTION

Modern software engineering is increasingly dependent on the well-being of large globally distributed communities and their social networks in software development. Knowing more about the organizational structures of these communities and their social characteristics as well as the factors that affect their quality is critical to software projects success [17, 33, 49].

Recent studies explored a set of socio-technical patterns that can negatively impact the organizational health of software projects and coined them as *community smells* [52, 54]. Community smells are connected to circumstances based on poor organizational and social practices that lead to the occurrence of social debt [51, 54]. Social debt is connected to negative organized structures that often lead to short and/or long term social issues within a project. These problems could manifest in several forms, *e.g.*, lack of communications, collaboration or coordination among members in a software development community. For example, one of the common community smells, is the "*organizational silo effect*" (OSE) [45, 56] which manifests as a recurring social network sub-structure featuring highly decoupled developer's community structure. From an analytical perspective, the OSE smell can be interpreted as a set of patterns over a social network graph and could be detectable using different graph connectivity characteristics.

Detecting community smells is still, to some extent, a difficult, time consuming, and manual process. Indeed, there is no consensual way to translate formal definition and symptoms into actionable detection rules. Typically, the number of potential bad organizational practices often exceeds the resources available to address them. In many cases, mature software projects are forced to be developed with both known and unknown poor socio-technical community

practices for lack of resources to deal with every individual community smell. Furthermore, recent studies showed that different types of community smells can have similar symptoms and can thus co-exist in the same project. That is, different symptoms can be used to characterize multiple community smells making their identification even harder and error-prone [4, 14, 45, 55, 56]. For example, the Organizational Silo Effect (OSE) smell is typically associated with the *Solution Defiance* (DF) smell which manifests in the form of independent subgroups in the development team due to the variance in their cultural and experience levels. Although there have been few studies to define, characterize, and identify community smells characteristics/symptoms in software projects, they are applied, in general, to a limited scope, and their generalizability requires a manual effort and human expertise to define and calibrate a set of detection rules to match the symptoms of each community smell type with the actual characteristics of a given software project [4, 14, 45, 55, 56].

In this paper, our aim is to provide an automated technique to detect community smells in software projects. We formulate the problem as a multi-label learning (MLL) problem to deal with the interleaving symptoms of existing community smells by generating multiple smells detection rules that can detect various community smell types. We use the ensemble classifier chain (ECC) technique [48] that converts the detection task of multiple smell types into several binary classification problems for each individual smell type. ECC involves the training of n single-label binary classifiers, where each one is solely responsible for detecting a specific label, *i.e.*, community smell type. These n classifiers are linked in a chain, such that each binary classifier is able to consider the labels identified by the previous ones as additional information at the classification time. For the binary classification, we exploit the effectiveness of genetic programming (GP) [16, 20, 28, 29, 37, 39] to find the optimal detection rules for each community smell. The goal of GP is to learn detection rules from a set of real-world instances of community smells. In fact, we use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms with their appropriate threshold values to detect various types of community smells.

We implemented and evaluated our approach on a benchmark of 103 open source projects hosted in GitHub. We first conducted a survey with developers to validate the identified instances of community smells found in the studied projects. To evaluate the performance of our GP-ECC, the statistical analysis of our results shows that the generated detection rules can identify the eight considered community smell types with an average F-measure of 89% and outperforms state-of-the-art MLL techniques. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of community smells. We find that standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality within the social network are the most influential characteristics.

To sum up, the paper makes the main following contributions:

- We introduce a GP-based ensemble classifier chain (GP-ECC) approach to detect multiple community smell types that can exist in software projects as a multi-label learning (MLL) problem. To the best of our knowledge, this the first approach that uses MLL and GP for the problem of community smells detection.
- We conduct an empirical study to evaluate our approach on a benchmark of 103 software projects. Our results show that GP-ECC outperforms state-of-the-art single- and multi-label learning techniques.
- We conduct a survey with developers to validate the existence of community smells in our benchmark [1].
- We conduct an exploratory investigation to assess the factors that best characterize community smells in software projects.

Replication package. Our dataset is available online for future extension and replication [1].

Paper organization. Section 2 provides the necessary background. Section 3 summarizes the related work. In section 4, we describe our GP-ECC approach for community smells detection. Section 5 presents our empirical evaluation, and discusses the obtained results. Section 6 discusses the threats to validity. Finally, in Section 7, we conclude and outline our future work.

2 BACKGROUND

2.1 Community Smells Definitions

Community smells are defined as a set of social-organizational circumstances that occur within the software development community, having a negative effect on the relations health within the development community which may cause social debt over time [54]. A number of community smells have been defined in the literature. We refer to the following community smell types [4, 54]:

- **Organizational Silo Effect (OSE):** The OSE smell is manifested when too high decoupling between developers, isolated subgroups, and lack of communication and collaboration between community developers occur. The consequence of this smell is an extra unforeseen cost to a project by wasted resources (*e.g.*, time), as well as duplication of code [51, 54].
- **Black-cloud Effect (BCE):** The BCE smell occurs when developers have a lack of information due to limited knowledge sharing opportunities (*e.g.*, collaborations, discussions, daily stand-ups, etc.), as well as a lack of expert members in the project that are able to cover the experience or knowledge gap of a community. The BCE may cause a mistrust between members and creates selfish behavioral attitudes [54].
- **Prima-donnas Effect (PDE):** The PDE smell occurs when a team of people is unwilling to respect external changes from other team members due to inefficiently structured collaboration within the community. The presence of this smell may create isolation problems, superiority, constant disagreement, uncooperativeness and raise selfish team behavior, also called "prima-donnas" [51, 54].
- **Sharing Villainy (SV):** The SV smell is caused by a lack of high-quality information exchange activities (*e.g.*, face-to-face meetings). The main side effect of this smell limitation is that community members share essential knowledge such as outdated, wrong and unconfirmed information [54].
- **Organizational Skirmish (OS):** The OS smell is caused by a misalignment between different expertise levels and

communication channels among development units or individuals involved in the project. The existence of this smell leads often to dropped productivity and affect the project's timeline and cost [54].

- **Solution Defiance (SD):** The solution defiance smell occurs when the development community presents different levels of cultural and experience background, and these variances lead to the division of the community into similar subgroups with completely conflicting opinions concerning technical or socio-technical decisions to be taken. The existence of the SD smell often leads to unexpected project delays and uncooperative behaviors among the developers [54].
- **Radio Silence (RS):** The RS smell occurs when a high formality of regular procedures takes place due to the inefficient structural organization of a community. The RS community smell typically causes changes to be retarded, as well as a valuable time to be lost due to complex and rigid formal procedures. The main effect of this smell is an unexpected massive delay in the decision-making process due to the required formal actions needed [54].
- **Truck Factor Smell (TFS):** It occurs when most of the project information and knowledge are concentrated in one or few developers. The presence of this smell eventually leads to a significant knowledge loss due to the turnover of developers [4, 19].

In this paper, we focus primarily on these smells as they are widely studied and most occurring in the software industry as well as in open-source projects based on recent studies [19, 45, 51, 54].

