Mining Developer Dynamics for Agent-Based Simulation of Software Evolution

Dissertation
zur Erlangung des mathematisch-naturwissenschaftlichen Doktorgrades
"Doctor rerum naturalium"
der Georg-August-Universität Göttingen

im Promotionsprogramm Computer Science (PCS)
der Georg-August University School of Science (GAUSS)

vorgelegt von

Verena Herbold
aus Hildesheim

Göttingen, 2019

Abstract

The steady growth of software in our daily life results in the need for quicker adaption of the software changing usage and requirements. This process is defined as software evolution. Primarily, it is concerned with changes that are responsible for the evolution. The most important contribution to this process results from developers, e.g., by adding code to the repository. This process is highly dynamic as the team constellation as well as the activity of individual developers is always changing. This is especially the case for open-source software (OSS) projects which are analyzed in this thesis because of the free availability.

We create and evaluate several models describing software evolution. The main focus of the approach described in this thesis is in the source of the changes, i.e., the developers. Using Agent-based simulation, project managers have the ability to try different scenarios and estimate possible software evolution trends. For example, it is possible to choose a team constellation and evaluate if the chosen team will be able to fix enough bugs using the simulation output. If not, more developers can be added to the simulation. In this case, the developers are agents who create, update, and delete software artifacts and possibly add or fix bugs at the same time. Huge parts of this thesis are dedicated to find suitable simulation parameters and estimate them by mining software repositories to gain a realistic simulation. Questions like the size of the software project, the activity of developers, the number of bugs, and the structure of the software under simulation can be answered. We apply methods from data mining, machine learning and statistics for our work.

For the simulation, the behavior of developers is estimated using heuristics gained from analyzing the history of different software projects. The resulting simulation model reflects different developer roles with varying workload. Although, the representation of OSS dynamics was limited. For a fine-grained developer contribution behavior, a state-based probabilistic model (Hidden Markov Model) was trained based on different levels of code-based and communication-based activities. This allows the developers to switch between different of activity. The same procedure is used to summarize the whole project activity with the aim to evaluate whether a project is still active. Therefore, we are interested in finding out how much activity is still performed in inactive projects, since a strict separation is difficult to find, but important for potential users of the project.

The results of three case studies show that Agent-based simulation is a promising approach for the prediction of software evolution and that many relations can be described this way. In particular, it turned out that a dynamic developer and project behavior is indispensable for the description of OSS evolution, because otherwise the representation of software processes is too static.

Zusammenfassung

Durch die wachsende Zunahme von Software in unserem alltäglichen Leben, nimmt auch die Anpassung der Software an die Nutzung und somit verbundenen ständig wechselnden Anforderungen zu. Dieser Prozess wird als Softwareevolution bezeichnet. In erster Linie geht es um Änderungen an der Software, die für die Evolution verantwortlich sind. Eine wichtige Rolle dabei spielen die Entwickler, da diese die Änderungen vornehmen, indem sie z.B. Code zum Repository hinzufügen. Dem Prozess liegen viele Dynamiken zugrunde, da sich sowohl das Entwicklerteam als auch die Aktivität der einzelnen Entwickler stets ändert. Dies ist vor allem in Open-Source Softwareprojekten der Fall, die hier aufgrund der Verfügbarkeit der Daten analysiert werden.

Mehrere Modelle zur Beschreibung von Softwareevolution werden erarbeitet und ausgewertet. Der in der Doktorarbeit verfolgte Ansatz beginnt dort, wo die Änderungen entstehen: bei den Entwicklern. Ein Agenten-basiertes Simulationsmodell ermöglicht es dem Softwareprojektmanager verschiedene Szenarien auszuprobieren und so mögliche Verläufe abzuschätzen. Zum Beispiel kann eine Teamzusammenstellung gewählt werden, um simulativ zu ermitteln, ob diese in der Lage sein wird, genügend Fehler zu beheben. Falls nicht, könnten weitere Entwickler zur Planung hinzugefügt werden. Agenten sind in diesem Fall die Entwickler, die die Softwareartefakte erstellen, ändern, löschen und dabei ggf. Fehler hinzufügen oder korrigieren. Ein großer Teil der Arbeit beschäftigt sich damit, geeignete Simulationsparameter zu finden und durch Mining von Softwarerepositorien zu schätzen, um eine möglichst realitätsnahe Simulation zu ermöglichen. Fragestellungen nach der Größe des Projekts, der Aktivität von Entwicklern, der Fehleranzahl und der Struktur der Software können dabei beantwortet werden. Hierzu werden Methoden aus den Bereichen Data Mining, Machine Learning und Statistik verwendet.

Für das Simulationsmodell wurde das Verhalten der Entwickler durch Heuristiken aus den Historien verschiedener Softwareprojekte gemittelt. Dieses Modell stellt bereits verschiedene Entwicklertypen mit unterschiedlicher Arbeitsintensität zur Verfügung. Allerdings konnte nur limitiert eine Dynamik widergespiegelt werden. Für ein verfeinertes Entwicklerverhalten wurde ein statistisches Modell (Hidden Markov Modell) basierend auf mehreren Ebenen Code-basierter und kommunikativer Aktivität trainiert, welches den Entwicklern erlaubt zwischen verschiedenen Aktivitätsleveln zu wechseln. Das gleiche Vorgehen wird genutzt, um Projektaktivität zusammenzufassen und zu bewerten, ob das Projekt noch aktiv ist. Das Hauptinteresse hierbei ist es herauszufinden wieviel Aktivität ein inaktives Projekt noch haben kann, denn eine klare Trennung ist schwierig, aber unabdingbar für potentielle Nutzer des Projekts.

Die Resultate von drei Fallstudien haben gezeigt, dass Agenten-basierte Simulation ein vielversprechender Ansatz zur Vorhersage von Softwareevolution ist und dass viele Zusammenhänge damit dargestellt werden können. Insbesondere hat sich gezeigt, dass ein dynamisches Entwickler- und Projektverhalten unabdingbar für die Beschreibung von Softwareevolution sind, da sonst Projektverläufe zu statisch abgebildet werden.

Acknowledgements

During the work on this thesis, several people supported me in many ways. First, I would like to thank my supervisor Prof. Dr. Jens Grabowski who was always available for fruitful discussions and provided helpful suggestions. He and his research group created a pleasant and constructive working environment which helped me to find my own way in research.

I would also like to thank my second supervisor Prof. Dr. Stephan Waack for his continuous support during my studies. Discussions with him often made me look at things from a different angle.

Additionally, I would like to thank all my current and former colleagues who were always willing to discuss with me and give me valuable and honest feedback. Especially, I thank Dr. Patrick Harms, Dr. Fabian Trautsch, and Alexander Trautsch for proof reading this thesis. In addition I thank Gunnar Krull for his technical support. Also, I would like to thank my collaborators from the SWZ project. By working on the project, I had the opportunity to set my work into context and to learn much about project work, e.g., managing responsibilities.

I am in the fortunate position to have two special colleagues I would like to thank. The first is my brother Daniel, with whom I share two important interests in my life, computer science and rock music. Without him, I would certainly be a different person today.

The second special colleague is my best friend and husband Steffen. He helped me to grow on a professional as well as on a personal level. In addition, he was always there to help me organize my thoughts. He motivated me all along and accepted all my moods for which I am very thankful.

Finally, I want to thank my son Jona. He taught me that not anything has to happen exactly the way that I expected and, even more surprisingly for me, that this is not fatal. He and my husband give me a lot of strength and I do not want to miss them in my life.

Contents

List of Acronyms

ABMS Agent-based Modeling and Simulation

ABS Agent-based Simulation

AP Activity Plot

AST Abstract Syntax Tree

BDI Belief Desire Intent

CMMI Capability Maturity Model Integration

CRF Conditional Random Field

DES Discrete Event Simulation

DEVCON Developer Contribution

GIS Geographic Information System

GQM Goal Question Metric

HMM Hidden Markov Model

ITS Issue Tracking System

KNN K-Nearest Neighbor

LOC Lines of Code

ML Mailing List

MSR Mining Software Repositories

OSS Open Source Software

PAC Probably Approximately Correct

PDE Partial Differential Equations

PSP Personal Software Process

RCS Revision Control System

SCSS Source Code Control System

SD System Dynamics

SG Sequence Group

SNA Social Network Analysis

SP States Plot

STEPS Software Trend Evolution Prediction in Simulation

UML Unified Modeling Language

UMM Unified Markov Models

VCS Version Control System

1. Introduction

Nowadays, software is everywhere, and it continuously adapts to meet corresponding usage scenarios. During the software development process, project managers and developers have to cope with changing requirements and environments. *Software evolution* deals with the adaption of the software system to exactly these changes. As such, it has become a strong research field belonging to the area of software engineering over the last decades. It is concerned with understanding the past of software projects, e.g., by analyzing logs, and tries to monitor the present in order to avoid future issues. In addition, gained knowledge can also be applied to predict the future.

A central point in the investigation of software evolution are software changes. They give information about who did what to a system and make the software evolve. The amount, impact, and intent of changes can vary a lot during the software lifecycle. Especially, with the rise of open source software projects this process can not be put into a straight scheme. A lot of dynamics, not only caused by a shifting developer base, have to be taken into account. The software development process strongly depends on the participating developers as well as on their behavior, i.e., personal work style, motivation, experience, and background. The involvement of different kinds of developers in a software project has a significant impact on the outcome, e.g., in the amount of lines of code written or in terms of quality, e.g., technical debt introduced [1]. In this thesis, an approach is presented that takes human factors, i.e., the behavior of developers, into account and utilizes this for building predictive models for software evolution. The work establishes different software evolution models that can be used for Agent-based simulation to forecast the future of software projects. Thereby, we intent to support project managers in making decisions and monitoring software quality.

1.1. Motivation

Software project managers have to deal with limited resources for software quality assurance. Therefore, methods and tools that aid them in their planning and decision making are beneficial. Keeping track on the diverse factors that affect software quality can be a strenuous task. Such factors include structural changes in the software, pressure of time, design decisions, the constellation of the software team, the introduction of bugs, the distribution of tasks, or an increase of the complexity of the software. To support software project managers in their decisions, a bunch of methods and tools exist, e.g., to estimate the risk of the project [2] or to predict maintenance and failure-prone releases [3]. Often, these tools are

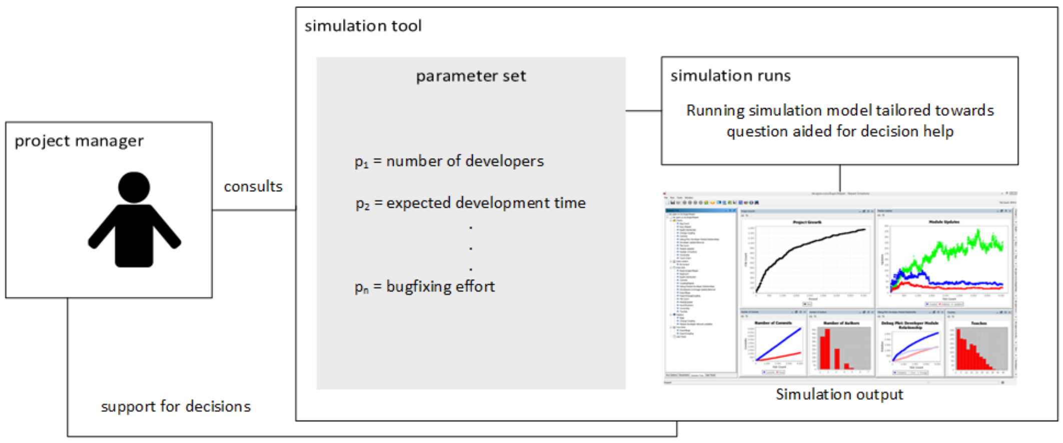

simulation tool
parameter set
p₁ = number of developers
p₂ = expected development time
pₙ = bugfixing effort
simulation runs
Running simulation model tailored towards question aided for decision help
Simulation output
project manager
consults
support for decisions

Figure 1.1.: Feedback loop for project managers [4].

tailored towards a specific problem. A broad picture of the whole software development process is hard to capture, because there are many factors involved that may promote or even contradict each other.

The overarching goal of our work is to establish the feedback loop for project managers illustrated in Figure 1.1. With the help of a simulation tool (large box), the project manager can forecast different evolutionary scenarios. These scenarios depend on a set of parameters (gray box) that reflect the current state of the project, e.g., the number of developers involved, the expected timespan, or the effort spent on fixing bugs. Based on that, running the simulation produces an interpretable simulation output that can help the project manager in making decisions. For example, if the predicted bug distribution is to high the bug fixing effort should be increased. Our tool is targeted on offering scenario-based predictions where the user can select the metrics she is interested in, such that both small scenarios as well as general trends of the whole project can be forecasted.

To establish such a simulation tool, it is fundamental to understand the underlying software evolution processes as well as their interplay. Generally, software evolution patterns can be derived by mining data about existing software development processes [5, 6, 7]. Using data mining as well as machine learning and statistical learning, the observations can be described suitable for a simulation model that is tailored towards a specific research question. The considered question determines model entities and attributes, e.g., for a model that is aimed to describe collaboration software developers and some information on past co-working developers has to part of it.

Research has shown that developers play a centric role in software evolution [8, 9]. The whole development process strongly depends on the individuals, their background, personality, training, and accomplishment of tasks, and, as such, on human behavior. Hence, an elaborate description of the behavior of developers is fundamental. For this purpose, we model developers from an Agent-based perspective, where individual agents are the drivers of the simulation.

The novelty in this approach lies in the combination from different disciplines that closely work together. Our work enfolds methods from data mining, machine learning, statistical learning, and Agent-based modeling and simulation.

1.2. Scope

In this thesis, we propose to use Agent-based Modeling and Simulation (ABMS) for describing software evolution scenarios which can be used as a decision help for project managers. We lay a special focus on the behavior of developers since this is a central part and driver in software evolution. The main underlying assumptions behind this are that we can model and simulate software evolution using agents and that project managers can benefit from that. We assume, that it is possible to find common patterns in software evolution which are valid for groups of projects or project entities. To evaluate this, we investigate the following superordinate research question regarding the application of Agent-based simulation for modeling software evolution:

- *RQ* 1: Can we model software evolution using Agent-Based simulation?

To answer this RQ in a whole, we split the problem into several subquestions, which we answer in this thesis:

- *RQ* 1.1: What are important parameters for simulating software evolution?
- *RQ* 1.2: How can these parameters be estimated?
- *RQ* 1.3: Which software evolution phenomena and trend can be simulated?

Since we decide to model software evolution from a developers' perspective, we especially focus on the way developers behave and contribute in a software project. For this, the overall research question is the following:

- *RQ* 2: How can we model developer contribution behavior?

The investigation of this topics is split into several subquestions:

- *RQ* 2.1: Is a state based probabilistic model appropriate for modeling developers' contribution behavior?
- *RQ* 2.2: Are the retrieved models similar for the same kinds of developers?
- *RQ* 2.3: Can we apply general contribution models in software engineering practice?
- *RQ* 2.4: How does the level of detail of a developers' contribution behavior model influence simulation results?
- *RQ* 2.5: Can a state based probabilistic models also be used for modeling project activity?

1.3. Goals and Contributions

The work conducted to answer the RQs stated above yields the following contributions:

- The identification and estimation of simulation parameters suitable for simulating software development processes by mining software repositories.
- The identification and description of different software evolution patterns.
- An Agent-based simulation model which is designed to answer different questions concerning software evolution. The model provides a feedback loop for project managers as a help for decisions.
- A Hidden Markov Model for the description of developers' contribution behavior combining code-based activity with communication. The model can be used both for building individual developer contribution models and for applying a general model for prediction.
- A Hidden Markov Model for the summarization of software project activity. The approach can be used to judge the level of activity as well to detect critical trends.
- Three case studies for the evaluation of the described approaches including a software evolution simulation model, (the simulation of) developer contribution behavior, and a characterization of project activity.

1.4. Impact

During the work and the above topics, the following papers have been published in peer reviewed conference proceedings:

- Verena Honsel [1], Steffen Herbold, Jens Grabowski, "Learning from Software Project Histories: Predictive Studies Based on Mining Software Repositories", in *Machine Learning and Knowledge Discovery in Databases: European Conference (ECML PKDD 2016)*, Proceedings, Part III, 2016
  Own contributions
  I came up with the idea to present and summarize all predictive studies employing Machine Learning techniques for software engineering worked out in the research group. For this, I summarized own work in the dedicated chapters.
- Verena Honsel[1], Steffen Herbold, Jens Grabowski, "Hidden Markov Models for the Prediction of Developer Involvement Dynamics and Workload", in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2016)*, 2016
  Own contributions
  I am the lead author of this publication. I performed most of the work including the

[1]maiden name

design of the approach as well as the implementation, analysis, and evaluation of the conducted case studies. The classification with Machine Learning Models was joined work with Dr. S. Herbold.

- Verena Honsel[1], Daniel Honsel, Steffen Herbold, Jens Grabowski, Stephan Waack, "Mining Software Dependency Networks for Agent-Based Simulation of Software Evolution", in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), The 4th International Workshop on Software Mining*, 2015
  Own contributions
  I am the lead author of the paper. I contributed significantly to the design of the approach, the mining process and the evaluation of the approach. The required simulation framework was provided by D. Honsel.
- Verena Honsel[1], Daniel Honsel, Jens Grabowski, Stephan Waack, "Developer Oriented and Quality Assurance Based Simulation of Software Processes", in *Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE 2015)*, 2015
  Own contributions
  This paper presents a summary of the papers [10], [4], and [11]. As such, it is joined work of all involved authors. Own contributions include the conceptual work as well as the summary of the proposed papers.
- Verena Honsel[1], "Statistical Learning and Software Mining for Agent Based Simulation of Software Evolution", in *Proceedings of the 37th International Conference on Software Engineering - Volume 2, Doctoral Symposium at the 37th International Conference on Software Engineering (ICSE 2015)*, Florence, Italy, 2015
  Own contributions
  This publication is a doctoral symposium paper where the idea and first results of the doctoral project were presented. I am the single author of this paper and established all of the work on my own.
- Verena Honsel[1], Daniel Honsel, Jens Grabowski, "Software Process Simulation based on Mining Software Repositories", in *Proceedings of the IEEE International Conference on Data Mining Workshop (ICDM 2014)*, short paper, 2014
  Own contributions
  As the lead author of this paper, I contributed to the design and evaluation of the approach. I was responsible for the mining process and analysis of mined data. The implementation of the simulation model was done by D. Honsel. Furthermore, the comparison of simulation and empirical data was joined work with D. Honsel.

Furthermore, some papers were published to which the author of this thesis contributed:

- Marlon Welter, Daniel Honsel, Verena Herbold, Andre Staedler, Jens Grabowski, Stephan Waack, "Assessing Simulated Software Graphs using Conditional Random

Fields“, in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, Springer, 2018
Own contributions
Own contributions for this paper include the conceptual work for the preparation of required software graphs. These graphs were embedded into the simulation tool by D. Honsel and than assessed by a tool developed by M. Welter. M. Welter also analyzed the impact of the tool.

- Daniel Honsel, Niklas Fiekas, Verena Herbold, Marlon Welter, Tobias Ahlbrecht, Stephan Waack, Jürgen Dix, Jens Grabowski, “Simulating Software Refactorings based on Graph Transformations“, in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, Springer, 2018
Own contributions
This paper presents a way to reflect refactoring in a simulation of software evolutions based on graph transformations. The design of the approach as well as the implementation of the simulation is done by D. Honsel. I contributed to the initial simulation model, which is adapted by the lead author for software refactorings.
- Tobias Ahlbrecht, Jürgen Dix, Niklas Fiekas, Jens Grabowski, Verena Herbold, Daniel Honsel, Stephan Waack, Marlon Welter, “Agent-based simulation for software development processes“, on *Proceedings of the 14th European Conference on Multi-Agent Systems (EUMAS 2016)*, Springer, 2016
Own contributions
I was involved in the design of the proposed approach as well as in the parameter ming for the simulation model. The distributed simulation framework is provided by T. Ahlbrecht and N. Fiekas. The modeling and implementation of the non distributed version was achieved by D. Honsel.
- Daniel Honsel, Verena Honsel[1], Marlon Welter, Jens Grabowski, Stephan Waack, “Monitoring Software Quality by Means of Simulation Methods“, in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2016)*, short paper, 2016
Own contributions
For this paper, I contributed to the conceptual work of the presented approach as well as to the case study design and evaluation. Needed simulation parameters were mined from real software projects by me. The behavior and strategies of agents were designed and implemented by D. Honsel. The automated assessment of software graphs was done by M. Welter.

In addition, the following book chapters have been published containing parts of the work established in this thesis:

- Philip Makedonski, Verena Herbold, Steffen Herbold, Daniel Honsel, Jens Grabowski, Stephan Waack, “Mining Big Data for Analyzing and Simulating Col-

laboration Factors Influencing Software Development Decisions“, in *Social Network Analysis: Interdisciplinary Approaches and Case Studies*, CRC Press, 2016
Own contributions
Own contributions of this book chapter contain the mining of software projects to build developer social networks as well as the analysis of these. This work establish an example application for the fine-grained developer behavior and collaboration model presented in this book chapter by Dr. P. Makedonski. A simulation of achieved networks was provided by D. Honsel. Besides, the incorporation of collaborative factors into a software defect prediction model was conducted by Dr. S. Herbold.

- Steffen Herbold, Fabian Trautsch, Patrick Harms, Verena Herbold, Jens Grabowski, “Experiences With Replicable Experiments and Replication Kits for Software Engineering Research“, 2019
  Own contributions
  I made a replication kit available for her work [12] which is taken as an example in the mentioned book chapter. The experience report is completely designed and written by the other authors.

1.5. Structure of the Thesis

This thesis is structured as follows. First, we pin the foundations of our work in Chapter 2. This chapter provides the background needed for understanding the work as a whole and is divided into several sections: Section 2.1 explains what software evolution is and how our approach fits to it. Then, we introduce mining software repositories (Section 2.2) which enables the investigation of evolutionary patterns as well as the estimation of simulation parameters. The basics about Agent-based simulation are presented in Section 2.3. Since one main question of this thesis is how to model developer behavior, we describe the corresponding existing definitions and approaches in Section 2.4. Then, we give an overview on Hidden Markov Models which represent an important instrument for our approach (Section 2.5). To complete the foundations, we present AB/BA crossover studies which are used for evaluation in this thesis (Section 2.6).

In Chapter 3, we list related work and show similarities as well as divergences in the course of the research. In doing so, we subdivide the related work according to different topics: In Section 3.1, similar work in the context of mining for software evolution is presented followed by research established in the area of describing the (contribution) behavior of developers in Section 3.2. Then, we report related studies performed for software process simulations in Section 3.3. We conclude this chapter by presenting related work in the context of open source project activity in Section3.4.

Our approach is two-folded and, thus, presented in tow different chapters. The first part of the approach deals with the definition of suitable simulation models for software evolution and is described in Chapter 4. In there, first the general process of the creation of simulation models tailored towards specific questions is illustrated (Section 4.1). We continue with the STEPS simulation model (Section 4.2) where all developers spend the same effort aside from their type and explain all entities needed for it as well as questions that can be answered using the model. Then, we define the DEVCON simulation model in Section 4.3 having the ability of developers to model dynamic developer behavior, e.g., to have phases of low, medium, or high activity. Afterwards, it is explained how software quality is assessed within the proposed model (Section 4.4) followed by strengths and limitations of the simulation models (Section 4.5).

The second pillar of the approach implies the instantiation of the defined models. This requires to determine all information needed for feeding the model. The process of instantiation is described in Chapter 5. Section 5.1 explains how needed parameters can be estimated and sets requirements for the subsequent sections. Then, the data collection and preparation process is illustrated (Section 5.2). The actual instantiation method for the STEPS model is given in Section 5.3 whereas needed methods for the instantiation of the DEVCON model are presented in Section 5.4.

Chapter 6 presents the conducted case studies. First, the overall design and objectives are introduced (Section 6.1). Then, we present three different case studies: A study on general simulation of software evolution (Section 6.2), a case study about dynamic developer contribution behavior (Section 6.3), and a case study dedicated to open source project activity (Section 6.4).

In Chapter 7, we answer our research questions from the introduction of this thesis (Section 7.1). Moreover, we amplify strengths and limitations of the work (Section 7.2) and state important threats to validity (Section 7.3).

Finally, we conclude our work in Chapter 8. In doing so, we summarize the main findings of the thesis (Section 8.1) and give an outlook an potential future work (Section 8.2).

2. Foundations

This chapter presents the foundations of this thesis which range over the areas of software evolution, mining software repositories, and agent-based simulation. We describe the basic concepts and how these work together. Since this thesis has a special focus on describing developer behavior, we finally give an overview on that topic.

2.1. Software Evolution

The term software evolution dates back to the 1970s where the first large software systems were build. At that time, Manny Lehman formulated the first version of his prominent laws of software evolution, where the aim was to understand the changes to the system. His findings, based on an IBM operating system, were confirmed later with other projects (e.g., [13],[14]). For this, he introduced the term *E-type systems* for systems that solve real-world problems with vague requirements and a continuing need of change.

The original waterfall life-cycle model of software processes proposed by Winston Royce [15] included the stages requirements, design, implementation, verification, and maintenance. In this model, maintenance presented the last stage after the delivery of the system, where only bug fixes and smaller changes took place. This view of software processes is outdated nowadays. Software engineers realized that this view in enclosed steps is too simple. The whole requirements are very rarely all clear in the beginning of the project. And also experiences, e.g., gained by the implementation, may give new insights on the design. Thus, communication between the different stages had to be enabled.

In general, evolution describes “continuous change from a lower, simpler, or worse to a higher, more complex, or better state“ [16]. This can be transfered to software thinking of a program which fulfills the initial requirements in the first version, but then the need for an additional feature emerges by the users. Thus, according to changing requirements, the program has to be adapted, and hence evolves to a more complex state.

Several definitions of the term software evolution exist. Lehman et al. (e.g, [17]) describes it as the “the consequence of an intrinsic need for continuing maintenance and further development of software embedded in real world domains“ . One big concern in studying software evolution is the understanding of the *what* and *why* of occurring software evolution phenomena, i.e., finding out causes and impact factors. Related to that, in other work [18], Lehman et al. determine the investigation of software evolution as to include “the complementary concerns relating to the achievement of evolution, i.e., the how, and the nature of the evolution phenomenon, i.e., what it is and why it occurs" . Understanding evolutionary software processes is an indispensable task for establishing good predictions and analyses in software evolution research.

Driven by the significant work of Lehman, software evolution became a popular research topic accepted as part of software engineering in the 1990s. Software evolution research is aimed to observe the past, control the present, and predict the future. Several studies, e.g., examine the history of open source software (OSS) projects with the aim to observe patterns or draw heuristics that can help understanding software changes. Most work is concerned with understanding involved people (e.g., developers, testers, users), artifacts (e.g., classes, files) and bugs in the software development process.

Software Maintenance

Often the terms *software evolution* and *software maintenance* get mixed up. According to Royce, software maintenance begins after the delivery of the first version of the system. That does not mean that the development process is independent from maintenance. Software maintenance planning should take place early in the development process. Decisions concerning maintenance often benefit from an understanding of software evolution processes. Thus, software maintenance can be viewed as a part of software evolution as the whole process from initial phase to maintenance. There exist four types of software maintenance: perfective (enhancements, improvements), adaptive (environment and hardware adaption), corrective (debugging), and preventive (prevention of future bugs and maintainability risks).

2.2. Mining Software Repositories

Mining Software Repositories (MSR) became a broad and popular research topic over the last years. It mainly deals with the analysis of different available data sources with information on the software systems under investigation. Often, it deals with analyzing the past to foreshadow the future. Because of nowadays large OSS communities, a wealth of data about software development is freely available and ready to analyze. Not only in the analysis part itself, but also in the facilitation of mining effort, e.g., linking data between multiple repositories, a lot of work was done, e.g., in [19] [20]. Repositories of interest include all systems where information about the developers and their project work is stored. These are the Version Control Systems (VCS), Issue Tracking Systems (ITS), Mailing Lists (ML), user forums, IRC communication, and Twitter. Nowadays, the data is mature enough to weight the main work of studies on the analysis and interpretation part for a lot of tasks [21]. In the context of software evolution the analysis aims to gain a better understanding of software changes, their causes, and their impact [5]. Popular topics in this context include among

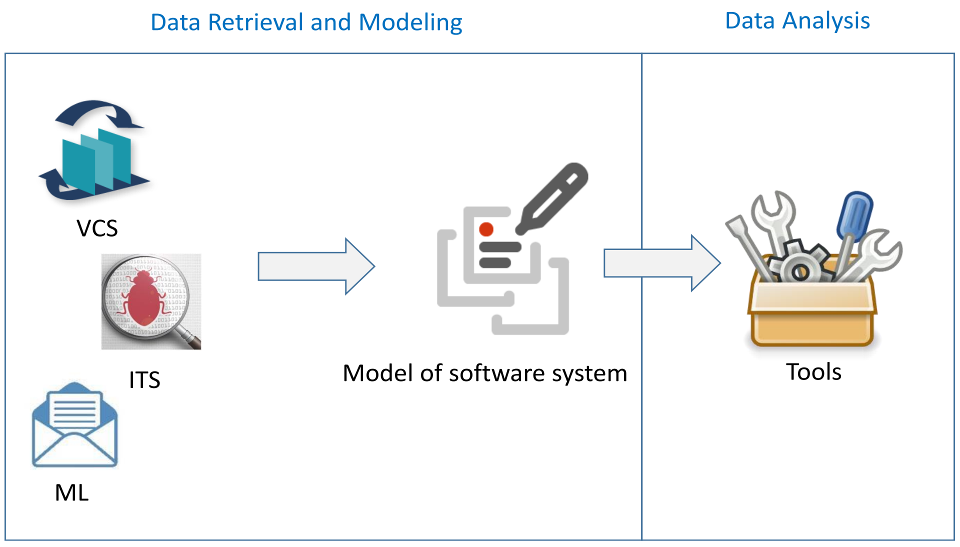


Figure 2.1.: Mining Software Repositories (adopted from D'Ambros et al. [5]).

others the effort spent by developers, change impact and origin analysis, and the prediction of bugs.

The general, underlying process is illustrated in Figure 2.1. The overall procedure is divided into two main steps: data retrieval and modeling, and data analysis. First, we start with a model of the software evolution task we want to investigate. For this, important aspects need to be identified and their interrelations have to be specified. A common problem is the linkage between the different model entities, e.g., between classes and bugs. Also, one has to be careful when determining the data needed for the analysis because otherwise this could impede the analysis effort. The second step is to build a concrete instance of the model based on the definition of the first step. Considering for example a model of project growth, than a concrete instance can be a real software project. Thus, all aspects and data defined in step one needs to be extracted and preprocessed. Data retrieval is concerned with collecting the desired history from the different data sources like VCS, ITS, and ML. Moreover, parsing and data mining techniques take place in this step. As Figure 2.1 shows, the second step is the data analysis. There, tools and methods to gain insights on the mining task and answer the posed questions are applied. The choice of the tools depends on the task, e.g., machine learning for prediction, visualization for exploring, and simulation for forecasting.

Different mining approaches exist, tailored towards the purpose and context of the mining task. These combinable approaches include MSR via VCS annotations, data mining, heuristics, and differencing [22]. With the annotation available in the version tracking of the system, basic questions like which files changed together can be answered. Besides, the comments in the VCS or ITS can be of interest, e.g., for identifying relations to points of interests in the file history or hot spots. Data Mining techniques aim to reveal patterns and make predictions about the data. Heuristics extend the mining via annotations. Here, basic derivable knowledge is already within the analysis, e.g., semantic or syntactic mappings. Finally, differencing is a technique used to analyze code-based differences between different versions of the software. For this purpose, often abstract syntax trees (ASTs) are used which gives information about added software entities like classes and methods and relations among them.

Since we want to simulate different possible outcomes of a software project, it is reasonable to rely the simulation model on real data. Therefore, the utilization of MSR to retrieve patterns and trends that describe software evolution phenomena and trends is beneficial. These can then be transferred into the simulation model.

Following, we describe data mining techniques, the data preparation process, involved data sources, and metrics often used in MSR research, that are also relevant for this thesis.

2.2.1. Software Metrics

Software metrics play an important role in software evolution research. According to the IEEE, a software (quality) metric is defined as “a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality“. Hence, with metrics the current state of software projects can be measured. There are three categories of metrics: (1) *process metrics* measuring aimed at the effectiveness of the development process itself, e.g., defects found, (2) *project metrics* that evaluate attributes concerning characteristics of the software project, e.g., costs, and (3) *product metrics* describing the delivered software product, e.g., in terms of portability.

Depending on the repository, different metrics can be calculated for the specified purpose and research question, respectively. Some metrics are easy to extract like the number of files or classes. For others, the measurement is more complex, e.g., for relations among the data such as files that are changed together or measures of the importance of developers or artifacts, e.g., by network measures.

The choice of relevant metrics is not always an easy task. Various approaches for the selection exist where the most famous one is the *Goal Question Metric (GQM)* approach introduced by Basili et al [23]. There, based on defined goals, metrics concerning software quality, the software process, or the software product are used to answer specific questions which arise directly from the goals. The process of metrics selection should be thought over carefully to get meaningful results. In software evolution research, metrics play an important role, since they are able to describe and measure both, the current state of the system and the development over the time. Used metrics in software evolution include information about software entities such as size measures (e.g., lines of code), complexity, and object-oriented measures, but also the number of developers who touched the entity as well as the number of bugs attached to it. Bug-related metrics cover information about the status of the bug (open, re-opened, closed), or the number of related comments. For measuring characteristics of developers, common metrics are the number of commits, bug comments, or mailing list posts as well as network measures such as the centrality in contribution networks.

2.2.2. Data Sources

The data under investigation in MSR research is stored in repositories, which are databases storing all past edits and the whole history of changes to the software system. A commit usually includes the author, the committer, the commit date, the files changed and a commit message describing the kind of changes. The author is not automatically the committer of a set of changes, e.g., if a patch written by the author is applied by a contributor (committer). The VCSs coordinate the work of different developers contributing to the same project. They store the code and keep track of all changes to software pieces including documentation. Hence, they enable developers to browse all versions of the source code. VCSs allow parallel development in form of branching, i.e., different parallel streams of code. The advantage of using branches is the possibility to change and test some parts of the software and still having a stable version in the trunk. When merging brnaches back to the trunk, the VCS supports developers to solve conflicts, i.e., if two developers performed changes on the same line of a file.

The wealth of information available in nowadays VCSs was not there from the beginning of version control. It started with *Source Code Control System (SCCS)* which only kept track of single files [24] and which was introduced in 1972. Although not practicable for large software projects, they already had the idea of using *deltas* for measuring changes, which include all changes to the file, for the differences between two versions. The comparison of these, functioning on comparing the distinct lines of the file, i.e., whether something was added, removed, or modified, is integrated in nowadays VCSs. SCCS was followed by RCS (Revision Control System) which facilitated file storage, retrieval, and merging. Still, the system lacks in sharing the code base with developers working on geographically distributed locations. Then, CVS (Concurrent Versioning System) came up which was the first to make use of the client/server model. As such, it manages the software system on a central server whereas involved developers have their working copy on their own client.

The client/server model is still common in currently used versioning systems. Generally, VCSs can be categorized into centralized version control and distributed version control. In the centralized case, all files are stored on a central server and each client can checkout the files from there. Popular systems functioning this way are, e.g., Subversion (SVN), CVS, and the Microsoft Team Foundation Server. There, every developer can commit the work done to the central repository as well as checkout changes made by others. If a conflict occurs, it has to be decided which version should be kept and which should be discarded. Non-conflicting pieces of work can be merged straightforward. It is also possible to revert your working copy to a previous version.

In contrast, distributed version control systems possess more than one repository. This circumvents the risk of data loss, e.g., when the central server is down. In this scenario, every contributor is provided with a working copy as well as an own repository. You commit and update to the local repository and for sharing, you pull the work of others to your repository and push the status of your repository to the central repository. For conflict management, merge tools exist, but you can also solve conflicts using the command line.

From a researchers point of view, the rise of decentralized systems has several advantages as well as disadvantages that we exemplify in the following. The data offered by the different forms of repositories varies a lot among them [25]. In decentralized repositories, more data is available since more information about the past is stored, e.g., branches. Because of the presence of more than one repository in the decentralized case, many (unintended) branches exist whereas in SVN and other centralized repositories branching is rare and thought-out, e.g, for releases. Thus, it may lack information about the "real" behavior of developers. On the other hand, with more information at hand, one has to be more careful in analyzing and interpreting. Mining decentralized repositories has the advantage that all metadata is local [25], i.e., the whole file history is available for investigation without further effort. In any case, it is important to understand the structure of the repository, in order to derive meaningful findings.

