Towards the Repayment of Self-Admitted Technical Debt

Giancarlo Sierra

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

January 2019

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Giancarlo Sierra**

Entitled: **Towards the Repayment of Self-Admitted Technical Debt**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Denis Pankratov

_____ Examiner
Dr. Yann-Gaël Guéhéneuc

_____ Examiner
Dr. Tse-Hsun Chen

_____ Supervisor
Dr. Emad Shihab

Approved by _____
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

_____ 2019

_____
Amir Asif, Dean
Faculty of Engineering and Computer Science

Abstract

Towards the Repayment of Self-Admitted Technical Debt

Giancarlo Sierra

Technical Debt is a metaphor used to express sub-optimal source code implementations that are introduced for short-term benefits that often must be paid back later, at an increased cost. In recent years, various empirical studies have focused on investigating source code comments that indicate Technical Debt, often referred to as Self-Admitted Technical Debt (SATD).

In this thesis, we survey research work on SATD, analyzing characteristics of current approaches and techniques for SATD, dividing literature in three categories: detection, comprehension, and repayment. To set the stage for novel and improved work on SATD, we compile tools, resources, and data sets made publicly available. We also identify areas that are missing investigation, open challenges, and discuss potential future research avenues. From the literature survey, we conclude that most findings and contributions have focused on techniques to identify, classify, and comprehend SATD. Few studies focused on the repayment or management of SATD, which is an essential goal of studying technical debt for software maintenance.

Therefore, we perform an empirical study towards SATD repayment. We conducted a preliminary online survey with developers to understand the elements they consider to prioritize SATD. With the acquired knowledge from the survey responses and previous literature work, we select metrics to estimate SATD repayment effort. We examine SATD instances found in software systems to see how it has been repaid and investigate the possibility of using historical data at the time of SATD introduction as indicators for SATD that should be addressed. We find two SATD repayment effort metrics that can be consistently modeled in our studied projects and surface the best early indicators for important SATD.

Related Publications

The following publication is related to the materials presented in this thesis:

- **G. Sierra**, E. Shihab, Y. Kamei, A Survey of Self-Admitted Technical Debt, In Journal of Systems and Software (JSS), 2018. [Major Revision]

The following publications are not directly related to the material presented in this thesis, but were conducted as parallel work to the research presented in this thesis.

- **G. Sierra**, A. Tahmid, E. Shihab, N. Tsantalis. Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?, In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2019.

- S. Mujahid, **G. Sierra**, R. Abdalkareem, E. Shihab, W. Shang, An Empirical Study of Android Wear User Complaints, In Empirical Software Engineering Journal (EMSE), 2018.

- S. Mujahid, **G. Sierra**, R. Abdalkareem, E. Shihab, W. Shang, Examining User Complaints of Wearable Apps: A Case Study on Android Wear, In Proceedings of 4th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), 2017.

Statement of Originality

I, Giancarlo Sierra, hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Dedications

To my mother Alexandra, and in loving memory of my grandmother Alicia.

Acknowledgments

First, I would like to express my greatest gratitude to my mother Alexandra, who is the role model and inspiration for pursuing my academic goals. You supported all my years of education with remarkable dedication and encouraged me to pursue my dreams and happiness throughout the way. You were there every time I hesitated to take the next step, and guided me to success. I certainly owe it all to you.

At the culmination of my masters degree, I am truly grateful to my supervisor Dr. Emad Shihab, without whom nothing of this would have been possible. Thank you for giving me the opportunity to tackle this challenge, for believing in me, and for enlightening me with all your knowledge and wisdom. Your drive and dedication is something I admire; you definitely pushed me to learn beyond expectation.

I extend my gratitude to the professors with whom I worked the closest for my degree and research, Dr. Yasutaka Kamei, Dr. Weiyi Shang, and Dr. Nikolaos Tsantalis. It was a delight to follow your teachings. In the same manner, I would like to thank my thesis examiners, Dr.Yann-Gaël Guéhéneuc and Dr. Tse-Hsun Chen for taking the time to review my thesis and for giving me valuable feedback on it.

An immense thank you to all the members of the Data-driven Analysis of Software (DAS) Lab, and our peer Software Engineering research labs. Suhaib, Rabe, Jinfu, Kundi, Mehran, Moiz, Everton, Ahmad, Mohamed, Xiaowei, Joy, Sara, Gui, Tahmid, Max, Hosein, Mahmood, Mouafak,

Atique, Olivier, and Juan. I am glad to not only call you my colleagues, but my friends. I learned a great deal from all of you. Working alongside you was not only a pleasure, but an honor. I wish you all success in the different paths you have chosen.

While pursuing this masters degree, I received the constant support of my closest friends outside academy. Daniel, Adriana, Diego, Audrey, Avery, Estefanía, Daniela, and Laura; thank you for always being there. Sometimes the simple things in life are what matter the most.

Last but not least, I would like to thank my family for their unconditional support. A especial thank you to my father Teodoro, my grandfather Rafael, and my sister Melissa. Despite the distance that sets us apart, I never felt alone in this journey with you by my side. You are the light that keeps me going.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

In Software Engineering, Technical Debt (TD) is a metaphor used to express sub-optimal source code implementations introduced for short-term benefits which often must be paid back later, at an increased cost. Cunningham (1992) first used this metaphor to explain that developers do not always contribute optimal code. Whether the introduction of such code is conscious or subconscious, it is a form of debt that has to be paid back later. A plethora of studies have studied TD in recent years (N. S. Alves et al., 2016; Li, Avgeriou, & Liang, 2015). Empirical studies focused on investigating source code comments that indicate Technical Debt often referred to as Self-Admitted Technical Debt (SATD). This term was first coined by Potdar and Shihab (2014) who explored the phenomenon for the first time. Recent work improved approaches that began by manual identification and classification of comments that denote SATD to more automated approaches that use machine learning to achieve increased performance (Huang, Shihab, Xia, Lo, & Li, 2018; Maldonado, Shihab, & Tsantalis, 2017b). SATD literature studied the circumstances under which SATD is introduced and removed from source code (Bavota & Russo, 2016; Maldonado, Abdalkareem, Shihab, & Serebrenik, 2017a; Zampetti, Serebrenik, & Di Penta, 2018), and even implications for software maintenance (Wehaibi, Shihab, & Guerrouj, 2016). Certainly, these studies shed light on the importance of SATD, bringing awareness to developers and allowing them to locate debt in their source code with relative ease.

One of the goals for studying the SATD phenomenon is to have awareness of the presence of debt instances, and to manage them appropriately or to fully remove them before the debt becomes too costly to repay. Not all SATD has to be repaid right away; it might be the case that SATD instances are trivial, non-critical or that they can simply be present in a system without causing further complications over time. Nevertheless, the contrary holds true as well; some SATD instances could passively increase maintenance efforts when the debt is not caught and handled in time. Studies proposed ways to quantify the increased difficulty to repay SATD (Kamei, Maldonado, Shihab, & Ubayashi, 2016), or estimate the effort it needs to be repaid (Mensah, Keung, Bosu, & Bennin, 2016; Mensah, Keung, Svajlenko, Bennin, & Mi, 2018). However, current research is yet to provide answers to what SATD should be repaid or not. Moreover, in cases where repayment of debt is needed, the question of which instance should be resolved first in a system and how remains unresolved.

We conducted an in-depth literature survey of recent work on SATD. We compiled and categorized SATD studies in 3 areas: detection, comprehension and repayment. Our findings from this survey point at gaps in SATD research and challenges to overcome to advance the state of the art. Surveying past work, we propose potential research tracks for work in SATD and compile the tools, approaches and datasets that are available to the research community to extend work in the area. The majority of SATD work has specialized on detecting, classifying or comprehending the SATD phenomenon. To this date, very few studies have focused on approaches or techniques to manage or resolve SATD, which is of critical importance.

Therefore, motivated by our findings from the literature survey, with the knowledge acquired from it, and a set of compiled tools and techniques at our disposition, we centralize our efforts and work towards the repayment of SATD. We begin by conducting an online survey with developers to understand the factors that they consider when deciding which SATD instance to resolve first, i.e., prioritizing SATD repayment. We combined the responses and insight from developers with previous work and our conjectures to determine a set of metrics that serve for SATD repayment effort. We performed an empirical study on more than 18,000 unique SATD instances (and their change history) detected in 8 open-source software systems. Our work investigates the possibility of using historical data taken at the time of SATD introduction to find indicators of SATD that

should be repaid. Our findings show that two metrics for SATD repayment effort can be modeled consistently in the studied projects, and that the best indicators of important SATD measure features intrinsic to the SATD file, and not extrinsic historical features such as the churn or entropy of a change that introduced SATD.

Below, we brief of the organization of this thesis by chapters and highlight the main contributions presented throughout this work.

1.2 Thesis Overview

The main content of this thesis is condensed in the following two chapters:

1.2.1 Chapter 2: Background and Literature Survey of Self-Admitted Technical Debt.

This chapter presents our literature survey on SATD. We begin by establishing our scope, definitions and approach to select papers related to SATD. Then, we proceed to list and compare the main findings and contributions of the surveyed work, organized in the categories of: detection, comprehension or repayment of SATD. To motivate further research in the area, we include a section detailing the opportunities to be tackled by the research community in future work, with explicit calls to action. We conclude Chapter 2 by mentioning the limitations of our literature review and recapping our findings from it.

1.2.2 Chapter 3: Towards Self-Admitted Technical Debt Repayment.

With the knowledge gained from the literature review in Chapter 2 and motivated to advance SATD research, we centralize our efforts towards the repayment of SATD. In Chapter 3, we review the online survey sent to developers to understand how they prioritize SATD in their projects. Then, we overview the approach for our empirical study and detail the step by step collection of data required for our study. We proceed to present the results of our study with linear regression models built for SATD repayment effort metrics. Lastly, we find indicators from historical data for important SATD that should be repaid.

1.3 Thesis Contributions

The main contributions of this thesis are the following:

- An in-depth literature survey on Self-Admitted Technical Debt research work. This thesis presents a compilation of the most recent publications in the area of SATD, highlighting gaps and opportunities as potential research avenues for future work. We overview challenges in the area and include a set of explicit calls to action that can be implemented by researchers to overcome them.

- A compilation of publicly available tools, approaches, techniques and online datasets that from surveyed work that can be used to extend SATD research.

- An empirical study of the change history of 18,242 unique SATD instances detected with state of the art approaches in 8 software systems investigating the possibility of using change history and defect prediction metrics as early indicators of SATD that should be addressed.

- We propose the use of a compound interest rate to measure the evolution of a SATD that allows developers to estimate the frequency and speed at which a debt instance gains interest.

- Our resulting dataset from the empirical study, as well as the questions and responses of the online survey to developers are made publicly available to facilitate and promote further research in the area o SATD. The dataset contains a wide set of product and process metrics of detected SATD instances (in all their versions) from eight open-source projects.

Chapter 2

Background and Literature Survey of Self-Admitted Technical Debt

2.1 Introduction

As software undergoes its development and maintenance, developers cannot not always contribute code as required by specification. In 1992, Cunningham first introduced the metaphor of considering the "not-quite-right code" as a form of debt (Cunningham, 1992), which came to be know as the *Technical Debt* (TD) metaphor. It explains the concept of delivering a solution that is not complete, temporary or sub-optimal; thus incurring in debt to obtain short-term benefits that have to be paid over the long-term with an increased cost. Developers have different reasons that can lead them to introduce technical debt, such as deadline pressure, existing low quality code, bad software process, or business reality (Lim, Taksande, & Seaman, 2012). Technical Debt can be introduced both consciously or unconsciously. Developers tend to underestimate the consequences of repaying the debt, possibly leading to ever-growing problems (Bellomo, Nord, Ozkaya, & Popeck, 2016). Because of its clear importance to the software process and quality, an abundant amount of research has investigated TD (N. S. Alves et al., 2016; Li et al., 2015). While in the past most studies focused on detecting and managing debt found in source code, the research scope has gradually grown to include additional software artifacts, e.g., documentation or requirements (N. S. R. Alves, Ribeiro, Caires, Mendes, & Spínola, 2014; Ernst, Bellomo, Ozkaya, Nord, & Gorton, 2015).

In 2014, Potdar and Shihab (2014) took a new research direction by conducting an exploratory study on source code comments that point to debt instances. The authors first referred to this phenomenon as *Self-Admitted Technical Debt* **(SATD)**. Their rationale being that when developers consciously introduce debt (i.e., code that is either incomplete, defective, temporary, or simply sub-optimal) and acknowledge so in the form of comments they *self-admit* it. Brief examples of these comments are: *"TODO: - This method is too complex, lets break it up"* from ArgoUml, and *"Hack to allow entire URL to be provided in host field"* from JMeter (Maldonado & Shihab, 2015; Maldonado, Shihab, & Tsantalis, 2017b).

Potdar and Shihab (2014) extracted a large set of source code comments from 4 large open-source systems and manually analyzed them to point at SATD instances. As found by their investigation, this phenomenon occurs commonly in software systems. Since then, a number of studies focusing on various aspects of SATD have emerged, exploring and improving approaches and techniques to better identify, understand and manage SATD. The recent and increasing turn out of empirical work in this branch of TD denotes the importance given to it by the Software Engineering community. Taking into consideration that this research track is fairly recent, the early efforts of current studies on SATD remain scattered in focus and face various challenges to overcome. We believe that it is the right time to reflect on recent accomplishments in the area and examine open problems to pave the path for future work.

Therefore, this chapter presents a survey of SATD studies from recent years, i.e., since the original ICSME paper that proposed SATD in 2014. Through our examination of the published papers, we find that the vast majority of SATD research work can be divided into three categories: work focusing on the **detection** of SATD, work that aims to improve the **comprehension** of SATD, and work focusing on the **repayment** of SATD. Hence, we structure our survey to reflect these 3 main categories. Specifically, our study provides an overview of past and current works on the detection, comprehension and repayment of SATD. Moreover, to support and promote further research in the domain, we identify potential future avenues for SATD research and discuss current challenges. Throughout this survey we also point at available resources such as tools and datasets that can serve as foundations or baselines for new SATD studies. A table with the published artifacts and online references from the surveyed work can be found in Table A.1 of Appendix A.

The remainder of this chapter is organized as follows: Section 2.2 describes the objectives, scope and literature selection for the survey; Section 2.3 analyses and compares the findings and contributions of current SATD studies; Section 2.4 goes over the possible future research avenues in this area and its challenges. Lastly, Section 2.5 presents the conclusions and limitations of the survey.

2.2 Preliminaries

This section details the scope and selection of studies for our survey. We also provide definitions for the terms we use throughout this chapter. Finally, we present a high-level overview of the SATD literature published to date.

2.2.1 Scope and Paper Selection

The focus of this survey is Self-Admitted Technical Debt as a sub-domain of Technical Debt. We clarify that work focusing entirely on Technical Debt (and not SATD specifically) is not in scope and refer our readers to recent literature that focused on that area, e.g., N. S. Alves et al. (2016); Li et al. (2015). To select the papers included in this survey we used both, the references from known SATD research, and academic work available online through popular search engines, namely: Google Scholar, ACM, and IEEE. We chose the exploratory study by Potdar and Shihab (2014) as the basis for this survey because it is the first to investigate the SATD phenomenon and remains as the most cited work in the area. Hence our survey encloses work published since its release year (2014) until the compilation date of this survey (July 2018). We searched for all the papers that cited (Potdar & Shihab, 2014) in the aforementioned online search engines using the keywords "SATD" and "Self-admitted Technical Debt", limiting the results to papers released since 2014. A complete list of the initial studies that we selected and did not select can be found in Table A.2 of Appendix A.

Once we identified a paper related to SATD, we applied a snowball approach to find other relevant cited work (Wohlin, 2014). We repeated this procedure for each work that cited Potdar and Shihab (2014), however, we did not find any other (new) SATD related papers that were not already

included in the initial list or found by the search engines. Given that SATD is fairly new and due to the amount of mainstream work in the area, we do not perform a systematic literature study; we leave that for the near future when the amount of SATD-related work justifies such kind of survey.

2.2.2 Definitions

We classified the surveyed papers into 3 main categories tied to the life cycle stages of SATD, i.e., the sequence of phases that an instance of SATD goes through, from its introduction, to its evolution, and lastly its removal from a software system. Hence, the work is aligned along: detection, comprehension, and repayment of SATD. We elaborate on what studies fall under each category below:

- **Detection** studies: those that focus on proposing, studying or improving: approaches, techniques, and tools to identify or detect instances of SATD.
- **Comprehension** studies: those that investigate the phenomenon of SATD itself and are dedicated to understand the life cycle of SATD. These studies encompass topics such as: introduction, diffusion, evolution, removal of SATD, or its relation with different aspects of the software process.
- **Repayment** studies: those that propose, validate, or replicate: approaches, techniques, and tools that seek to remove (i.e., fully repay) or mitigate (i.e., partially repay) SATD instances.

2.2.3 Overview of Selected Papers

Given the scope and definitions above, Table 2.1 presents a chronologically-ordered overview of the primary SATD studies. Those marked with a star (*) are studies which focus in not dedicated to SATD, however, a relevant portion of them addresses to SATD and presents findings related to its comprehension or detection, so we consider them within the primary group. Although related work without a direct contribution or finding on SATD is not considered within the selected group of papers, we mention and reference such work throughout this survey because they support the papers we selected or serve as links to potential future avenues in this area. In Table 2.1 we observe that

50% of the primary SATD papers focus on comprehension, 55% on detection, while only 10% focus on repayment. Note that 3 studies are classified as having 2 topics of focus, hence these percentages overlap. Regarding the paper's publication avenues, 60% of them are published in conferences, 20% in journals, and another 20% were presented in workshops.

Table 2.1: Overview of primary SATD studies.

Reference	Title	Venue	Venue Type	Focus
Potdar and Shihab (2014)	An Exploratory Study on Self-Admitted Technical Debt.	ICSME	Conference	Comprehension, detection
Maldonado and Shihab (2015)	Detecting and Quantifying Different Types of Self-Admitted Technical Debt.	MTD	Workshop	Comprehension, detection
Freitas Farias, de Mendonça Neto, da Silva, and Spínola (2015a)	A Contextualized Vocabulary Model for Identifying Technical Debt on Code Comments.	MTD	Workshop	Detection
Wehaibi et al. (2016)	Examining the Impact of Self-admitted Technical Debt on Software Quality.	SANER	Conference	Comprehension
Freitas Farias, Santos, Kalinowski, Mendonça, and Spínola (2016c)	Investigating the Identification of Technical Debt Through Code Comment Analysis.	ICEIS	Conference	Detection
Bavota and Russo (2016)	A Large-Scale Empirical Study on Self-Admitted Technical Debt.	MSR	Conference	Comprehension
Vassallo et al. (2016)	Continuous Delivery Practices in a Large Financial Organization.	ICSME	Conference	Comprehension*
Kamei et al. (2016)	Using Analytics to Quantify the Interest of Self-Admitted Technical Debt.	TDA	Workshop	Comprehension
Mensah et al. (2016)	Rework Effort Estimation of Self-Admitted Technical Debt.	TDA	Workshop	Repayment, detection
Ichinose et al. (2016)	ROCAT on KATARIBE: Code Visualization for Communities.	ACIT	Conference	Detection*
Maldonado, Shihab, and Tsantalis (2017b)	Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt.	TSE	Journal	Detection
Palomba, Zaidman, Oliveto, and De Lucia (2017)	An Exploratory Study on the Relationship between Changes and Refactoring.	ICPC	Conference	Comprehension*
Miyake, Amasaki, Aman, and Yokogawa (2017)	A Replicated Study on Relationship Between Code Quality and Method Comments.	ACIT	Conference	Comprehension*
Maldonado, Abdalkareem, et al. (2017a)	An Empirical Study on the Removal of Self-Admitted Technical Debt.	ICSME	Conference	Comprehension
Zampetti, Noiseux, Antoniol, Khomh, and Di Penta (2017)	Recommending when Design Technical Debt Should be Self-Admitted.	ICSME	Conference	Detection

Mensah et al. (2018)	On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt.	JSS	Journal	Repayment
Huang et al. (2018)	Identifying Self-Admitted Technical Debt in Open Source Projects using Text-Mining.	EMSE	Journal	Detection
Liu et al. (2018)	SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool.	ICSE	Conference	Detection
Zampetti et al. (2018)	Was Self-Admitted Technical Debt Removal a real Removal? An In-Depth Perspective.	MSR	Conference	Comprehension
Yan et al. (2018)	Automating Change-level Self-admitted Technical Debt Determination.	TSE	Journal	Detection

2.3 Analysis and Comparison of Current SATD Work

In this section we first go over the techniques, tools, and approaches presented by current research work in SATD. We first present work that focused on identifying instances of debt, then we present empirical studies that have studied the phenomenon to understand it, and finally contributions that aim to manage and repay it. A list of the software systems studied by the surveyed work along with how each study validates TD is available in Table A.3 of Appendix A.

2.3.1 Detection of SATD

In the life cycle of SATD, debt instances are first introduced by developers into the source code; thus naturally, the first step to study this phenomenon is to identify it. In the past, several studies focused on source code comments, their management, and co-evolution with code; while others focused on the identification and management of Technical Debt (N. S. Alves et al., 2016; Fluri, Wursch, & Gall, 2007a; Li et al., 2015; Storey, Ryall, Bull, Myers, & Singer, 2008; Tan, Yuan, Krishna, & Zhou, 2007). However, these studies did not investigate or relate the presence of technical debt within the content of comments. Inspired by such previous work, Potdar and Shihab (2014) were the first to look at source code comments to identify technical debt, and introduced the term of *Self-Admitted Technical Debt*, referring to code that is either incomplete, defective or temporary, and that is knowingly introduced by developers. Seven different approaches to detect SATD have appeared in literature since; 6 of them identify SATD at the file level looking at the revision history of a repository, while 1 approach aims to detect SATD at the change level. In this subsection, we present the 6 approaches that work at the file level divided in two groups: i) those approaches that are based on the identification of textual patterns in comments, which we name "pattern-based approaches"; and ii) those based on more advanced and automated techniques, such as machine learning classifiers or natural language processing, which we name "machine learning approaches". Lastly, we present the only approach that focuses on detecting SATD at the change level, and a comparison between the surveyed approaches.

Pattern-based Approaches

As a first step in SATD identification at the file level, Potdar and Shihab (2014) extracted 101,762 source code comments from 4 large open-source systems using the *srcML* toolkit (Collard, Decker, & Maletic, 2011), and manually read through them to expose patterns that indicate SATD. In total, the authors identified 62 patterns and made them publicly available to enable further research (Potdar & Shihab, 2015b). Some examples of the identified patterns are: *hack, fixme, is problematic, this isn't very solid, probably a bug, hope everything will work, fix this crap*. Using these patterns, their study found that SATD can exist in up to 31% of files of a system; a finding that triggered further research in this domain.

For the remaining of this work, we will refer to the usage of these 62 patterns as the **pattern-based detection** approach. This approach allows for an easier SATD identification than simple manual inspection of comments, which is time-consuming and requires expertise. However, because these patterns resulted from analyzing 4 projects only, they may not generalize to detect SATD in other software systems, compromising the accuracy of the approach. Additionally, in case the set of patterns has to be extended, additional effort must be spent manually inspecting source code comments from different projects and surfacing new patterns that can be used for detecting TD in comments.

Following up to the previous findings, Maldonado and Shihab (2015) manually inspected the comments of another 5 open-source systems, this time however, with a motivation to explore the different types of SATD contained in them. They found 5 main types of SATD: design, defect, documentation, requirement and test debt (See 2.3.2). Instead of *srcML*, the tool *JDeodorant* was used to parse the extracted comments (Tsantalis, Chaikalis, & Chatzigeorgiou, 2008). Four filtering heuristics were introduced to remove irrelevant comments, which are: a) removing license comments; b) aggregating consecutive single-line comments; c) removing commented source code; and d) removing Javadoc comments. To ensure these heuristics do not filter out SATD instances, comments containing task-reserved words ("todo", "fixme", or "xxx") were not removed. The implementation of these heuristics proved to reduce the amount of comments to analyze manually by 77% on average, easing detection efforts. To contribute with the identification of specific types of SATD, the output dataset of classified comments by types was made publicly available to the

community (Potdar & Shihab, 2015a).

Motivated to facilitate the detection of SATD using the pattern-based approach, Ichinose et al. (2016) extended their code visualization tool *ROCAT*, which renders the source code of a system as city-like virtual reality environments, to support SATD. With this visualization model, buildings are constructed for each source file, their dimensions are based on software product metrics, and SATD instances are rendered as buildings of different sizes and colors based on comments that contain the patterns surfaced by Potdar and Shihab (2014). This visualization provides developers with a high-level view of a system's source code that includes visual cues of SATD instances, removing the need of reading comments to visualize where SATD occurs in their source code. *ROCAT* was integrated with *Kataribe* (Fujiwara et al., 2014), a Git hosting service; thus, any project registered on *Kataribe* can benefit from *ROCAT's* visualization capabilities.

An alternative and extension to the pattern-based detection approach was later proposed by Freitas Farias et al. (2015a), who introduced **CVM-TD**, a Contextualized Vocabulary Model for Identifying TD of different types in source code comments. This model relies on identifying word classes, namely: nouns, verbs, adverbs, and adjectives that are related to Software Engineering terms and code tags used by developers such as "TODO" (Freitas Farias, de Mendonça Neto, da Silva, & Spínola, 2015d). The goal of applying the CVM-TD model, which can be automated, is to obtain a subset of comments that will likely contain SATD. The proposed vocabulary (Freitas Farias, de Mendonça Neto, da Silva, & Spínola, 2015b) focuses on words that can be systematically related to each other and then mapped to different types of TD as defined by N. S. R. Alves et al. (2014). To validate CVM-TD, an empirical study was conducted on Apache Lucene and JEdit, from which comments were extracted using *eXcomment* (Freitas Farias, de Mendonça Neto, da Silva, & Spínola, 2015c), a tool that uses an Abstract Syntax Tree to store useful comment-related information and to filter them with heuristics similar to the ones proposed by Maldonado and Shihab (2015). The empirical evaluation of the model showed a considerable difference in the comments returned by the model and the ones validated to contain SATD. This finding suggested a low detection performance and pointed at the need to enhance how the word classes are mapped to different types of SATD to improve the model.

Later in 2016, Freitas Farias et al. (2016c) conducted an additional experiment on CVM-TD to

characterize its overall accuracy and the factors that influence its detection. This time, the CVM-TD model was applied to ArgoUML; the output comments were given to 3 researchers with expertise in TD to create an oracle of comments indicating TD. The same output was also given to 32 Software Engineers with varied experience and different English reading levels to flag those suggesting TD. The experiment found that English reading skills of participants influenced their identification of TD, but their classifying capability was not affected by their experience. Based on the TD oracle, the CVM-TD model's output served experienced and non-experienced developers alike, allowing them to have an accuracy on average of 0.673 when detecting TD comments; a better performance than previously reported (Freitas Farias et al., 2015a). The experiment also requested participants to highlight the patterns that induced marking a comment as TD, which surfaced common patterns and TD indicating comments to extend the vocabulary of CVM-TD (Freitas Farias, Santos, Kalinowski, Mendonça, & Spínola, 2016a, 2016b). Note that in both empirical studies, i.e., (Freitas Farias et al., 2015a, 2016c), the authors did not explicitly refer to source code comments that aid in the detection of TD as SATD, nevertheless, we consider both studies within scope as they study this same precise phenomenon.

Mensah et al. (2016) proposed the use of text-mining in SATD detection. Their approach aimed to estimate the effort needed to resolve SATD (See 2.3.3) and was composed of 5 phases. The first 3 phases of the approach are aimed at the extraction, detection and classification of SATD; which is built on top of a pattern-based approach and a dictionary from the dataset of comments classified into different SATD types published by Maldonado and Shihab (2015). We will refer to this approach as **Text-mining**. Improving from the pattern-based approach, this one first preprocesses comments to remove special punctuation characters and stop words. However, this introduces a drawback. Removing punctuation characters such as *!* or *?* can potentially take away semantic meaning from comments, i.e., the removal of a simple question mark could alter the meaning or intention of a developer's comment. Moreover, no filters such as the heuristics proposed and used previously, e.g., (Freitas Farias et al., 2016c; Maldonado & Shihab, 2015) were applied to reduce preprocessing.

Machine Learning Approaches

Moving towards more advanced SATD detection approaches at the file level, Maldonado, Shihab, and Tsantalis (2017b) used NLP techniques to automatically identify design and requirement SATD from source code comments. We will refer to this approach as **NLP detection**. The authors extracted, filtered, and manually classified a dataset of 62,566 comments from 10 open-source projects into 5 different types of SATD: design, test, defect, documentation, and requirement debt. This dataset combined 29,473 comments extracted from 5 open-source projects, and 33,093 others extracted from 5 additional projects in previous work (Maldonado & Shihab, 2015). With it, the authors trained an NLP maximum entropy classifier (Stanford Classifier) focusing on requirement and design SATD, as they are the most recurrent debt types, making up more than 90% of the SATD comments (Maldonado & Shihab, 2015). The NLP classifier generates a set of feature words that contribute positively or negatively to the classification of a comment. A 10 fold cross-project validation training on 9 projects and testing on the remaining showed that the NLP detection achieved an accuracy that surpassed the previous pattern-based detection. For design debt, the classifier scored an average F1-measure of 0.620, 0.403 for requirement debt, and 0.636 disregarding debt types. The study also presented top-10 lists of textual features that can be directly used to identify SATD in approaches that do not rely on NLP techniques. These features were found to differ among one another, indicating that developers use distinct vocabularies to admit different kinds of SATD.

Training an NLP classifier can be expensive because it relies on a manual classification of comments. However, Maldonado, Shihab, and Tsantalis (2017b) showed that, to achieve 90% of the classifiers performance, approximately 23% of the SATD comments were needed for training, which eases the replication of this approach. To enable further research on SATD, the full resulting dataset of manually classified comments and their resulting NLP classification was made publicly available (Maldonado, Shihab, & Tsantalis, 2017a).

The most recent SATD detection technique was presented in 2017 by Huang et al. (2018), who proposed an approach to automatically detect SATD using text-mining and a composite classifier. We will refer to this as the **Ensemble text-mining** approach. Its root concept is to determine if a comment indicates SATD or not (without focusing on SATD types) based on training comments

from different software projects. For this, the authors leveraged a dataset of 212,413 comments classified by Maldonado, Shihab, and Tsantalis (2017b) from 8 open-source projects. This approach preprocesses comments by tokenizing, removing stop-words and stemming their descriptions to obtain textual features. Feature selection (Information Gain) is then applied to detect the top 10% most useful features to predict the label of a comment, indicating if it contains SATD or not. Multiple sub-classifiers are trained with a Naive Bayes Multinomial (NBM) technique to determine the label of a comment based on their number of contributing features. A composite classifier takes the vote per comment of each sub-classifier to reach a final classification. Several aspects of the ensemble text-mining performance were evaluated in terms of F1-score. The approach was benchmarked against the pattern-based and NLP detection of SATD, finding that it performed better than both, had a superior runtime performance, and also required a small portion of comments for training.

The ensemble text-mining approach was implemented very recently by Liu et al. (2018) as an Eclipse plugin named *SATD Detector* to facilitate the detection and management of debt instances directly from an IDE environment. *SATD Detector* parses the source code of a system when it is loaded or edited and applies the ensemble text-mining approach to detect and report SATD instances along with their respective locations. This plugin completely automates the detection of SATD with a built-in classifier that can be used out of the box to leverage the best-performing SATD detection technique to this date.

From a different SATD detection perspective, Zampetti, Noiseux, et al. (2017) proposed **TEDIOuS** (Technical Debt Identification System), a machine learning approach that recommends to developers when they should self-admit design TD. Instead of analyzing comments, the idea is to leverage source code level features. When a developer adds new code, the approach can analyze it and recommend if it should be flagged (i.e., to be self-admitted as debt) or not. TEDIOuS' identification capabilities relies on readability and structural metrics extracted with a srcML-based tool, and the warnings raised by PMD and CheckStyle, two static analysis tools.

TEDIOuS was evaluated using the classified comments of 9 projects from the dataset made available by Maldonado, Shihab, and Tsantalis (2017b). Since these comments were detected at the file level, a matching of comments to the method level was required for TEDIOuS features'

scope. Different classifiers were tested with balanced and unbalanced training data using cross validation within a project and across all studied projects. TEDIOuS achieved it best performance using a Random Forest classifier, with a cross-project prediction precision of 67%, 55% recall, and an accuracy of 92%. The features related to readability and structural metrics used by TEDIOuS had a major contribution in recommending design SATD. When compared against DECOR (Moha, Guéhéneuc, Duchien, & Meur, 2010), a smell detector tool that leverages different code features, the SATD recommending performance of TEDIOuS proved to be superior.

Change-level Detection

All previous SATD detection studies aimed to identify debt instances at the file level. Yan et al. (2018) proposed a novel approach to automate the detection of SATD at the change level. The idea is to catch the introduction of SATD when a software change occurs, instead of inspecting if a file that was changed previously contains SATD. The authors built a determination model using a Random Forest classification with data labeled from comment analysis, and features extracted from source code repositories. The data labeling leverages an enhanced version of the dataset made available by Maldonado, Abdalkareem, et al. (2017a); it contains 100,011 manually classified software changes of 7 open-source projects, where each change is labeled as TD-introducing or not. Each change is considered TD-introducing when the resulting file version is the first to contain SATD. A total of 25 change features were extracted from the source control repository of the studied systems to characterize each change. These features were divided into 3 dimensions in the study: 16 for the *diffusion* of a change (i.e., amount of changed LOC, files, subsystems, programming languages), 3 for its *history* (i.e., information of the changed files and the developers who made the change), and 6 for its *message* (i.e., information extracted from the change logs).

The proposed model was evaluated performing a stratified 10-fold cross validation repeated 10 times for each of the 7 studied projects. This evaluation considered 2 performance measures: AUC (area under the receiver operating characteristic curve), and Cost-effectiveness, analyzed by controlling the amount of changed LOC inspected by the model. To contrast the model's performance, 4 other baseline models were studied: Random Guess, Naive Bayes, Naive Bayes Multinomial, and Random Forest (the last 3 models used a classification based on change messages

only). The study results showed that the proposed model achieves a better performance in terms of AUC (0.82) and cost-effectiveness (0.80) when compared to baseline models, being able to detect more TD-introducing changes across a wide range of changed LOC to inspect. When investigating the importance of the extracted features, the results indicated that all 3 dimensions improve significantly improve the performance of the compared models, and that the *diffusion* dimension has the most influence when determining TD-introducing changes. The performance achieved by this SATD detection approach is not contrasted with others in Table 2.2 as the SATD detection of these approaches occur in to different stages of development and thus they differ in nature. The reported performance of the change-level SATD detection is also reported in terms of AUC and not as an F1-score.

Comparison and Limitations of Current Approaches

The original pattern-based approach for SATD detection has the benefit of being simple to replicate with a fixed set of patterns to match against textual comments. However, it has the drawback of leading to up 25% of false positives, as found by Bavota and Russo (2016). Although the text-mining and CVM-TD approaches later built on top of the pattern-based approach with added heuristics, both are still affected by an underlying accuracy problem and are more complex to replicate. These early approaches lead to SATD datasets that supported the creation of more accurate and automated techniques, such as the NLP, TEDIOUS, and ensemble text-mining approaches, which implement machine learning. While TEDIOUS recommends when to self-admit technical debt, it scopes to design debt only and is not comparable with other approaches as it looks at source code instead of comments to base its recommendations. In contrast, the NLP detection and ensemble text-mining approaches focus of finding SATD in comments with good accuracy. While the NLP approach is limited to detect design and requirement debt only, the ensemble text-mining approach disregards SATD types, and thus, is a more effective all-around approach when looking for SATD in a software repository. Another benefit when compared to other detection approaches, is that TEDIOUS does not require manual inspection of comments, which aside from being time-consuming is prone to human error. Furthermore, since it was recently implemented as an IDE tool (SATD Detector plugin), it can now be used to detect SATD during or after development.

Table 2.2: Average accuracy benchmark of SATD detection approaches.

Detection Approach	**Reported F1-score**
Pattern-based	0.123
NLP	0.576
Ensemble text-mining	0.737

A performance comparison between SATD detection approaches is presented in Table 2.2 as benchmarked by Huang et al. (2018). This comparison uses the average accuracy values for detecting SATD disregarding debt types. The Text-mining and CVM-TD approaches are not included in the benchmark as their TD detection performance were not reported by (Freitas Farias et al., 2016c) and (Mensah et al., 2016). The F1-score for the NLP approach in Table 2.2 is lower than the value reported by Maldonado, Shihab, and Tsantalis (2017b) (0.636); in either case, the performance of the ensemble text-mining approach is higher.

As a recap, the studies that focused on the detection of SATD contributed with approaches that evolved from simple manual inspection of comments to complex automated approaches that identify SATD instances accurately, removing manual steps. Similarly, the text-mining approach, evolved the classification of SATD types from manual inspection to an automated tool. In Table 2.3 we overview the main findings and contributions per SATD detection study, the number of studied projects, and the technique for comment extraction, where applicable. The visualization technique presented by Ichinose et al. (2016) can be applied to multiple systems, thus no specific one is studied and no comment extraction is performed. A similar case happens with the contribution by Liu et al. (2018), which is a tool implementing the approach proposed by Huang et al. (2018). From the observations made in this section, we consider the ensemble text-mining detection approach (implemented in the SATD Detector tool) to be the most promising approach to enable future SATD research. Due to its performance and practicality, we believe this tool will promote the detection of SATD, and the compilation of richer datasets to improve the validity of SATD studies.

Table 2.3: Overview of main contributions per SATD detection study.

Reference	Main Contribution(s) / Finding(s)	Studied Systems	Comment Extraction
Potdar and Shihab (2014)	Pattern-based detection approach. SATD exists in 2.4% to 31% of files.	4	scrML-based
Maldonado and Shihab (2015)	Dataset of classified SATD comments per type. Filtering heuristics.	5	Jdeodorant
Freitas Farias et al. (2015a)	CVM-TD detection approach.	2	eXcomment
Ichinose et al. (2016)	City-like code and SATD visualization in a virtual reality environment.	N/A	N/A
Freitas Farias et al. (2016c)	Set of Patterns and comments for TD identification in comments.	1	eXcomment
Mensah et al. (2016)	Text-mining detection/classification approach.	4	*Not reported*
Maldonado, Shihab, and Tsantalis (2017b)	NLP Detection approach. Data set of classified SATD.	10	JDeodorant
Huang et al. (2018)	Ensemble text-mining detection approach.	8	NLP Dataset
Zampetti, Noiseux, et al. (2017)	TEDIOuS approach for recommending when to self-admit TD.	9	NLP Dataset
Liu et al. (2018)	Eclipse plugin to automatically detect SATD.	9	NLP Dataset
Yan et al. (2018)	Change-level SATD detection approach.	7	Relies on Maldonado, Abdalkareem, et al. (2017a)

2.3.2 Comprehension of SATD

Different studies were conducted to understand the SATD phenomenon throughout its life cycle, while others investigated its repercussion on the software process itself. A better understanding of SATD enables researchers and practitioners to develop approaches that can be used to manage it. One of the first efforts towards understanding SATD were given by Potdar and Shihab (2014); in their exploratory study they tried to understand the occurrence of SATD, why it is introduced into software systems, and how much of it is removed after its introduction. By using a pattern-based detection in 4 software projects, SATD was found to be common, happening in 2.4% to 31% of studied system's files. Regarding the introduction of SATD, Potdar and Shihab (2014) investigated how the experience of developers, time to release pressure, or the complexity of changes induced the addition of debt. Contrary to what was expected, they found that experienced developers introduced most of the SATD, while tight deadlines and change complexity did not affect its introduction. In relation to SATD removal, they found that the majority of SATD is removed in the immediate next release.

Types of SATD

Once SATD was found to be a common phenomena, Maldonado and Shihab (2015) quantified and classified the different types of SATD that exist in software projects. In a previous study, N. S. R. Alves et al. (2014) classified Technical Debt into 13 different types and proposed indicators to identify each of them. Based on these types, Maldonado and Shihab (2015) manually analyzed 33,093 comments and classified them, observing that 5 types of SATD existed in source code (design, defect, documentation, requirement, and test debt) (Potdar & Shihab, 2015a). We include brief examples of debt comments as classified by Maldonado and Shihab (2015) to help understand the detected SATD types:

- **Design debt:** *"/*TODO: really should be a separate class */"* from ArgoUml.
- **Defect debt:** *"Bug in the above method"* from Apache JMeter.
- **Requirement debt:** *"//TODO no methods yet for getClassname"* from Apache Ant.
- **Documentation debt:** *"**FIXME** This function needs documentation"* from Columba.
- **Test debt:** *"//TODO enable some proper tests!!"* from Apache JMeter.

The remaining 8 types of TD defined by N. S. R. Alves et al. (2014) were not found because they are not likely to appear in source code comments but in other artifacts. Build debt for example, would appear in build files and not in the inspected comments extracted from Java files. The quantification results of the study revealed that from over 33 thousand analyzed comments, 7.42% of them (2,457) contained SATD. Regarding the quantification per type, the majority (42% to 84%) of SATD found was design debt, followed by requirement debt, making up 5% to 45% of the debt instances. Defect, documentation, and test debt accounted for less than 10% of the classified SATD cases when combined.

Large-scale Studies

To broaden the understanding of the phenomenon, Bavota and Russo (2016) conducted a large-scale empirical study in 159 software systems (120 from the Apache ecosystem and 39 from the Eclipse ecosystem) aiming to make a differentiated replication of the initial findings by Potdar and Shihab (2014). Using the pattern-based detection, they investigated the diffusion of SATD in open-source systems and its evolution across the change history of the studied systems to see if: i) it increases or decreases over time, ii) how long it remains in the system, iii) how frequently it is fixed, and iv) who introduces or fixes SATD.

A closer look at a statistically significant sample of SATD instances revealed that, in contrast with previous findings by Maldonado and Shihab (2015), code debt was the most occurring debt type making up 30% of the cases, against a lower 13% for design debt. Furthermore, this inspection showed that over 25% of the comments flagged by the pattern-based detection were false positives. Bavota and Russo (2016) looked at the introduced, removed and unaddressed SATD comments in the projects' change history and observed that it increases over time because of debt instances being added but not addressed. Although 57% of SATD was found to be removed from source code, it has a long survivability, lasting for more than 1,000 commits on average before being fixed. Inspecting the removed SATD showed that 63% of the time, the developer who removes a debt instance is the same one who introduced it; while in the remaining 37% of cases the developers who fix SATD have higher experience than those who introduce it. The study also measured the partial correlation between quality code metrics (Coupling, Complexity and Readability) and SATD, but found that it

is not significant between any of them, an in-line observation with Potdar and Shihab (2014).

Impact of SATD

Instead of looking at code quality metrics which were validated to have no clear correlation with SATD, Wehaibi et al. (2016) investigated the relation between SATD and the quality of software by looking at defects. Their study used a pattern-based detection to find files that contain SATD in the repositories of 5 open-source systems; in total 10.17% to 20.14% of files were labeled as SATD files. To find defects, the change history of every system was mined to find patterns that indicate defects, such as: "defect", "bug ID", "fixed issue #ID". With both datasets, the study investigated: i) the amount of defects in files with and without SATD; ii) the percentage of SATD related changes that are defect-inducing; and iii) if changes that involve SATD files are more difficult than the ones that do not. The authors compared the percentage of defects in SATD vs. non-SATD files, and the amount of defects in SATD files before and after the debt introduction. They found no clear relation between defects and SATD. To observe if SATD-related changes introduced future defects, they used a bug-introducing change identification algorithm proposed by Śliwerski, Zimmermann, and Zeller (2005) (SSZ) as implemented in Commit Guru (Rosen, Grawi, & Shihab, 2015), and found that they are less prone to introduce future defects. Lastly, using 4 change difficulty measures from previous work, the authors found that SATD-related changes were more difficult than non-SATD ones.

To clarify the relation between non-SATD source code comments and software quality, Miyake et al. (2017) partially replicated the study by Wehaibi et al. (2016) on 4 open-source projects. Their results agreed with the previous study, finding that SATD files are more prone to undergo a defect fix. However, they also found that the mere existence of comments at the method or file level is related to more future code fixes, even if they do not contain SATD. Nevertheless, SATD comments were found to be more effective to identify fix-prone files and methods than comments without SATD.

Removal of SATD

Most of the previous comprehension studies targeted the introduction, diffusion, and evolution of SATD. Early studies also looked into the final stage of SATD, its removal (Bavota & Russo, 2016; Potdar & Shihab, 2014). Their efforts were not dedicated specifically to the removal of debt. Recently, Maldonado, Abdalkareem, et al. (2017a) studied precisely this, investigated: i) how much SATD is removed from source code; ii) who removes it; iii) how long does it remain in a system; and iv) what leads to removal activities. The authors studied 5 well-commented systems written in Java, which vary in size, domain and number of contributors. Their study showed that 40.5% to 90.6% of SATD was removed from the study systems. Comparing the names and e-mail addresses of the developers who introduced and removed SATD from the repository commits showed that on average 54.5% of SATD is self-removed, i.e., by the one who introduced the debt; confirming the finding first presented by Bavota and Russo (2016). A comparison between self-removed SATD and the one removed by others indicated that the second survives for longer in a system. Concerning the median survival of SATD, the study found that it can remain in a system between 18 to 172 days before being removed. A survey of developers was also conducted in order to understand what activities lead to the removal and introduction of SATD (Maldonado, Abdalkareem, Shihab, & Serebrenik, 2017b). The survey revealed that developers mostly add SATD to track potential bugs or code that needs improvement; similar to the finding of Vassallo et al. (2016). On the other hand and in-line with the observation by Palomba et al. (2017), participants indicated that they mostly remove SATD when fixing bugs or adding features, but not as a dedicated activity.

After the above observations on the removal of SATD, Zampetti et al. (2018) conducted an in-depth quantitative and qualitative empirical study on the removal of SATD. The authors built on top of the previous work of Maldonado, Abdalkareem, et al. (2017a) by analyzing their same dataset, focusing on the underlying circumstances of SATD removal from source code. The study investigated how much debt was removed by accident, i.e., without the intention of resolving debt, but as a collateral of software evolution. The study found that 25% to 60% of SATD comments, as they were removed due to full class or method removals. However, 33% to 63% of SATD comments were removed as part of a change in their corresponding method. In the remaining instances,

comments were removed without any actual code change, possibly due to developers removing an outdated SATD comment or accepting the debt's risk. By computing the cosine similarity between SATD comments and commit messages, the authors looked for documented evidence of SATD removals, finding that only about 8% of the cases mentioned addressing the debt or justifying why it is not required to do so anymore. The study also looked at the types of changes that happen along SATD removals, finding that developers often apply complex changes across the code but also specific ones related to method (API) calls and control logic. On removals associated with API changes, 55% belong to the addition or editing of features; while removals linked to conditional changes are more diverse but often involve the removal of code.

SATD Interest

Several works shed light on the SATD life cycle stages. Nevertheless, none had yet proposed a concrete way to measure the interest of SATD, i.e., the increased cost of repaying debt in the future. A recent study by Kamei et al. (2016) focused on determining a way to measure this cost precisely. It investigated if the debt instances incur a positive interest (i.e., they become more difficult to repay), negative interest (i.e., become less difficult to repay), or no interest over time. Sixteen different code complexity metrics were first evaluated and then filtered down to two, namely LOC and Fan-In. LOC was used because it is highly correlated with most of the metrics evaluated initially, excluding Fan-In, thus both were selected. This work performed a case study on Apache JMeter and used JDeodorant to extract raw comments, which were then filtered and manually validated to contain SATD. To measure the incurred interest, the study scoped to the method-level for the SATD instances and computed the LOC and Fan-In metrics at the moment of their introduction and removal. Results showed that for both measures, 42% to 44% of SATD incurs a positive interest; while around 8% to 13% and 42% to 49% has negative and no interest, respectively. The interest quantification of SATD is a proxy to estimate the effort needed to repay it. In the following subsection we go over additional studies with this focus.

Other Empirical Findings Related to SATD

Two recent studies presented observations related to SATD while looking at different aspects of software development. While studying the continuous integration practices of 152 practitioners from a large financial organization (ING Netherlands), Vassallo et al. (2016) showed that 88% of the practitioners mentioned self-admitting their bad implementations of code through comments (i.e., SATD). This observation reflects the practical importance of addressing SATD during the development process. In an alternate scenario, while investigating the relation between 3 types of code changes and refactoring activities, Palomba et al. (2017) noticed that in feature-introducing changes, the refactored files often had SATD in its previous version. Because of this, they applied a pattern-based detection to spot SATD in each refactoring activity. Their results showed that 46% of the classes had a SATD instance before being refactored, and 67% of the commits that refactored code also removed a debt instance. This indicates that developers mostly apply refactorings to repay existing debt before introducing new features into their source code.

To summarize the findings and contributions of the above comprehension studies, we present them in Table 2.4, along with the number of studied software systems. Since comprehension studies rely on a SATD detection approach, we also include them along with the comment extraction tools used in Table 2.4. Note that most comprehension studies used a manual inspection or a pattern-based detection, while only one study implemented a NLP approach. Certainly, this trend is caused by the difficulty to replicate different detection approaches. However, it compromises their effectiveness of studying the phenomenon. We expect and encourage future studies to implement the more recent and accurate SATD detection approaches.

Table 2.4: Overview of main findings per SATD comprehension study.