2.2 Search Based Software Engineering

Search-Based Software Engineering (SBSE) consists of the application of a computational search to solve optimization problems in software engineering [23]. The term SBSE was coined by Harman and Jones in 2001, and the goal of the field is to move software engineering problems from human-based search to machine-based search, using a variety of techniques from the metaheuristic search and evolutionary computation paradigms [21, 23]. SBSE provides best practice in formulating a software engineering problem as a search problem, by defining a suitable solution representation, fitness function, and solution change operators. Indeed, there are a multitude of search algorithms ranging from single to many-objective techniques that can be applied to solve that problem [24, 38, 40–44]. In this paper, we apply SBSE to the problem of community smells detection in software projects. Hence, we show how genetic programming (GP) can effectively explore a large space of solutions, and provide intelligible detection rules with ECC.

2.3 Multi-label learning

Multi-label learning (MLL) is the machine learning task of automatically assigning an object into multiple categories based on its characteristics [15, 48, 57, 58]. Single-label learning is limited by one instance with only one label. MLL is a non-trivial generalization by removing the restriction and it has been a hot topic in machine learning [15, 58]. MLL has been explored in many areas in machine learning and data mining fields through classification techniques [57]. There exists different MLL techniques including (1) problem transformation methods and algorithms, *e.g.*, the classifier chain (CC) algorithm [48], the binary relevance (BR) algorithm [57], label powerset (LP) algorithm [59], and (2) algorithm adaptation methods such as the K-Nearest Neighbors (ML.KNN) [61], as well as (3) ensemble methods such as the ensemble classifier chain (ECC) [48] and random k-labelset (RAKEL) [57]. MLL was successfully applied to solve different software engineering problems [18, 31, 47, 60].

The Classifier Chain (CC) model. The CC model combines the computational efficiency of the BR method while still being able to take the label dependencies into account for classification. With BR, the classifier chains method involves the training of q single-label binary classifiers and each one will be solely responsible for classifying a specific label $l_1, l_2, ..., l_q$. The difference is that, in CC, these q classifiers are linked in a chain $\{h_1 \rightarrow h_2 \rightarrow ... \rightarrow h_q\}$ through the feature space. That is, during the learning time, each binary classifier h_j incorporates the labels predicted by the previous $h_1, ..., h_{j-1}$ classifiers as additional information. This is accomplished using a simple trick: in the training phase, the feature vector x for each classifier h_j is extended with the binary values of the labels $l_1, ..., l_{j-1}$.

The Ensemble Classifier Chain (ECC) model. One of the limitation of the CC model is that the order of the labels is random. This may lead to a single standalone CC model be poorly ordered. Moreover, there is the possible effect of error propagation along the chain at classification time, when one (or more) of the first classifiers predict poorly [48]. Using an ensemble of chains, each with a random label order, greatly reduces the risk of these events having an overall negative effect on classification accuracy. A majority voting method is used to select the best model. Moreover, a common advantage of ensembles is their well-known effect of generally increasing overall predictive performance [48].

In our study, we bridge the gap between MLL and SBSE based on the ECC method to solve the problem of community smells detection, where each project may contain different interleaving community smells, *e.g.*, OSE SV and BCE. For the binary labels, our ECC model adopts a search-based approach using genetic programming (GP) to learn detection rules for each smell type.

3 RELATED WORK

In the last few years, the software engineering community studied the importance of organizational and social patterns in open source software systems [4, 45, 55]. Tamburri et al. investigated potential community smells in software projects, and found that such smells lead to a social debt in sub-optimal organizational-social structures in software systems [19, 45, 54].

A number of approaches have been devised to support and analyze community smells in software projects. Tamburri et al. defined a set of community smells based on various socio-technical characteristics and patterns which may lead to the emergence of social debt [51]. Later, Tamburri et al. proposed a tool called YOSHI to automate to detect organizational structure patterns across open-source communities [55]. The proposed tool maps open-source projects onto community patterns, and introduces measurable attributes to identify organizational and social structure types and characteristics through formal detection rules. CodeFace is another tool developed as a Siemens product [27] and designed to identify developers communities based on building developer networks. Another tool called *Codeface4Smell* has been proposed later as an extension

of CodeFace to identify community issues based on community metrics and statistical values to measure the quality and health characteristics of development communities [56]. *Codeface4Smell* uses the development history and mailing lists to assess the social network among developers. Avelino et al. introduced a tool called *Truck Factor* [4, 19] to measure information concentration within community members and support the software development community to deal with turnover of developers. The proposed approach estimates the truck factor values of Github projects based on historical information about developers contributions and collaborations from the commit log. Recently, Palomba et al. found that community smells are connected to code smells [45].

Besides studies on community smells, many research efforts have carried out to study social debt in software engineering. Several studies on open-source and industrial projects have shows that collaboration and social structure are critical in software development and have an impact on the final software product quality [9, 10, 25, 33]. Other research works investigated the effect of organizational decisions on different collaborations and software product quality . Tamburri et al. studied social debt by a comparison with technical debt based on common real-life scenarios that exhibit sub-optimal development communities [53]. Bird et al. used different social network analysis techniques to assess the coordination between groups of developers with socio-technical dependencies [5, 6]. Cataldo et al. studied socio-technical congruence, *i.e.*, the degree to which technical and social dependencies match, using formal terms and conducted an empirical study to assess its impact on software product quality [7, 8, 10].

While the existing approaches attempt mainly to characterize and analyze organizational-social structures in software development communities, they do not cover multi-perspective characteristics of software projects. They mostly devise a limited and generic list of characteristics and symptoms to generate smells detection rules that characterize community smells. However, such generating detection rules will need a substantial human effort and expertise to calibrate these rules for each smell type and adapt them to different projects, organizations and contexts. Hence, we believe that an appropriate detection is needed to fill this gap. Our approach aims at learning from existing smells to detect new ones to help developers better allocate their resources and save time and efforts through automated smells detection.

4 APPROACH

In this section, we provide the problem formulation for community smells detection as a MLL problem. Then, we describe our approach.

4.1 Problem formulation

We define the community smells detection problem as a multi-label learning problem. Each community smell type is denoted by a label l_i. A MLL problem can be formulated as follows. Let $X = R^d$ denote the input feature space. $L = \{l_1, l_2, ...l_q\}$ denote the finite set of q possible labels, *i.e.*, smell types. Given a multi-label training set $D = \{(x_1, y_1), (x_2, y_2), ...(x_N, y_N)\}(x_i \in X, y_i \subseteq L)$, the goal of the multi-label learning system is to learn a function $h : X \rightarrow 2^L$ from D which predicts a set of labels for each unseen instance based on a set of known data.

In our approach, we used the ensemble classifier chain (ECC) model [48]. While the existing MLL methods, *e.g.*, BR and LP are flexible for solving MLL problems, they have the limitation of ignoring the correlations between labels. To address this issue in our approach, we adopt the ECC model [48], an extension of BR to exploit the advantage of label correlations, *i.e.*, smells correlation.

4.2 Approach Overview

Our approach starts from the observation that it is easier for developers to identify a set of detection rules to match the symptoms of a community smell with the actual characteristics of a given software project rather than relying on manual effort and human expertise [52]. The main goal of our approach is to generate a set of detection rules for each community smell type while taking into consideration the dependencies between the different smell types and their interleaving symptoms.

Figure 1 presents an overview of our approach to generate community smells detection rules using the GP-ECC model. Our approach consists of two phases: training phase and detection phase. In the training phase, our goal is to build an ensemble classifier chain (ECC) model learned from real-world community smells identified from software projects based on several GP models for each individual smell. In the detection phase, we apply this model to detect the proper set of labels (*i.e.*, types of community smells) for a new unlabeled data (*i.e.*, a new project).