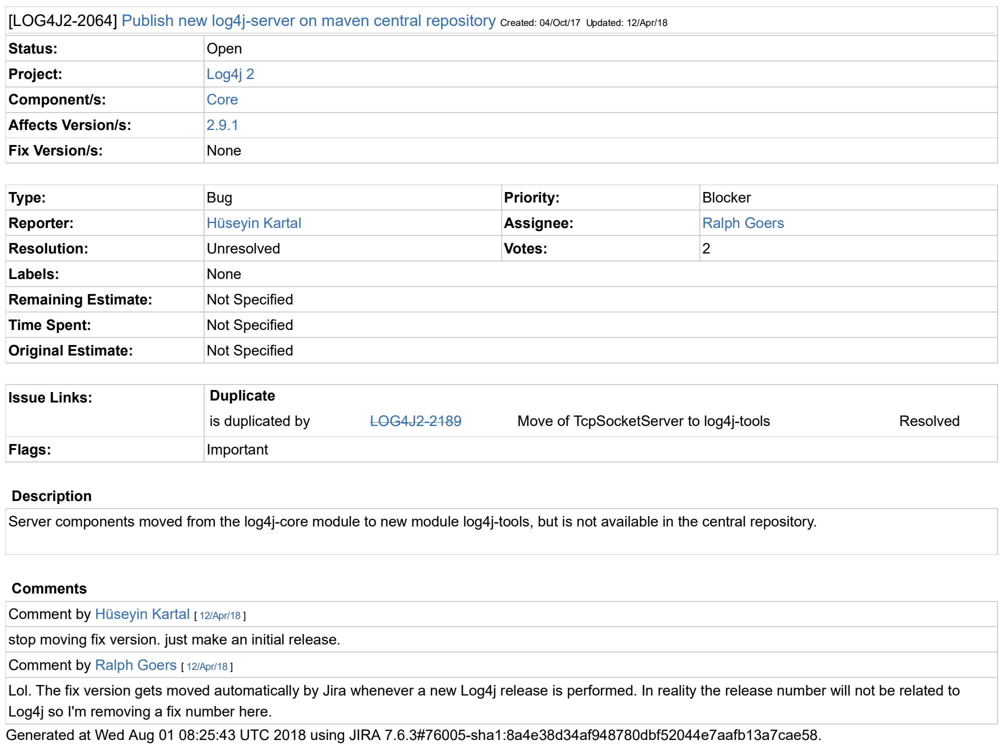
Another important data source for researchers active in software evolution is the issue tracking system. There, information about all issues related to a software project are stored and managed in a database. Issues are not necessarily bugs, they can also be a request for improvement or addition of functionality. Users of the system can report issues including the following information: issue id, assigned developer/maintainer, severity, priority, status, date of creation, description, reporter, and the affected software component. When a new issue is created in the ITS, usually its status is *unconfirmed* until it gets confirmed, and, thus receives the status *new*. If a developer takes over the responsibility for the issue, it is assigned to the developer. When the maintenance work on the issue is done, it switches to the status *resolved* which can be confirmed by quality assurance such that the issue is *closed*, otherwise it may be *reopened* for further improvement. Possible severities express the significance of the issue, e.g., minor, major, critical, or enhancement. In addition, the ITS tracks all comments and discussions on issues. Examples for popular ITSs are Bugzilla [2], JIRA [3], and Redmine [4].

A typical example of an issue is depicted in Figure 2.2. Here, in the heading the issue id is stated ([LOG4J2-2064]) as well as a short description of the issue. Moreover, information on, e.g., the status, component, assignment, and the importance is given followed by related comments.

[2]http://www.bugzilla.org/

[3]https://www.atlassian.com/software/jira

[4]https://www.redmine.org/



[LOG4J2-2064] Publish new log4j-server on maven central repository Created: 04/Oct/17 Updated: 12/Apr/18

Status:	Open
Project:	Log4j 2
Component/s:	Core
Affects Version/s:	2.9.1
Fix Version/s:	None

Type:	Bug	**Priority:**	Blocker
Reporter:	Hüseyin Kartal	**Assignee:**	Ralph Goers
Resolution:	Unresolved	**Votes:**	2
Labels:	None		
Remaining Estimate:	Not Specified		
Time Spent:	Not Specified		
Original Estimate:	Not Specified		

Issue Links:	**Duplicate**			
	is duplicated by	LOG4J2-2189	Move of TcpSocketServer to log4j-tools	Resolved
Flags:	Important			

Description

Server components moved from the log4j-core module to new module log4j-tools, but is not available in the central repository.

Comments

Comment by Hüseyin Kartal [12/Apr/18]

stop moving fix version. just make an initial release.

Comment by Ralph Goers [12/Apr/18]

Lol. The fix version gets moved automatically by Jira whenever a new Log4j release is performed. In reality the release number will not be related to Log4j so I'm removing a fix number here.

Generated at Wed Aug 01 08:25:43 UTC 2018 using JIRA 7.6.3#76005-sha1:8a4e38d34af948780dbf52044e7aafb13a7cae58.

Figure 2.2.: JIRA example from the project log4j.

Finally, the third big data source for the extraction of software mining data are mailing lists. They involve the communication between developers, and sometimes also users. Together with the VCS and ITS, MLs constitute an extensive set of information on the history of a software project.

The linkage between the VCS and the ITS is often established using traceability links [26]. This means that the commit messages in the VCS are searched to locate issue ids of the ITS. This allows to determine commits representing specific developer activities, e.g., bug fixes. Also, keywords are used to detect bug fixing commits. For this, the famous approach of Sliwerski et al. [27] is commonly used among researchers. For linking source code with the ITS as well as the ML, the identities of involved developers are matched for which several algorithms exist [28].

One more thing to consider is that the mining of software artifacts also differs depending on the type. Naturally, repositories include a mixture of both structured and unstructured data. Structured data include call graphs, meta data, and logs whereas unstructured data capture all artifacts utilizing natural language such as bug reports, source code, comments, mailing list entries, and requirements documents.

Apart from extracting data directly from the repository, some freely available research datasets exist. They contain datasets that are mostly tailored towards a specific research direction, e.g., defect prediction, effort estimation, and code analysis [5].

In the last years, software engineering research underwent a big gain in empirical studies to support software development. Such approaches rely on the quality and the processing of an often huge amount of repository data. To facilitate the mining, some frameworks arose using cloud-based infrastructure which allows for better scaling and powerful computations. An example is [29].

Still, the different platforms support different types of analytics. A framework which is developed within the Institute of Computer Science at the University of Göttingen and makes in-depth analysis of software projects possible, is SmartSHARK [30]. It is a framework which is designed to facilitate the software mining process and it was developed in parallel to the main work on this thesis.

The design of SmartSHARK tackles different problems concerning repository mining with the overall aim to ensure replicable and validated studies. To achieve this, the data is stored in a cloud available for all users. The analysis part is a two step process: Researchers have to select the targeted project data, which is automatically loaded into a MongoDB. Then, researchers write an analysis program in Java or Python that is being submitted via an Apache Spark job. Apache Spark is a distributed computing framework.

SmartSHARK allows to combine different repositories from which data can be extracted: VCS, ITS, and ML data are available for analysis. Though, the merging of different identities occurring for the same person across the repositories has to be done independently if it is desired. The platform already collects a bunch of software metrics like size, complexity, and coupling metrics on different layers, e.g., class-level or function-based, for a subset of the projects available.

Within this work, SmartSHARK is used for one of our case studies.

2.2.3. Data Preparation

As stated above, mining different repositories requires a linkage between them as a first step. But depending on the data and the purpose, more work beforehand may be necessary in order to prepare the data for the actual mining process.

Hemmati et al. [21] pointed out that MSR data can be very noisy due to different reasons. For example, co-changed files may not be semantically related or it may be hard to identify the actual set of changes belonging to a commit. Thus, all underlying assumptions have to be proved and validated. Moreover, empty entries can occur, e.g, empty commit messages. Depending on the research context, one has to decide how to handle this problem. Furthermore, the identification of merge commits as well as a closer look at very large commits may be useful to reflect relevant aspects [31]. Other problems of noisy data may be dupli-

[5]http://openscience.us/repo/

cated entries or missing values. In some cases, it may also be helpful to look for outliers in the data which may later influence the mining process.

2.2.4. Data Mining

Many techniques used for MSR origin from the field of data mining. Data mining can be described as the “extraction of implicit, previously unknown, and potential useful information from data“ [32]. Here, raw data constitute recorded facts, whereas information means all underlying observations that can describe the data, e.g., by patterns. In contrast to the data, the information is not visible at first sight. A lot of approaches exist to uncover valuable information from data stored in databases. Data mining techniques can be of descriptive or predictive nature. Descriptive techniques reveal facts that are already there, but the information has to be filtered or put into a comprehensible format. In predictive data mining, the goal is to use some information that is already there to forecast characteristics of other information, e.g., prediction of future trends. Besides software engineering, data mining has a wealth of applications, e.g., in marketing, health, and bio informatics.

The main goal of data mining is to find patterns that fit the data and as such make the data understandable and interpretable. For finding nontrivial patterns in data, a lot of automated processes exist. Generally, desired patterns have to reveal meaningful characteristics of the data for a better understanding (descriptive) or are able to make powerful predictions about a specific outcome (predictive) [33]. Patterns always capture the underlying structure of the data. Such a pattern of software evolution can for example be a rule like: *If file A and B are changed, often file C is changed as well.* In data mining research, a lot of techniques that origin from the field of machine learning are used.

Generally, a selected element from the input data is referred to as an *instance*. Often, an instance is represented by a set of attributes, which are defined as *features* (e.g., [34]). Features usually are multi-dimensional and, thus, represented as vectors.

The output of machine learning algorithms can be of qualitative nature or of quantitative nature. Quantitative output measures give information about the relationships of the data. In the case of qualitatively measured output, we get a finite set of categorical or discrete values that put the data into categories [35]. Here, the input data is used to predict the output. This type of learning task is called *supervised* learning. This means, that the learning process takes place with knowing the outcome for a set of instances. The input data, called *training set*, is used to create a *predictor*. The predictor is then able to classify so far unseen data. For assessing the goodness of a predictor, it usually gets applied to a set of new, unseen data, the *testing set*. For the testing set, the expected output of the predictor is known and compared with the predictors actual output.

In Figure 2.3, supervised learning is illustrated on the right side. There we have a set of labeled (green,red) instances on the top serving as training data. Once the predictor is learned, all new instances can be labeled as green or red as well. This way, the data is separated according to the labeling. In contrast, unsupervised techniques deal with input

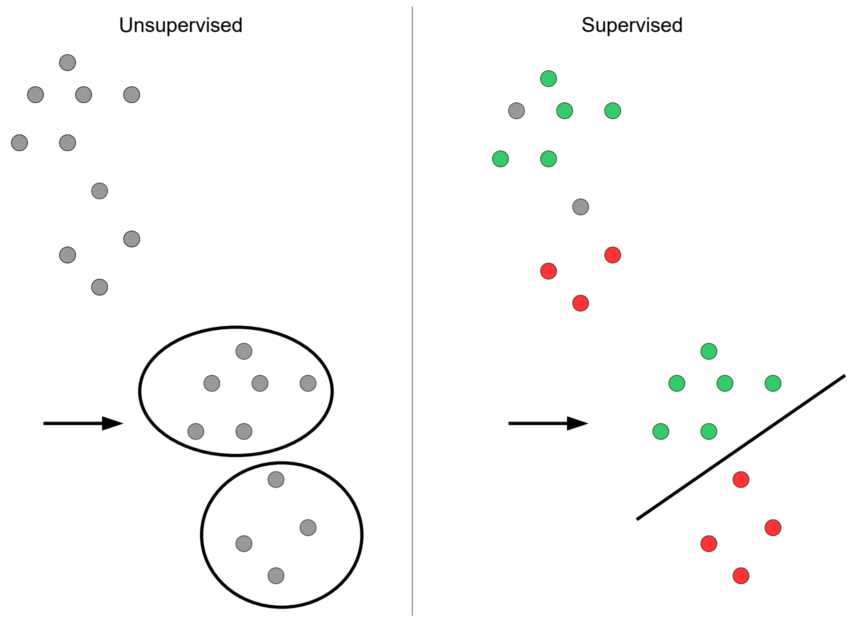


Figure 2.3.: Unsupervised vs. Supervised Learning.

data where no information about the outcome is available. Following the example in Figure 2.3, the unlabeled data points on the top left are the input. There, no predictor can be learned, but structural properties of the data. Thus, the output could look like the grouped instances on the bottom of the picture. The labels in software evolution often indicate if a bug is within a revision which can simply be described with 1 for true and 0 for false. Besides numerical values, categorical values, e.g., for the severity of bugs, are possible. In these cases, supervised learning is possible. Unfortunately, often the data for other mining tasks has no such labels available, e.g., for the roles of software developers. In this scenario, either unsupervised methods are available or the usage of heuristics, the distribution of the data, or expert knowledge to classify a part of the data can be taken into account to get such a label and make the data suitable for other learning tasks.

Learning Techniques

The following techniques fall into the category of supervised learning. Here, two main tasks of prediction exist: *regression* – where the output is measured quantitatively and *classification* – where the output is of qualitative nature. Both can be seen as an approach to approximate some kind of function [35]. We describe the ideas behind the relevant techniques for this thesis in the following.

1. *Linear and Polynomial Regression*: The application of regression methods is the modeling of relationships between input and output variables. For a numeric output and numeric variables polynomial regression can be used to model the combination of the variables. Generally, this relationship can be expressed by $f(x) = a_0 + a_1x + a_2x^2 + ... + a_nx^n$ with $a_0, ..., a_n$ being the variables to model. In the linear case the degree n equals 1. Higher order regression is called polynomial regression according to the type of the resulting curve. Different algorithms exist to find the best fit, e.g., least squares. In software evolution, often curves of trends such as the growth of the system are calculated to describe the course.

2. *k-Nearest Neighbor*: In practice, nearest neighbor methods are used to classify unseen data by available labeled data that is similar. There each unseen instance is compared to other already known instances using a distance measure. In doing so, the class of the closest instance is assigned to the new one. For *k*-Nearest Neighbor the *k* closest entities are considered and a majority vote on the classes determines the class of the unseen instance. Unusually, the Euclidean distance is used for the computation of the distance.

3. *Decision Trees and Random Forests*: Another way to learn a predictor for classification as well as for regression is to use decision trees. They are used to classify instances by conditions based on the feature set. Here, a tree structure is learned where the interior nodes represent the decisions with two or more branches and the leaf nodes imply the outcome, i.e., the predicted class or probability. On the whole, for each combination of input variables, a path to a decision exist. Although a bunch of algorithms exist for learning, most of them adhere to the following main idea (ID3/C4.5) [34]: The algorithm builds the tree from the root to the leaves starting with the determination of the most powerful node, if taken alone, as the root node. Then, for all reasonable values a successive node is created and, again, it is tested which attribute is the best to test in this place. Altogether, the algorithm implements a greedy search without backtracking [34]. One big advantage of decision trees is that they are easy to interpret if they are not too large.
   Closely related, for random forests a set of decision trees is build based on a randomized algorithm. The randomness is generated by searching for the most prominent feature in a randomly generated feature subset. Then, the nodes are divided according to the evaluation on this random subset. As a result, a combination of the most consistent predictions is used. In comparison with basic decision trees, random forests are often more accurate, but decision trees may be more valuable in terms of knowledge representation and interpretation.

4. *Threshold Learning*: In mining software repositories, often the attributes under investigation are measured by software metrics, e.g., the lines of code to some point in time or the number of developers involved. Usually, more than one metric is of interest

and, thus, the common trend of the project is harder to interpret. To deal with such metric sets, it can be of support to classify the metric values into problematic or not based on proper boundaries, i.e., thresholds. To calculate such thresholds, it is common to use a PAC (Probably Approximately Correct) learner [36]. In the algorithm, a d-dimensional axis-aligned rectangle is learned which splits the data according to their label (inside and outside of the rectangle) with d the number of attributes.

For advanced applications, especially when using machine learning, the data has to be prepared carefully. If the approaches work in the given context, strongly depends on the data. We now review different possibilities to transform data into a suitable format. In many machine learning algorithms, the selection of important features is integrated. Nevertheless, often they reach a better performance when applying a selection beforehand [32]. Such techniques include ranking, filtering, and wrapping. Another widespread method is sampling. There, subsets of the data are created based on the distribution of the data. This is especially applicable in larger scale applications. A technique which is relevant for our work, is the transformation of multi-class problems into a two-class problem. For this, it is common to split the data into two-class problems (called one-vs.-rest), but also pairwise classification can aid to solve this problem [32]. In this scenario, a classifier is learned for every couple of classes and a majority vote decides about the final classification.

2.2.5. Software Analytics and Statistical Learning

Software analytics are first and foremost designed to answer questions about the software project to aid management [37]. Besides the quality of the software system and the (productivity of the) development process under investigation, it also includes the users and their satisfaction [6]. Not only a specific answer, but also the processing of information to get most valuable insights constitutes a main pillar in software analytics. Different approaches exist to assist managers in doing so: visualization for a better understanding, analysis algorithms, and large-scale computing [38] for big datasets.

Buse and Zimmermann [37] gave an overview as well as a guideline of performing software analysis studies. They divide the types of software analytics according to the time - if it affects the past, present, or future, as well as the kind of method - exploratory, analytic, or experimental. The types of analyses enfold, e.g., the observation of trends, the usage of detected trends to forecast the future development, modeling, benchmarking, and simulation. In this thesis, software analytics pose the field of application whereas the mining of software repositories functions as a vehicle to extract and process the needed information.

Another method that found usage in software engineering research and can help to describe software evolution is statistical learning. Essentially, there is a thin line between Machine Learning and Statistical Learning because they have some methods in common, e.g., regression analysis, classification, and clustering can be put in both categories. Still,

[6]https://www.microsoft.com/en-us/research/group/software-analytics/

models derived by statistical learning are usually designed to prove or disprove a hypothesis about the data whereas machine learning also functions without any expectations about the data. Moreover, statistical models naturally include more mathematical approaches like distribution fitting. In software evolution, possible applications of statistical learning are, e.g., regression analysis to describe growth trends [39], the usage of topic models for understanding text-based software artifacts [40], and Unified Markov Models (UMMs) to test web applications [41].

2.3. Agent-Based Simulation

Simulation is defined as an "imitation of a real-world-process over time“ [42]. As such, it is considered to be a solution to many real-world problems. It is used to "describe and analyze the behavior of a system, ask what-if questions about the real system, and aid in the design of real systems“ [42]. The systems under simulation do not have to exist in real. Indispensable for building a simulation is creating a model of the system to be simulated. Models are abstract images of the real world. Modeling and simulation go hand in hand in simulation research.

There are different simulation paradigms. One is Discrete Event Simulation (DES). DES is very widespread especially in Operational Research [43]. It is designed to let entities switch between different stages over the time (simulation). Thereby, entities often represent people, documents, tasks, or messages. The entities travel through blocks in a flowchart where they stay in queues, get processed, delayed, seize resources, and many more [44]. As an example, one can think of a hospital simulation handling doctors, nurses, and patients waiting for treatment.

Another paradigm is System Dynamics (SD). It is appropriate for business simulations, but also for social science simulations. In SD, entities, e.g., people are represented as so-called stocks. Actions are presented as flows including necessary information about the actions described in a mathematical model in form of differential equations. SD uses an abstract view on the whole system instead of single actions [44]. A simple example models the population of people in the world, where the population stock simply depends on the birth and death rates and factors affecting these.

The paradigm we focus on in this thesis is ABMS. Agent-Based Modeling and Simulation is a relatively new approach in the area of simulations. It is well suitable for modeling and simulating complex systems. The method arose with the wish to model real-scenarios from an individual perspective. In ABMS, the heart of the simulation are the agents, which drive the simulation based on their behavior. In the following, we describe our notion of agents and the detailed concepts of ABMS.

2.3.1. What is an Agent?

To define an Agent-Based model, three fundamental elements are required: a set of agents, a set of relationships, and the definition of the agents' environment [45]. The identification of these three is a difficult modeling task. One key characteristic is the autonomy of agents. This means, they it can act independently which is also reflected in their actions. This concept is the only well agreed agent property in literature. The actions of agents and their goals are described in their behavior. The rules can reach from simplistic to complex artificial intelligence. In literature, more *essential* properties of agents can be found, that help to understand the complex concept of agents [45]:

- *self-containedness*: Also acting as part of the simulation, the agents behave on their own and are uniquely identifiable and recognizable by others. Agents have strict boundaries distinguishing between own and shared characteristics.
- *behavior control*: Agents are autonomous. They are in control of their behavior and, thus can act independently from the defined environment or the other agents as well as the interactions with them.
- *adaptiveness*: Possessing a high degree of flexibility, agents can learn from their behavior and experiences and consequently adapt their behavior according to what they learn. The experiences are stored in some kind of memory. Concretely, the behavior is implemented and triggered by defined rules, which are updated according to the agents experience also controlled by defined rules.
- *goal-orientation*: Derived from their behavior, agents can have goals that they plan to achieve. Therefore, they need the ability to compare and evaluate whether they reached their goals or not.
- *sociability*: Agents are socialized and can interact with each other. This interaction and all kinds of communication are internally organized by protocols.

Generally, an agent consists of attributes and methods with which it can be fully defined. For attributes, we distinguish between non-changeable static attributes, e.g., names, and changeable dynamic attributes, e.g., the neighbors of an agent or its memory. The methods represent the previously mentioned behavior and rules.

Modeling the behavior of agents can be a complex task, especially if no empirical foundations are known to be valid for the desired kind of behavior. It is a common approach to start with a basic model and adapt it step by step. Supportingly, there also exist behavioral modeling frameworks such as Belief-Desire-Intent (BDI) [46], based on which the agent choose the best course of action. Its beliefs represents the actual state of the simulation and all its entities, its desire are the goals the agent wants to fulfill, and its intents consist of the possible plans to reach these goals. Thus, the agents are able to select their actions according to their plans and given circumstances. If the behavior of agents in the desired context is unknown, machine learning techniques can be used to learn the behavior from empirical data and to find patterns.

Another important part in agent-based modeling is the identification and description of interrelations between different agents. When required connections between agents are identified, these connections need to be described. Agents only possess local information stored in their neighborhood, and thus, their interaction area is restricted to this but can be updated running the simulation. ABMS is decentralized, i.e., no global knowledge about the system exists. The neighborhood of agents can be defined using different topologies as basis for social interaction [47]. A simple example presents an aspatial model, where the agents have no location. Another option is to present agents' relations as cellular automata, i.e., the neighborhood is defined as the cells in a grid or lattice directly surrounding the agent. Here, the agents shift from cell to cell and cells cannot be occupied with more than one agent at the same time. Moreover, there are euclidean space models and Geographic Information System (GIS) models. In GIS models, the agents can move over a realistic geospatial landscape. Finally, networks can be used as topology. Macal and North distinguish between static networks, where the links are fixed, and dynamic networks, where the links may change.

2.3.2. Applications and Tools

In practice, ABMS is used in many research directions, e.g., biology, physics, social sciences, geology, air traffic control, and economy. Naturally, Agent-based methods are suitable for all kinds of real world problems where the evolution is driven by specific behavior of groups of individuals. A major advantage is that the complexity in modeling arising from this individuality can be captured with ABMS. Even relatively complex real world scenarios can be modeled easily. Of course, this aspect depends strongly on the application context. Generally, agent-based models can be implemented using ordinary programming languages and tool. There also exist special tools especially suited for the agent concept. The simplest way can be to use spreadsheets. Still, this is not suitable for very complex models reflecting sophisticated agent behavior. Also mathematical and standard programming languages can be used for Agent-based modeling. However, for large-scale Agent-based models, dedicated toolkits offer more possibilities.

ABMS tools can be divided into Desktop Computing and Large-Scale Environments [48]. Desktop ABMS include spreadsheets and Agent-based environments like NetLogo [49] or Repast Symphony [50]. The method followed in this thesis mainly focuses on Repast Symphony, a Java-based ABMS toolkit. It allows agent behavior and results specification. A model designer allows to design the spatial and logical structure of the models in a visually supported way. NetLogo is based on a Logo-related programming language and also provides ABMS specific features. In addition, mathematical oriented tools like MATLAB and Mathematica are used for ABMS purposes. There, the agent side needs to be developed by oneself, but it allows a wealth of numerical modeling opportunities, e.g., partial differential equations (PDEs), which are often used to model phenomena in physics like fluid flows.

As the requirements for computational power and distributed simulation steadily grew over the last years, also the recent (Agent-based) simulation frameworks try to encounter this problem offering large-scale ABMS software. Scalable platforms are for example Repast HPC, which shares the core of Repast, but allows for parallel distribution. Another, but commercial, tool is AnyLogic. A rather advanced tool also confronting the growing complexity is GAMA [51]. GAMA uses its own modeling language and provides the usage of Geographic Information System integration (GIS) data, i.e., realistic landscapes and spatial representation.

2.3.3. Simulation of Software Processes

The simulation of software processes is no new topic [52]. But its way of solving this task has changed and evolved over the years. Whereas in the beginning, the focus laid more on the business and strategic aspects, this changed with the growing amount of open source projects and their impact into additionally understanding software evolution and occurring patterns. Generally, in software engineering, simulation is used to aid project managers in their decision making. It is used to forecast possible future scenarios depending on project properties defined as parameters. However, often such simulations lack to present a whole picture of the software development process including the interplay of different parts of the process.

An overview by Zhang et al. [53] reported the usage of different simulation techniques in software engineering research. They found out, that most simulation studies focus on the time, effort or cost, (product) quality, (requirement) size, and resource, e.g., staffing level. For the simulation paradigms, they recognized SD and DES as most popular. However, considering the point in time of that literature review, the results may have changed meanwhile. Still, ABMS is rarely used in this context nowadays.

Thinking of a software project as a set of humans striving for the fulfillment of requirements and coordinating themselves as well as with each other, it seems fruitful to model software evolution from the starting point of the developers and their behavior. Thus, ABMS poses an appropriate choice for modeling software development processes. Although following the same goal of achieving high quality software, the developers or agents occupy their own individuality, i.e., they act on their own. For simulating software evolution from a developers perspective, the need for a deep understanding of the underlying processes and relationships triggering software evolution is presumed. Therefore, knowledge from past projects can be of help to model software evolution and to simulate it. This knowledge is usually gained from software repository mining (see Section 2.2).

One aim of our work is to capture the dynamics occurring in OSS development. OSS lives from contributions made by a diffuse set of programmers, often volunteers [54]. This influences the projects development and outcome. In contrast, it leads to commercial development to special dynamic patterns which depend on many individual factors, e.g., team constellation, activity, motivation, experience, project context, and project size. Build OSS models should reflect this unsteadiness.

2.4. Developer Contribution Behavior

For modeling software evolution, the developers are an important part of the system. They actively control the evolution process [8] and have an direct impact on other entities in the evolution life-cycle like the artifacts they change or the bugs they create. To describe developer behavior close to reality, a deeper understanding of developer contribution behavior and related effects is required. Therefore, we study this in more detail.

Developers' behavior can depend on many factors, such as motivation, experience, background, and organizational structure. Changes in the organizational structure can have an impact on the overall quality [55]. Thus, it is practicable to begin our investigation from the inside. A way to describe the structure of a software project is the use of developer roles. Discriminators for this can be the importance analysis of nodes in developer social networks [56], based on contribution profiles derived by mining [57], or machine learning methods like clustering [58]. It is a well known phenomenon that such structures and roles of developers may change over the time [59]. Hence, it is of importance to reflect potential dynamics and their related impact in respective contribution models.

To describe the behavior of developers according to their role, it is possible to measure their activity over the time and derive specified models from that. The contribution of a developer can be measured using various approaches and metrics: Lines Of Code (LOC) written [60], number of commits [61], or amount of files changed. Also social and communication factor can be taken into account, e.g., mailing list activity, bug comments, forum entries, and IRC participation [62].

Within studying the behavior of developers, it is likely to test real world observations towards specific expectations on how and why developers behave a certain way. It is desirable to put behavioral aspects into categories and summarize similar characteristics. Using such characteristics, one get a broad picture of developers' behavior. Considering involvement and activity factors of individuals, can help to evaluate the state of the software project

Moving this motivation from the individual-based to the project-based level, it is possible to access the ongoings in the project itself. In contrast to developer activity, project activity is also influenced by the user participation and feedback which can be given, e.g., through mailing lists, forums, issue reports, or download rates. As such, user feedback an be seen as a driving force in software quality improvement [63]. Often, user interest is seen as an contributing factor for project success [64, 65].

From the viewpoint of a project manager or a developer considering the import of a certain OSS project, it can be of help to identify the potential risks of usage beforehand. For example, a low project activity may indicate an unmaintained or dying project [66]. Still, the point in time since when a project can be seen as "inactive", is difficult to determine. Often, certain thresholds like a timespan of continuing inactivity are defined to find unmaintained projects, e.g., in [67]. But, similar to developer activity, a software project may not be inactive even if no (or few) commits are visible in the VCS, e.g., the project may be in a discussion phase. Thinking the other way around, not only complete absence of activities, but also a small amount or very irregular activities, may indicate risks of using the project. Thus, the challenge is to identify risky trends as well as problematic states of project activity.

2.5. Hidden Markov Models

To model not only the aspects of software evolution that are visible from the outside and can be easily extracted and measured, but also non apparent patterns, it can be convenient to make use of Hidden Markov Models (HMMs) for modeling OSS dynamics. HMMs are stochastic finite automatons able to handle discrete time sequences. Each state is independent of every other (Markov property) [68]. Given such a sequence, the theory of HMMs assumes that there exist a corresponding sequence of hidden states which describe underlying, hidden processes. In our case, the sequence presents, e.g., the contributions of a developer in a certain amount of time.

In the following, we state the definitions we need to define HMMs and that are used in this thesis. A HMM consists of initial probabilities for the states, transition probabilities between these states, and emission probabilities for all observations that can occur.

2.5.1. Notations

Let $S = \{S_1, ..., S_N\}$ be the set of states with N being the number of the states. Let $V = \{v_1, ..., v_M\}$ be the set of observations with M being the number of distinct observation symbols. First, we have the initial state distribution indicating how likely it is to be in the different states S_i in the beginning. This can be expressed by $\pi = \pi_i$ with π_i being the probability for state S_i at point $t = 0$. Next, we define the transitions between states. Therefore, we consider the probabilities a_{ij} for passing on to sate S_j when residing in state S_i. These probabilities are managed in the so-called transition matrix $A = \{a_{ij}\}$. The third thing to define a HMM are the emission probabilities. These describe the probabilities of the possible observations to occur in the different states and can be defined by $B = \{b_j(k)\}$ with $b_j(k)$ being the probability for v_k to occur in state S_j. This means that $b_j(k) = p(v_k|S_j)$. Altogether, a HMM can be wholly defined by $\lambda = (A, B, \pi)$.

Following Rabiner [68], there are three basic problems for the application of HMMs. The first real-world problem deals with calculating the probability of an observation sequence to occur given a fixed model. The second problem also starts with an observation sequence and a given model as input, but its aim is to detect an appropriate state sequence that explains the observations best. Most important for our work, is the third problem being the adjustment of model parameters $\lambda(A, B, \pi)$ to maximize the probability of observations.

These observations can for example be a sequence of signals, words spoken, nucleotides, or diverse kinds of observed behavior. This problem also states the most complex one. Given a concrete sequence of observations described by $O = \{obs_1, ..., obs_n\}$, then the aim is to train the model $\lambda(A, B, \pi)$ that has the best fit for it, i.e., to maximize the likelihood $L = p(O|\lambda)$. For this, the application of different optimization steps and algorithms is needed. In the following, we shortly describe the algorithms that can be used to solve this problem.

2.5.2. Baum-Welch algorithm

For the HMM training, we start with determined initial model parameters for the transition matrix A and the initial state distribution possibilities π_i for the i states. The initial emission can also be determined, but it is more convenient to estimate them from the observations if possible. This can be done by calculating means and variances of all observations. Based on an auxiliary function, the initial model parameters $\lambda(A, B, \pi)$ are re-estimated by the Baum-Welch algorithm. Within this step, the algorithm calculates the amount of occurrences of each transition as well as emission. The re-estimate can be intuitively interpreted [69]: The adapted emissions present the relationship (ratio) between the expected number of an observation to occur in a state and the expected number of times the state is emitted. For the transitions, also expectation values of the occurrences of the transitions are calculated and the parameters adapted based on these values. This step is repeated until convergence.

2.5.3. Viterbi algorithm

After training the model with the Baum-Welch algorithm, we can apply the Viterbi algorithm [68] to calculate the corresponding state sequence to a given observation sequence $O = \{obs_1, ..., obs_n\}$ which produced the observations most likely. This can also be referred to as the decoding step, since it reveals the corresponding states to the produced output, e.g., a word in speech recognition. The algorithm determines the most probable partial path of length n and use this together with the emission probability from the predecessor node to the current node to trace back the most likely path.

The overall process is depicted in Figure 2.4. There a given observation sequence $O = \{obs_1, ..., obs_n\}$ is taken as input and the state sequence that most likely produced these observations is desired (as output). For the actual training of the HMM, i.e., to determine the transition probabilities a_{ij} and the emission probabilities $b_j(k)$, the Baum-Welch algorithm is used. Then, using Viterbi, the most likely corresponding state sequence is calculated.

2.6. AB/BA crossover

All kinds of experiments require a careful design as well as evaluation to draw accurate conclusions. So-called crossover studies origin from medicine and other healthcare-related disciplines as well as psychology [70]. In crossover designs each treatment is applied to

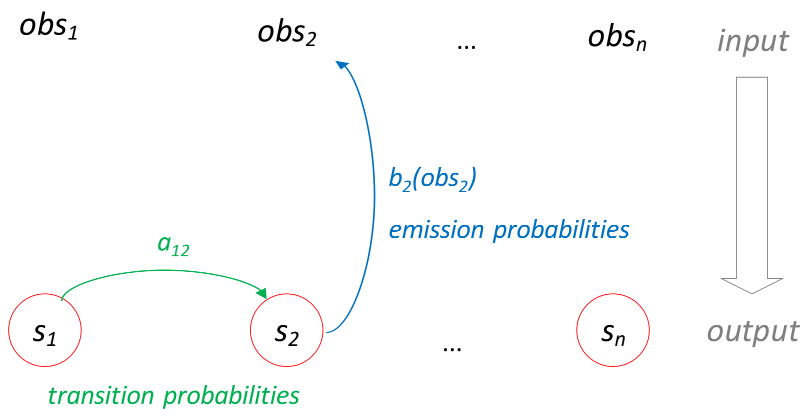


Figure 2.4.: HMM training and prediction of the most likely state sequence given a sequence of observations. [12]

every participant, but the order of treatment is different. In medicine, it is used to investigate potential side-effects of medicaments and therapies. AB/BA crossover experiments divide the participants into two groups: One group use technique A before technique B and the second group vice versa. The first mentioned group is called Sequence Group 1 (SG1) and the second group is called Sequence Group 2 (SG2). Hence, we have two techniques and two periods in time, the first period when the first group uses technique A and the second group uses technique B, and the second period where the first group uses technique B whereas the second group uses technique A. Note that, this kind of study design is not able to measure effects that stem from the order of techniques used.

Applications of AB/BA crossover in software engineering research range from the assessment of comprehensibility of (annotated) UML diagrams [71] to comparisons of different test case design techniques [70]. Vegas et al. also reported some best practices to adhere when performing crossover studies in (software engineering) research.

A critical view on AB/BA crossover studies in software engineering was recently published by Madeyski and Kitchenham [72]. They pointed out the importance of the usage of different descriptive statistics, i.e., non-standardized and standardized effect sizes. Effect sizes generally measure the dimension of a phenomenon, e.g., the degree of the treatment effect. They make different outcomes of a study comparable. Often, the effect size is calculated subtracting the averages of the outcomes (standardized difference of means). An example for such an effect size is Cohen's d [73] which divides the mean difference by the standard deviation. Still, it is important to assess whether the effect size presents a meaningful result is important. For Cohen's d, the effect size is small when $d = 0.2$, medium for $d = 0.5$, and large for $d = 0.8$.

Following Madeyski and Kitchenham [72], effect sizes can be measured using the following formulas and rationals.

Let τ_A be the effect of technique A and τ_B be the effect of technique B. Moreover, let τ_{AB} be considered as the difference between both techniques, i.e., $\tau_{AB} = \tau_A - \tau_B$. This directly implies $\tau_{BA} = -\tau_{AB}$. Let π be the period effect size which measures the difference between using a technique in the first and in the second time period. Madeyski et al. [72] also defined λ_A and λ_B as the period by technique interaction for applying method B after method A and method A after method B, respectively. Then, λ_{AB} is the difference between these two interactions and, thus, the mean period by interaction effect size. Finally, let μ_i be the mean outcome for participant i with $i \in 1, ..., n$ the number of participants.

The following calculations are taken from Madeyski et al. [72]. Let MCO_1 be the average crossover difference of the n_1 participants in $SG1$ and MCO_2 be the average crossover difference of the n_2 participants in $SG2$ with $n_1 + n_2 = n$. Comparing the average difference in the results achieved with method A followed by method B with the average difference in the results achieved by using method B first determines the technique effect size. Following, we denote the following for the technique effect size τ_{AB}:

$$\tau_{AB} = \frac{MCO_1 + MCO_2}{2}, \tag{2.6.1}$$

In contrast, the period effect size considers the differences between the outcomes in the two periods instead of the techniques. For the period effect size π, we get the following equation:

$$\pi = \frac{-(MCO_1 - MCO_2)}{2}. \tag{2.6.2}$$

Now, let $MSG1$ be the participant total of $SG1$ and $MSG2$ the participant total of $SG2$. Considering the difference between the results of all participants within the two sequence groups defines the technique interaction effect size. Thus, for the estimation of the technique interaction effect size λ_{AB} we get:

$$\lambda_{AB} = MSG1 - MSG2, \tag{2.6.3}$$

For the variances, we stick to the following calculations. First, we calculate the within period and within technique variance:

$$s_{IG}^2 = \frac{\sum_{t,p}(n_t - 1)(y_{t,p,j} - \hat{y}_{t,p})^2}{2n_1 + 2n_2 - 4}, \tag{2.6.4}$$

with $y_{t,p,j}$ being the outcome of participant j with technique t in time period p.

Next, the difference score variance is defined as follows:

$$s_{diff}^2 = \frac{(n_1 - 1)\sum_j(CODiff_j - MCO_1)^2 + (n_2 - 1)\sum_k(CODiff_k - MCO_2)^2}{n_1 + n_2 - 2}, \tag{2.6.5}$$

with $CODiff_j$ being the crossover differences of participants in $SG1$ and $CODiff_k$ the differences in $SG2$.