Reference	Contribution(s) / Finding(s)	Studied Systems	Detection Approach	Comment Extraction
Potdar and Shihab (2014)	- More experienced developers tend to introduce more SATD. - Time to release pressure and change complexity do not play a major role in SATD introduction. - Most of SATD is removed in the next immediate next release.	4	Manual	srcML based
Maldonado and Shihab (2015)	- Identified 5 different types of SATD. - The most common type of SATD is design or requirement debt.	5	Manual	JDeodorant
Bavota and Russo (2016)	- There is no clear relation between code quality metrics and SATD. - The amount of SATD increases over time in a system. - Code debt occurs more than design and requirement debt. - SATD lasts for a long time in source code before being removed. - About 57% of SATD is removed from source code; 63% of the time by who introduced it, 37% by other experienced developers.	159	Pattern based	srcML
Wehaibi et al. (2016)	- There is no clear relation between defects and SATD. - TD files defectiveness increases after the introduction of TD. - SATD changes lead to less future defects than non-SATD changes. - SATD changes are more difficult to perform. - Empirical evidence that TD affects the development process by making it more complex. - The impact of SATD is not related to defects, rather in making	5	Pattern based	Ad-hoc. Python

	future changes more difficult to perform.			
Vassallo et al. (2016)	- Most practitioners self-admit their bad implementations of code through comments.	N/A	N/A	N/A
Kamei et al. (2016)	- 42% to 44% of SATD incurs in positive interest. 8% to 13% and 42% to 49% has negative and no interest, respectively.	1	Manual	JDeodorant
Miyake et al. (2017)	- SATD comments are more effective than non-SATD comments when identifying fix-prone files and methods.	4	Pattern based	Ad-hoc, Java
Palomba et al. (2017)	- Developers mostly apply refactorings to repay SATD before introducing new features.	3	Pattern based	srcML
Maldonado, Abdalkareem, et al. (2017a)	- SATD can remain in a system between 18 to 172 days. - Developers mostly remove SATD when fixing bugs or adding features, and use SATD to track future bugs and bad implementation areas. - Most of SATD is removed, and most of it is also self-removed.	5	NLP detection	srcML based
Zampetti et al. (2018)	- A large percentage of SATD removals are accidental. - Only around 8% of SATD removals are documented in commits. - While removing SATD, developers mostly apply complex changes but also, specific ones to method calls and conditionals.	5	NLP detection	srcML based

2.3.3 Repayment of SATD

We surveyed work that contributed towards the comprehension of SATD on its removal (section 2.3.2), and interest growth (section 2.3.2). Although those studies explain how and who removes SATD, and propose a way to measure the growth or decline of SATD over time, they do not propose approaches towards managing or repaying debt. In this section, we describe studies that tackle this problem.

As a subset of Technical Debt, the ultimate goal of studying SATD is to propose approaches that focus on removing it from a system, i.e., repaying the admitted debt. In this regard, a couple of recent studies have presented techniques to estimate the effort and prioritize the resolution of SATD. In 2016, Mensah et al. (2016) proposed an approach to estimate the rework effort needed to resolve SATD, measured in LOC. The authors used the text-mining approach to identify debt instances in 4 open-source projects and classify them by type with a dictionary derived from the work by Maldonado and Shihab (2015). The measure of estimated rework effort was calculated giving term weights to debt instances based on their frequency of SATD indicators, i.e., one of the patterns found by Potdar and Shihab (2014), and expressed the average commented LOC per SATD-prone file (files that contain comments with debt indicators) in a system. The study found that, on average, an effort of between 13 and 32 commented LOC need to be addressed per SATD-prone file. This estimated effort fluctuates based on the type of debt to be addressed, with documentation requiring the least amount of effort, and design debt needing the most.

More recently, Mensah et al. (2018) extended their rework effort estimation study and combined it with a 6-step SATD prioritization scheme. This new approach aims to inspect SATD instances and classify them by how urgently they need to be addressed and estimate the rework effort they require. Similarly to their previous work, this estimation is computed in a multi-phased approach, where initial steps handle the extraction of comments, identification and classification of debt instances into their types using the text-mining approach. Before computing the rework effort estimation, the extracted comments were manually categorized based on their textual indicators as: i) *major* if they are urgent, or *minor* if they can wait; ii) *complex* based on their difficulty, and *significant* based on their importance; iii) *expected* if the task is pending, and *expedited* if it denotes a rushed or poor

implementation. SATD instances that should be prioritized were marked as *vital few* tasks or as *trending-many* tasks, and assigned a possible cause of introduction. Along with the proposal of a repayment approach, this work also presented interesting empirical findings, showing that 31% to 39% of SATD comments are major tasks, and 58% to 69% are minor; while most of the major tasks are complex to resolve for developers. Among the possible causes for SATD introduction, the study found 4 that are the most prominent, being: code smells (23%), complicated and complex tasks (22%), inadequate code testing (21%), and unexpected code performance (17%). Regarding the effort required for the resolution of vital few tasks, i.e., those that should be prioritized, developers would need to address 10 to 25 commented LOC per SATD file.

The concept of classifying the SATD comments into different classes that indicate how difficult, important, and urgent they are can serve as a great contribution to deciding which debt to resolve first. However, is important to note that for both of the above works on repayment output results in commented LOC, which might not be intuitive for developers or managers, nor the best or only measure to estimate effort or prioritize debt resolution. In either way, both approaches compel the most recent in SATD repayment.

2.4 Future of SATD Research

In this section, we present promising research avenues based on gaps and opportunities that we observe in current studies and discuss the challenges to overcome in order to advance the state of the art. The ideas and calls to actions presented throughout this section are new proposals deduced from our observations, which we support with related literature.

2.4.1 Future Challenges in SATD Setection

Improving Validity

SATD detection can benefit from improved validity, future work should enrich existing datasets and expose new ones using state of the art detection and classification approaches. Since TD can also be self-admitted in other software artifacts, such as commit messages or issue comments, datasets should not be limited to SATD found in source code comments. We expand on these ideas below:

- **Richer datasets.** As we saw in the work surveyed in Section 2.3, most of recent work relies on data from design and requirement SATD (Huang et al., 2018; Maldonado, Shihab, & Tsantalis, 2017b; Yan et al., 2018; Zampetti, Noiseux, et al., 2017; Zampetti et al., 2018). This data originates in the dataset made available by Maldonado and Shihab (2015), in which design and requirement debt was detected far more frequently than other debt types. This limits approaches such as the NLP and ensemble text-mining approaches to be restricted on classifying debt instances in all existing types. Using a tool such as SATD Detector can support the creation of larger datasets with more instances of the rarer SATD types. Such datasets can then be complemented by artificial balancing techniques to enable better classification approaches. Another challenge with current datasets is that they are scarce, and limited in size and diversity of projects they contain. Huang et al. (2018) found that cross-project training increased the performance of identification classifiers. Thus, SATD detection approaches will benefit of having richer datasets to train on.

- **Detection in other software artifacts.** The majority of work surveyed in Section 2.3.1 detected SATD through source code comments. There are other software artifacts that contain extracts of human interaction and communication, such as issue messages, commit messages, or even discussions in git repositories. These artifacts can also hold text where technical debt is self-admitted by developers. Dai and Kruchten (2017) studied the possibility of detecting TD with issue comments, finding that although developers do not explicitly mention TD inside issues, they do so indirectly. Their study identified over 114 useful key words that can be used to detect different types of TD from the description and summaries of issues. This is a similar finding to the patterns surfaced by Potdar and Shihab (2014) for SATD. Bellomo et al. (2016) also investigated the existence of TD indicators within issues messages and found that developers are aware of the concept of TD, and they refer to it when filing issues. This might indicate that technical debt is also self-admitted in issue messages.

  Nowadays there is a plethora of repositories that can be mined to investigate the occurrence and diffusion of SATD in alternate software artifacts. One example is *JIRA*, a repository presented by Ortu et al. (2015) which contains data from the Jira Issue Tracking System. The

JIRA consists of over one thousand open-source projects with 700 thousand issue reports, and 2 million issue comments. As its authors suggest, it can be mined to retrieve information about TD, and thus potentially, SATD. The investigation of how much debt found within issues is also self-admitted by developers and the usefulness of this approach remains as future work. Considering the above software artifacts for an approach such as the SATD change-level determination proposed by Yan et al. (2018) could also yield a promising future. Including features extracted from different software artifacts can complement the 3 dimensions studied by Yan et al. (2018) to extend the set of features taken from source code and change history, potentially resulting in improved TD determination models. As detecting SATD at the change level presents different benefits to software developers in contrast to detection at the file level, there is broad potential and room for further investigation on the topic.

Calls to action:

- *Mine larger sets of software repositories from different domains to produce richer SATD datasets.*
- *Study the presence of SATD in other software artifacts, such as the messages and descriptions of issues and commits.*

Improving Traceability and Adoption

In Section 2.3.1, we surveyed several approaches for SATD detection with different characteristics and techniques that allow them to achieve performances that surpass their predecessors. Each has an application, as well as points in favor and against that facilitate their replication. For example, one could argue that manual detection and pattern-based approaches (see Section 2.3.1) are the easiest to replicate, but doing so is time-consuming and relies on human expertise. On the other hand, automated approaches that use machine learning are scalable but rely on a training dataset to achieve a comprehensive performance (see Section 2.3.1). Future work should aim to facilitate the replication of detection approaches to promote their adoption, and to develop tools to increase the admittance, quality, and traceability of SATD. One materialized example for this is SATD Detector, where the ensemble text-mining was implemented as a tool ready for use in development time.

Certainly, any approach or technique that can be offered as a tool is the best proxy to improve the traceability and adoption of SATD. We describe actionable ideas that can support this based on opportunities we observe from previous related work below:

- **Visualization tools.** Alongside improved detection techniques, both researchers and practitioners can always benefit from tools that implement them. An interesting avenue comes from the visualization approach presented by Ichinose et al. (2016); city-like views in a virtual reality environment combined with an automated detection and classification approach could provide a highly intuitive interface for SATD identification and management. Visualization tools can also be extended to estimate the repayment effort of detected SATD with an approach such as the one proposed by Mensah et al. (2018). In this scenario visual cues could point at debt that can be repaid in the source code. The development of a tool that can display where SATD is located and offer an estimation of the effort required to address it would strongly enable developers to manage and repay SATD in their repositories.

- **Annotation of comments.** While classifying grammar smells, Stijlaart and Zaytsev (2017) pointed at the "Shortage Smells" as missing pieces of grammar. As a subset of this, "Debt" smells were defined to happen when comments clearly denote debt but are missing an annotation that will facilitate its traceability, such as *"TODO"* or *"FIXME"*. In this case, an approach or tool that adds these annotations would solve grammar smells by self-admitting the technical debt. For this to be feasible, researches can use one of the more recent SATD detection approaches and add special annotations to comments that are missing them. In this way, SATD would be easier to trace by developers using IDEs that support the tracking of these annotations.

- **Reduction of false positives.** Another important challenge is to reduce false positives in SATD detection. One of the issues with the approaches analyzed in Section 2.3 is that most approaches look at comments directly, disregarding the source code context. For example, the pattern-based approach was found to produce over 25% of false positives (Bavota & Russo, 2016). Although more advanced detection approaches have been presented, they still focus on source code comments only. Such approaches might find cases indicating debt that was

already repaid but its corresponding self-admitted annotation was never removed. On this regard, Sridhara (2016) proposed a technique to validate the up-to-date status of comments that include *ToDo* annotations. This is a hybrid approach that considers both, source code and comments. Future work can improve on such technique and extend it to work on any comment that indicates SATD, and not only those with *ToDo* annotations. Moreover, as seen in section 2.3.1, TEDIOUS is the only detection approach that inspects source code instead of comments to recommend when design technical debt should be self-admitted. Certainly, a way to mitigate false positives in future SATD detection efforts can emerge from using a hybrid approach that inspects the source code in scope and comments of a debt instance.

Calls to action:

- *Develop tools that enable a categorized visualization of SATD to support its management.*
- *Develop a detection approach that adds annotations to debt comments that are missing them.*
- *Develop detection approaches that inspect and analyze both, comments and source code for improved accuracy.*

2.4.2 Future and Challenges in SATD Comprehension

To deepen the understanding of SATD, research work should identify observations on this phenomena that apply across projects and can be generalized. In Section 2.3.2, we surveyed work that studied large sets of systems or specifically tried to diversify their subjects in domain and programming language (Bavota & Russo, 2016; Wehaibi et al., 2016). Nevertheless, a clear challenge to overcome is that most findings and contributions on SATD (see Table 2.4) and its effects in software development came from studying open source systems that were mostly written in Java (see the software systems studied by the surveyed work in Section 2.3). Future research should extend to investigate proprietary software or systems that are written in various programming languages. This will aid towards the generalization of current findings or contrast new observations in different scenarios and environments. Similar to previous efforts such as the empirical SATD study by Bavota and Russo (2016) on 159 projects, important findings on SATD should be investigated on large scale to confirm they generalization.

We remark that the studies covered by this literature survey consider a scenario where identifying the introduction of TD is valuable for the development process, and where the management and repayment of TD are desired practices. More importantly, in the case of SATD, the assumed scenario is one where the use of source code comments is intrinsic to the development process. However, this may not generalize to all software development, as it depends on the used methodologies and policies in place. In our survey, we did not find any SATD study that worked on proprietary software systems. Investigating the relation between the introduction, management, and repayment of SATD in different development methodologies remains as future work. This will help to achieve a more general and thorough comprehension oh the phenomena. Below, we present actionable ideas for future research to broaden the comprehension of SATD:

- **Examine other kinds of impact.** Previous work has investigated the impact of SATD on software quality, but only in the scope of software defects, which do not seem to have a direct relationship with SATD (Wehaibi et al., 2016). However, this is the only finding on the impact of SATD among the papers that focus on the comprehension of the phenomenon (see Section 2.3.2). Therefore, we believe that future work should seek a deeper understanding of different aspects in which SATD can impact the development process. We observe the opportunity to investigate on the impact of SATD in aspects such as: effort in future maintenance and evolution (e.g., code decay), the ability of a system to adapt to new technologies or changes in process, and even the socio-technical impact of SATD.

- **Qualitative classifications.** So far, source code comments that point to TD have been classified following the categories defined by N. S. R. Alves et al. (2014), such as in the classification work on SATD by Maldonado and Shihab (2015). This is a high-level classification of the comments as they indicate what the debt is about. Another perspective is to investigate their implication in the development process. As an example, the comment: *"//Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes."* from ArgoUML was classified by Maldonado and Shihab (2015) as design debt. This classification does not inform the developers about its implication; perhaps it implies a feature addition, a bug fix or another software maintenance tasks. A study using such level of taxonomy was

presented by Panichella et al. (2015), who classified mobile app user reviews into useful categories related to maintenance tasks. Replicating such taxonomy in the area of SATD can provide developers with better insight on the implications of SATD. Improving the overall understanding of the debt instances on their systems to support their management.

Calls to action:

- *Investigate SATD in proprietary software systems and in various programming languages (other than Java).*
- *Investigate the impact of SATD on various software engineering aspects, such as maintainability and evolution.*
- *Produce a qualitative taxonomy that reflects the implications of SATD in software maintenance tasks.*

2.4.3 Future Challenges in SATD Repayment

Quantitatively Prioritizing Repayment

Proposing approaches and techniques to mitigate and repay debt is of utmost importance in SATD research. Studies in the past few years have shed light on the importance of this phenomena, but they have mostly focused on detecting and understanding SATD, rather than directly pursuing its resolution. Merely 11% of the studies that we surveyed focus on repayment efforts, thus, there is much work to be done in this area. We present the main challenges to overcome in SATD repayment below:

- **Effort Estimation.** SATD repayment contributions have scoped to prioritize its resolution based on the estimated effort for addressing a debt instance (Mensah et al., 2016, 2018). However, this approach outputs an estimation value in commented LOC, which might not be the best, and certainly not the only measure to estimate effort (Shihab, Kamei, Adams, & Hassan, 2013). Undoubtedly, how to measure effort remains a challenge to overcome and a milestone to reach when deciding which debt to repay first.

- **Prioritization of SATD.** Certainly, prioritizing SATD repayment must be part of future research work. Given a set of instances of SATD in a system, developers need an approach to recommend which debt instances to resolve first. Thus, approaches that measure the growth of debt instances and their resolution cost must be combined. Akbarinasaji and Bener (2016) presented the idea of adding TD as a financial obligation that can be recorded as a type of liability in a balance sheet. To achieve this, TD needs to be identified, quantified, and monetized. Although an approach to monetize SATD has not been presented, some efforts already took a step forward, such as the quantification SATD interest by Kamei et al. (2016). We argue that SATD prioritization is one of the most important challenges that require attention in this domain.

- **Acceptance of SATD.** Not all SATD has to be repaid, fixing a shortcut or hack in the source code can be more expensive than beneficial. A proper measurement of TD repayment effort could aid developers to decide whether to live with the debt and its risks or not. Such repayment estimation has to consider the potential evolution of the debt as it can incur in positive interest over time (Kamei et al., 2016). Future work should study the extent of SATD acceptance in software systems and under which conditions.

Calls to action:

- *Investigate new measures to estimate the effort required to repay SATD.*
- *Develop approaches to prioritize the repayment of SATD.*
- *Investigate to which extent SATD is or can be accepted in software systems.*

Integrating the repayment of SATD

The activity of repaying TD must be integrated into the software process. To this matter, the development of new tools and techniques that motivate and facilitate the repayment of SATD is required. We present two ideas that can facilitate this below:

Gamification of SATD repayment. SATD research not only needs to give answers on which debt instance to address first, and to ease and promote a culture of resolving debt instances as part

of the normal activities in the development process. In this regard, the use of mechanisms such as Gamification (Deterding, Dixon, Khaled, & Nacke, 2011), i.e., the application of game-like features in non-game context could be of benefit. Gamification has increasingly been proving its usefulness to motivate, accelerate and ease human productivity and it has already been studied in the context of software development, i.e.,(Biegel, Beck, Lesch, & Diehl, 2014; Dubois & Tamburrelli, 2013). Thus, it has the potential to support and motivate the repayment of SATD among developers.

Identify who introduced the debt. Knowing which developer self-admitted debt in the first place and the rationale for doing so is important. Siegmund (2016) suggested supporting the task of identifying developers who are responsible for a component, and helping them communicate with others who have introduced SATD. Such scenario would require an approach that identifies SATD and determines the developer who introduced it. Enabling a channel of communication between developers can shed light into the rationale behind a debt instance to support is repayment. However, it can be problematic as a debt-introducing developers may no longer be available. Thus, its applicability is limited by the phase at which SATD is managed.

Call to action:

- *Study the usage of gamification techniques to motivate the repayment of SATD.*
- *Complement SATD detection approaches by identifying who introduced the debt to enable communication between developers, facilitating repayment.*

2.5 Conclusions and Limitations

We surveyed empirical research work in the arising topic of SATD, which has developed rather quickly in recent years. This literature survey has been performed on studies related to self-admitted technical debt, as defined by the exploratory study of Potdar and Shihab (2014). We used this study as the basis for our survey and applied snowballing to find related work from it. Although we complemented the lookup for SATD-related work with results from academic search engines, we found no studies that focus on SATD that were not originally found during the snowballing process. Thus, the papers encompassed in this survey are limited to those released after 2014 and until the compilation of this survey in July of 2018. The selected papers are also limited to those returned by

the search engines and keywords we used, and only to those that mainly focus on studying SATD (see Section 2.2.1).

From our survey subjects, we observe how researchers have evolved current approaches from manual observations to automated techniques for detecting and classifying debt instances, and have advanced the overall understanding of the SATD phenomenon in the software development process. Naturally, the focus of SATD studies was centered in detecting the presence of debt, and understanding its life-cycle. Once detection approaches were accurate and replicable, the focus switched to studying how SATD grows over time and how it is removed from software repositories.

Our work highlights several of the challenges to overcome in the area, and presents various promising avenues for future studies based on the gaps and opportunities seen in current research work. Furthermore, this survey compiles the tools and datasets that can be used as a foundation to motivate and facilitate the submission of novel and improved approaches in the area. We believe SATD will continue receiving attention in the field the upcoming years.

We notice a solid foundation for the detection and comprehension of SATD, therefore, this is the right moment to centralize efforts towards the most notorious gap in the area, which is on SATD repayment. Although we noticed that researchers have stepped into the idea of repaying SATD already, we certainly observed a lack of studies focusing on this, which is of critical importance. Motivated by these observations, and with an available set of tools and techniques to extend SATD research, we focus our efforts into SATD repayment. Consequently, the following chapter of this thesis presents an empirical study conducted precisely towards the repayment of SATD.

Chapter 3

Towards Self-Admitted Technical Debt Repayment

3.1 Introduction

Previous research has proposed different approaches to detect and classify Self-Admitted Technical Debt (SATD) in source code comments (Huang et al., 2018; Maldonado & Shihab, 2015; Maldonado, Shihab, & Tsantalis, 2017b; Potdar & Shihab, 2014). These approaches enable software developers to locate and potentially address debt instances to improve the quality of their code base. However, the mere task of detecting SATD does not suffice; large software systems can have broad sets of debt pending resolution. These debt instances have the potential to become more complex and costly to repay over time, and thus, should be managed as part of regular software maintenance and evolution practices. Managing technical debt in a system is indeed a complete area of its own, thus our interests reside in SATD particularly. In this regard, an idealized goal of studying SATD repayment is to formalize approaches that determine what SATD should be resolved, and when to do so. Certainly, not all technical debt has to be resolved. Empirical evidence shows that SATD stays in software systems for long periods of time before being removed, and that multiple instances of debt remain in them (Bavota & Russo, 2016; Maldonado, Abdalkareem, et al., 2017a; Potdar & Shihab, 2014). This suggest that either some of this debt is not harmful and systems can evolve with them, or that when it is harmful, developers are not aware of the consequences. Asides from intuition,

empirical work has shed light into this, showing that indeed, some SATD becomes more complex to repay over time. Kamei et al. (2016) referred to this increase in difficulty to repay the debt as SATD interest, and found that 42% to 44% of debt instances incurred positive interest. Evidently, repaying SATD is of importance for software maintenance, and yet very few studies have focused on doing so.

With the knowledge gained from the literature review in Chapter 2, we are motivated to fill the gap in SATD research and concentrate our efforts towards the repayment of SATD. With several recent findings and contributions on the detection and comprehension of the SATD phenomenon, we believe it is feasible to move towards the goal of determining which SATD instance should be prioritized, i.e, decide which instance of debt should be resolved first in a system. However, this can not be achieved in a simple or straightforward fashion; prior work is required to understand different measures for SATD instances with repayment in mind. We performed a preliminary online survey to developers to gain insight on the elements they use to decide which SATD instance should be resolved first. Developers who participated in the online survey mainly consider the effort required to repay a debt instance as the deciding factor to do so. Yet, participants had distinct definitions on how to estimate effort. We used the insight gained from this survey in combination with measures for effort estimation proposed in previous work and our own conjectures to determine a set of metrics for SATD with repayment in mind. Because these metrics serve to determine the amount of work needed to repay the debt, we refer to them simply as *SATD repayment effort metrics* throughout our work.

In this chapter, we perform an empirical study of SATD instances found in open-source software systems. We determine metrics to measure SATD repayment effort based on insight from developers and previous work in the area. Then, we extract a set of historical metrics that we use as our independent variables to build linear regression models for SATD repayment effort metrics, i.e., our response variables. We surface the SATD repayment effort metrics that can be consistently modeled in our studied projects, and the historical metrics that can be used as good early indicators for SATD instances that should be repaid. We formalize our study with the following research questions:

- **RQ1:** ***Is there a consistent SATD repayment effort metric that can be modeled using historical data?***

  We investigate a set of metrics used to measure the effort required to address SATD instances and see if change history data can determine those that should be addressed. Our goal is to see if there are consistent SATD repayment effort metrics that can be used across systems.

- **RQ2:** ***What are the best indicators for SATD repayment effort metrics?***

  Following up to RQ1, we are interested in knowing which of the metrics extracted from historical data work best as early indicators of SATD that should be addressed.

Overall, we study more than 18 thousand unique SATD instances and their change history detected in over 5 million source code comments, and across more than half a million commits in 8 software repositories: Camel, Hibernate, JMeter, Ant, Hadoop, PMD, EMF, and Tomcat. We evaluated 7 different metrics to estimate SATD repayment effort (SATD removal time in days, number of words in SATD comments, interest in terms of LOC and Fan-In, compound interest rate in terms of LOC and Fan-In, and change proneness) and 9 different change history and defect prediction metrics (contributors of a file, author experience, bug-fixing commits, entropy, file churn, commit churn, modified files, modified directories, and modified subsystems).

We built linear regression models with our SATD effort estimation metrics as response variables and the historical metrics as exploratory variables. We found that: i) change proneness and the SATD removal time could be consistently modeled in studied systems; and ii) the best early indicators for SATD that should be repaid are: the number of developers that contribute to a SATD file, amount of file churn, and the experience of the developer who introduced the debt. This implies that looking at historical data at the time when SATD is introduced can be of value to developers to estimate important SATD to repay in future maintenance.

This study is organized as follows: Section 3.2 reviews the online survey sent to developers; Section 3.3 overviews our study approach and details our data collection steps; Section 3.4 presents the results of our study; Section 3.5 presents threats to validity; and lastly, Section 2.5 concludes the study and mentions future work.

3.2 Preliminary Work

As preliminary work to our empirical study, we wanted to get insight from developers on the different elements that they consider when deciding to resolve SATD in their systems. To do this, we sent out an initial online survey to 200 developers of public repositories. To ensure the recipients of the survey had experience, we selected those who had over 3,000 commits pushed to public repositories on GitHub. The survey was composed of 10 questions aimed at: i) distinguishing traits in the developers, such as their roles, level of activity and experience in general software development, ii) determining their experience with SATD, and mainly iii) identifying the elements they use to prioritize SATD. The participants were given a brief introduction on what is SATD and a set of prioritization elements to pick from (multiple options could be selected). The set of elements were selected based on a combination of our own conjecture and elements that have been used in previous work on SATD repayment (Kamei et al., 2016; Mensah et al., 2016, 2018). We did not restrict the addition of other elements that were not part of the default set; an option to add others was included. We also asked developers to provide feedback on how they measure the different elements provided as options. A complete list of questions of the survey and their options are detailed in Appendix B.2. The survey received 19 responses in total (9.5% response rate), which may appear to be a small rate. However, it is comprehensive in the software engineering field (Singer, Sim, & Lethbridge, 2008). We made the survey responses publicly available online[1]. The published data has been anonymized, thus question 10 of the survey has been omitted as it conveys personal information.

From the 19 answers received, we characterized our participants as good subjects. Only 1 participant reported to have less than 5 years of development experience, indicating participants were mostly experienced. Regarding activity and code contribution, 85% of the participants reported contributing frequently to their projects, on a daily to monthly basis. On their development roles, 14 developers replied to mostly work on feature additions, 7 on bug fixing, 6 on code reviewing, 4 on code testing, and 2 others reported working on a mix of activities. Thus, participants work mainly on adding features, but also on a variety of other development activities in parallel. When asked

[1] http://das.encs.concordia.ca/uploads/SATDPrioritizingResponses.csv

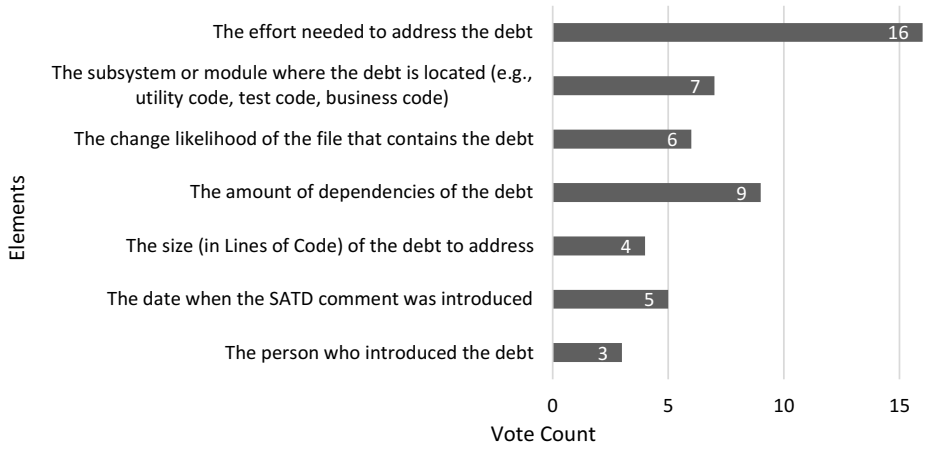

Figure 3.1: Vote count per SATD prioritization element.

about previous experience with SATD, 15 out of 19 developers replied that they have introduced SATD in the past. Furthermore, 16 developers had addressed SATD in the past, while only 3 others had not. Indicating participants are mostly familiar whit resolving SATD.

The core of this initial survey was to find the elements developers consider to prioritize SATD repayment. Figure 3.1 overviews the answers we received. We can observe that most participants consider the effort it takes to address a debt to decide which to resolve first, followed with spread votes to: the amount of dependencies of the debt, the subsystem were it is located, and its change likelihood. The element less voted was if to consider the person who introduced the debt. Besides the votes seen answers, 2 developers also added that they do not use any of the elements provided as options. One answered to address SATD only by a requirement coming from an external issue tracking system, where the actual SATD instance simply serves as a pointer for the requirement in the source code. Another developer responded that to address SATD, she/he only evaluates if the debt instance is impeding fixing a bug or adding features, disregarding the specific elements provided as options for the question.

Participants were not consistent in specifying how they use the voted elements to prioritize SATD. They responded to prioritize SATD located in subsystems of high importance, but gave no details on how they determine critical subsystems. On change likelihood, participants mentioned that they use previous changes in the repository but did not mention how to prioritize using this element. Participants reported using different ways to estimate effort, which was the most common

answer. They mentioned to estimate effort based on: size and complexity of code, amount of dependencies, number of test activities and test code. Although the participants of the survey were experienced, the elements that they use to decide which SATD to resolve first tend more towards a group of measures than to a single straightforward answer. This reveals that, lacking a formal approach, practitioners have not reached a consensus on how to prioritize SATD. Based on the survey, we can conclude that effort is seen as the major decisive factor to repay SATD, but there are many ways to estimate it.

3.3 Case-Study Setup

The approach to our study naturally begins by determining the metrics for SATD repayment effort and by detecting SATD instances. Then, we proceed to measure SATD repayment effort in debt instances and study if the metrics can be modeled with indicators from historical data. To answer our research questions, we use the metrics for SATD repayment effort as our response variables while the historical metrics are used as explanatory variables for linear regression models.

In this section, we explain the setup of our study and the required data collection steps. First, we explain the selection of metrics for SATD repayment effort. Then, we focus on the collection of change history information from our selected projects and detecting SATD in their source code. Later, we investigate the extent to which SATD has been removed in the systems, and under which conditions. Lastly, we describe how we collect different pieces of information, divided in two main datasets: i) data related to different features of SATD instances, i.e., product and process software metrics that are needed for our SATD repayment effort metrics; and ii) historical data that explains the characteristics of the repositories before SATD instances are introduced. To ease the querying and processing of information in our step-by-step data collection, all the extracted data resulting from our work was stored in a relational database. We make this database publicly available online[2] to motivate further research in SATD.

[2] http://das.encs.concordia.ca/uploads/SATD-DB.zip

3.3.1 SATD Repayment Effort Metrics

To decide on a set of metrics to use as SATD repayment effort indicators, we considered those used in previous work and those derived from the answers received in the online survey to developers. In the scope of our work, we selected metrics that can be extracted directly from historical data in the source code and change history of projects. Below, we describe the SATD repayment effort metrics and indicate the rationale behind selecting each:

- **Time of removal (in days)**. Knowing which SATD instances have been removed from a project (i.e., it has been repaid), we can measure the time it took for developers to address the debt since its introduction. Debt instances with a faster removal times could indicate those that were prioritized (Maldonado, Abdalkareem, et al., 2017a). Moreover, this metric is analogue to the element "the date when the SATD comment was introduced" from the online survey answers.

- **Interest gained** in terms of dependencies (Fan-In) and size (LOC). as proposed by Kamei et al. (2016), who used measures of LOC and Fan-In to quantify added effort to repay SATD. LOC is used as a proxy of size and complexity of the debt, while Fan-In measures amount of dependencies. The survey confirmed developers consider both of these elements to estimate SATD repayment effort.

- **Compound interest rate** in terms of size LOC and Fan-In. We build on top of the interest measure to calculate a compound interest rate of growth, which also considers the timespan a debt has remained in a system and the changes it has gone through. This metric is used as a more accurate way to represent the evolution of a SATD instance but is still a measure of their size, complexity and amount of dependencies.

- **Number of words in SATD comments**. Mensah et al. (2016, 2018) used the number of commented lines of code as a measure of effort estimation for SATD in previous work. Instead, we count the number of words in SATD comments as a more precise proxy to determine their size.

- **Likelihood of change**. This was another of the frequently voted elements in the survey

answers. Our conjecture is that SATD files with a higher change proneness should be repaid first as their debt will more likely impact maintenance efforts. We measure the change proneness of SATD files following previous work (Bieman, Straw, Wang, Munger, & Alexander, 2003; Khomh, Penta, & Guéhéneuc, 2009).

Participants reported that they also consider the person who introduced the debt to prioritize SATD, the subsystem or module where the debt is located, and to estimate effort based on the number of test activities and amount of test code. We did not select the person who introduced the debt as one of our metrics because of it is qualitative information that requires additional analysis, instead on a quantitative way to estimate SATD repayment effort. Furthermore, it was the least voted element in the online survey. Nevertheless, we do study if SATD instances were repaid by the same developer who introduced the debt or not. To use the amount of test code and number of tests, we would require deeper investigation to match test code with the actual code of methods with SATD, which was not possible to do accurately for our studied projects. To use the subsystem or module where the debt is located as measure because we requires business logic to identify critical subsystems of systems. Both cases go beyond the scope of our study, and therefore, we leave it for future work. Although we might not be selecting all metrics that can be used to measure SATD repayment effort, we believe our selection is comprehensive and can be obtained with empirical data from source code and change history.

3.3.2 Project Selection

To begin our study, we selected 8 open-source Java projects of different size in lines of code, files, commit history, and contributors. These projects were selected considering they were active, well commented (with different ratios of comments to source code), and belonged to different application domains. Because finding SATD in these projects is crucial for our investigation, we also selected those that had been studied in previous work to confirm that they had a considerable (but varying) amounts of SATD (Maldonado, Abdalkareem, et al., 2017a; Maldonado & Shihab, 2015; Maldonado, Shihab, & Tsantalis, 2017b; Mensah et al., 2018; Yan et al., 2018). The projects are: Camel, a Java framework for integrations with message-oriented middleware; Hibernate ORM,

Table 3.1: Characteristics of studied projects

Project	Java files		Commits	File Versions	SLOC	CLOC	Comment to code ratio	Contributors
	Current	All time						
Camel	18,662	24,772	31,262	137,852	1,193,627	501,669	0.42	537
Hibernate	9,563	29,129	8,824	110,894	701,145	185,165	0.26	426
JMeter	1,284	1,947	13,721	26,427	132,194	73,826	0.56	28
Ant	1,296	5,204	13,940	49,885	140,314	107,560	0.77	66
Hadoop	10,406	23,734	17,128	107,656	1,612,236	523,850	0.32	213
PMD	1,953	9,040	8,706	41,761	112,492	34,728	0.31	49
EMF	3,123	4,725	8,380	29,837	653,133	357,573	0.55	34
Tomcat	2,364	3,904	19,597	48,138	321,170	162,371	0.51	37

An object-relational mapping (ORM) library and framework; JMeter, an application used to load and performance test different applications; Ant, a tool used to automate software builds based on Java; Hadoop, a library and framework for distributed processing; PMD, a static code analyzer; EMF, a modeling framework based on Eclipse for building structured data-model applications; and Tomcat, a servlet container and HTTP Web server. All the selected projects are using git for version control and are written in Java. The reason for targeting Java systems is that previous work on SATD has focused on this programming language; this gives our study a baseline for comparison and soundness. More importantly, several of the tools we use in our pipeline for data collection have been developed to work with Java. Thus, we believe studying a diverse set of projects written in this language will serve as a feasible and practical first step towards SATD prioritization.

Table 3.1 presents the characteristics of the projects selected for this study. The number of current Java files, source lines of code (SLOC), commented lines of code (CLOC), and the ratio of comments to code were taken as reported by SciTools Understand, a tool for static code analysis (SciTools, 2018b). These metrics were taken from the most current version of each project by the date when our data collection was done (September 2018). The number of all-time Java files, commits, and file versions were extracted from the repositories directly using git. The all time Java files represent all the unique file names that have existed in the repository, including renamed or moved files. Whenever any of these files were changed in a commit, a new file version is produced. Lastly, the number of contributors was initially extracted with git by counting distinct pairs of author names and emails in the commit history. However, a single developer can have multiple identity entries of author name and email pairs (Kouters, Vasilescu, Serebrenik, & van den Brand,

2012; Wiese, d. Silva, Steinmacher, Treude, & Gerosa, 2017). Therefore, to mitigate this threat, we merged the redundant entries to count for a single contributor; this is the final number reported in the rightmost column of Table 3.1.

3.3.3 Collection of Repository Data

To start the data collection, we cloned the repositories of our selected projects using git, targeting the main branch of each repository (e.g., main or trunk). We extracted all the unique file paths that have existed in each repository using the *git log* command with the *–first-parent -m* options and filters to get all added, copied or renamed file paths. The *–first-parent -m* options are used to have a simplified and more consistent change history of the main branch of the repositories (Gauthier, 2015; Git, 2018). The set of file paths was used in combination with the commit history to obtain and checkout all the versions of each file in the repository, and the corresponding commit type for each file version (i.e., addition, modification or deletion of a file).

The metadata that can be mined from a software repository is essential to our study. Because we analyze the history of each instance of SATD, we need to track all the changes that happen to them at the method and file levels. Thus, we extract and store the hash, date, and author information from each commit. To have and accurate tracking of SATD instances, we also collect the information of files that have been renamed or moved to another directory using the built-in tracking capabilities of git with the '–follow' option (German, 2017; Git, 2018). This allows us to create a chronological linked-list of file paths and their changes over time. These linked-lists are used in later steps of the data collection to asure that we consider the full history of the files where SATD is located.

3.3.4 Extraction of Source Code Comments

Once we obtained all the versions of each file in the repositories, the next step was to parse their source code to extract source code comments with potential SATD. To do this, we created 2 Java command-line applications that are compiled an executed as runnable JAR files. The first component, named "CX" takes a Java file as an input and invokes a process of the tool scrML (Tool, 2018), which decorates the file's source code with XML. The decorated code is then parsed with the Java SAX Parser (Orcale, 2017) to extract found source code comments. Note that in this step

we do not simply collect the raw text of the source code comments because we must determine the context of each source code comment, we query the parsed code to also collect the following complementary data:

- The type of the comment (Javadoc, line, block, or license).
- Start and end number lines.
- The declaration (type and name) to which the comment belongs.
- The container construct (type and name) where the comment is located.

The start and end number lines are taken to measure the size of a comment. The declaration and container of a comment are used to find the context of a comment, i.e., the portion of code in scope to which it belongs. For example, consider the following simplified code snippet with a SATD comment from Camel:

```
public class ProceedType
{
  // TODO we should be just returning the outputs!
  public createProcessor(RouteContext routeContext)
  {
   //return routeContext.createProceedProcessor();
   return createOutputsProcessor(routeContext);
  }
}
```

The SATD comment in the third line of the snippet belongs to the method declaration *createProcessor*, while it is contained within the definition *ProceedType*. Since the declaration and container of a method can be of different types, while parsing the source code we consider the following Java definitions in the CX component:

- Normal or abstract methods (functions and function declarations).
- Anonymous classes (declaration statements).
- Lambda functions.
- Classes (nested, root, abstract).

- Interfaces (root, nested).
- Constructors.
- Annotation definitions (root, nested).

To get the declaration to which a comment belongs to, we find the closest Java definition below the comment, giving a priority to methods and then other Java definitions in the same order as the types presented above. To get the container of a method, we simply get the Java construct that is enclosing the comment, if there is one. When a comment is found within a method, its declaration is set to that method, and its container to the construct enclosing that method. In some cases, comments are not enclosed by another construct, for example in license comments that are placed at the beginning of a file before a class definition. In such cases, we only use the declaration that the comment belongs to as the context of the source code.

The second component, named "CF", is in charge of filtering the comments extracted with the CX component. To do this, we implemented the approach used by Maldonado *et al.* (Maldonado, Shihab, & Tsantalis, 2017b), which uses 5 heuristics to: a) remove all License comments; b) remove Javadoc comments; c) group consequent single line comments to consider them as 1 comment; d) remove auto generated comments, which can be "Auto-generated method stub", "Auto-generated constructor stub" or "Auto-generated catch block"; and e) remove commented Java source code, as it most likely is unused code and does not contain SATD. To avoid removing Javadoc or License comments that could contain SATD, we do not filter those with a "TODO:", "FIXME:" or "XXX:" task annotation. The filtering phase reduced the dataset of comments by 48%. The first four columns on the left of Table 3.2 present the total number of extracted, filtered (discarded), and useful comments per project in our study.

3.3.5 Detection of SATD

With a filtered set of comments, the next step was to identify those that express SATD. To do so, we leveraged *SATD Detector*, a tool made publicly available by Liu et al. (2018). This tool is an implementation of an approach for SATD detection presented by Huang et al. (2018), which uses text-mining and a machine learning classifier with cross project training to detect SATD

Table 3.2: Detailed amount of processed comments per project and found SATD.

Project	Extracted comments	Discarded comments	Useful comments	SATD comments	% of SATD	SATD Instances
Camel	1,433,291	683,285	750,006	23,917	2%	3,456
Hibernate	822,475	414,385	408,090	52,930	6%	4,514
JMeter	698,064	271,244	426,820	33,513	5%	2,599
Ant	1,124,265	798,917	325,348	19,184	2%	2,193
Hadoop	3,119,464	1,238,878	1,880,586	66,785	2%	5,329
PMD	262,640	150,863	111,777	11,292	4%	1,694
EMF	907,790	642,449	265,341	10,771	1%	1,695
Tomcat	2,191,264	1,006,238	1,185,026	60,697	3%	4,132
Total	10,559,253	5,206,259	5,352,994	279,089	3%	25,612

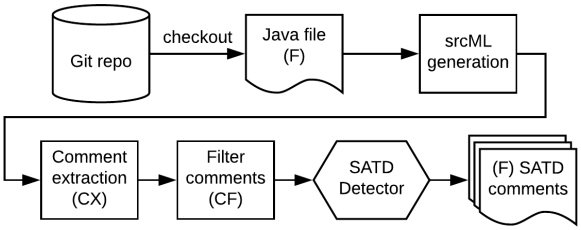

Figure 3.2: SATD collection process.

in source code comments. As reported in a benchmark by their authors, SATD Detector has a better performance than previous SATD detection techniques, with an average F1-score of 0.73. We rely on this tool because to the best of our knowledge, it is the best-performing automated SATD detection method. This tool was published as an IDE plugin tool for Eclipse, however, we implemented its command line version with batch automation in mind. Details on this approach and tool can be read in (Huang et al., 2018; Liu et al., 2018). We invoke a process of this tool and pass each comment found in the extraction process as an argument; then each comment that is identified by the tool as SATD is flagged and stored in our relational database. We automated this process to execute for every comment extracted from every version of all files in each of the studied project repositories. Figure 3.2 presents an overview of the complete SATD collection process; it starts by checking out one version of a Java file *F*, srcML is then used to decorate its source code, the CX component to extract its comments, the CF component to filter those that are useful, and lastly, SATD Detector to detect the SATD comments.

The last 3 columns on the right side of Table 3.2 show the amount of source code comments with SATD and the percentage of SATD in each studied project. Note that these numbers represent

the summation of comments from all versions of all files. However, because in our study we investigate the evolution of SATD instances across their change history, we had to group all the SATD comments that belong to the same instance or case of SATD. For example, the comment *"// FIXME - log configuration problem"* was found in 38 versions of the file *ApplicationFilterFactory.java* in Apache Tomcat. These 38 comments belong to the same SATD instance.

To group the SATD comments and identify the unique instances of SATD, we applied a heuristic that is based on the source code context of the comment, and the similarity to other comments in the same source code context but from a previous file version. We did this because comments in the same source code context (i.e., same project, file, container construct and declaration it belongs to) are likely to be part of the same SATD instance. We used the linked list of renames we extracted earlier from the revision history of the repositories to make sure we consider different file path versions of the same file. Nevertheless, this does not guarantee a correct grouping in cases where there are multiple SATD instances in the same code context. This could be for example, a method with 2 or more comments reflecting different technical debt. In such cases we use the location (start and end lines) of the comments to differentiate the cases. The last exception happens when in the same code context (and line location) a change modifies SATD comment, but now indicating a different technical debt case. For such cases, we compute the Levenshtein (edit) distance for string similarity between comments (Levenshtein, 1966). We selected this metric because its usefulness to measure the edits done to a strings, considering editions, additions and substitutions; these changes occur naturally to comments during software development. A Levenshtein distance of 0 means the compared strings are the same, while the maximum Levenshtein distance is the length of the longer compared string. Therefore, we normalize this metric with a score (from 0 to 1) with the length (*len*) of the longer comment under evaluation (Yujian & Bo, 2007). The similarity score *S* is computed as follows:

$$S = 1 - \left(\frac{lev(str_1, str_2)}{max\{len(str_1), len(str_2)\}} \right)$$

Where $lev\,(str_1, str_2)$ is the Levenshtein distance metric between str_1 and str_2, the strings of two SATD comments. In the history of a file, SATD comments can change slightly without altering

the essence of the acknowledged debt. From observing the dataset of comments, we observed that often punctuations, special characters, and blank spaces were changed in comments, but the SATD instance remained the same. In those cases, the edits done to the comments are minor, specially from one file version to the immediate next. With this in mind, and considering we already know the comments we compare have the same code context (likely to belong to the same SATD instance), we set our similarity score threshold at 0.8. Thus, when $S \geq 0.8$, the compared comments are similar. Previous evaluations of string similarity between evolving source code comments had set lower thresholds (Fluri, Wursch, & Gall, 2007b). However, by looking at the distribution of the computed similarity scores and considering that the compared comments had the same code context, we decided to set higher threshold that fits the data. Figure 3.3 shows the distribution of the similarity scores computed between SATD comments from all projects. The plotted horizontal line at 0.8 in the similarity score axis depicts the set threshold. We assigned an identifier to each unique SATD instance by grouping debt comments that have the same code context and are similar; in this way, we can trace the complete change history of an SATD comment. We report the amount of unique SATD instances found per project in the rightmost column of Table 3.2.

3.3.6 Identification of SATD Removals

Ordering different versions of each unique SATD instance by their commit dates allowed us to identify the first change that introduced the SATD, and the last commit where the SATD was found. Analyzing these two points in the timeline of a SATD instance exposes information of what happened to it (i.e., if the debt has been removed from the system or not, by whom, and under which state of the source code). Figure 3.4 presents a timeline example of a SATD instance in a file with multiple versions since the start of the project and until the most current commit. Here, the first and last commits where the SATD instance was found are represented as black circles. Other different versions of the file with SATD are shown in gray, while circles in white represent versions without SATD. Notice that the last change in the figure marks the removal of the SATD instance. When the SATD is no longer found in the file, we consider that the debt instance has been removed from the system.

In our goal of prioritizing the resolution of SATD, we collect the information from the introduction

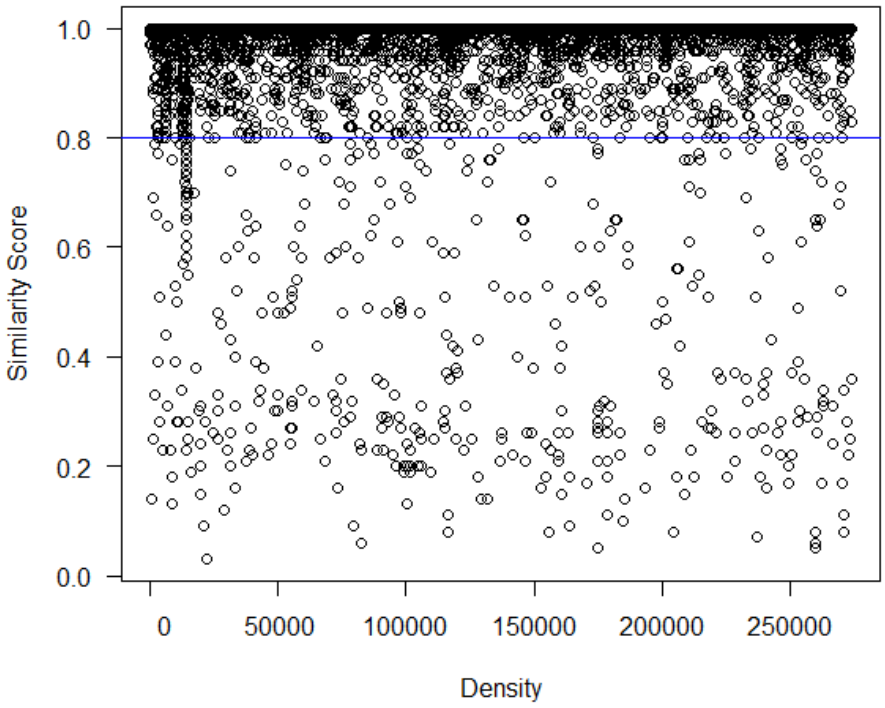

Figure 3.3: Distribution of all computed similarity scores.

and removal points in a SATD timeline, as they are essential to our study. First, we analyze if each SATD has been removed from the system or not by inspecting if there is a commit that touched the file that contains the SATD instance (considering renames) immediately after the last commit where the debt was found. If it exists, we mark it as the removal commit (the new version of the file does not contain the SATD instance). Looking at the type of the removal commit allows us to differentiate two cases: i) the file was changed (SATD is removed by a developer); and ii) the removal commit deleted the file. In both cases, we measure the time of removal as the number of days it took for a SATD instance to be removed by comparing the removal and introduction dates. Previous studies by Maldonado, Abdalkareem, et al. (2017a) and Zampetti et al. (2018) have applied the same principles to detect SATD removals. Similar to these studies, we check if SATD instances are self-removed by checking if the developer who introduced the SATD is the same who removed it. This information surfaces if developers prioritize removing SATD they introduced,

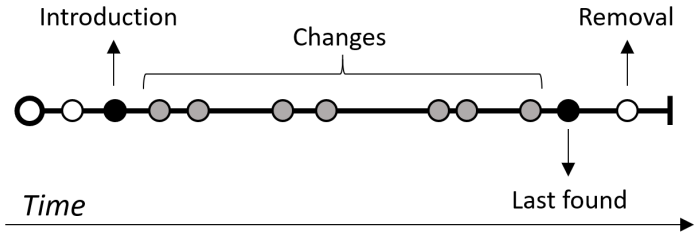

Figure 3.4: Example timeline of a SATD instance.

i.e., self-removed SATD. As described earlier in Section 3.3.2, we are aware a developer can have multiple identity entry pairs (name and email) in a repository (Kouters et al., 2012; Wiese et al., 2017). To mitigate the risk of classifying a SATD instance as self-removed, when comparing the names and emails, we merged duplicate names or emails into a single identity entry per developer. We consider that a SATD instance has not been removed when there are no more commits after the one where the debt was last found. In such cases, we measured the time (in days) the SATD instances have remained in a system by comparing the debt's date of introduction to the latest commit date in a repository. We refer to this time as time of permanence. Table 3.2 presents the amount of SATD instances that have been removed from the repositories, indicating how much debt has been self-removed and how much was removed by deletion.