Our framework takes as inputs a set of software projects with known labels, *i.e.*, community smells (phase A). Then, extracts a set of features characterizing the considered community smell types from which a GP algorithm will learn (phase B). Next, an ECC algorithm will be built (phase C). The ECC algorithm consists of a set of classifier chain models (CC), each with a random label order. Each CC model, learns eight individual GP models for each of the eight considered smell types. The i^{th} binary GP detector will learn from the training data while considering the existing i already detected smells by the $i-1$ detected smells generate the optimal detection rule that can detect the current i^{th} smell. In total, the ECC trains n multi-label CC classifiers $CC_1, ..., CC_n$; each classifier is given a random chain ordering; each CC builds 8 binary GP models for each smell type. Each binary model uses the previously predicted binary labels into its feature space. Then, our framework searches for the near optimal GP-ECC model from these n multi-label chain classifiers using an ensemble majority voting schema based on each label confidence [48].

In the detection phase, the returned GP-ECC model is a machine learning classifier that assigns multiple labels, *i.e.*, community smells types, to a new project based on its current features, *i.e.*, its socio-technical characteristics (phase D). In the next subsections, we provide the details of each phase.

4.3 Phase A : Training data collection

An important step to solving the problem of community smells detection is to prepare the learning dataset.

Projects selection: To build a base of real-world community smell examples that occur in software projects, we selected a set of software projects which are diverse in nature (*e.g.*, size, application domains, etc.) that have experienced community smells. We considered a set of open-source projects from Github to access to their

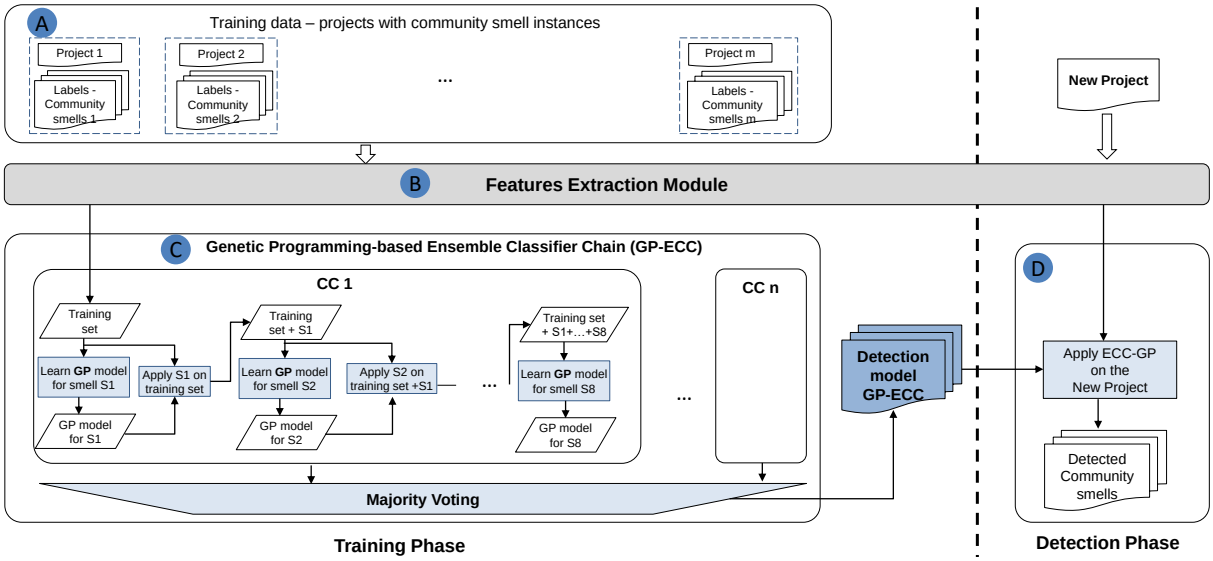

Figure 1: The community smells detection framework using the GP-ECC model.

development history. The considered filters to collect the training data are the following:

- *Commit size*: the projects vary from medium size >10 KLOC, large size (10- 60) to very large size < 60 KLOC.
- *Community size*: the projects team size vary from medium (<100 members), large (100-900), to very large (> 900).
- *Programming language*: the selected projects are implemented in different programming languages, including Java, C#, Python and C.
- *Existence of community smells:* the project should contain at least one community smell.

Base of examples: One of the main aspects of any attempt to apply a machine learning approach is to collect a base of examples to be used to train the model. We collected a set of community smell instances based on the symptoms and guidelines provided in the literature [19, 45, 51, 53–56, 56]. We manually analyzed the selected projects to identify potential smell occurrences.

To collect our base of examples, the authors checked individually all identified community smells if they match with the state-of-the-art definitions, characteristics and symptoms. All community smell instances that did not reach a full agreement were excluded from our base of examples. After the manual inspection of potential existence of community smells, we finally ended with 103 projects that have diverse types of community smells [1].

As an attempt to validate our identified smells, we conducted a survey with the original developers of the selected projects to get their feedback by following an opt-in strategy [26] for our survey. While the survey will help us to validate our identified symptoms, it can also help to understand whether developers are conscious of the presence of such smells in their projects. We extracted from GitHub the email address of the developers who have acted in a project at least 30 commit changes during the last 12 months and participated in the project in the last 3 years. In such a way, we focused only on developers having adequate experience with the project's community [50]. Before we sent the survey's questions, we first sent an email asking permission to participate in our study. As a result, we obtained a positive response from 62 out of 432 developers (14%) who were later contacted with the actual survey. We received answers from developers of 31 different projects in the dataset, which covered 29.3% of all considered smells. In our study, we are aware that the different opinions on this survey may not be necessarily generalized, but this analysis helps us to confirm whether our smells symptoms analysis match with the participants perception on such smells in their projects.

The survey's link was sent via email to all participants with a brief introduction. The list of questions is divided into two main parts as shown in Table 1. The first part consists of three control questions on the project name and background information about the participants occupation and experience with the project.

Then, we asked developers to rate the validity of 11 statements that we extracted from the community smells definitions (cf. Section 2) using a 5 point Likert scale [30] ranging between "*Strongly Disagree*" to "*Strongly Agree*". This part presents typical situations in which community smells occur without mentioning the term social debt or smells in the questionnaire. To avoid any possible bias in the responses, we did not ask the developers directly to rate how healthy was their community. For instance, the statement "*Do you think there is a lack of communications and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?*" was aimed at understanding whether developers actually recognize the presence of the symptoms of an OSD smell in their project. The complete questions list of our survey presented in Table 1. After collecting the participants answers, we compared their answers with the smell instances detected manually in our base of examples to make sure that they match.

4.4 Phase B : Features Extraction Module

To capture community smells symptoms, we rely on a set of metrics defined in previous studies [4, 14, 34, 35, 46, 56], as well as a new set of generated metrics to capture more community-related proprieties that can be mined from the projects history. These metrics analyze different aspects in software development communities

Table 1: Questionnaire filled in by the study participants.