The within participant variance s_w^2 can be derived by:

$$s_w^2 = \frac{s_{diff}^2}{2}. \tag{2.6.6}$$

Anymore, the correlation between the outcomes in both time periods is defined as:

$$\hat{\rho} = \frac{s_{IG}^2 - s_w^2}{s_{IG}^2}. \tag{2.6.7}$$

Then, the variance of the technique effect size can be determined with the following formula:

$$var(\tau_{AB}) = \frac{var(MCO_1) + var(MCO_2)}{4}. \tag{2.6.8}$$

In addition, for the standard error of this effect size we have:

$$se_{\tau_{AB}} = s_w \sqrt{\frac{(n_1 + n_2)}{2n_1 n_2}}. \tag{2.6.9}$$

Finally, for testing the significance of τ, we use the $t - test$ [74]:

$$t = \frac{\tau_{AB}}{se_{\tau_{AB}}}, \tag{2.6.10}$$

with $df = n_1 + n_2 - 2$ being the degrees of freedom. From this, the $p - value$ can be calculated to test the significance of the result.

3. Related Work

Our work combines different prominent research topics. The specialty lies in the interplay of different methods from these topics. The mining of software repositories is a powerful vehicle to approach the problem of describing all aspects belonging to the software evolution life-cycle. Thus, we start with an overview on related work directions in mining software repositories to detect software evolution trends. Since the investigation of developer behavior constitutes a big part of this research, we continue to set our work in context to research done in this area. Afterwards, we report the state of the art on software process simulation, because the simulation of the retrieved knowledge represents a relevant application to our work. We complete the chapter giving an overview on related work on software project activity.

3.1. Mining of Software Evolution Trends

The investigation and analysis of software evolution trends is recently a well-researched topic. This lies in the rich amount of freely hosted data as well as in the rising availability and quality of analysis tools. Hence, it is very attractive to researches. The treated dimensions of problems range from software development phases, programming languages, and environments to the software management process [75]. The diverse toolbox for researchers include visualization techniques, empirical analysis, machine learning, statistical learning, simulation techniques, and data mining. Since software evolution deals at the first instance with the changes to the software system that make the system evolve, the analysis of changes play a key role in mining software evolution trends. The mining process can be a laborious task due to missing links between repositories, noisy data, or inhomogeneous naming conventions. Thus, the repository data has to be preprocessed carefully. One of the first aspects to analyze is system growth.

For this, often the number of modules [39], files, classes or methods (changed) is used. Usually, this growth follows a sub-linear trend decreasing with growing maturity of the project [76]. Paulson et al. [77] investigated the growth trends of OSS compared with closed-source software projects. They also compare different growth measures such as the number of functions, lines of code, and complexity. For all measurements, a linear function could be fitted. Furthermore they observed a similar growth in number of functions and lines of code for open-source software as well as closed-software projects. In addition, also super-linear trends can be found in OSS [78]. Finally, Capiluppi et al [79] considered the growth in terms of number of files and number of folders. In doing so, segments of growth could be identified with sub-linear parts. In our work, we consider the growth of the software system in the number of files.

How developers collaborate and how this collaboration evolves over the time also constitutes a major part of our work. From the view of a project manager, it is important to keep track of the interaction between developers and how they collaborate. For this purpose, often social network analysis (SNA) is used. Therefore, different networks are created for different types of research, e.g, to identify core developers [80] or to predict failures [81]. It is common to use information retrieved from the VCS (collaboration-based) [3] as well as from the ITS and ML (communication-based) [82] to build theses networks. The structure of these graphs can offer valuable clues to the distribution of work among developers as well es the connectivity. As a project manager, a less fragmented social developer network is more desirable, because the opposite can have a negative impact on software quality [55]. An approach similar to our work is presented by Pinzger et al. [83]. There, so-called *contribution networks* are defined which equals developer-file networks. This means for every commit by a developer an edge is created to every file included in the commit. This way, not only the collaboration of developers can be traced, but also important files can be construed. To this reason, we decided to model developer dependencies in the same way. In their work, Pinzger et al. used different network centrality metrics (degree, closeness, betweenness) and calculated the correlation with post-release failures. As a result, they observed a high correlation for two of the three metrics.

To represent the relationship between software artifacts, different software graphs can be taken into account. These are call graphs based on the caller-callee relationship between classes and functions, hierarchy graphs based on inheritance, or graphs based on co-changes of entities (change coupling). Following Ball et al. [84], a change coupling graph draws edges between files that are frequently changed together. Ball et al. showed that these graphs are useful to discover clusters of files that are often changed together and that files within the same cluster are semantically related. For software quality, D'Ambros et al. [85] found out that a high change coupling can indicate hard to maintain areas and structural issues such as architecture decay. They studied the relationship between change coupling and software defects and also found out that change coupling is a better predictor for software defects than complexity. Thus, defect prediction models could be advanced using knowledge retrieved from change coupling networks. Another application of change coupling graphs was presented by Knab et al. [86]. Within their work, they use several metrics from the change coupling network for the prediction of the number of defects. However, in their study no positive effect on the prediction model could be detected. Finally, Zhou et al. [87] trained a Bayesian network to predict change coupling behavior given information on past networks such as co-change frequencies and co-changed entities. This way, they provide support for developers working on a change request by recommending related change candidates.

In this thesis, we also consider change coupling network as a representation of file dependencies and analyze the evolution of clusters. Recently, some studies raised that investigate the effects of the evolution of change coupling on, e.g., the number of defects [88] or bug localization [89]. Thereby, the representation of software artifact dependencies as change coupling graphs is an interesting topic.

Besides developers and artifacts, e.g., files, software evolution is concerned with bugs. A lot of studies exist to assist in bug fixing [90], bug-artifact linking [19], the prediction of buggy changes [91], or the prediction of the severity of bugs [92]. For the simulation of quality trends, we restricted our work to the occurrences of bugs and the lifespan of different types of bugs. Weiß et al. [93] used text similarity to predict the time until an issue gets fixed with machine learning. The predictor is applicable to the time of the bug report, which could help project managers in their planning. They also use a machine learning technique (decision trees) to classify the bugs into fast and slowly fixed. The following attributes were observed to be the most influential ones: the assignee, reporter, and month in which the issue was opened.

3.2. Developer Contribution Behavior

Developer's (contribution) behavior is a well researched topic [9, 59, 94, 7]. Generally, developer contribution can be seen as all activities a developer performs during software development [95]. Nevertheless, no unique definition in software engineering exist. Manifold measurements exist to express contribution: the number of LOC written, the number of commits, number of files changed, involvement in developer social networks, or a combination of some of these metrics. As Gousios et al. [95] pointed out, the aim of the investigation of contribution behavior is to assess the course of the software project examined as well as to assist in future project planning. In the following, we list some work dealing with the assessment of developer behavior, and work that gathers dynamics in contribution behavior.

Girba et al. [8] presented an approach to understand the behavior of developers. They defined the *ownership* of a file that determines the developer who edited the most part of it. Based on that, they provide a characterization of developer behavior by analyzing patterns that indicate different activities like it is shown in Figure 3.1. The colors represent different authors performing different activities (e.g., edits, fixes) on a file over the time. As an example, the action *takeover* describes the behavior where a developer performs a few large commits in a short amount of time taking over the possession of the file. However, this work is designed for visual exploration whereas the focus of our work lies on the analysis part.

An interesting study where the authors consider also feedback from project leaders on commonly used measurements of developer activity is presented by Lima et al. [7]. In their paper, they report the results of discussions on the usefulness of contribution metrics with project and team leaders, since for them it is most important to interpret these measures. As such measures, they consider LOC, the average complexity per method, introduced bugs,

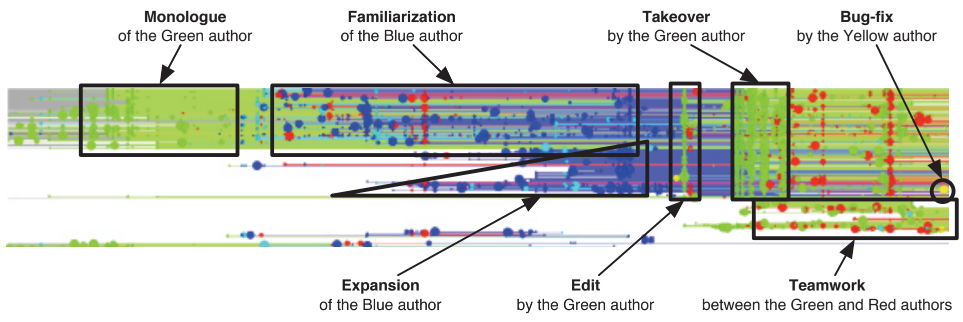


Figure 3.1.: An example of an ownership map [8].

and bug fixing contribution (relative to other developers). Lima et al. found out, that code contribution metrics are useful for project leaders, but should be extended with other information, e.g., complexity of tasks. Similarly, bug introducing and fixing rates should also be put into context of the amount and complexity of tasks. Summarily, metrics should never be considered isolated. Likewise, we also use a combination of different metrics to characterize developer contribution behavior.

A very broad picture of developer behavior is provided by Makedonski [96]. There, a diverse investigation of developer-related aspects is provided including different levels of granularity, a deep analysis of the impacts and causes of changes, and the application to different software engineering tasks, such as the prediction of bugs. Moreover, the mining process is realized as a model-based approach which can be adapted to different mining tasks. A strength of this work is that it offers a lot of opportunities for analyzing software repositories and that the mining, modeling, and analysis go hand in hand. This constitutes a different approach to mining software repositories where the focus lies on the modeling compared to our work where we focus on the analysis part. Both approaches are combinable.

For understanding project collaboration as well as to assist in developer team constellation planning, it is essential to consider developer roles. As mentioned in Section 2.4, there exist a couple of ways in doing so, e.g., count-based classification or network-based classification.

Bhattacharya et al. [3] identified different roles to describe the expertise of developers. The role definition enfolds testers, assists, triager, analysts (all bug related) as well as core developers, bug fixers, and quality improver (source code related). Developers can serve different roles at once based on the kind of contributions performed. Afterwards, collaboration graphs are build from that and then hierarchically ordered. In our work, we consider only four specialized roles (core, maintainer, major, minor).

Besides such specialized approaches, it is common to classify developers into core and peripheral [58, 97, 98, 99]. Though, core developers are responsible for decisions and perform more work, whereas peripheral developers are less active. The underlying assumption is that the OSS structure can be described as the well-accepted onion model [100] where a small cluster of developers do most of the work. Note that, peripheral developers can also be users and bug reporters. For sustainability, it is important to keep the core alive. The classification is often done by taking the top 20% committers over a certain period of time [101]. We proceed similar, but we split the peripheral developers into major and minor ones. Terceiro et al. [99] investigated the relationship between this role structure and the introduction of structural complexity. By that, they concluded that core developers introduce less complexity than peripheral developers. In our work, we only consider the amount of work and the kind of changes done by developers exhibiting different roles, but not related to complexity introduced by the changes.

A recent study on this topic was conducted by Joblin et al. [102]. They evaluated the insights gotten from count-based developer contribution measures and compare them with metrics retrieved from hierarchy networks. Although they found that the count-based metrics produce consistent results, they showed that insights gained from networks based on mailing lists or collaboration extracted from the VCS can enrich the investigation. Thus, we are confident to use count-based measures for e.g., commit activity and also use mailing list activity, but do not consider collaboration networks for this task, since we are mainly interested in a model that allows for different levels of activity and their impact on the output.

As most related to our work we identified the approach of Singh et al. [103]. Within their work, they studied the effects of peer learning compared with individually gained knowledge concerning the experience of software developers. Like we do, they build a HMM to investigate learning dynamics for OSS developers. In a large case study with 25 open source software projects with 251 developers involved they validate their work. The retrieved findings are compared to a classical learning curve model. By training the HMM, different states of project activity over the time for each developer can be learned. As a result, they found out that, e.g., developers resigning in a low state benefit more from peer learning activities while in higher states individual learning is more fruitful. Our work differs from the work of Singh et al. in so far that we do not focus on the learning itself and, instead, consider the learning as implicitly given part of our model. In addition, our model needs less input metrics: Whereas Singh et al. also take, e.g., the rank of the project and the project age into account, we solely focus on contribution as well as on communication activities which makes our approach easy to use.

3.3. Software Process Simulation

Using simulation techniques to aid software project managers in decision making is known for a while. The topic was dominated by discrete event simulation (DES) as well as system dynamics (SD) in the past. Software process simulation modeling is recent since 1998 [52], where the first requirements and guidelines for software process simulations were postulated. Since then, some work has been done in this area [53], but it often lacks in considering the software as a whole and, instead, focuses on particular factors. Only a few studies take ABMS for software evolution into account.

An approach simulating the evolution of developer networks is presented by Gao and Madey [104]. In their work, they use SWARM [105] to calibrate a simulation model iteratively to replicate observations retrieved from SourceForge. There, developers can select whether to join, stay in, or leave projects. This approach is more global than ours and aims to understand OSS evolution at first sight. Interestingly, this is one of the few studies taking empirical data for the simulation parameter estimation as we do.

Another study proceeding this way, was introduced by Spasic et al. [106]. They provide an Agent-based simulation model that helps project managers to estimate the completion date as well the required number of developers for a set of software artifacts. Moreover, they concentrate on the design and development phase given by Capability Maturity Model Integration (CMMI, e.g.,[107]). CMMI is tool to optimize software processes. Thus, this study is more specific than our work, but can be seen as an indicator that it is worth investigating ABMS for software project management. Both Spasic et al.'s model and our model compare empirical with simulated data for their validation.

An approach that compares ABS for software processes with SD taking individual factors such as experience and competence of developers into account, is presented by Cherif and Davidsson [108]. They state that SD is easier to configure, because it does not need as much information on the behavior of individuals. On the other hand, ABS produces more realistic results. In contrast, our approach also allows for individual developer behavior like developer roles and alternating levels of project involvement and, thus, output.

Using ABS, Agarwal et al. [109] also present an approach that considers different individual factors, e.g., LOC, time spent, and defects removed using Personal Software Process (PSP). The decisions of developers are based on individual PSP data, which stores statistics over past activities. Compared with our work, this approach is more tailored towards specific types of projects, since it is controlled by a given process, Extreme Programming. A similarity of their approach and ours is the allowance for individual developer behavior expressed by agents.

An approach uses Agent-based simulation for software evolution as we do was presented by Smith et al. [110]. In their work, they present an approach to simulate software evolution in terms of size, complexity, and distribution of changes. The active agents are the developers and they are defined on a grid walking randomly around. In doing so, they can come across a requirement or module to work on. If they do so, depends on the interestingness of the module for the developer and the complexity of the module. Besides, they can leave the project by moving outside the grid. The work of developers on the passive agents, i.e., the modules, include the creation, extension, and refactoring of modules. These actions have an immediate effect on the module metrics: fitness and complexity. To measure the systems size, Smith et al. use the number of functions. The most significant difference of our work compared to Smith et al. lies in the topology design: We use networks to represent relationships between software entities instead of a grid. This provides several advantages. For example, other types of relations apart from proximity can be taken into account [47]. As such, developer collaboration, file dependencies, and bug links are provided in our work. Besides, our model allows for tailored actions of developers (create, modify, delete files) according to their role.

3.4. OSS Project Activity

As already highlighted, OSS lives from the developers contributing to the project [98]. As such, developers form a huge pillar of the whole project activity. Other factors influencing the project activity, are the users [111]. Besides from accounting users for judging the success of OSS projects, they are also beneficial in the guidance of software development (decisions). For evaluating the sustainability, some work exists taking project activity into account like it is done in this thesis. In the following, we summarize the most related work in this area.

A recent study by Coelho et al. [66] propose a machine learning based model to identify software projects on GitHub that are not maintained anymore. The employed feature set includes several activity metrics, e.g., the number of forks, amount of new developers joined, opened and closed pull requests, and commits, that are extracted for different periods of development differencing in the length and interval. They evaluated their approach in a large case study with over 6000 projects and identified as most relevant features the number of commits, the maximum days without commits, and the maximum contributions by a developer in a certain period. Unlike other studies, they were able to identify unmaintained projects although there are commits done in the last year.

Many studies have in common to use certain thresholds. For example, a project is considered inactive after one year of inactivity [67]. In other studies, sporadic contribution activities are considered inactive [66] too, because few commits performed may be due to very small changes. The usage of thresholds can be problematic as it is not clear when a project is in a critical (dying) state and when is it really inactive (dead).

An interesting study presented by Khondhu et al. [67] investigates characteristics of projects becoming inactive. They divide OSS projects into three categories: *active* with visible activity, *dormant* where the activity stopped for a certain period of time, and *inactive* as explicitly stated so by the developers. For the differentiation between active and dormant they defined a threshold of one year of inactivity. Using this classification for all SourceForge projects from a given data-dump, the authors compared different metrics across the activity categories to find similar patterns. As a result, they showed that active projects consistently grow larger than others. Another interesting result revealed that the maintainability index of inactive projects for most projects remain stable.

We combine different aspects of the mentioned studies in our approach: We also apply a learning technique to identify active projects. But instead of machine learning we use statistical learning. As metrics we also take developer activity into account in combination with user participation.

4. Simulation Models

In this chapter, we first sketch the general modeling approach followed by an introduction of our basic simulation model, which establishes the starting point of our work, and is used for all further refinements. Afterwards, we present two refined models for the simulation of software evolution. The first includes the representation of developer contribution behavior allowing for dynamic software project participation. The second model deals with software development phases of the regular life-cycle.

4.1. General Modeling Process

In this section, we describe the overall process we identified to create simulation models which describe software evolution. This process is illustrated in Figure 4.1.

The rectangles represent the steps and the ellipses represent the input/output of the steps. Starting with a concrete research question regarding software evolution in mind, the first step is to identify important model parameters and their interrelations which are needed to answer this question (*Agent-Based Modeling* in Figure 4.1). As an example consider the following question: *How will the growth in LOC of the Eclipse project evolve in the future given the current developer team constellation?* Then, model entities of interest are software entities, e.g., files or classes with LOC as attribute and an environment where the system LOC as sum over all entities is stored and a counter for the simulation round, e.g., representing one day in the software development life-cycle. In addition, the commit behavior of developers and the size of the commits in LOC is needed. To instantiate such a software evolution model, the parameters need to be estimated – or simply set if known. e.g., from literature. For this estimation step, we use freely available open source repositories and retrieve knowledge from that by mining (*Mining Software Repositories* in Figure 4.1). In the example scenario, the project under investigation is the Eclipse project and, thus, only information from Eclipse repositories need to be extracted. Relevant information in this context are the history of each software entity, i.e., when it was changed and the LOC added or deleted in this change. It can be the case that metrics have to be calculated if not directly available. As a next mining step, the extracted data needs to be analyzed with the aim to find patterns or descriptions for the desired scenario. Following our example, this could be the analysis of the evolution of the total LOC. A possible method for this is the usage of regression analysis that tries to fit a curve that can describe the observed behavior. For the commit behavior of developers, averages of the overall activity can be taken into

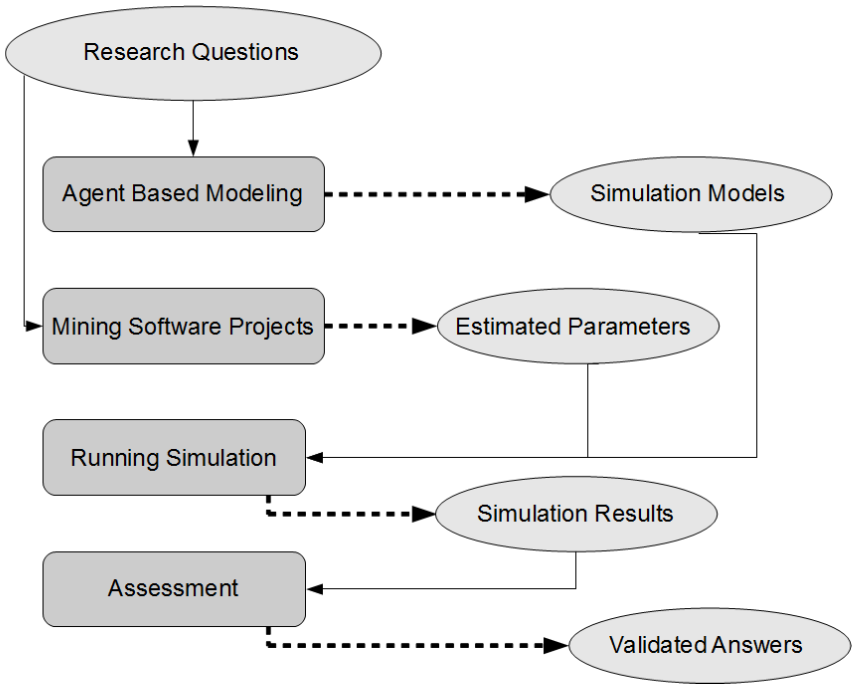


Figure 4.1.: Process of building tailored simulation models. [112]

account. With the gained knowledge the model can be instantiated for this project. Using simulations allows to forecast the future under given parameters which can be changed, i.e., the team constellation and its impact on the system growth. Since by the nature of simulation there can be a discrepancy between empirical and simulated behavior, the project under simulation can behave different than in reality. If this gap is too big, it may be necessary to examine the simulation results carefully and potentially adapt the model (*Running Simulation* in Figure 4.1). When this is done, the results can be interpreted and assessed (*Validated Assessment* in Figure 4.1). However, the assessment is not part of this work and is covered, e.g., in [113].

During our work, we established three different models reflecting different facets of software evolution. We will introduce these models in the next sections starting with our essential model that serves as foundation for the afterwards following refinements.

4.2. STEPS Simulation Model

This section gives an overview on the general model we created for describing and simulating software evolution named STEPS (Software Trend Evolution Prediction in Simulation). We tried to keep the model simple, though having the ability to answer a set of central ques-

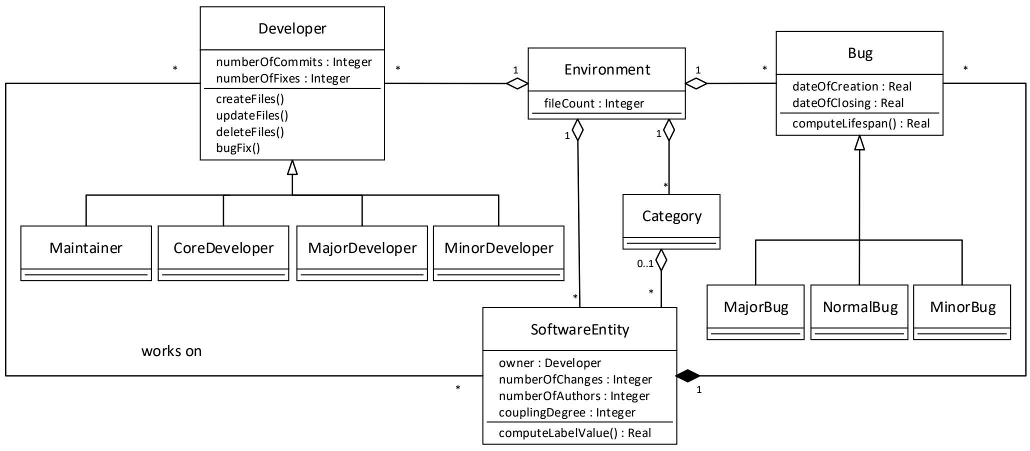


Figure 4.2.: Essential Agent-Based Simulation Model for Software Evolution (adapted from [112]).

tions concerning software evolution. For every model entity we describe why it is selected and how it is related to other entities. The proposed model is shown in Figure 4.2.

Since software evolution is concerned with people, artifacts, and bugs (see Section 2.1), those constitute the main parts of the model. From an Agent-based perspective, developers are individuals who live in their environment (software project) and struggle with the different requirements and (conflicting) interests. Triggered by defined behavioral rules, e.g., the transaction of commits as well as their consequences and relation to other agents, complex scenarios can be mirrored in an Agent-based simulation, e.g., the overall (quality) trend of the software project. Defined in their environment, agents can also interact with each other. In the following, we explain the model entities step by step.

Developer

Developers represent the heart of the simulation. As the only active agents, they are responsible for all explicitly performed changes to the software system. The description of the agents behavior is a key aspect in ABMS. Thus, it is important to model how developers work and react. Developer behavior can be modeled using differing metrics [95]. For our first model we decided to describe this behavior by commits and bug fix commits the developers perform to the system, since these metrics are easy to retrieve and understand while they present a good indicator for the work performed in a software project [114]. The amount of (bug fix) commits is stored for each agent. Within a commit, the developers perform a set of actions that consist of the following types of changes: create software entities, update entities, and delete entities. In this thesis, we refer the software entities to be files, but other types like classes or modules are manageable within the model. Consequently, this represents the direct link to the software entities. Of course, they are also responsible for the creation of bugs introduced by certain commits.

Moreover, we define a specialization of the developer such that a developer agent can be of different types: *CoreDeveloper*, *MajorDeveloper*, *MinorDeveloper*, and *Maintainer*. This decision is based on our understanding of developers' contribution behavior. As mentioned in Section 2.4, the use of developer roles can help to understand the organizational structure and its impact on the software (quality). Some studies exist (e.g., [115, 99]) dealing with the special role of core developers in open source projects. Thus, they play an important role and are worth to be considered separately. Their specialty lies in their workload and knowledge of the system. In some projects, there also is the special role of a maintainer present (e.g., [116]), who is responsible for maintaining (special parts of) the system. Since we aim to give an overall statement on the quality of the software, this is a suitable role for our simulation. Besides, we distinguish between major and minor developers because the impact of the changes performed by less experienced developers can vary in comparison to more well versed ones [61].

SoftwareEntity

The active developer agents perform their changes on the passive software entities describing the daily work in a software project. The defined actions on the entities *additions*, *modifications*, and *deletions* date back to the work of Lehman [13] and are still current. While for some applications, e.g. description of refactorings, a more fined-grained description may be beneficial, our description is sufficient to describe basic software changes. Some attributes can directly be calculated by the changes to the software entities: the *numberOfChanges* as count of all actions performed on a certain entity and the *numberOfAuthors* as the amount of distinct developers that performed at least one action on the software entity. The *owner* of an entity can also be retrieved directly. Bird et al. [61] describe the term ownership as an description for the responsibility of a person for a software component. There exist several approaches to define the ownership, such as the percentage of the development activity coming from one developer [61]. It can be calculated by the ratio of changes by a certain developer relative to the total number of changes. Another way to describe ownership would be for example the maximum number of lines of code written by a developer. We stick to the definition of Bird et al., but instead of using ownership from the point of an software entity, i.e., having a low or high ownership, we store the author with the highest ownership, i.e., most changes to it, as its owner. Modeling ownership can also be beneficial for assessing software quality [117].

Furthermore, we consider the *couplingDegree* of an software entity. For important relations between agents we decided to use networks as topology. All networks are described later in Section 4.2.1. Conceptually, the *couplingDegree* can be seen as the number of relations to other entities. To know which entities are related can help to gain a better understanding of the overall structure of the system.

The *computeLabelValue* function calculates a quality label based on the amount of bugs that affect the software entity. This label is mainly needed for the further assessment and is not covered in detail in this thesis. However, since the simulation is aimed to assess software quality, this presents an important part of our work, although more simple assessments like the raw number of bugs, are imaginable.

The relation of entities to bugs is reasonable as during development different bugs occur which are visible in the corresponding entity it affects. Software entities are defined in the environment and created by the developers through the according method. Entities also belong to a category (see Section 4.2)

Bug

The third main component of the model are bugs. Software evolution research often deals with questions concerning the introduction of bugs [91], assignment of bugs [90], or the prediction of bugs [85]. Hence, it is an important topic in software evolution research, especially with regard to software quality. Bugs are introduced by the developers and assigned to the entities. As well, understanding the life cycle of bugs [93] [118] can help to predict the needed effort to fix it or to understand bug fixing patterns. We simplified the lifespan of bugs by considering the *dateOfCreation* for the simulation round it was introduced and the *dateOfClosing* when it was solved. From this, one can calculate the lifespan as the difference with the *computeLifespan* function. Of course, the actual lifespan of bugs is not always simply like that, because bugs can be re-opened or forgotten to be marked. This is taken into account in the parameter estimation.

Usually, a reported bug has a given severity indicating how fatal that issues is. Common severities are blocker, critical, major, normal, minor, or trivial,(e.g. [92]). In contrast to the priority of bugs, the severity gives no direct time constraint. We merge bug types that behave similar into three groups: *major*, *normal*, and *minor*. Major categorize all reported bugs that are major, critical, or blocker, minor bugs include minor and trivial bugs, and normal the ones tagged as normal. For the simulation, the probabilities of the different types to occur depend on the issue reports in the ITS. In reality, also other factors like the experience of developers or the project type influence their occurrences.

Category

Every software entity can be assigned to a category at the moment of its creation. The category constitutes a conjunction to which an entity semantically belongs. The category selection is based on the structure of the relationships between software entities, i.e. entities with a strong coupling belong to the same category. The investigation of relationships between software entities, e.g., as software graphs, can in general facilitate software development and maintenance [3]. Structural changes can be an indicator of important incidents in software development. This also includes the evolution of interdependencies, e.g., community detection. Communities can be defined as "...groups of vertices which probably share common properties and play similar roles within the graph [119]". In a software graph, this can be, e.g., all entities related to the Graphical User Interface (GUI) or test files. The communities in our approach are retrieved from mining and than treated as category. This means that all highly coupled software entities form a category. The coupling is the amount of common changes.

Environment

The environment is in control of the whole agent space. It creates the agents and coordinates their behavioral processes. In the simulation of software processes, this means that the environment coordinates the developers in their work and counts global variables, here the number of files (*fileCount*).

4.2.1. Topology Design: Networks

In this section, we introduce the different networks that are used as topology to model relationships between the defined agents.

Before going into detail with the different networks used, we define general terms to describe networks formally. Usually, graph structures are used to solve problems which can naturally be represented as graphs, i.e., can be seen as a set of nodes and edges representing interactions. Network theory provides methods and algorithms to analyze resulting graph structures and solve such problems. For a basic definition of a graph [120], let G be a graph and $V(G)$ a finite set of elements containing the vertices or nodes of G. The second set needed to fully define a graph G is $E(G)$. It represents the set of edges where $e = \{x, y\}$ denotes an edge between vertex x and vertex y. A weight $w(e)$ can be assigned to an edge e, indicating, e.g., distances, costs, number of interactions. Moreover, graphs can be directed or undirected indicating the orientation of the edge, e.g., an inheritance graph is directed since the inheritance relation is directed. In this thesis, we primarily deal with weighted, undirected graphs.

Developer-Entity-Network

The developer-entity-network presents all relations between the developers and software entities both as nodes in the network. Every time, a developer performs one of the defined changes on a software entity an edge is created between developer and entity if not present before, otherwise the weight of the edge is increased with every further touch. This networks provides the direct information on the owner of a software entity, the number of authors, and the number of changes. In our example in Figure 4.3 (a), developer *dev 1* changed *file A* and *file B* one time, whereas *dev 2* worked three times on *file B*.

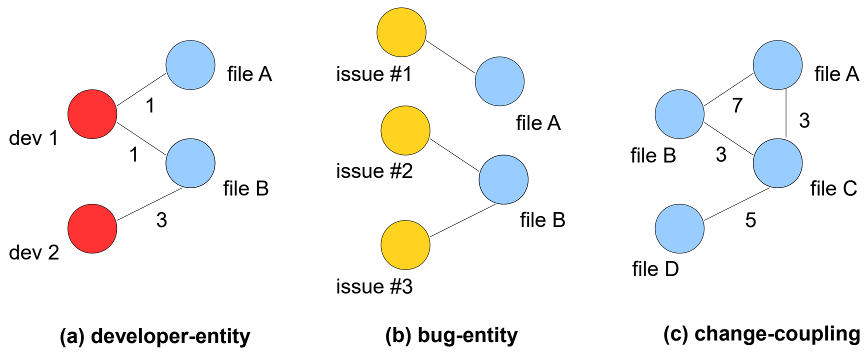


Figure 4.3.: Different networks used as ABMS topology.

Bug-Entity-Network

In this network, the links between software entities and assigned bugs are stored. When the environment creates a bug, this is created as node and randomly linked to an software entity node. This network contains information about the status of the bug, i.e., whether it is fixed or not. Following the example in Figure 4.3 (b), *issue#1* is assigned *file A* and *issue#2* and *issue#3* affect *file B*.

Entity-Network (Change-Coupling-Network)

To manage dependencies between software entities, we introduce the change-coupling-network. This is created based on the changes performed by the developers. Every time a developer touches two files in the same commit, an edge is created between the two files. For every further common change the weight of the edge is increased by one. This network represents a semantic description of the software. In Figure 4.3 (c) an example is illustrated, where four files $A-D$ and their amount of co-changes (weight of the edges) are shown. For example, file A has been changed three times together with file C and seven times together with file B.

4.2.2. Behavior Design: Strategies

In the following, we describe the behavior design of the different agents which is expressed by defined strategies. The core and most sophisticated part of the model is the formalization of developers behavior since it is responsible for the whole evolutionary process.

Commit Strategy

The commits performed by the developers let the system grow and can include a bug fix. In our STEPS model, the commit probability solely depends on the type of the developer (core, maintainer, major, minor). Based on the role, it is determined whether the developer works in a simulation round (day). For the actual work, i.e., file changes, the STEPS model makes use of a geometric distribution to draw the number of files to be added, updated, and deleted. Each of the actions follow a different distribution. Generally, a geometric distribution models failures and successes of independent trials. More precisely, the number of failures before the first success is modeled (see, e.g., [121]). We assume that developers perform actions based on their necessity, therefore, the actions can be seen as failure, since an adaption is needed. The other way around, if the system is in a satisfying state, this can be interpreted as a success, since no work needs to be done. Thus, we have $P(action) = (1-p)^k p$ with p the probability for the file creations, deletions, and updates and k the trial. Moreover, the probabilities of creations and deletions decreases with increasing system growth. In addition, the growth is restricted by the expected size of the software system.

File Selection Strategy

In this model, the file selection is modeled rather simply, since it is difficult to retrieve information about the intention of developers from mining OSS projects. The intention determining the files that need to be adapted, can be, e.g., the addition of functionality, maintaining a certain part of the software, or debugging. A more precise investigation of particular intentions, especially refactorings, can be found in [122].

For the STEPS model, the first software entity to work on is selected randomly, but we assume that the probability of choosing entities already known by the developer is slightly higher. This means that the developer has touched the file in a past simulation round. If the commit includes more than one file, the other files are selected based on information on the first one: the owner of that file, the category, and former changed entities (of the category).

Bug Fixing Strategy

As stated above, commits can also be targeted to fix a bug. In the STEPS simulation model, this is a special commit. The probability for this kind of commit depends on the role of the developer, e.g., a maintainer more likely intends to perform a bug fix commit. Also, the experience on the file is taken into account. Like all simulation parameters, these probabilities are estimated based on the mining of real software projects. For a bug fix commit, the developer selects an entity containing a bug and up to five connected entities. This value traces back to the work of Hattori and Lanza [114]. They found out that corrective maintenance activities like bug fixes often refer to tiny commits with one to five entities. Overall, they identified about 80% of all commits to be small commits.

4.3. DEVCON Simulation Model

The DEVCON (DEVeloper CONtribution) simulation model is an extension of the STEPS simulation model. Both models share the topology design as well as the strategies. But, the DEVCON model allows for dynamic project participation, e.g., developers can contribute to a changing extent in the amount of work, i.e., in the number of commits and fixes. They are also able to contribute to the project by communication activities which are considered to be important for the project involvement of individual developers. For example, a developers code contribution may be low due to time restrictions or other duties/interests, but she may also be active in answering questions concerning pieces of software on which she possesses knowledge.

Since the DEVCON model represents a refinement of the essential STEPS model, we focus on motivating the design decisions that distinguish the STEPS model from the DEVCON model. The refined model is shown in Figure 4.4. The main differences lie in the refinement of the developer roles, the impact on their behavior, and the incorporation of the new *state* class. Moreover, DEVCON summarizes maintainer and core developers into core developer, because maintainers are not present in every software project, but the instantiation of the DEVCON model requires several representatives form the different classes.

The activities of developers directly impact the evolution of the other software entities. How and how often the different types of developers change the software controls, e.g., the occurrences of bugs, the software system growth, and the relations among files. By taking the average behavior of developers, the general trends can be reproduced, but it may lack of phases with less or particularly high project activity. Thus, we expect to add the ability of reflecting such phenomena like an unsteadiness in the growth of the system and the work of developers by introducing dynamic developer phases to the simulation model. In doing so, the types of developers remain constant, but their behavior is adjusted according to state probabilistic model realized by a HMM. It comprises probabilities to switch between low, medium, and high involvement determined by the *transitionMatrix* for each developer type. Additionally, it models the workload by a probability distribution (*emissions*) that is responsible for the amount of contribution activities for the developers in the different states. In the following, we describe the introduced model entity *state* and the impact on the model as a whole.