Zampetti *et al.* found that between 9% to 49% of SATD had been removed by chance when evolving software. These removals were tied to complete file deletions in the repository, instead of developers actually resolving a SATD instance. Because of this, we discard all the SATD instances that we identified as removed by a file deletion from the next steps in our study. In total, 28.8% SATD instances were discarded. Detailed amounts of instances that were removed by deletion per project can be found in Table 3.3. We find that: i) 48.1% to 81.4% (median 66.1%) SATD instances were removed from the systems; ii) on average, 27.8% of the removals were not intentional, i.e., they were removed by file deletions; iii) on average, 49.8% of the removed SATD instances were addressed by the same developer who introduced the debt, i.e., they were self-removed; and lastly, iv) 18.6% to 51.9% (median 33.9%) of SATD instances remain in the repositories. All these observations on SATD removals are inline with the previous findings of Maldonado, Abdalkareem, et al. (2017a) and Zampetti et al. (2018).

To have a better perspective on the removal of our studied SATD instances, we present the

Table 3.3: Detailed amounts of removed and remaining SATD instances.

Project	# Total instances	Removed		Removed by deletion		Self-removed		Remaining	
		#	%	#	%	#	%	#	%
Camel	3,456	2,341	67.7	381	16.3	1,314	56.1	1,115	32.3
Hibernate	4,514	2,913	64.5	1,119	38.4	1,663	57.1	1,601	35.5
JMeter	2,599	2,116	81.4	210	9.9	893	42.2	483	18.6
Ant	2,193	1,778	81.1	745	41.9	913	51.3	415	18.9
Hadoop	5,329	2,564	48.1	341	13.3	659	25.7	2,765	51.9
PMD	1,694	953	56.3	586	61.5	356	37.4	741	43.7
EMF	1,695	910	53.7	243	26.7	528	58.0	785	46.3
Tomcat	4,132	2,944	71.2	419	14.2	2,087	70.9	1,188	28.8
Average	-	-	65.5	-	27.8	-	49.8	-	34.5
Median	-	-	66.1	-	21.5	-	53.7	-	33.9
Total	25,612	16,519	64.5	4,044	24.5	8,413	50.9	9,093	35.5

distribution of elapsed times (in days) for SATD removals per project in Figure 3.5. Similarly, Figure 3.6 presents the distribution of the permanence time (in days) for SATD instances that remain in the projects. The distributions in in Figure 3.5 indicate that when SATD is removed from the studied projects, this mostly happens within a year of the debt's introduction. On the other hand, Figure 3.6 indicates that SATD instances that have not been removed remain in source code for long periods of time, often years after the debt's introduction.

3.3.7 Collection SATD Instance Metrics

Having narrowed down to the debt instances to focus in our study, the next step was to collect the set of effort repayment metrics. In the previous subsection we recapped how we capture the time of removal from SATD instances. Below, we first explain the collection of product metrics from SATD instances that will be used to measure SATD interest gained and compound interest rate. Then, we explain how we obtain the measures of effort in commented words, and change proneness (likelihood of change).

Product Metrics

Similar to the approach taken by Kamei et al. (2016) to extract metrics from SATD methods, we leveraged the capabilities of SciTools Understand (SciTools, 2018b) to extract 21 complexity and volume metrics available at the method level from each of the versions of our SATD instances. A

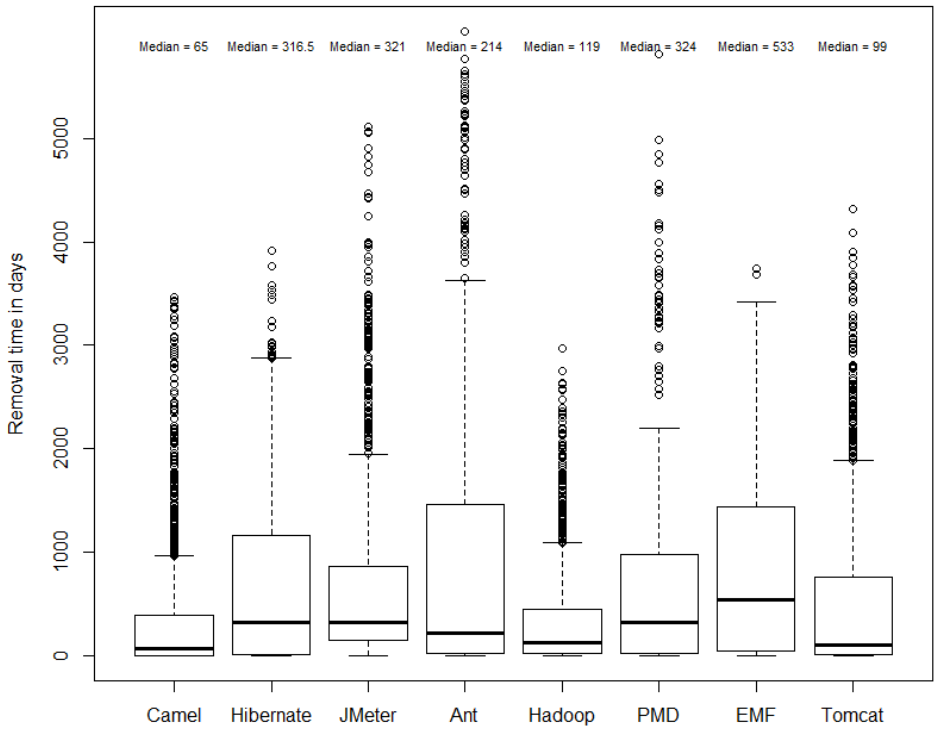

Figure 3.5: Distribution of removal times for the studied SATD instances per project.

description of the list of extracted product metrics can be found in Appendix B.1, Table B.1. We used the API version of Understand to create UDB files (Understand database file) containing all the entities and dependencies of an SATD file (i.e., a file that contains SATD). The UDB files were then analyzed to query and extract the method-level metrics. In this step, we use the declaration statement that a SATD belongs to, i.e., a method, to capture only the metrics of debt instance, and not the rest of the file. However, not all SATD instances could be linked to a method; often, SATD comments were belong to other Java constructs, containers, such as nested classes, interfaces or declaration statements. Up to this point, we have discarded SATD instances for: i) being removed by chance (deletion of files), and ii) not being able to collect metrics from them. We report the total amount of SATD instances discarded by both preceding reasons in Table 3.4. Since the product metrics we extract in this step are required for the analysis, and to other metrics we collect in proceeding steps depend on them, we focus on these instances only. Therefore, we study 18,242 SATD instances,

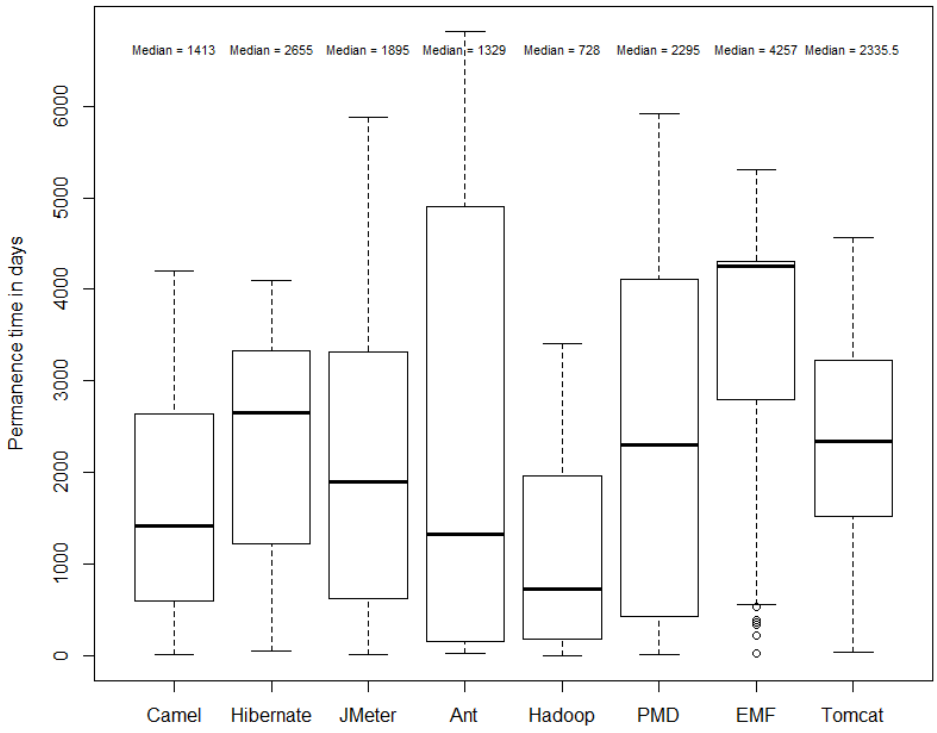

Figure 3.6: Distribution of the permanence of remaining SATD instances per project.

which corresponds to 71.2% of the total debt found in all projects.

Effort in Commented Words

Certainly, from the observations in our preliminary survey to developers and from our own rationale, the effort required to address a SATD instance plays a major role for its resolution and prioritization in a project. However, effort is particularly hard to measure for debt instances. This can be also exemplified in our developer survey answers. When we asked developers how do they estimate effort, the answer where not concrete and pointed at different ways to obtain an effort estimate. In previous studies that worked on prioritizing the resolution of SATD, Mensah et al. (2016, 2018) proposed a measure for effort estimation expressed in terms commented LOC. As found in their investigation after manually classifying SATD comments that were vital for a project (i.e., those that should be prioritized), developers would need to address 10 to 25 commented LOC

Table 3.4: Detailed amount of studied SATD instances.

Project	(#) SATD instances	Discarded		Studied	
		#	%	#	%
Camel	3,456	806	23.3	2,650	76.7
Hibernate	4,514	1,618	35.8	2,896	64.2
JMeter	2,599	692	26.6	1,907	73.4
Ant	2,193	871	39.7	1,322	60.3
Hadoop	5,329	1,231	23.1	4,098	76.9
PMD	1,694	746	44.0	948	56.0
EMF	1,695	594	35.0	1,101	65.0
Tomcat	4,132	812	19.7	3,320	80.3
Total	25,612	7,370	28.8	18,242	71.2

(CLOC) per SATD file to repay them. However, this measure was generated as an average per project and can be counterintuitive. In our study, we are interested in measuring the effort for each particular SATD instance, so we used the word count of the commented text. This measure was taken to have a more individual estimation of effort based on the size of SATD comments. Given that SATD comments change over time, we extracted this metric from every version of a SATD instance. From now on, we refer to this metric simply as *Effort in words* for simplification.

Change Proneness

Another important measure of SATD repayment effort is the likelihood of change of files with SATD. This is because the more it changes, the more likely it will sprout dependencies and potentially be more complex and costly to repay in the future. As done in previous work (Bieman et al., 2003; Khomh et al., 2009), we measure the change proneness of an SATD instance as the number of file changes in the repository (considering past renames) until the date of the analyzed file version. For consistency, the change proneness metric was taken for all SATD versions.

3.3.8 Measurement of SATD Interest

When SATD is introduced, it evolves over time, potentially increasing or reducing the severeness of the technical debt. Kamei et al. (2016) investigated this phenomena and referred to it as *SATD interest*, which reflects the additional cost of difficulty in repaying the debt. Intuiting that the complexity of a piece of code increases when it becomes harder to repay later (i.e., incurred interest),

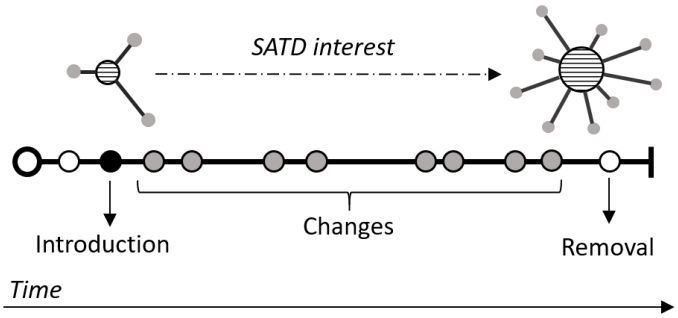

Figure 3.7: Theoretical visualization of positive interest in SATD.

the authors used complexity metrics as a proxy to quantify SATD interest. In particular, they proposed the use of LOC and Fan-In (amount of dependencies of a function) as measures to quantify the interest of SATD. These two metrics were chosen after finding that other complexity and volume metrics to be highly correlated with LOC, except Fan-In. Figure 3.7 presents a theoretical visualization of the interest growth of a SATD instance from its introduction to removal points in time. The size of the node with horizontal stripes represents the LOC of a SATD instance, while the amount of edges to outer smaller nodes represents its dependencies.

We measured the Spearman correlation (Myers & Sirois, 2004) between the complexity and volume metrics we extracted and had a similar observation to Kamei et al. (2016), noticing they were also correlated. We also noticed that LOC and Fan-In were had a slight correlation in our dataset. Despite this, we kept both metrics as they measure two elements of interest in our study: LOC serves as an indicator for size of the debt to address, while Fan-In indicates the amount of dependencies of the debt. More details on the correlation of these metrics can be found in Section 3.4. We computed the interest measures in terms of LOC and Fan-In with our collected data as proposed in (Kamei et al., 2016), obtaining positive and negative interest rates per debt instance. Since the resulting rate from the direct comparison of values between the introduction and removal times, we refer to it as *simple interest*. Table 3.5 shows the incurred interest of studied SATD instances in terms of LOC and Fan-In per project. In contrast to the finding of Kamei et al. (2016), we find that a lower amount of SATD instances incur positive interest, both in terms of LOC (16.2%) and Fan-In (10%) on average. The majority of instances do not have any change (75.7% LOC and 85.3% Fan-In), and the remaining minority present a negative interest (4.7% to 8.1%).

Table 3.5: Incurred SATD interest in LOC and Fan-In.

Project	Measure	Positive	No change	Negative
Camel	LOC	14.9%	78.2%	6.9%
	Fan-In	8.3%	88.9%	2.8%
Hibernate	LOC	19.5%	74.7%	5.8%
	Fan-In	9.6%	86.2%	4.1%
JMeter	LOC	16.8%	68.0%	15.2%
	Fan-In	12.8%	80.8%	6.4%
Ant	LOC	14.6%	74.7%	10.7%
	Fan-In	11.6%	81.8%	6.6%
Hadoop	LOC	22.5%	69.4%	8.1%
	Fan-In	14.3%	79.7%	6.0%
PMD	LOC	18.4%	74.9%	6.8%
	Fan-In	8.2%	88.4%	3.4%
EMF	LOC	6.9%	90.6%	2.5%
	Fan-In	3.9%	93.7%	2.4%
Tomcat	LOC	16.1%	74.9%	9.0%
	Fan-In	11.3%	82.8%	5.9%
Average	**LOC**	16.2%	75.7%	8.1%
	Fan-In	10.0%	85.3%	4.7%

3.3.9 Measurement of Compound Interest Rate

When seen along the metaphor of technical debt, the interest generated by SATD instances is tied to how they evolve over time. Therefore, the rate of such interest will vary based on the nature of the debt itself and the effects it has on the code base. However, this is not entirely reflected by the measure of simple interest because it does not consider the speed of the debt's evolution or the amount of times it evolves. Translated to the timeline of a repository, inspecting how fast and frequently a SATD instance changes is important to determine those that should be prioritized. Kamei et al. (2016) also pointed at the need of investigating time as an element to better quantify SATD interest.

Continuing the metaphor of technical debt and translating the aforementioned problem to a financial perspective, we propose the usage of a *compound interest rate* (Lewin, 1970). The rationale behind this is that the resulting rate of a compound interest will also reflect the number of times a SATD instance has changed over its timespan, i.e, the time it has remained in the project. Therefore, the resulting rate is a construct of recalculating (compounding) the interest every time a SATD instance is changed. In contrast to simple interest where changes done in-between introduction and

removal times are ignored (see Figure 3.7), a compound rate will better represent the speed and frequency of SATD evolution.

As an example, consider the parallel introduction of two SATD instances in methods M_1 and M_2, which after a year of development were changed 5 times. By the end of the year, we observe that M_1 has grown in Fan-In and LOC to twice its original metric value, while noticing M_2 had the same growth but it only occurred in the last 3 months. We can consider M_2 to be a more urgent debt to address because it is generating interest at a faster pace than M_1. It is also more likely that M_2 will keep worsening the debt in the future. Here, the simple interest will be the same for both SATD instances, but the compound interest rate for M_2 will be different. Having the starting and final metric values in LOC and Fan-In for a given SATD, its timespan and frequency of change, we can apply the standard compound interest formula (Lewin, 1981), defined as:

$$A = P\left(1 + \frac{r}{n}\right)^{n*t}$$

Where *P* is the principal amount value; *A* is the future value, including interest; *n* is the compounding frequency; *t* is the length of time in which the interest is applied; and *r* is the interest rate. Since we are trying to obtain the compound interest rate, i.e., how quickly it changes of SATD and not the interest value, we can derive from the above formula and solve for *r* with the following:

$$r = n * \left(\left(\frac{A}{P}\right)^{\frac{1}{n*t}} - 1\right)$$

In the context of SATD, *P* is the value in LOC or Fan-In for a debt instance at its introduction time; *A* is the new value in LOC or Fan-In after a change. *n* is the number of commits or changes that affected the debt instance in a timespan; and *t* is the timespan in days of a debt instance since its introduction. We use days as our time measure because the changes we observe in the repositories mostly happen daily. The result from applying this formula is a change-compounding interest rate *r* that reflects the changes in the timespan of an SATD instance. Notice that the magnitude of *r* will change based not only on the LOC or Fan-In values, but also on the amount of changes that the debt went through and the time it has remained in the system. Thus, it will better reflect the evolution of a SATD measuring how fast and frequently the effort to repay the debt is changing. In the case the

amount of LOC or Fan-In values decrease over time, i.e., the SATD incurs negative interest, r will accordingly be negative, indicating the debt has shrunk.

Because this interest rate is measured with additional data from the history of a debt instance, which is different in each SATD case, it can be used as a more accurate metric to estimate SATD repayment effort than with a simple interest. The r values in terms of Fan-In and LOC were computed for all versions of SATD instances in our dataset.

3.3.10 Collection of Historical Metrics

Having obtained different metrics of SATD instances and studied what happened to them over time, our next step was to collect historical change metrics that reflect different features of files at the point where SATD was introduced into them. The metrics we selected have been used in previous work as indicators for defect proneness, and for fault and risk prediction (Hassan, 2009; Kamei et al., 2013; Matsumoto, Kamei, Monden, Matsumoto, & Nakamura, 2010; Mockus & Weiss, 2000). We are interested to see if they can also serve as indicators for the effort required to repay SATD. We used git to mine a some of these change history metrics, and leveraged the capabilities of Commit Guru to obtain others that are built into the tool's features. Commit Guru is a public tool for predicting risky commits in software repositories (Rosen, Grawi, & Shihab, 2018). We collected the following metrics at the introduction commit of each SATD instance; we provide their extraction source and description:

- **Contributors** (git); the number of unique developers that have contributed changes to the SATD file[3] (Matsumoto et al., 2010).
- **Author experience** (git); number of commits done by a developer in a repository up to a given point in time (Kamei et al., 2013).
- **Bug-fixing commits** (Commit Guru); the number of bug-fixing commits (i.e., commits that fixed a bug) that a file has gone through (Rosen et al., 2015).
- **Entropy** (Commit Guru); a measure for the distribution of change across the files modified in a commit (Hassan, 2009).

[3]Unique developers were found as described in Section 3.3.6 and 3.3.2.

- **File churn** (git); number of added and deleted lines in a modified file (Munson & Elbaum, 1998).
- **Commit churn** (Commit Guru); the total number of added and deleted lines from all files modified in a commit (Nagappan & Ball, 2005).
- **Modified directories** (Commit Guru); number of different directories modified in a commit (Mockus & Weiss, 2000).
- **Modified files** (Commit Guru); number of files modified in a commit (Nagappan, Ball, & Zeller, 2006).
- **Modified subsystems** (Commit Guru); number of different subsystems modified in a commit (Mockus & Weiss, 2000).

More details on the implementation of these metrics can be found in (Kamei et al., 2013), while details on Commit Guru itself and how it identifies bug-fixing commits can be found in (Rosen et al., 2015). We analyzed all our projects with Commit Guru and extracted its metrics from the downloadable database dump it generates. This data was then combined with those metrics extracted from git for the commits that introduced SATD. The 9 metrics described above conform the set of explanatory variables in our study.

3.4 Results

After completing the data collection for the study, we measured the intercorrelation of dependent and independent variables using Spearman correlation coefficients (Myers & Sirois, 2004) to make sure they were not giving us the same information and thus, they measure different things about the SATD instances. The measurements of intercorrelations between both sets of variables will help us to spot those that can be removed to simplify our models. In our study, the dependent or response variables are the SATD repayment effort metrics taken at the latest found version of a SATD instance. On the other hand, the independent variables are the set of historical metrics extracted at the point of SATD introduction. The intercorrelations between our response variables surfaced

that most of the SATD effort estimation metrics we selected are providing different information; the correlations were consistent among projects. The only exception was between the metrics of simple interest in terms of LOC and Fan-In, which were positively correlated in 6 out of 8 studied projects (coefficients of: 0.86 in Camel, 0.51 in Hibernate, 0.66 in Ant, 0.68 in Hadoop, 0.66 in PMD, 0.8 in Tomcat). Table 3.6 shows the intercorrelation of response variables in all projects, and highlights the significant correlation found (0.62) between SATD interest in terms of LOC and Fan-In. Despite this observation, we kept both metrics as they measure two elements of importance in our study: LOC serves as an indicator for size and complexity of debt, while Fan-In indicates the amount of dependencies it has. Interestingly, we found no significant correlation between the measures of simple and compound interest in terms of LOC or Fan-In, this tells us that these SATD effort repayment metrics are different.

The intercorrelations between historical metrics showed a different case, we observed that often several metrics were correlated, but with differences in every project. We are interested in keeping a healthy amount of independent variables as we are investigating which can be good indicators. Therefore, we looked at intercorrelation coefficients of each project and only discarded variables that were highly correlated (≥ 0.8). Table 3.7 lists the explanatory variables that we kept for each project. Overall, we kept between 5 to 7 variables that were not highly correlated.

Table 3.6: Intercorrelation coefficients between response variables.

	Effort in words	Interest Fan-In	Interest LOC	Comp. Interest LOC	Comp. Interest Fan-In	Change Proneness	Removal Time (days)
Effort in Words	1.00	-0.01	-0.01	-0.02	0.00	-0.04	0.07
Interest Fan-In	-0.01	1.00	**0.62**	0.13	0.17	0.05	0.07
Interest LOC	-0.01	**0.62**	1.00	0.17	0.05	0.06	0.03
Compound Interest LOC	-0.02	0.13	0.17	1.00	0.04	0.01	-0.02
Compound Interest Fan-In	0.00	0.17	0.05	0.04	1.00	0.01	-0.04
Change Proness	-0.04	0.05	0.06	0.01	0.01	1.00	-0.02
Removal Time (days)	0.07	0.07	0.03	-0.02	-0.04	-0.02	1.00

Table 3.7: Explanatory variables kept per project.

	Camel	Hibernate	JMeter	Ant	Hadoop	PMD	EMF	Tomcat
Contributors	✔	✔	✔	✔	✔	✔	✔	✔
Bug-fixing commits	✘	✘	✔	✘	✔	✔	✔	✘
Author experience	✔	✔	✔	✔	✔	✔	✔	✔
Entropy	✔	✔	✘	✘	✘	✘	✔	✘
File churn	✔	✔	✔	✔	✔	✔	✔	✔
Commit churn	✔	✘	✘	✔	✘	✔	✘	✘
Modified files	✔	✘	✔	✔	✘	✔	✔	✔
Modified directories	✘	✘	✘	✘	✔	✘	✘	✘
Modified subsystems	✔	✔	✔	✔	✔	✔	✔	✔

RQ1: ***Is there a consistent SATD repayment effort metric that can be modeled using historical data?***

We built linear regression models for every SATD repayment effort metric using the kept independent variables per project, i.e., those that were not intercorrelated to others as seen in Table 3.7. To do so, we first plotted the distribution of all kept independent variables per project to have a visualization of their distributions. This aided us to identify variables that had suffered from a high skewness and kurtosis in the dataset. The complete set of these distribution plots for our independent variables can be found in Appendix B, Sections B.3 to B.10. Noticeably, our set of independent variables mostly suffered from a positive skewness, and often from an excess of kurtosis. We applied a log transformation to variables that fell in either case.

To build the linear regression models, we used the *lm* command in the R statistical software (The R Foundation, 2018). We looked at the statistical significance of each independent variable to the model using their p-values and removed those that were not contributing to the model of a response variable. In parallel, to achieve a good fit for the models, we looked at the coefficient of determination (R-squared value) of the linear regression model after each iteration of removing an independent variable that was not significant. A similar approach to build logistic regression models has been used in previous software engineering work (Shihab, Jiang, Ibrahim, Adams, & Hassan, 2010). We use the R-squared coefficient of determination because it quantifies the fit of the model. A higher percentage R-squared value indicates a better fit on a linear regression model. We repeated this process with all the independent variables for each response variable and for each studied project. The complete set of resulting linear models and their respective R-squared values can be seen in Tables 3.8 to 3.15.

Previous software engineering work has reported equivalent R-squared values ranging between 11.2% to 25.2% and higher as indicator of useful logistic regression models (Shihab et al., 2010). In our scenario, we use a similar range to determine models with good fit. We highlight significant R-squared values in Tables 3.8 to 3.15. We find that among all the response variables we evaluate, the only one that is consistent among all projects with R-squared values ranging from 43% to 82% is change proneness. The models built for removal time in days of SATD also present good results for 5 out of 8 projects, with R-squared values between 16% to 40%. Effort in words showed an

R-squared value of 29% in EMF, and 10% in Ant. The remaining response variables had much lower R-squared values that are indicate a weak relationship to the built models.

The complete correlation plots of these variables that support the built models can be seen in Appendix B, Sections B.3 to B.10.

> ***Two SATD repayment effort metrics can be consistently modeled using historical data: Change Proneness (43% to 82% R-squared values) in all projects and Removal Time (16% to 40% R-squared values) in days in 5 out of 8 projects.***

RQ2: ***What are the best indicators for SATD repayment effort metrics?***

Continuing on the findings of RQ1, we investigated which of the independent variables can be used as early indicators of SATD instances that should be repaid. To do this, we looked at the frequency that different independent variables contributed to the linear models highlighted in RQ1 with significant values. In the resulting models for change proneness, the number of contributors to a file was significant in 7 out of the 8 studied projects, while the amount of file churn, modified subsystems, and bug-fixing commits were significant indicators in 4 out of 8 projects. In the models built for SATD removal time (in days), the number of contributors, file churn, modified subsystems and author experience were significant appearing in 3 out of the 5 models with high R-squared values. Furthermore, for the response of amount of commented words (effort in words), the number of contributors, file churn, and modified subsystems appeared in the 2 models with significant R-squared values. By counting the independent variables that were recurrent across all models built for the response variables with strong R-squared values, we identified that the number of contributors, file churn, and author experience taken at the introduction point of SATD were the most frequent and significant contributors to the models. Therefore, they are the top 3 contributing indicators for SATD repayment effort metrics. This implies these historical metrics can be used as good early indicators of SATD instances that are important to repay.

Two other indicators that followed the top 3 contributing to the resulting models were: the number of bug-fixing commits that a file with SATD has gone through. This indicator was also a significant contributor in the models build for change proneness and removal time (in days), however

less frequent than those indicated above as the top contributors. This last independent variable was not significant for the 2 resulting linear models for effort in words. We can notice a trend of the best indicators for SATD repayment effort being those that look into the history of the particular file where SATD was introduced. In contrast, other indicators, such as number of modified directories in a commit, commit churn, or entropy are all metrics that consider changes to files other than where the SATD was added, but that were part of the same commit. These metrics that consider extrinsic characteristics of a SATD files did not contribute to the built models as well as those measuring intrinsic features. In particular, we observed that entropy and the number of modified directories were not good indicators. In the case of entropy, it only appeared in 1 of the models for change proneness, and 2 of the models of removal time, while the amount of modified directories never had a significance on any of the linear regression models. This suggests that is more valuable to focus on historical data at the SATD file level rather than looking at indicators that measure features of the complete change or commit when the SATD was introduced.

The goal for this research question was to find the best indicators for SATD repayment effort metrics. Naturally, the amount of indicators that result of this investigation is key because models with many independent variables are more complex to explain, and more expensive to apply. The removal of independent variables that were intercorrelated plus the approach taken to build the linear regression models resulted in our models being composed of considerably less variables than our starting set of 9 independent variables. For the models built for change proneness and removal time in days as dependent variables, we observe the average amount of independent variables is 4 (between 2 and 5) in both cases. This means the resulting models are not only easier to explain, but the data needed to implement them is easier to collect. For example, the top 3 indicators for SATD repayment effort metrics can be easily retrieved by querying the history of a repository until the point of SATD introduction, and the change history of the file with added debt.

The number of contributors, file churn, and author experience at the time of SATD introduction are the top 3 early indicators for debt instances that are important to repay. These indicators pertain to historical data of the file where SATD is introduced.

Table 3.8: Camel models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ AuthorExperience + Entropy + log(1+CommitChurn) + ModifiedSubsystems`	0.01606
Effort in words	`EffortInWords ~ Contributors + AuthorExperience + Entropy + CommitChurn + ModifiedFiles`	0.0201
Interest Fan-In	`InterestFanIn ~ log(1+Contributors) + log(1+AuthorExperience) + log(1+Entropy) + log(1+FileChurn) + CommitChurn + ModifiedFiles + log(1+ModifiedSubsystems)`	0.001199
Interest LOC	`InterestLOC ~ Contributors + log(1+AuthorExperience) + Entropy + CommitChurn + ModifiedFiles`	0.002853
Compound Interest Fan-In	`CompoundedFanIn ~ AuthorExperience + Entropy + FileChurn + CommitChurn + ModifiedSubsystems`	0.01087
Compound Interest LOC	`CompoundedLOC ~ log(1+Contributors) + log(1+AuthorExperience) + Entropy + log(1+FileChurn) + log(1+CommitChurn)`	0.03587
Change Proneness	`ChangeProness ~ Contributors + FileChurn + log(1+ModifiedSubsystems)`	**0.7757**

Table 3.9: Hibernate models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ log(1+Contributors) + log(1+AuthorExperience) + Entropy + FileChurn + ModifiedSubsystems`	**0.4192**
Effort in words	`EffortInWords ~ AuthorExperience + log(1+Entropy) + FileChurn + ModifiedSubsystems`	0.004341
Interest Fan-In	`InterestFanIn ~ log(1+Contributors) + log(1+AuthorExperience) + ModifiedSubsystems`	0.004682
Interest LOC	`InterestLOC ~ Contributors + AuthorExperience + log(1+Entropy) + log(1+FileChurn) + log(1+ModifiedSubsystems`	0.000941
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+Contributors) + log(1+Entropy) + FileChurn`	0.001656
Compound Interest LOC	`CompoundedLOC ~ Contributors + log(1+AuthorExperience) + log(1+Entropy) + FileChurn + log(1+ModifiedSubsystems)`	0.002742
Change Proneness	`ChangeProness ~ Contributors + FileChurn + ModifiedSubsystems`	**0.4375**

Table 3.10: JMeter models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ BugFixingCommits + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles)`	0.08007
Effort in words	`EffortInWords ~ log(1+BugFixingCommits) + log(1+Contributors) + ModifiedFiles + ModifiedSubsystems`	0.04623
Interest Fan-In	`InterestFanIn ~ BugFixingCommits + Contributors + AuthorExperience + log(1+ModifiedFiles) + log(1+ModifiedSubsystems)`	0.01064
Interest LOC	`InterestLOC ~ log(1+BugFixingCommits) + log(1+Contributors) + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems)`	0.001849
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+BugFixingCommits) + Contributors + AuthorExperience + ModifiedFiles + ModifiedSubsystems`	0.008001
Compound Interest LOC	`CompoundedLOC ~ log(1+Contributors) + FileChurn + log(1+ModifiedFiles)`	0.00352
Change Proneness	`ChangeProness ~ log(1+BugFixingCommits) + Contributors + log(1+AuthorExperience) + FileChurn + log(1+ModifiedFiles)`	**0.5791**

Table 3.11: Ant models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ log(1+Contributors) + log(1+AuthorExperience) + CommitChurn + log(1+ModifiedSubsystems)`	**0.1611**
Effort in words	`EffortInWords ~ Contributors + FileChurn + log(1+CommitChurn) + ModifiedFiles + log(1+ModifiedSubsystems)`	**0.1049**
Interest Fan-In	`InterestFanIn ~ Contributors + log(1+AuthorExperience) + ModifiedFiles + log(1+ModifiedSubsystems)`	0.02842
Interest LOC	`InterestLOC ~ log(1+AuthorExperience) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedSubsystems)`	0.02439
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+Contributors) + AuthorExperience + log(1+FileChurn) + ModifiedSubsystems`	0.00153
Compound Interest LOC	`CompoundedLOC ~ log(1+Contributors) + log(1+AuthorExperience) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems)`	0.004533
Change Proneness	`ChangeProness ~ Contributors + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems)`	**0.8271**

Table 3.12: Hadoop models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ BugFixingCommits + AuthorExperience + ModifiedDirectories + ModifiedSubsystems`	0.04387
Effort in words	`EffortInWords ~ log(1+Contributors) + ModifiedDirectories + ModifiedSubsystems`	0.01269
Interest Fan-In	`InterestFanIn ~ BugFixingCommits + Contributors + log(1+ModifiedDirectories) + log(1+ModifiedSubsystems)`	0.006613
Interest LOC	`InterestLOC ~ BugFixingCommits + Contributors + ModifiedDirectories + ModifiedSubsystems`	0.0448
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+Contributors) + log(1+AuthorExperience) + log(1+FileChurn) + ModifiedSubsystems`	0.000629
Compound Interest LOC	`CompoundedLOC ~ AuthorExperience + FileChurn + ModifiedDirectories + ModifiedSubsystems`	0.00152
Change Proneness	`ChangeProness ~ BugFixingCommits + Contributors`	**0.6952**

Table 3.13: PMD models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ log(1+BugFixingCommits) + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles)`	**0.2741**
Effort in words	`EffortInWords ~ AuthorExperience + FileChurn + ModifiedFiles + ModifiedSubsystems`	0.01063
Interest Fan-In	`InterestFanIn ~ log(1+BugFixingCommits) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedSubsystems)`	0.06831
Interest LOC	`InterestLOC ~ log(1+BugFixingCommits) + Contributors + AuthorExperience + FileChurn + log(1+ModifiedFiles)`	0.05768
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+Contributors) + AuthorExperience + log(1+FileChurn) + log(1+ModifiedSubsystems)`	0.02168
Compound Interest LOC	`CompoundedLOC ~ log(1+BugFixingCommits) + AuthorExperience + FileChurn + ModifiedFiles`	0.005869
Change Proneness	`ChangeProness ~ BugFixingCommits + Contributors + CommitChurn + log(1+ModifiedFiles)`	**0.5412**

Table 3.14: EMF models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~log(1+BugFixingCommits) + AuthorExperience + log(1+FileChurn) + log(1+ModifiedFiles)`	**0.2952**
Effort in words	`EffortInWords ~ BugFixingCommits + Contributors + log(1+FileChurn) + log(1+Entropy) + log(1+ModifiedFiles) + ModifiedSubsystems`	**0.2904**
Interest Fan-In	`InterestFanIn ~ BugFixingCommits + log(1+FileChurn) + Entropy + ModifiedSubsystems`	0.01201
Interest LOC	`InterestLOC ~ BugFixingCommits + log(1+FileChurn) + Entropy + ModifiedSubsystems`	0.06
Compound Interest Fan-In	`CompoundedFanIn ~ log(1+BugFixingCommits) + log(1+Contributors) + AuthorExperience + Entropy + log(1+ModifiedFiles)`	0.003876
Compound Interest LOC	`CompoundedLOC ~ log(1+Contributors) + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems)`	0.04082
Change Proneness	`ChangeProness ~ BugFixingCommits + Contributors + log(1+FileChurn) + log(1+Entropy)`	**0.7927**

Table 3.15: Tomcat models and R-squared values.

Response variable	Model	R-squared
Removal time (days)	`RemovalTimeInDays ~ log(1+Contributors) + log(1+AuthorExperience) + log(1+ModifiedSubsystems)`	**0.2672**
Effort in words	`EffortInWords ~ AuthorExperience + log(1+FileChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems)`	0.04606
Interest Fan-In	`InterestFanIn ~ log(1+AuthorExperience) + log(1+FileChurn) + ModifiedFiles + log(1+ModifiedSubsystems)`	0.002857
Interest LOC	`InterestLOC ~ log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems)`	0.002955
Compound Interest Fan-In	`CompoundedFanIn ~ Contributors + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedSubsystems)`	0.001773
Compound Interest LOC	`CompoundedLOC ~ log(1+Contributors) + log(1+AuthorExperience) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems)`	0.001419
Change Proneness	`ChangeProness ~ Contributors + AuthorExperience + log(1+ModifiedFiles) + ModifiedSubsystems`	**0.6682**

3.5 Threats to Validity and Limitations

3.5.1 Internal Validity

Our study detects SATD by inspecting source code comments. We did not analyze the source code in context of the SATD we identify to confirm the existence of technical debt. A recent study has considered this and focused on analyzing source code to recommend adding absent SATD comments (Zampetti, Noiseux, et al., 2017). However, the approach was tested with design SATD only. In our work we focus on all SATD, regardless its type or classification. Therefore, it might be the case that technical debt we detected was removed from a piece of code, but its comment was not amended, or vice versa. On this regard, previous work has found that source code and comments do co-evolve; comment are changed in the same revision as their corresponding piece of code 97% of the time (Fluri et al., 2007a).

We acknowledge we might not have selected all metrics that can be used to estimate the effort required to repay SATD, and equally, all metrics that could be retrieved from historical data as indicators. As mentioned earlier in Section 3.3.1, to dampen this threat, we selected a comprehensive set of metrics that can be obtained from source code and change history information, and that we could support with previous work. The preliminary online survey was sent to 200 developers on what they consider to prioritize SATD had a 9.5% response rate (19 answers in total). Although this appears to be a small rate, it is comprehensive among similar survey work in software engineering (Singer et al., 2008). Even with this in mind, we considered the responses received with our own conjectures and not as the rule of thumb, supporting our decision with metrics used in prior SATD studies. While selecting exploratory variables, our rationale was to settle on a set of metrics that could be early indicators of technical debt that should be repaid. Although we leveraged most change history and defect predictions metrics that could be extracted from our dataset, we were limited to pick those that would not bias our models and that could be supported by previous work. Still, we do not claim to have evaluated all metrics that can be used for this purpose.

3.5.2 Construct Validity

One key factor in our study is the detection of SATD instances across the change history of the studied projects. This presents the threat of not detecting all SATD instances, or miss identifying comments that are not actual technical debt (false positives). To mitigate the basis of this, we rely on the SATD detector tool, which to date is the best performing SATD detection approach proposed in literature. Similarly, across our case study setup, we relied on well know tools that have been used in previous software engineering empirical studies, such as: git, srcML, and SciTools Understand. Naturally, our study is tied to the false positives, errors or limitations from these tools and approaches. An example of this happened when we were unable to map all SATD comments to methods or functions using Understand to extract product metrics from the debt instances. To manage this, we reported and discarded the SATD instances facing this limitation from the study. A step that did not rely in external tools was our mapping of detected SATD comments to others in the same source code context and across history. In said step, we needed to decide on a threshold for string similarity to determine if comments belonged to the same debt instance or not. To mitigate misclassification, we inspected the distribution of all similarity scores we computed, and decided to set the threshold at ≥ 0.8 by looking at our data. This resulted in a threshold set higher than previous work that evaluated comment similarity, which in our scenario was beneficial.

3.5.3 External Validity

We studied SATD in 8 software systems, selecting those with different sizes and characteristics of interest, such as: amount of comments, changes in their repositories and contributors. However, all of these projects are open-source and written in Java. Thus, our findings may not generalize to projects written in other programming languages or to proprietary software. Several of the tools and approaches we implement throughout our study are tailored for Java as well; this is a limitation that requires the attention of the research community in the future. Most (if not all) findings and work done in the area of SATD are still restricted to Java projects. Lastly, we only targeted repositories with sufficient source code comments; therefore, our findings may not generalize to systems that are less commented.

3.6 Conclusions and Future Work

With the motivation gained from Chapter 2, this study worked towards the repayment of SATD. We focused on finding indicators of SATD instances that will become more difficult or costly to repay over time, targeting the earliest point in time, i.e., when SATD is added into a system. We began by conducting an online survey to gain insight from developers on the elements they consider to prioritize the resolution of SATD. Based on the responses from developers, metrics used by previous work in the area, and our own conjectures, we selected 7 different SATD repayment effort metrics. As a set of indicators, we selected 9 change history and defect prediction metrics that we extracted from historical data to serve as our exploratory variables. In total, we collected data from 18,242 unique SATD instances (and their change history) detected in 5,352,994 source code comments, and across 121,558 commits in 8 software repositories. Linear regression models were built for every SATD repayment effort metric as response and using the historical metrics as explanatory variables, this was repeated for each project independently. We found that two SATD repayment effort metrics could be modeled consistently: change proneness in all projects, and SATD removal time (in days) in 5 out of 8 studied projects. We investigated what were the best early indicators for SATD that should be repaid and found the top 3 were: number of developers that contribute to a SATD file, amount of file churn, and the experience of the developer who introduced the debt. We noticed that these indicators relate to intrinsic characteristics of the file where SATD is added, and that indicators that relate to extrinsic characteristics were not useful indicators. This suggests that when using historical data to estimate SATD that should be repaid, is more valuable to look for intrinsic indicators of SATD at the file level, rather than at the change or commit level.

On the bigger picture towards SATD repayment, the questions of what SATD should be repaid, and which debt instance should be prioritized remain open. In future work, we aim to continue on this research track, and as an immediate next step, we plan on investigating the usefulness of using measures of SATD interest, in particular, the compound interest rate proposed in this work. We envision an approach to rank and recommend which SATD instance should be addressed first among those that remain in a system and are candidates for repayment. Thus, we plan to study on a way to prioritize the resolution of SATD.

Chapter 4

Summary, Contributions, and Future Work

4.1 Summary of Addressed Topics

This thesis focused on working towards the repayment of Self-Admitted Technical Debt (SATD). First, we performed and in-depth survey of empirical research in the area Chapter 2. We used the study by Potdar and Shihab (2014) as our cornerstone and applied a snowball approach to find related work, complementing our lookup with academic search engines. In this literature review, we surfaced how the research community has evolved detection and classification approaches from manual inspection to advanced automated techniques. Similarly, we observed how research has progressed to better understand the SATD phenomenon, how it is introduced and removed from software repositories and its implications on software maintenance. By highlighting gaps and opportunities in the surveyed work, we spotted a lack of studies dedicated to the repayment and management of SATD, which is one of the goals of studying this phenomena.

Our literature review also pointed at several of the challenges to overcome in the area, presenting various promising avenues for future SATD work and actionable calls to action for the research community. Furthermore, we compiled publicly available tools, techniques and datasets that can be used as a foundation to advance the state of the art on SATD and encourage the submission of novel ideas.

In Chapter 3, we focused on working towards the repayment of SATD using the knowledge gained from Chapter 2. We began by conducting an online survey to developers to gain insight on the elements they consider when prioritizing SATD. Then, based on the responses we obtained and previous work on SATD repayment, we conducted an empirical study, in which we focus on finding indicators for SATD instances that will become more difficult to repay over time. 7 different SATD repayment effort metrics were selected as our response variables, while 9 change history and defect prediction metrics were chosen to serve as our explanatory variables.

In total, we collected data from 18,242 unique SATD instances (and their change history) detected in 5,352,994 source code comments, and across 121,558 commits in 8 software repositories. We built linear regression models for each of our response variables evaluating our set of exploratory variables in each studied project. The results of this investigation exposed two responses that could be modeled consistently, which are: change proneness in all projects, and removal time (in days) in 5 out of 8 projects. To find the best indicators for SATD repayment effort, we looked at those that contributed most to the built models of interest, and found the top 3 indicators to be: the number of developers that contribute to a SATD file, amount of file churn, and the experience of the developer who introduced the SATD instance. In contrast to other indicators that were not as useful, the best found indicators measured intrinsic features of SATD files. This suggests that when using indicators from historical data for SATD that should be repaid, is more valuable to look for intrinsic indicators at the file level, than at the change or commit level.

4.2 Contributions

The main contributions presented in this thesis are:

- An in-depth literature survey on Self-Admitted Technical Debt research work. This thesis presents a compilation of the most recent publications in the area of SATD, highlighting gaps and opportunities as potential research avenues for future work. We overview challenges in the area and include a set of explicit calls to action that can be implemented by researchers to overcome them.
- A compilation of publicly available tools, approaches, techniques and online datasets that

from surveyed work that can be used to extend SATD research.

- An empirical study of the change history of 18,242 unique SATD instances detected with state of the art approaches in 8 software systems investigating the possibility of using change history and defect prediction metrics as early indicators of SATD that should be addressed.
- We propose the use of a compound interest rate to measure the evolution of a SATD that allows developers to estimate the frequency and speed at which a debt instance gains interest.
- Our resulting dataset from the empirical study, as well as the questions and responses of the online survey to developers are made publicly available to facilitate and promote further research in the area o SATD. The dataset contains a wide set of product and process metrics of detected SATD instances (in all their versions) from eight open-source projects.

4.3 Future Work

Certainly, the materials presented on this thesis leave room for vast future research on SATD, particularly on SATD repayment. In Chapter 2, we dedicated Section 2.4 solely to discuss future work on SATD, mentioning gaps in current work, research opportunities, listing challenges to overcome, and even explicitly pointing at actionable calls to action. Although we could pursue any of the opportunities for future work proposed in an immediate future, its extensiveness surpasses our capability. Therefore, this must be addressed in combined efforts with the research community. We are optimistic that SATD will continue to receive attention in the field for the upcoming years.

Regarding our plans for the future, we are interested on continuing the line of research of this thesis, focusing on the repayment of SATD. Stepping back to see the bigger picture, the questions of what SATD should be repaid, and which debt instance should be prioritized remains open. The work we presented in Chapter 3 is one step in the right direction, but many others are needed. Several observations made in our empirical study spark new lines of investigation. For example, we observed that only a small amount of debt instances incurred positive or negative interest. This was consistent among our studied projects, and might explain why models for SATD interest measures did not have a good fit.

As immediate future work, we want to investigate the usefulness of SATD interest measures, in particular, of the compound interest rate proposed in Chapter 3. We envision an approach to rank and recommend which SATD instance should be addressed first among those that remain in a system and are candidates for repayment. Thus, we plan to study on a way to prioritize the resolution of SATD. Finally, we consider that our goal of repaying SATD will not be completed until a replicable and generalizable approach to recommend SATD that should be prioritized is available. Therefore, our efforts will be also dedicated to implement our future research findings as tools for both, researchers and practitioners.

Appendix A

Literature Review

The following set of tables complement the material presented in Chapter 2 of this thesis.

Table A.1: Published artifacts and online references.

Reference	Published Artifacts	Online Reference
Potdar and Shihab (2014)	62 SATD detection patters	https://users.encs.concordia.ca/$\sim$eshihab/data/ICSME2014/satd.html
Maldonado and Shihab (2015)	Dataset of classified SATD	http://users.encs.concordia.ca/$\sim$eshihab/data/MTD2015/MTD_15_data.zip
Freitas Farias et al. (2015a)	CVM-TD, vocabulary and filtered comments	http://goo.gl/4IvwtA
	Relational table of SE Nouns and TD types	http://goo.gl/5jWY2C
	Study data set (TD selected comments)	http://goo.gl/HBc5nt
	eXcomment Tool	http://goo.gl/9Mgl9m
Freitas Farias et al. (2016c)	Experimental Package	https://drive.google.com/file/d/0BwwEbWFwapG1UzhVZDlxczlDX0E/view
	Most selected patterns by participants	https://drive.google.com/file/d/0BwwEbWFwapG1UlNZbG5lekN1UUk/view
	Comments by Ratio	https://drive.google.com/file/d/0BwwEbWFwapG1Y2hRaEt1bGFGa2s/view
Bavota and Russo (2016)	Replication Package (R Scripts and data sets)	http://www.inf.unibz.it/$\sim$gbavota/reports/satd
Ichinose et al. (2016)	Tool demo video	https://www.youtube.com/watch?v=ZQTTO91v4No
Maldonado, Shihab, and Tsantalis (2017b)	Dataset of classified SATD with NLP detection	https://github.com/maldonado/tse.satd.data
Maldonado, Abdalkareem, et al. (2017a)	Study data set, survey form and responses	http://das.encs.concordia.ca/uploads/2017/07/maldonado_icsme2017.zip

Huang et al. (2018)	Source code and study data set	https://github.com/tkdsheep/TechnicalDebt
Liu et al. (2018)	SATD Detector Tool	https://goo.gl/ZzjBzp
	Tool demo video	https://youtu.be/sn4gU2qhGm0
Zampetti et al. (2018)	Data set of SATD Removals	http://home.ing.unisannio.it/fiorella.zampetti/datasets/ReplicationSATDRemoval.zip

Table A.2: Paper selection based on citations of Potdar and Shihab (2014).