Part 1: Background Information:	
1	What is your project name on GitHub ? * [*Required Answer*]
2	How do you describe your occupation in this project? [*Student – Part-time employee – Full-time employee – Unemployed – Retired – Other*]
3	How long time you have been working in open source projects? [*Less than 3 years – 3 years or more – Other*]
Part 2: Social aspects perception	
4	Do you think there is a lack of communication and cooperation within the development team (share knowledge, experiences, exchange activities, etc.) in your project?
5	Do you think there is a delay in communications between developers in your project?
6	Do you think there is/are developer(s) in your project working on the same code and communicating by means of a third person?
7	Do you think there is/are in your project developer(s) who have selfish and/or egoistical behaviors?
8	Do you think there are different communication and expertise levels that may cause misalignment within the development team in your project?
9	Do you think there are different levels of cultural and experiential backgrounds in the development team?
10	Do you think there is/are isolated subgroup(s) within the development team of your project?
11	Do you think there is in your project a formal and complex organizational structure (with potentially unnecessary "regular procedures")?
12	Do you think there is in your project some unique knowledge and information brokers toward different developers?
13	Do you think there is in your project a risk that a core developer who can unexpectedly leave the project and can influence the development process?
14	Do you think there is in your project a waste of resources (*e.g.*, time) over the development life-cycle?

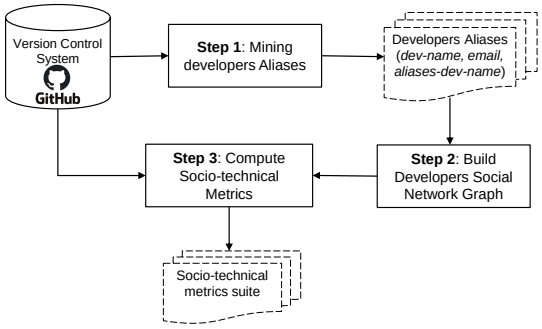

Figure 2: An overview of the features extraction module.

including organizational dimensional, social network characteristics, developers collaborations, and truck numbers. Table 2 depicts our list of considered features. Our proposed metrics extend existing metrics to provide more details including developers social network, community structures, geographic dispersion, and developer network formality. For instance, the geographic dispersion metrics, *e.g.*, the *average number of commits per time zone* and the *standard deviation of developers per time zone* would capture the distribution of commits and developers per time zone. Our features extraction module calculates the set of metrics from a given project by analyzing its repository through commit information history available in its version control system, *e.g.*, GitHub. Figure 2 depicts an overview of our features extraction module which consists of two main steps to extract metrics.

Step 1. Mine developers aliases: The author alias mining and consolidation consists of the following sub-steps [4]: (*i*) retrieval all unique dev-emails, where for each git commit has a dev-email associated with it, (*ii*) Retrieval of GitHub logins related to developers, (*iii*) similarity matching of emails and logins: by applying Levenshtein distance [4], all aliases are compared and, with a certain degree of threshold value at most one, consolidated, and (*iv*) replacement of author emails by their respective aliases [4]. The final transformation goes through all the commits once again and replacing original authors by their main alias. As a result, if there is a developer associated with commits with different names, we consider them as a single developer, and the output will be presented in a new aliases list. For example, "Bob.Rob" and "Bob Rob" are different names for a single developer associated to commits. Thus, we consider them as the same developer in a new aliases list as a single identical substitution.

Step 2. Build a social network graph: Social networks analysis (SNA) have been used for studying and analyzing the collaboration and organization of developers who are working in teams within software development projects [32]. Our developers network model is based on a socio-technical connection during a software project development. Different social network analysis metrics have been devised to describe a community structure and predict quality factors in a software development project. Our approach builds a developer network from the version control system by tracking the change logs. Our adopted developers network is presented as a graph of nodes and edges, where the nodes represent developers and edges are the connections between two developers that are working on the same file and where they make a version control commit within one month of each side. Such social network allows then to calculate the different metrics including the degree centrality, closeness centrality, network density, etc. (cf. Table 2).

Step 3. Compute Socio-technical Metrics: In this step, we use the collected informations and social network graph to compute a variety of 30 socio-technical metrics as described in Table 2.

4.5 Phase C : Genetic Programming-based Ensemble Classifier Chain (GP-ECC)

As explained earlier in Sections 4.2 and 2.3, our approach is based on the ECC method [48] that transforms the multi-label learning task into multiple single-label learning tasks. Our multi-label ECC model aims at building a detection model to detect different instances of community smells in a software project. Each classifier chain (CC) builds a GP model for each smell type while considering the previously detected smells (if any), *i.e.*, each binary GP model uses the previously predicted binary labels into its feature space. Our choice for GP is motivated by the high performance of GP in solving challenging software engineering problems including design defects, code smells and anti-patterns detection [29, 36, 37, 39].

In our approach, we adopted the Multi-objective Genetic Programming (MOGP) as search algorithm to generate smells detection rules. MOGP is a powerful and widely-used evolutionary algorithm which extends the generic model of GP learning to the space of programs. Unlike other evolutionary search algorithms, in MOGP, solutions are themselves programs following a tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols [28].

Table 2: Socio-technical metrics framework.

Dimension	ID	Definition	Ref.
Developer Contributions metrics	NoD	*Number of developers (NoD)*: the total number of developers who have changed the code in a project.	[34]
	NAD	*Number of Active Days of an author on a project (NAD)* : the percentage of the total number of active days for each developer with respect to a project's lifetime on the total number of developers in a project.	[34]
	NCD	*Number of Commits per Developer in a project (NCD)* : the total number of times that the code has been changed by a developer with respect to the total number of commits and the total number of developers in a project.	[34]
	SDC	*Standard Deviation of Commits per developer in a project (SDC)* : the standard definition of commits per developer in a project. It provides a view of the distribution of the developers contributions.	[34]
	NCD	*Number of Core Developers (NCD)*: the total number of core developers in a project. A developer is considered as a core community member if he/she has a larger degree than peripheral developers within the developer's social network.	[56]
	PCD	*Percentage of Core Developers (PCD)*: the percentage of core developers with respect to the total number of developers.	[56]
	NSD	*Number of Sponsored Developers (NSD)*: the total number of sponsored developers in a project. We consider a developer that hold a sponsored status if at least 90% of her/his commits are performed during weekdays and the working day time (8am-6pm).	[56]
	PSD	*Percentage of Sponsored Developers (PSD)*: the percentage of sponsored developers over the total number of developers.	[56]
Social Network Analysis metrics	GDC	*Graph Degree Centrality (GDC)*: the number of connections that a developer has. The more connections with others a developer has, the more important the developer is.	[46]
	SDD	*Standard Deviation of a graph Degree centrality in a project (SDD)* : the standard deviation of the degree centrality (DC) of each developer in a project. It provides a view of the distribution of DC in a project.	[56]
	GBC	*Graph Betweenness Centrality (GBC)*: a measure of the information flow from one developer to another and devised as a general measure of social network centrality. It represents the degree to which developers stand between each other. A developer with higher BC would have more control over the community as more information will pass through her/him.	[46]
	GCC	*Graph Closeness Centrality (GCC)*: a measure of the distance between a developer to other developers in the network. This metric is strongly influenced by the degree of connectivity of a network.	[46]
	ND	*Network Density (ND)*: a measure of a social network as a dense or sparse graph.	[56]
Community metrics	NC	*Number of Communities (NC)*: the total number of communities in a project.	[56]
	ACC	*Average of Commits per Community (ACC)*: the average number of commits per community in a project.	New
	SCC	*Standard deviation of Commits per Community (SCC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
	ADC	*Average number of Developers per Community (ADC)*: the average number of developers per community in a project with respect to the total number of developers in a project.	New
	SDC	*Standard deviation of Developers per Community (SDC)*: the standard deviation of commits performed by each community with respect to the total number of commits in a project.	New
Geographic Dispersion metrics	TZ	*Number of Time Zones (TZ)*: the total number of different time zones of developers in a project.	[35]
	ACZ	*Average of Commits per time Zone (ACZ)*: the average number of commits per time zone in a project.	New
	SCZ	*Standard deviation of Commits per time Zones (SCZ)*: the standard deviation of commits performed in each time zone with respect to the total number of commits in a project.	New
	ADZ	*Average number of Developers per time Zone (ADZ)*: the average number of developers per time zone in a project.	New
	SDZ	*Standard deviation of Developers per time Zones (SDZ)*: the standard deviation of developers per time zones in a project.	New
Formality metrics	NR	*Number of Releases in a project (NR)*: the total number of releases delivered in a project.	[2]
	PCR	*Parentage of Commits per Release (PCR)* : the percentage of commits of each release over the total number of releases in a project.	New
	SCR	*Standard deviation of Commits per Release (SCR)* : the standard deviation of developers per release in a project.	New
	FN	*Formal Network (FN)*: the number of milestones assigned to the project with respect to the lifetime of the project.	[14]
Truck Number metrics	BFN	*Bus Factor Number (BFN)*: is the percentage of active developers present in a project with respect to the total number of developers.	[13]
	TFN	*Truck Factor Number (TFN)*: the number of key developers in a project who can be unexpectedly lost, *i.e.*, hit by a truck before the project is discontinued.	[4]
	TFC	*Truck Factor Coverage (TFC)*: the percentage of core developers and their associated authored files in a project.	[4]