State

Every developer occupies a state to every point in time (simulation round). We distinguish between three different states: *low*, *medium*, and *high*. Since the developers can only change their role every month, the check whether a change takes place or not is executed every 30^{th} simulation round. The period of one month is chosen because we want to summarize and combine developer activities and, therefore, some space for possible contributions is required. The probabilities of such changes are defined in the transition probabilities de-

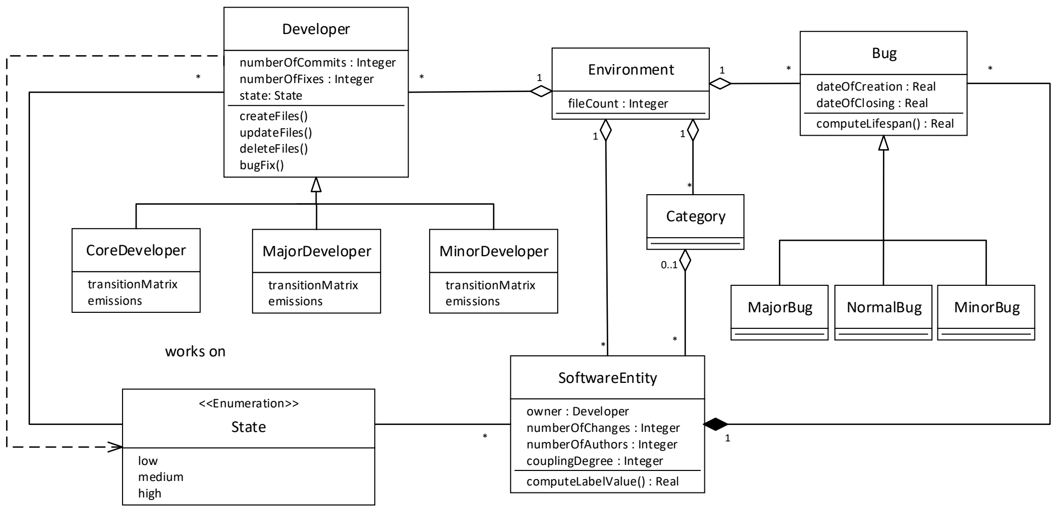


Figure 4.4.: Agent-Based Simulation Model for Software Evolution Including Developer Involvement states.

scribed in Section 2.5. The state determines the amount of work spent by the developer in the next month. For every state and developer role, a normal distribution constituting the emission probabilities schedules the commits and bugfixes. This way, the agents are able to vary their involvement over the time, which is, especially in OSS projects, a major factor.

4.4. Software Quality Assessment

The overall quality assessment of the software system under development is designed two-layered. The first layer contains the bug occurrences. Therefore, one indicator of software quality is the total amount of bugs in the system. Respecting the introduced change-coupling-networks, it is also possible to identify problematic regions in the software, i.e., areas of connected entities with a lot of bugs assigned. For the second layer, we label every entity according to its assigned bugs. Since we consider different severities, we determine different factors that emerge to the quality label of the entity. These factors state how much the different bug types affect the quality. All factors for bugs assigned to the entity multiplied produce the label for the entity. Any further assignment is again multiplied to the current value. We consider the following bug factors: 0.825 for major bugs, 0.9 for normal bugs, and 0.98 for minor bugs. Created entities start with a value of 1. We also preliminary classify entities with a label higher than 0.8 as acceptable and otherwise as problematic. Thus, an entity containing one major and two minor bugs is labeled with $0.825 \cdot 0.8 \cdot 0.8 = 0.528$ and, hence, a problematic one.

4.5. Challenges and Limitations

The introduced model is designed to answer general questions concerning software evolution. Although the model is capable of forecasting general future trends for software projects, there is space for further refinements of some model assumptions that could help to improve the model and its accuracy. In the STEPS model, the developers under simulation spend nearly the same effort in every simulation round depending on their role which can display the average commit activity of developers but lacks of a deeper description of their work including active and inactive phases as well as role changes. This is incorporated in the DEVCON model. In addition, according to their behavior design described in Section 4.2.2, the agents do not choose the files according to their intent, e.g., to choose all files affected by a bug when fixing it. For this, the underlying aim that belongs to the activities visible in their daily work, is of interest, but not that easy to retrieve.

The STEPS simulation model also leaves different project phases like development and maintenance out. Instead, the corresponding activities are distributed according to the developer roles, but this cannot reflect the occurrences of such phases completely.

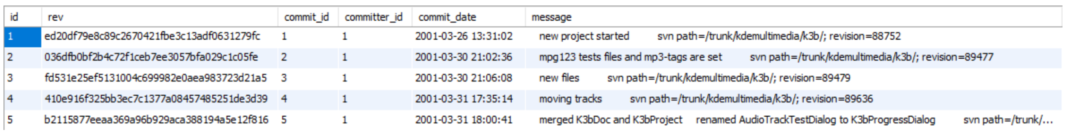
5. Instantiation of Simulation Models

In this chapter, we describe how we instantiate the models introduced in Chapter 4. We start by stating the general process of parameter estimation for simulation models. Afterwards, an overview of the data collection and preparation process is given. Following our examples, the approaches for the instantiation of the STEPS simulation model and the DEVCON simulation model are explained in detail.

5.1. Parameter Estimation

For modeling software evolution scenarios and to support software project managers in their decision making, we are interested in detecting rules and patterns describing the evolution of software projects that can serve as input for our models. We aim to simulate real projects and, therefore, make use of open source projects for the analyses. The main idea is to parameterize the simulation models with information retrieved from a set of open source projects, but we also test if we can forecast the evolution of projects using the parameters retrieved from software mining for one particular software project. The starting point of our analyses are the developers, since the evolution of the software and its artifacts strongly depend on their behavior. This establishes the connection to ABMS. Agents own an individual behavior retrieved from real world observations [123]. For the description of the behavior, insights on the work of developers, their collaboration, and communication is important. Besides, underlying factors like intentions and strategies are valuable, but more difficult to grasp.

Needed information can be retrieved from VCSs, ITSs and MLs by mining. In addition, the data has to be preprocessed to function as input for the analyses. This process is explained in the next section. To describe a certain behavior in the simulation, we use observed patterns from the mining to fill the model with knowledge. To get the data ready for analysis, it has to be preprocessed tailored towards the purpose. This step reaches from reformatting to natural language processing. A miscellaneous toolbox is available for exploration of the prepared data depending on the focus of the analysis, e.g., Weka [124] for machine learning applications. For the modeling part, we use Repast Symphony [50], an open source ABMS platform. When the adjustment of the parameters in the simulation model is done, the scenario under investigation can be run according the retrieved parameter set. By running the simulation, it can be checked whether the results produced are reasonable. It can be the case that the model has to be adapted to get homogeneous results.



id	rev	commit_id	committer_id	commit_date	message
1	ed20df79e8c89c2670421fbe3c13adf0631279fc	1	1	2001-03-26 13:31:02	new project started svn path=/trunk/kdemultimedia/k3b/; revision=88752
2	036dfb0bf2b4c72f1ceb7ee3057bfa029c1c05fe	2	1	2001-03-30 21:02:36	mpg123 tests files and mp3-tags are set svn path=/trunk/kdemultimedia/k3b/; revision=89477
3	fd531e25ef5131004c699982e0aea983723d21a5	3	1	2001-03-30 21:06:08	new files svn path=/trunk/kdemultimedia/k3b/; revision=89479
4	410e916f325bb3ec7c1377a08457485251de3d39	4	1	2001-03-31 17:35:14	moving tracks svn path=/trunk/kdemultimedia/k3b/; revision=89636
5	b2115877eeaa369a96b929aca388194a5e12f816	5	1	2001-03-31 18:00:41	merged K3bDoc and K3bProject renamed AudioTrackTestDialog to K3bProgressDialog svn path=/trunk/...

Figure 5.1.: VCS data excerpt stored in the MySQL database.

We evaluate our models by comparing empirical with simulated data which is one of current procedures in validating software simulation models [125].

5.2. Data Collection and Preparation

Although there are different simulation models aimed to answer different research questions regarding software evolution, the starting point of our investigation is the same: we populate a release history database [126] with data from GitHub using the tool CVSAnalY [127] for the raw data extraction and store the information into a MySQL database. The tool can handle SVN and git repositories. The repository comprises the whole history of a software project including the timestamps, developer, the action (add, delete, update) on selected files, commit message, and revision hash for each commit as depicted in Figure 5.1. In addition, we extended this with bug information retrieved from the ITS. This means, that additional bugfix information on commits is provided, i.e., if the commit implies a fix. For this, we use the method introduced by Philip Makedonski [96].

With this as a basis, the data has to be preprocessed tailored towards the purpose. Based on the mining process and the structure of the data, it can occur that commits are parsed twice. Thus, first, we remove duplicate entries in the database. One common problem when analyzing commits, is the merging of identities. Often, developers use different mailing addresses or login names. Especially, for the matching across different platforms, this a challenging task. Several tools and approaches exist to overcome this problem [28]. In our approach, we use regular expressions for the matching process. These expression include different combinations of the surname and the last name of authors, e.g., for *Alex Wild* also *awild* and *alexw* is checked. Afterwards, the results are checked manually.

Another common problem is the missing link between introduced bugs and affected files [19]. Often, this is solved by searching for the issue IDs in the commit message. Still, this methods depend, e.g., on logging conventions, and the possibility to miss commits is high. The other way around, it is possible to track responsible commits for an occurring bug that is reported [128]. We address this by estimating the bug population through mining and building heuristics on the emergence of bugs, e.g., that often changed modules are more likely to be infected, or that developers who are not that experienced are more likely to introduce new bugs.

Since the simulation aims to predict the overall quality of the software, the bug population is fundamental for the approach. From the ITS, we can observe when an issue is reported and when it is resolved. Thus, the lifespan of bugs can be easily calculated. We consider the bug resolved when it is closed the last time, i.e., it can be reopened in the meantime. Additionally, the ITS provides information on the severity of the bug. In Bugzilla, different severities can occur which we summarize into minor, normal, and major bugs following the method introduced in Section 4.2. The mined frequencies of the different types serve as input for our simulation models.

5.3. Instantiation of the STEPS Simulation Model

In this section, we explain our ideas to fill the introduced STEPS simulation model with knowledge. Since the model aims to answer basic questions regarding software evolution, we concentrate on basic relationships and behavior which can be reflected within a simulation. As the only active agents, the developers present a prominent role in the simulation. Thus, the mining of their behavior is the main part of this work. Closely connected to their work are the different network evolutions. The developers are directly responsible for all structural changes in the networks such as commits and bugs introduced. Thus, developer types and software networks are discussed in more detail now.

5.3.1. Developer Types

We already stressed the importance of respecting developer roles in Section 2.4. In our study, the classification is made by evaluating commit and bugfix heuristics. For this, we tested several boundary values for the separation of developer into the three types: core, major, and minor. From the commit distribution gained by the mining process, the following boundaries proved to be applicable: over 30% of all commits for core developers and over 2% of the commits for major developers, the rest is classified into minor developers. In addition, we defined the role maintainer, who is active in coding as well in maintenance work such as resolving bugs. Formally, a maintainer is a core developer or a major developer with more than 15% of own commits as fixes. This role is quite specific and cannot be detected in every project. This is due to the distribution of the work in the project, especially in the open source context.

Another approach to classify developers into types is the popular onion model [100]. The onion model assumes that 80% of the work is done by only 20% of the developers. Thus, we calculated the 80% percentile of the number of commits made by each developer. To illustrate the impact of the developer classification, we provide the following example: The project K3b has 125 active developers in the observed time period. Using our described approach, we identify one core, four major, and 120 minor developers. With the onion model, we get 99 peripheral and 26 core developers.

Figure 5.2 shows all involved developers and their number of commits. The color indicates the role identified. For the onion model, all developers below the 80% percentile

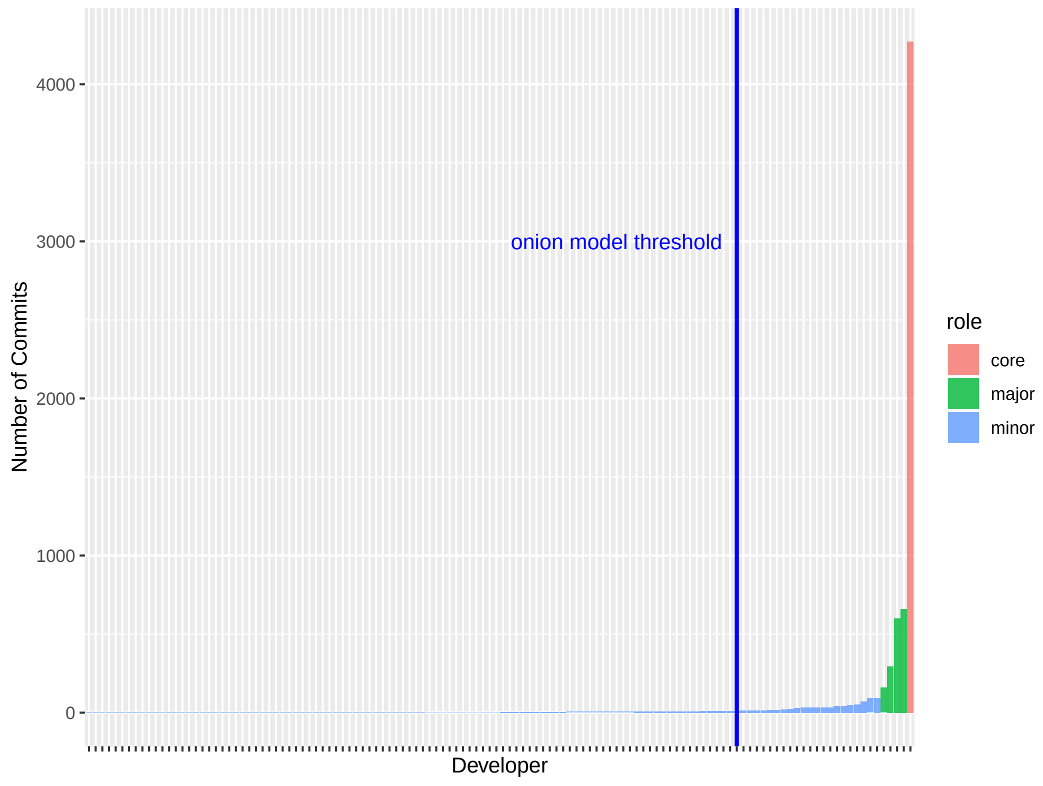


Figure 5.2.: Developer role classification own approach vs. onion model .

threshold are classified into peripheral and all developers with an equal or higher commit rate are classified into core developers. Because of the huge amount of tiny developers with only few commits this threshold is quite low with 13 commits. The blue line in Figure 5.2 indicates this threshold dividing the developers into core (right side of the line) and peripheral (left side of the line). Thus, for projects with many tiny contributors visible in the long tail in Figure 5.2, the splitting is not suitable for the pursued approach.

The work the developers perform strongly depends on their role. Following the commit strategy explained in Section 4.2.2, for each type the probabilities of file adds, updates, and deletes have to be determined. This way, the geometric distribution depicting the project growth can be specified. Besides, we use heuristics about bugfix probabilities of the different types. For this, we use the bugfix label introduced in Section 5.2 and identify the amount bug fixing commits in relation to other commits.

5.3.2. Software Networks

In order to represent dependencies between the model entities, we use the networks introduced in Section 4.2.1: the developer-entity-network, bug-entity-network, and change-coupling network. For all three networks the knowledge can be retrieved from the VCS and the ITS. We are especially interested in the characteristics, as well as the evolution of these networks. This can shed light on software evolution phenomena like developer turnover [59], collaboration [129], and failure prediction [83]. The understanding of such observations is a main driver in our research. The graph exploration tool Gephi [130] facilitates this part of our approach. Beneficially, the tool also provides several network metrics. In the following, we list the network metrics which are relevant for our work. We use the definitions introduced in Chapter 4.2.1.

- *node degree*: Let $E_x(G) = \{(x,y) \mid y \in V(G)\}$ be the subset of edges tied to node $x \in V(G)$, then the degree of v is defined as $deg\ x = |E_x(G)|$. Given this definition, we are able to calculate the amount of dependencies of every node. The degree can be interpreted as measurement for the involvement of the node in the network [131].
- *weighted node degree*: To take also the intensity of the ties into account, we consider the weighted node degree, which sums up all weights $w(e)$ of edges tied with the node. Formally, the weighted node degree can be calculated by $deg_{weighted}x = \sum_{e \in E_x(G)} w(e)$.
- *network modularity*: The modularity *mod(G)* of a graph approximates how good the graph can be divided into highly connected areas, i.e., communities.
- *network diameter*: Another structural property is the diameter *dia(G)*, which is defined as the maximum shortest path between a pair of nodes. This can be seen as an indicator on the size as well as on the density of the network.

Real-world networks exhibit a characteristic structure [119], from which the term *community* arises. Communities can also be called *clusters*. The detection of those clusters is a wide-spread task in graph analyses, especially in social network theory, e.g., identifying groups and finding important persons within these groups. There exist some algorithms to find communities in graphs [132]. Gephi uses a rather simple and fast method proposed by Blondel et al. [133]. The algorithm tests the effects on the modularity for all possibilities of nodes moving from the current to a neighbor community. Based on this knowledge, a new network is built using the optimized community structure. In doing so, the network can be divided into clusters using the modularity. In the change coupling graph, clusters can represent semantically connected entities, e.g. GUI elements. Considering our developer networks, clusters could be a good indicator of working groups. The (weighted) node degree can reveal insights on how active developers are and how strong they are involved in the project. In case of the software entities in the change coupling graphs, important modules can be detected. Besides, the modularity and the diameter helps us to understand the structure of these graphs. For the maintainability of a software a low diameter is better, because the distances are shorter [3]. Evaluating the interplay of nodes, we consider the *average node degree*, which provides a measurement for the granularity of the whole network. Thus, highly connected graphs achieve a higher average degree. The same holds for the *average weighted degree* providing an additional information on the magnitude.

For the simulation of software quality, we are interested in the relationship between structural changes and events like an increase of bugs introduced or changes in the team constellation. Moreover, the effects of bug introducing and fixing on neighbor nodes, i.e., files, are relevant for this part of our research.

For the concrete instantiation of the change coupling graph, we calculate the modularity-based cluster size of the network. The network evolves based on this value and the general derived evolution of these networks given the average degree distribution.

5.4. Instantiation of the DEVCON Simulation Model

Our second simulation model additionally allows for a variable activity of developers and a more fine-grained description of their work in general. Since we also wanted to include communication among developers, we incorporated mailing lists into our database. For this, we use the *MailboxMiner* by Bettenburg et al. [20]. Thereby, we have collected all information that we need for the analysis of developer contribution on the code level as well as on a social interaction level.

Figure 5.3 visualizes our data processing cycle. Boxes represent processing steps while ellipses indicate the output of the steps. First, the required data needs to be extracted and collected for retrieving combined observation sequences (Section 5.4.1). Then, these observation sequences are classified into low, medium, and high developer involvement using different classifiers (see Section 5.4.2). After using these sets for the HMM training of individual developer models, it is finally possible to predict the most likely sequence of involvement states that has produced the input observation sequence given the trained model (Section 5.4.3). The trained models can be applied separately for prediction as well as integrated in other applications like software process simulation as described in Section 5.4.4.

5.4.1. Mining of Developer Contribution Behavior

To derive a meaningful picture of developers' contribution behavior we combine coding activity in the number of monthly commits and bug fixes (from the VCS) with communication activities visible in the number of bug comments to the ITS and posts in the ML. Communication behavior is a significant part of the daily work of developers, e.g., conferring about how a bug should be fixed or the time plan and responsibilities for the next release. We decide to use a state-based probabilistic model because we are not only interested in the output produced by developers, but also in the underlying process that leads to the activities which can be seen as states of project involvement controlling their work. Characterizing

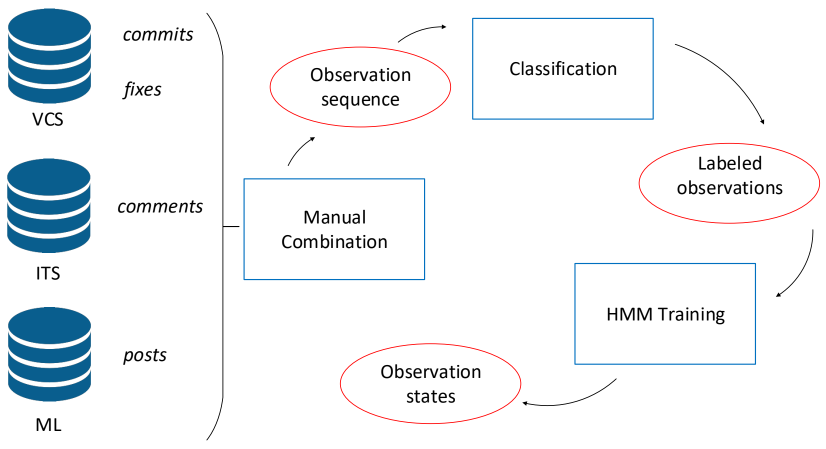


Figure 5.3.: Data collection and processing for the learning of observation states (adapted from [12]).

developer behavior this way, can lead to a new method to summarize and predict developer contribution behavior.

Using the information retrieved from the described data sources, we get a sequence of four dimensional observations $X = x_1, \ldots, x_n$ over the project duration in month n, where the null vector is permissible. For example $x_i = (20, 3, 5, 11)$ for 20 commits, 3 bug fixes, 5 bug comments, and 11 ML posts performed by a developer in month $i \in \{1, \ldots, n\}$. For handling multi dimensional observation spaces, it is convenient to reduce the size by mapping the observations [134]. We use a classifier to solve this problem, that divides the observations into *low*, *medium*, and *high* representatives. Then the contribution data is ready for the actual training done by the Baum-Welch algorithm. With the trained model, we predict the most likely sequence of observation states, i.e., developer involvement states.

Since we target to make statements that are true for groups of developers, we classify the developers into roles based on their commit behavior. This step does not influence the individual models, but it is valuable for drawing conclusions and comparing results as well as for the construction of general contribution models.

5.4.2. Classification

In order to train a HMM, we need labeled data for estimating the model parameters. Unfortunately, neither a corpus of labeled developer involvement data nor an already existing algorithm or criteria for how to create such labels exist. We create an overall involvement based on three different sub-types of project involvement: coding involvement, mailing list involvement, and issue tracking involvement. We consider three states of involvement: low involvement, medium involvement and high involvement. This means that a developer contributes to a small, medium, or high amount to the project at a point in time with respect to his single activities. Within this work, we use a semi-automated three step approach to create the overall involvement from observed developer data. Step one is to create a manual labeling for each sub-type, step two is to create a model for automated labeling of the remainder of the data, and step three combines the three sub-involvement types, one for each repository, into the overall involvement.

Manual labeling

Since there is no labeled data for developer contributions available, our first step is to create a manual labeling of developer contributions. We require nine sets of manually labeled data: one for each combination of the developer roles (core, major, minor) and the sub-types of involvement (coding, mailing lists, issue tracking). For each set, we manually label twenty monthly developer contributions. The manual labeling is expert-based and considers the output of developers. For coding, the experts use the number of commits and number of bugfixes as features, for mailing lists the number of ML posts, and for issue tracking the number of ITS posts.

Machine learning for classification

The manually labeled data is the foundation for step two, i.e., the automation of the developer contribution labeling through machine learning. We consider three machine learning algorithms: a threshold learner, the *k*-Nearest Neighbor (KNN) algorithm, and random forests.

The threshold learning approach was proposed by [135] for the determination of optimal thresholds for software metrics. We selected this approach due to three reasons [12]. The first reason is interpretability for practitioners. Thresholds can be interpreted without any knowledge of machine learning or the trained model. This kind of understandability is important for the acceptance of models by practitioners. Moreover, this facilitates the interpretation by experts to gain further insights. The second reason is the simplicity of the approach. If such a simple approach already yields a good model, there is no reason to consider more complex models, e.g., support vector machines. The third reason is the that the intuition behind thresholds matches well with how involvement can be estimated. For example, the number of commits a developer performs is a logical estimator for the coding activity.

The drawback of the threshold learning approach is that [135] proposed a rectangle based learning algorithm. Basically, Herbold et al. suggest to learn an axis aligned rectangle that separates two classes and use the lower bound of the rectangle as threshold. This kind of algorithm can only deal with two-class problems, whereas we consider a three class problem in this thesis (low, medium, high). To resolve this, we use the one-vs-followers approach for multi-class learning. Basically, this means that we apply the rectangle learning twice: 1) for learning a threshold that separates the low involvement from the medium and high involvement; and 2) for learning a threshold that separates the low and medium involvement from the high involvement. Using both thresholds together, we get can separate the contribution behavior into the three classes.

As comparison to the threshold approach, we use the KNN algorithm. We use three different values for the neighbor hood size: $k \in \{1,3,5\}$. Our reason for using KNN is that the algorithm is almost as simple as the threshold approach. Especially with $k = 1$, we simply select the most similar developer we can find and assign the same label. Moreover, the intuition of selecting the most similar developer to determine the involvement also makes sense from an expert-oriented perspective. Finally, KNN handles the multi-class problem naturally and does not require a workaround like the one-vs-follower approach [136]. With $k = 3$ and $k = 5$ the KNN loses the advantage of the easy interpretability, as the label becomes a mean value over a larger neighborhood. However, the general quality of the label should improve due to this. Hence, the additional values for k are used to estimate the impact of using larger neighborhoods.

Finally, we use random forests. The reasons for this is that random forest [137] are one of the most powerful approaches for classification problems [138]. Our rational for using random forests is that we want to compare the two simple algorithms based on thresholds and KNN to one powerful machine learning algorithm. This way, we want to determine if the simple approaches suffice, or if a more powerful and less interpretable classification technique is required. We selected random forest over other powerful approaches like, e.g., support vector machines [139], because they handle the multi-class problem naturally.

Final label assignment

To gain the overall label of a four dimensional observation, we follow the approach depicted in Figure 5.4. Different contribution activities are classified one by one into low, medium, or high, where commits and fixes together represent the code activities (VCS) besides bug activity (ITS), and ML activity using the approach discussed above. This results in three classification values for every developer and every month. After this, we assign the overall classification value using a scored majority vote over the three sub-involvement types. A low contribution on one of sub-types gives one point, medium contribution two points, and high contribution three points. If the overall score is less than 5, the overall involvement is

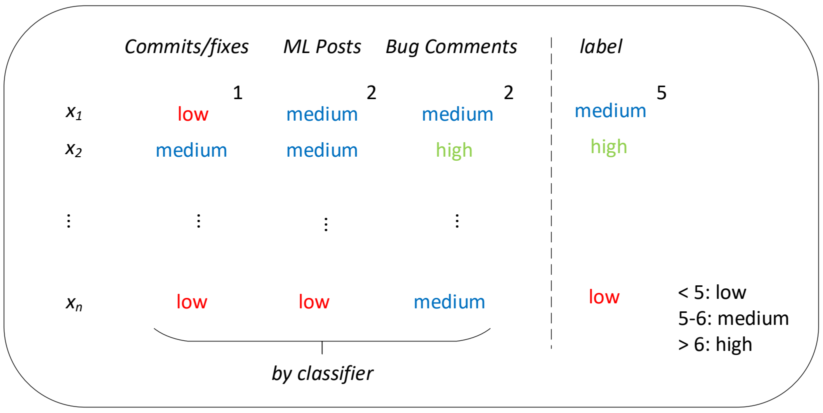


Figure 5.4.: Classification of observations [12].

low, if the score is 5 or 6, the overall involvement is medium, and if the score is larger than six the overall involvement is high.

Figure 5.4 depicts the example of an observed developer with low coding involvement, medium ML involvement, and medium ITS involvement (x_1). Thus, we get one point for coding, two points each for ML and ITS involvement, i.e., five points overall. Thus, the overall involvement of the observed developers' contribution is medium.

5.4.3. HMMs for Developer Contribution Behavior

HMMs are appropriate for labeling observational sequences. The dynamics that can be observed in software development can be presented by the changeover between the different states and thus, poses a more realistic model of developers' contributions than, e.g., an average model. Since we make no additional conditional assumptions on the model, there is no need for a more complex model like Conditional Random Fields (CRFs).

The problem where HMMs are used to detect the most likely sequence of hidden states given a sequence of observations can be regarded as an unsupervised learning problem. We aim to derive the hidden structure behind unlabeled data. For this, the recursive Baum-Welch algorithm operates for the training of the HMM.

For the implementation we use the *mhsmm* package for R [140] that facilitates the parameter estimation process and provides predictions. Following the definitions given in Section 2.5, we have $Y = \{y_1, y_2, y_3\} = \{low, medium, high\}$ $(N = 3)$. The initial observation space consists of vectors $x_t \in \mathbb{R}^4, t = 1, \ldots, M$ where M is the number of distinct observations. As stated in Section 5.4.1, we reduce the multiple observation space using classification. Thereby, the observations $X = x_1, \ldots, x_n$ where n is the project duration in months, are classified into low, medium, and high involvement with 20 manually classified observations. Based on this allocation three training sets are defined (one for each state), and thus the parameters of the multivariate normal emission distributions describing the output probabilities can be derived. This is done in the HMM training for each developer under investigation individually. Also the parameters for the transition matrices A are estimated step by step. The matrices comprise the probabilities for the developer on how likely they change the states from one to another. The same way, the initial distribution π is calculated. As starting point for the estimation, averages of the observations are taken into account. For the specification of the HMM, we simulate data from the retrieved distribution and fit the individual model with this. After this step, the HMM $\lambda = (A, B, \pi)$ is completely defined. It is possible, that no HMM can be trained, e.g., if the observation sequence is too sparse. The model also respects two state models, i.e., developers never reach the highest state. In addition, also trivial models with only one state are possible, but this is very rare.

The last step involves the detection of the most likely sequence of hidden states that generates $X = x_1, ..., x_n$. For this, the Viterbi algorithm is used. For each observation and every point in time the corresponding state is calculated.

General Model

We combine the individual, developer-specific models that can be derived by building the average for each developer role: core, major, and minor. This way, we can draw generalized conclusions valid for groups of developers. Moreover, this kind of model can be used vice versa, e.g., for the prediction of developer involvement and workload in the project based on the role. One major benefit of a general, role-specific model is the handling of models where the developer-specific calculation failed, thus it is appropriate for every developer.

The creation of the model treats every model parameter of $\lambda = (A, B, \pi)$ separately. The initial distribution π as well as the transition matrices A are combined building the entity-wise average.

The emission distributions get combined with linear transformations. Let n be the number of developer models available for role $i \in \{core, major, minor\}$. Besides, let $T = diag(\frac{1}{n})$ the transformation matrix. Then, we can compute the mean $\mu_i = \sum_{m=1}^{n} T \cdot \mu_m$ and the covariances $\Sigma_i = \sum_{m=1}^{n} T \cdot \Sigma_m \cdot T^t$. This also results in a multivariate Gaussian [141]. The great advantage of the general model is the applicability for all developers. The resulting models are also practically interpretable, since they provide evidence on how developers of the same role work and communicate. For our analysis, we compare the individual with the general models and test the impact of different classifiers. All models are evaluated in our case studies. The general models are evaluated in our applications: the prediction of the activity of developers and the incorporation into a simulation model for software quality assurance.

5.4.4. Simulation of Software Processes with a Contribution Phases Model

For the DEVCON simulation model we concentrate on the new introduced states of the developer roles and how they are responsible for the contribution of one developer role. During the simulation, each developer is always in one of the following states: low, medium, or high. The state stands for the involvement of the developer and thus, the resulting workload (drawn from the normal distribution) is based on her role and involvement state. For example, a core developer in a medium state usually performs more commits than a minor developer in a high state. The state is recomputed every month, based on the transition matrix of the developer role. Since one simulation round represents one day, the computation is executed every 30^{th} round. After the state for the current month is computed, the number of commits and bug fixes, i.e., the emissions, has to be determined. This number is normal distributed with different parameters for each of the three states. The communication activity is currently not included in the simulation. The contribution of a developer is represented by the number of commits and bug fixes she performs monthly. This model is more realistic than the average commit behavior.

6. Case Studies

We designed three case studies aimed to answer our research questions and validate our approach. These case studies are described in detail in the subsequent sections. For every study, the setup, the results and a discussion of the results are reported.

6.1. Overall Design and Objectives

Generally, the case studies performed are designed to answer the research questions posed in Section 1.2 with the superordinate research questions *RQ* 1 and *RQ* 2 in the back of our mind. The studies have some steps in common that build the methodological foundation for the tended analyses. Since all our studies rely on real data, the data has to be selected, extracted, collected, and processed to be suitable for investigation. How this is done in principle, is described in Section 5.2. Of course, the data has to be treated differently according to the purpose, e.g., for the application of machine learning algorithms the data has to be in a processable format.

As a starting point, we always build a software evolution model which sets all entities needed for answering the research question in relation. Then, the parameters for the model are estimated using mining techniques and afterwards incorporated into the model. This way the software evolution model is instantiated, but may be adapted after running the simulation, if the results are unfitting. This has to be done very careful because the causes for this are versatile: an error in the implementation, a wrong model assumption, some mistake in the analysis, or some simulation side effects. Thus, verification has to take part already in the model building process as well as in both the implementation and analysis. The case study presented first instantiates the simulation model as described in Section 5.3 and aims to answer *RQ* 1.1, *RQ* 1.2, and partly *RQ* 1.3. This model reflects basic software evolution trends, e.g., commit behavior, bug occurrences, system growth, and the evolution of software networks.

The second case study is not mainly concerned with the estimation of simulation parameters, but instead it validates our approach of a dynamic developer contribution model. It reuses the main parts of the STEPS simulation model and extends it by phases of developers allowing them to switch between different degrees of project involvement. The model is instantiated like described in Section 5.4. Here, a big part of the work is of methodological nature. Overall, the case study is directed to the answer of *RQ* 2.1, *RQ* 2.2, *RQ* 2.3, and *RQ* 2.4 as well as altogether as answer to *RQ* 2.

The third case study transfers the worked out approach into the context of project activity. Basically, the study focuses on finding a model that distinguishes between active and inactive software projects. The main challenge present is to determine how much activity can still be considered as inactive. The HMM takes this problem into account and produces representatives of active and inactive projects. We also test the approach in praxis performing a crossovers study with students as participant. Overall, the case study is designed to answer *RQ* 2.5.

6.2. Case Study 1: Feasibility of Simulation of Software Evolution

Within this case study, the model introduced in Section 4.2 (Figure 4.2) is instantiated with data from a real software project and validated by comparing empirical with simulated results as well as the transfer of the model to another software project context similar in size and duration.

6.2.1. Setup

Overall, the study is aimed to investigate whether an Agent-based simulation model of software evolution yields realistic results and what size of the parameter space is sufficient. In addition, it tempts to set a balance between the parameter space and established model assumptions. For the basic initialization, we selected *K3b* [7] as reference model. This project has been chosen for various reasons: first, it has a sufficient long history with over ten years of development. Besides, the design of the model requires the behavior of different types of developers active in the project. In *K3b*, all types of developers could be identified. Since we also need bug information about the project, it was also important that an ITS in addition to the VCS is available. For validation, the project *Log4j* [8] was chosen, because we were looking for a project which has similar characteristics as *K3b*, but diverges in at least one point to test the transferability of the simulation model. In Table 6.1, the attributes of the two projects are listed. Both possess over a decade of change history and are similar in the size measured in the maximum number of files, although the amount for *Log4j* seems higher at first sight which is caused by a more unsteady growth of the system. The most prominent difference lies in number of developers involved in the project and, thus, in the distribution of work. In *K3b*, 124 developers were active, whereas for *Log4j* only 20 have been identified in the examined timespan.

For the actual instantiation, we distinguish between parameters gained by the mining process and behavior that is implemented according to our model assumptions described

[7] https://userbase.kde.org/K3b
[8] https://logging.apache.org/log4j/2.x/

Project	Years	#Developers	#Commits	Max(Files)
K3b	11	124	5605	1308
Log4j	14	20	3428	1999

Table 6.1.: Attributes of selected projects (adapted from [4]).

in Section 4.2. This includes the commit, file selection and bug fixing strategies (Section 4.2.2), as well as the computation of the quality label (Section 4.4) and the stepwise construction of the change coupling network based on the initial cluster size (Section 5.3.2).

The remaining parameters to set depend on the project properties. For this, we retrieve knowledge about the number of developers as well as their inhibited role, their probabilities of the different types of software changes, the assumed duration of the project, and the expected size in number of files. How these are retrieved and validated, is described in the following sections.

6.2.2. Evaluation Criteria

Generally, our approach is two-folded, since it uses software mining for the estimation of parameters and investigation of software evolution as well as simulation for the prediction of software evolution trends enriched with knowledge form the mining process. Thus, the validation is also two-folded, whereat the simulation results naturally depends on the goodness of the mining process.

In this case study, we perform a lot of basic calculations which do not require a sophisticated validation. Nevertheless, we cross check each retrieved value carefully. For the validation of regression models for the average growth trends including the coupling degree evolution, we use the adjusted R-squared value (e.g., [142]).

Comparable to the R-squared value, the adjusted R-squared value indicates how well the regression fits the data. The main difference of the two measures lies in the following characteristic: Whereas the regular R-squared value increases with every term added to the model. Sometimes, this can lead to overfitting. In contrast, the adjusted R-squared value takes this into account and does only increase if the model really gets better with additional terms. Thus, for higher order models it is a good choice to use the adjusted R-squared value for assessing the goodness of fit of a regression model.

Besides from testing the simulation framework itself, we validate the simulation results by comparing metrics observed in real software projects with metric values produced by the simulation which represents a common evaluation method in simulation studies [125].

6.2.3. Results

In this section, we present the results achieved within this case study. This includes a probabilistic model of software changes, a definition of developer behavior, heuristics about bug occurrences, as well as a representation of relations between software entities as networks, i.e., developer-file networks and change coupling networks.

Software changes

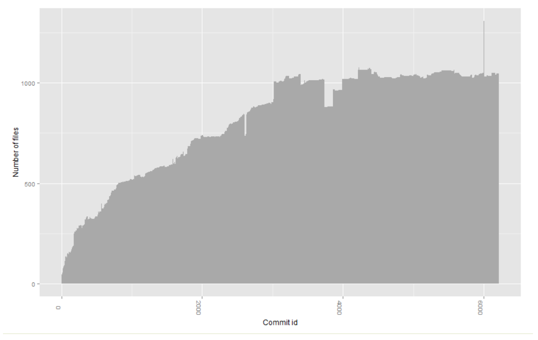
The behavior of developers relates directly to the growth of the software, since it results from the addition, deletions, and modification of files performed by the developers. Since the file growth is modeled as the geometric distribution $P(action) = P(X = k) = (1-p)^k p$ with p the probability for the file creations, deletions, and modifications and k the trial as motivated in Section 4.2.2, the first thing to derive are the probabilities p for each developer type. Here, one of the classified major developers is treated as maintainer due to the large portion of maintenance work in the project. Maintenance commits are identified using the introduced bugfix label. The application of these manually adapted heuristics led to an identification of one core developer, one maintainer, three major developers, and 120 minor developers.