Reference	Title	Selected
Maldonado and Shihab (2015)	Detecting and Quantifying Diferent Types of Self-Admitted Technical Debt	Yes
Freitas Farias et al. (2015a)	A Contextualized Vocabulary Model for Identifying Technical Debt on Code Comments	Yes
Ortu et al. (2015)	The JIRA repository dataset: Understanding social aspects of software development	No
Bavota and Russo (2016)	A Large-Scale Empirical Study on Self-Admitted Technical Debt	Yes
Wehaibi et al. (2016)	Examining the Impact of Self-admitted Technical Debt on Software Quality	Yes
Freitas Farias et al. (2015a)	Investigating the Identifcation of Technical Debt Through Code Comment Analysis	Yes
Vassallo et al. (2016)	Continuous Delivery Practices in a Large Financial Organization	Yes
Kamei et al. (2016)	Using Analytics to Quantify the Interest of Self-Admitted Technical Debt	Yes
Mensah et al. (2016)	Rework Effort Estimation of Self-Admitted Technical Debt	Yes
Ichinose et al. (2016)	ROCAT on KATARIBE: Code Visualization for Communities	Yes
Bellomo et al. (2016)	Got technical debt? Surfacing elusive technical debt in issue trackers	No
Akbarinasaji and Bener (2016)	Adjusting the balance sheet by appending technical debt	No
Ortu et al. (2016)	The emotional side of software developers in JIRA	No

Siegmund (2016)	Program comprehension: Past, present, and future	No
Silva, Valente, and Terra (2016)	Does technical debt lead to the rejection of pull requests?	No
Ghanbari (2016)	Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study	No
Steneker (2016)	Towards an empirical validation of the TIOBE Quality Indicator	No
Stijlaart and Zaytsev (2017)	Towards a taxonomy of grammar smells	No
Maldonado, Shihab, and Tsantalis (2017b)	Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt	Yes
Palomba et al. (2017)	An Exploratory Study on the Relationship between Changes and Refactoring	Yes
Miyake et al. (2017)	A Replicated Study on Relationship Between Code Quality and Method Comments	Yes
Maldonado, Abdalkareem, et al. (2017a)	An Empirical Study on the Removal of Self-Admitted Technical Debt	Yes
Zampetti, Noiseux, et al. (2017)	Recommending when Design Technical Debt Should be Self-Admitted	Yes
Ghanbari (2017)	Investigating the causal mechanisms underlying the customization of software development methods	No
Falessi, Russo, and Mullen (2017)	What if i had no smells?	No
Bhaalerao (2017)	Determination of Hotspots and a Study of Technical Debt in OSS Projects and Their Forks	No
Ziegler (2017)	GITCoP: A Machine Learning Based Approach to Predicting Merge Conflicts from Repository Metadata	No
Zampetti, Ponzanelli, et al. (2017)	How developers document pull requests with external references	No
Mensah et al. (2018)	On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt	Yes
Huang et al. (2018)	Identifying Self-Admitted Technical Debt in Open Source Projects using Text-Mining	Yes
Liu et al. (2018)	SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool	Yes
Zampetti et al. (2018)	Was Self-Admitted Technical Debt Removal a real Removal? An In-Depth Perspective	Yes
Yan et al. (2018)	Automating Change-level Self-admitted Technical Debt Determination.	Yes
Leßenich, Siegmund, Apel, Kästner, and Hunsen (2018)	Indicators for merge conflicts in the wild: survey and empirical study	No
Verdecchia, Malavolta, and Lago (2018)	Architectural Technical Debt Identification: The Research Landscape	No

Alfayez, Behnamghader, Srisopha, and Boehm (2018)	An Exploratory Study on the Influence of Developers in Technical Debt	No
Amanatidis, Mittas, Chatzigeorgiou, Ampatzoglou, and Angelis (2018)	The developer's dilemma: factors affecting the decision to repay code debt	No
Ghanbari, Vartiainen, and Siponen (2018)	Omission of Quality Software Development Practices: A Systematic Literature Review	No

Table A.3: List of studied systems per surveyed paper and TD validation.

Reference	Studied systems	#	TD Validation
Potdar and Shihab (2014)	ArgoUML, Eclipse, Chromium OS, Apache Httpd	4	Manual inspection by the first author
Maldonado and Shihab (2015)	Apache JMeter, ArgoUML, Apache Ant, Columba, JFreeChart	5	Manual inspection by the first author
Freitas Farias et al. (2015a)	JEdit, Apache Lucene	2	Manual inspection by 4 software engineering master students
Wehaibi et al. (2016)	Hadoop, Tomcat, Chromium OS, Cassandra, Spark	5	Manual inspection by the first author
Freitas Farias et al. (2016c)	ArgoUML	1	Manual inspection by 3 TD specialists and 32 software engineering master students
Bavota and Russo (2016)	120 projects from the Apache Ecosystem; 39 from the Eclipse Ecosystem	159	Manual inspection of a statistically significant sample by both authors
Kamei et al. (2016)	Apache JMeter	1	Manual inspection by the second author
Mensah et al. (2016)	ArgoUML, Eclipse, Chromium OS, Apache Httpd	4	Relies on a pattern-based detection (Potdar & Shihab, 2014)
Maldonado, Shihab, and Tsantalis (2017b)	ArgoUML, Apache Ant, Columba, EMF, Hibernate ORM, JEdit, JFreeChart, Apache JMeter, JRuby, SQuirrel SQL Client	10	Manual inspection by the first author

Palomba et al. (2017)	ArgoUML, Apache Ant, Apache Xerces-J	3	Manual inspection by the authors
Miyake et al. (2017)	PMD, SQuirrel SQL Client, Free Mind, Hibernate ORM	4	Manual inspection by the authors
Maldonado, Abdalkareem, et al. (2017a)	Camel, Gerrit, Hadoop, Log4j, Tomcat	5	Relies on the data set provided in (Maldonado, Shihab, & Tsantalis, 2017b)
Zampetti, Noiseux, et al. (2017)	ArgoUML, Apache Ant, Columba, Hibernate ORM, JEdit, JFreeChart, Apache JMeter, JRuby, SQuirrel SQL Client	9	Relies on the data set provided in (Maldonado, Shihab, & Tsantalis, 2017b)
Mensah et al. (2018)	PMD, SQuirrel SQL Client, Free Mind, Hibernate ORM	4	Manual inspection of a statistically significant sample by the authors
(Huang et al., 2018)	ArgoUML, Columba, Hibernate ORM, JEdit, JFreeChart, Apache JMeter, JRuby, SQuirrel SQL Client	8	Relies on the data set provided in (Maldonado & Shihab, 2015)
(Liu et al., 2018)	ArgoUML, Apache Ant, Columba, Hibernate ORM, JEdit, JFreeChart, Apache JMeter, JRuby, SQuirrel SQL Client	9	Relies on the data set provided in (Maldonado & Shihab, 2015)
(Zampetti et al., 2018)	Camel, Gerrit, Hadoop, Log4j, Tomcat	5	Manual inspection by the authors on the data set provided in (Maldonado, Abdalkareem, et al., 2017a)
(Yan et al., 2018)	Apache JMeter, Apache Ant, Camel, Gerrit, Hadoop, Log4j, Tomcat	7	Manual inspection by the first author and an independent Ph.D. Student

Note: The studies by Vassallo et al. (2016), and Ichinose et al. (2016) have been excluded from this list due to non-available information.

Appendix B

Empirical Study

B.1 Extracted Product Metrics

Table B.1 lists the 21 complexity and volume metrics collected using the API SciTools Understand in Section 3.3.7 as described by the tool's documentation SciTools (2018a).

Table B.1: Description of collected product metrics.

Metric Name	Type	Description
Paths	Complexity	Number of possible paths, not counting abnormal exits or gotos.
Cyclomatic Complexity	Complexity	McCabe Cyclomatic complexity.
Modified Cyclomatic Complexity	Complexity	Cyclomatic Complexity, except that each decision in a multi-decision structure statement is not counted and instead the entire multi-way decision structure counts as 1.
Strict Cyclomatic Complexity	Complexity	Cyclomatic Complexity with logical conjunction and logical and in conditional expressions also adding 1 to the complexity for each of their occurrences.
Essential Complexity	Complexity	Cyclomatic complexity after iteratively replacing all well structured control structures with a single statement. Structures such as if-then-else and while loops are considered well structured.
Knots	Complexity	Measure of overlapping jumps. If a piece of code has arrowed lines indicating where every jump in the flow of control occurs, a knot is defined as where two such lines cross each other.
Max Knots	Complexity	Maximum Knots after structured programming constructs have been removed.
Nesting	Complexity	Maximum nesting level of control constructs (if, while, for, switch, etc.) in the function.
Minimum Knots	Complexity	Minimum Knots after structured programming constructs have been removed.
Outputs	Count - Size	The number of outputs that are SET. This can be parameters or global variables.
Inputs (Fan-In)	Count - Size	The number of inputs a function uses plus the number of unique subprograms calling the function.
Physical Lines	Count - Size	Number of physical lines.
Blank Lines of Code	Count - Size	Number of blank lines.
Source Lines of Code (LOC)	Count - Size	The number of lines that contain source code.
Declarative Lines of Code	Count - Size	Number of lines containing declarative source code.
Executable Lines of Code	Count - Size	Number of lines containing executable source code.
Lines with Comments	Count - Size	Number of lines containing comments.
Statements	Count - Size	Number of declarative plus executable statements.
Declarative Statements	Count - Size	Number of declarative statements.
Executable Statements	Count - Size	Number of php executable statements.
Comment to Code Ratio	Count - Size	Ratio of number of comment lines to number of code lines.

B.2 Description of Questions in the Survey to Developers

The following describes the questions that composed the online survey sent to developers on how they prioritize SATD resolution, as overviewed in Section 3.2.

Q1: *How many years of software development experience do you have?* Multiple choice, 4 options:

- 0 to 1
- 1 to 3
- 3 to 5
- 5+ (more than 5)

Q2: *How often do you contribute code to your project(s)?* Multiple choice (single selection) 5 options:

- Never
- Rarely (e.g., once a year)
- Sometimes (e.g., once a month)
- Often (e.g., once a week)
- Very often (e.g., daily)

Q3: *Which of the following best describes your current development activities?* Multiple choice (single selection), 5 options:

- Bug fixing
- Feature addition
- Code testing

- Code reviewing
- Other (short text answer)

Q4: *Have you ever introduced SATD comments?*

Yes / no question.

Q5: *Have you ever addressed SATD comments?*

Yes / no question.

Q6: *Given a project with multiple SATD instances, please select the element(s) you would consider while deciding which SATD comment to address FIRST?*

Multiple choice (checkboxes), 8 options:

- The date when the SATD comment was introduced
- The size (in Lines of Code) of the debt to address
- The amount of dependencies of the debt
- The person who introduced the debt
- The subsystem or module where the debt is located (e.g., utility code, test code, business code) (Please also answer Q7).
- The change likelihood of the file that contains the debt (Please also answer Q8).
- The effort needed to address the debt (Please also answer Q9).
- Other (short text answer)

Q7: *If you selected: "The subsystem or module..." option above, please indicate which ones you prioritize over others when addressing SATD in your project(s).*

Long text answer.

Q8: *If you selected: "The change likelihood..." option above, please detail how you measure the change likelihood of a file in your project(s).*

Long text answer.

Q9: *If you selected: "The effort needed to address the debt" option above, please detail how you measure this effort in your project(s).*

Long text answer.

Q10: *Would you be interested in knowing the outcome of our study? If yes, please provide your email address below.*

Short text answer.

B.3 Camel

Model: RemovalTimeInDays ∼ AuthorExperience + Entropy + log(1+CommitChurn) + ModifiedSubsystems. **R-squared:** 0.01606

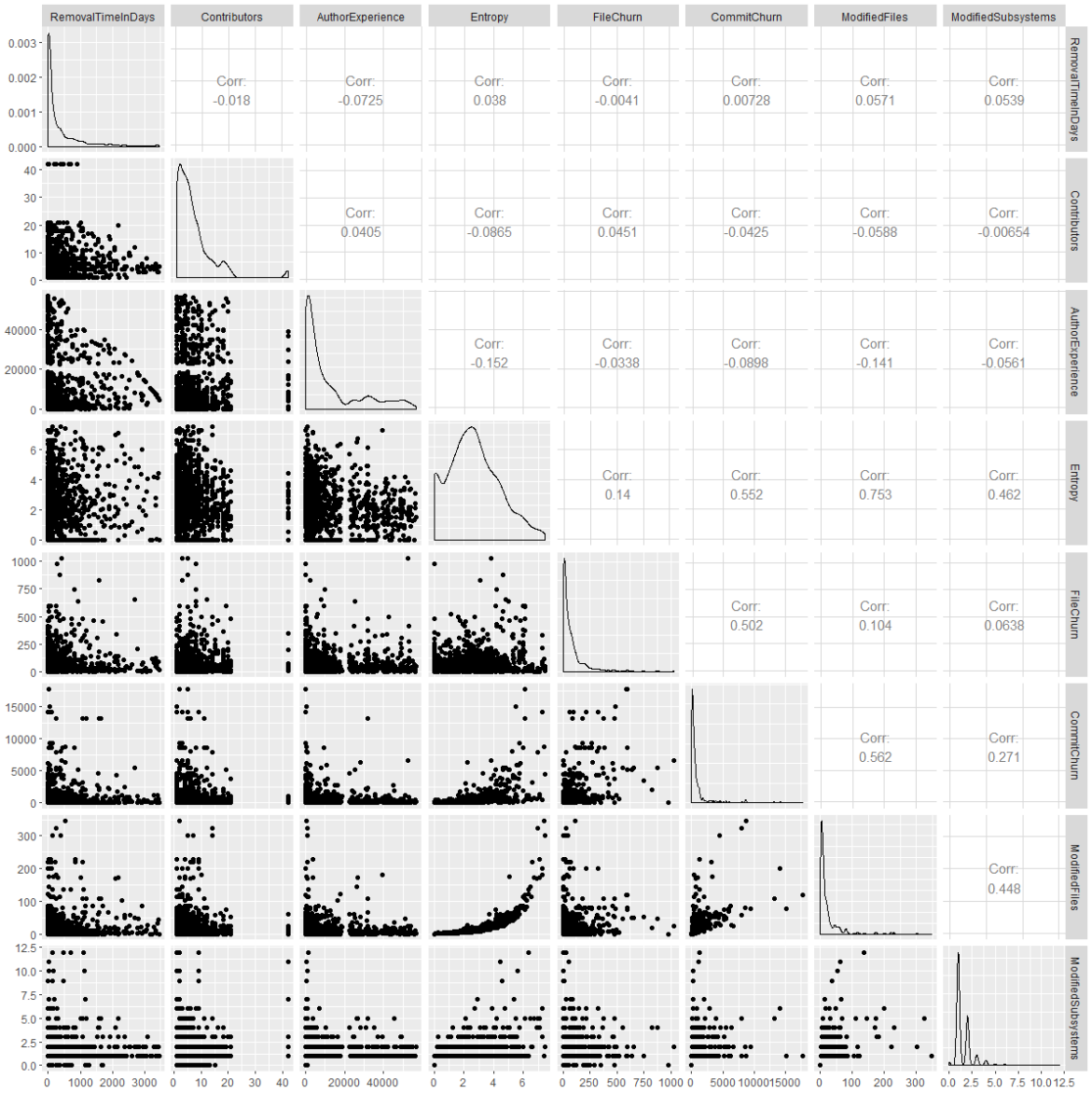

Figure B.1: Camel - Removal time (days).

Model: EffortInWords ∼ Contributors + AuthorExperience + Entropy + CommitChurn + ModifiedFiles. **R-squared:** 0.0201

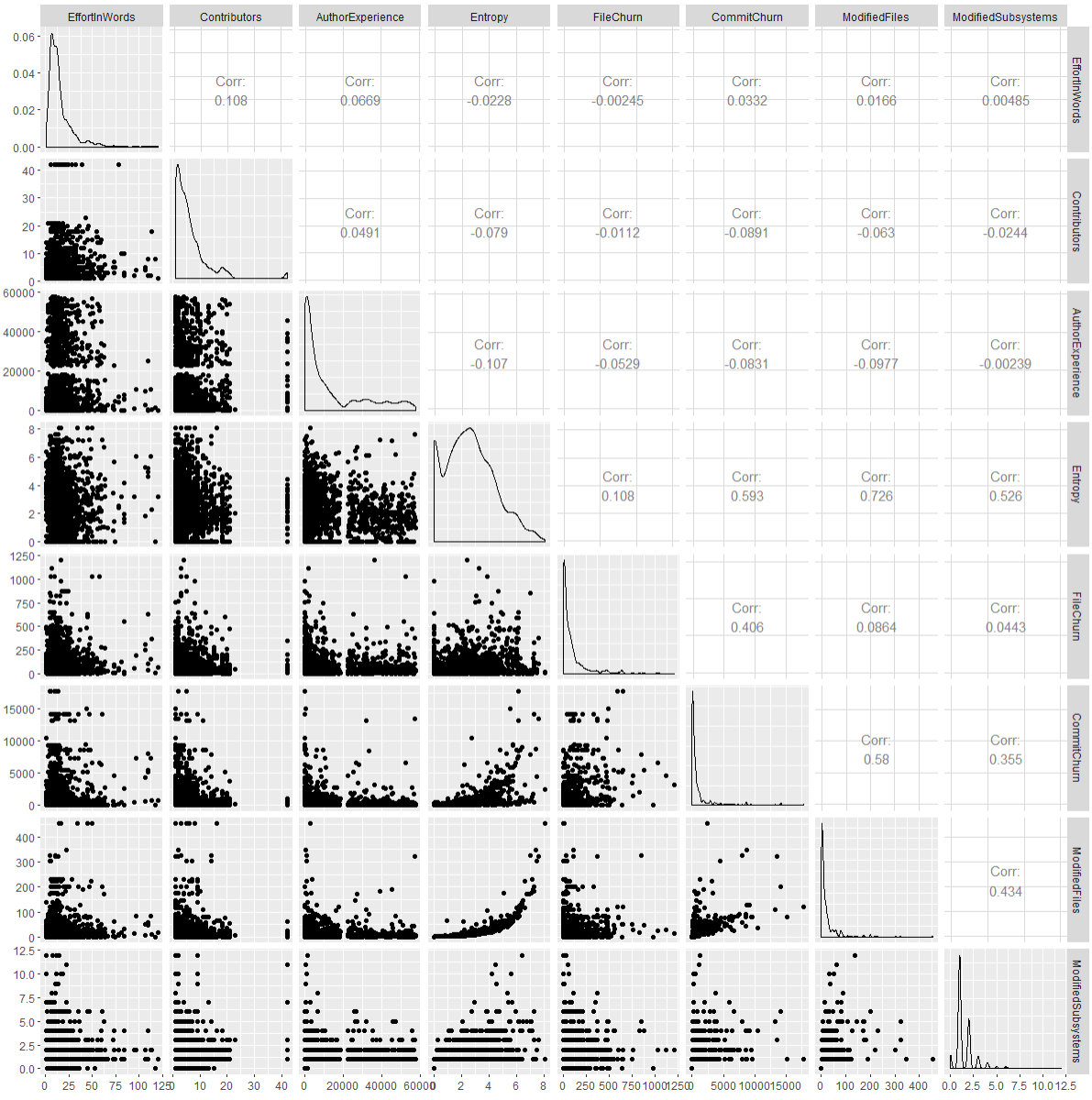

Figure B.2: Camel - Effort in Words (days).

Model: InterestFanIn ∼ log(1+Contributors) + log(1+AuthorExperience) + log(1+Entropy) + log(1+FileChurn) + CommitChurn + ModifiedFiles + log(1+ModifiedSubsystems).
R-squared: 0.001199

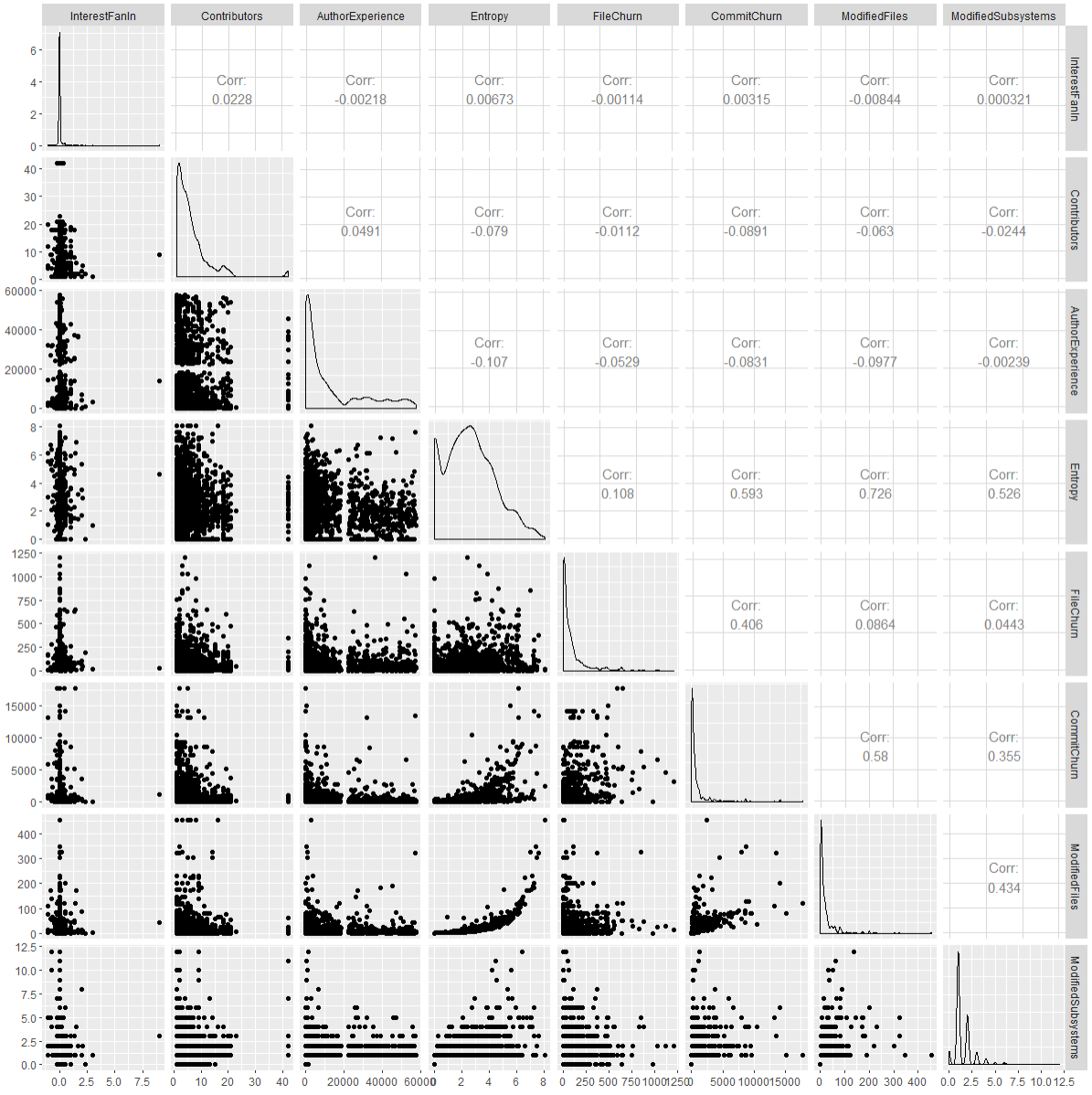

Figure B.3: Camel - Simple Interest - Fan-In

Model: InterestLOC ∼ Contributors + log(1+AuthorExperience) + Entropy + CommitChurn + ModifiedFiles. **R-squared:** 0.002853

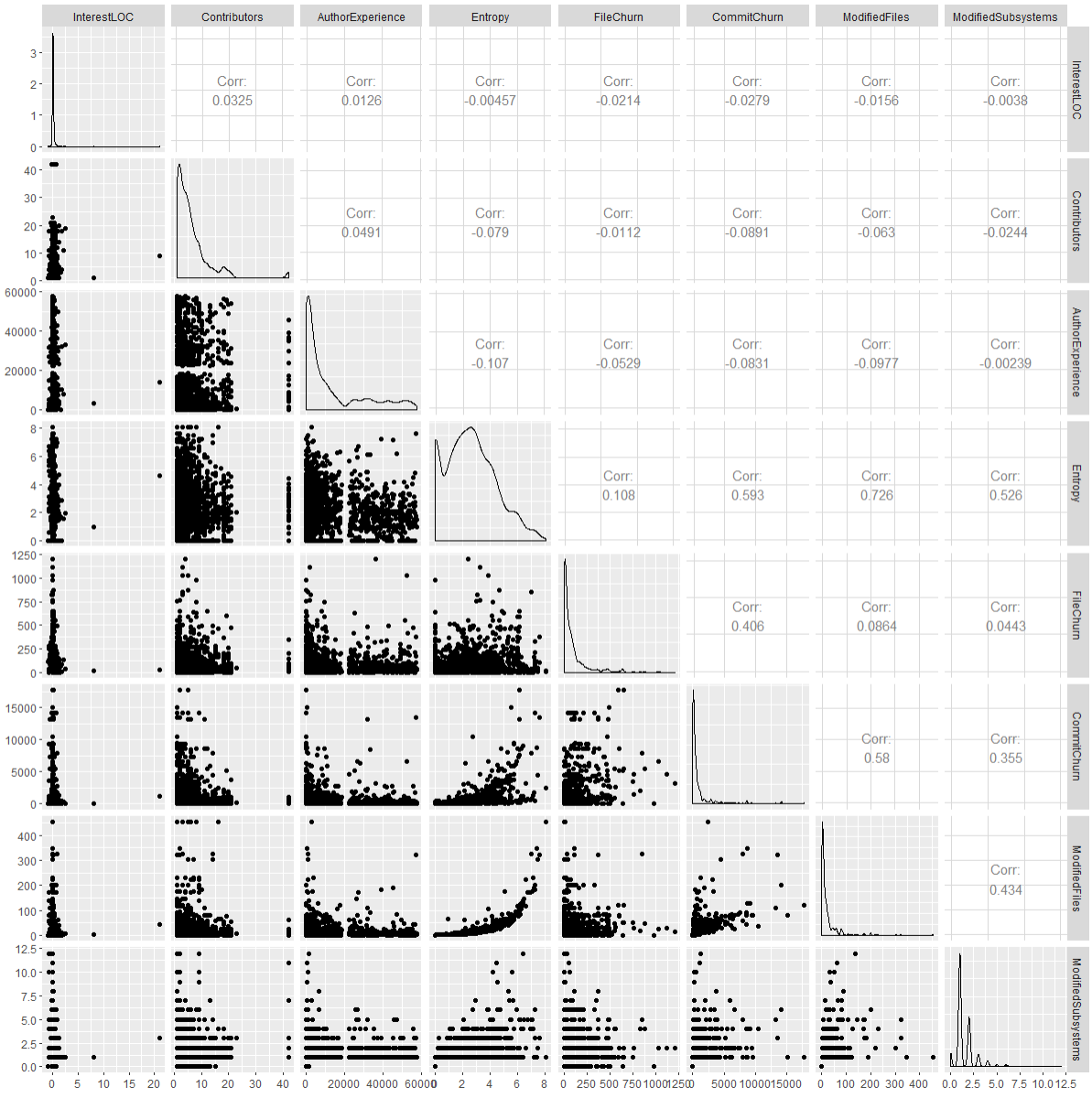

Figure B.4: Camel - Simple Interest - LOC

Model: CompoundedLOC ∼ log(1+Contributors) + log(1+AuthorExperience) + Entropy + log(1+FileChurn) + log(1+CommitChurn). **R-squared:** 0.01087

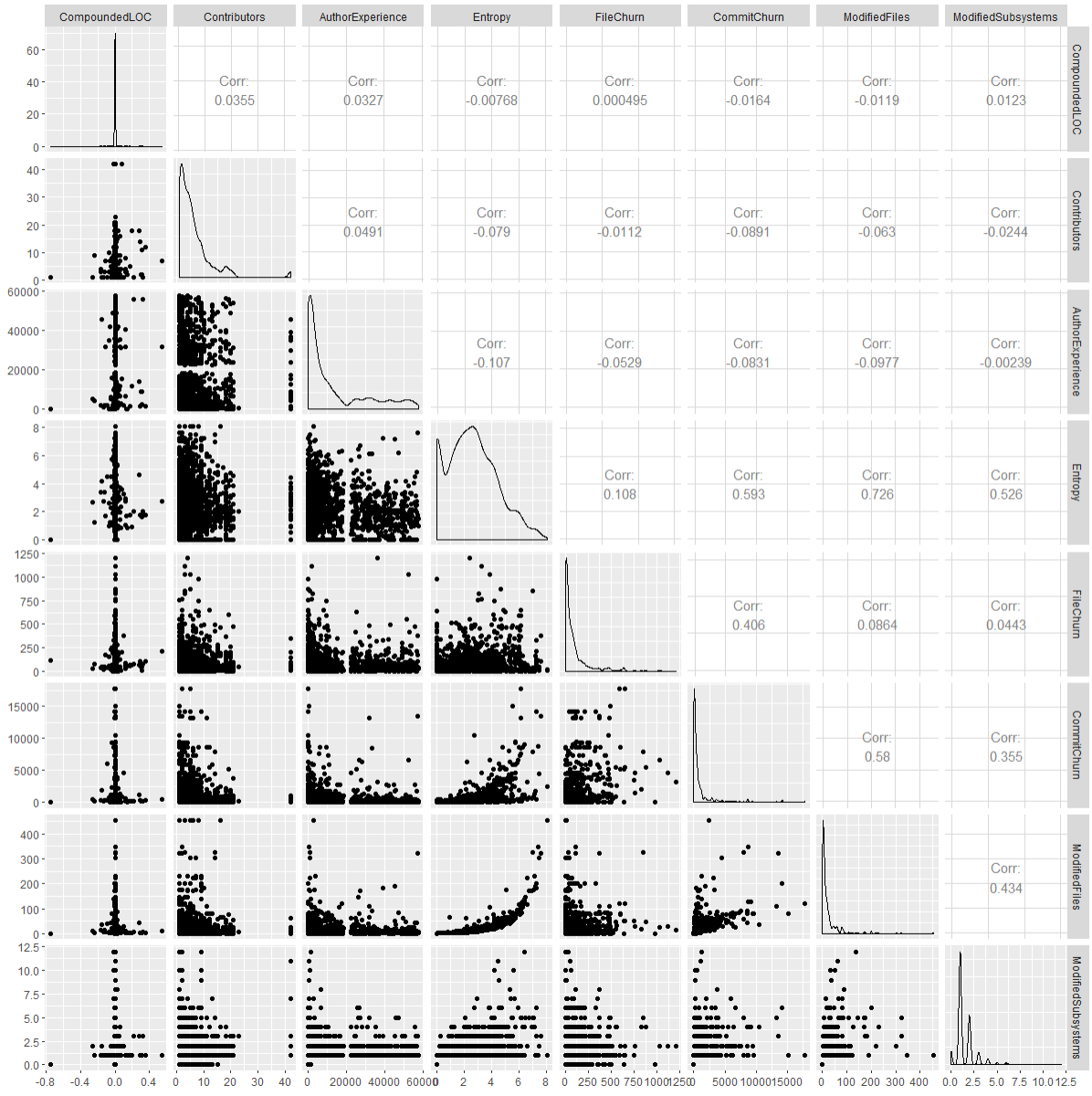

Figure B.5: Camel - Compounded Interest - LOC

Model: CompoundedFanIn ∼ AuthorExperience + Entropy + FileChurn + CommitChurn + ModifiedSubsystems. **R-squared:** 0.03587

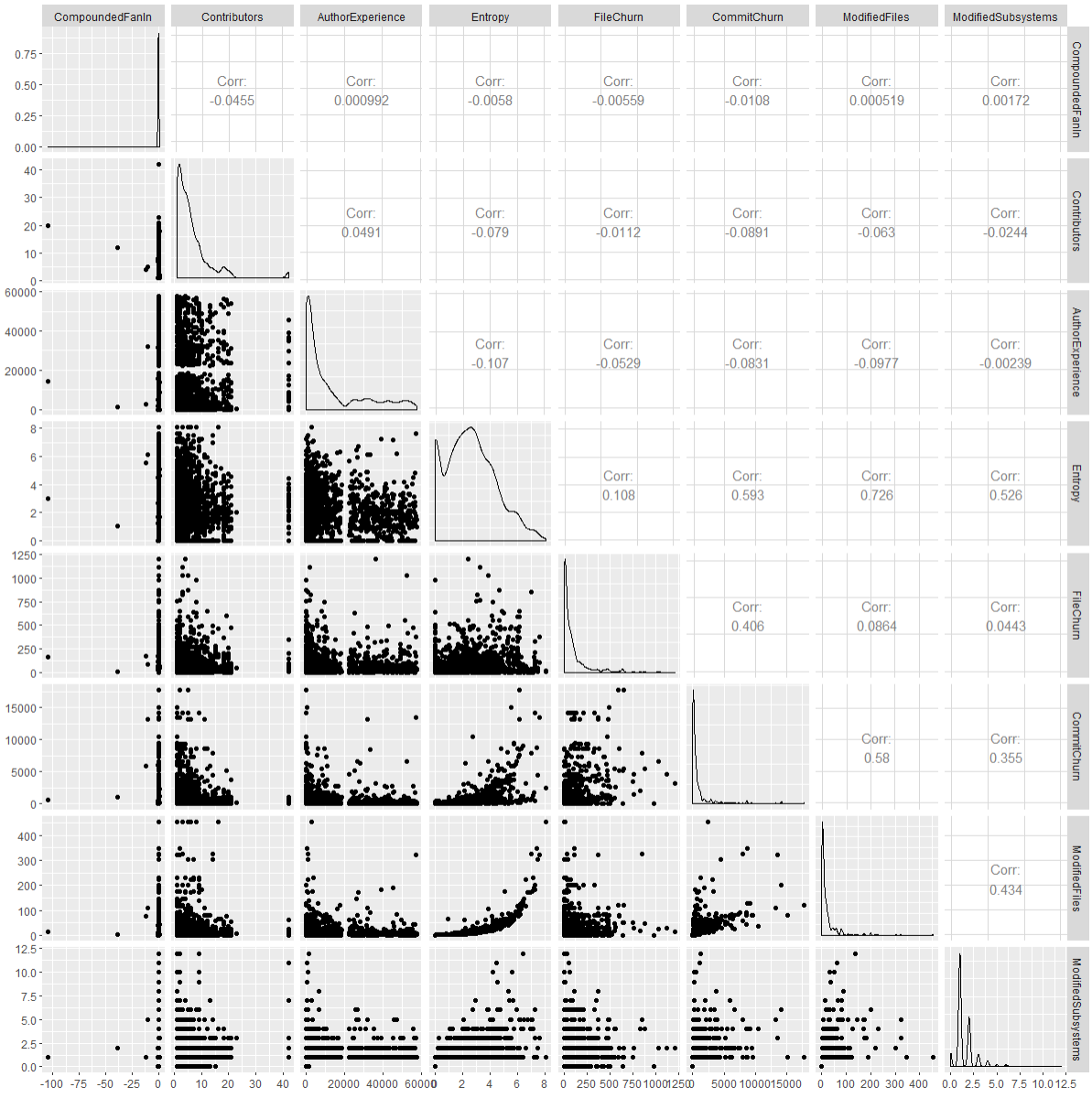

Figure B.6: Camel - Compounded Interest - Fan-In

Model: ChangeProness ∼ Contributors + FileChurn + log(1+ModifiedSubsystems).
R-squared: 0.7757

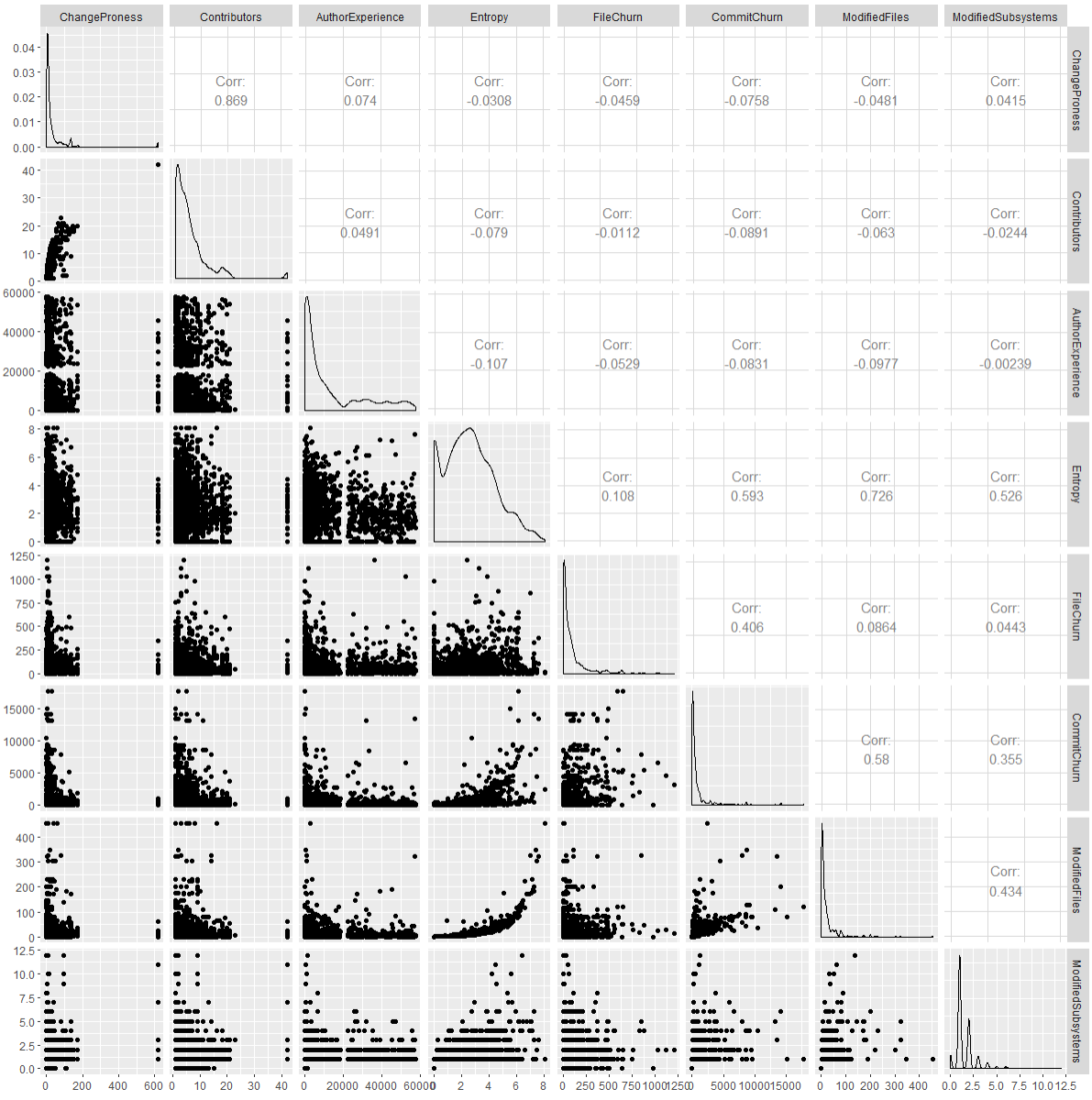

Figure B.7: Camel - Change Proneness

B.4 Hibernate

Model: RemovalTimeInDays ∼ log(1+Contributors) + log(1+AuthorExperience) + Entropy + FileChurn + ModifiedSubsystems. **R-squared:** 0.4192

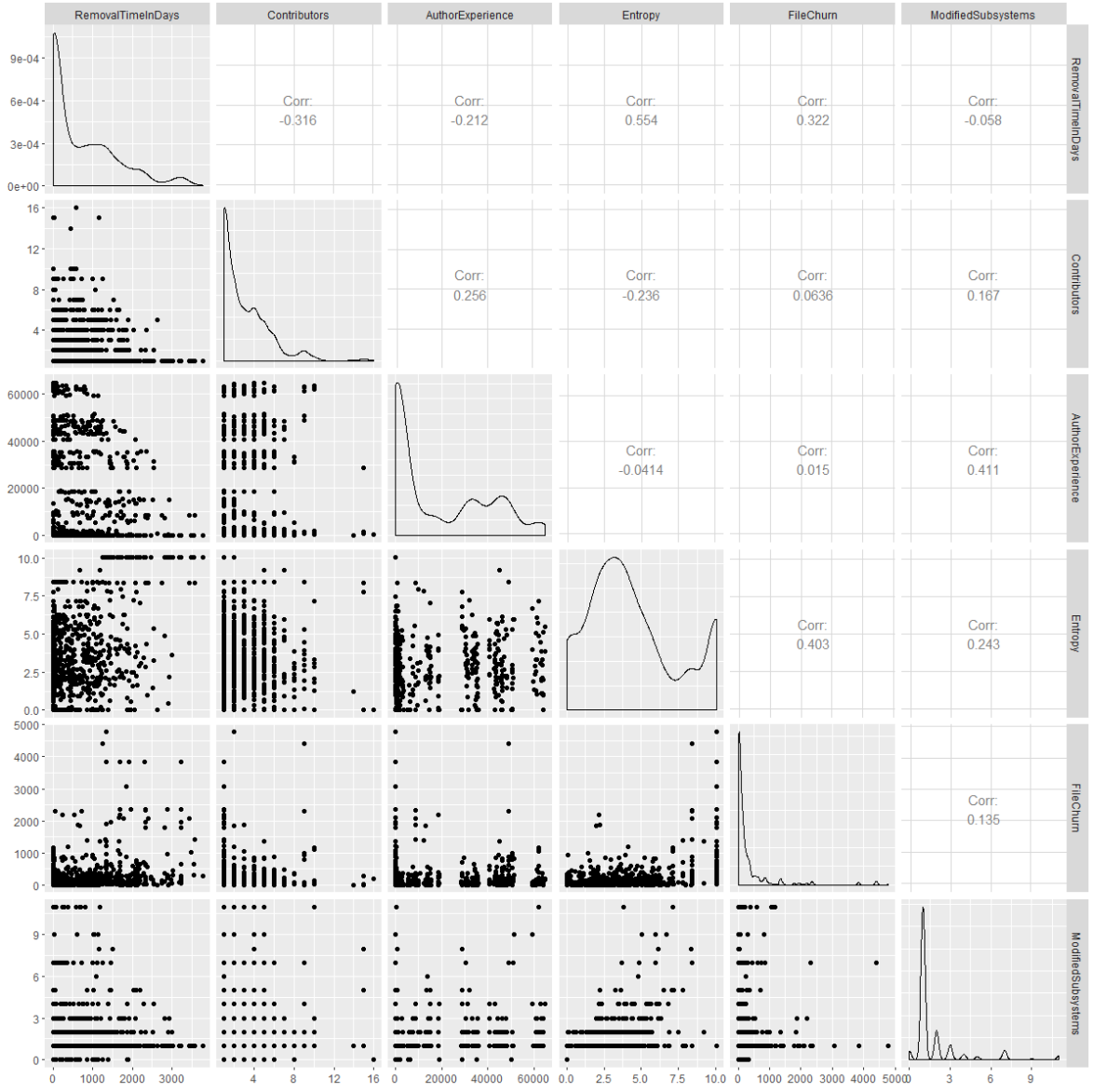

Figure B.8: Hibernate - Removal time (days).

Model: EffortInWords ∼ AuthorExperience + log(1+Entropy) + FileChurn + ModifiedSubsystems. **R-squared:** 0.004341

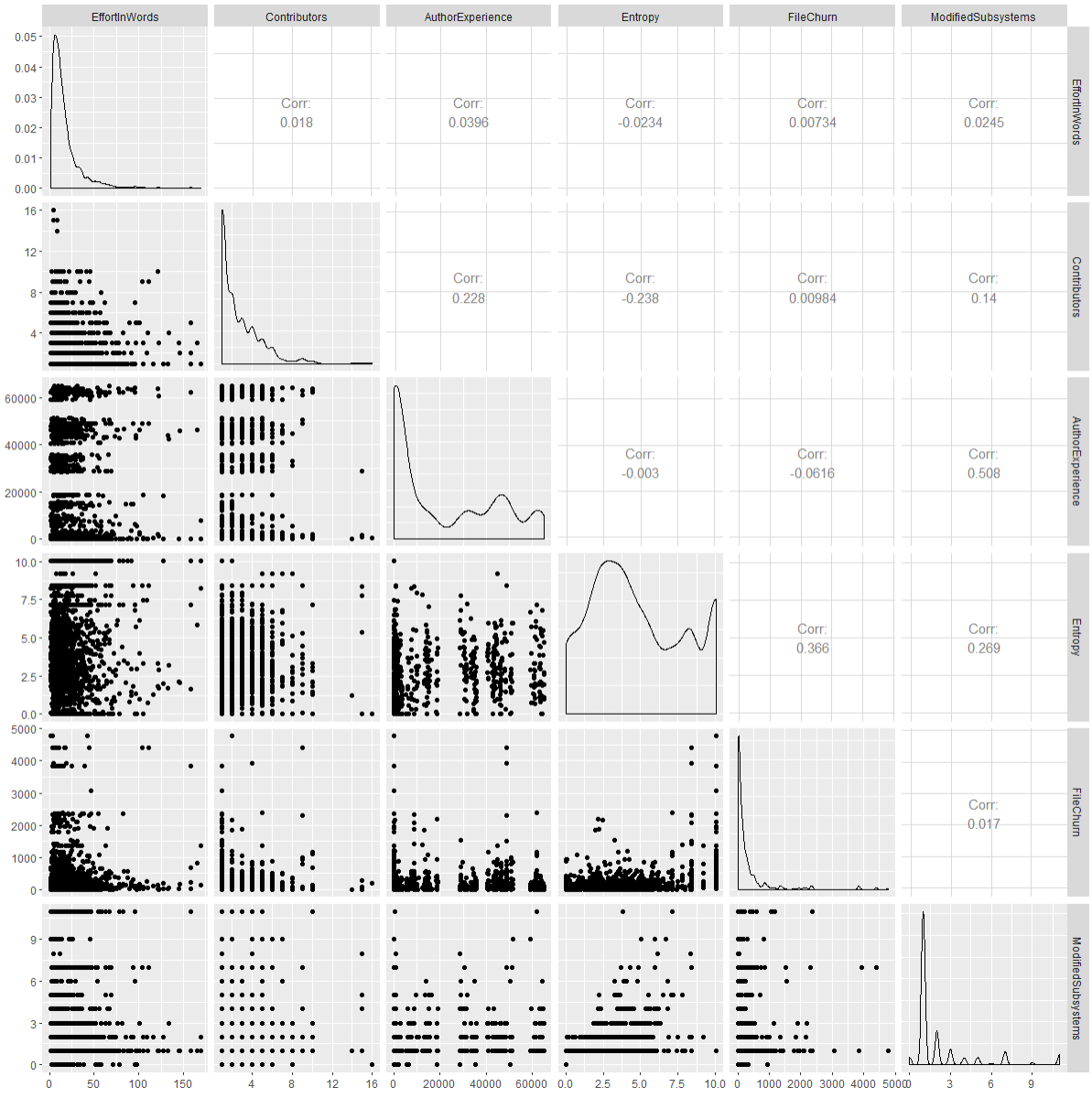

Figure B.9: Hibernate - Effort in Words (days).