As described in Algorithm 1, MOGP starts by randomly creating an initial population P_0 of individuals encoded using a specific representation (line 1). Then, a child population Q_0 is generated from the population of parents P_0 (line 2) using genetic operators (crossover and mutation). Both populations are merged into an initial population R_0 of size N (line 5). *Fast-non-dominated-sort* [16] is the technique used by MOGP to classify individual solutions into different dominance levels (line 6) [16]. The whole population that contains N individuals (solutions) is sorted using the dominance principle into several fronts (line 6). Solutions on the first Pareto-front F_0 get assigned dominance level of 0. Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front F_1 of the remaining population; solutions on this second front get assigned dominance level of 1, and so on. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When MOGP has to cut off a front F_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [16] to make the selection (line 9). This parameter is used to promote diversity within the population. The front F_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of F_i are chosen (line 14). Then, a new population Q_{t+1} is created using selection, crossover and mutation (line 15). This process will be repeated until reaching the last iteration according to stop criteria (line 4).

We describe in the following subsections the three main adaptation steps: (*i*) solution representation, (*ii*) the generation of the initial generation (*iii*) fitness function, and (*iv*) change operators.

Algorithm 1 High level pseudo code of the adopted MOGP

```
Create an initial population P_0
Create an offspring population Q_0
t = 0
while stopping criteria not reached do
    R_t = P_t ∪ Q_t
    F = fast-non-dominated-sort(R_t)
    P_{t+1} = ∅ and i = 1
    while | P_{t+1} | + | F_i | ⩽ N do
        Apply crowding-distance-assignment(F_i)
        P_{t+1} = P_{t+1} ∪ F_i
        i = i + 1
    end while
    Sort(F_i, ≺ n)
    P_{t+1} = P_{t+1} ∪ F_i[N− | P_{t+1} |]
    Q_{t+1} = create-new-pop(P_{t+1})
    t = t+1
end while
```

(*i*) Solution representation. A solution consists of a rule that can detect a specific type of community smells in the form of IF-THEN: In MOGP, a solution is represented as a tree composed of terminals and functions. The terminals correspond to different socio-technical specific features (cf. Table 2) with their threshold values. The functions that can be used between these metrics are logic operators OR (union), AND (intersection), or XOR (eXclusive OR). A solution is represented as a tree a binary tree such that: each leafnode (Terminal) contains one of metrics described in Table 2 and their corresponding threshold values generated randomly. Each internal-node (Functions) belongs to the Connective (logic operators) set $C = \{AND, OR, XOR\}$. The threshold values are selected randomly along with the comparison and logic operators. Figure 3 shows a simplified example of a solution for the OSE smell using the metrics GDC, ADC, GBC and SDZ.

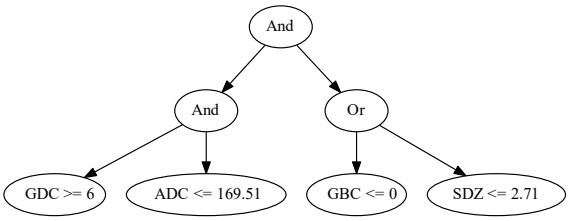

Figure 3: A simplified example of a solution for OSE smell.

(*ii*) Generation of the initial population. The initial population of solutions is generated randomly by assigning a variety of metrics and their thresholds to the set of different nodes of the tree. The size of a solution, *i.e.*, the tree's length, is randomly chosen between lower and upper bound values. These two bounds have determined and called the problem of bloat control in GP, where the goal is to identify the tree size limits. Thus, we applied several trial and error experiments using the HyperVolume (HP) performance indicator [3] to determine the upper bound after which, the sign remains invariant.

(*iii*) Fitness function. The fitness function evaluates how good is a candidate solution in detecting community smells. Thus, to evaluate the fitness of each solution, we use two objective functions, based on two well-known metrics [22, 29, 37], to be optimized, *i.e.*, precision and recall. The precision objective function aims at maximizing the detection of correct community smells over the list of detected ones. The recall objective function aims at maximizing the coverage of expected community smells from the base of examples over the actual list of detected smells. Precision and recall of a solution S are defined as follows.

$$Precision(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Detected smells}\}|} \quad (1)$$

$$Recall(S) = \frac{|\{\text{Detected smells}\} \cap \{\text{Expected smells}\}|}{|\{\text{Expected smells}\}|} \quad (2)$$

(*iv*) Change operators. Crossover and mutation are used as change operators to evolve candidate solutions towards optimality.

Crossover. We adopt the "standard" random, single-point crossover. It selects two parent solutions at random, then picks a sub-tree on each one. Then, the crossover operator swaps the nodes and their relative subtrees from one parent to the other. Each child thus combines information from both parents.

Mutation: It can be applied either to a function node or a terminal node. This operator can modify one or many nodes. For a selected solution, the mutation operator first randomly selects a node in the tree. Then, if the selected node is a terminal (metric), it is replaced by another terminal (metric or another threshold value); if the selected node is a function (AND-OR-XOR operators), it is replaced by a new function (*e.g.*, AND becomes OR). If a tree mutation is to be carried out, the node and its sub-tree are replaced by a new randomly generated sub-tree.

ECC majority voting. As shown in Figure 1, for each CC, MOGP will generate an optimal rule for each community smell type, *i.e.*, binary detection. Then, ECC allows to find the best CC that provides the best MLL from all the trained binary models. Each CC_i model is likely to be unique and able to achieve different multi-label classifications. These classifications are summed by label so that each label receives a number of votes. A threshold is used to select the most popular labels which form the final predicted multi-label set. This is a generic voting scheme and it is straightforward to apply an ensemble of any MLL transformation method [48].