As a next step, the above stated change probabilities p are calculated for populating the geometric distribution. These probabilities arise from the different types of file changes belonging to each commit, e.g, in a certain commit three files may be added, two more modified, and one deleted. Summarizing all additions, modifications, and deletions of all commits belonging to a developer type and building the average leads us to the desired probabilities. Since the mean of the used version of the geometric distribution is defined as $E(X) = \frac{1}{p}$, it directly implies $p = \frac{1}{E(X)}$. Hence, calculating the mean of the additions, deletions, and modifications, respectively, per commit leads directly to the desired probability p and, thus, to the population of the geometric distribution for each file change action.

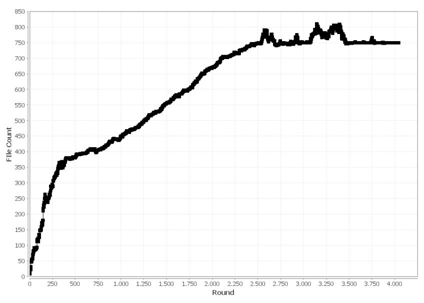
Developer	#Commits	#Fixes	Add	Update	Delete
Core	3397	874	0.6	5.5	0.4
Maintainer	509	152	0.9	3.5	0.3
Major	1353	362	0.2	5.2	0.4
Minor	346	127	0.1	2.0	0.04

Table 6.2.: Developers average commit behavior in K3b (adapted from [4]). Add, update, and delete values are averages per commit.

The results are shown in Table 6.2. There, the overall amount of commits by the different types are presented as well as the amount of comprised bugfix commits. Besides, the

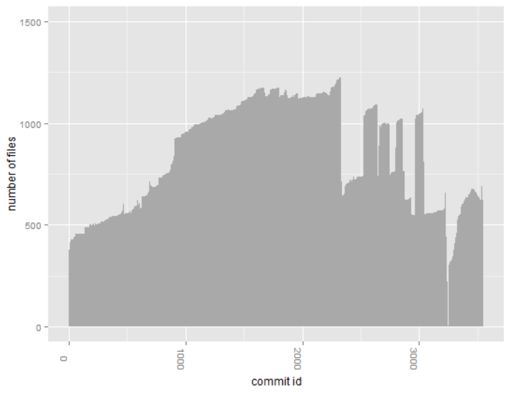


(a) Empirical

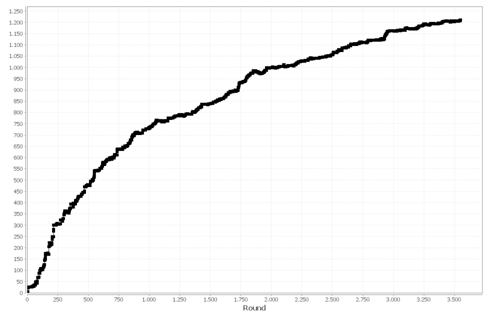


(b) Simulated

Figure 6.1.: Empirical and simulated growth of K3b [10].



(a) Empirical



(b) Simulated

Figure 6.2.: Empirical and simulated growth of Log4j.

change probabilities for the different file actions per commit are shown. Taking the average commit behavior of developers adjusted with the change probabilities, is already sufficient to perform basic simulation runs irrespective of the relationships between software entities. The work shown in the table is distributed among one the identified portion of the core developer, the maintainer, the major developers, and the minor developers.

Note that all simulation results produced for *Log4j*, origin from the instantiation of *K3b* with just a few project-specific parameters adapted, i.e., we adjust the expected size, the number of simulation rounds (days of development), as well as the team constellation (see Table 6.2). In doing so, the growth in number of files, the general effort spent in commits, and the average coupling degree could be mirrored. In the case of *Log4j*, we also have one core developer and one maintainer, but five major developers and 13 minor developers. The expected size can be estimated from the actual file growth (Figure 6.2a).

In Figure 6.1a the actual file evolution of *K3b* can be retraced which displays a sub-linear growth rate. This conforms with the predominant file growth observed in the literature [76]. To assess the closeness of the simulation to reality, we compare the empirical growth trend with the one produced by the basic simulation depicted in Figure 6.1b. Note that the scale differs slightly since the two figures are generated from different tools, i.e., the latter is a figure directly produced by the simulation tool. The comparison shows that the simulation is able to reproduce the basic growth rate in number of files. In contrast, the growth of *Log4j* is depicted in Figure 6.2a. This illustrates an example for an unsteady growth which can be due to switches of branches, e.g., for a new (stable) release, integration of external functionalities, or major refactorings. In the case of *Log4j*, imports from CVS to SVN are responsible for a big portion of the unsteadiness.

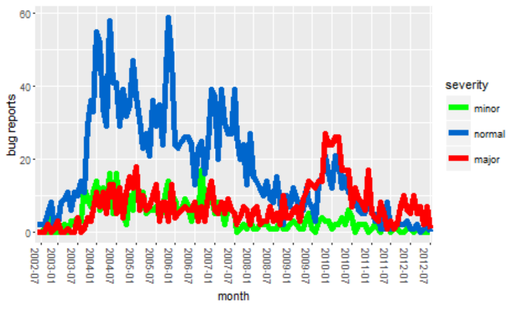
Bug occurrence

For the creation of bugs, a rather simple method is used: From the ITS, heuristics about the bug occurrence rate as well as the lifetime for bugs are retrieved. In doing so, we distinguish between major (including critical and crash) bugs, normal bugs, and minor (including wishlist) bugs.

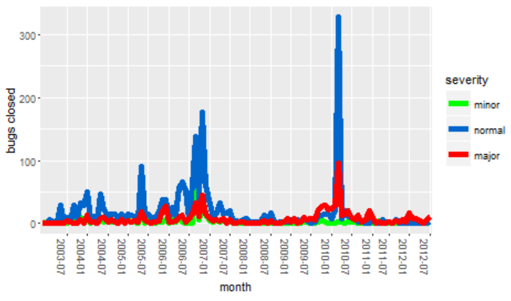
The mined rates can be viewed in Figures 6.3a and 6.3b. It is noticeable that whereas the bug reports occur steadily over the project duration, the closing rates seem to cumulate at certain points in time. We explain this phenomenon by the closeness of the high rates of bugs closed to the dates of major releases. In the beginning of 2010, *K3b* 1.0 was released and in spring 2010 *K3b* 2.0 was released both after high closing rates. This is due to the fact that a lot of bug fixing is done before a release whereas bug reports – often from users of the software project – come in every time. For *Log4j*, the bug reports and closing rates are illustrated in Figures 6.4a and 6.4b, respectively. The observed trends are similar to *K3b*: Bugs are reported steadily and often closed before a release, e.g., in May 2002 *log4j* 1.2 was released. For the purpose of simulation, we build averages on these rates and distribute the bugs among the active developers introduced when they commit. The later bug fixes are performed based on the experience and role of developers as declared in Section 4.2.2.

Software networks

As a next step, we also consider relationships between the different software entities involved expressed by networks (see Section 4.2.1). The first network we investigate reflects the work of developers on the files, i.e., the developer-entity network. For tracing the evolution of this network, we mined yearly networks mirroring the current state of collaboration. Developers, who have a short path to reach each other, collaborate to a higher degree than those whose distances are longer.

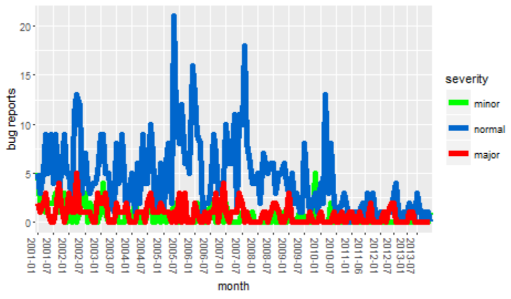


(a) Reported

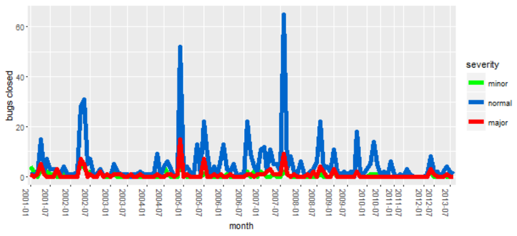


(b) Closed

Figure 6.3.: Bug report and close rates in K3b.



(a) Reported



(b) Closed

Figure 6.4.: Bug report and close rates in Log4j.

The yearly developer-file networks are shown in Figure 6.5. There, the red colored node represents the identified core developer of *K3b*. It can be seen, that the central status of this developer increases from year to year most dominating in 2006. Then, another developer (blue node on the left side) spent much work in the project and inherits the central role more and more. We were able to identify this developer as the maintainer of the project. This so-called turnover in software projects is a common observation in the evolution of OSS projects [143, 59, 144].

This observation can be supported by analyzing the evolution of the modularity value for the yearly networks which is shown in Figure 6.6. The modularity (see Section 5.3.2) gives information on the appropriateness of separation into connected entities, i.e., communities. Therefore, a low value indicates that the work is not balanced, e.g., there is one predominant developer or, on the other hand, that developers work is distributed among numerous software entities, and, hence, the project may lack specialists. Thus, from a software engineering perspective, a medium to moderately high modularity is desired as it implies the most balanced work. In Figure 6.6, it can be seen that with the predominance of the core

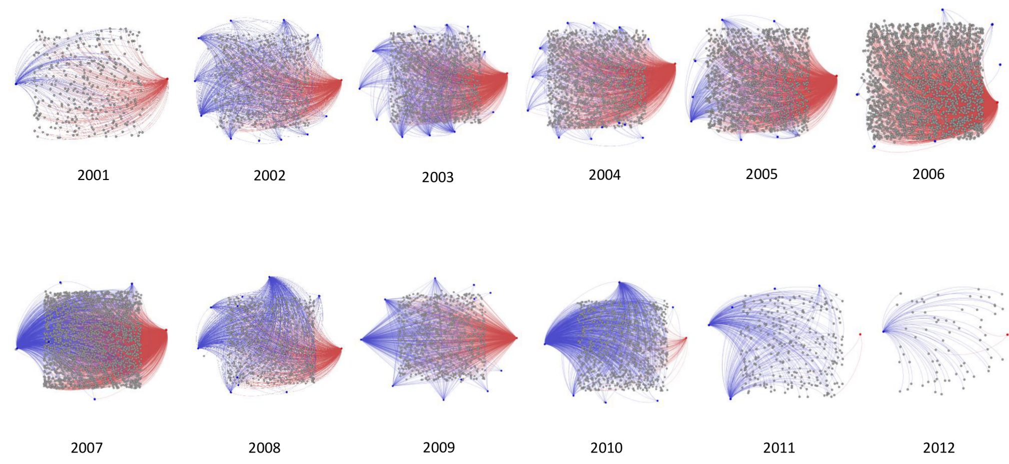


Figure 6.5.: Yearly Developer-file Networks for K3b [10].

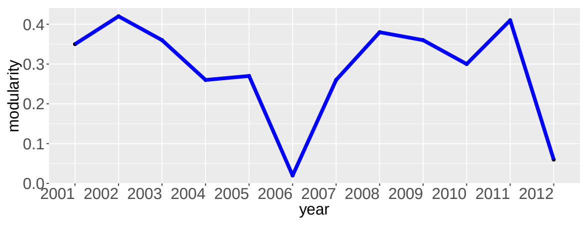


Figure 6.6.: Modularity of Developer-file Networks.

developer in 2006, we also have a very low modularity. The same holds true for the year 2012, when the maintainer takes over.

Similarly, we analyzed the evolution of the change coupling networks for modeling relationships between software entities. These graphs also grow directly from the changes recorded in the VCS. Files, that are changed together at least twice, are considered to be related and, thus linked in the network. Every new created file is introduced as node. We observe the evolution of file dependencies by building yearly change coupling networks. As a first analysis step, we visualize the yearly networks with Gephi and calculate different network metrics (see Section 5.3.2) for G the change coupling graph with $V(G)$ the files and $E(G)$ the change coupling links: $average(deg(x))$, $average(deg_{weighted}(x))$, $m(G)$, and $dia(G)$, for $x \in V(G)$.

A selection of the yearly coupling graphs is shown in Figure 6.7. Besides, the corresponding values for the modularity as well as the average degree are reported. Comparing the graphs for *K3b* and *Log4j*, it is noticeable that the general growth for *K3b* converges

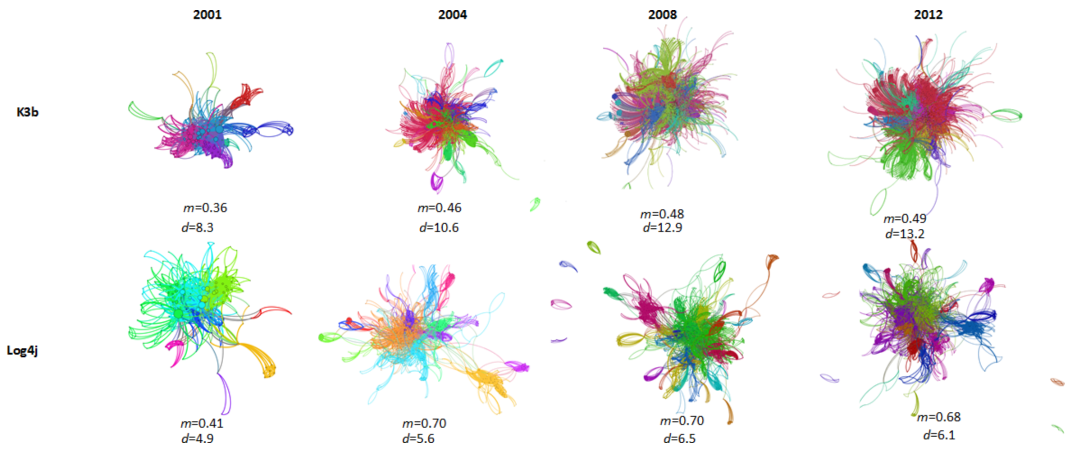


Figure 6.7.: File dependency graphs of K3b and Log4j including network modularity $m(G)$ and $d := average(deg(x))$ [4].

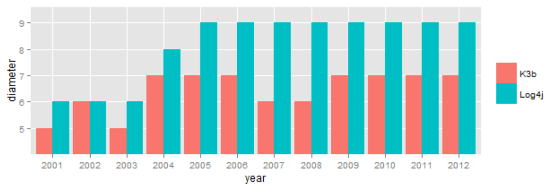
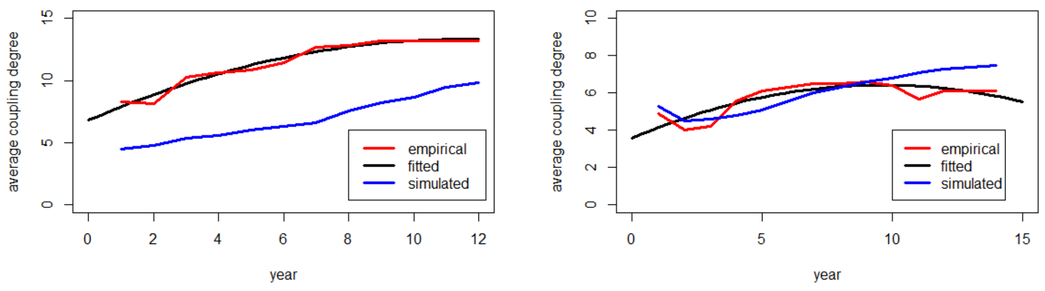


Figure 6.8.: Diameter $dia(G)$ over the years for selected projects [4].

more dense than for *Log4j*. The color indicates the community based on the modularity factor and identifies semantically connected entities, e.g., certain sets of tests or GUI elements. For both projects, the number of clusters increases faster in the beginning until (nearly) stagnating. As the average degree indicates how many relations a file has in average, our assumption is that it rises (slightly) together with the system growth. From the values in Figure 6.7 this assumptions gets confirmed apart from the change coupling network of *Log4j* in 2012, where d slightly decreases. The overall amount of dependencies is higher for *K3b* since the network evolves very dense and can not be fragmented into strong communities very well. This observation is supported by the modularity values m. In contrast, *Log4j* exhibits a clearer community structure. Although *K3b* includes a fewer number of files, it has higher values for d, because the higher amount of loosely connected files lower d in the case of *Log4j*.

Supportingly, we derive the diameter $dia(G)$ for each years' network shown in Figure 6.8. There, the values for *Log4j* are also higher than for *K3b*, indicating hat the paths from one node to another are longer.



(a) K3b (b) Log4j

Figure 6.9.: Average degree evolution for selected projects [4].

For the general evolution trend of the average number of dependencies (degree) per file, we fit a regression curve to the empirically observed curve over the years. Hereby, we also observe a sub-linear growth.

In Figure 6.9a The actual values for *K3b* are visualized as the red line in the graph whereas the fitted line is presented as the black line. The best fit is achieved using a second order model with an adjusted R-squared value of 0.97. The same analysis applied to *Log4j* results in similar trend even though the adjusted R-squared value is lower (0.67) due to the unsteady beginning and also end of the empirical coupling curve which can be retraced in Figure 6.9b.

Besides, the simulated average coupling degree is shown in Figures 6.9a and 6.9b. In the simulation, *Log4j* reaches a better approximation.

The construction of the change coupling network under simulation uses the number of communities derived by the modularity as well as the assumption that a developer works more often on files that she knows. Based on the change probabilities the network establishes strong dependencies among the communities. However, the simulation permits a more abstract view of software evolution focusing on the structural properties of the network.

6.2.4. Discussion

Predominantly, this case study is designed to investigate which aspects of software evolution can be modeled from an Agent-based perspective and, thus, produce reasonable results. In addition, possible restrictions and challenges are expected to be exposed.

To model software evolution from the starting point of agents, we take the entities that affect software evolution into account as agents in our model: software entities, bugs, and developers. Since the developers influence the other model entities directly, they present the only active agents. Hence, for a convenient model of software evolution, the work of developers as well as the impact of their work on the software entities including bug introduction has to be described. Essentially, the model which is introduced in Section 4.2 and concretely instantiated in this case study presents the answer to *RQ* 1.1.

To parametrize the identified STEPS model, we make use of software mining. In this case study, we showed the feasibility of this approach by the application to two open source software projects. In Section 5.3, we already presented the strategies to estimate these parameters which were put into practice within this case study. Together, the strategies proposed and exemplified this way can answer *RQ* 1.2, i.e., we can estimate parameters suitable for Agent-based simulation by mining software repositories.

RQ 1.3 focuses on the identification of software evolution trends that can be simulated. Since this question is stated quite broadly, an all-embracing answer to this is elusive. But, through our case studies we can definitely state what kind of trends can be mirrored by the simulation and also what is more challenging to reproduce. Within this case study, we illustrated that the following trends can be mirrored: (sub-linear) file growth, average developer contribution rates, average bug introducing rates for different kinds of bugs, and the evolution of change coupling networks. However, the work also raises some challenges in simulating software evolution.

Among these challenges is the observation that, although the project growth in number of files can be controlled by the effort spent by developers, the simulation lacks of the ability to mirror very unsteady growth trends like it is shown in Figure 6.2a.

Moreover, in the STEPS simulation model and, thus, in this case study, we use the average commit behavior of developers. In doing so, we are able to reproduce the number of commits performed. Still, it is a well-known phenomenon that developers have different levels of activity in certain points in time, especially in OSS projects. We concentrate on this challenge in detail in case study 2.

Finally, the bug assignment to affected entities in this case study is based on a randomized method which can still display the occurrences. To mitigate the randomness, possible solutions include bug introduction strategies based on certain commits [91].

6.3. Case Study 2: Dynamic Developer Contribution Model

The presented case study is designed to instantiate and evaluate the DEVCON simulation model introduced in Section 4.3. The model gets instantiated with the average dynamically adapted behavior of developers of eight software projects and evaluated in the simulation applied to a subset of six projects.

6.3.1. Setup

The main goal of the performed studies is to support our hypothesis that we can describe contribution behavior using HMMs. Moreover, we lay the foundations for our subsequent applications. In this extension of the case study presented in [12], we use eight open source software projects: *Amarok*, *Ant*, *Egit*, *K3b*, *Konsole*, *Log4j*, *Poi*, and *Rekonq*. The projects are selected based on different properties: the availability of developer information from all three data sources, the quantity of developers matching our role definition, and a minimum of three years continuous development. Table 6.3 shows the properties for the chosen software projects. It is notable, that tiny contributers are omitted for the HMM training, since for them the observation space is too small. Thus, the presented amount of minor developers can deviate.

Project	Commits	Developers $(core \mid major \mid minor)$	Duration in months
Amarok	28043	(1|13|14)	104
Konsole	5746	(1|5|9)	190
Log4j	3498	(2|4|9)	160
Ant	15220	(1|5|14)	175
Poi	6095	(1|5|10)	152
Egit	3219	(0|4|7)	59
Rekonq	2606	(1|3|6)	36
K3b	6217	(1|4|4)	126

Table 6.3.: Overview of projects.

For all of the listed developers in the table, we train a HMM individually. Due to the nature of HMMs, this is not possible if the observation sequence is too sparse. Therefore, we are interested in examining how similar the contribution models for the same developer role are. If the observed similarity is high, then the creation of general models as an average over all individual models of a certain role can be helpful. This enables the prediction of state sequences for every developer. Moreover, we are interested in the applicability of the resulting models in the context of other software projects, e.g., prediction of developers' workload, or the simulation of software phases.

We follow the process introduced in Section 5.4. The results of the different steps as well as our method to evaluate them is presented in the following. Moreover, we discuss our findings.

6.3.2. Evaluation Criteria

We aim to derive a dynamic contribution model for software developers based on HMMs. Therefore, we build individual models for each developer. To know, if the resulting models are appropriate we measure the accuracy of reproducing real contribution behavior.

We label our observation ourselves by classification as there exist no ground truth for the right label. Thus, for the measurement of the accuracy, we count the number of differences between classified instances and predicted instances by the HMM. Let S_{class_i} the state given by the classifier and S_i the HMM predicted state. Then we define:

$$mr := \frac{1}{n}\sum_i \mathbb{1}(S_i \neq S_{class_i})$$

with $i \in (1,...,n)$ the number of the project duration in months, i.e., the number of states produced, and $\mathbb{1}$ the indicator function counting the occurrence of mismatches.

For calculating the pairwise correlations between contribution models for each developer role we use Pearson's product-moment correlation [145]. With this, we also check the correlation between the different observed activities of developers, i.e., commits, bug fixes, bug comments, and mailing list posts.

The evaluation of the general model simply applied and used as stand-alone tool happens implicitly, since our developer general models are validated separately in the following section.

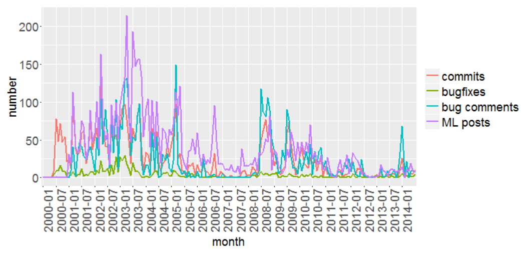
The validation and verification of agent-based simulation models is a complex task. Following [146], the validation proceeds on three different layers: the conceptual model validation, the computational model verification, and the operational validation. Data validation may be considered additionally, but the data used for model building origins from open source software repositories and this establishes adequateness for modeling software evolution phenomena naturally. The validation of the conceptual model requires ensuring that assumptions made for the model creation are correct. In this study, the conceptual model presents the developer contribution models and thus, is validated using *mr* and correlation analyses. The computational verification is done internally by testing. The most extensive step in our work is the operational validation which deals with evaluating the models output behavior. Since we are mostly interested in a software quality trend analysis [112], we compare the trends of simulated results with empirical observations to determine whether the simulation model produces sufficient results. The simulation used there has also been validated itself. We also use an average over several simulation runs for handling the stochastic variability.

6.3.3. Results

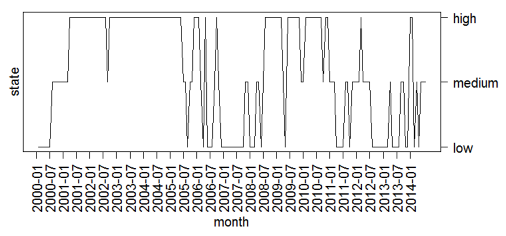
In the following we present our results for describing developers' contribution behavior with HMMs according to a given role classification. We first sketch our findings on individual models and the general applicability of HMMs for that purpose and, afterwards, generalize the results for general contribution models which can be used in practice.

Individual Developer Models

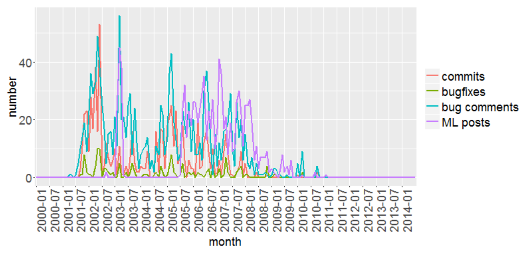
Starting with the extracted raw data consisting of the monthly commits, fixes, bug comments, and ML posts, the observations first need to be combined for every developer. This



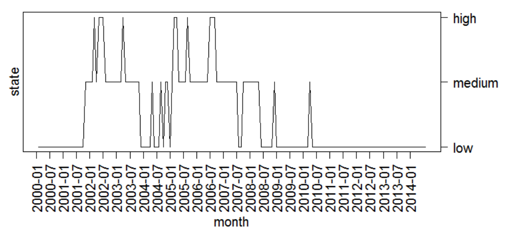
(a) Core developer



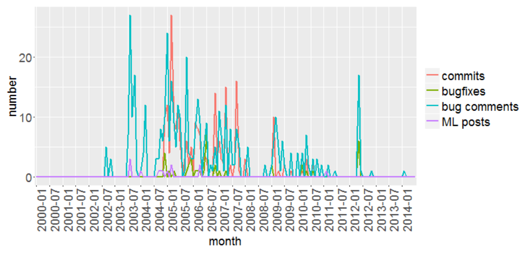
(b) Core developer predicted



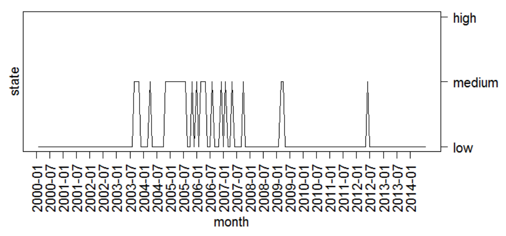
(c) Major developer



(d) Major developer predicted



(e) Minor developer



(f) Minor developer predicted

Figure 6.10.: Developer contribution patterns and resulting involvement states of Ant [12].

is done using regular expressions on the names and mail aliases followed by a manual validation and merge of remaining developers.

Naturally the observation space of developer activities is very big. Thus, we narrow it down by classifying it into low, medium, and high activity. For this, commits and fixes are considered together as commit activity, whereas bug comments are seen as bug activity, and the ML posts are considered as ML activity. For the classification, we use basically three different classifiers.

For each classifier, we use the same set of 20 manually classified observations for a project. For the threshold classification, there is the advantage that it produces not only the labels, but also the threshold, which can easily be interpreted as an indicator for the overall work and communication in the project. For an overview of the thresholds for six of the investigated projects see [12]. KNN selects the most similar developer, whereas Random Forest selects the most voted label from the decision trees learned from the input behavior. Then, an overall score is calculated for each observation as described in Section 5.4.2. All of these methods result in a labeled observation sequence for each developer.

Now, all preprocessing is done and the actual HMM training takes place. The output of the training combines the different activity layers and puts the result into one state expressing the involvement of the developer to each point in time. This state sequence is predicted using the individually trained HMM $\lambda_j = (A_j, B_j, \pi_j)$ for developer j. Developers tend to start with low involvement, i.e., the initial probability distribution converges to $\pi = (1,0,0)$. The resulting transition matrices and emission distributions are analyzed in more detail in the next section.

Developer contributions together with their predicted state sequences produced via the Viterbi algorithm are depicted in Figures 6.10a-6.10f. On the left side (Figures 6.10a, 6.10c, 6.10e) the monthly activities for one core, one major, and one minor developer of Ant are presented, whereas on the right side (Figures 6.10b, 6.10d, 6.10f) the according involvement states are shown. Figure 6.10b represents a good example of a state pattern for core developers and Figure 6.10d presents a basic pattern for major developers. Besides, Figure 6.10f shows an example of a minor developer who never reaches high involvement. For minor developers, this case occurs approximately for half of the contributors.

On the whole, we were successful in training an individual HMM for 42 of 125 developers included in our case study with the thresholder and random forest. For the KNN, we could create 41 models, with $k = 3$ we reached 44 models, and for $k = 5$, 48 contribution models were built successfully. In the other cases, the observation space was too sparse. To evaluate the goodness of fit of the derived models, we calculated the misclassification rate *mr* as defined in Section 6.3.2 that measures the distance between the classified states from the raw observation data and the predicted states using HMMs.

Figure 6.11 shows the project-wise misclassification rate of all individual HMMs with each classifier used. There we observed that, generally, all classifiers perform well with few outliers. For some projects no or only one developer model could be built (poi with thresholds, *K3b* with all classifiers). The best results were achieved for *Konsole* and *Amarok* with almost all values lower than 0.1. The average *mr* for every classifier and every project can be retraced in Table 6.4 with the best result for KNN3 having an average discrepancy of 8.8%.

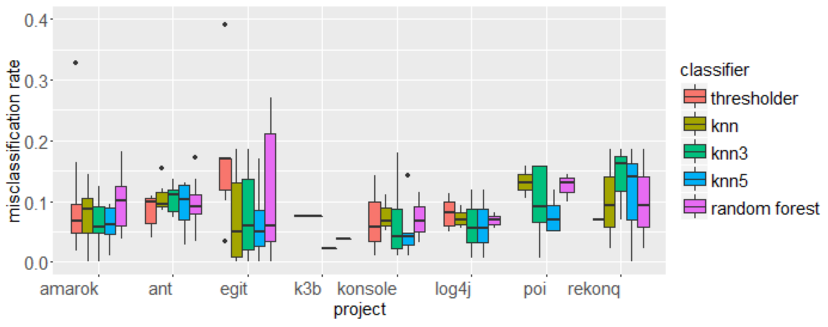


Figure 6.11.: Misclassification rate for individual contribution models.

Project	Thresholds	KNN	KNN3	KNN5	Random Forest
Amarok	0.089	0.077	0.066	0.062	0.096
Ant	0.085	0.110	0.105	0.094	0.096
Egit	0.172	0.073	0.079	0.063	0.113
K3b	0.076	0.076	0.023	–	0.038
Konsole	0.070	0.077	0.068	0.052	0.072
Log4j	0.080	0.292	0.060	0.060	0.403
Poi	–	0.132	0.236	0.249	0.125
Rekonq	0.070	0.101	0.140	0.109	0.101
average	0.097	0.108	0.100	0.088	0.127

Table 6.4.: Average *mr* for individual models.

General Developer Models

Knowing that we can model developers contribution behavior with HMMs, we are interested in examining the similarity between the retrieved models according to our role definition. If the observed similarity is high we are confident to build general models as an average per role to use these models later for prediction purposes.

The dynamics of developers during open source software development become visible within the transition matrices of the different developers. These matrices indicate how likely the contributors switch between low, medium, and high involvement. Before we build general models, we calculate the correlation between the transitions and compare the emission distributions.

For transitions, Table 6.5 showcases the project-wise correlations for major models using KNN5 (most individual models available). For *K3b*, no individual major models could be retrieved. In the table, it can be seen that the transitions are highly correlated. The results for core developers range from 0.979 to 0.999 and for minor developers the values spread between 0.992 and 0.999. With other classifiers it behaves nearly the same unless for minor

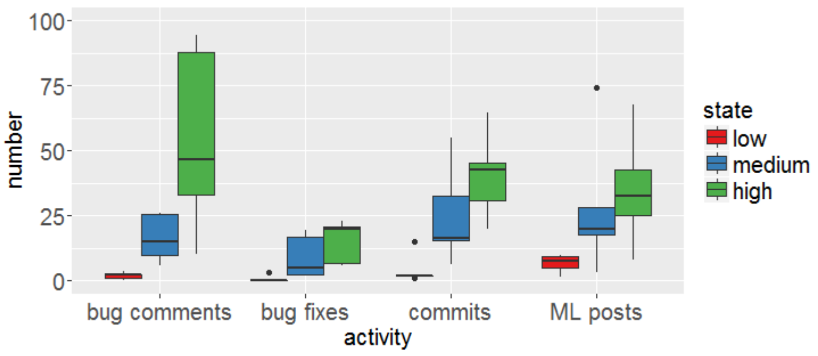


Figure 6.12.: Means μ_k of individual core models.

models, low negative correlation can occur. We believe this to be due to the fact that minor models are the most diverse under the investigated models and can depend strongly on the project and even on particular developers. Moreover, some models only switch between low and medium and, thus, never reach the high state. We observed the same considering the entity-wise standard deviations. All correlations are reported in Appendix A.

The emissions present the workload of developers. The distribution models how likely the commit, bug, and ML activities occur in the different states.

In contrast to the transitions, we observed a great diversity in the emissions distributions of individual developer contribution models. This means, that although the underlying dynamics are similar for the same developer role, the workload is hard to characterize generally. The amount of work and communication activity may depend much on the personal background, experience, and sociality of individual contributors.

Figure 6.12 illustrates this phenomenon. There, the mean values μ_k with $k = \{low, medium, high\}$ for all individual core models can be retraced. Generally, for low involvement, there is no great diversity. As well, the expected values for bug fixes do not spread that much. The highest variation occurs for bug comments in a high involvement

Project	Amarok	Ant	Egit	Konsole	Log4j	Poi	Rekonq
Amarok	–	0.991	0.994	0.993	0.982	0.988	0.982
Ant	0.991	–	0.996	0.968	0.950	0.998	0.949
Egit	0.994	0.996	–	0.978	0.961	0.994	0.960
Konsole	0.993	0.968	0.978	–	0.997	0.965	0.997
Log4j	0.982	0.950	0.961	0.997	–	0.946	0.999
Poi	0.988	0.998	0.994	0.965	0.946	–	0.945
Rekonq	0.982	0.949	0.960	0.997	0.999	0.945	–

Table 6.5.: Project-wise correlations of transition matrices of major developers.

state with a difference of over 60 comments. However, we believe that this divergence can be comprised by a general distribution as an average over the individual models of a role. Thus, we built general models and compare the performance of individual and universal models.

Following the process described in Section 5.4.3, we derived three general HMMs $\lambda_i = (A_i, B_i, \pi_i)$ with $i \in \{core, major, minor\}$, one for each developer role. Since all individual models converge towards $\pi = (1,0,0)$, we consequentially get $\pi_i = (1,0,0)\ \forall i$.

In Figure 6.13, the transition matrices of the general models are shown resulting from the averages of all individual models. Here, we also chose the general models using KNN5 as classifier, because it produced most individual models. The differences between the models retrieved from the other classifiers are only slightly. All transition matrices as well as means and covariance matrices of emissions are listed in Appendix B.

$$A_{core} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.65 & 0.28 & 0.07 \\ 0.39 & 0.48 & 0.13 \\ 0.09 & 0.22 & 0.69 \end{pmatrix} \end{array}, \quad A_{major} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.27 & 0.06 \\ 0.37 & 0.52 & 0.11 \\ 0.08 & 0.17 & 0.75 \end{pmatrix} \end{array},$$

$$A_{minor} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.70 & 0.24 & 0.06 \\ 0.30 & 0.51 & 0.19 \\ 0.09 & 0.16 & 0.75 \end{pmatrix} \end{array}.$$

Figure 6.13.: General transition matrices for developer roles over all projects.

Figure 6.13 shows how likely it is to switch between the states low, medium, and high for the different developer roles. There we observed that it is more likely to stay in a state than to switch between states (high values on the diagonal). Comparing the matrices between the developer roles, we observed that the general differences in terms of transitions of all developers are very low. Thus, the dynamics in developer project involvement are similar among all developers.

Contrariwise, the workload expressed by the emission distributions can vary a lot among

State	Core	Major	Minor
low	$(1,0,5,1)$	$(1,0,2,1)$	$(1,0,1,1)$
medium	$(22,6,27,13)$	$(12,3,6,11)$	$(7,3,2,12)$
high	$(33,11,50,45)$	$(14,4,11,27)$	$(3,1,4,5)$

Table 6.6.: Means μ_k of emissions for the general model.