Model: InterestFanIn ∼ log(1+Contributors) + log(1+AuthorExperience) + ModifiedSubsystems.
R-squared: 0.004682

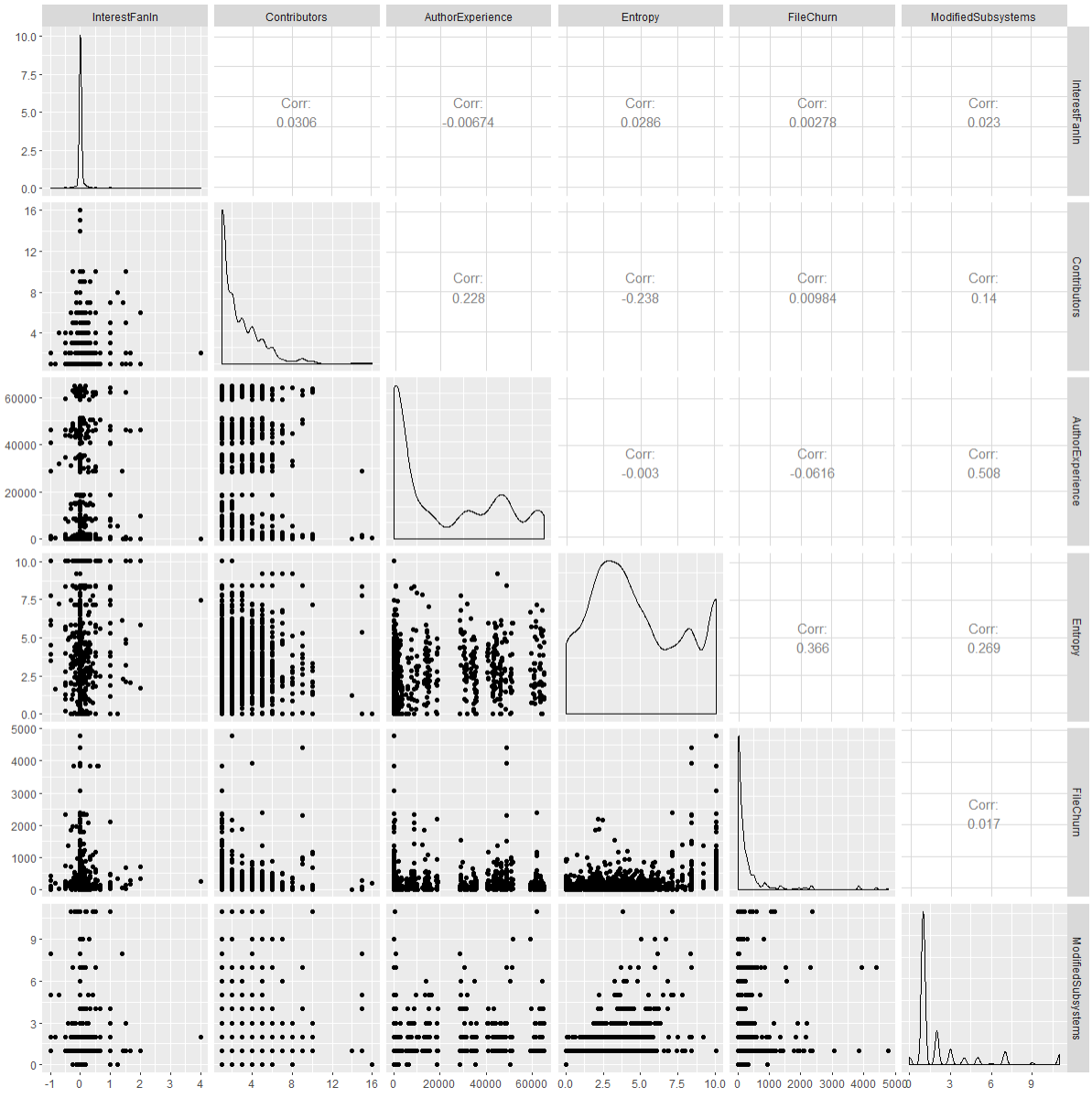

Figure B.10: Hibernate - Simple Interest - Fan-In

Model: InterestLOC ∼ Contributors + AuthorExperience + log(1+Entropy) + log(1+FileChurn) + log(1+ModifiedSubsystems. **R-squared:** 0.000941

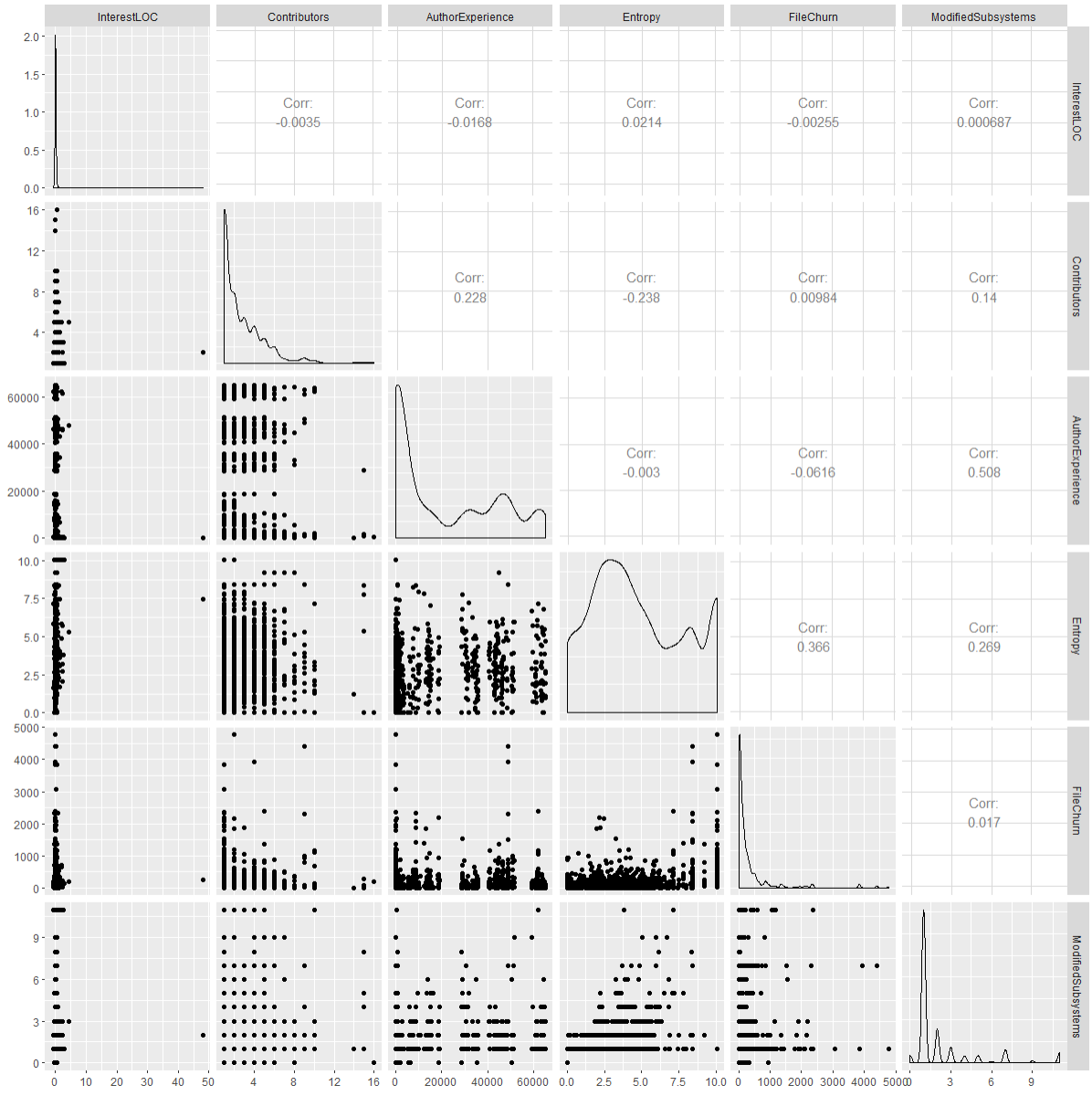

Figure B.11: Hibernate - Simple Interest - LOC

Model: CompoundedLOC ∼ Contributors + log(1+AuthorExperience) + log(1+Entropy) + FileChurn + log(1+ModifiedSubsystems). **R-squared:** 0.002742

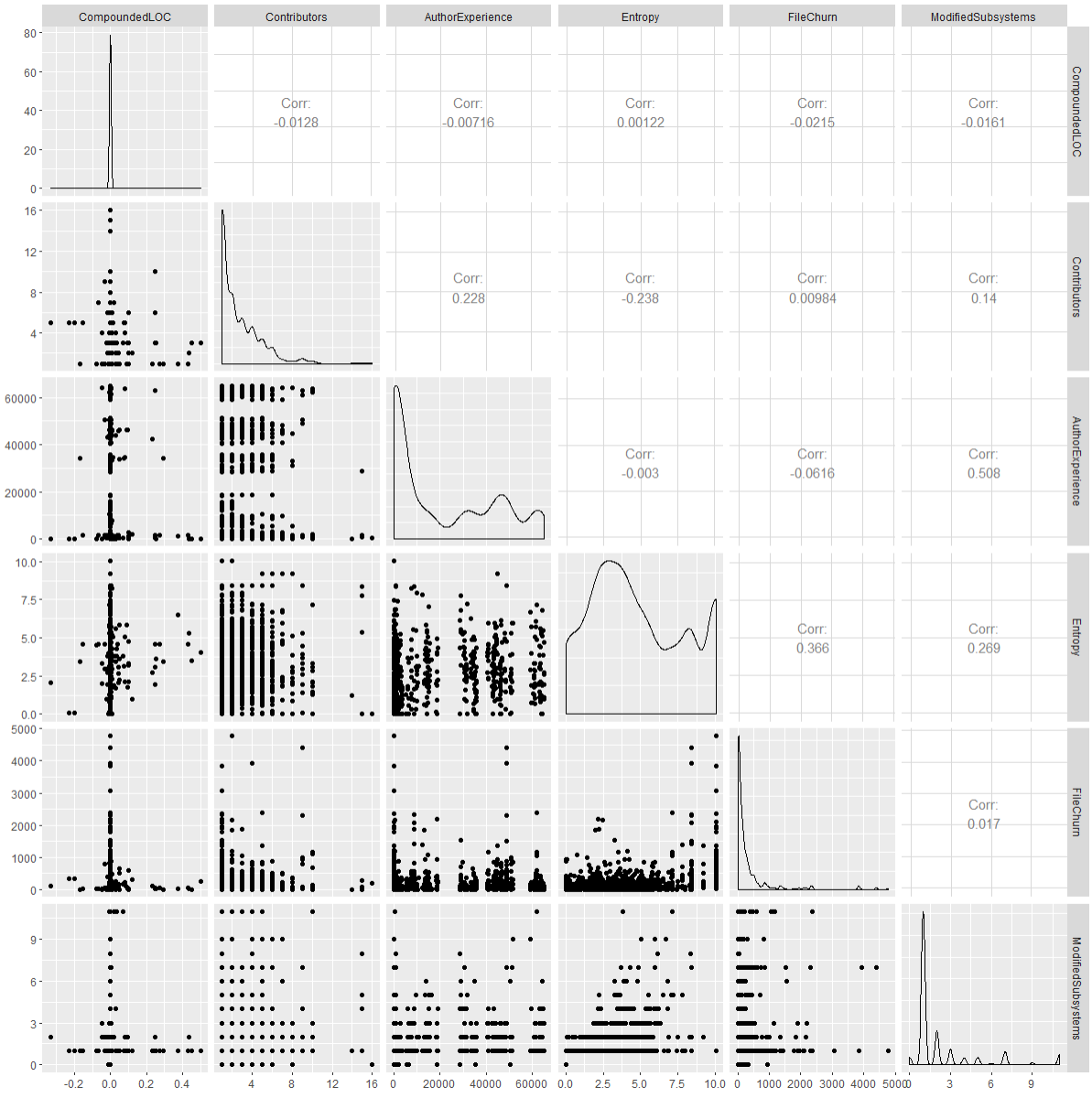

Figure B.12: Hibernate - Compounded Interest - LOC

Model: CompoundedFanIn ∼ log(1+Contributors) + log(1+Entropy) + FileChurn.
R-squared: 0.001656

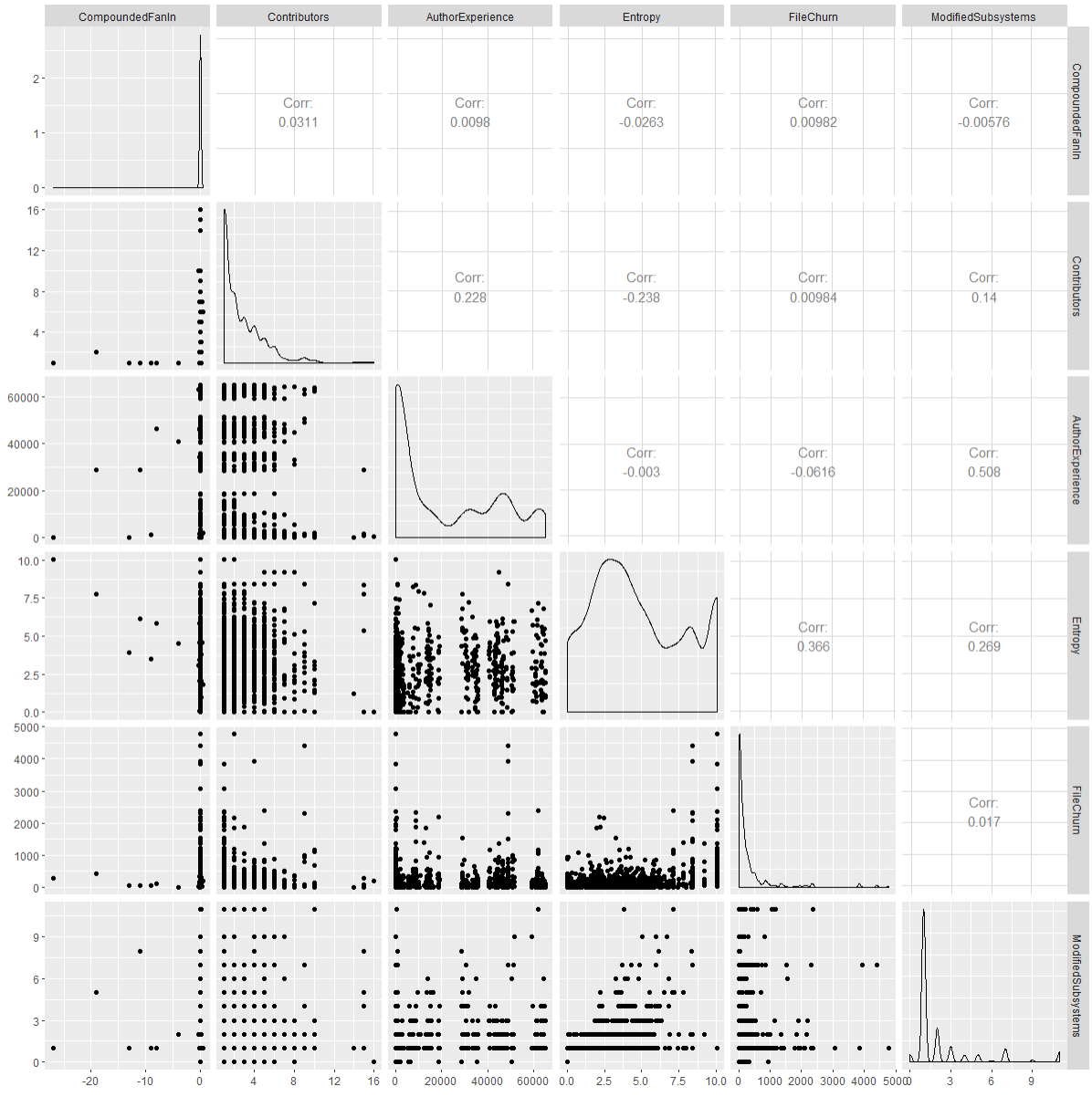

Figure B.13: Hibernate - Compounded Interest - Fan-In

Model: ChangeProness ∼ Contributors + FileChurn + ModifiedSubsystems.
R-squared: 0.4375

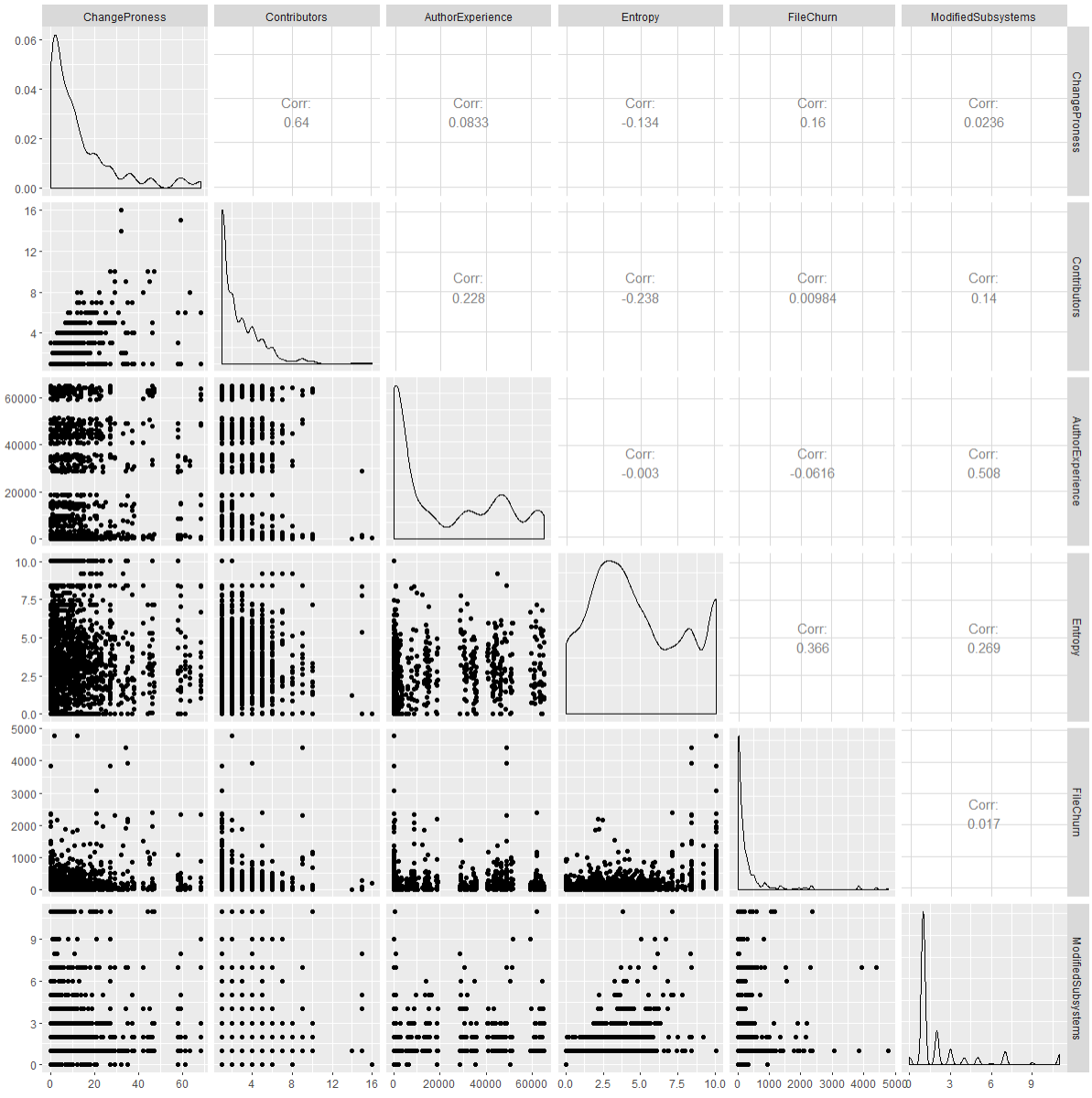

Figure B.14: Hibernate - Change Proneness

B.5 JMeter

Model: RemovalTimeInDays ∼ BugFixingCommits + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles). **R-squared:** 0.08007

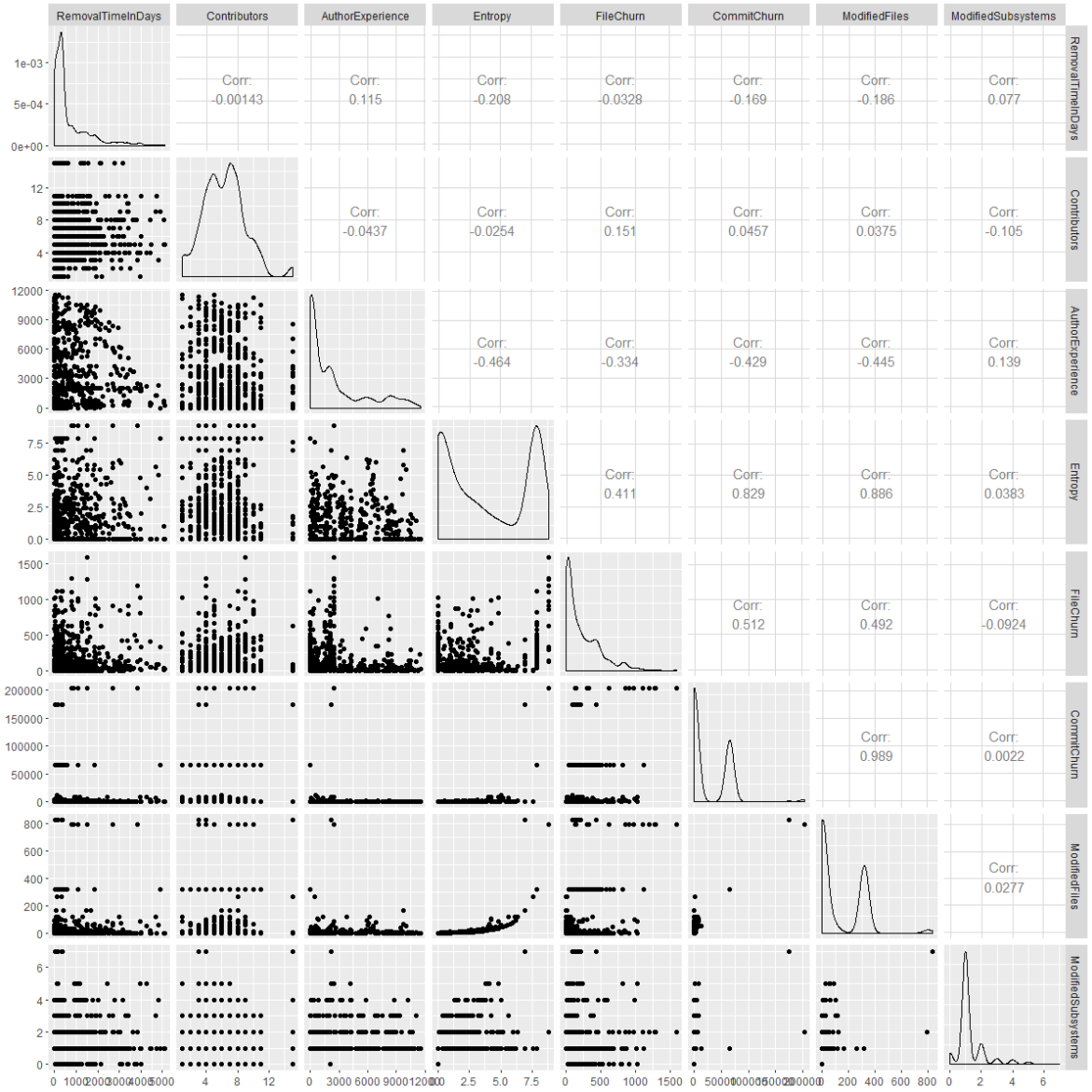

Figure B.15: JMeter - Removal time (days).

Model: EffortInWords ∼ log(1+BugFixingCommits) + log(1+Contributors) + ModifiedFiles + ModifiedSubsystems. **R-squared:** 0.04623

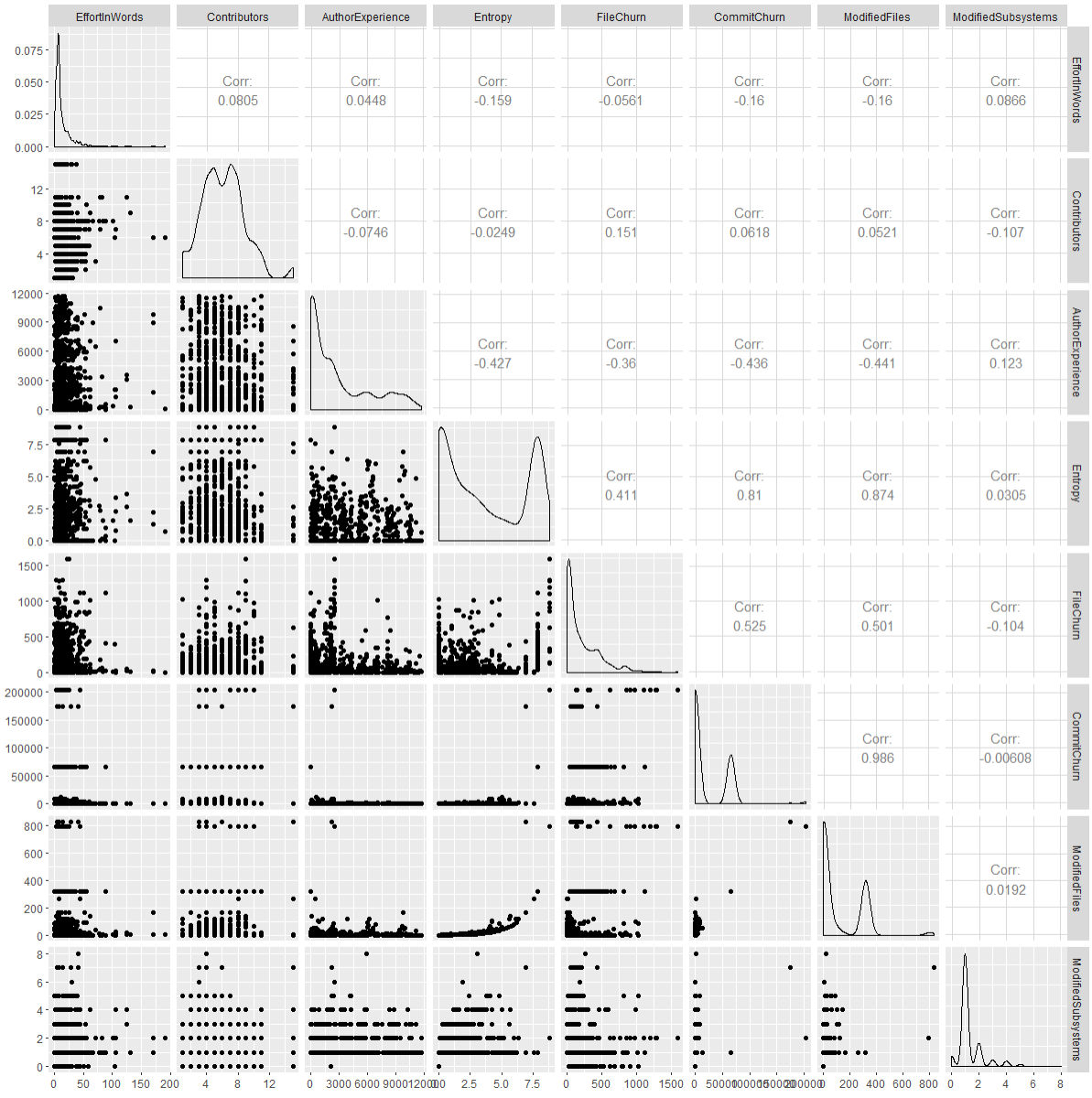

Figure B.16: JMeter - Effort in Words (days).

Model: InterestFanIn ∼ BugFixingCommits + Contributors + AuthorExperience + log(1+ModifiedFiles) + log(1+ModifiedSubsystems). **R-squared:** 0.01064

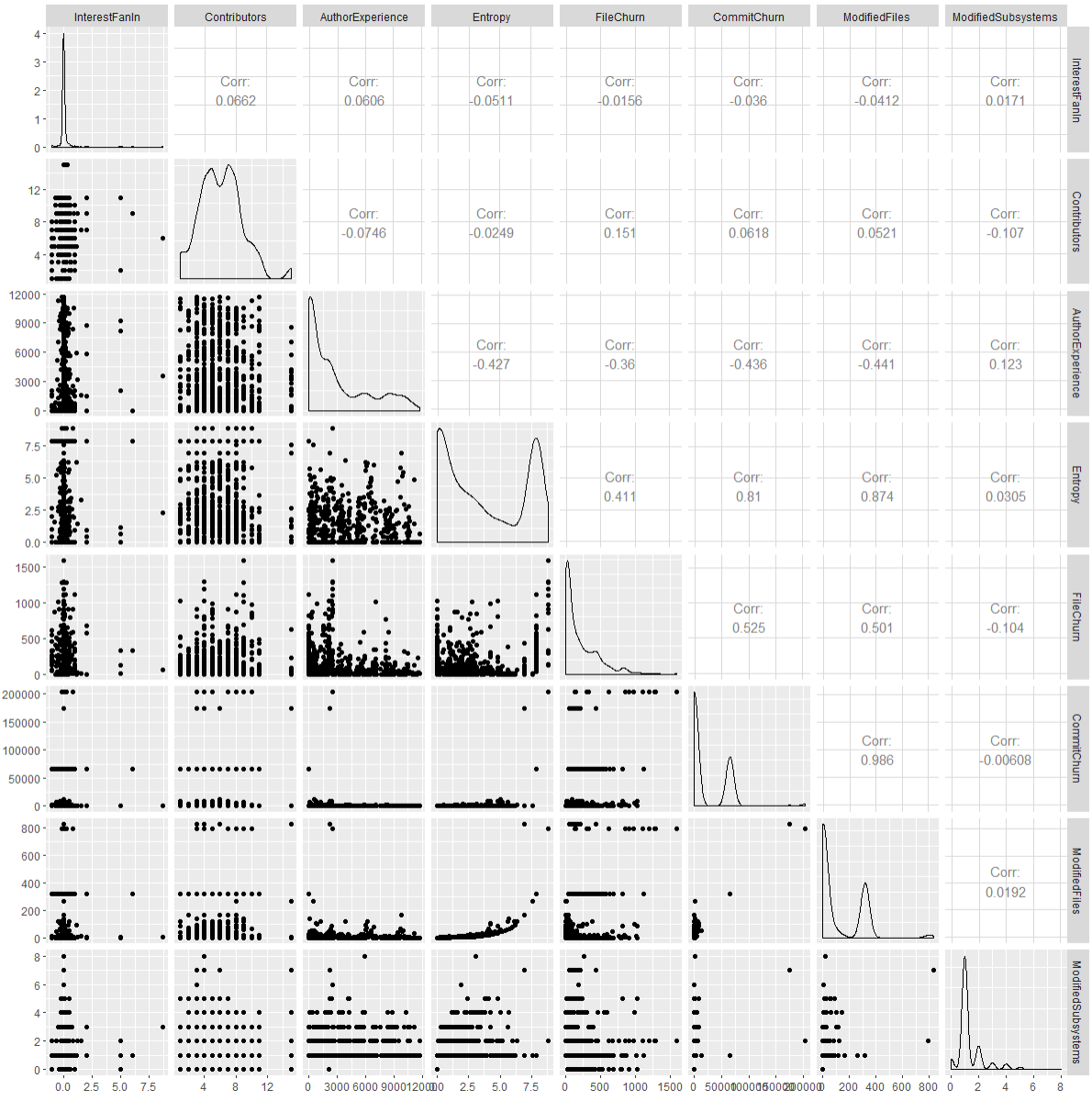

Figure B.17: JMeter - Simple Interest - Fan-In

Model: InterestLOC ∼ log(1+BugFixingCommits) + log(1+Contributors) + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems). **R-squared:** 0.001849

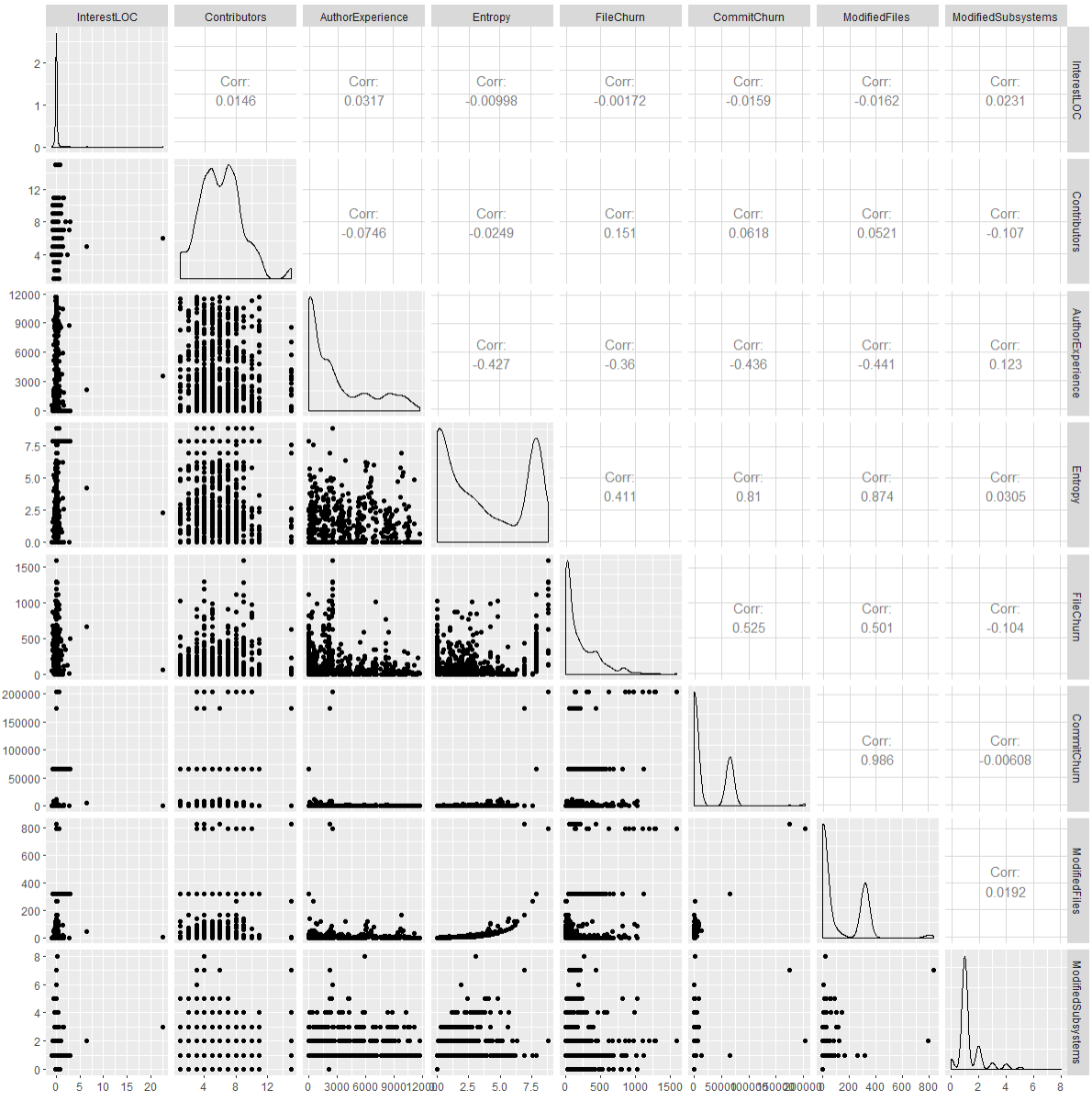

Figure B.18: JMeter - Simple Interest - LOC

Model: CompoundedFanIn ∼ log(1+BugFixingCommits) + Contributors + AuthorExperience + ModifiedFiles + ModifiedSubsystems. **R-squared:** 0.008001

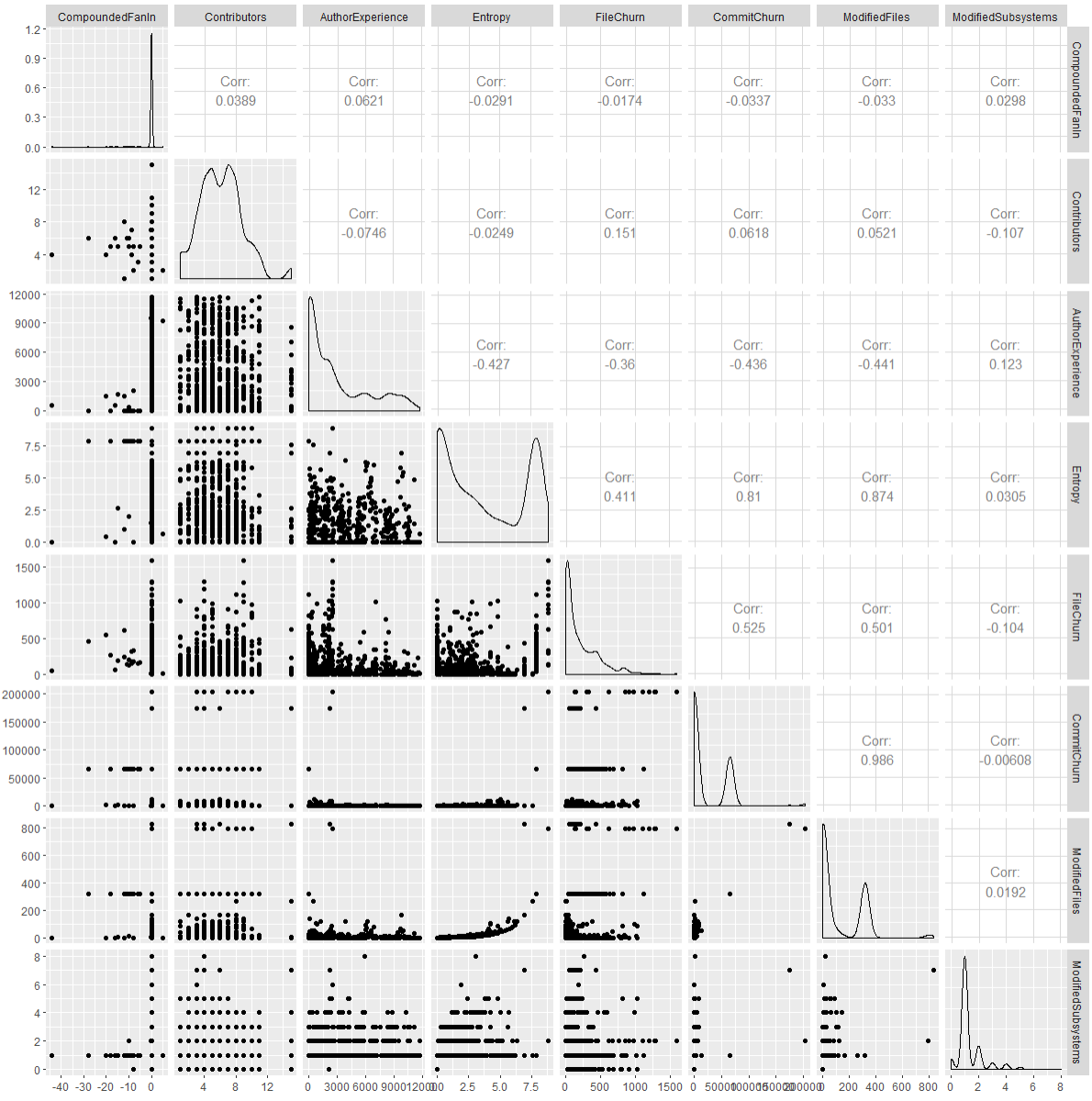

Figure B.19: JMeter - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ log(1+Contributors) + FileChurn + log(1+ModifiedFiles).
R-squared: 0.00352

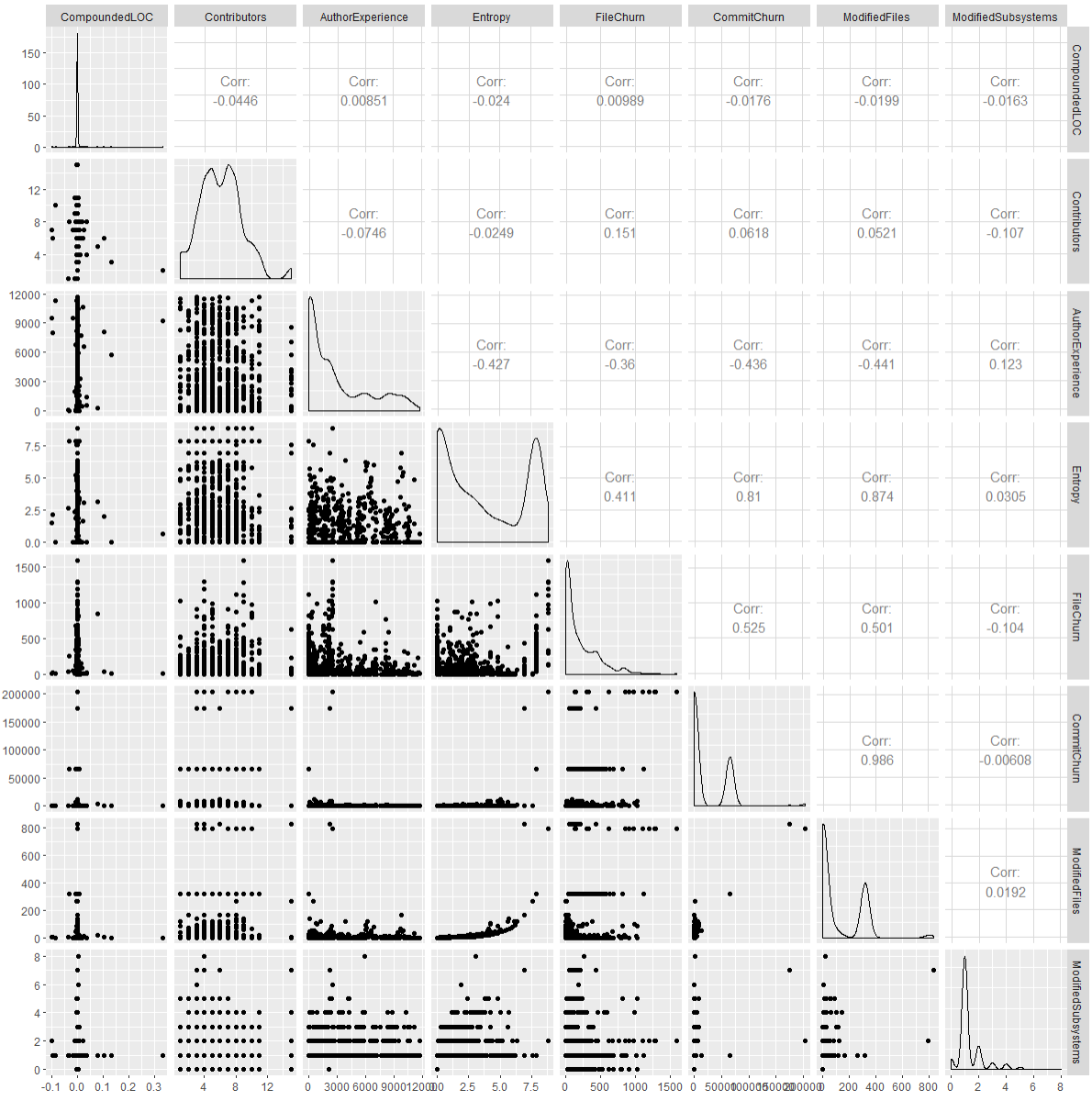

Figure B.20: JMeter - Compounded Interest - LOC

Model: ChangeProness ∼ log(1+BugFixingCommits) + Contributors + log(1+AuthorExperience) + FileChurn + log(1+ModifiedFiles). **R-squared:** 0.5791

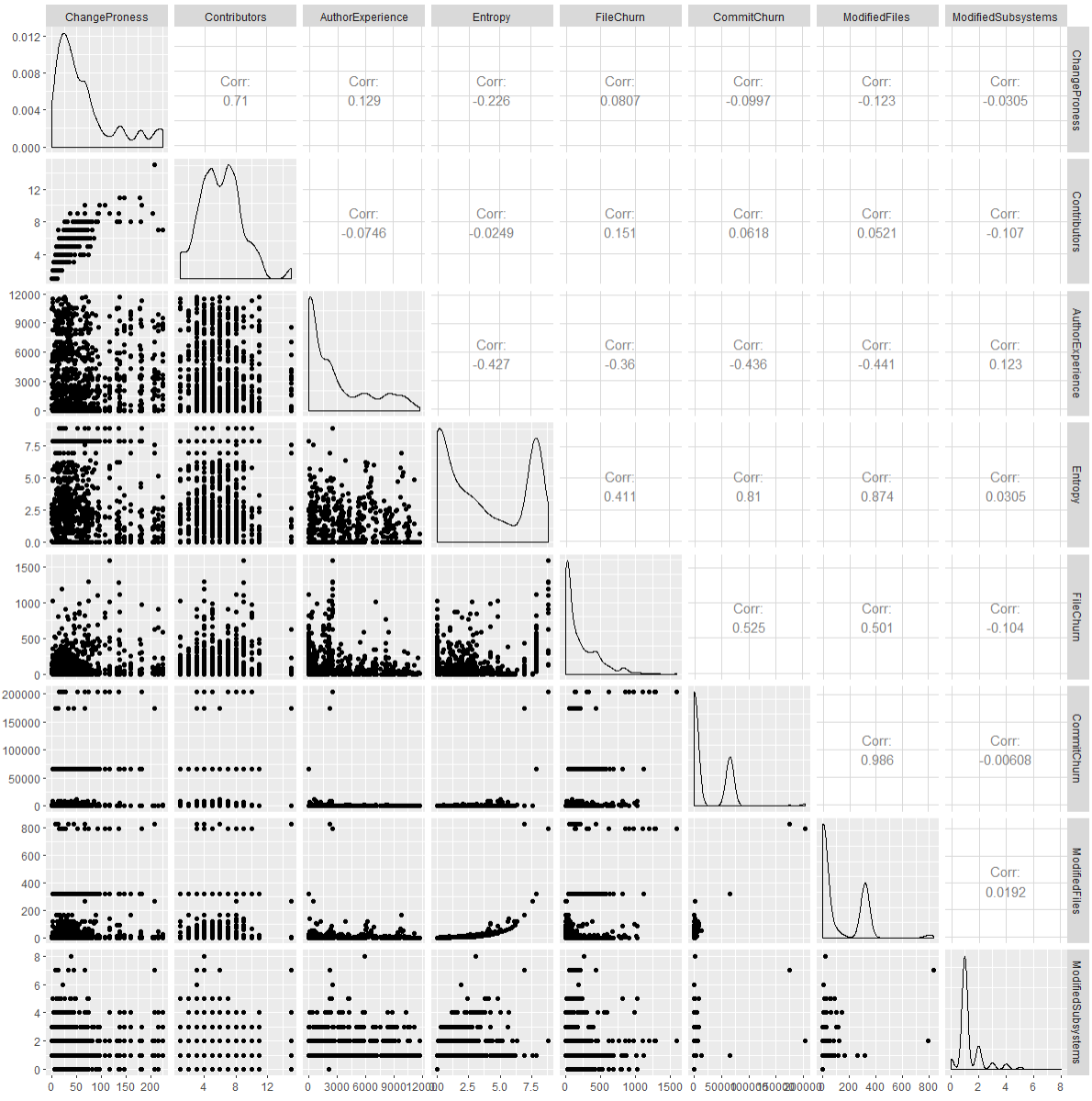

Figure B.21: JMeter - Change Proneness

B.6 Ant

Model: RemovalTimeInDays ∼ log(1+Contributors) + log(1+AuthorExperience) + CommitChurn + log(1+ModifiedSubsystems). **R-squared:** 0.1611

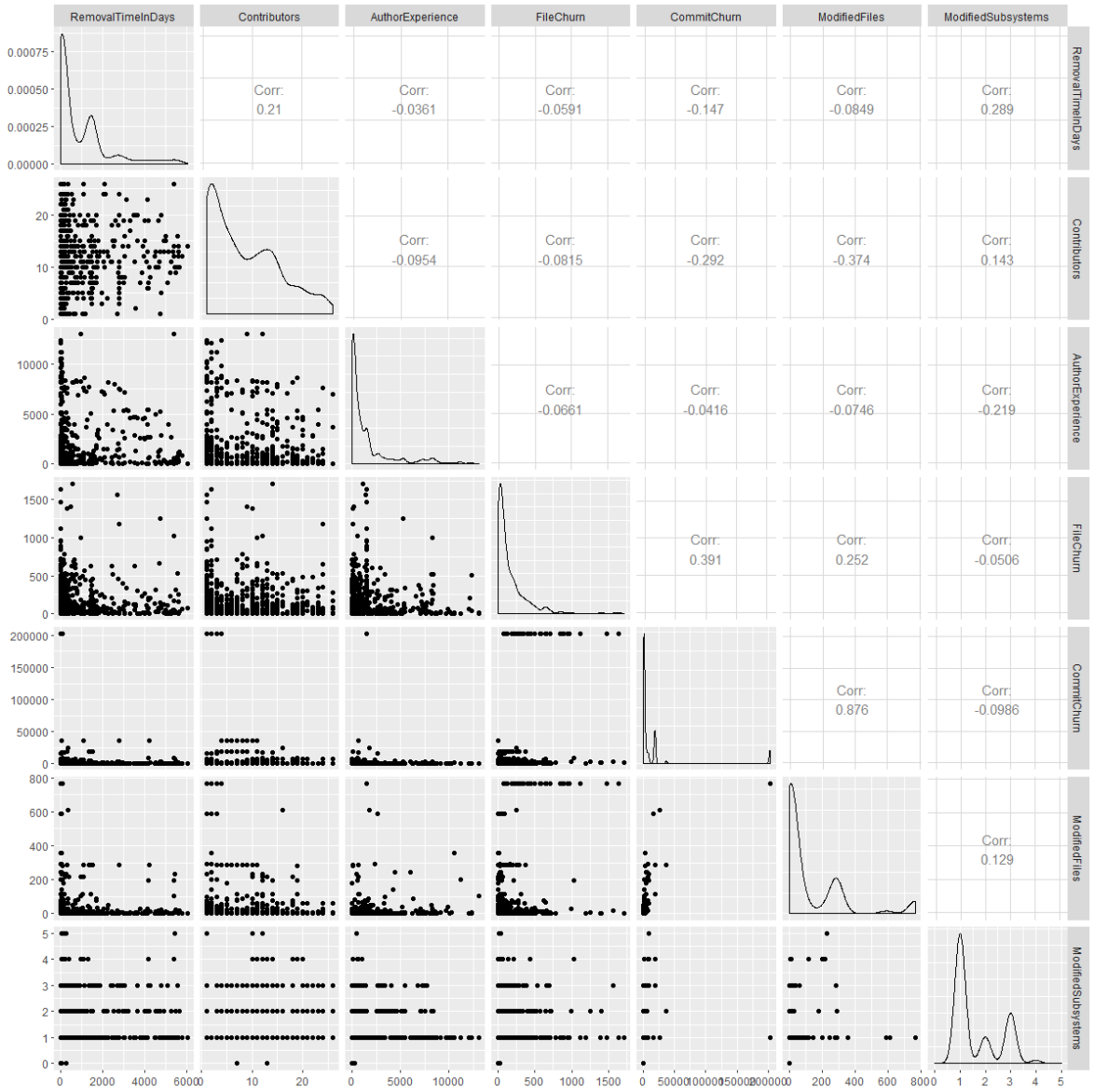

Figure B.22: Ant - Removal time (days).