4.6 Phase D : Detection Phase

After the GP-ECC model is constructed, in the training phase, it will be then used to detect a set of labels for a new project. It takes as input the set of features extracted from a given project using the feature extraction module. As output, it returns the detection results for each individual label, *i.e.*, community smell type.

5 EVALUATION

This section reports our empirical study to evaluate our approach including the research questions, experiments setup and results.

5.1 Research Questions

We designed our empirical study to answer the three following research questions.

- **RQ1: (Performance)** How accurately can our GP-ECC detect community smells?
- **RQ2: (Sensitivity)** What types of community smells does our GP-ECC approach detect correctly?
- **RQ3: (Features influence)** What are the most influential features that can indicate the presence of community smells?

5.2 Analysis method

To evaluate our approach, we collected a set of community smells as discussed in Section 4.3. Table 3 summarizes the collected smells. Furthermore, as a sanity check, all smells were manually inspected and validated based on guidelines from the literature as well as through our survey with the original developers. Furthermore, our dataset is available online for future extension and replication [1].

We considered eight common types of community smells, *i.e.*, *organisational silo effect* (OSE), *black-cloud effect* (BCE), *prima-donnas effect* (PDE), *sharing villainy* (SV), *organisational skirmish* (OS), *solution defiance* (SD), *radio silence* (RS), and *truck factor* (TF), (cf. Section 2). In our experiments, we conducted a 10-fold cross-validation procedure to split our data into training data and evaluation data.

Table 3: Dataset statistics.

Data	Statistic
Number of projects	103
Number of projects having at least one smell	103
Total number of smells	407
Average number of smells per project	4.6
Average number of developers per project	229
Number of projects with <50 developers	40
Number of projects with 50 − 150 developers	34
Number of projects with >150 developers	29
Average number of commits per project	1,103
Average number of days in each project	3,233

To answer **RQ1**, we carry out a set of experiments to justify our GP-ECC approach. We first compare the performance of our meta-algorithm ECC to two well-known meta-algorithms with proven success in MLL, *random k-labelset* (RAKEL) [57] and *binary relevance* (BR) [57]. We next used GP, *decision tree* (J48) and *random forest* (RF) as their corresponding underlying classification algorithms. We also compared with the widely used MLL algorithm adaptation method, *K-Nearest Neighbors* (ML.KNN) [61]. Thus, in total, we have 10 MLL algorithms to be compared. One fold is used for the test and 9 folds for the training.

To compare the performance of each method, we use common performance metrics, *i.e.*, precision, recall, and F-measure [29, 37, 48, 60]. Let l a label in the label set L. For each instance i in the smells learning dataset, there are four outcomes, True Positive (TP_l) when i is detected as label l and it correctly belongs to l; False Positive (FP_l) when i is detected as label l and it actually does not belong to l; False Negative (FN_l) when i is not detected as label l when it actually belongs to l; or True Negative (TN_l) when i is not detected as label l and it actually does not belong to l. Based on these possible outcomes, precision (P_l), recall (R_l) and F-measure (F_l) for label l are defined as follows:

$$P_l = \frac{TP_l}{TP_l + FP_l} \quad ; \quad R_l = \frac{TP_l}{TP_l + FN_l} \quad ; \quad F_l = \frac{2 \times P_l \times R_l}{P_l + R_l}$$

Then, the average precision, recall, and F-measure of the $\mid L \mid$ labels are calculated as follows:

$$Precision = \frac{1}{\mid L \mid} \sum_{l \in L} P_l \quad ; \quad Recall = \frac{1}{\mid L \mid} \sum_{l \in L} R_l \quad ; \quad F1 = \frac{1}{\mid L \mid} \sum_{l \in L} F_l$$

Statistical test methods. To compare the performance of each method, we perform Wilcoxon pairwise comparisons [12] at 99% significance level (*i.e.*, $\alpha = 0.01$) to compare GP-ECC with each of the 9 other methods. We also used the non-parametric effect Cliff's delta (d) [11] to compute the effect size. The effect size d is interpreted as Negligible if $\mid d \mid < 0.147$, Small if $0.147 \leq \mid d \mid < 0.33$, Medium if $0.33 \leq \mid d \mid < 0.474$, or High if $\mid d \mid \geq 0.474$.

To answer **RQ2**, we investigated the community smell types that were detected to find out whether there is a bias towards the detection of specific smell types.

To answer **RQ3**, we aim at identifying the features that are the most important indicators of whether a project has a given community smell or not. For each smell type, we count the percentage of rules in which the feature appears across all obtained optimal rules by GP. The more a feature appears in the set of optimal trees, the more the feature is relevant to characterize that smell.

Algorithms parameters. For all the GP, RF and J48 algorithms, the maximum depth of the tree is set to 10. For GP, the population size is 200, number of iterations is 3,000, crossover and mutation rates are 0.9 and 0.1, respectively. For RF and J48, we used the default parameters of Weka. The number of neighbors of ML.KNN is set to 10. For ECC, we set the ensembles size $n = 20$. For RAKEL, we set $n = 20$, and the labels subset $k = 4$.

5.3 Results

Results for RQ1 (Performance). Table 4 reports the average precision, recall and F-measure scores for the 10 methods. We observe that ECC competes well against the other 2 meta algorithms RAKEL and BR methods. Looking at the base learning methods (GP, J48 and RF), we used GP-ECC as the base for determining statistical significance. In particular, the GP-ECC method achieves the highest F-measure with 0.89 compared to the other methods with medium and large effect sizes, except with GP-RAKEL for which the results were statistically different but with small different effect size. The same performance was achieved in terms of precision and recall, with 0.87 and 0.91, respectively. Moreover, we observe that GP-ECC achieves comparable performance as GP-RAKEL in terms of recall which confirms the suitability of the GP formulation compared to decision tree and random forest algorithms. We can also see overall superiority for GP-ECC compared to the transformation method ML.KNN in terms of precision, recall and F-measure with large effect size. One of the reasons that BR does not perform well is that it ignores the label correlation, while RAKEL and ECC consider the label correlation by using an ensemble of classifiers. Moreover, among the 3 base learning algorithms, GP performs the best, followed by decision tree (J48) and random forest (RF).

Results for RQ2 (Sensitivity). Figure 4 reports the sensitivity analysis of each specific community smell type. We observe that GP-ECC does not have a bias towards the detection of any specific smell type. As shown in the figure, GP-ECC achieved good performance and low variability in terms of both precision (ranging from 0.84 to 0,9) and recall (ranging from 86 to 92) across the 8 considered smell types. The highest precision and recall was obtained for the organisational silo effect (OSE), the track factor (TF), and solution defiance (SD) which heavily relies on the notion of developers social network and sub-groups. This higher performance is reasonable since the existing guidelines [19, 51, 53–56, 56] rely heavily on the notion of social network. But for smells such as organisational skirmish (OS) and radio silence (RS), the notion of social network

Table 4: The achieved results by each of the meta-algorithms ECC, RAKEL and BR with their base learning algorithms GP, J48, and RF; and ML.KNN.