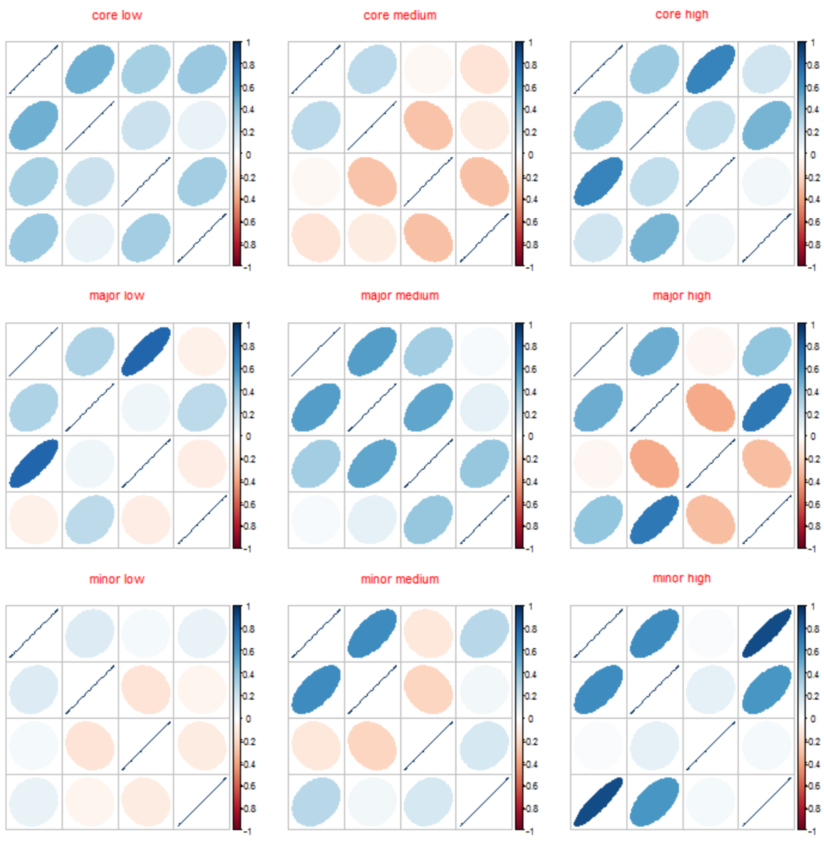


Figure 6.14.: Correlation plot for general models.

roles, and even within the same group of developers even though not to that extent. The distributions cover the likelihood of the observations to occur in the current state.

In Table 6.6, the means μ_k, $k = \{low, medium, high\}$ are reported for each developer role. As an example, a major developer in a high involvement state is expected to perform 14 commits, 4 fixes, 11 ML posts, and 27 bug comments in average. Surprisingly, we observed higher means for minor developers in a medium state than in a high state. We suppose this to be due to the small amount of information available for minor developers in a high state (resulting also in a high standard variation). The only factor that is higher in this case is the ML activity, which means that minor developers benefit more from discussions and, thus, also contribute in a high manner.

Figure 6.14 visualizes the correlation matrices Σ_k of the general emission distributions. Ellipses that are inclined to the right mean positive correlation whereas ellipses inclined to the left represent negative correlations. The stronger the ellipse is filled, the stronger the correlation is. For all developer roles, we observe a (slight to strong) correlation between

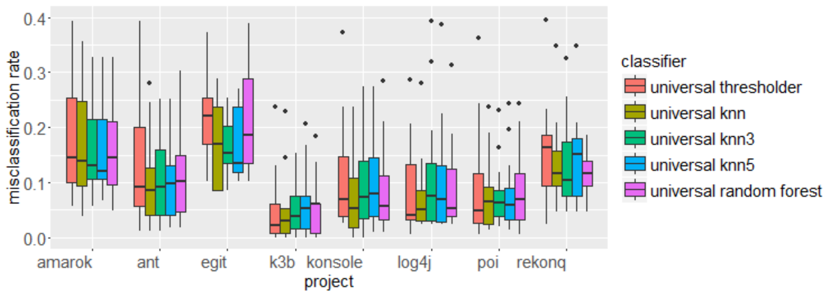


Figure 6.15.: Misclassification rate for general contribution models.

commits and fixes. That points out that more code developed leads naturally to more bugs fixed. In a high state, there is for every developer role a correlation between bug fixes and bug comments. This effect can be caused by the need of exchange with other developers about bugs when fixing more. Other patterns that occur seem to be more irregular. The sporadic negative correlations between, e.g., ML activity and bug fixes may indicate temporary concentration on a piece of work.

project	Thresholds	KNN	KNN3	KNN5	Random Forest
Amarok	0.189	0.164	0.158	0.161	0.168
Ant	0.137	0.096	0.102	0.0974	0.105
Egit	0.305	0.242	0.240	0.239	0.279
K3b	0.056	0.058	0.058	0.070	0.060
Konsole	0.109	0.075	0.096	0.102	0.089
Log4j	0.114	0.102	0.114	0.103	0.124
Poi	0.091	0.077	0.074	0.079	0.087
Rekonq	0.188	0.137	0.140	0.147	0.151
average	0.149	0.120	0.124	0.125	0.133

Table 6.7.: Average *mr* for universal models.

For a better comparability of the performance of individual and general models, we also calculated the misclassification rate *mr* for our general models. The results are shown in Figure 6.15. The best results are achieved for the projects *K3b*, *Konsole*, *Log4j*, and *Poi*. Average misclassification rates for the used classifiers and projects can be retraced in Table 6.7. The average *mr* ranges from 12% for KNN to 14,9% for the thresholds based approach. Thus, it only performs about 1-5% worse in comparison to individually retrieved contribution models. Moreover, the general models are appropriate for all developers, i.e., for all 125 developers we successfully predicted the most likely sequence of involvement states given the observations. This highlights the major advantage of the general models.

To demonstrate the usage of the general model, we present two application scenarios:

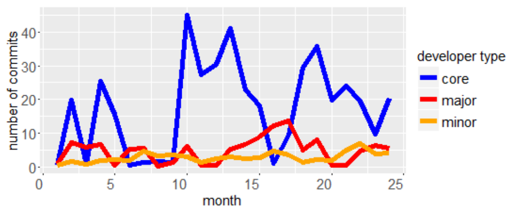


Figure 6.16.: Prediction of commits for 24 months.

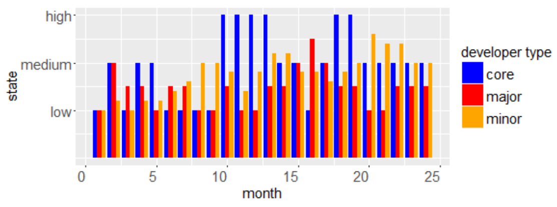


Figure 6.17.: Prediction of involvement states for 24 months.

one for the usage of the HMM general model as stand-alone predictor and one embedded into our simulation model which constitutes our DEVCON simulation model.

Application 1: Prediction of Workload

We now show an example application for the prediction of developers' project output. This means, that we consider a software project from the view of a project manager. It is important to plan the projects resources foresighted. For example, when making decision about the team constellation, it can be of help to estimate in advance what developer activity, e.g., in number of commits, will be present. This estimation can give the project manager feedback in her decision making. Assuming that the project manager aims to estimate the workload and project involvement of the intended team members for the next 24 months. She plans to involve one core developer, two major developers, and five minor developers. With our general models for the distinct roles, the manager can forecast the activity of the team for the next two years. Figure 6.16 shows the predicted outcome in terms of the commits performed of the different developers for one simulation run. Major and minor developers' values are presented as an average of all developers of the same role. The corresponding involvement states are shown in Figure 6.17. Over the whole forecasting period, 323 commits of the core developers, 149 commits of the involved major developers, and 63 commits of all minor developers are predicted considering the average over ten simulation runs.

Application 2: Simulation of Software Processes

The second application demonstrates the prospects of using the proposed model in context of our simulation tool. The results of a simulation model taking different software planning factors into account, can help to estimate the risks of the project (see [147]). The proposed simulation model aims to forecast software (quality) trends like the growth, bug population, and activity spent by the developers.

For the estimation of suitable model parameters, we mine real projects. Since the general models are built as an average of open source projects, the simulation model used for

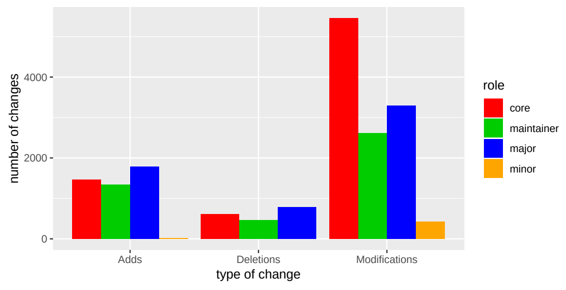


Figure 6.18.: Types of changes by type for Log4j.

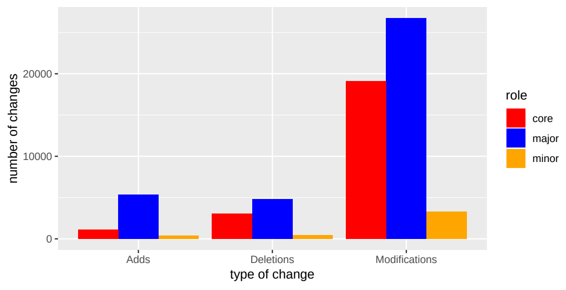


Figure 6.19.: Types of changes by type for Ant.

the evaluation was also fed with information of these projects except of *K3b* and *Rekonq*, since they were not part of the first study in this context exhibiting only very slight differences compared to our preliminary work [12]. Therefore, a lot of parameters describing the behavior of software developers (as agents) needed to be re-estimated: activity, workload (number of commits/fixes), and types of changes per commit. The developer activity and commit frequencies are modeled on the one hand based on the general models and on the other hand based on average heuristics for the used projects for comparability.

In figures 6.18 and 6.19 the changes per developer role are shown. These changes include adds, deletions, and modifications. The changes performed can be project-specific and also depend on the amount of developers of the different roles involved (compare Table 6.3). *K3b* and *Log4j* are the only projects having a real maintainer (marked as core in the table). They naturally perform a lot of modifications since they fix a huge amount of bugs.

After the mining, the parameters of the simulation model are adapted according to the mining results. Then, for each project the parameter set is adapted: number of different developers per role, expected size, and project duration. For all experiments an average of five simulation runs is considered.

A general observation and achievement is that with the adapted simulation model, we can simulate more unsteady growth trends with high activity phases, which more likely represent the reality than an idealized growth curve.

An example is illustrated in Figure 6.20. There the growth rises and falls time dependent from the workload of developers constituting in their states displayed in Figure 6.21. The states are presented as average over all involved developers of the same role. In *Egit*, only major and minor developers are identified. For example, the large amount of major developers contributing high in simulation round 100-300 (Figure 6.21) entails a strong increase in the total number of files (Figure 6.20). Such behavior also has an effect on the evolution of coupling, bug population, and overall software quality.

For the validation, we compare empirical project trends, with the simulated trends. We observe, that the greatest difference between the new standard average model and the im-

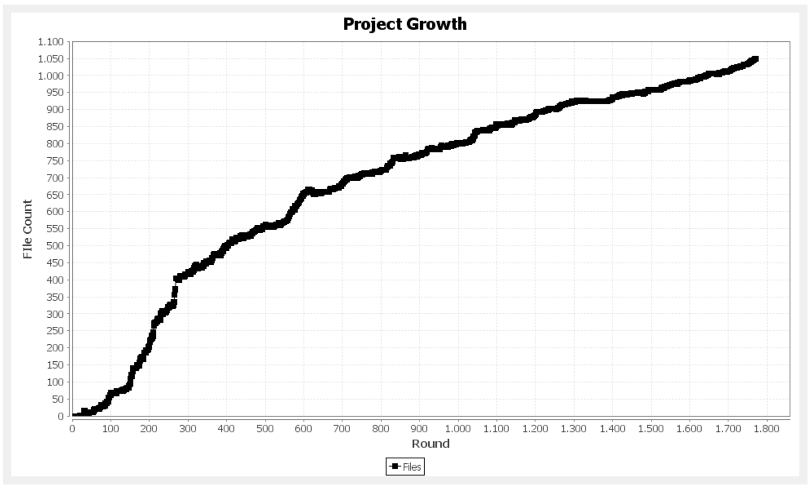


Figure 6.20.: Simulated project growth in Number Of Files (NOF) for Egit.

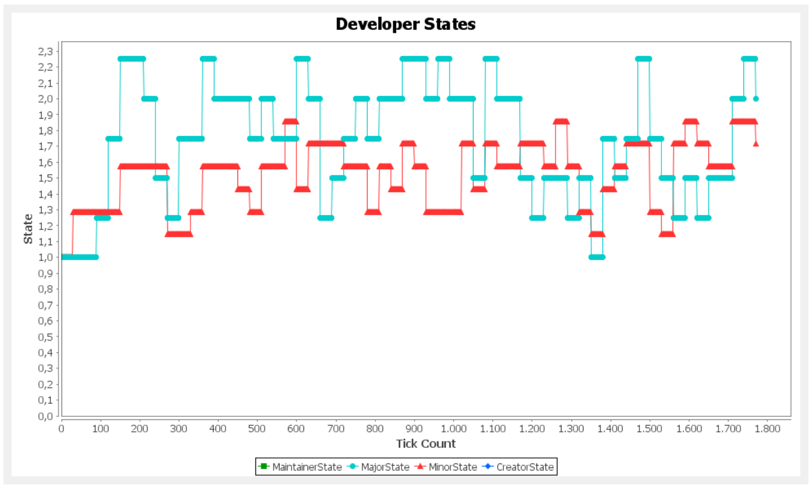


Figure 6.21.: Simulated developer states for Egit.

plemented HMM-based general model, arise in terms of the activity of developers, i.e., commits performed and files touched. Thus, we report the affected metrics *NOF* and *NOC* (Number of Commits) in this work. Other trends were just slightly affected and still representable.

In Table 6.8, the metrics *NOF* and *NOC* are reported for all projects under simulation. We observed, that the simple average model as an instantiation of the STEPS model in almost all cases produces too few files and in half of the cases also too few commits. For *Konsole* and *Poi* the simulated results match the empirical observations, but for *Log4j* too much

Project	NOF	NOF_{STEPS}	NOF_{DEVCON}	NOC	NOC_{STEPS}	NOC_{DEVCON}
Amarok	3200	1500	2364	28043	6100	13797
Ant	2200	1364	2056	15220	7483	15598
Egit	1450	525	963	3219	995	2663
Konsole	315	314	350	5746	6939	15362
Log4j	620	524	696	3498	6177	13184
Poi	4000	1730	2916	6095	5738	12278

Table 6.8.: Empirical and simulated metrics.
NOF=Number of Files, NOF_{STEPS}=NOF simulated, NOF_{DEVCON}=NOF simulated with developer phases, NOC=Number of Commits, NOC_{STEPS}=NOC simulated, NOC_{DEVCON}=NOC simulated with developer phases.

activity is simulated. On the contrary, the DEVCON simulation model performs better in NOF, even though a little too less in some cases. Considering NOC, the developers are too active except for *Ant* and *Egit* where it fits well, and *Amarok*, where the simulated results are too low.

6.3.4. Discussion

First, we analyzed individual developer contribution models retrieved via HMMs. Generally, the state sequence patterns are similar for the same developer role. We observed that core developers tend to be continuously active. They can also stay in a high state although contributing less. The fact that open source developers may work on specific tasks and distribute their work independently is more visible in the behavior of the other roles. This results in more irregular state patterns. We also observed that minor developers seem to benefit more from discussions than coding some times which helps them to be in a higher involvement state.

The average misclassification rate of the individual models reaches from 8.8% to 12.7% depending on the classifier. In summary, we can answer *RQ* 2.1 like following: Developers' contribution behavior can be modeled accurately using multi-dimensional HMMs.

Furthermore, we observed that developers tend to behave similar in terms of their dynamics, but their workload can differ significantly. That matches our intuition in so far that the amount of work spent by developers do not only depend on their role and involvement in the project, but also is influenced by other factors, e.g., their expertise [57], experience, interests, and collaboration factors. For a more accurate model, personal aspects may be taken into consideration. This presents an interesting conclusion, because the involvement dynamics can be expressed generally whereas the actual workload is more complex to model. Notably, the general model performs only about 1-5% worse in comparison to individual models. This already states the answer to *RQ* 2.2.

For the simple use of the general contribution models, we showed that it can simulate basic behaviors of developers and their activity in a project. This could also be used for companies to estimate the course of open source projects before using it for development. For a more sophisticated feedback, practitioners could respect more factors belonging to the software development life cycle, e.g., like a simulation tool. These observations form the answer to *RQ* 2.3.

Considering the application of the general HMMs for our simulation, we can say that the average models perform best for mid-size projects and that the phases simulation performs slightly better for the given projects. Moreover, more realistic curves could be produced with the phases model. But, it is a major challenge to build a model which is valid for projects different in size and workload. To tackle this problem, one could introduce a project size parameter to adapt the workload of developers according to the project size. Alternatively, models for different sizes or different development strategies could be learned. Practitioners should be cautious using the introduced method since our approach is based on randomly selected projects and may not represent the desired project context. Our approach is also aimed to be interpreted as a decision help in terms of trend analyses instead of construe numbers. These observations pose the answer to *RQ* 2.4.

6.4. Case Study 3: Dynamic Project Activity Model

The aim of this case study is to evaluate the approach for OSS dynamics in another similar context. The statistical learning via HMMs is transfered into the summarization of the project activity as a whole to assess whether a project is still under (active) development.

6.4.1. Setup

For evaluating project activity, we consider developer activity visible in commits, developer interest retrieved by mailing list posts, and user interest defined by posts in the user mailing list. The assumption behind the choice of these attributes is that a "healthy" software project lives from the contribution of developers as well as the usage and discussions by users of the software. The importance of the developer as well as the user mailing list for project communication is, e.g., highlighted in [148].

The case study is designed to assess project activity based on commits (developer activity) and ML posts by developers (developer interest) and users (user interest). Therefore, we counted commits from the VCS as well as ML posts from the dedicated mailing lists. In comparison to our other studies, we used the SmartSHARK platform [30] to retrieve the desired information. SmartSHARK allows to process mailing list data which can be very large and can additionally used directly with *R*. Our approach is summarized in Figure 6.22. After collecting the data, we let our analysis run and get a sequence of underlying activity states via the HMM training and the Viterbi algorithm. The main idea behind this is the

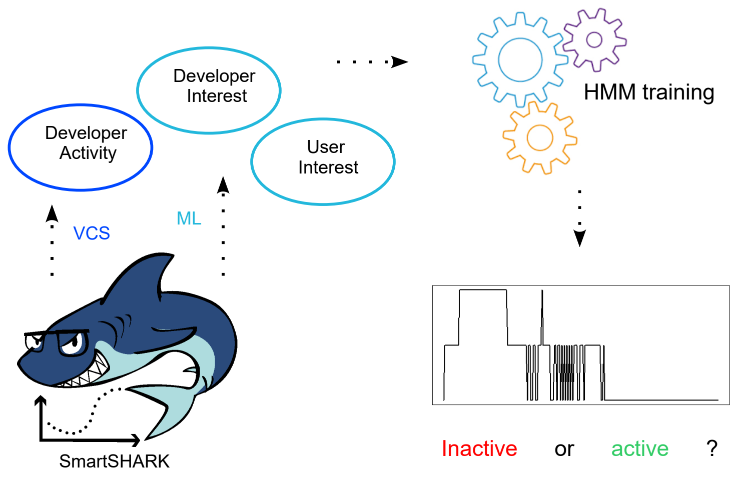


Figure 6.22.: Data Mining and Processing for the Evaluation of Project Activity.

assumption that the summary of the different layers of activity into one (non-observable) state facilitates the evaluation of a project and therefore, aids decisions concerning the use (for managers, users) or participation (for developers) of the project.

The modeling and classification is done similar to the learning of developers' contribution behavior as explained in Section 5.4.2. The only difference is the length of the input observation, since we have three observations for each month instead of four, e.g., $x_i = (31, 58, 112)$ with 31 commits, 58 developer posts, and 112 user posts representing the project activity in month $i \in \{1, ..., n\}$ and n the duration of the project in month. Still, the observations are treated the same way, since in Case Study 2, two observations were summarized into one activity level (code contribution) which also ends up in three pre-labeled states.

Data Selection and Cleansing

To assess project activity, we mine commit as well as mailing list data. Thus, a prerequisite for selected projects in this case study is the availability of the VCS and the ML. For the mailing list, we distinguish between developers and user based on the name of the mailing list, e.g, *zookeeper-dev* for the developers and *zookeper-user* for the users. The availability of these two lists establishes another prerequisite. Besides, for some projects contained in the SmartSHARK database the user mailing list is used only sporadically such that the data is too few for our experiments. Moreover, all projects belonging to Apache

Commons are not appropriate for this study, since they share a common mailing list. After this filtering, a subset of 16 Apache projects remained. In the following, we give an overview of selected projects containing the field of applications and project characteristics.

Accumulo [`accumulo.apache.org`] Accumulo is a distributed key-value store database engine using Hadoop and Zookeeper. Aa many other projects, Accumulo was incubated by Apache and its usage is widespread.

Ant Ivy [`ant.apache.org/ivy/`] Apache Ant Ivy is a sub-project of Ant, functioning as package dependency manager. Ant is a XML-based tool for the automation of build processes.

Archiva [`archiva.apache.org`] Archiva is a web-based repository management system to build artifacts. Therefore, it can work together with maven repositories providing on-demand mirroring of the artifacts.

Cayenne [`cayenne.apache.org`] The Cayenne project develops a Java object-to-relational mapping framework. With the integrated modeler, users are able to generate code directly from database schemes.

Deltaspike [`deltaspike.apache.org`] Deltaspike provides a set of CDI (Context and Dependency Injection) for Java projects. As an extension, it has to be used in conjunction with an CDI implementation.

Kafka [`kafka.apache.org`] Kafka is a distributed streaming platform that runs as cluster and is able to contain different data centers. Thereby, Kafka provides storing and processing of streams.

Mahout [`mahout.apache.org`] Mahout is a distributed mathematical and machine learning framework. As such, it is especially convenient for large scale algorithms and applications. Often, it is used together with Apache Spark, a popular computing framework, as back-end.

Nutch [`nutch.apache.org`] Nutch is a scalable web crawler. It originates from Lucene, which is an information retrieval library. Nutch supports batch processing (using Hadoop) and rich storage possibilities.

Opennlp [`opennlp.apache.org`] OpenNLP is a machine learning based toolkit for natural language processing (NLP).

Pig [`pig.apache.org`] Pig is a platform that aims to facilitate the analysis of large data sets. The analysis programs usually run on Hadoop clusters. It uses a high-level language which is able to handle MapReduce jobs.

Storm [`storm.apache.org`] Storm is a distributed real-time computation framework. Hence, it is comparable to Apache Spark. Both platforms are highly scalable and flexible. Depending on your need, Storm can offer some advantages over Spark and vice versa, e.g., in multi-language support (provided by Storm).

Struts [`struts.apache.org`] Struts is a Java web application framework. It separates the elements of the Model View Controller (MVC) concept, such that larger applications are easier to develop and maintain.

Tez [`tez.apache.org`] Tez is a distributed execution framework aims to support data processing on Hadoop. For this, it uses complex directed-acyclic-graphs (DAGs) of tasks and can reduce the amount of MapReduce jobs this way.

Tika [`tika.apache.org`] Tika is a meta data and text analysis toolkit. It supports a large set of file types and is flexible for different usages (e.g., as Java library, command-line tool). Tika is often used for search engine indexing.

Xerces [`xerces.apache.org/xerces2-j/`] Generally, Xerces provides a set of software libraries for parsing and processing XML files. The version integrated in the SmartSHARK database is the xerces2 java parser.

Zookeeper [`zookeeper.apache.org`] Zookeeper started as sub-project of Hadoop and provides similar features for large distributed systems, such as synchronization, coordination of processes, and configuration service.

Project properties are listed in Table 6.9. The table contains the observed time period, the total number of commits in the timespan, the corresponding amount of posts in the developer mailing list, as well as the number of posts done by users.

The period of selected projects starts at the initial commit at GitHub. Sometimes the starting point has to be adapted when some old project history is imported, but the mailing list discussions starts later. This can be the case for projects which became an Apache project during their life-cycle like Apache *Ant Ivy* which adopted *Ivy* as sub-project of *Ant*. Since the SmartSHARK database is steadily updated and the mining of mailing lists was not complete for the year 2018, we cut all gathered observations at the end of 2017 to guarantee the availability of all data needed for our experiments.

We discovered a special case for the *Zookeeper* project which had four MLs: two for developers and two for users. This is due to the import of the mailing lists before becoming

Project	Period	Commits	Developer posts	User posts
Accumulo	10/2011 – 12/2017	9760	11314	7393
Ant-Ivy	06/2005 – 12/2017	3175	45347	9121
Archiva	11/2005 – 12/2017	10250	3710	4437
Cayenne	01/2007 – 12/2017	6546	8960	12224
Deltaspike	12/2001 – 12/2017	2296	5540	2192
Kafka	08/2011 – 12/2017	6544	26547	35737
Mahout	01/2008 – 12/2017	4133	15184	22647
Nutch	01/2005 – 12/2017	3498	9102	33995
Opennlp	09/2008 – 12/2017	2684	3307	2898
Pig	09/2003 – 12/2017	5064	5729	15184
Storm	10/2013 – 12/2017	11953	4554	14173
Struts	02/2006 – 12/2017	5944	41116	215461
Tez	03/2013 – 12/2017	3568	1546	2022
Tika	03/2007 – 12/2017	4816	5003	3186
Xerces	11/1999 – 12/2017	7787	12711	14202
Zookeeper	11/2007 – 12/2017	2889	6773	11361

Table 6.9.: Projects properties.

an Apache project. Therefore, we combined the two lists belonging to the same group of people.

The mining of mailing lists can be a tedious task since the data is of unstructured nature. Therefore, a careful preprocessing is required [20] even if we are not interested in the content of the messages. Besides removing duplicates, we also removed empty messages as well as automatically generated messages produced by the VCS or ITS. For filtering theses messages we created a list of keywords which was extended step by step during the mining process. All messages containing a listed keyword in the subject were filtered before analysis. The identified keywords are the following:

GitHub, cvs commit, svn commit, jira, Build, Hudson build, Jenkins build, ANNOUNCE, DO NOT REPLY, Nutch Wiki.

For the user mailing list, we only had to filter empty messages. For cleaning the commit data, we filtered duplicated commits which occurred when a developer committed on a selected branch and later on another branch. These commits could be identified comparing the author date and the commit message.

6.4.2. Evaluation Criteria

We aim to describe the activity level of OSS projects based on different communication and contribution factors indicating the interest in the project. In addition, we aim to detect patterns for active and inactive projects which can be used for interpreting whether a project is likely to become inactive. For the HMM training, we use labels generated by different classifiers. To check whether the states produced by the HMM mirror these activity estimate, we use the misclassification rate *mr* introduced in Section 6.3.2 and used in Case Study 2.

To evaluate the application of HMMs for assessing project activity, we compare our aggregated approach expressed by the resulting sequence of hidden states with a simple view (plot) on the single activities. For this, we perform an AB/BA crossover study which is explained in the next section.

For the evaluation of the crossover study, we calculate the effect sizes and their variances according to Madeyski et al. [72] as introduced in Section 2.6.

Since no well-accepted definition of (in)activity of software projects exist, we establish a ground truth by an expert opinion. In doing so, we ask two software engineers working at our institute to label the sixteen projects as active or inactive, respectively. These experts take the projects commit history, as well as the project website, release history, and stats provided by GitHub into account for judging the projects. In the case of disagreement, the two experts discuss until they reach an agreement.

We compare this definition with the threshold-based, where a project is declared as inactive where no commits are visible in the VCS over a certain period of time, often 12 months [149].

6.4.3. Results

The results of this case study are two-folded. It consists of a mining and statistical learning part to recognize patterns and levels of project activity. For evaluation of the results, we performed an AB/BA crossover study to assess the intepretability of a summarization of different activities into one state. Thus, we first report all results concerning the mining part followed by the results of the AB/BA crossover study.

Mining

From SmartSHARK, we derive the monthly observations in number of commits, developer posts, and user posts for each project. To narrow down the observation space, we classify the observations into low, medium, and high activity. For this, we use a subset of 20 manually classified observations as input for the classifiers. As classifiers, we use KNN3 and Random Forests, since the choice of the classifier only had a small impact on the results as shown in Case Study 2 (Section 6.3.3). Still, we take two different classifiers for comparison into account. Again, the overall score is assigned via a majority vote.

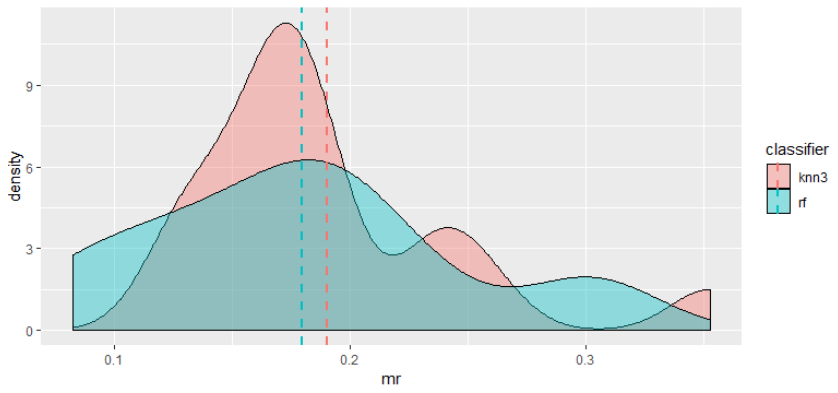


Figure 6.23.: Density of misclassification rate for all project activity models. Dashed lines represent the corresponding means.

The pre-labeled observation sequence is used for the HMM training. As an output, we get a summary of the three levels of activity into one state. The resulting sequence of these states which most likely produced the observations is calculated using the project-based fitted HMM $\lambda_j = (A_j, B_j, \pi_j)$ for each project j. Comparable to the developer contribution models, all projects tend to start with low activity, i.e, $\pi_j = (1,0,0)\ \forall j$.

In Figure 6.23, the misclassification rate *mr* for all projects and classifiers is shown. The mean over all projects for each classifier is drawn as a dashed line. For K-nearest neighbor, all projects could be trained, whereas for Random Forests one project failed. For KNN, we reach a value of $mean(mr) = 0.19$ and for Random Forests we get $mean(mr) = 0.18$. Thus, Random Forests perform slightly better. Generally, the error rate is higher than for developers' contribution models. This is due to the smaller sample size and, thus projects which are adverse for the HMM training, e.g., that often switch between low and high skipping the medium state, carry more weight into the mean score. Examples for such projects are *Zookeeper* and *Cayenne*.

A deeper analysis of the resulting HMMs for project activity is conducted after the labeling by experts, since then we can examine characteristics of active and inactive projects and compare the results for the identified subsets of projects.

Expert Labeling

For the separation of OSS projects into different groups of activity, we employed an expert team of two software engineers working at the institute of computer science. The experts task was to classify the projects by means of activity visible on the projects website and GitHub page, e.g., by examining the amount and history of commits, forks, and releases. As a first step, each expert build an opinion on their own and as a second step, they discuss when they disagree.

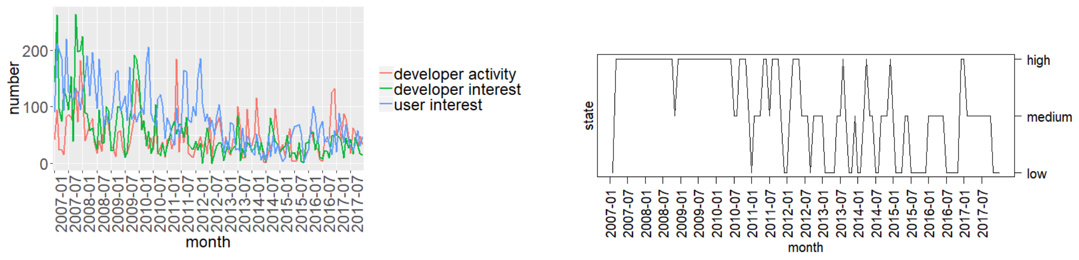
During the experts' discussions, the issue arose that some projects seem to be in between, not active in contributing in a fairly fashion anymore, but still fixing bugs and bringing up releases. Thus, the experts decided to introduce a third group which contain projects under maintenance. As a result, they came up with four active projects, ten maintenance projects, and two inactive projects. For our crossover study, we include maintenance projects in the group of active projects.

Taking the definition by Khondhu et al. [149] into account, the only project of our case study classified as inactive would be the *Ant* project which is considered as under maintenance in our study. The reason for this is that this is the only project where there were no commits performed over a subsequent time period of 12 months. In our approach, the other activities (developer and user interest) were enough to keep the project alive.

Model Comparison

The starting point of our model comparison are the OSS projects classified by the experts. This expert classification resulted in four active projects, ten projects under maintenance, and two inactive projects. First, we compared the state-based representations of the different groups of projects visually. By this, we identified similar patterns for the different levels of activity. Hence, we give an example for each group and describe the observed pattern.

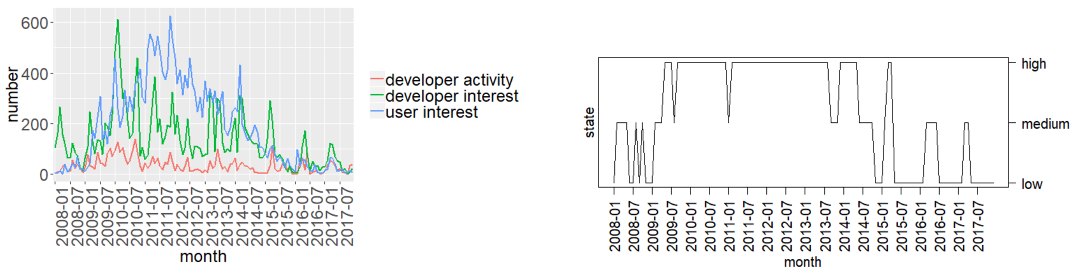
Active Projects Active projects often exhibit an alive beginning with a falling level of activity, but still regular actions. The example project in Figure 6.24 shows this trend. The course can also be steadily increasing.



(a) activity plot (b) states plot

Figure 6.24.: Example of a project classified as active (Cayenne).

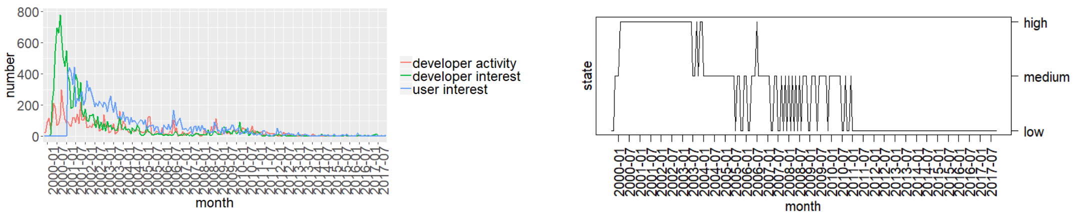
Maintenance Projects Similar to active projects, projects under maintenance show an active beginning with decreasing trend. Still, in contrast to the active ones, here the current activity can be more irregularly or very little. Figure 6.25 shows a project classified as maintenance project. This pattern is caused by a typical phenomenon in OSS projects: At the beginning there is an increasing interest in the project and a high development effort. After the projects evolve stable, only bugfixes for new releases or other adoptions to evolutionary effects have to be conducted.



(a) activity plot (b) states plot

Figure 6.25.: Example of a project classified as under maintenance (Mahout).

Inactive Projects In our study, the inactive projects are clearly distinguishable from the others. After a period of mixed activity (mostly medium and high) a long period of low activity follows. In Figure 6.26, such an example is shown for the project xerces.



(a) activity plot (b) states plot

Figure 6.26.: Example of a project classified as inactive (Xerces).

The benefit of using HMMs for the assessment of project activity, is the summarization of activities into one representative state. Thus, it implicitly determines how many activity is enough to be in a medium or high state. Vice versa, it provides thresholds for the amount of activity which can occur for inactive projects.

Like we did for the individual developer HMMs (Case Study 2), we perform a correlation analysis for models of the same type. All correlations can be found in Appendix C. Showing a strong correlation, we calculated universal models for each activity type proceeding the same way: we build averages for each matrix entry for the transformation matrices. In addition, the emissions modeled as multivariate Gaussians could be combined using linear transformations (see Section 5.4.3).

$$A_{active} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.25 & 0.08 \\ 0.35 & 0.45 & 0.20 \\ 0.12 & 0.25 & 0.63 \end{pmatrix} \end{array}, \quad A_{maintenance} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.25 & 0.07 \\ 0.34 & 0.46 & 0.20 \\ 0.07 & 0.30 & 0.62 \end{pmatrix} \end{array},$$

$$A_{inactive} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.27 & 0.06 \\ 0.37 & 0.47 & 0.16 \\ 0.06 & 0.33 & 0.61 \end{pmatrix} \end{array}.$$

Figure 6.27.: General transition matrices for project activity over all projects.

Figure 6.27 shows the retrieved general transitions matrices based on the k-Nearest Neighbor classification. The difference for Random Forests as classifier is only marginal. All transitions and emissions for Random Forests can be found in Appendix D. For the transition matrices, we observed that the probabilities for switching between the states are quite similar. Though, some differences could be figured out: Active projects have a higher probability to go from a high state directly into a low state and vice versa. The reason for this could be that active projects are more often in a high state and also switch more frequently between all three states due to the nature of open source projects. Moreover, inactive projects generally tend to have lower probabilities for adopting the state of high activity. This is nearby since they always show a longer period of low activity or even inactivity.

State	Active	Maintenance	Inactive
low	$(35,80,118)$	$(14,37,51)$	$(15,43,66)$
medium	$(33,120,202)$	$(17,64,128)$	$(9,33,139)$
high	$(79,198,264)$	$(31,100,229)$	$(16,72,194)$

Table 6.10.: Means μ_k of emissions for the general project activity model.

The emission distributions for each universal model consist of three three-dimensional vectors μ_k describing the mean values of the summarized activities developer activity, developer interest, and user interest, with $k \in \{low, medium, high\}$. There, we observed that for low and medium activity the developer interest as well as the user interest are more influential since the differences for the means of developer activity are tiny or even decreasing (for active and inactive projects). This means that discussions are more important than code

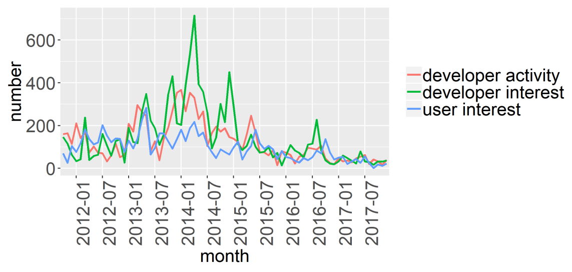


Figure 6.28.: Example of an activity plot (method A).