Model: EffortInWords ∼ Contributors + FileChurn + log(1+CommitChurn) + ModifiedFiles + log(1+ModifiedSubsystems). **R-squared:** 0.1049

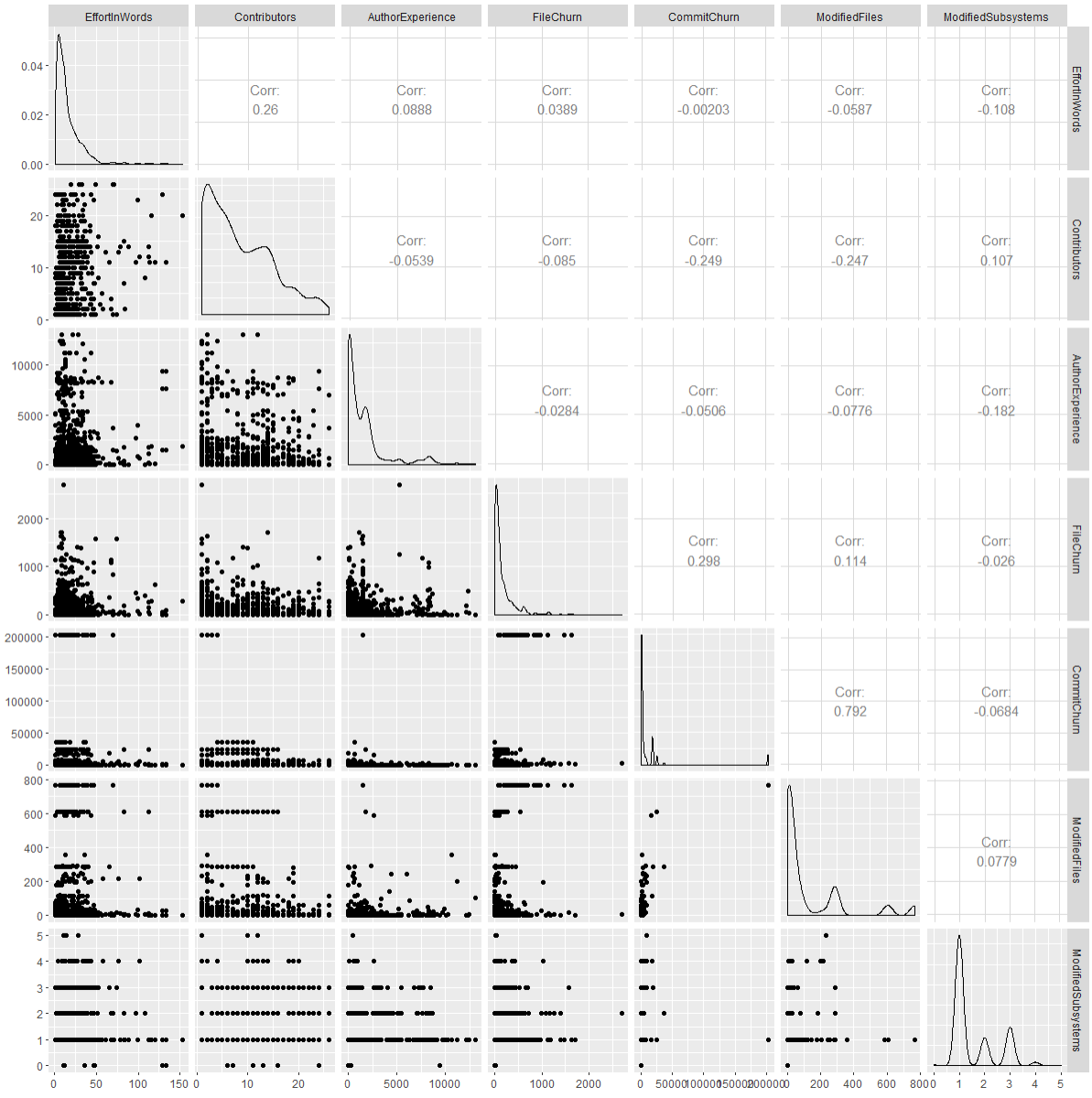

Figure B.23: Ant - Effort in Words (days).

Model: InterestFanIn ∼ Contributors + log(1+AuthorExperience) + ModifiedFiles + log(1+ModifiedSubsystems). **R-squared:** 0.02842

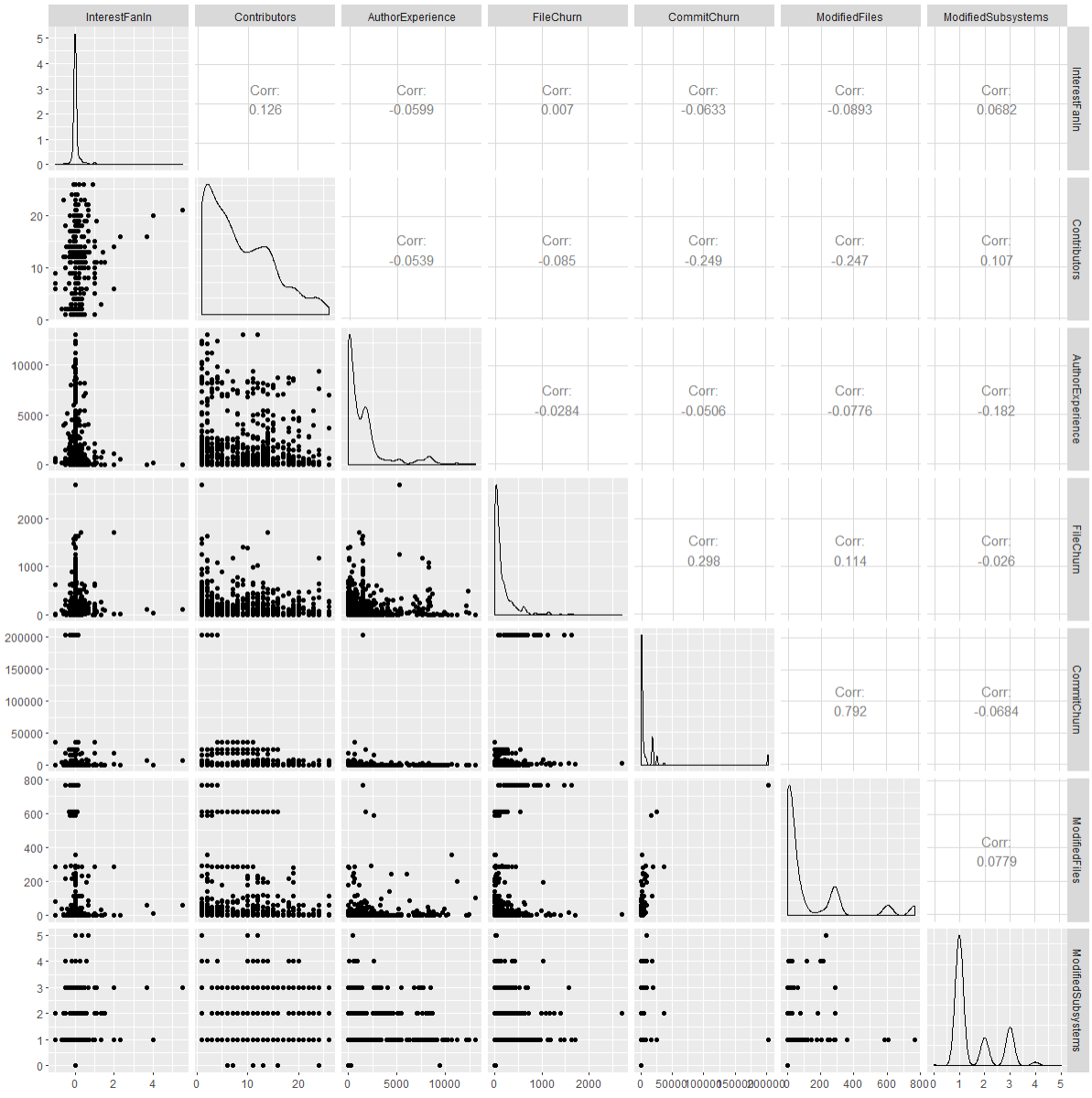

Figure B.24: Ant - Simple Interest - Fan-In

Model: InterestLOC ∼ log(1+AuthorExperience) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedSubsystems). **R-squared:** 0.02439

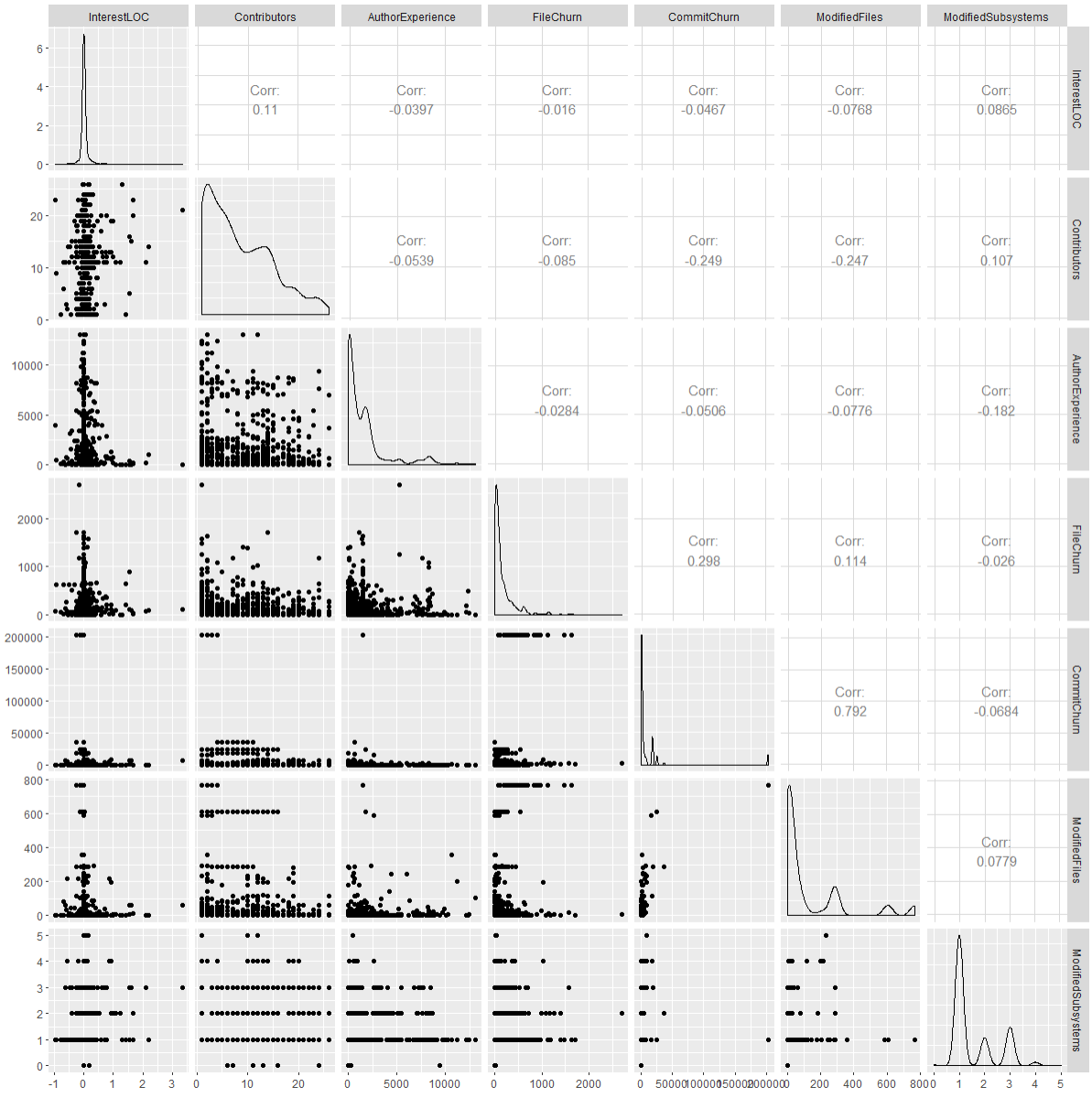

Figure B.25: Ant - Simple Interest - LOC

Model: CompoundedFanIn ∼ log(1+Contributors) + AuthorExperience + log(1+FileChurn) + ModifiedSubsystems. **R-squared:** 0.00153

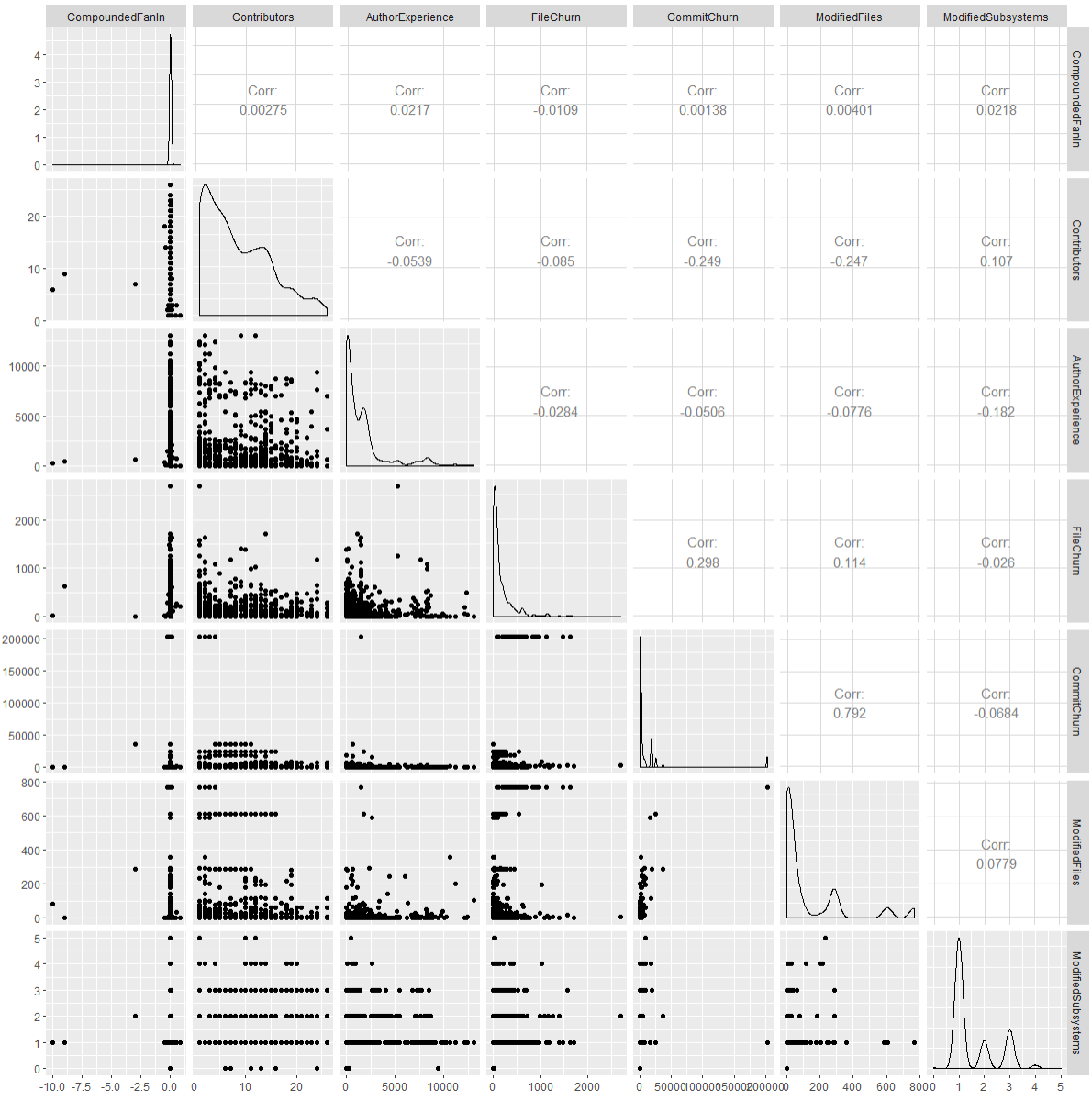

Figure B.26: Ant - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ log(1+Contributors) + log(1+AuthorExperience) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems).
R-squared: 0.004533

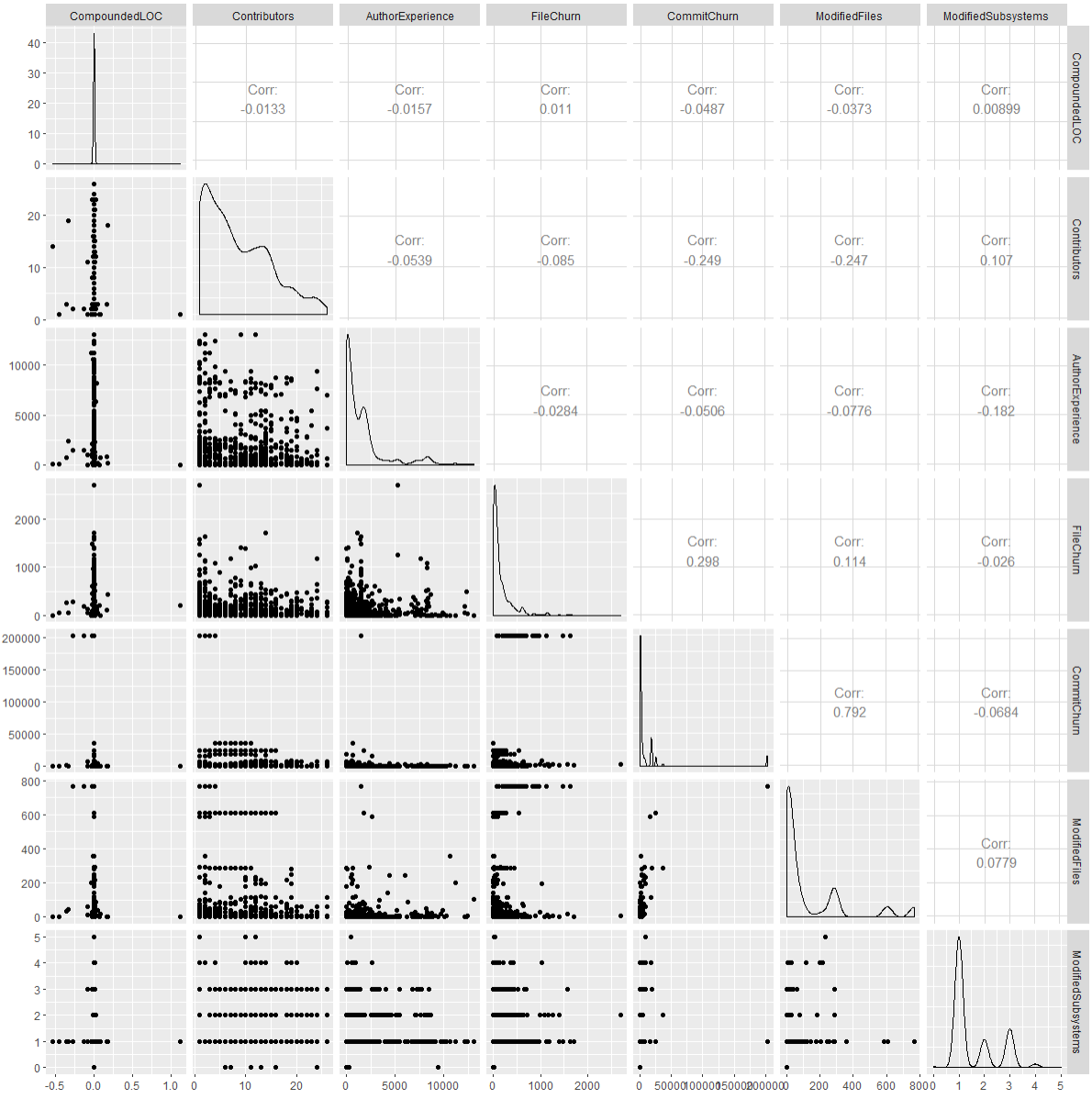

Figure B.27: Ant - Compounded Interest - LOC

Model: ChangeProness ∼ Contributors + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems). **R-squared:** 0.8271

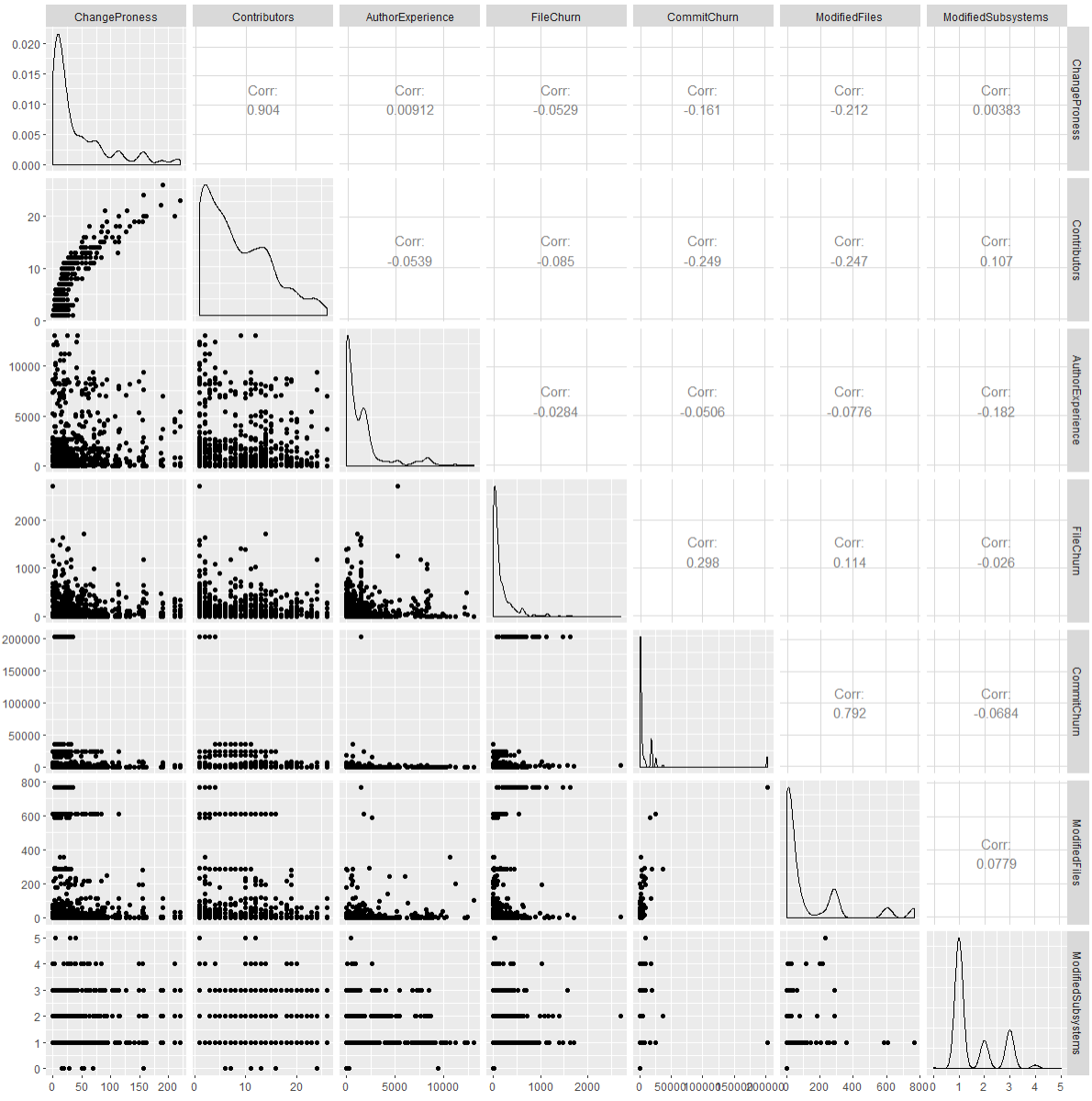

Figure B.28: Ant - Change Proneness

B.7 Hadoop

Model: RemovalTimeInDays ∼ BugFixingCommits + AuthorExperience + ModifiedDirectories + ModifiedSubsystems. **R-squared:** 0.04387

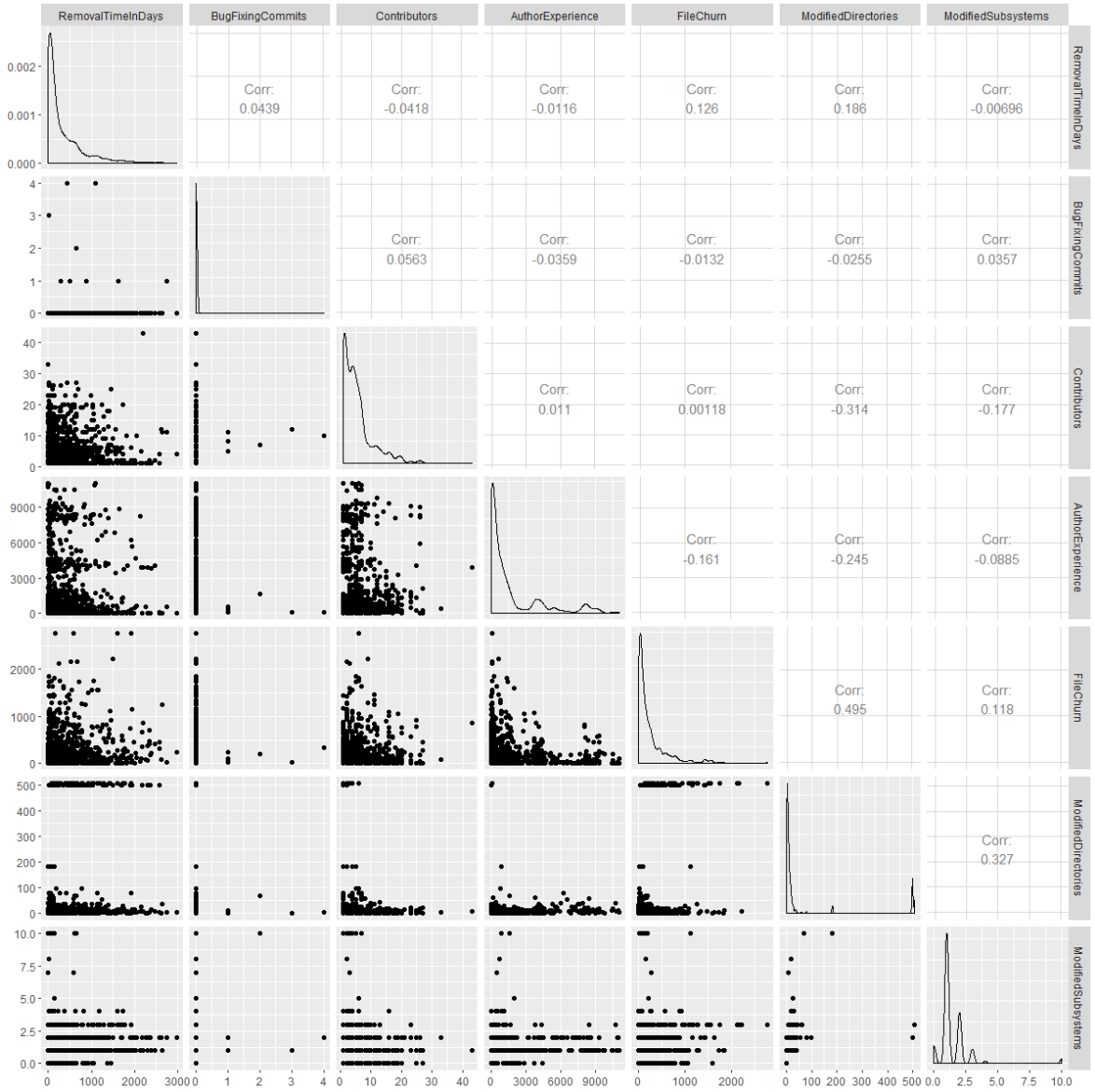

Figure B.29: Hadoop - Removal time (days).

Model: EffortInWords ∼ log(1+Contributors) + ModifiedDirectories + ModifiedSubsystems.
R-squared: 0.01269

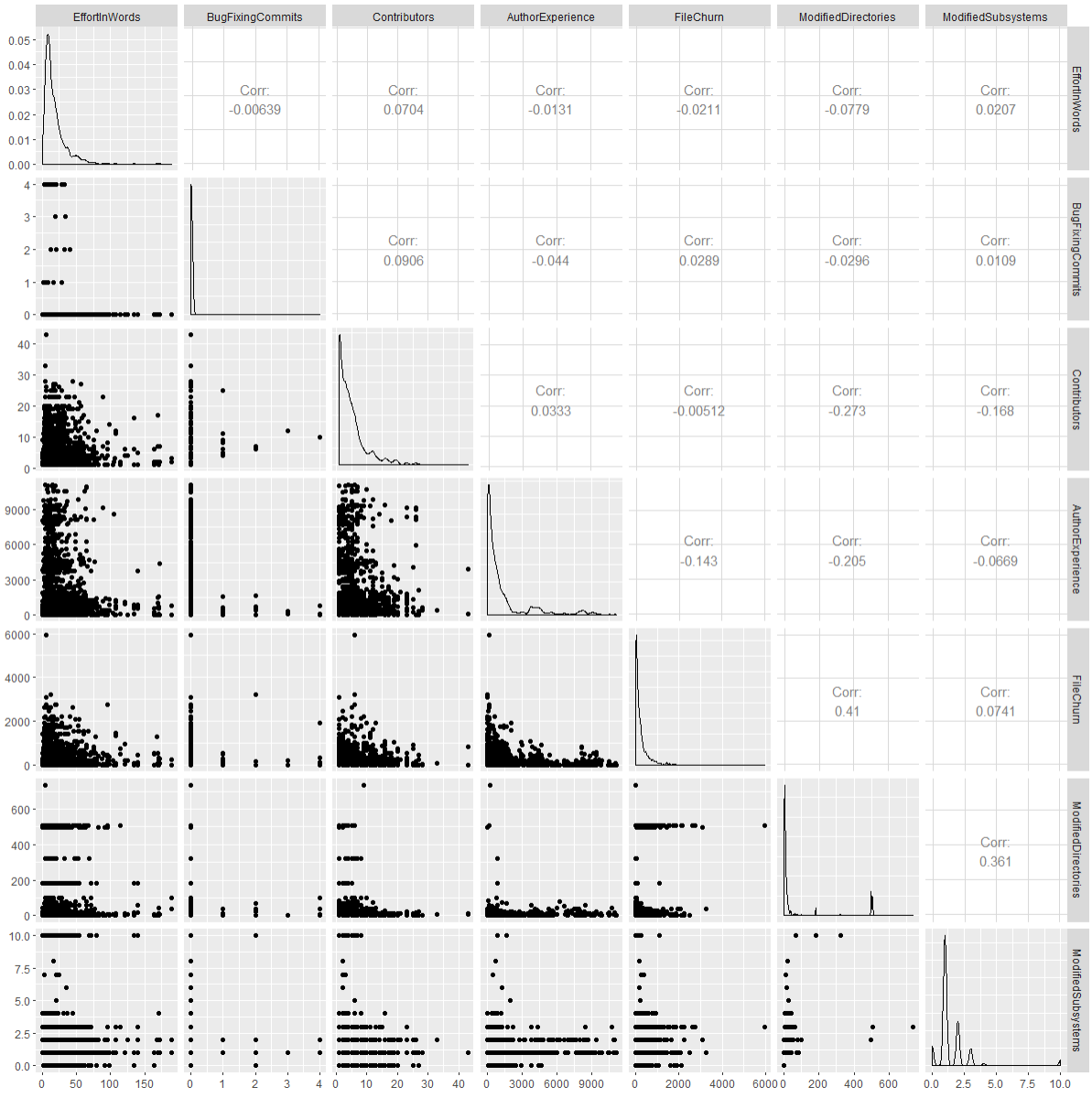

Figure B.30: Hadoop - Effort in Words (days).

Model: InterestFanIn ∼ BugFixingCommits + Contributors + log(1+ModifiedDirectories) + log(1+ModifiedSubsystems). **R-squared:** 0.006613

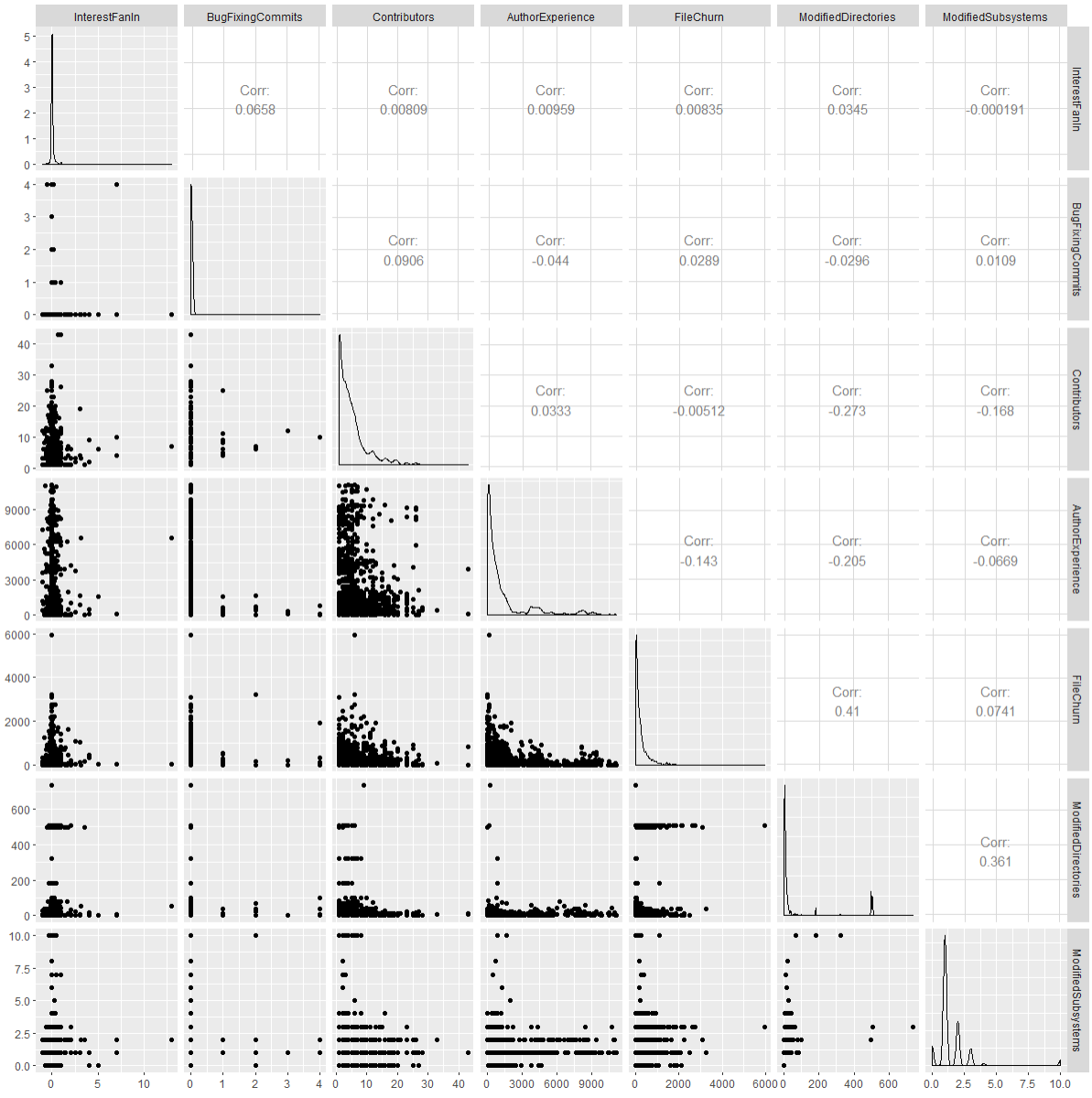

Figure B.31: Hadoop - Simple Interest - Fan-In

Model: InterestLOC ∼ BugFixingCommits + Contributors + ModifiedDirectories + ModifiedSubsystems. **R-squared:** 0.0448

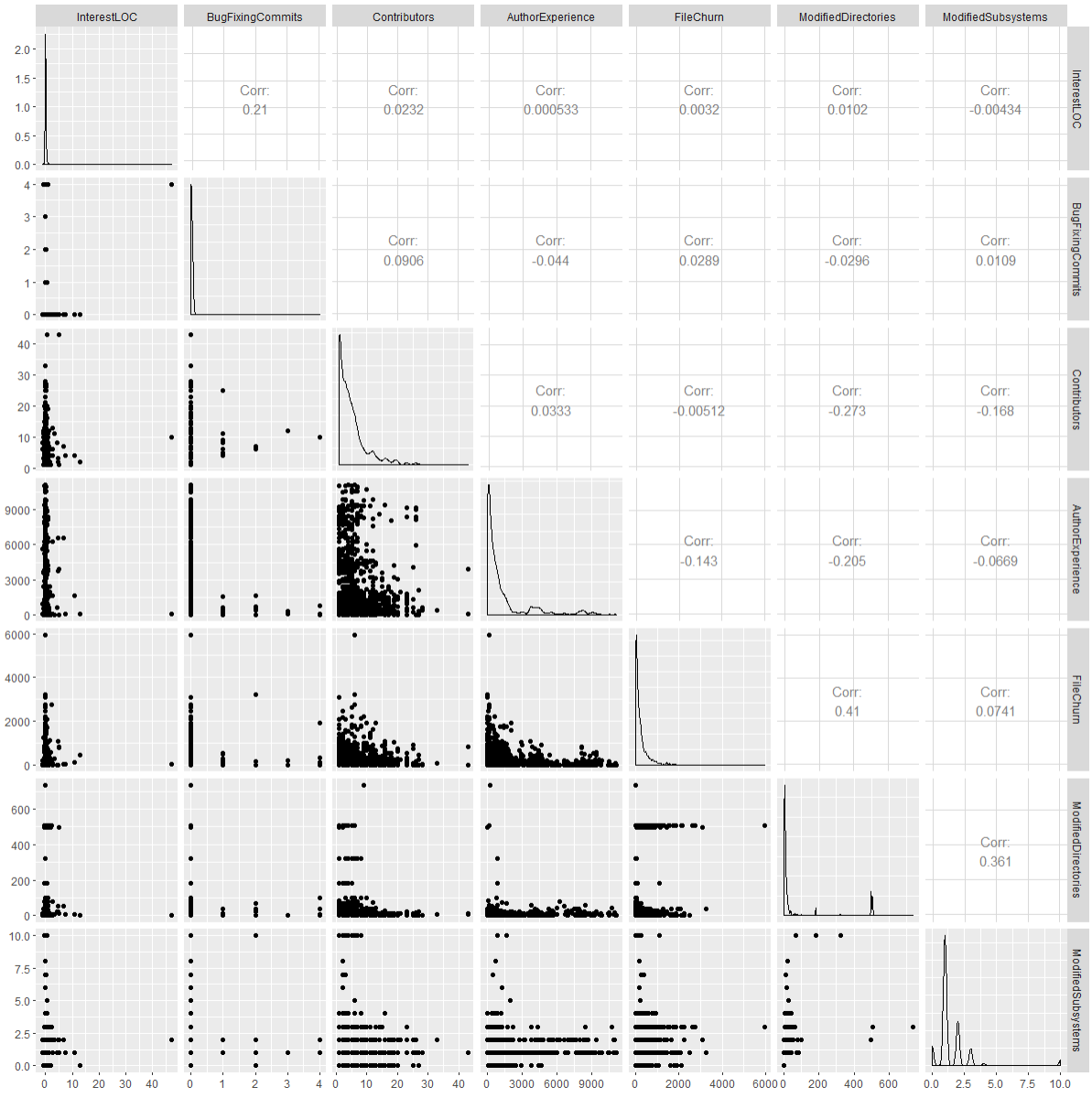

Figure B.32: Hadoop - Simple Interest - LOC

Model: CompoundedFanIn ∼ log(1+Contributors) + log(1+AuthorExperience) + log(1+FileChurn) + ModifiedSubsystems. **R-squared:** 0.000629

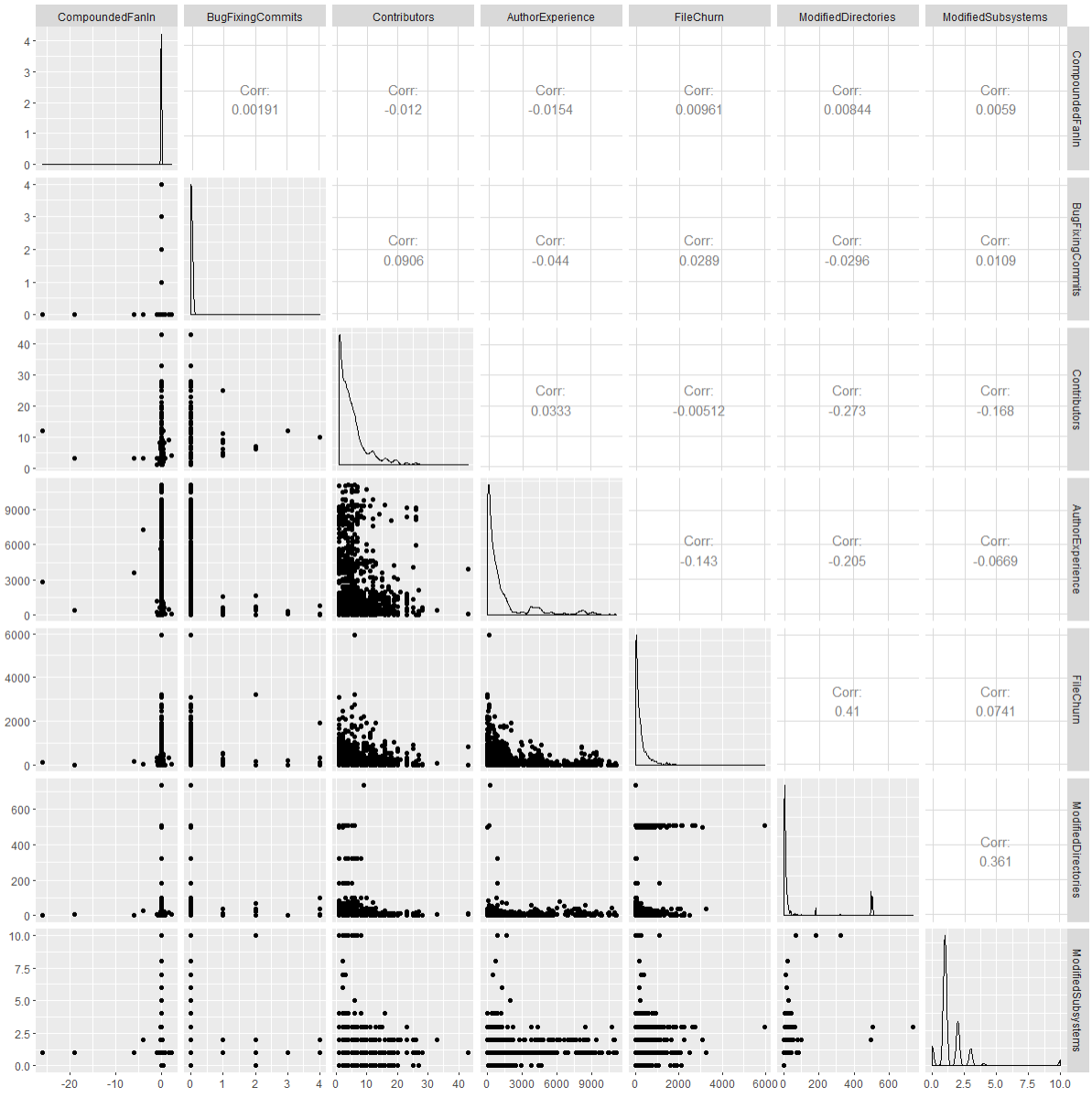

Figure B.33: Hadoop - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ AuthorExperience + FileChurn + ModifiedDirectories + ModifiedSubsystems. **R-squared:** 0.00152

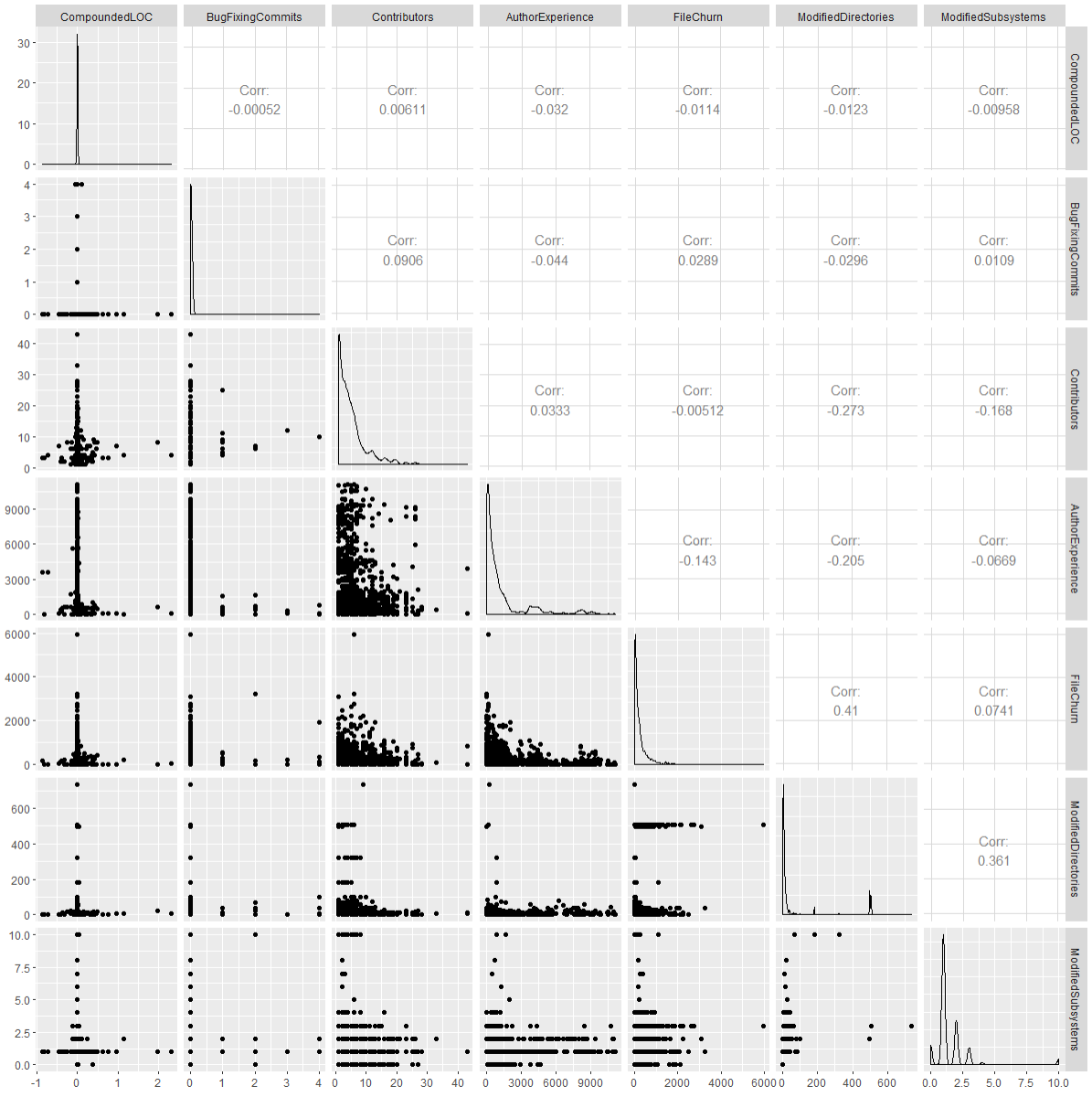

Figure B.34: Hadoop - Compounded Interest - LOC

Model: ChangeProness ∼ BugFixingCommits + Contributors. **R-squared:** 0.6952

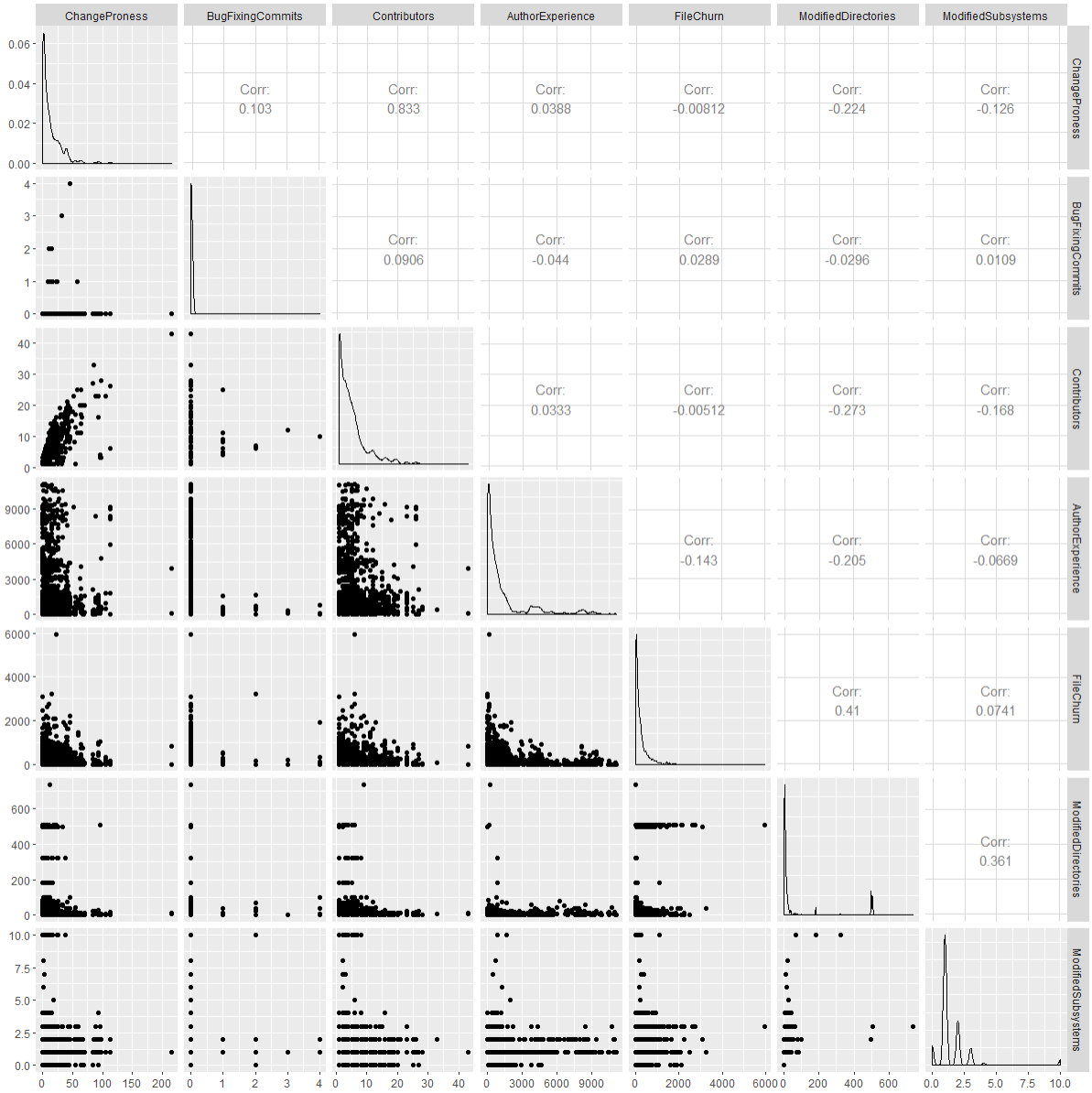

Figure B.35: Hadoop - Change Proneness

B.8 PMD

Model: RemovalTimeInDays ∼ log(1+BugFixingCommits) + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles). **R-squared:** 0.2741

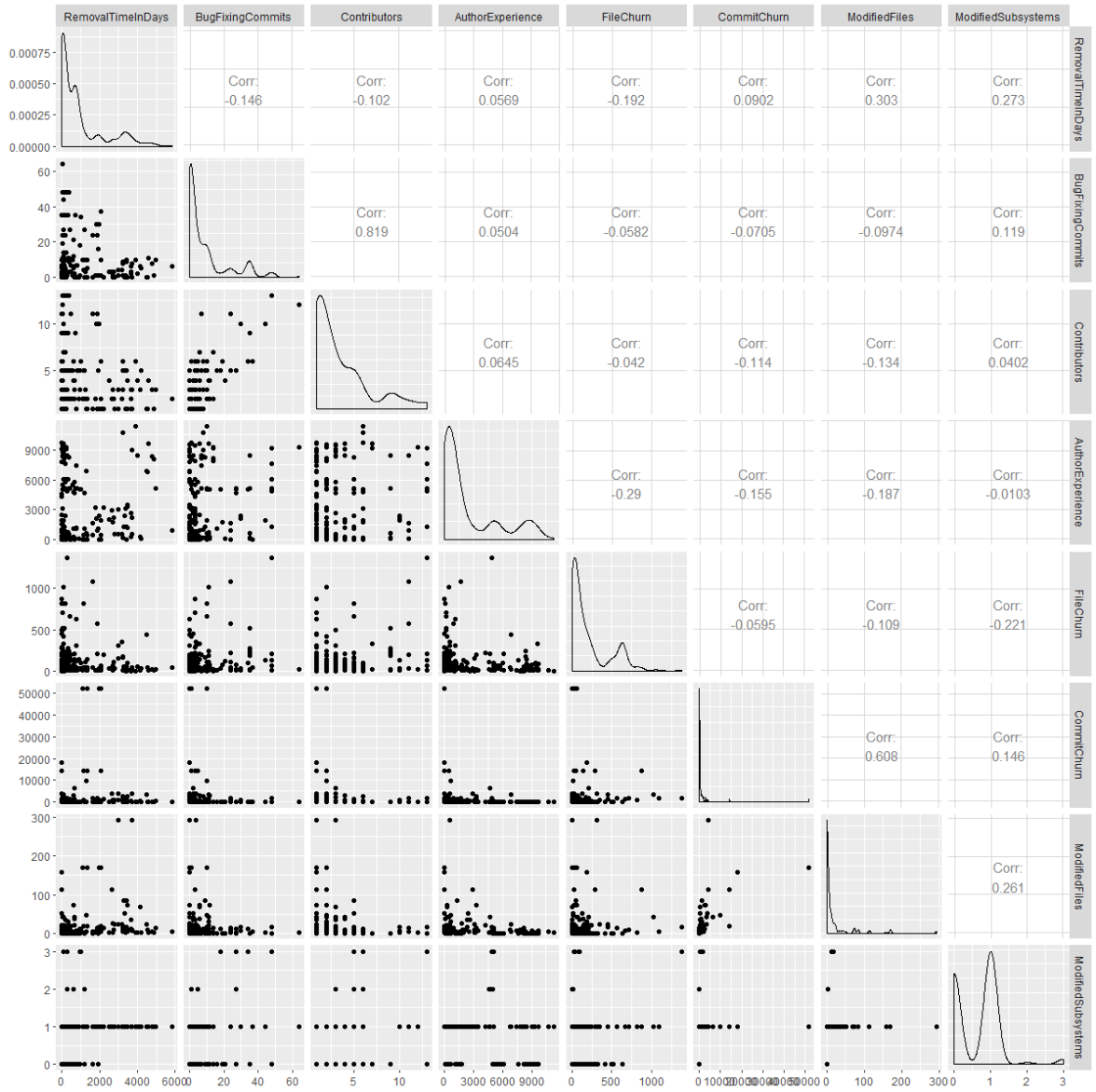

Figure B.36: PMD - Removal time (days).