Algorithm	Precision		Recall		F1	
	score	p-value (d)*	score	p-value (d)*	score	p-value (d)*
GP-ECC	**0.87**	-	**0.91**	-	**0.89**	-
J48-ECC	0.84	<0.01 (M)	0.89	<0.01 (S)	0.86	<0.01 (M)
RF-ECC	0.84	<0.01 (M)	0.87	<0.01 (M)	0.85	<0.01 (M)
GP-RAKEL	0.85	<0.01 (S)	**0.91**	No. Stat. Sig.	0.88	<0.01 (S)
J48-RAKEL	0.83	<0.01 (L)	0.88	<0.01 (M)	0.85	<0.01 (L)
RF-RAKEL	0.84	<0.01 (M)	0.86	<0.01 (M)	0.85	<0.01 (M)
GP-BR	0.83	<0.01 (L)	0.85	<0.01 (M)	0.84	<0.01 (L)
J48-BR	0.81	<0.01 (L)	0.82	<0.01 (M)	0.81	<0.01 (L)
RF-BR	0.82	<0.01 (L)	0.82	<0.01 (L)	0.82	<0.01 (L)
ML.KNN	0.82	<0.01 (L)	0.84	<0.01 (L)	0.83	<0.01 (L)

* p-value(d) reports the statistical difference (p-value) and effect-size (d) between GP-ECC and the algorithm in the current row.
The effect-size (d) is N : Negligible − S : Small − M : Medium − L : Large

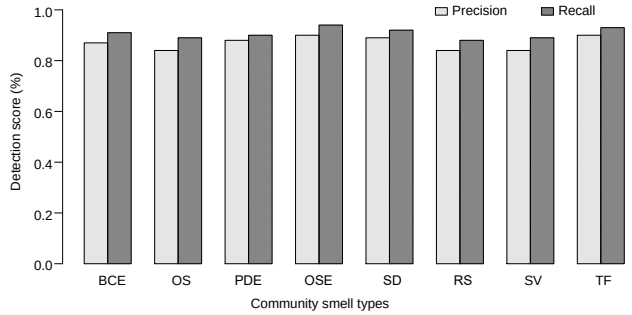

Figure 4: The achieved precision and recall scores by GP-ECC for each communiy smell type.

is less important and this makes this type of smells hard to detect using such information.

Results for RQ3 (Features influence). To better understand what features are most selected by our GP to generate detection rules among all the generated rules, we count the percentage of rules in which the feature appears. Table 5 shows the statistics for each smell type with the top-10 features (cf. Table 2), from which the three most influencing features values are in bold. We observe that the graph betweeness, closeness and degree centrality (GBC, GCC, and GDC), the network density (ND), the standard deviation of developers per community and per time zone (SDC and SDZ) as well as the number of communities (NC). We thus observe that different social network patterns play a crucial role in the emergence of community smells. These findings suggest that more attention has to be paid to these particular socio-organizational characteristics within the software project community to avoid the presence of smells and their impact on the software project.

6 THREATS TO VALIDITY

Threats to construct validity could be related to the performance measures. We basically used standard performance metrics such as precision, recall and F-measure that are widely accepted in MLL and software engineering [29, 37, 39, 48, 60]. Another potential

Table 5: The most influential features for each smell.

metric	OSE	BCE	PDE	SV	OS	SD	RS	TF
GDC	**95**	**91**	**92**	**92**	**90**	83	**96**	**95**
GCC	91	**88**	**89**	**95**	**93**	**91**	91	**93**
SDZ	87	53	**89**	**88**	62	88	72	63
ND	**96**	81	82	**88**	**90**	**92**	92	93
GBC	**93**	**92**	**90**	71	**91**	**96**	92	92
NC	82	62	72	43	52	62	**95**	72
SDC	91	**88**	**89**	**88**	**91**	66	82	**95**
TZ	76	48	72	62	53	**92**	**97**	47
TFN	18	15	21	41	22	42	39	**100**
PSD	53	21	18	45	32	62	65	81

threat could be related to the selection of classification techniques. Although we use the GP, J48 and RF techniques which are known to have high performance, there are other techniques. To mitigate this threat, we plan to compare with other ML techniques.

Threats to internal validity relate to errors in our experiments. We have double checked our experiments and the datasets collected following the literature guidelines and a survey with developers [51, 52, 54, 56], still there could be errors that we did not notice.

Threats to external validity relate to the generalizability of our results. We have analyzed a total of 407 smell instances from 103 different open source projects, different community sizes and programming languages. In the future, we plan to reduce this threat further by analyzing more projects from more industrial and open-source software projects.

7 CONCLUSION AND FUTURE WORK

We introduced in this paper an automated approach to detect community smells in software projects. We formulate the problem as a multi-label learning problem using the ECC meta-algorithm with an underlying GP model. Our GP-ECC aims at generating detection rules for each smell type. We use GP to translate regularities and symptoms that can be found in real-world community smell examples into detection rules. A detection rule is a combination of socio-technical attributes/symptoms with their appropriate threshold values to detect various types of community smells. We evaluated our approach on a set of 103 projects and 407 smell instances across 8 common types of community smells. Results show that our GP-ECC approach can identify all the considered community smell types with an average F-measure of 89% and outperforms 9 state-of-the-art MLL techniques that rely on different meta-algorithms (ECC, BR and RAKEL) and different underlying learning algorithms (GP, J48, and RF); and a transformation method ML.KNN. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of community smells. We find that the standard deviation of the number of developers per time zone and per community, and the social network betweenness, closeness and density centrality are the most influential characteristics.

As future work, we plan to extend our approach with more open source and industrial projects to provide ampler empirical evaluation. We plan also to extend our approach to provide software project managers with community change recommendations to avoid social debt in their projects. We also plan to assess the impact of community smells on different aspects of software projects.

REFERENCES

[1] 2020. Replication Package. https://github.com/GP-ECC/community-smells

[2] T. Mukhopadhyay A. Gopal and M. S. Krishnan . 2002. The role of software processes and communication in offshore software development. In *Communications of the ACM April 2002*. Association for Computing Machinery ,New York, NY, United States, USA, 1106–1113. https://doi.org/10.1145/505248.506008

[3] Peter John Angeline. 1994. Genetic programming and emergent intelligence. *Advances in genetic programming* 1 (1994), 75–98.

[4] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10.

[5] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. 2009. Does distributed development affect software quality? An empirical case study of Windows Vista. In *Proceedings of the 31st international conference on software engineering*. 518–528.

[6] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. 2009. Putting it all together: Using socio-technical networks to predict failures. In *20th International Symposium on Software Reliability Engineering*. 109–119.

[7] Marcelo Cataldo, James D Herbsleb, and Kathleen M Carley. 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*. 2–11.

[8] Marcelo Cataldo, Audris Mockus, Jeffrey A Roberts, and James D Herbsleb. 2009. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering* 35, 6 (2009), 864–878.

[9] Marcelo Cataldo and Sangeeth Nambiar. 2009. On the relationship between process maturity and geographic distribution: an empirical analysis of their impact on software quality. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 101–110.

[10] Marcelo Cataldo and Sangeeth Nambiar. 2012. The impact of geographic distribution and the nature of technical coupling on the quality of global software development projects. *Journal of software: Evolution and Process* 24, 2 (2012), 153–168.

[11] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.