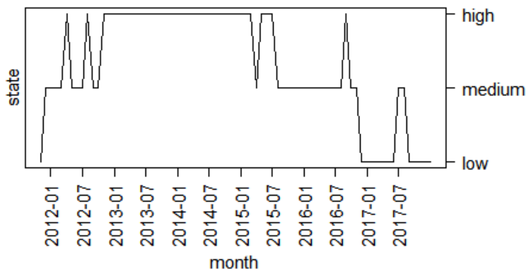


Figure 6.29.: Example of a states plot (method B).

contribution in periods of less activity to keep the project alive. Naturally, active projects have the highest means in general followed by maintenance and inactive projects. Interestingly, in a low state, inactive projects are slightly more active on average than maintenance projects. This can be due to general similar behavior of the two types with the difference that in maintenance projects the activity in higher states is more distinctive due to regular bug fixes and preparation for releases.

AB/BA Crossover Study

To evaluate the benefit of using HMMs for assessing project activity, we perform an AB/BA crossover study. Therefore, we compare a technique which directly process developer activity, developer interest, and user activity pictured as a multiple line plot - referred to as method A - with a plot showing the aggregated activity retrieved from applying Viterbi to the project-specific trained HMM which is defined as method B.

Figure 6.28 shows the different levels of activity, i.e. the activity plot (method A) whereas Figure 6.29 presents the HMM-based representation, i.e., the states plot (method B) for the same project (*Accumulo*). The expected benefit of using technique B, is that the overall project activity can be assessed more intuitively. With this, we aim to facilitate the decision making process to join as well as to use the investigated software project.

For the crossover study, we designed a questionnaire with the following characteristics:

- Each of the 16 projects is presented one time either by method A or by method B.
- Each participant has to judge eight projects with method A and eight projects with method B.
- Half of the participants judge eight projects with method A followed by eight projects with method B and the other half vice versa. The first group is referred to as Sequence Group 1 (SG1) and the second group establishes Sequence Group 2 (SG2).
- Each project has to be evaluated independently and marked as active or inactive.

In addition, the participants are asked to mark how sure the feel about their decision. For this, there are four options available: very, fairly, halfway, and little. This design decision is motivated by the projects where the experts had problems to put them into active or inactive and came up with the maintenance state. We assume, that the sureness decreases even for those projects. To take this for our evaluation into account, we determine the correlation between sureness and wrong classifications. Another suggestion by the experts was to consider maintenance projects as active in the questionnaire since it requires some background knowledge about software development to judge whether a project may be under maintenance. But, everybody can judge a plot whether there is much or low activity visible.

The first page of the questionnaire gives a short introduction into the task and the last page presents an overall evaluation comparing the two methods, where the participant should evaluate which method can be used more intuitively.

To generate the questionnaires meeting the requirements listed above, we apply stratified sampling [150]. There, the data is divided in disjunct groups called strata and a sample is drawn using a specified design. We apply stratified random sampling, i.e., the chosen design is random. This way, we guarantee that each participant get the same amount of representatives of the two strata active and inactive. An example of a designed questionnaire can be found in Appendix E.

As participants, we chose a students course on software testing offered at the University of Göttingen including 28 master students. Sometimes, researches argue on the representativeness of students as software professionals, but a larger study [151] on this topic pointed out that concerns are valid when experience matters, but for the introduction of a new technique, no significant differences could be found.

To evaluate the results, we calculate the score achieved by each participant. The score indicates the percentage of projects classified by the participant in agreement with the experts classification. During analysis, one questionnaire turned out to be invalid, since the participant marked all projects as active with high sureness. Thus, we omit the participant for our analyses.

Sequence Group	Statistic	Score AP	Score SP	Diff	Participant Total
SG1	Mean	0.6786	0.625	0.0536	1.3036
	Var	0.0234	0.0409	-0.0175	0.0642
	Obs.	14	14	14	14
SG2	Mean	0.6923	0.6538	0.0385	1.3462
	Var	0.0563	0.0317	0.0246	0.0879
	Obs.	13	13	13	13

Table 6.11.: Descriptive statistics of AB/BA study.

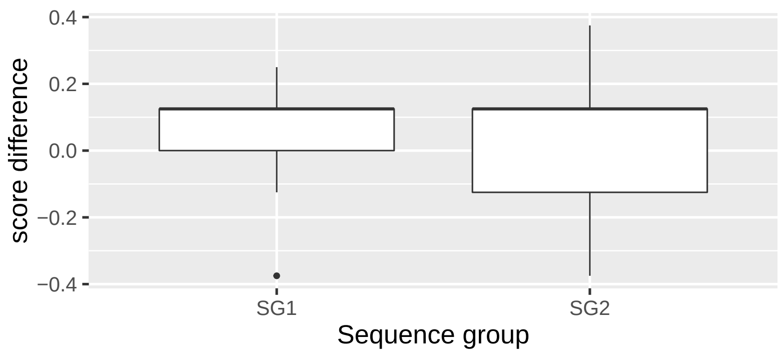


Figure 6.30.: Boxplot of score differences between techniques in SG1 and SG2.

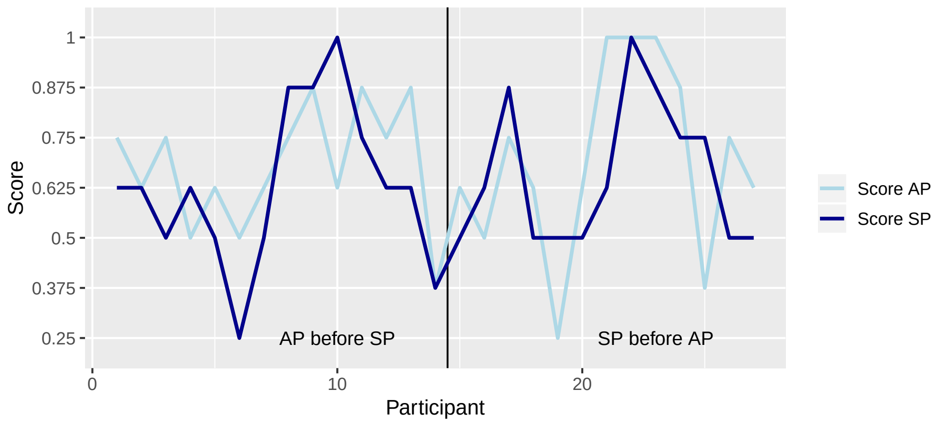


Figure 6.31.: Individual scores of participants of each sequence group.

For comparing the outcomes of participants produced by the two chosen techniques, we first calculated the basic descriptive statistics of the experiment. Those are shown in Table 6.11. For each sequence group, we calculate the means and variances of the scores as well as the differences between method A and method B. *Score AP* is referred to the score achieved by judging the activity plots whereas *Score SP* is the score reached by judging the states plot.

Additionally, Figure 6.30 shows a boxplot of the score differences in the two sequence groups SG1 and SG2. Both median values are at 0.125. The only difference lies in the size of the boxes, which indicates a slightly higher variance in the data for SG2. However, both boxes are close to zero, which indicates no significant effect between method A and method B.

Moreover, we report the individual outcome of participants in Figure 6.31. In both groups, eight participants performed better using method A (Score AP). In SG1 (AP before SP), three participants reached a higher score with method B (Score SP), while in SG2 (SP before AP), this is true for four of the participants. The rest performed equally with both techniques. This could be an indicator that the activity plots are more intuitive. Still, if this is a significant effect cannot be judged yet. Thus, we proceed with calculating the effect sizes as well as their variances.

Calculation of Effect Sizes

For a deeper analysis, we need to have a closer look at the data from the crossover study. From the basic statistics shown in Table 6.11, we can calculate all needed measurements for a meaningful analysis of the crossover study. The calculation relies on the formulas introduced in Section 2.6. To ensure the correctness of the values, we additionally use the *R* package *lme*4 [152] as suggested by Madeyski et al. [72] which is designed to fit linear models. For comparison, the *R* output is provided in Appendix F.

Statistic	Description	Value
$\hat{\tau}$	technique effect size	0.0460
$\hat{\pi}$	period effect size	-0.0076
$\hat{\lambda}_{AB}$	period by treatment interaction effect	-0.0426
s^2_{IG}	within period and within technique variance	0.0368
s^2_{diff}	difference score variance	0.0373
s^2_w	within participant variance	0.0182
$\hat{\rho}$	correlation between outcomes in both periods	0.4940
$var(\hat{\tau})$	variance of technique effect size	0.0014
$se_{\hat{\tau}}$	standard error of technique effect size	0.0367
t	t-test for significance of technique effect size	1.2534

Table 6.12.: Statistics of AB/BA study.

Table 6.12 shows all retrieved measurements for the conducted study. First of all, the period effect size $\hat{\pi}$ is extremely small. The technique effect size with 0.0460 as well as the interaction effect size with -0.0426 are larger in comparison with the period effect size but still low. This observation implies that the score achieved is neither dependent on the sequence of the techniques nor the technique itself or the time period. Generally, the students performed slightly better using the activity plots (method A). For the states plot (method B), it turned out that the students' judgments were slightly better using this technique first. Anyhow, no significant differences could be figured out.

Furthermore, the correlation between the outcomes in both periods is with 0.4940 relatively small, i.e., the correlation between repeated measures is low [72].

Given the t-statistic of 1.2534 and the degrees of freedom $df = n_1 + n_2 - 2 = 14 + 13 - 2 = 25$, the $p - value$ is 0.2217, i.e., the result is not significant at $\alpha = 0.05$.

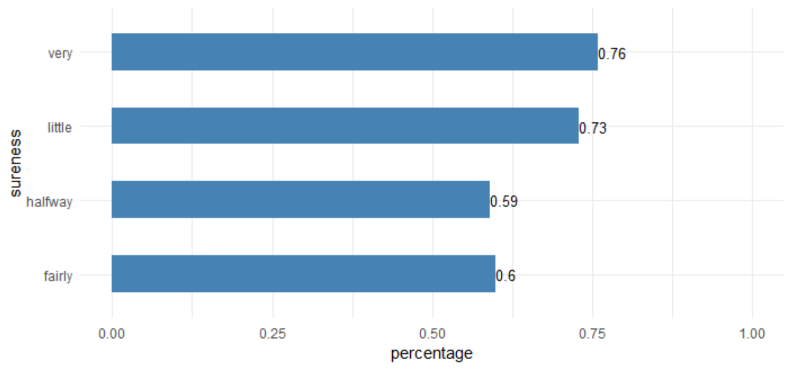


Figure 6.32.: Percentage of correctly classified projects per sureness.

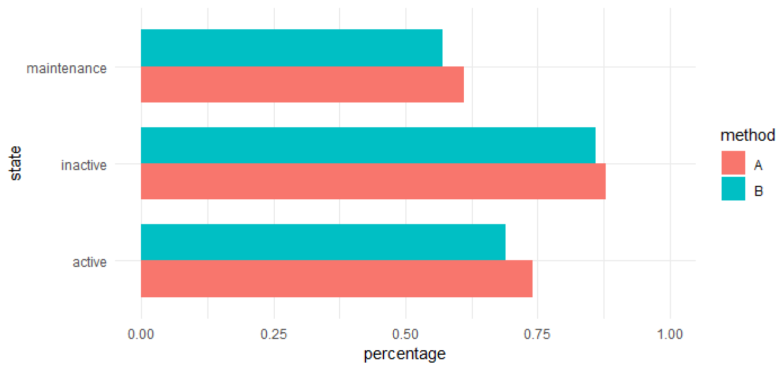


Figure 6.33.: Percentage of correctly classified projects per type and method.

As a next step, we check our assumption, whether the sureness of the students correlates with wrong classifications, especially for maintenance projects. First of all, we analyzed how strong the sureness correlates with right classifications. For this, we calculated the percentage of correct answers per sureness level.

Figure 6.32 shows that the answer where the participants were sure reached the best score with 76% correctness. Surprisingly, this is followed by little sureness with 73%. We believe this to be due to the nature of guessing as well as the difficulty to judge projects with low activity, particularly maintenance projects, as still active.

For fairly and halfway sure, the participants reach 60% correctness and 59%, respectively. This means, that their intuition matches roughly the reality.

Additionally, we are interested in the investigated behavior for the different types of projects: active, inactive, and maintained.

Figure 6.33 depicts the ratio of correctly assessed projects per project type and method. From this, we inferred that inactive projects were most easy to identify for the participants.

They reach 0.88% with method A and 0.86% with method B for inactive projects. For active projects, the participants were able to identify 74% of the projects with method A and 69% of the active projects with method B. Finally, the analysis confirmed our assumption that maintenance projects are the most difficult to judge displayed by the relatively low percentage of correct assignments: 61% correctly identified maintenance projects and 57% accordance using method B. This finding deliver insight into the issue of judging whether a software project is still maintained.

In summary, we identified common activity state patterns that helps practitioners to judge a project as still active or maintained with the help of HMMs. The advantage by the visual simplification of the different levels of activity could not be confirmed within our AB/BA study. Though, the expected values for activities together with the observed pattern can be used as an indicator whether a project is active or not.

6.4.4. Discussion

Firstly, we transferred our approach of using state based probabilistic models for open source software project dynamics to assess project activity. Different from Case Study 2, we only take the aggregated number of commits and ML posts by all developers into account to get a whole picture of project activity instead if individual contributions. We enrich this information with user interest in the project (user posts). We successfully trained a HMM for every software project and the average misclassification rate *mr* resulted in 0.18 for Random Forests and 0.19 with KNN. Although this values are not as low as for the developer contribution model, we still can state that we can model projects activity with a state based probabilistic model, i.e. HMM, which poses the first part of the answer to research question *RQ* 2.5.

To classify the projects into active and inactive, we performed an expert labeling. Out of it, the experts identified many projects lying "in between" and, thus, introduced a third category: maintenance projects.

To assess possible advantages of the state based representation of project activity, we performed an AB/BA study where we compared this with a basic multiple line plot for the three levels of activity: developer activity, developer interest, and user interest. We calculated all important effect sizes following the guidelines by Madeyski et al. [72] revealing no significance in the results. Anyway, the crossover study supported our assumption that maintenance projects are the most difficult to assess with the lowest percentage of correctly identified projects with about 20% less than for inactive projects.

Again, we build general HMMs for our analysis, one for each activity layer: Active, inactive, and maintenance. Generally, we found similar patterns for the same type of project concerning the state sequence. Supportingly, the learned average models show differences which help to characterize project activity. One finding indicates that in periods of little activity, discussions are a major factor to keep the project alive. As such, it can help practitioners to evaluate project activity and, thus, aid decisions regarding joining and using software projects. This finding completes the answer to research question *RQ* 2.5.

To provide a more concrete classification going without manual interpretation, a classifier could be learned based on the resulting models, e.g., the number of times the different states are occupied relatively to the the projects duration, the expected values of the different activity layers, and the trend displayed by the the activity of the last months related to the overall activity. However, the data provided in this case study is too few and, thus, hinders the learning of such a classifier.

7. Discussion

This section discusses the results gained from our case studies and interpret them related to the approaches applied. We start with a summary of the answers to our research question from the beginning of this thesis. Following, we report strengths and limitations of the approach. Finally, we discuss threats of validity of our work.

7.1. Answers to Research Questions

We summarize the answers to the research questions that were introduced in Section 1.2 and that arise from the finding of the conducted case studies.

7.1.1. *RQ* 1: Can we model software evolution using Agent-Based simulation?

In *RQ* 1, we ask whether we can model software evolution using Agent-based simulation. This research question is addressed in one large case study concentrating on different aspects: First, we need to identify what entities should be part of a meaningful model of software evolution as well as which parameters are needed to describe the process which relates to *RQ* 1.1. According to the software evolution literature and our idea of software evolution, we determined developers, software entities, e.g., files, classes, and methods, and bugs as model entities. To complement the model, relationships and attributes need to be identified. All this has to be done before the actual model can be evaluated. Thus, to answer *RQ* 1.1, a lot of preprocessing steps are necessary. Essential design decisions like the work of developers arise from the nature of software evolution since their work is visible by their commit behavior (changes to the software entities). Still, the granularity, e.g., the size of the commit, affected lines, has to be determined to find balance between model complexity and simplification. The proposed model (STEPS) to answer the research question mirrors basic commit behavior of different types of developers, bug occurrences, system growth, and relationships expressed by software networks, e.g., the change coupling graph for related software entities, i.e., files, induced by common changes to the repository. From this, we inferred suitable parameters by software mining that can instantiate software projects and measure the desired outcome. In the related case study, we identified the following changeable parameters able to display the mentioned software evolution scenarios: number of core, major, and minor developers, expected size of the system, expected project duration, and the initial cluster size of the change coupling graph. Thus, the answer to *RQ* 1.1 is the following:

Important parameters for the simulation of software evolution are the team constellation of developers, the expected system size, the project duration and the initial cluster size of the change coupling graph.

The next question *RQ* 1.2 deals with the estimation of required simulation parameters that are fixed and serve as input for the simulation model, e.g, the amount of work done by a core developer. The defined model determines the parameters that are needed and then in our approach retrieved by software repository mining. We make use of machine learning as well as statistical learning to grasp complex behavior, e.g., fitting by distribution to the number of commits performed instead of just taking the average of all developers. The approach is evaluated in two case studies, one for general software evolution (STEPS) and one for the extended DEVCON model reflecting a more fine-grained developer behavior. The results showed that the strategy can be successfully used to estimate suitable parameters. In the case studies, we compared empirical with simulated trends and compared values for, e.g., the number of files, the average change coupling degree, the number of nodes (files) and the number of commits. Consequently, we answer *RQ* 1.2 the following:

Simulation parameters can be estimated using mining software repositories and applying statistical learning methods.

Considering *RQ* 1.3, that asks which software evolution phenomena can be simulated, the answer cannot be entire, since there no register where all software evolution phenomena are listed exist. There might be more phenomena which we are not aware of at the moment. For this reason we selected different aspects of software evolution and tested on how good they can be mirrored using Agent-based simulation. Our case studies illustrated possible simulations of the following trends and phenomena: (sub-linear) file growth, average (dynamic) developer contribution behavior, bug introduction rates, and the evolution of change coupling networks. Thus, we answer *RQ* 1.2 the following:

Our simulation of software evolution can mirror software evolution phenomena like sub-linear file growth, developer dynamics, bug introduction rates, and the evolution of change coupling networks.

7.1.2. *RQ* 2: How can we model developer contribution behavior?

The second main research question *RQ* 2 is dedicated to modeling developers' contribution behavior. We answer this question with the help of several subquestions. The first subquestion *RQ* 2.1 ask whether a state-based probabilistic model is appropriate for modeling developers' contribution behavior. The idea of using a state-based model for the activities of developers is inspired by the nature of OSS projects. There, developers can choose freely which project they want to join, at which time of day they want to work and to which extent they contribute. Therefore, a model allowing for different states of involvement is beneficial. We choose HMMs as suitable for the approach. Another advantage of using HMMs is that the multi-dimensional version allows for multiple observations at the same point in time. We choose to include communication activities as well as code-related activities to describe developers' contribution. To train HMMs from empirical software repository data, a classification step needs to be done beforehand since no labeled contribution data, i.e., whether a developer is low, medium, or highly involved a point in time, exists. Thus, within our case study we evaluated on the one hand the choice of the classifier and on the other hand the goodness of fit of the HMM to the data. This is done by measuring the amount of mismatches between the learned hidden state and the classified one (misclassification rate). For every developer an individual HMM is trained. The results showed that the average misclassification rate for the individual models reaches from 8.8% to 12.7% depending on the classifier. These findings already state the answer to *RQ* 2.1:

Developers' contribution behavior can be modeled accurately using state-based probabilistic models like multi-dimensional HMMs.

To address *RQ* 2.2, that asks for the similarity of retrieved contribution models for the same developer type, we perform a correlation analysis for the transitions and compare the emissions of the general models. Since the transitions are highly correlated for the same developer type, we built general models taking the average over all HMMs that belong to the role. Thus, we build three general HMMs, one for each developer type (core, major, minor). These general models reveal some interesting insights: Firstly, the transitions are similar among all types of developers, but the emissions are quite different. This matches our intuition since the underlying dynamics maybe similar for all OSS developers, but their workload depend besides the assigned role on more factors, e.g., personality, expertise, technical and social interest in the project. Thus, the workload of developers is more complex to model. Furthermore, we evaluated the general models applied for training each developer state sequence again. The general models perform only $1-5\%$ worse in comparison to the individually trained models. A major advantage of the general models is that they are also appropriate for developers where an individual model cannot be trained, e.g., due to sparse observation data. Together, these observations present the answer to *RQ* 2.2:

Retrieved HMMs for the developer role are very similar in terms their transitions, but their emissions can be more diverse.

The next subquestion *RQ* 2.3 asks whether the retrieved general models can be applied in practice. To demonstrate the applicability we use the general developer models as standalone method to predict contribution behavior of a given set of developers. Besides, we embedded the general models into our simulation tool as refinement of the implemented developer behavior. Both methods are demonstrated in our case studies supporting the usage of general models in practice. This answers *RQ* 2.3:

General contribution models can be applied in practice, e.g, to predict future contributions of developers or for the simulation of software evolution.

For *RQ* 2.4, that asks if such a fine-grained developer contribution model improve simulation results, we compared simulation results for the STEPS (average contribution behavior) as well as the DEVCON model (fine-grained state-based contribution behavior). In doing so, we compared metrics like the project growth in number of files and the total number of commits. There we observed that for both simulation models the best matching metric values are achieved for mid-size projects. A significant improvement of simulation results cannot be confirmed which answers *RQ* 2.4:

Although able to reflect more realistic trends in the simulation of software evolution, significant results, e.g., for the comparison of empirical and simulated data, cannot be confirmed.

The last subquestion *RQ* 2.5 addresses the transferability of the proposed approach. We investigate this question in a separate case study where we use HMMs for summarizing project activity and describe the underlying dynamics of developer contribution as well as developer and user discussions. In contrast to the work done so far, we take the aggregated number of commits and ML posts into account, since we want to generalize from the individual level to the project level. The HMM training was successful for every project included in the case study. Although the average misclassification rate was higher than for modeling developer contribution, we can say that project activity can be modeled using state-based probabilistic models like HMMs. One major goal of this application is a possible characterization of project activity and, more importantly, project inactivity based on the trained state sequence. The early detection of software project that are likely to become inactive is a challenging task in software engineering. No labeled data of project activity exist which is the first burden in preprocessing the data. Therefore, we performed an expert labeling. The experts detected many projects that are not completely active, but also not yet inactive. Thus, we introduced a third state specifying maintenance projects. By analyzing general models built for each type of activity, we observed common characteristics, e.g., in terms of the state sequence pattern and the output produced in a low, medium, or high state. Although the approach produced fruitful results, for universal guidelines the data used is too small. Though, projects that are repeatedly in a low state, are more risky to become inactive. The advantage of our approach to existing approaches is that a low state does not necessary mean a total lack of activity. Instead, the HMM decides how much activity can be tolerated in a low state. Altogether, we can answer *RQ* 2.5 positively although limited by the amount of data used for the study:

A state based probabilistic model can be used for modeling project activity. On the project-specific level, they can help users and developers to get an overview of the project activity.

7.2. Strengths and Limitations

A major benefit of using simulation for software evolution is that it can be instantiated for every project and it is flexible to test different sets of parameters. This way, a feedback loop is established that helps project managers in making decisions. Different trends concerning software evolution can be simulated. However, the projects used for mining and parameter estimation may not be representative for other projects. Still, our case studies showed that the used simulation model produces reliable results matching the empirical values. Moreover, the size of the software graphs is limited due to the platform used. Very large projects with more than 10,000 files may lead to runtime problems. Thus, the next step is to improve the scalability, e.g., by using another platform or another type of execution of the simulation. Besides, there are many factors influencing software evolution a project manager may be interested in. Therefore, the proposed simulation model is extensible for other factors effecting the software development process.

The results of the HMM training for software developers can display actual contribution behavior like demonstrated in our case studies. It provides an overview on contributions on the individual level as well as on the project level when using general models. The HMMs can describe the underlying dynamics of OSS developers very well by allowing different states of involvement varying over the time. The general models can be applied solely as well as, e.g., for a fine-grained extension of developer behavior in the simulation. The later enables more realistic contribution and consequential file growth trends within the simulation. The HMMs also proved to be usable for summarizing project activity and offering a characterization for active, inactive, and maintenance projects. Due to the amount of data collected, no universal classifier could be trained.

Finally, the mining that constitutes a large part of this work depends partly on tools that are not developed on our own. This is good on the one hand since saving time and effort, but it can be also laborious because of potential adoptions needed to ensure desired functionality and reliance on the functioning of the tools.

7.3. Threats to Validity

In this Section, we discuss the threats to validity of our work. In doing so, we distinguish between internal validity and external validity.

7.3.1. Internal Validity

For the internal validity we recognize the threat that the choice of metrics to describe developers' contribution behavior is discretionary and strongly depend on the expectations on the desired model as well as its presentation. For example, one could focus the analysis more on the technical contribution, taking additional factors like LOC written and documentation work into account. Also, focusing on social involvement, co-editing or social ties could be worth investigating. Regardless, based on the related work and the focus of this work we believe that a combination of commit and communication activity can describe the contribution behavior of developers appropriately.

As input for the classification step, we decided to manually classify 20 items of the observation sequence. We are aware that this may introduce a research bias. Other approaches for the role identification like clustering [58] or network measures [153] were not considered in this thesis. This may also have an impact on the misclassification rate and the model parameters. The same issue occurs for the expert labeling in the case of project activity. Other labeling approaches may lead to different results.

The data preprocessing steps are highly relevant for the usefulness of the data. We tried to tackle this by using a combination of automated approaches followed by a manual inspection. Nevertheless, other preprocessing techniques could reduce or increase problems with the data. The same holds true for the classifiers and machine learning techniques used with an impact on the goodness of fit since strongly depending on the data and underlying assumptions on the data.

7.3.2. External Validity

We calculated over 40 individual HMMs and predicted the most likely states for 125 developers of eight open source software projects. These projects differ in their size, background, and team constellation. The insights gained from this study may not be precisely transferable to other project contexts, especially closed source, because of the representativeness of the data. A lot of factors can be different in closed software development, e.g., motivation, effort, attitude, organizational structures, background of developers. Hence, the need for more studies in other project types arises.

Simulation studies raise additional threats to validity to software engineering research. [154] identified possible threats and also published a set of guidelines for simulation based studies in software engineering. One big issue is the output analysis of simulated results. The nature of simulation involves stochastic processes. Thus, every simulation run differs slightly. It is important to figure out the number of simulation runs and parameter sets needed to evaluate the designed simulation model. Moreover, the underlying stochastic needs to be tested and compared carefully. Moreover, the number of assumptions on the simulation model should be moderate to gain comprehensible results. We use repository mining to enrich our model with valid assumptions. The rules implemented are validated individually. Also, we use several simulation runs and parameter settings for every simulation model and check the transferability of results. Nevertheless, we cannot be sure that the ground truth we assume, gained form repository mining, really represents the real world. In the construction of the simulation some model assumptions are made, and thus, other design decisions may yield different results.

8. Conclusion

In this section, we conclude the work done for this thesis. In doing so, we summarize our findings and give an outlook on future work.

8.1. Summary

This thesis presents an approach to describe software evolution and its main factors involving humans, artifacts, and bugs, from the starting point of developers' behavior. Software developers influence the state of the software entities including their quality directly by their daily work, i.e., creating and modifying code, taking part in discussion, and producing and resolving bugs. We investigate different facets of software evolution with the focus on software developers: project involvement dynamics, general software evolution patterns, the impact of developer roles (classification), and project activity. Several vehicles aid our research in these directions: software repository mining, Agent-based modeling and simulation, statistical learning, especially HMMs, and machine learning. For evaluation, we performed three major case studies presenting different methods to assess the overall aim.

First of all, an idea of how software evolution can be modeled and what are important parameters for an Agent-based simulation that describe the software evolution process had to be identified before focusing on the special role of developers within this construct. To pursue this goal, we present an approach that uses a model of software evolution (STEPS) that contains the following agents: developers as active agents performing actions like creations, updates, and deletions on software artifact agents and bug agents that can occur and be fixed (both passive).

The determination of modeling parameters as well as the description of interactions between agents presents the first part of the approach. Although the estimation of model parameters is the more laborious step, the model is equally important since a simulation can only be as good as its model. The main challenge in building a good model is the tradeoff between empirical realism and simplicity [155]. Therefore, simulation models should not be too complex, but also not to superficial. In the proposed simulation model, software changes as driving factors of software evolution are responsible for the system growth and the occurrence of bugs which can be resolved by the developers. Additionally, we differentiate between different types of developers, i.e., core, major, and minor, as well as different kinds of bugs, i.e., major, normal, and minor bugs. Developer collaboration is expressed by co-editing software artifacts. For the evolution of the software structure, links between software entities that are frequently changed together are taken into account. All described parameters need to be examined and their evolution described to feed the simulation.

The parameter estimation to instantiate our simulation models is based on data gathered by mining open source software repositories. In doing so, patterns and heuristics are learned which serve as input for the desired models. The case studies showed the feasibility of Agent-based simulation for software evolution, i.e, they revealed important simulation parameters, produced realistic results, but also demonstrated challenges to respect in the following studies, e.g., temporary inactivity of developers or unsteady project growth patterns.

Since the findings so far showed that the simplistic simulation missed switches in the work and resulting growth patterns, we focused on building a model reflecting developer dynamics. This approach is based on HMMs for developer contribution on different layers, i.e., it considers monthly commits, bug fixes, mailing list posts, and bug comments for individual developers. Based on this observations, a HMM is trained which describes the individual developer behavior. The produced models allows for dynamic behavior in so far that developers can vary their involvement by transitioning different states, i.e., low, medium, and high.

Since the retrieved models were highly correlated for the same type of developers, we build general models as an average of all core, major, and minor models. Furthermore, we demonstrated the application of these in practice in different scenarios: On the one hand, general models can be used to predict the workload of developers given the team constellation. On the other hand, we showed how the general models can be embedded into the simulation tool and, thus allowing dynamic behavior in the simulation as the related case study shows. However, the conducted case study also showed some limitations of the approach: The general models cannot explicitly predict whether a developer will became inactive in the future, it rather describes the general trend and, thus, inactivity is mirrored in low states over a larger period of time. Besides, the simulation is sensitive to the project size and workload of individuals. The latter can be tempered by the dynamic contribution models.

The last presented case study pursues the goal to summarize project activity based on activities done by developers and users and judge whether a project is still active or maintained based on this input. For this, the HMM learning is applied to the project activity data retrieved form the VCS as well as developer and user MLs. To establish a ground truth for project activity, we use an expert labeling grouping the projects in active, under maintenance, and inactive. Again, we build general models to identify group-specific characteristics. The related case study emphasizes the special role of maintenance projects. The difficulty of the determination whether software projects are still maintained gets also supported by a performed crossover study with students as participants. Summarily, we provide some guidelines to identify such projects.

8.2. Outlook

The work performed for this thesis advances the state of the art in simulations of software evolution and developers' contribution behavior. Our research can be used as the basis for multiple further research directions. The current simulation of software processes presented in this thesis focuses on selected software evolution scenarios. Although offering multiple facets of the software development process as well as potential side-effects, there is space for further investigations. On the one hand, the selection as well as the amount of the mined software projects influence the parameter estimation process and, thus, the knowledge that serves the simulation models. In order to maximize the generality of the results, it is required to investigate more projects, especially large size projects. On the other hand, apart from generality, the opposite, i.e., specialty of software processes is an interesting issue. For example, it is an open question whether we can predict special software evolution scenarios like the loss of central contributors, abrupt (periods) of project inactivity, or major design/structural changes with a high certainty. This thesis the problem the other way around by offering the play through of different scenarios by changing parameters.

In the area of characterizing developer behavior, role changes of developers are not explored, e.g., under which circumstances a major developer advance to a core developer. Some deeper explorations to find rules for this phenomenon, especially if they can be deviated from the HMM would be interesting. We already tested the impact of the role classification on the HMMs, but did not include social network analysis for detecting developer roles. For example, the most influential (identified by degree or centrality measures) developers could by seen as the core developers. By the evolution of the network, there are role changes directly visible for every snapshot. Besides, other networks like bug-based collaboration networks [129] could be used as an input for such an analysis.

Moreover, a concrete classifier for project activity could not be learned. Therefore, more data needs to be collected and extracted. An additional idea to deepen this work is to use the gained experiences to provide an alert system which warns the user if the risk is increased that the project will become inactive within the next time.

Bibliography

[1] R. Alfayez, P. Behnamghader, K. Srisopha, and B. Boehm, "How does contributors involvement influence open source systems," in *2017 IEEE 28th Annual Software Technology Conference (STC)*, Sept 2017, pp. 1–8.

[2] Y. Hu, X. Zhang, E. Ngai, R. Cai, and M. Liu, "Software project risk analysis using bayesian networks with causality constraints," *Decision Support Systems*, vol. 56, pp. 439 – 449, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923612003338

[3] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *Proceedings of the 34th Intern.Conf. on Softw. Eng. (ICSE)*. IEEE, 2012.

[4] V. Honsel, D. Honsel, S. Herbold, J. Grabowski, and S. Waack, "Mining software dependency networks for agent-based simulation of software evolution." The Fourth International Workshop on Software Mining, 2015.

[5] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger, "Analysing software repositories to understand software evolution," in *Software Evolution*. Springer Berlin Heidelberg, 2008, pp. 37–67.

[6] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi, "Empirical studies of open source evolution," in *Software Evolution: State-of-the-art and research advances*, T. Mens and S. Demeyer, Eds. Springer Verlag, 2008, ch. 11, pp. 263–288.

[7] J. Lima, C. Treude, F. F. Filho, and U. Kulesza, "Assessing developer contribution with repository mining-based metrics," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, Sept 2015, pp. 536–540.

[8] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, ser. IWPSE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 113–122.

[9] X. Ben, S. Beijun, and Y. Weicheng, "Mining developer contribution in open source software using visualization techniques," in *Proceedings of the Third International*

Conference on Intelligent System Design and Engineering Applications (ISDEA), 2013, pp. 934–937.

[10] V. Honsel, D. Honsel, and J. Grabowski, “Software process simulation based on mining software repositories.” The Third International Workshop on Software Mining, 2014.

[11] V. Honsel, “Statistical learning and software mining for agent based simulation of software evolution,” in *Doctoral Symposium at the 37th International Conference on Software Engineering (ICSE)*, 2015.

[12] V. Honsel, S. Herbold, and J. Grabowski, “Hidden markov models for the prediction of developer involvement dynamics and workload,” in *12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, 2016.

[13] M. M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.

[14] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, “Metrics and laws of software evolution - the nineties view,” in *Proceedings of the 4th International Symposium on Software Metrics*, ser. METRICS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–.

[15] W. W. Royce, “Managing the development of large software systems: concepts and techniques,” *Proc. IEEE WESTCON, Los Angeles*, pp. 1–9, August 1970, reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338. [Online]. Available: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

[16] M.-W. Online, “Merriam-webster online dictionary,” 2016. [Online]. Available: http://www.merriam-webster.com

[17] M. Lehman and J. Ramil, “Towards a theory of software evolution - and its practical impact (working paper),” in *Invited Talk, Proceedings Intl. Symposium on Principles of Softw. Evolution, ISPSE 2000, 1-2 Nov*. Press, 2000, pp. 2–11.

[18] M. M. Lehman, G. Kahen, and J. F. Ramil, “Behavioural modelling of long-lived evolution processes: Some issues and an example,” *Journal of Software Maintenance*, vol. 14, no. 5, pp. 335–351, Sep. 2002. [Online]. Available: http://dx.doi.org/10.1002/smr.259

[19] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, “The missing links: bugs and bug-fix commits.” in *SIGSOFT FSE*, G.-C. Roman and K. J. Sullivan, Eds. ACM, 2010, pp. 97–106.

[20] N. Bettenburg, E. Shihab, and A. E. Hassan, "An empirical study on the risks of using off-the-shelf techniques for processing mailing list data," in *2009 IEEE International Conference on Software Maintenance*, Sept 2009, pp. 539–542.

[21] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey, "The msr cookbook: mining a decade of research." in *MSR*, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE Computer Society, 2013, pp. 343–352. [Online]. Available: http://dblp.uni-trier.de/db/conf/msr/msr2013.html#HemmatiNBKWHG13

[22] H. Kagdi, M. Collard, and J. I. Maletic, "Towards a taxonomy of approaches for mining of source code repositories," vol. 30, pp. 1–5, 07 2005.

[23] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*. Wiley, 1994.

[24] N. B. Ruparelia, "The history of version control," *SIGSOFT Softw. Eng. Notes*, vol. 35, no. 1, pp. 5–9, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1668862.1668876

[25] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, ser. MSR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSR.2009.5069475

[26] K. Ayari, P. Meshkinfam, G. Antoniol, and M. D. Penta, "Threats on building models from cvs and bugzilla repositories: the mozilla case study," in *CASCON*, 2007.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, ser. MSR '05. New York, NY, USA: ACM, 2005, pp. 1–5. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[28] M. Goeminne and T. Mens, "A comparison of identity merge algorithms for software repositories," *Science of Computer Programming*, vol. 78, no. 8, pp. 971 – 986, 2013, special section on software evolution, adaptability, and maintenance & Special section on the Brazilian Symposium on Programming Languages. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642311002048

[29] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486844

[30] F. Trautsch, S. Herbold, P. Makedonski, and J. Grabowski, “Addressing problems with replicability and validity of repository mining studies through a smart data platform,” *Empirical Softw. Engg.*, vol. 23, no. 2, pp. 1036–1083, Apr. 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9537-x

[31] T. Zimmermann and P. Weißgerber, “Preprocessing cvs data for fine-grained analysis,” in *In MSR ’04: Proceedings of the 1st International Workshop on Mining Software Repositories*. IEEE Computer Society, 2004, pp. 2–6.