Model: EffortInWords ∼ AuthorExperience + FileChurn + ModifiedFiles + ModifiedSubsystems.
R-squared: 0.01063

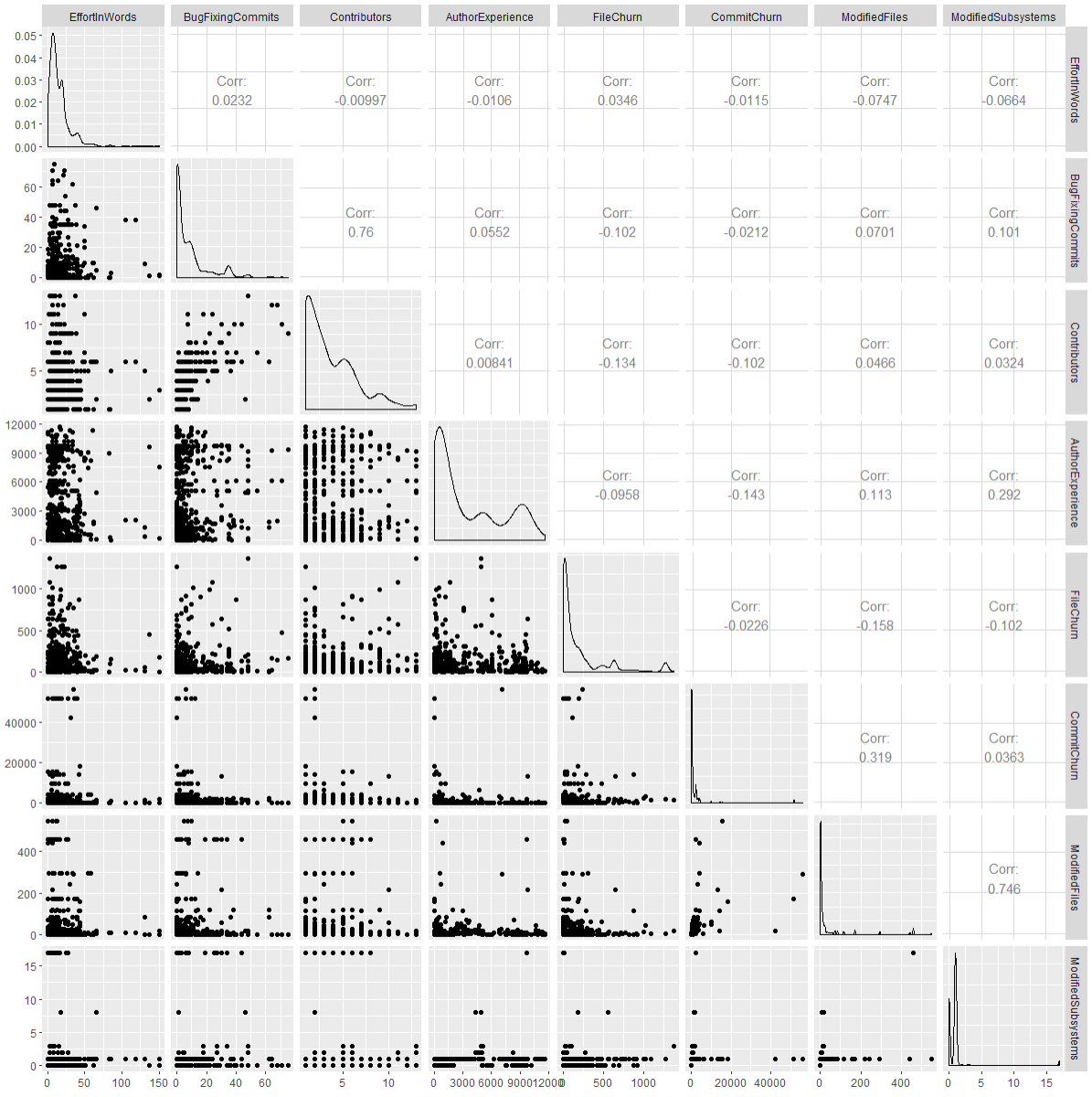

Figure B.37: PMD - Effort in Words (days).

Model: InterestFanIn ∼ log(1+BugFixingCommits) + log(1+FileChurn) + log(1+CommitChurn) + log(1+ModifiedSubsystems). **R-squared:** 0.06831

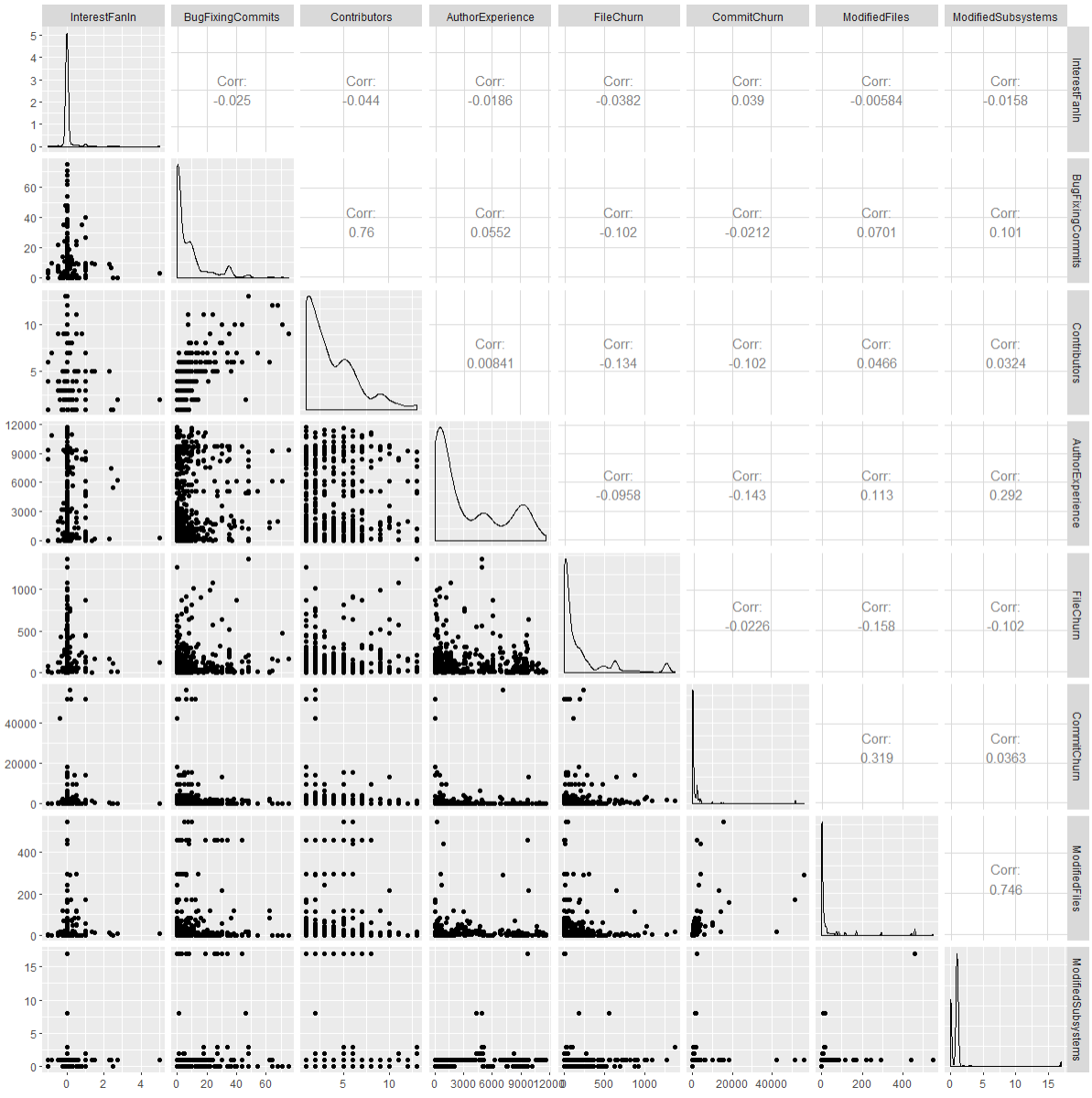

Figure B.38: PMD - Simple Interest - Fan-In

Model: InterestLOC ∼ log(1+BugFixingCommits) + Contributors + AuthorExperience + FileChurn + log(1+ModifiedFiles). **R-squared:** 0.05768

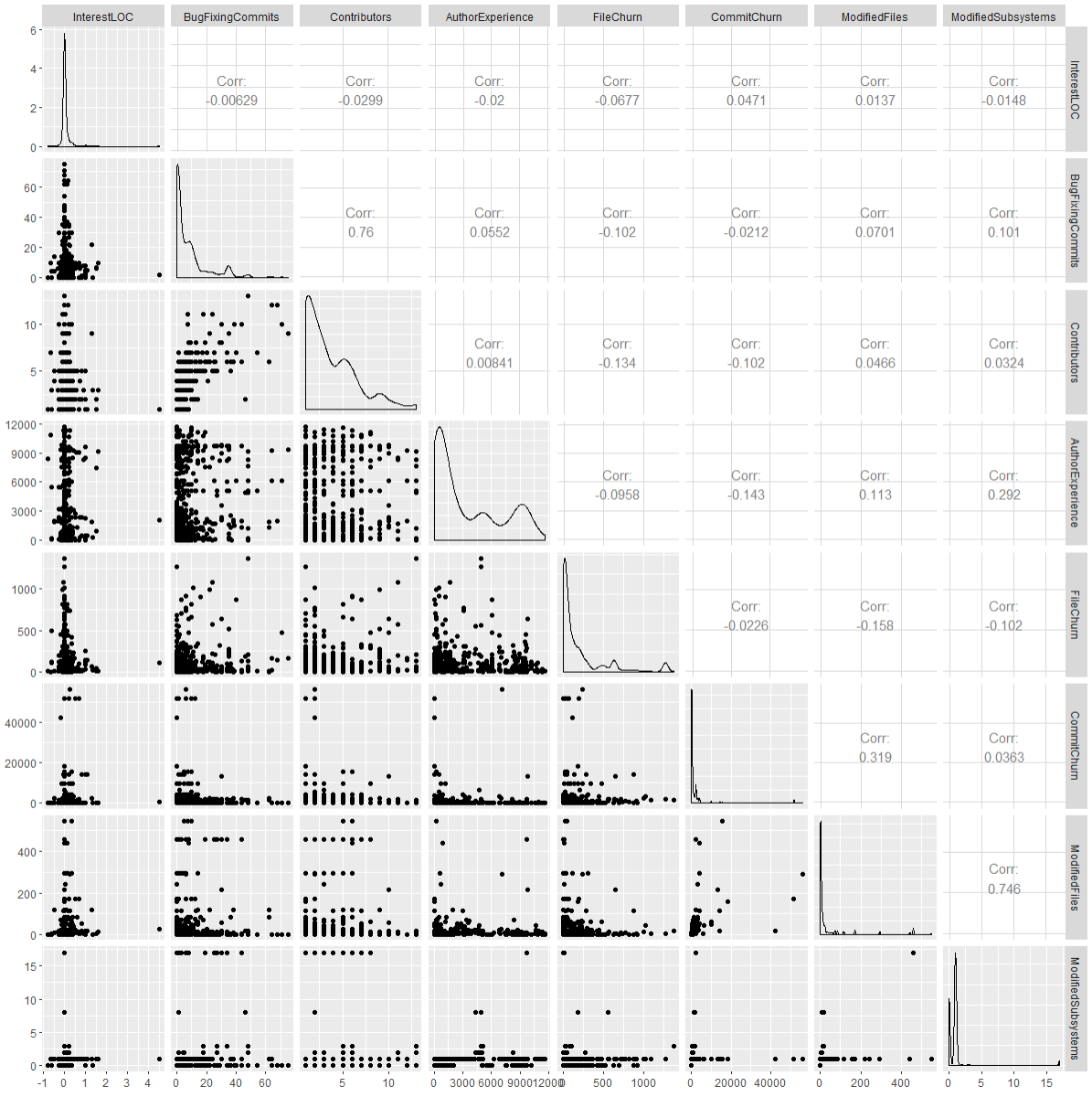

Figure B.39: PMD - Simple Interest - LOC

Model: CompoundedFanIn ∼ log(1+Contributors) + AuthorExperience + log(1+FileChurn) + log(1+ModifiedSubsystems). **R-squared:** 0.02168

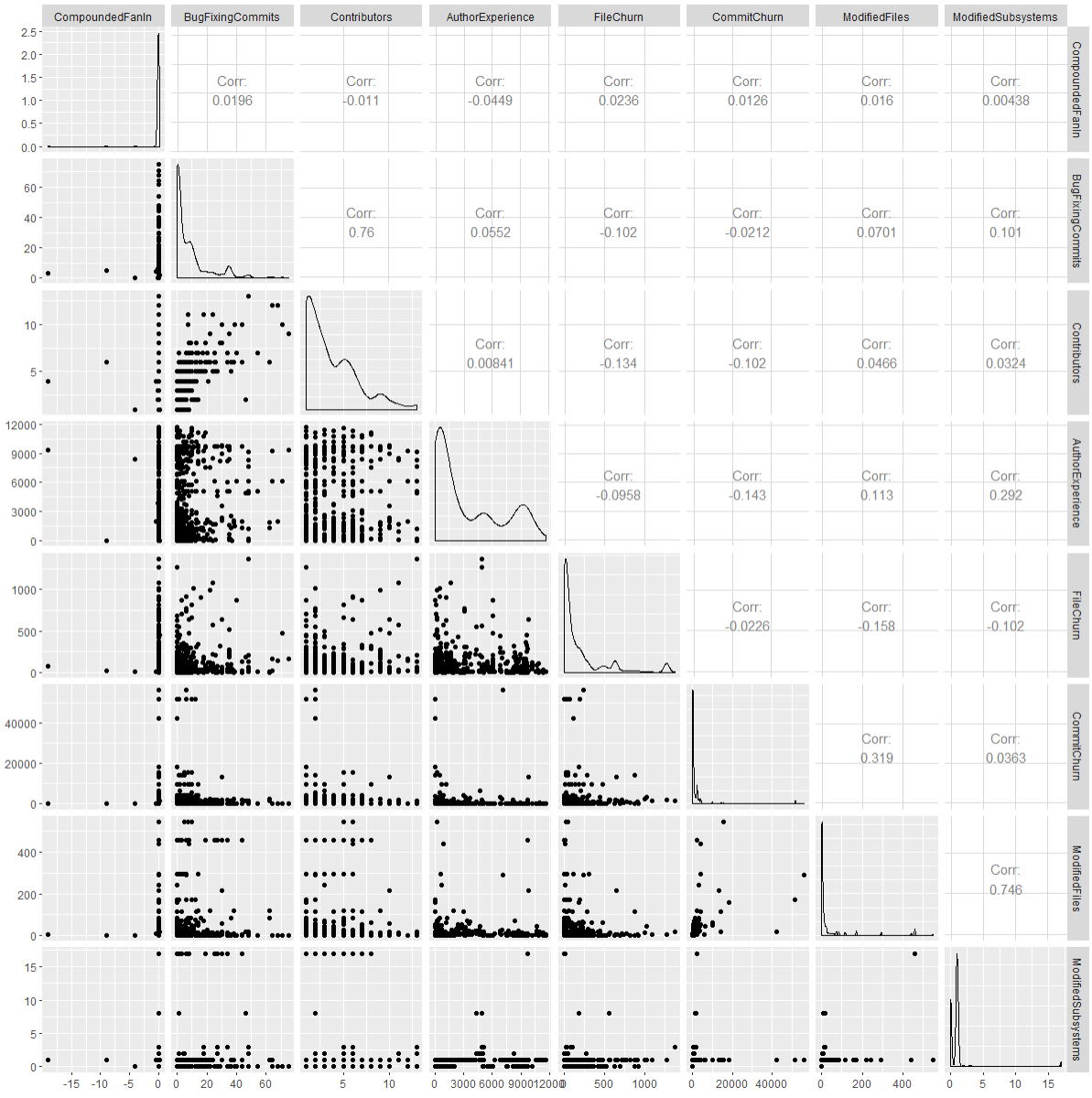

Figure B.40: PMD - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ log(1+BugFixingCommits) + AuthorExperience + FileChurn + ModifiedFiles. **R-squared:** 0.005869

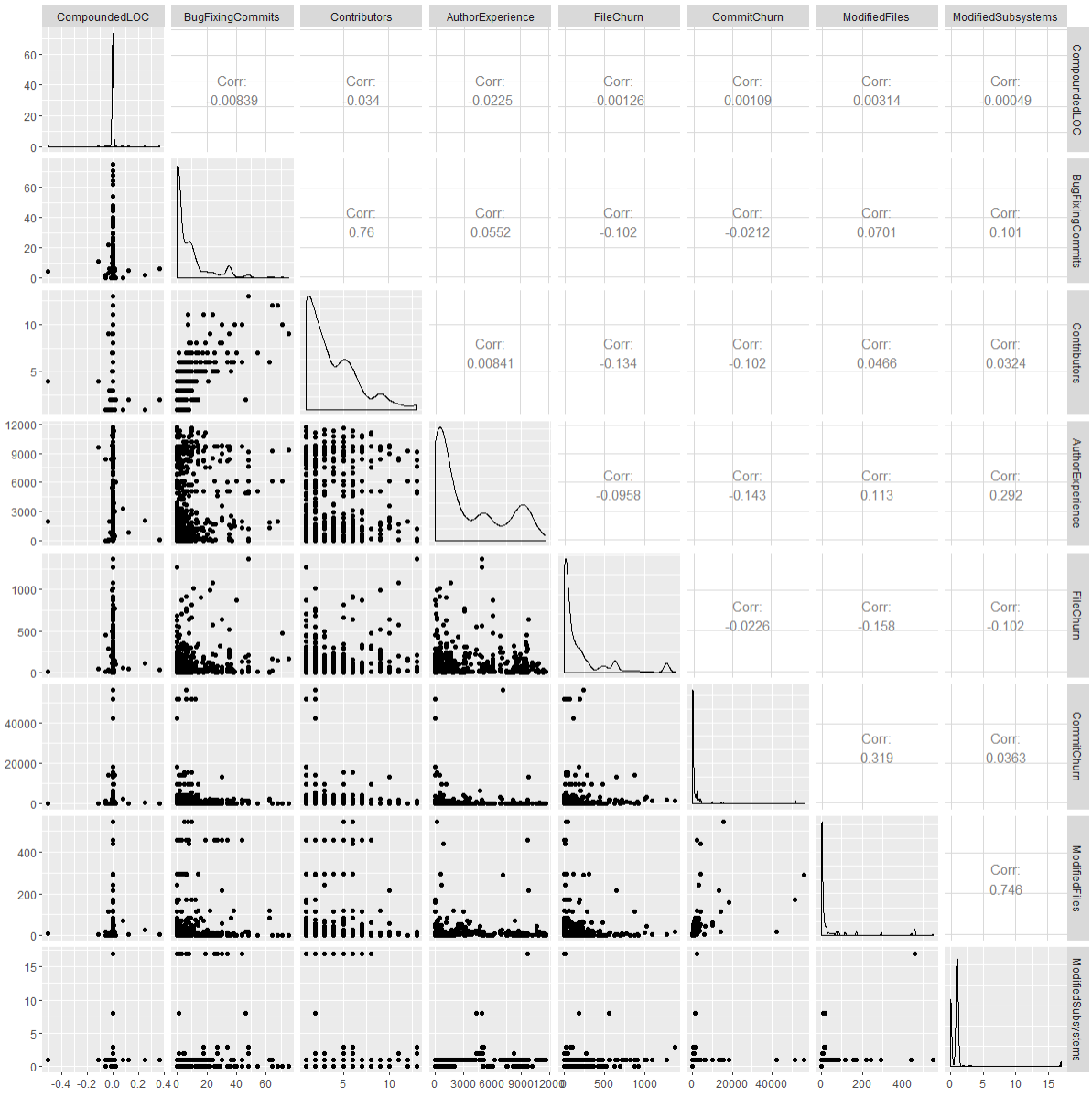

Figure B.41: PMD - Compounded Interest - LOC

Model: ChangeProness ∼ BugFixingCommits + Contributors + CommitChurn + log(1+ModifiedFiles). **R-squared:** 0.5412

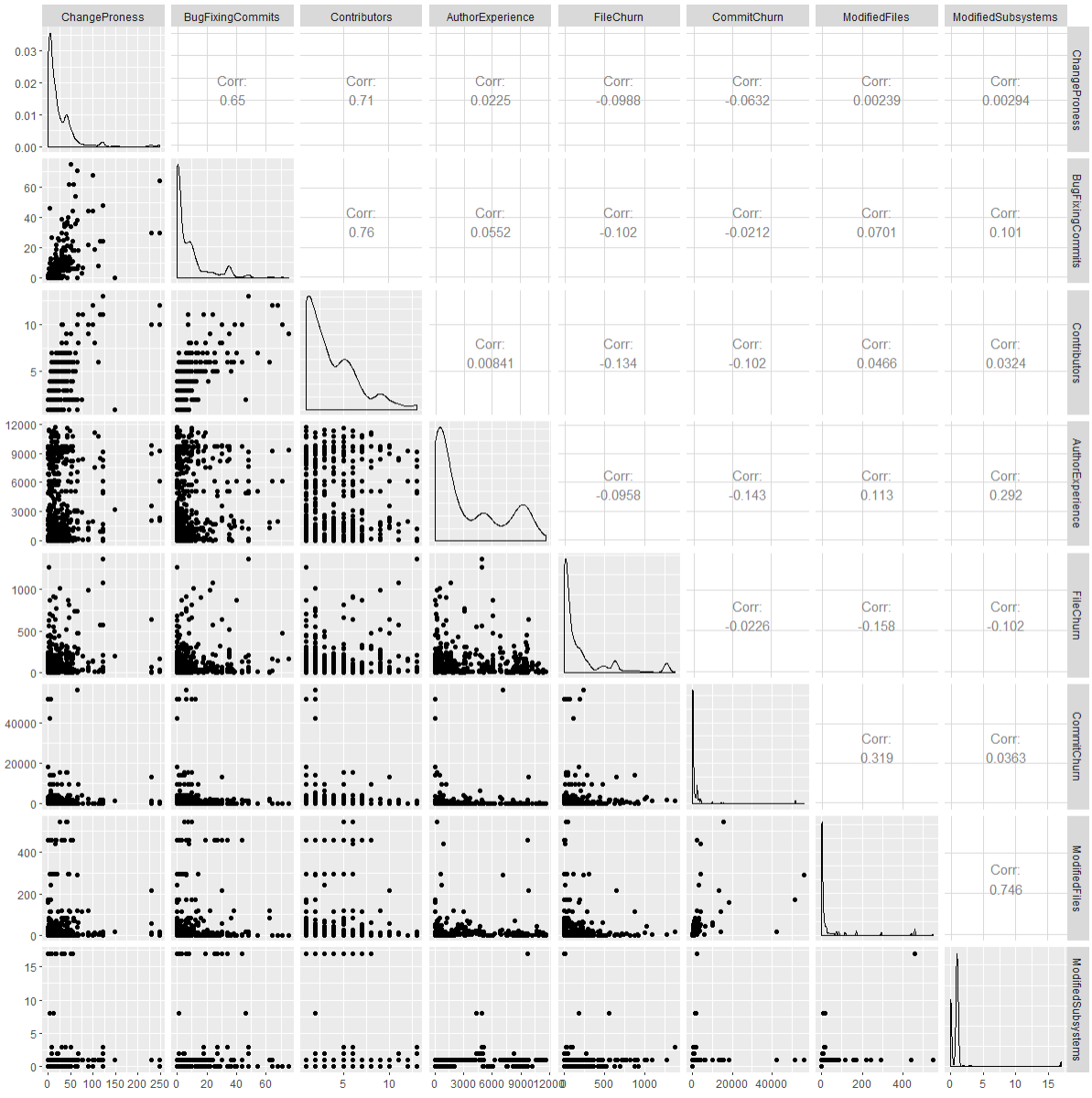

Figure B.42: PMD - Change Proneness

B.9 EMF

Model: RemovalTimeInDays ∼ log(1+BugFixingCommits) + AuthorExperience + log(1+FileChurn) + log(1+ModifiedFiles). **R-squared:** 0.2952

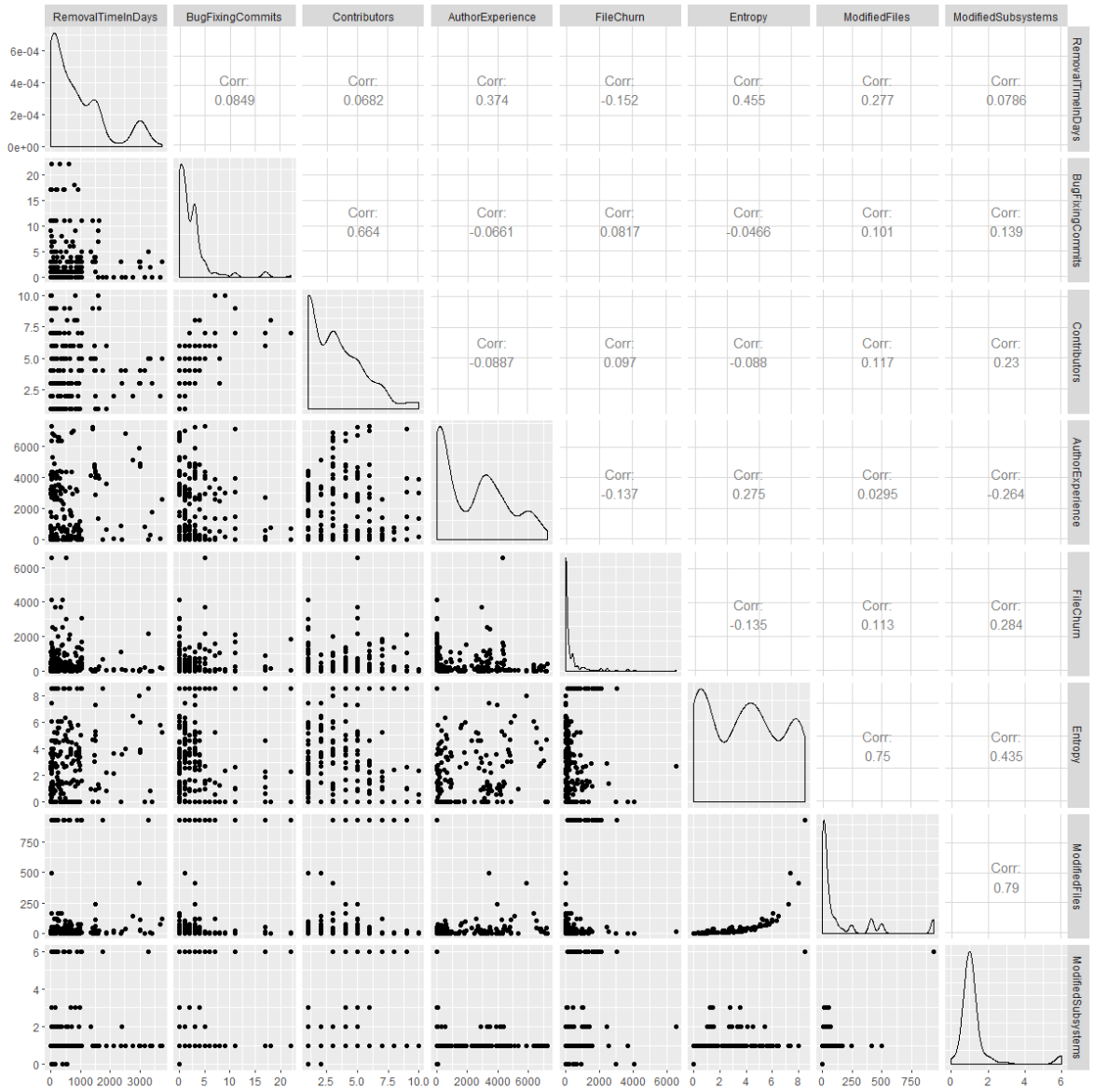

Figure B.43: EMF - Removal time (days).

Model: EffortInWords ∼ BugFixingCommits + Contributors + log(1+FileChurn) + log(1+Entropy) + log(1+ModifiedFiles) + ModifiedSubsystems. **R-squared:** 0.2904

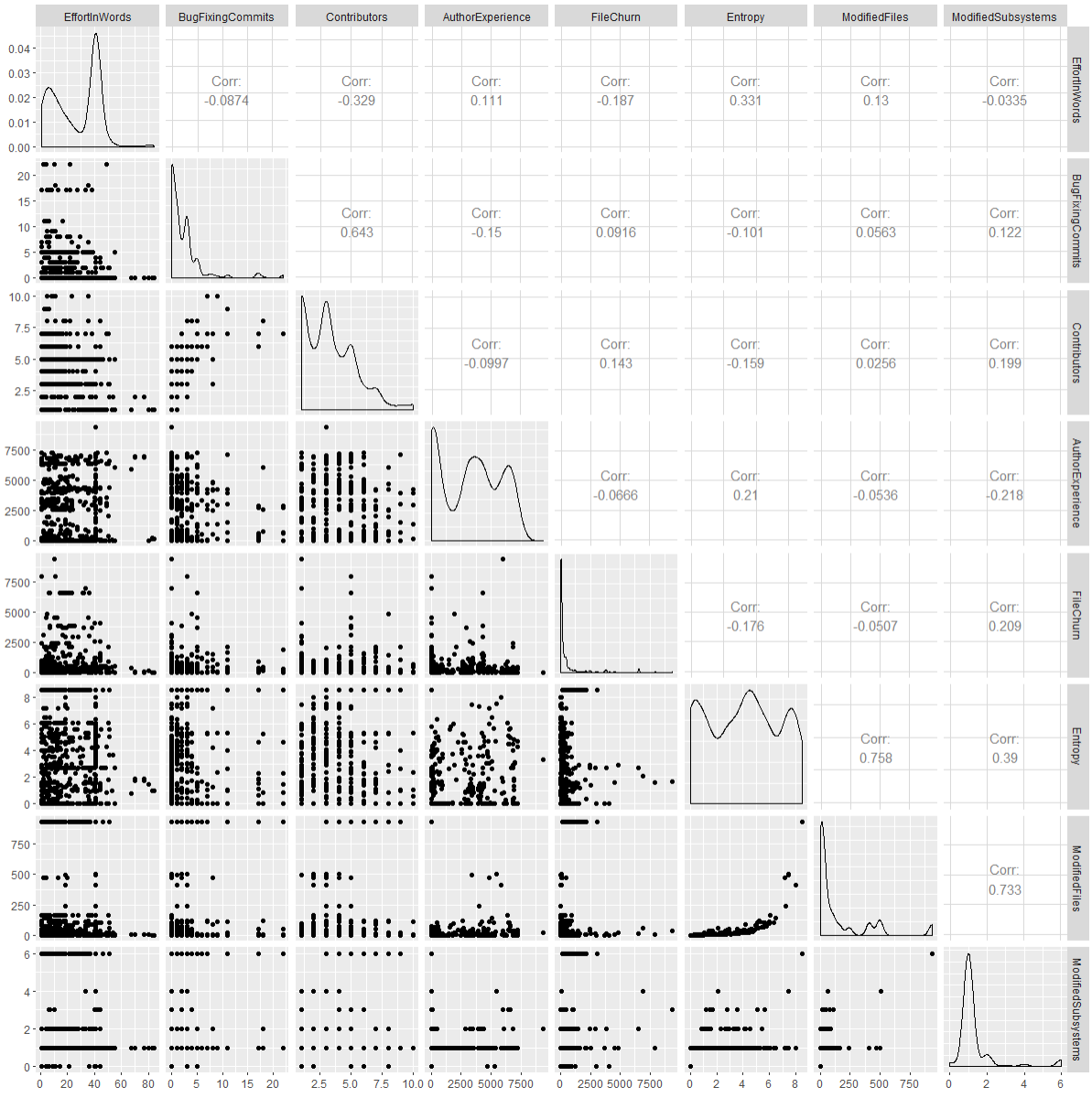

Figure B.44: EMF - Effort in Words (days).

Model: InterestFanIn ∼ BugFixingCommits + log(1+FileChurn) + Entropy + ModifiedSubsystems. **R-squared:** 0.01201

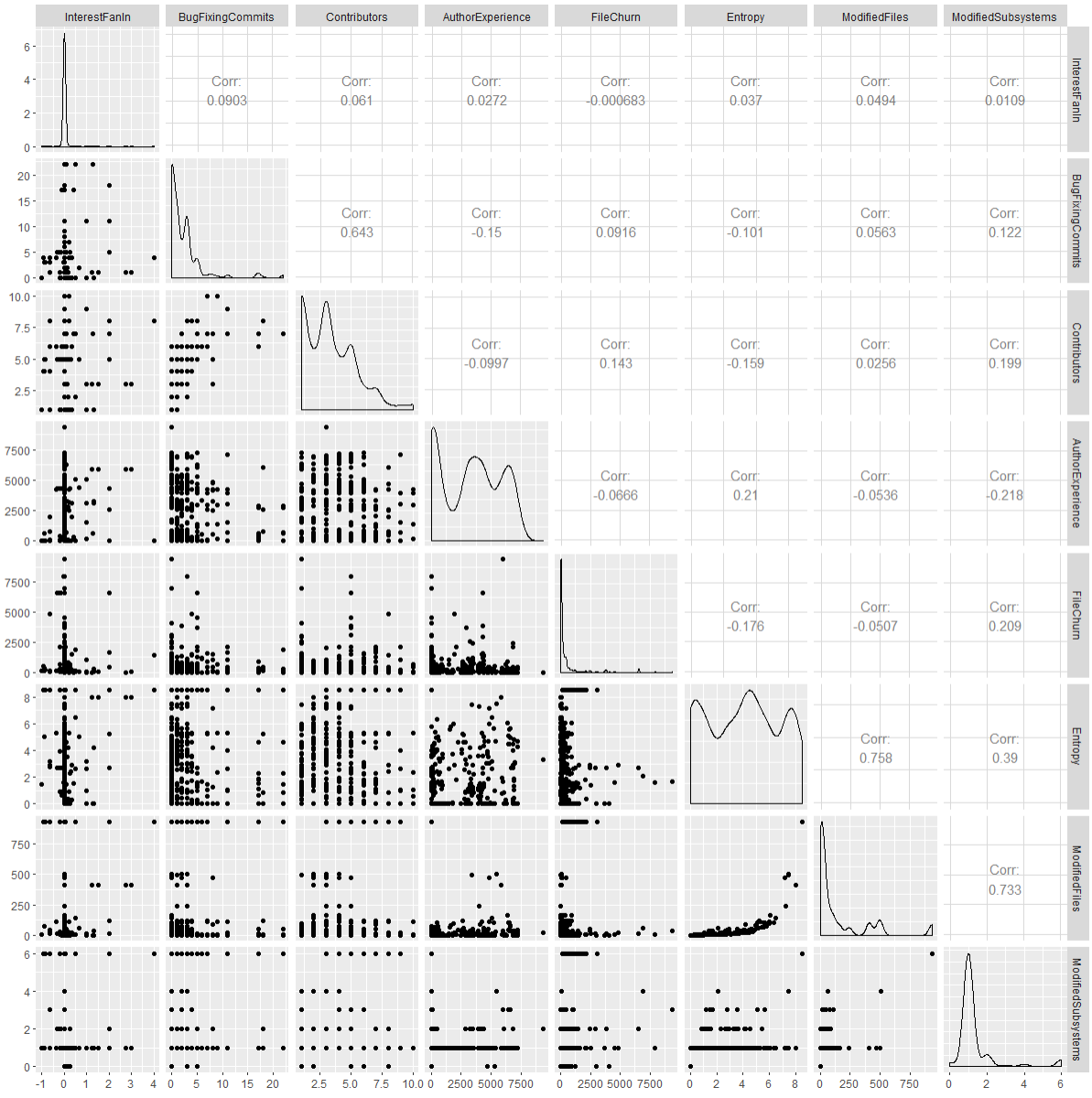

Figure B.45: EMF - Simple Interest - Fan-In

Model: InterestLOC ∼ BugFixingCommits + log(1+FileChurn) + Entropy + ModifiedSubsystems. **R-squared:** 0.06

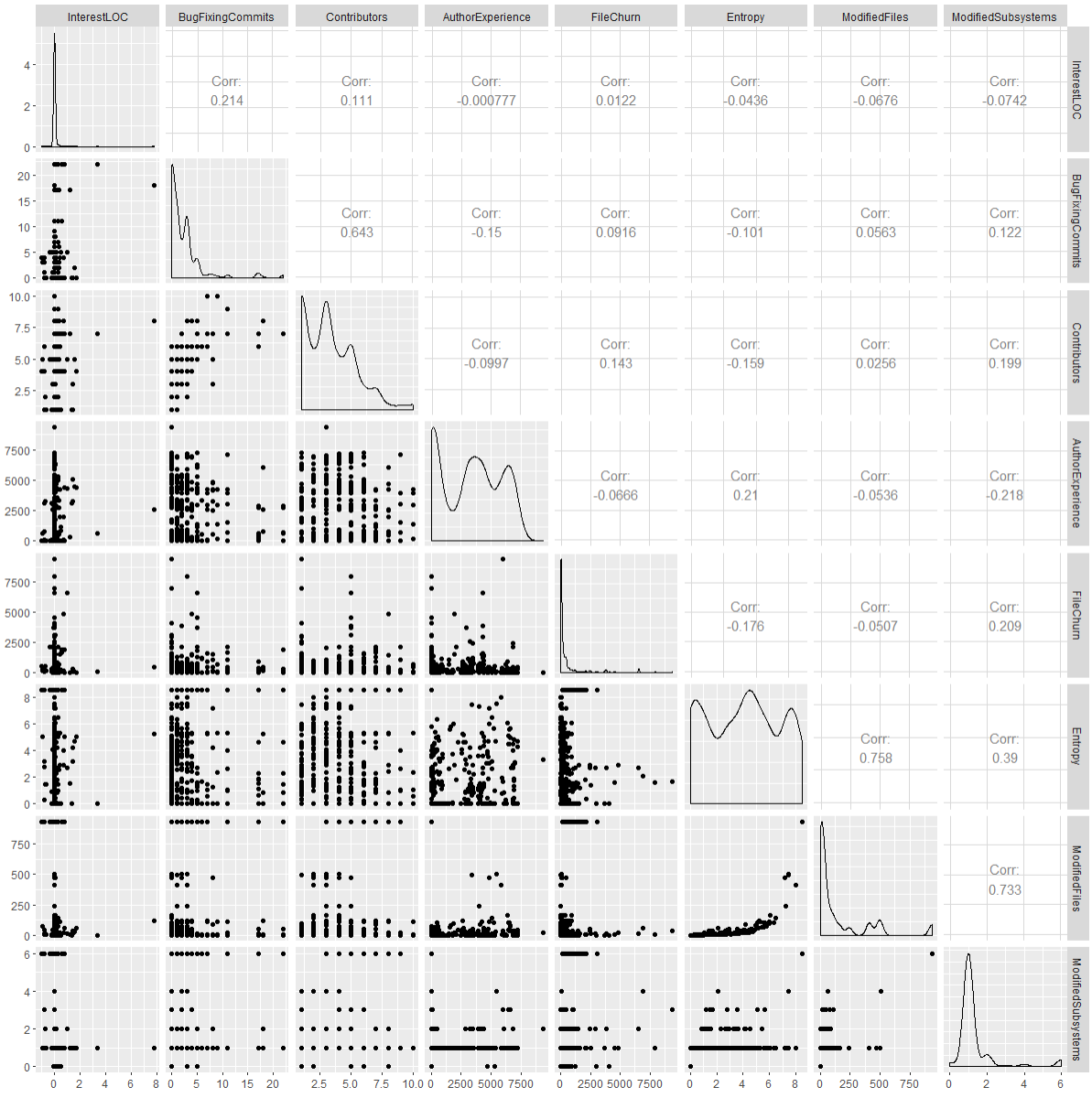

Figure B.46: EMF - Simple Interest - LOC

Model: CompoundedFanIn ∼ log(1+BugFixingCommits) + log(1+Contributors) + AuthorExperience + Entropy + log(1+ModifiedFiles). **R-squared:** 0.003876

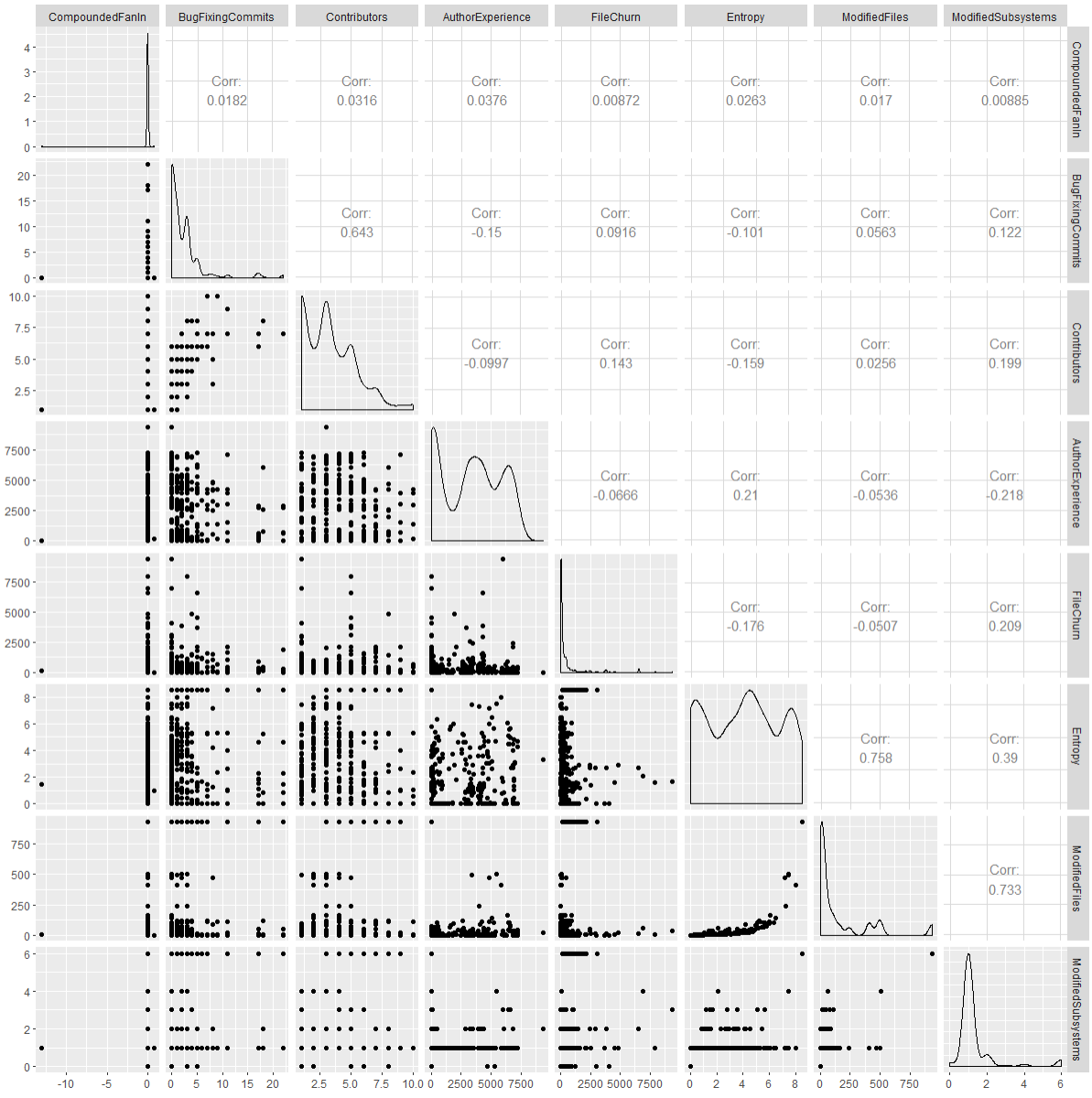

Figure B.47: EMF - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ log(1+Contributors) + log(1+AuthorExperience) + FileChurn + log(1+ModifiedSubsystems). **R-squared:** 0.04082

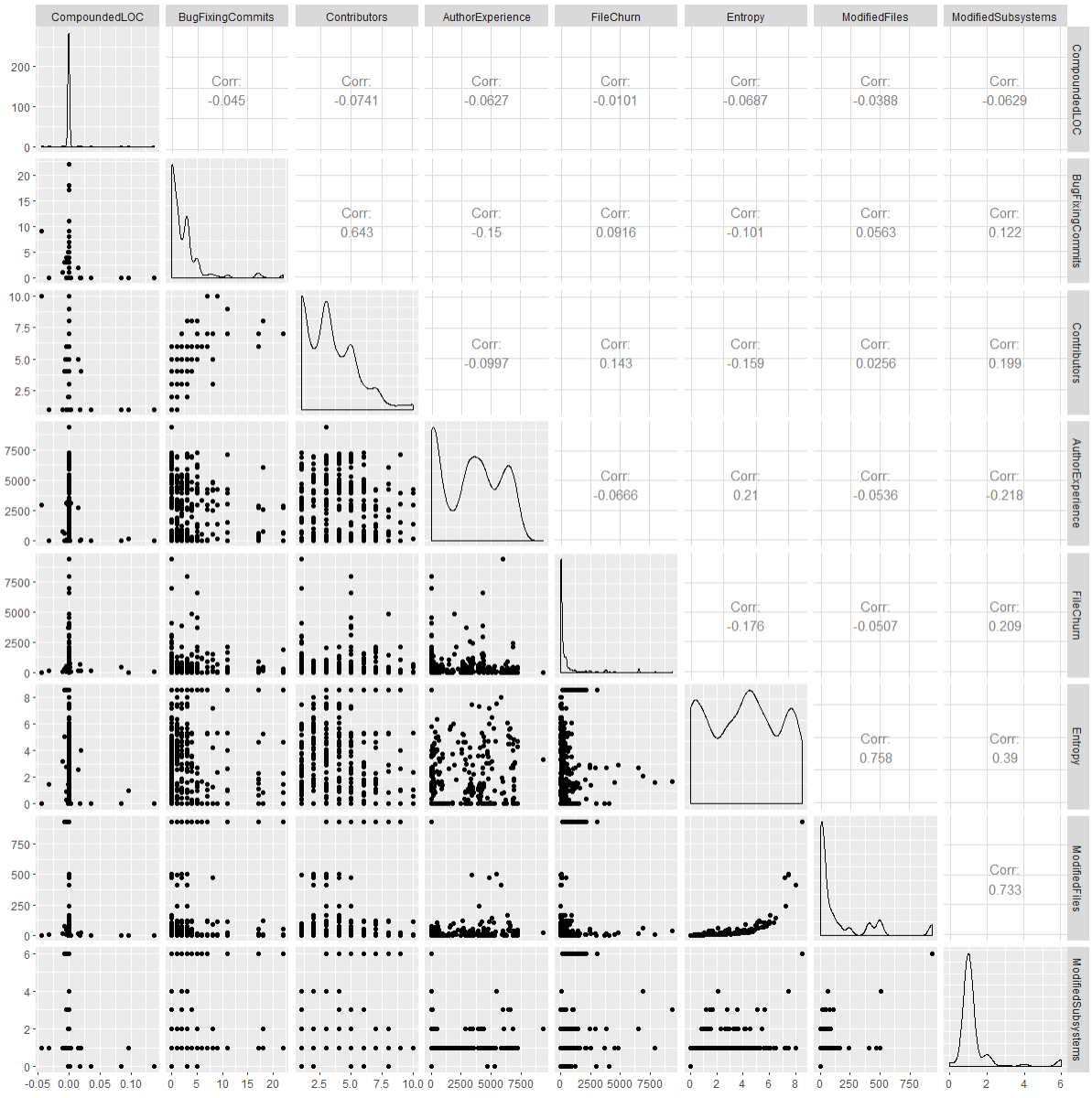

Figure B.48: EMF - Compounded Interest - LOC

Model: ChangeProness ∼ BugFixingCommits + Contributors + log(1+FileChurn) + log(1+Entropy). **R-squared:** 0.7927