[12] Jacob Cohen. 2013. *Statistical power analysis for the behavioral sciences*. Routledge.

[13] V. Cosentino, J. L. C. Izquierdo, and J. Cabot. 2015. Assessing the bus factor of Git repositories. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 499–503.

[14] Stefano Invernizzi Elisabetta Di Nitto Damian A. Tamburri, Simone Gatti. 2016. Re-Architecting Software Forges into Communities: An Experience Report. In *JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS*. 1–26.

[15] André C. P. L. F. de Carvalho and Alex A. Freitas. 2009. *A Tutorial on Multi-label Classification Techniques*. 177–195.

[16] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[17] Yvonne Dittrich, Jacob Nørbjerg, Paolo Tell, and Lars Bendix. 2018. Researching cooperation and communication in continuous software engineering. In *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 87–90.

[18] Yang Feng and Zhenyu Chen. 2012. Multi-label software behavior learning. In *34th International Conference on Software Engineering (ICSE)*. 1305–1308.

[19] Mívian Ferreira, Guilherme Avelino, Marco Tulio Valente, and Kecia AM Ferreira. 2016. A Comparative Study of Algorithms for Estimating Truck Factor. In *Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. 91–100.

[20] Fred W Glover and Gary A Kochenberger. 2006. *Handbook of metaheuristics*. Vol. 57. Springer Science & Business Media.

[21] Mark Harman. 2007. The current state and future of search based software engineering. (2007), 342–357.

[22] Mark Harman and John Clark. 2004. Metrics are fitness functions too. In *10th International Symposium on Software Metrics*. 58–69.

[23] Mark Harman and Bryan F Jones. 2001. Search-based software engineering. *Information and software Technology* 43, 14 (2001), 833–839.

[24] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.

[25] James D. Herbsleb and Audris Mockus. 2003. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on software engineering* 29, 6 (2003), 481–494.

[26] Katherine J Hunt, Natalie Shlomo, and Julia Addington-Hall. 2013. Participant recruitment in sensitive surveys: a comparative trial of 'opt in'versus 'opt out'approaches. *BMC Medical Research Methodology* 13, 1 (2013), 3.

[27] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. 2015. From Developer Networks to Verified Communities: A Fine-Grained Approach. In *37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 1. 563–573.

[28] M. John R. Koza. 1992. Genetic Programming: On Programming Computers by means of Natural Selection and Genetics. In *MIT Press, Cambridge, MA, 1992*. Association for Computing Machinery ,New York, NY, United States.

[29] M. Kessentini and A. Ouni. 2017. Detecting Android Smells Using Multi-Objective Genetic Programming. In *IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 122–132.

[30] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[31] Stuart McIlroy, Nasir Ali, Hammad Khalid, and Ahmed E Hassan. 2016. Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews. *Empirical Software Engineering* 21, 3 (2016), 1067–1106.

[32] Andrew Meneely and Laurie A. Williams. 2011. Socio-technical developer networks: should we trust our measurements? *2011 33rd International Conference on Software Engineering (ICSE)* (2011), 281–290.

[33] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 521–530.

[34] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. 2008. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proceedings of the 30th International Conference on Software Engineering*. 521–530.

[35] Martin Nordio, H Christian Estler, Bertrand Meyer, Julian Tschannen, Carlo Ghezzi, and Elisabetta Di Nitto. 2011. How do distribution and time zones affect software development? a case study on communication. In *2011 IEEE Sixth International Conference on Global Software Engineering*. IEEE, 176–184.

[36] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. 2015. Web service antipatterns detection using genetic programming. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)*. 1351–1358.

[37] A. Ouni, M. Kessentini, K. Inoue, and M. Ó. Cinnéide. 2017. Search-Based Web Service Antipatterns Detection. *IEEE Transactions on Services Computing* 10, 4 (July 2017), 603–617.

[38] Ali Ouni, Marouane Kessentini, and Houari Sahraoui. 2013. Search-based refactoring using recorded code changes. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 221–230.

[39] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. 2013. Maintainability defects detection and correction: a multi-objective approach. *Automated Software Engineering* 20, 1 (2013), 47–79.

[40] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2012. Search-based refactoring: Towards semantics preservation. In *28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 347–356.

[41] Ali Ouni, Marouane Kessentini, Houari Sahraoui, and Mohamed Salah Hamdi. 2013. The use of development history in software refactoring using a multi-objective evolutionary algorithm. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 1461–1468.

[42] Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. 2016. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 3 (2016), 1–53.

[43] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based peer reviewers recommendation in modern code review. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 367–377.

[44] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M German, and Katsuro Inoue. 2017. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology* 83 (2017), 55–75.

[45] Fabio Palomba, Damian Andrew Andrew Tamburri, Francesca Arcelli Fontana, Rocco Oliveto, Andy Zaidman, and Alexander Serebrenik. 2018. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE transactions on software engineering* (2018).

[46] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. 2008. Can Developer-module Networks Predict Failures?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2–12.

[47] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. 2003. Automated support for classifying software failure reports. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 465–475.

[48] Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. 2011. Classifier chains for multi-label classification. *Machine learning* 85, 3 (2011), 333.

[49] Motoshi Saeki. 1995. Communication, collaboration and cooperation in software development-how should we support group work in software development?. In *Proceedings 1995 Asia Pacific Software Engineering Conference*. IEEE, 12–20.

[50] WIlliam Sugar. 2014. *Studies of ID practices: A review and synthesis of research on ID current practices*. Springer.

[51] Damian A Tamburri, Rick Kazman, and Hamed Fahimi. 2016. The architect's role in community shepherding. *IEEE Software* 33, 6 (2016), 70–79.

[52] Damian A Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. 2013. What is social debt in software engineering?. In *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 93–96.

[53] D. A. Tamburri, P. Kruchten, P. Lago, and H. van Vliet. 2013. What is social debt in software engineering?. In *International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. 93–96.

[54] Damian A. Tamburri, Philippe Kruchten, Patricia Lago, and Hans van Vliet. 2015. Social debt in software engineering: insights from industry. *Journal of Internet Services and Applications* 6, 1 (04 May 2015), 10.

[55] Damian A. Tamburri, Fabio Palomba, Alexander Serebrenik, and Andy Zaidman. 2018. Discovering community patterns in open-source: a systematic approach and its evaluation. *Empirical Software Engineering* (2018).

[56] D. A. A. Tamburri, F. Palomba, and R. Kazman. 2019. Exploring Community Smells in Open-Source: An Automated Approach. *IEEE Transactions on Software Engineering* (2019), 1–1.

[57] Grigorios Tsoumakas and Ioannis Katakis. 2007. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining* 3, 3 (2007), 1–13.

[58] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. 2010. *Mining Multi-label Data*. 667–685.

[59] Grigorios Tsoumakas, Ioannis Katakis, and Ioannis Vlahavas. 2010. Random k-labelsets for multilabel classification. *IEEE Transactions on Knowledge and Data Engineering* 23, 7 (2010), 1079–1089.

[60] Xin Xia, Yang Feng, David Lo, Zhenyu Chen, and Xinyu Wang. 2014. Towards more accurate multi-label software behavior learning. In *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 134–143.

[61] Min-Ling Zhang and Zhi-Hua Zhou. 2007. ML-KNN: A lazy learning approach to multi-label learning. *Pattern recognition* 40, 7 (2007), 2038–2048.