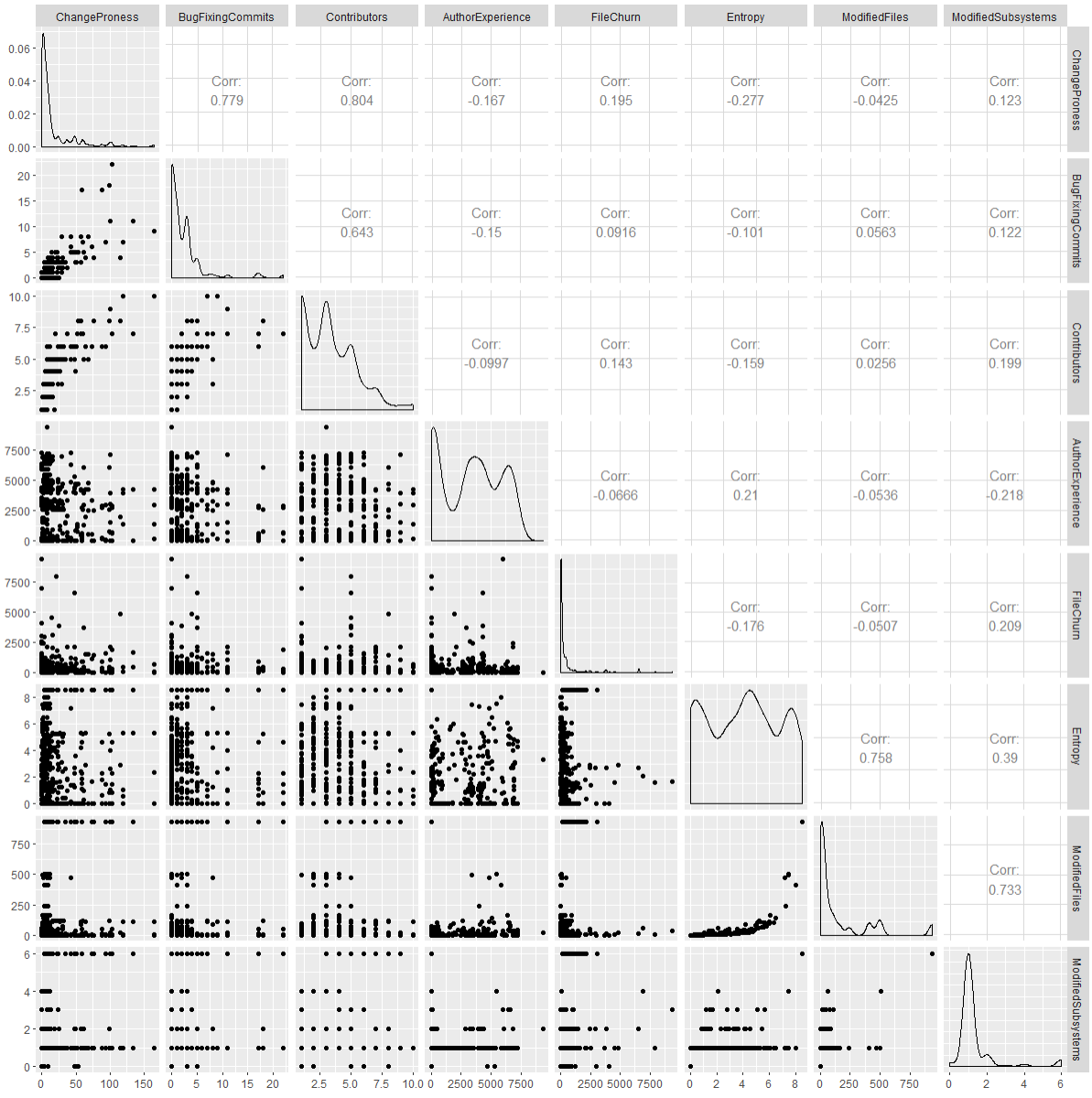

Figure B.49: EMF - Change Proneness

B.10 Tomcat

Model: RemovalTimeInDays ∼ log(1+Contributors) + log(1+AuthorExperience) + log(1+ModifiedSubsystems). **R-squared:** 0.2672

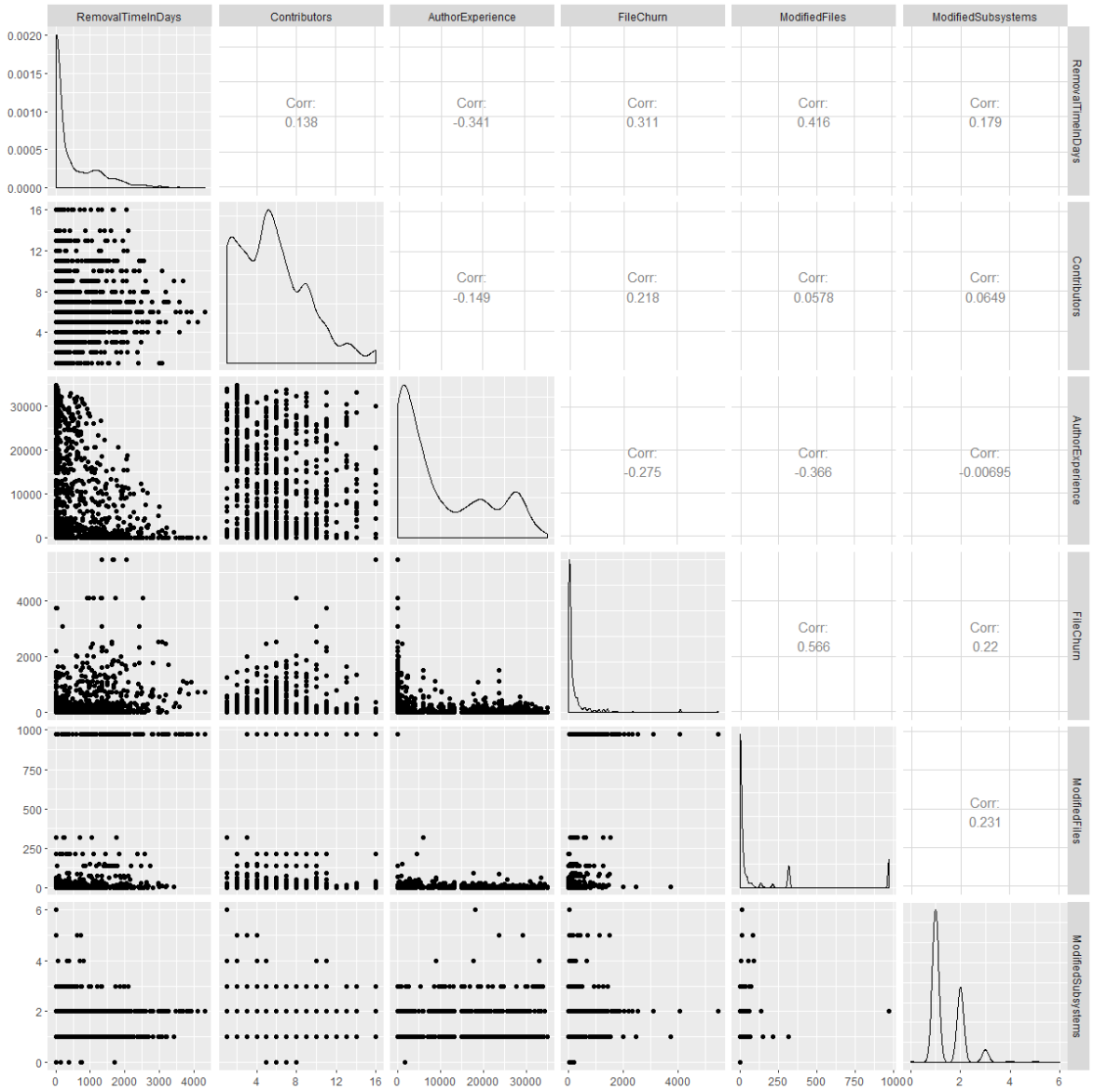

Figure B.50: Tomcat - Removal time (days).

Model: EffortInWords ∼ AuthorExperience + log(1+FileChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems). **R-squared:** 0.04606

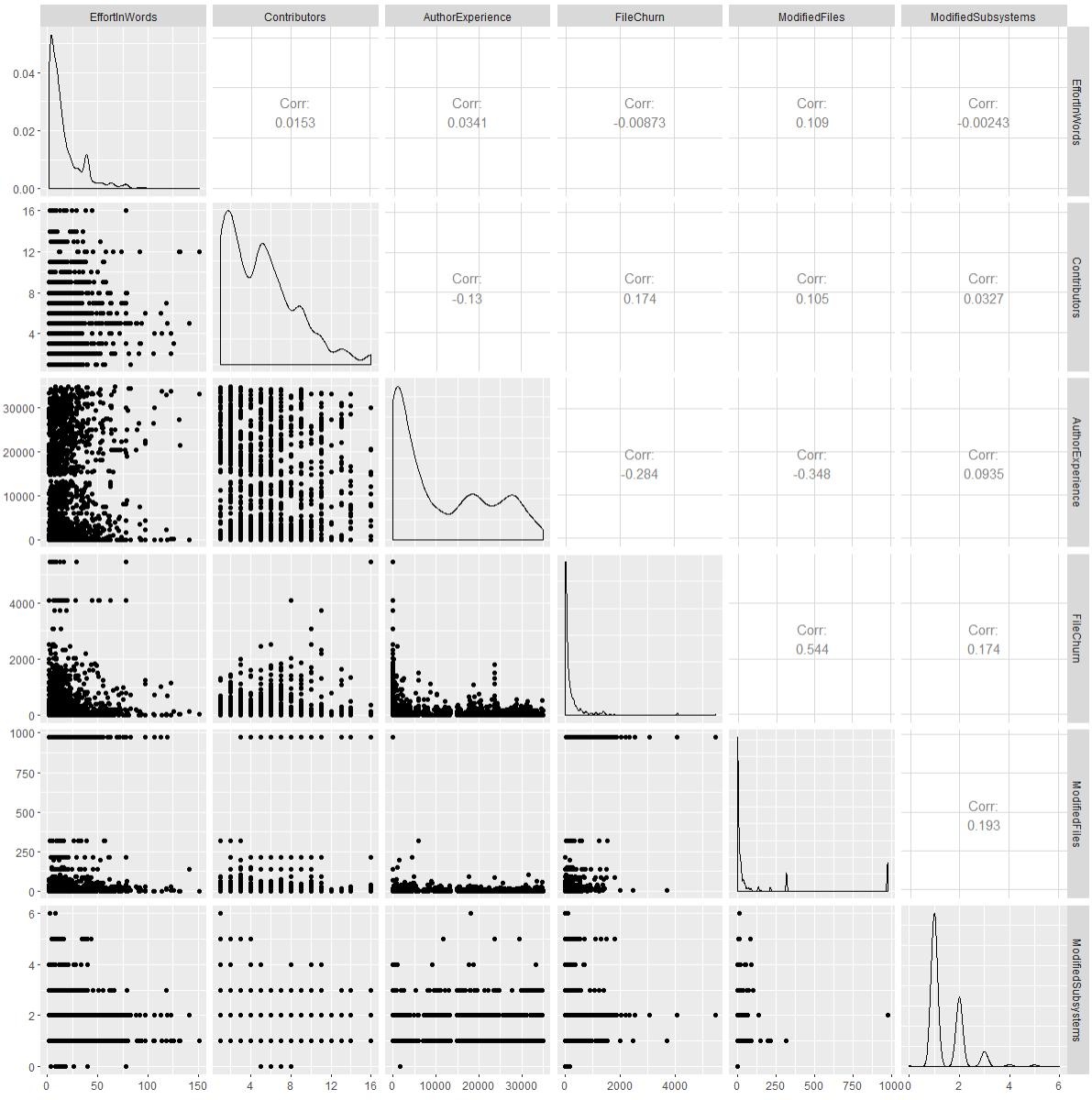

Figure B.51: Tomcat - Effort in Words (days).

Model: InterestFanIn ∼ log(1+AuthorExperience) + log(1+FileChurn) + ModifiedFiles + log(1+ModifiedSubsystems). **R-squared:** 0.002857

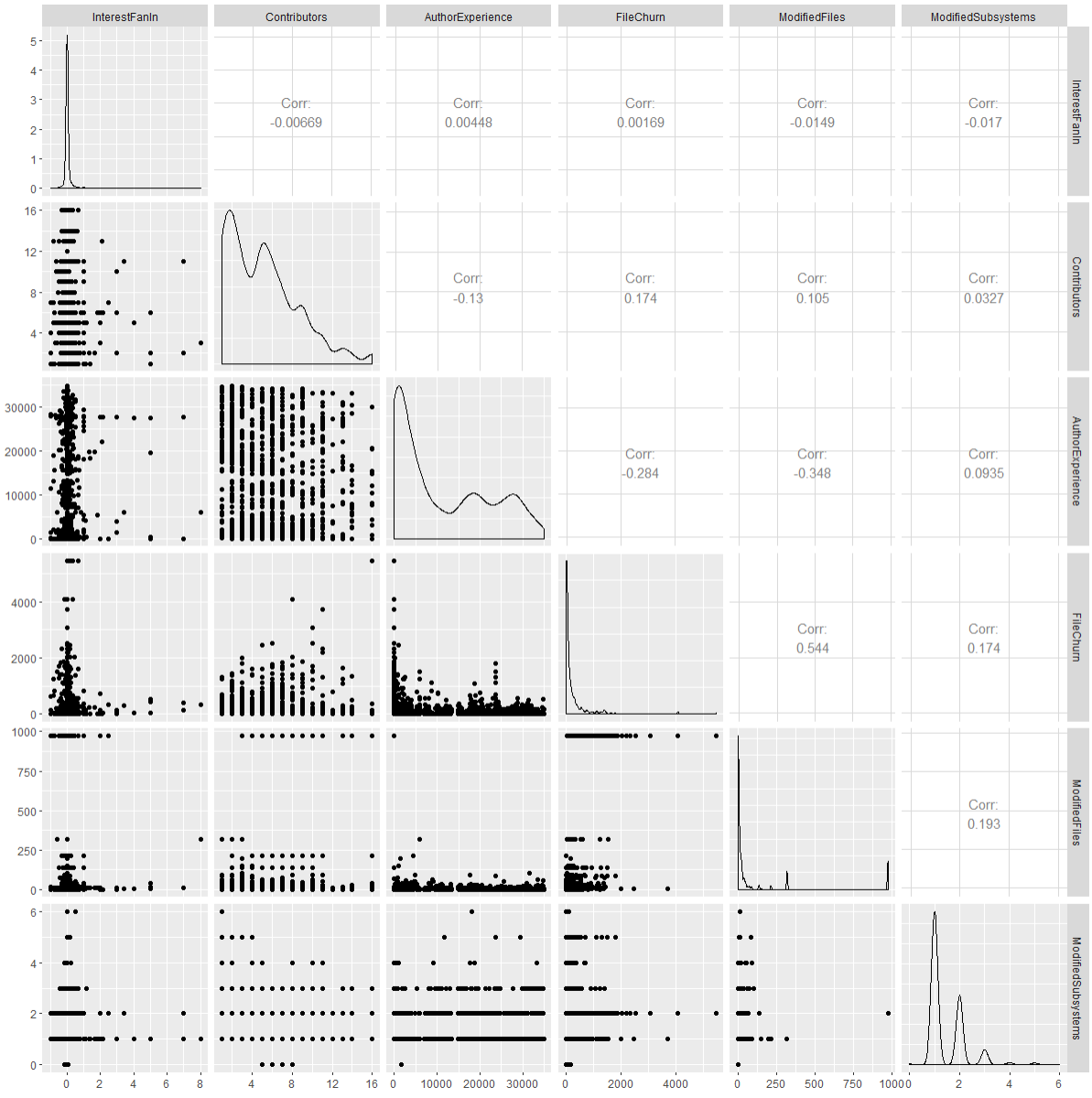

Figure B.52: Tomcat - Simple Interest - Fan-In

Model: InterestLOC ∼ log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems). **R-squared:** 0.002955

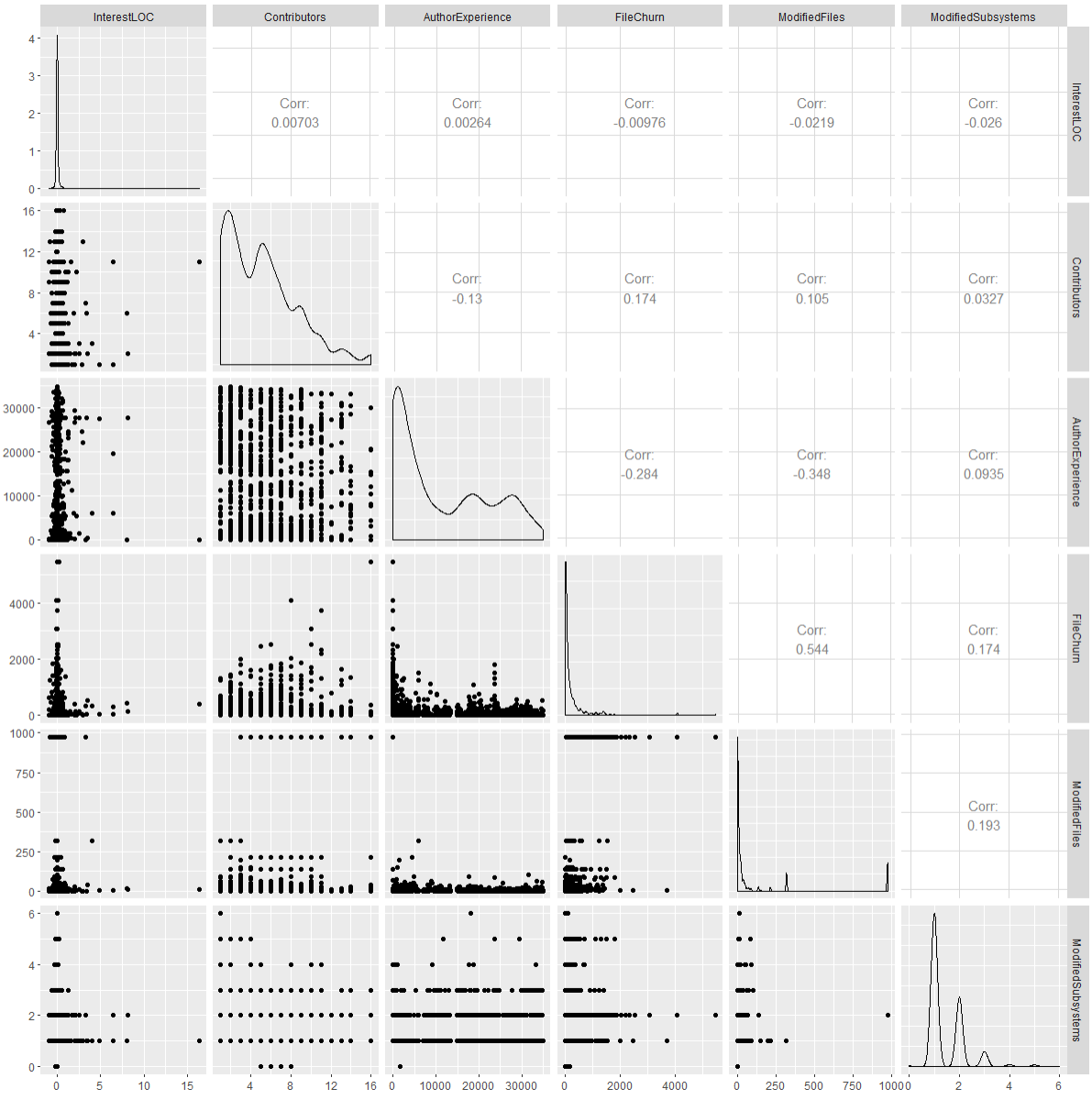

Figure B.53: Tomcat - Simple Interest - LOC

Model: CompoundedFanIn ∼ Contributors + log(1+AuthorExperience) + log(1+FileChurn) + log(1+ModifiedSubsystems). **R-squared:** 0.001773

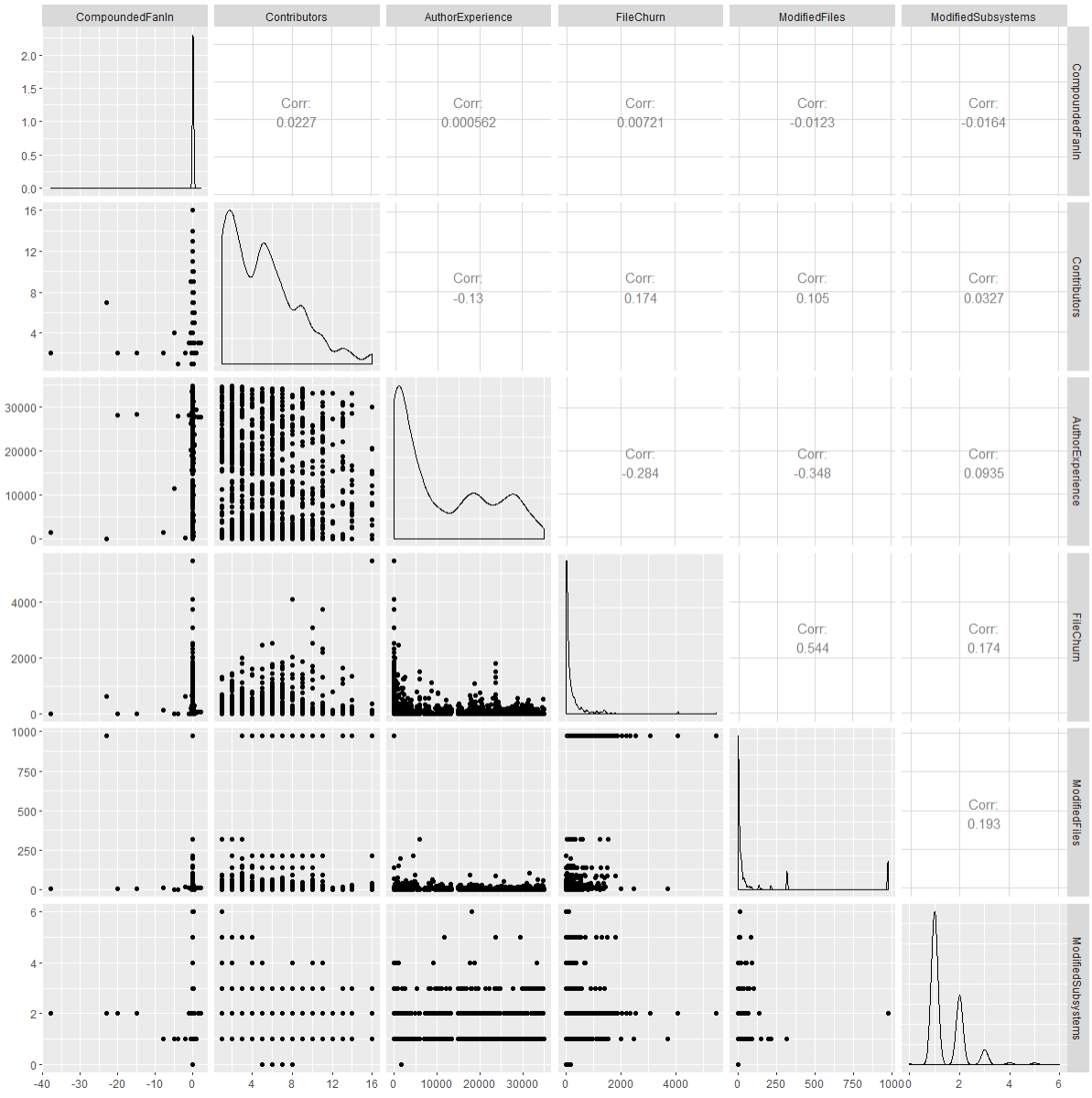

Figure B.54: Tomcat - Compounded Interest - Fan-In

Model: CompoundedLOC ∼ log(1+Contributors) + log(1+AuthorExperience) + log(1+ModifiedFiles) + log(1+ModifiedSubsystems). **R-squared:** 0.001419

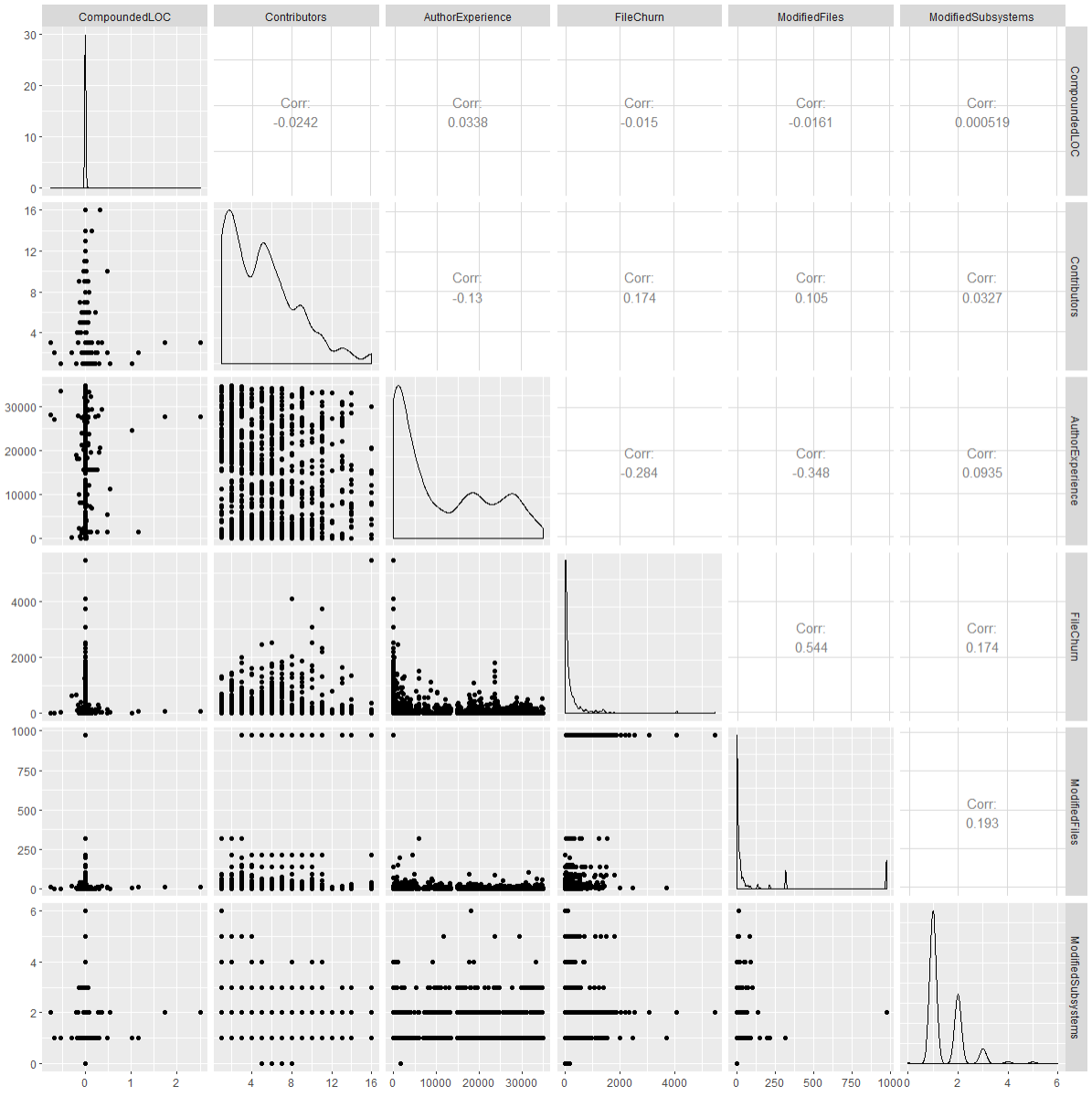

Figure B.55: Tomcat - Compounded Interest - LOC

Model: ChangeProness ∼ Contributors + AuthorExperience + log(1+ModifiedFiles) + ModifiedSubsystems. **R-squared:** 0.6682

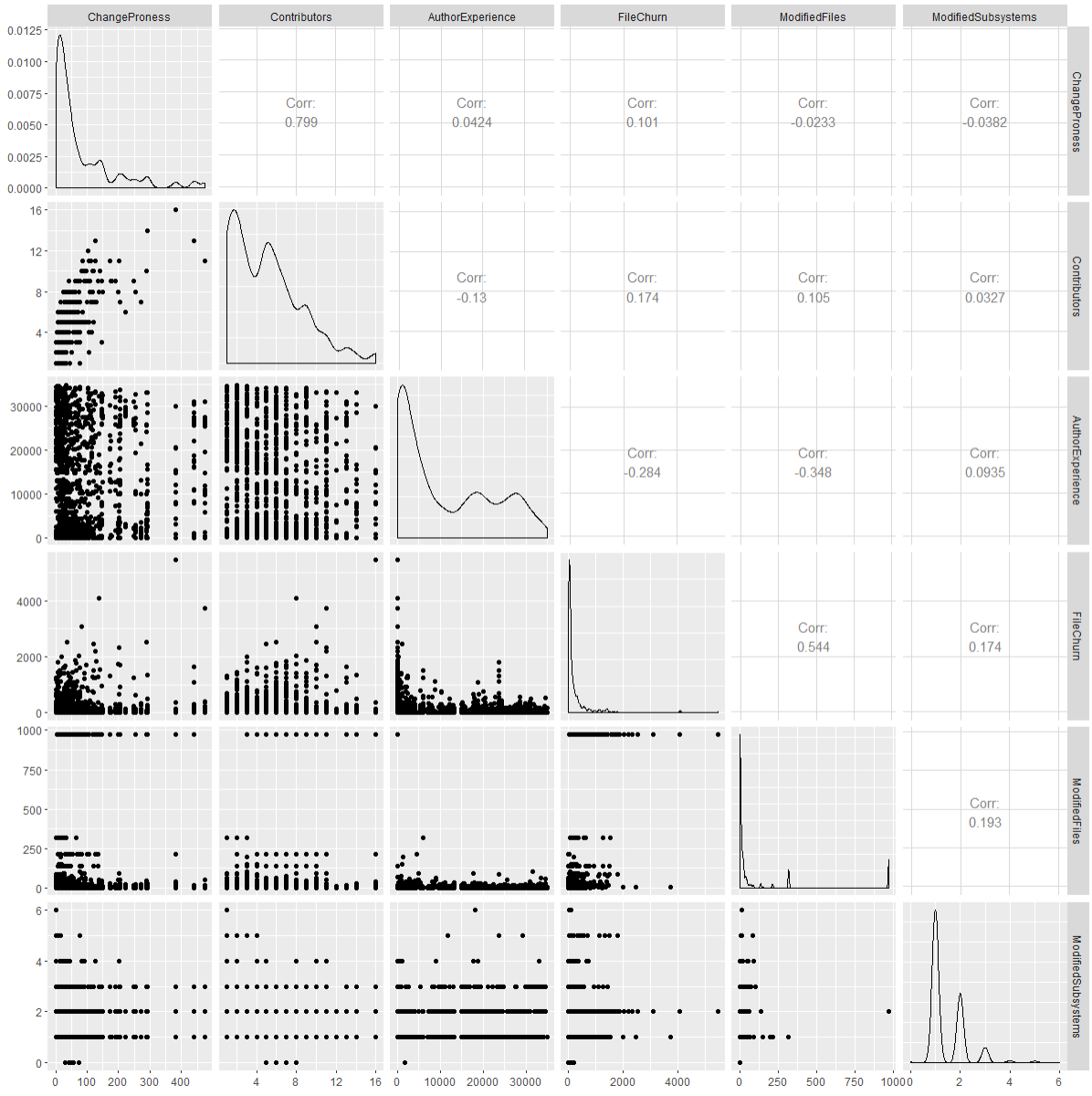

Figure B.56: Tomcat - Change Proneness

References

Akbarinasaji, S., & Bener, A. (2016). Adjusting the balance sheet by appending technical debt. In *Proceedings of the 8th international workshop on managing technical debt* (pp. 36–39).

Alfayez, R., Behnamghader, P., Srisopha, K., & Boehm, B. (2018). An exploratory study on the influence of developers in technical debt. In *Proceedings of the 1st international conference on technical debt* (pp. 1–10). ACM.

Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., & Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, *70*, 100–121.

Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., & Spínola, R. O. (2014). Towards an ontology of terms on technical debt. In *Proceedings of the 6th international workshop on managing technical debt* (pp. 1–7).

Amanatidis, T., Mittas, N., Chatzigeorgiou, A., Ampatzoglou, A., & Angelis, L. (2018). The developer's dilemma: factors affecting the decision to repay code debt. In *Proceedings of the 1st international conference on technical debt* (pp. 62–66).

Bavota, G., & Russo, B. (2016). A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th international conference on mining software repositories* (pp. 315–326).

Bellomo, S., Nord, R. L., Ozkaya, I., & Popeck, M. (2016). Got technical debt? surfacing elusive technical debt in issue trackers. In *Proceedings of the 13th international conference on mining software repositories* (pp. 327–338).

Bhaalerao, A. S. (2017). Determination of hotspots and a study of technical debt in oss projects and

their forks. *ProQuest Dissertations and Theses*, 1–83.

Biegel, B., Beck, F., Lesch, B., & Diehl, S. (2014). Code tagging as a social game. In *Proceedings of the 30th international conference on software maintenance and evolution* (pp. 411–415).

Bieman, J. M., Straw, G., Wang, H., Munger, P. W., & Alexander, R. T. (2003, Sept). Design patterns and change proneness: an examination of five evolving systems. In *Proceedings of the 5th international workshop on enterprise networking and computing in healthcare industry* (p. 40-49).

Collard, M. L., Decker, M. J., & Maletic, J. I. (2011). Lightweight transformation and fact extraction with the srcml toolkit. In *Proceedings of the 11th ieee international working conference on source code analysis and manipulation* (pp. 173–184).

Cunningham, W. (1992). The wycash portfolio management system. *Proceedings on Object-oriented Programming Systems, Languages and Applications*, *4*(2), 29–30.

Dai, K., & Kruchten, P. (2017). Detecting technical debt through issue trackers. In *5th international workshop on quantitative approaches to software quality (quasoq)* (pp. 59–65).

Deterding, S., Dixon, D., Khaled, R., & Nacke, L. (2011). From game design elements to gamefulness: defining gamification. In *Proceedings of the 15th international academic mindtrek conference: Envisioning future media environments* (pp. 9–15).

Dubois, D. J., & Tamburrelli, G. (2013). Understanding gamification mechanisms for software development. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering* (pp. 659–662).

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 10th joint meeting on foundations of software engineering* (pp. 50–60).

Falessi, D., Russo, B., & Mullen, K. (2017). What if i had no smells? In *Proceedings of the 11th acm/ieee international symposium on empirical software engineering and measurement* (pp. 78–84).

Fluri, B., Wursch, M., & Gall, H. C. (2007a). Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th working conference on reverse engineering* (pp. 70–79).

Fluri, B., Wursch, M., & Gall, H. C. (2007b, Oct). Do code and comments co-evolve? on the relation between source code and comment changes. In *14th working conference on reverse engineering (wcre 2007)* (p. 70-79).

Freitas Farias, M. A., de Mendonça Neto, M. G., da Silva, A. B., & Spínola, R. O. (2015a). A contextualized vocabulary model for identifying technical debt on code comments. In *Proceedings of the 7th international workshop on managing technical debt* (pp. 25–32).

Freitas Farias, M. A., de Mendonça Neto, M. G., da Silva, A. B., & Spínola, R. O. (2015b). *Cvm-td vocabulary.* http://homes.dcc.ufba.br/~marioandre/page/cvm-td/vocabulary_comments.pdf. ((Accessed on 01/15/2018))

Freitas Farias, M. A., de Mendonça Neto, M. G., da Silva, A. B., & Spínola, R. O. (2015c). *Excomment tool.* http://goo.gl/9Mgl9m. ((Accessed on 01/15/2018))

Freitas Farias, M. A., de Mendonça Neto, M. G., da Silva, A. B., & Spínola, R. O. (2015d). *Type of technical debt x software engineering nouns.* http://homes.dcc.ufba.br/~marioandre/page/cvm-td/TD_SENouns.pdf. ((Accessed on 01/15/2018))

Freitas Farias, M. A., Santos, J. A., Kalinowski, M., Mendonça, M., & Spínola, R. O. (2016a). *Cvm-td - comments by ratio.* https://drive.google.com/file/d/0BwwEbWFwapG1Y2hRaEt1bGFGa2s/view. ((Accessed on 01/23/2018))

Freitas Farias, M. A., Santos, J. A., Kalinowski, M., Mendonça, M., & Spínola, R. O. (2016b). *Cvm-td - most selected patterns by participants.* https://drive.google.com/file/d/0BwwEbWFwapG1UlNZbG5lekN1UUk/view. ((Accessed on 01/23/2018))

Freitas Farias, M. A., Santos, J. A., Kalinowski, M., Mendonça, M., & Spínola, R. O. (2016c). Investigating the identification of technical debt through code comment analysis. In *Proceedings of the 18th international conference on enterprise information systems* (pp. 284–309).

Fujiwara, K., Hata, H., Makihara, E., Fujihara, Y., Nakayama, N., Iida, H., & Matsumoto, K. (2014). Kataribe: A hosting service of historage repositories. In *Proceedings of the 11th working conference on mining software repositories* (pp. 380–383).

Gauthier, M. (2015, Aug). *Git log's –first-parent option.* http://marcgg.com/blog/2015/08/04/git-first-parent-log/. ((Accessed on 11/28/2018))

German, D. (2017, Mar). *git-mining/tutorial.org at master · dmgerman/git-mining · github.* https://github.com/dmgerman/git-mining/blob/master/tutorial.org. ((Accessed on 11/28/2018))

Ghanbari, H. (2016). Seeking technical debt in critical software development projects: An exploratory field study. In *Proceedings of the 49th hawaii international conference on system sciences* (pp. 5407–5416).

Ghanbari, H. (2017). Investigating the causal mechanisms underlying the customization of software development methods. *Jyväskylä studies in computing*(258), 1–119.

Ghanbari, H., Vartiainen, T., & Siponen, M. (2018). Omission of quality software development practices: A systematic literature review. *ACM Computing Surveys (CSUR)*, *51*(2), 38.

Git. (2018). *git-log documentation.* https://www.git-scm.com/docs/git-log. ((Accessed on 11/28/2018))

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st international conference on software engineering* (pp. 78–88). IEEE Computer Society.

Huang, Q., Shihab, E., Xia, X., Lo, D., & Li, S. (2018). Identifying self-admitted technical debt in open source projects using text-mining. *Empirical Software Engineering*, *23*(1), 418–451.

Ichinose, T., Uemura, K., Tanaka, D., Hata, H., Iida, H., & Matsumoto, K. (2016). Rocat on kataribe: Code visualization for communities. In *Proceedings of the 4th intl. cong. on applied computing and information technology/3rd intl. conf. on computational science/intelligence and applied informatics/1st intl. conf. on big data, cloud computing, data science & engineering* (pp. 158–163).

Kamei, Y., Maldonado, E. d. S., Shihab, E., & Ubayashi, N. (2016). Using analytics to quantify interest of self-admitted technical debt. In *Proceedings of the 1st international workshop on technical debt analytics* (pp. 68–71). CEUR-WS.

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, *39*(6), 757–773.

Khomh, F., Penta, M. D., & Guéhéneuc, Y. (2009, Oct). An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th working conference*

on reverse engineering (p. 75-84).

Kouters, E., Vasilescu, B., Serebrenik, A., & van den Brand, M. G. J. (2012). Who's who in gnome: Using lsa to merge software repository identities. In *Proceedings of the 28th ieee international conference on software maintenance* (p. 592-595).

Leßenich, O., Siegmund, J., Apel, S., Kästner, C., & Hunsen, C. (2018). Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering*, *25*(2), 279–313.

Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (Vol. 10, pp. 707–710).

Lewin, C. G. (1970). An early book on compound interest: Richard witt's arithmeticall questions. *Journal of the Institute of Actuaries*, *96*(1), 121–132.

Lewin, C. G. (1981). Compound interest in the seventeenth century. *Journal of the Institute of Actuaries*, *108*(3), 423–442.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193–220.

Lim, E., Taksande, N., & Seaman, C. (2012). A balancing act: what software practitioners have to say about technical debt. *IEEE software*, *29*(6), 22–27.

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., & Li, S. (2018). Satd detector: A text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th international conference on software engineering: Companion proceedings* (pp. 9–12). ACM.

Maldonado, E., Abdalkareem, R., Shihab, E., & Serebrenik, A. (2017a). An empirical study on the removal of self-admitted technical debt. In *Proceedings of the 33rd international conference on software maintenance and evolution* (pp. 238–248).

Maldonado, E., Abdalkareem, R., Shihab, E., & Serebrenik, A. (2017b). *Icsme study and survey data.* http://das.encs.concordia.ca/uploads/2017/07/maldonado_icsme2017.zip. ((Accessed on 01/30/2018))

Maldonado, E., & Shihab, E. (2015, Oct.). Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th international workshop on managing technical debt* (pp. 9–15).

Maldonado, E., Shihab, E., & Tsantalis, N. (2017a). *Replication package for using natural language processing to automatically detect self-admitted technical debt.* https://github.com/maldonado/tse.satd.data. ((Accessed on 01/16/2018))

Maldonado, E., Shihab, E., & Tsantalis, N. (2017b, Nov). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, *43*(11), 1044-1062.

Matsumoto, S., Kamei, Y., Monden, A., Matsumoto, K.-i., & Nakamura, M. (2010). An analysis of developer metrics for fault prediction. In *Proceedings of the 6th international conference on predictive models in software engineering.* ACM.

Mensah, S., Keung, J., Bosu, M. F., & Bennin, K. E. (2016). Rework effort estimation of self-admitted technical debt. In *Proceedings of the 1st international workshop on technical debt analytics* (pp. 72–75). CEUR-WS.

Mensah, S., Keung, J., Svajlenko, J., Bennin, K. E., & Mi, Q. (2018). On the value of a prioritization scheme for resolving self-admitted technical debt. *Journal of Systems and Software*, *135*, 37–54.

Miyake, Y., Amasaki, S., Aman, H., & Yokogawa, T. (2017). A replicated study on relationship between code quality and method comments. In *Applied computing and information technology* (pp. 17–30). Springer.

Mockus, A., & Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, *5*(2), 169–180.

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

Munson, J. C., & Elbaum, S. G. (1998). Code churn: A measure for estimating the impact of code change. In *Proceedings of the 1998 international conference on software maintenance* (pp. 24–31).

Myers, L., & Sirois, M. J. (2004). Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences*, *12*.

Nagappan, N., & Ball, T. (2005). Use of relative code churn measures to predict system defect

density. In *Proceedings of the 27th international conference on software engineering* (pp. 284–292).

Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th international conference on software engineering* (pp. 452–461). ACM.

Orcale. (2017). *Parsing an xml file using sax (the java tutorials - java api for xml processing (jaxp) - simple api for xml).* https://docs.oracle.com/javase/tutorial/jaxp/sax/parsing.html. ((Accessed on 03/16/2017))

Ortu, M., Destefanis, G., Adams, B., Murgia, A., Marchesi, M., & Tonelli, R. (2015). The jira repository dataset: Understanding social aspects of software development. In *Proceedings of the 11th international conference on predictive models and data analytics in software engineering* (pp. 1:1–1:4). New York, NY, USA: ACM.

Ortu, M., Murgia, A., Destefanis, G., Tourani, P., Tonelli, R., Marchesi, M., & Adams, B. (2016). The emotional side of software developers in jira. In *Proceedings of the 13th international conference on mining software repositories* (pp. 480–483). ACM.

Palomba, F., Zaidman, A., Oliveto, R., & De Lucia, A. (2017). An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th international conference on program comprehension* (pp. 176–185).

Panichella, S., Di Sorbo, A., Guzman, E., Visaggio, C. A., Canfora, G., & Gall, H. C. (2015). How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the 31st international conference on software maintenance and evolution* (pp. 281–290).

Potdar, A., & Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *Proceedings of the 30th international conference on software maintenance and evolution* (pp. 91–100). IEEE.

Potdar, A., & Shihab, E. (2015a). *Satd mtd data.* http://users.encs.concordia.ca/~eshihab/data/MTD2015/MTD_15_data.zip. ((Accessed on 01/12/2018))

Potdar, A., & Shihab, E. (2015b). *Satd patterns.* http://users.encs.concordia.ca/~eshihab/data/ICSME2014/satd.html. ((Accessed on 01/12/2018))

Rosen, C., Grawi, B., & Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (pp. 966–969). ACM.

Rosen, C., Grawi, B., & Shihab, E. (2018). *Commit guru.* http://commit.guru/. ((Accessed on 12/13/2018))

SciTools. (2018a). *Understand — metrics list.* https://scitools.com/support/metrics_list/. ((Accessed on 12/08/2018))

SciTools. (2018b). *Understand — scitools.com.* https://scitools.com/features/. ((Accessed on 12/08/2018))

Shihab, E., Jiang, Z. M., Ibrahim, W. M., Adams, B., & Hassan, A. E. (2010). Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of the 2010 acm-ieee international symposium on empirical software engineering and measurement* (pp. 4:1–4:10). ACM.

Shihab, E., Kamei, Y., Adams, B., & Hassan, A. E. (2013). Is lines of code a good measure of effort in effort-aware models? *Information and Software Technology*, *55*(11), 1981–1993.

Siegmund, J. (2016). Program comprehension: Past, present, and future. In *Proceedings of the 23rd international conference on software analysis, evolution, and reengineering* (Vol. 5, pp. 13–20).

Silva, M. C. O., Valente, M. T., & Terra, R. (2016). Does technical debt lead to the rejection of pull requests? *Brazilian Symposium on Information Systems*, 1–7.

Singer, J., Sim, S. E., & Lethbridge, T. C. (2008). Software engineering data collection for field studies. In *Guide to advanced empirical software engineering* (pp. 9–34). Springer.

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2nd international workshop on mining software repositories* (Vol. 30, pp. 1–5).

Sridhara, G. (2016). Automatically detecting the up-to-date status of todo comments in java programs. In *Proceedings of the 9th india software engineering conference* (pp. 16–25).

Steneker, M. (2016). *Towards an empirical validation of the tiobe quality indicator* (Unpublished doctoral dissertation). Eindhoven University of Technology.

Stijlaart, M., & Zaytsev, V. (2017). Towards a taxonomy of grammar smells. In *Proceedings of the*

10th acm sigplan international conference on software language engineering (pp. 43–54).

Storey, M. A., Ryall, J., Bull, R. I., Myers, D., & Singer, J. (2008). Todo or to bug. In *Proceedings of the 30th international conference on software engineering* (pp. 251–260).

Tan, L., Yuan, D., Krishna, G., & Zhou, Y. (2007). /* icomment: Bugs or bad comments?*. In *Proceedings of the 21st symposium on operating systems principles* (Vol. 41, pp. 145–158).

The R Foundation. (2018). *R: The r project for statistical computing.* https://www.r-project.org/. ((Accessed on 12/22/2018))

Tool, S. (2018). *srcml.* http://www.srcml.org/. ((Accessed on 11/28/2018))

Tsantalis, N., Chaikalis, T., & Chatzigeorgiou, A. (2008). Jdeodorant: Identification and removal of type-checking bad smells. In *Proceedings of the 12th european conference on software maintenance and reengineering* (pp. 329–331).

Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Penta, M. D., & Zaidman, A. (2016, Oct). Continuous delivery practices in a large financial organization. In *Proceedings of the 32nd international conference on software maintenance and evolution* (pp. 519–528).

Verdecchia, R., Malavolta, I., & Lago, P. (2018). Architectural technical debt identification: The research landscape. In *Proceddings of the 1st international conference on technical debt* (pp. 11–20). ACM.

Wehaibi, S., Shihab, E., & Guerrouj, L. (2016). Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the 23rd international conference on software analysis, evolution, and reengineering* (Vol. 1, pp. 179–188).

Wiese, I. S., d. Silva, J. T., Steinmacher, I., Treude, C., & Gerosa, M. A. (2017). Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *Proceedings of the 32nd international conference on software maintenance and evolution* (p. 345-355).

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (pp. 38:1–38:10).

Yan, M., Xia, X., Shihab, E., Lo, D., Yin, J., & Yang, X. (2018). Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering*, 1-1.

Yujian, L., & Bo, L. (2007, June). A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *29*(6), 1091-1095.

Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., & Di Penta, M. (2017). Recommending when design technical debt should be self-admitted. In *Proceedings of the 33rd international conference on software maintenance and evolution* (pp. 216–226).

Zampetti, F., Ponzanelli, L., Bavota, G., Mocci, A., Penta, M. D., & Lanza, M. (2017). How developers document pull requests with external references. In *Proceedings of the 25th international conference on program comprehension* (pp. 23–33).

Zampetti, F., Serebrenik, A., & Di Penta, M. (2018). Was self-admitted technical debt removal a real removal? an in-depth perspective. In *Proceedings of the 15th international conference on mining software repositories* (p. 11 pages).

Ziegler, T. (2017). *Gitcop: A machine learning based approach to predicting merge conflicts from repository metadata* (Unpublished doctoral dissertation). University of Passau.