[32] I. H. Witten, E. Frank, and M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[33] B. Leventhal, “An introduction to data mining and other techniques for advanced analytics,” *Journal of Direct, Data and Digital Marketing Practice*, vol. 12, no. 2, pp. 137–153, Oct 2010. [Online]. Available: https://doi.org/10.1057/dddmp.2010.35

[34] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[35] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[36] M. J. Kearns, U. V. Vazirani, and U. Vazirani, *An introduction to computational learning theory*, 1994.

[37] R. P. L. Buse and T. Zimmermann, “Information needs for software development analytics,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 987–996. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337343

[38] D. Zhang, S. Han, Y. Dang, J.-G. Lou, H. Zhang, and T. Xie, “Software analytics in practice,” *IEEE Softw.*, vol. 30, no. 5, pp. 30–37, Sep. 2013. [Online]. Available: http://dx.doi.org/10.1109/MS.2013.94

[39] W. M. Turski, “Reference model for smooth growth of software systems,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 599–600, Aug. 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=235681.235686

[40] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, “Validating the use of topic models for software evolution,” in *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, Sept 2010, pp. 55–64.

[41] J. Hao and E. Mendes, “Usage-based statistical testing of web applications,” in *Proceedings of the 6th International Conference on Web Engineering*, ser.

ICWE '06. New York, NY, USA: ACM, 2006, pp. 17–24. [Online]. Available: http://doi.acm.org/10.1145/1145581.1145585

[42] J. Banks, *Handbook of simulation : principles, methodology, advances, applications, and practice*, ser. A Wiley-Interscience publication. New York: J. Wiley & sons, 1998. [Online]. Available: http://opac.inria.fr/record=b1094393

[43] R. Maidstone, "Discrete event simulation, system dynamics and agent based simulation: Discussion and comparison," 2012.

[44] A. Borshchev and A. Filippov, "From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools," in *The 22nd International Conference of the System Dynamics Society*, Jul. 2004.

[45] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation," in *Proceedings of the 37th Conference on Winter Simulation*, ser. WSC '05. Winter Simulation Conference, 2005, pp. 2–15.

[46] A. S. Rao and M. P. Georgeff, "Bdi agents: From theory to practice," in *Proceedings of the first International Conference on Multi-Agent Systems (ICMAS)*, 1995, pp. 312–319.

[47] C. M. Macal and M. J. North, "Agent-based modeling and simulation," in *Winter Simulation Conference*, ser. WSC '09. Winter Simulation Conference, 2009, pp. 86–98.

[48] C. Macal and M. North, "Introductory tutorial: Agent-based modeling and simulation," in *Proceedings of the 2014 Winter Simulation Conference*. IEEE Press, 2014, pp. 6–20.

[49] S. Tisue and U. Wilensky, "Netlogo: A simple environment for modeling complexity," in *in International Conference on Complex Systems*, 2004, pp. 16–21.

[50] M. North, "The repast suite," online. [Online]. Available: http://repast.sourceforge.net/

[51] A. Drogoul, E. Amouroux, P. Caillou, B. Gaudou, A. Grignard, N. Marilleau, P. Taillandier, M. Vavaseur, D.-A. Vo, and J.-D. Zucker, "GAMA: multi-level and complex environment for agent-based models and simulations (demonstration)," in *international conference on Autonomous agents and multi-agent systems*, United States, 2013, pp. 1361–1362. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00834498

[52] M. I. Kellner, R. J. Madachy, and D. M. Raffo, "Software process simulation modeling: Why? what? how?" *Journal of Systems and Software*, vol. 46, no. 2–3,

pp. 91 – 105, 1999. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0164121299000035

[53] H. Zhang, B. Kitchenham, and D. Pfahl, *Software Process Simulation Modeling: An Extended Systematic Review*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 309–320. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14347-2_27

[54] S. Athey and G. Ellison, "Dynamics of open source movements," *Journal of Economics & Management Strategy*, vol. 23, no. 2, pp. 294–316, 2014. [Online]. Available: http://dx.doi.org/10.1111/jems.12053

[55] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM, 2008.

[56] A. Meneely and L. Williams, "Socio-technical developer networks: should we trust our measurements?" in *ICSE*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 281–290. [Online]. Available: http://dblp.uni-trier.de/db/conf/icse/icse2011.html#MeneelyW11

[57] P. Bhattacharya, I. Neamtiu, and M. Faloutsos, "Determining developers' expertise and role: A graph hierarchy-based approach." in *ICSME*. IEEE Computer Society, 2014, pp. 11–20.

[58] L. Yu and S. Ramaswamy, "Mining cvs repositories to understand open-source project developer roles," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 8–11.

[59] M. Foucault, M. Palyart, X. Blanc, G. C. Murphy, and J.-R. Falleri, "Impact of developer turnover on quality in open-source software," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 829–841.

[60] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb, "Configuring global software teams: A multi-company analysis of project productivity, quality, and profits," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 261–270. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985830

[61] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 4–14. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025119

[62] A. Iqbal, M. Karnstedt, and M. Hausenblas, “Analyzing social behavior of software developers across different communication channels (s).” in *SEKE*. Knowledge Systems Institute Graduate School, 2013, pp. 113–118. [Online]. Available: http://dblp.uni-trier.de/db/conf/seke/seke2013.html#IqbalKH13

[63] L. V. Galvis Carreño and K. Winbladh, “Analysis of user comments: An approach for software requirements evolution,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 582–591. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486865

[64] C. Subramaniam, R. Sen, and M. L. Nelson, “Determinants of open source software project success: A longitudinal study,” *Decision Support Systems*, vol. 46, no. 2, pp. 576 – 585, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167923608001814

[65] R. Sen, S. S. Singh, and S. Borle, “Open source software success: Measures and analysis,” *Decision Support Systems*, vol. 52, no. 2, pp. 364 – 372, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016792361100159X

[66] J. Coelho, M. T. Valente, L. L. Silva, and E. Shihab, “Identifying unmaintained projects in github,” in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM ’18. New York, NY, USA: ACM, 2018, pp. 15:1–15:10. [Online]. Available: http://doi.acm.org/10.1145/3239235.3240501

[67] J. Khondhu, A. Capiluppi, and K.-J. Stol, “Is it all lost? a study of inactive open source projects,” in *Open Source Software: Quality Verification*, E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 61–79.

[68] L. R. Rabiner and B. H. Juang, “An introduction to hidden markov models,” *IEEE ASSp Magazine*, 1986.

[69] A. B. Poritz, “Hidden Markov models: a guided tour,” in *International Conference on Acoustics, Speech, and Signal Processing*, vol. 1, 1988, pp. 7–13. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.1988.196495

[70] S. Vegas, C. Apa, and N. Juristo, “Crossover designs in software engineering experiments: Benefits and perils,” *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 120–135, Feb 2016.

[71] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, and G. Tortora, “On the impact of uml analysis models on source-code comprehensibility and modifiability,”

ACM Trans. Softw. Eng. Methodol., vol. 23, no. 2, pp. 13:1–13:26, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2491912

[72] L. Madeyski and B. Kitchenham, "Effect sizes and their variance for ab/ba crossover design studies," *Empirical Software Engineering*, vol. 23, no. 4, pp. 1982–2017, Aug 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9574-5

[73] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

[74] H. B. Mann, "Nonparametric tests against trend," *Econometrica: Journal of the Econometric Society*, pp. 245–259, 1945.

[75] T. Mens and S. Demeyer, *Software Evolution*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[76] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Proc. Int'l Conf. Software Maintenance (ICSM).* Los Alamitos, California: IEEE Computer Society Press, 2000, pp. 131–142.

[77] J. W. Paulson, G. Succi, and A. Eberlein, "An empirical study of open-source and closed-source software products," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 246–256, Apr. 2004. [Online]. Available: https://doi.org/10.1109/TSE.2004.1274044

[78] I. Herraiz, G. Robles, and J. u. M. Gonzalez-Barahon, "Comparison between slocs and number of files as size metrics for software evolution analysis," in *Proceedings of the Conference on Software Maintenance and Reengineering*, ser. CSMR '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 206–213. [Online]. Available: http://dl.acm.org/citation.cfm?id=1116163.1116405

[79] A. Capiluppi and J. F. Ramil, "Studying the evolution of open source systems at different levels of granularity: two case studies," in *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.*, Sept 2004, pp. 113–118.

[80] S.-K. Huang and K.-m. Liu, "Mining version histories to verify the learning process of legitimate peripheral participants," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083158

[81] A. Meneely, L. Williams, W. Snipes, and J. Osborne, "Predicting failures with developer networks and social network analysis," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 13–23. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453106

[82] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, “Mining email social networks,” in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR ’06. New York, NY, USA: ACM, 2006, pp. 137–143. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138016

[83] M. Pinzger, N. Nagappan, and B. Murphy, “Can developer-module networks predict failures?” in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT ’08/FSE-16. New York, NY, USA: ACM, 2008, pp. 2–12.

[84] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy, “If your version control system could talk,” in *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[85] M. D’Ambros, M. Lanza, and R. Robbes, “On the relationship between change coupling and software defects,” in *Proc. of the 16th Working Conf. on Rev. Eng.* IEEE Computer Society, 2009.

[86] P. Knab, M. Pinzger, and A. Bernstein, “Predicting defect densities in source code files with decision tree learners,” in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR ’06. New York, NY, USA: ACM, 2006, pp. 119–125. [Online]. Available: http://doi.acm.org/10.1145/1137983.1138012

[87] Y. Zhou, M. Würsch, E. Giger, H. C. Gall, and J. Lü, “A bayesian network based approach for change coupling prediction,” in *2008 15th Working Conference on Reverse Engineering*, Oct 2008, pp. 27–36.

[88] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, and M. A. Gerosa, “An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project,” in *Open Source Systems: Adoption and Impact*, E. Damiani, F. Frati, D. Riehle, and A. I. Wasserman, Eds. Cham: Springer International Publishing, 2015, pp. 3–12.

[89] C. Tantithamthavorn, A. Ihara, and K. ichi Matsumoto, “Using co-change histories to improve bug localization performance,” *2013 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 543–548, 2013.

[90] J. Anvik, L. Hiew, and G. C. Murphy, “Who should fix this bug?” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 361–370.

[91] S. Kim, E. J. Whitehead, and Y. Zhang, “Classifying Software Changes: Clean or Buggy?” *Software Engineering, IEEE Transactions on*, vol. 34, no. 2, pp. 181–196, Mar. 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2007.70773

[92] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, “Predicting the severity of a reported bug,” in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, May 2010, pp. 1–10.

[93] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, “How long will it take to fix this bug?” in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR ’07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.13

[94] M. Gharehyazie, D. Posnett, and V. Filkov, “Social activities rival patch submission for prediction of developer initiation in oss projects.” in *ICSM*. IEEE Computer Society, 2013, pp. 340–349.

[95] G. Gousios, E. Kalliamvakou, and D. Spinellis, “Measuring developer contribution from software repository data,” in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR ’08. New York, NY, USA: ACM, 2008, pp. 129–132.

[96] P. Makedonski, “Developer-Centric Software Assessment,” Ph.D. dissertation, Göttingen, Germany, Jun. 2018.

[97] K. Crowston, K. Wei, Q. Li, and J. Howison, “Core and periphery in free/libre and open source software team communications,” in *Proceedings of the 39th Annual Hawaii International Conference on System Sciences - Volume 06*, ser. HICSS ’06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 118.1–. [Online]. Available: https://doi.org/10.1109/HICSS.2006.101

[98] C. Amrit and J. van Hillegersberg, “Exploring the impact of socio-technical core-periphery structures in open source software development,” *Journal of Information Technology*, vol. 25, no. 2, pp. 216–229, Jun 2010. [Online]. Available: https://doi.org/10.1057/jit.2010.7

[99] A. Terceiro, L. R. Rios, and C. Chavez, “An empirical study on the structural complexity introduced by core and peripheral developers in free software projects,” in *Software Engineering (SBES), 2010 Brazilian Symposium on*. IEEE, 2010, pp. 21–29.

[100] K. Crowston and J. Howison, “The social structure of free and open source software development,” *First Monday*, vol. 10, no. 2, 2005.

[101] G. Robles, J. M. Gonzalez-Barahona, and I. Herraiz, “Evolution of the core team of developers in libre software projects,” in *2009 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 167–170.

[102] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer, “Classifying developers into core and peripheral: An empirical study on count and network metrics,” in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE ’17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 164–174. [Online]. Available: https://doi.org/10.1109/ICSE.2017.23

[103] P. V. Singh, Y. Tan, and N. Youn, “A hidden markov model of developer learning dynamics in open source software projects.” *Information Systems Research*, vol. 22, no. 4, pp. 790–807, 2011.

[104] Y. Gao and G. Madey, “Towards understanding: A study of the sourceforge.net community using modeling and simulation,” in *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*, ser. SpringSim ’07. San Diego, CA, USA: Society for Computer Simulation International, 2007, pp. 145–150.

[105] N. Minar, R. Burkhart, C. Langton, M. Askenazi *et al.*, “The swarm simulation system: A toolkit for building multi-agent simulations,” 1996.

[106] B. Spasic and B. S. S. Onggo, “Agent-based simulation of the software development process: a case study at avl.” in *Winter Simulation Conference*, O. Rose and A. M. Uhrmacher, Eds. WSC, 2012, pp. 400:1–400:11. [Online]. Available: http://dblp.uni-trier.de/db/conf/wsc/wsc2012.html#SpasicO12

[107] M. B. Chrissis, M. Konrad, and S. Shrum, *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Addison-Wesley Professional, 2011.

[108] R. Cherif and P. Davidsson, “Software development process simulation: Multi agent-based simulation versus system dynamics,” in *Multi-Agent-Based Simulation X*, G. Di Tosto and H. Van Dyke Parunak, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 73–85.

[109] R. Agarwal and D. Umphress, “A flexible model for simulation of software development process,” in *Proceedings of the 48th Annual Southeast Regional Conference*. ACM, 2010, p. 40.

[110] N. Smith and J. F. Ramil, “Agent-based simulation of open source evolution,” in *Software Process Improvement and Practice*, 2006.

[111] U. Abelein and B. Paech, “Understanding the influence of user participation and involvement on system success – a systematic mapping study,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 28–81, Feb 2015. [Online]. Available: https://doi.org/10.1007/s10664-013-9278-4

[112] D. Honsel, V. Herbold, M. Welter, J. Grabowski, and S. Waack, "Monitoring software quality by means of simulation methods," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. ACM, 2016, pp. 11:1–11:6. [Online]. Available: http://doi.acm.org/10.1145/2961111.2962617

[113] M. Welter, D. Honsel, V. Herbold, A. Staedler, J. Grabowski, and S. Waack, "Assessing Simulated Software Graphs using Conditional Random Fields," in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, ser. Communications in Computer and Information Science (CCIS). Springer, 2018.

[114] L. Hattori and M. Lanza, "On the nature of commits." in *ASE Workshops*. IEEE, 2008, pp. 63–71.

[115] J. Long, "Understanding the role of core developers in open source software development," *Journal of Information, Information Technology, and Organizations*, vol. 1, pp. 75–85, 2006.

[116] H. Xia, "A collective-intelligence view on the linux kernel developer community," *Int. J. Knowl. Syst. Sci.*, vol. 1, no. 3, pp. 20–32, Jul. 2010. [Online]. Available: http://dx.doi.org/10.4018/jkss.2010070102

[117] F. Rahman and P. Devanbu, "Ownership, experience and defects: A fine-grained study of authorship," in *Proc. of the 33rd Intern. Conf. on Softw. Eng. (ICSE)*. ACM, 2011.

[118] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 298–308. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070530

[119] S. Fortunato, "Community detection in graphs," *Physics Reports*, vol. 486, no. 3-5, pp. 75 – 174, 2010.

[120] R. J. Wilson, *Introduction to Graph Theory*. New York, NY, USA: John Wiley & Sons, Inc., 1986.

[121] N. Balakrishnan and V. Nevzorov, *A Primer on Statistical Distributions*. Wiley, 2003. [Online]. Available: https://books.google.de/books?id=aOC5lwEACAAJ

[122] D. Honsel, N. Fiekas, V. Herbold, M. Welter, T. Ahlbrecht, S. Waack, J. Dix, and J. Grabowski, "Simulating Software Refactorings based on Graph Transformations,"

in *Post-Proceedings of the Clausthal-Göttingen International Workshop on Simulation Science 2017*, ser. Communications in Computer and Information Science (CCIS). Springer, 2018.

[123] C. M. Macal and M. J. North, “Tutorial on agent-based modelling and simulation,” *Journal of Simulation*, vol. 4, no. 3, pp. 151–162, 2010.

[124] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, “The weka data mining software: an update,” *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[125] H. Gong, H. Zhang, D. Yu, and B. Liu, “A systematic map on verifying and validating software process simulation models,” in *Proceedings of the 2017 International Conference on Software and System Process*, ser. ICSSP 2017. New York, NY, USA: ACM, 2017, pp. 50–59. [Online]. Available: http://doi.acm.org/10.1145/3084100.3084106

[126] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 23–32.

[127] C. G. Campos, “Cvsanaly,” 2014. [Online]. Available: http://metricsgrimoire.github.io/CVSAnalY/

[128] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, “Rclinker: Automated linking of issue reports and commits leveraging rich contextual information,” in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ser. ICPC ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 36–47. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820282.2820290

[129] Q. Hong, S. Kim, S. C. Cheung, and C. Bird, “Understanding a developer social network and its evolution.” in *ICSM*. IEEE, 2011, pp. 323–332.

[130] M. Bastian, S. Heymann, M. Jacomy *et al.*, “Gephi: an open source software for exploring and manipulating networks.” in *Proc. of the 3rd Intern. AAAI Conf. on Weblogs and Social Media (ICWSM)*, 2009.

[131] T. Opsahl, F. Agneessens, and J. Skvoretz, “Node centrality in weighted networks: Generalizing degree and shortest paths,” *Social Networks*, vol. 32, no. 3, pp. 245 – 251, 2010.

[132] J. Leskovec, K. J. Lang, and M. Mahoney, “Empirical comparison of algorithms for network community detection,” in *Proceedings of the 19th International Conference*

on World Wide Web, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 631–640. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772755

[133] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2008, no. 10, p. P10008, 2008. [Online]. Available: http://stacks.iop.org/1742-5468/2008/i=10/a=P10008

[134] L. Xiaolin, M. Parizeau, and R. Plamondon, "Training hidden markov models with multiple observations-a combinatorial method." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 4, pp. 371–377, 2000.

[135] S. Herbold, J. Grabowski, and S. Waack, "Calculation and optimization of thresholds for sets of software metrics," *Empirical Software Engineering*, vol. 16, no. 6, pp. 812–841, 2011.

[136] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[137] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1010933404324

[138] R. Caruana and A. Niculescu-Mizil, "An empirical comparison of supervised learning algorithms," in *Proceedings of the 23rd International Conference on Machine Learning*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 161–168. [Online]. Available: http://doi.acm.org/10.1145/1143844.1143865

[139] T. van Gestel, J. Suykens, B. Baesens, S. Viaene, J. Vanthienen, G. Dedene, B. de Moor, and J. Vandewalle, "Benchmarking Least Squares Support Vector Machine Classifiers," *Machine Learning*, vol. 54, no. 1, pp. 5–32, 2004.

[140] J. O'Connell and S. Højsgaard, "Hidden semi markov models for multiple observation sequences: The mhsmm package for R," *Journal of Statistical Software*, vol. 39, no. 4, pp. 1–22, 2011. [Online]. Available: http://www.jstatsoft.org/v39/i04/

[141] M. Taboga, *Lectures on probability theory and mathematical statistics*. CreateSpace Independent Pub., 2012.

[142] W. D. Berry, W. D. Berry, S. Feldman, and D. Stanley Feldman, *Multiple regression in practice*. Sage, 1985, no. 50.

[143] Y. Tymchuk, A. Mocci, and M. Lanza, "Collaboration in open-source projects: Myth or reality?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 304–307.

[144] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "The rise and fall of a central contributor: Dynamics of social organization and performance in the gentoo community," *CoRR*, vol. abs/1302.7191, 2013.

[145] M. Hollander and D. A. Wolfe, *Nonparametric Statistical Methods, 2nd Edition*, 2nd ed. Wiley-Interscience, Jan. 1999. [Online]. Available: http://www.worldcat.org/isbn/0471190454

[146] R. G. Sargent, "Verification and validation of simulation models," in *Proceedings of the Winter Simulation Conference*, ser. WSC '11. Winter Simulation Conference, 2011, pp. 183–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=2431518.2431538

[147] M. Uzzafer, "A simulation model for strategic management process of software projects," *Journal of Systems and Software*, vol. 86, no. 1, pp. 21 – 37, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016412121200177X

[148] A. Guzzi, A. Bacchelli, M. Lanza, M. Pinzger, and A. v. Deursen, "Communication in open source software development mailing lists," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 277–286. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487139

[149] J. Khondhu, A. Capiluppi, and K.-J. Stol, "Is it all lost? a study of inactive open source projects," in *Open Source Software: Quality Verification*, E. Petrinja, G. Succi, N. El Ioini, and A. Sillitti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 61–79.

[150] S. Thompson, *Sampling*, ser. CourseSmart. Wiley, 2012. [Online]. Available: https://books.google.de/books?id=-sFtXLIdDiIC

[151] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, May 2015, pp. 666–676.

[152] D. Bates, M. Mächler, B. Bolker, and S. Walker, "Fitting linear mixed-effects models using lme4," *Journal of Statistical Software*, vol. 67, no. 1, pp. 1–48, 2015.

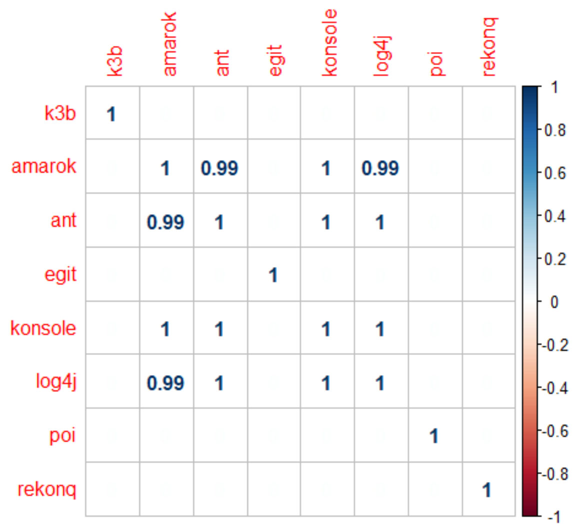
[153] M. Pohl and S. Diehl, "What dynamic network metrics can tell us about developer roles," in *Proceedings of the 2008 International Workshop on Cooperative and Human Aspects of Software Engineering*, ser. CHASE '08. New York, NY, USA: ACM, 2008, pp. 81–84.

[154] B. B. N. de França and G. H. Travassos, “Experimentation with dynamic simulation models in software engineering: planning and reporting guidelines,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1302–1345, Jun 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9386-4

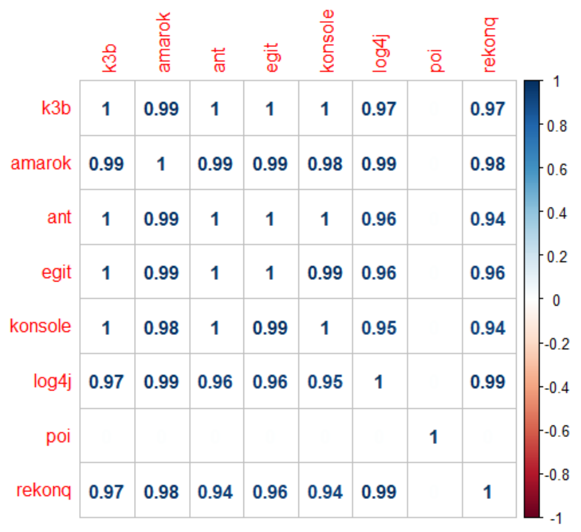
[155] A. Maria, “Introduction to modeling and simulation,” in *Proceedings of the 29th conference on Winter simulation*. IEEE Computer Society, 1997, pp. 7–13.

A. Correlations of Transition Matrices of Individual Developer Behavior

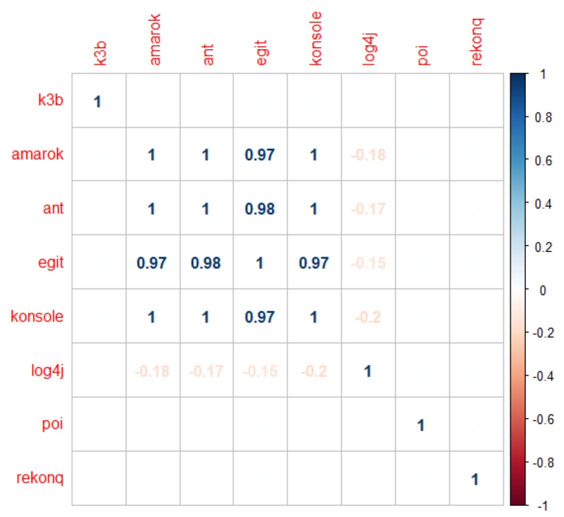
In the following, we list all project-wise correlations for the transitions matrices of developers sharing the same role.



(a) Core developers

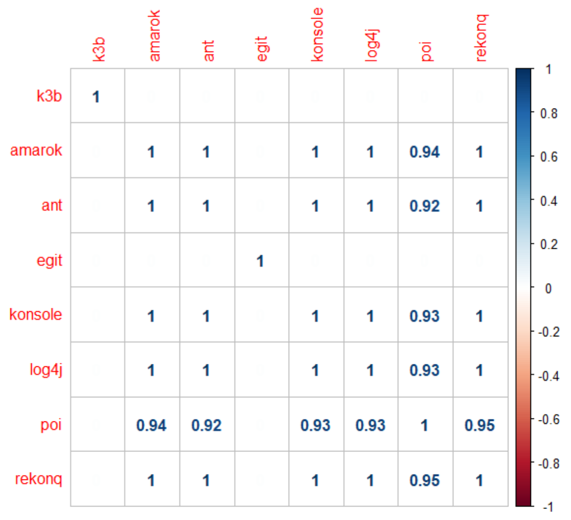


(b) Major developers

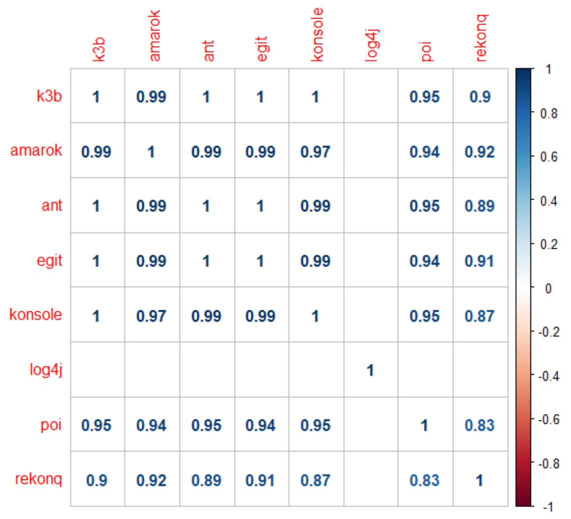


(c) Minor developers

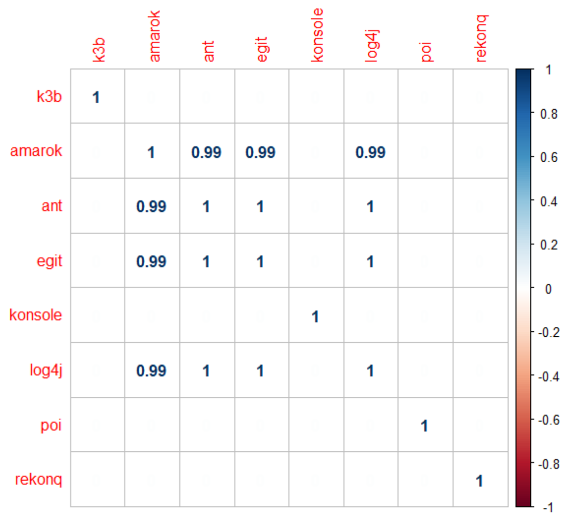
Figure A.1.: Role-wise correlation plots for all individual developer models using thresholds as classifier.



(a) Core developers

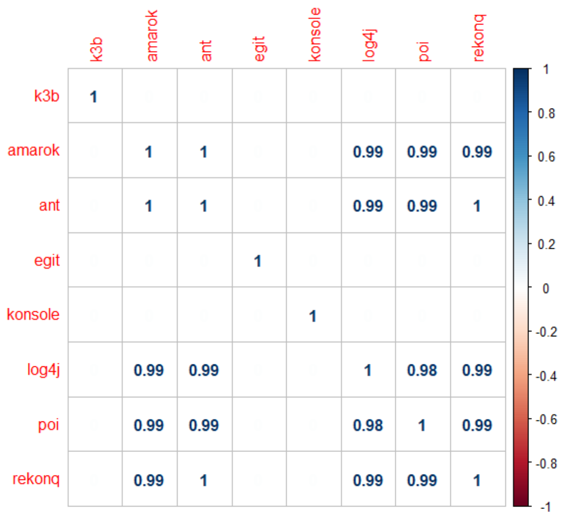


(b) Major developers

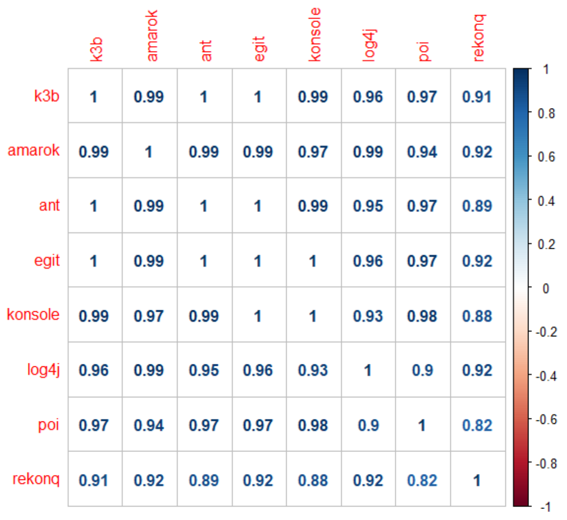


(c) Minor developers

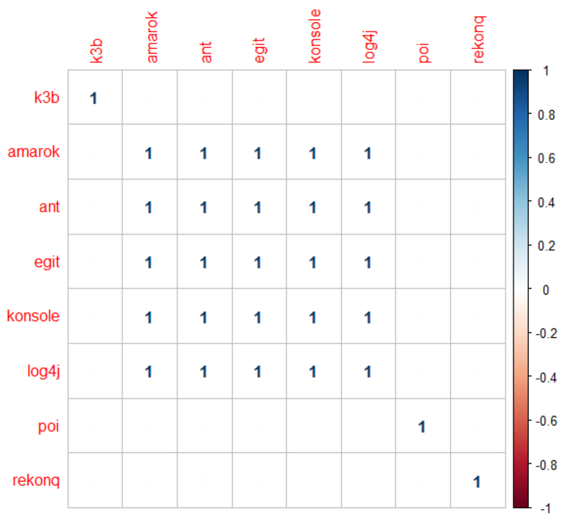
Figure A.2.: Role-wise correlation plots for all individual developer models using KNN as classifier.



(a) Core developers

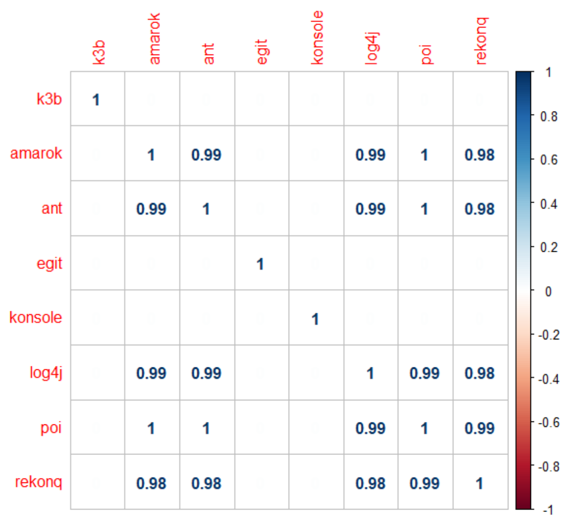


(b) Major developers

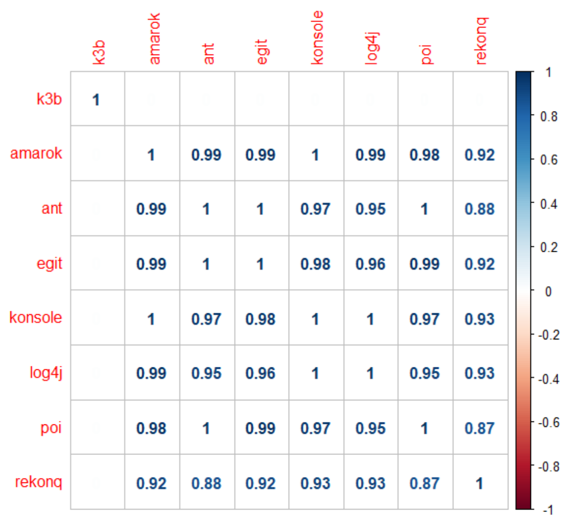


(c) Minor developers

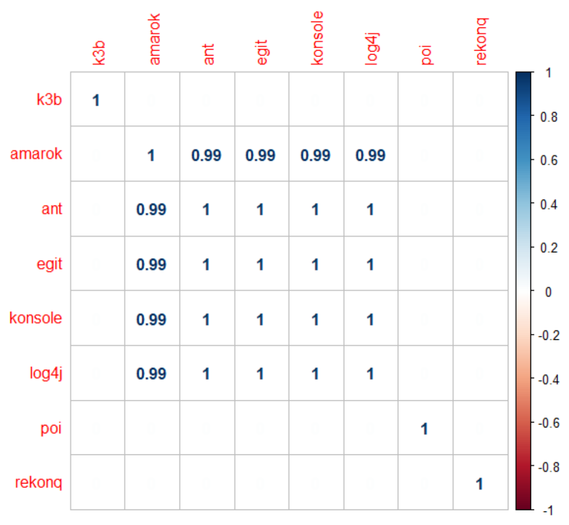
Figure A.3.: Role-wise correlation plots for all individual developer models using KNN3 as classifier.



(a) Core developers

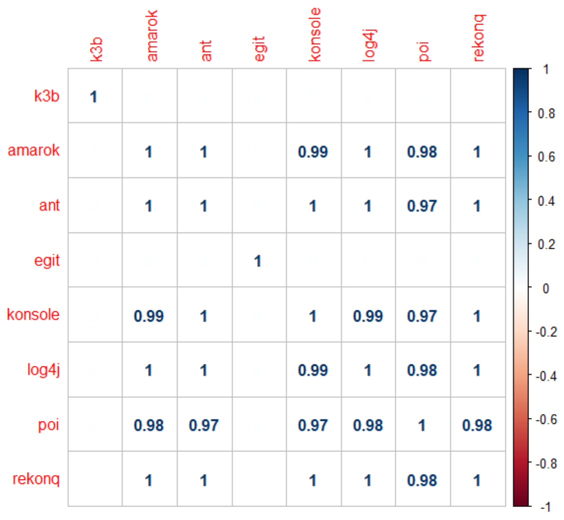


(b) Major developers

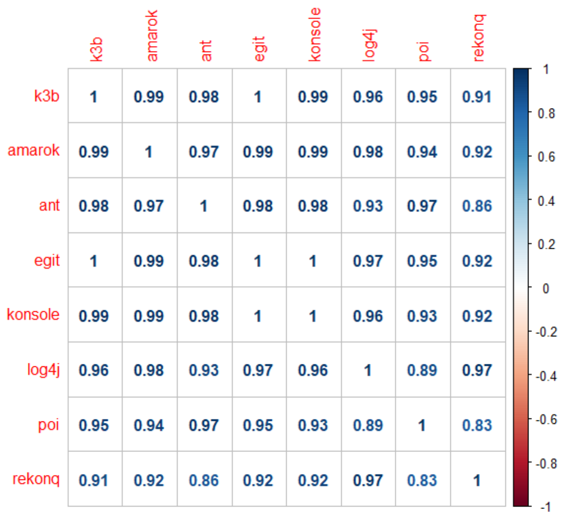


(c) Minor developers

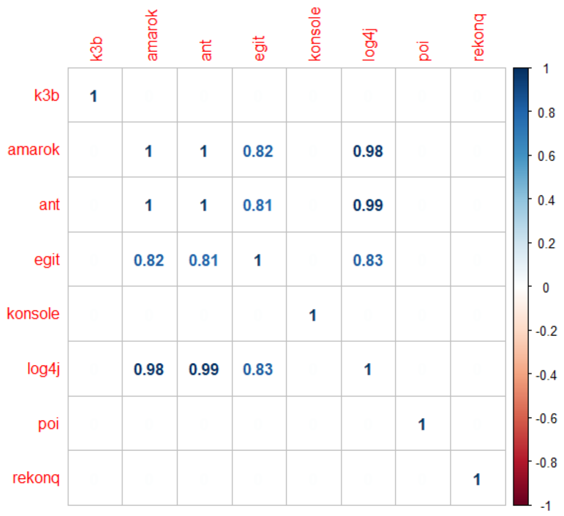
Figure A.4.: Role-wise correlation plots for all individual developer models using KNN5 as classifier.



(a) Core developers



(b) Major developers



(c) Minor developers

Figure A.5.: Role-wise correlation plots for all individual developer models using Random Forests as classifier.

B. Hidden Markov Models for Universal Developer Contribution Behavior

In the following, we list all produced general contribution models that are not taken as an example within the thesis. This means, we report for each classifier the transition matrices, means of the emissions, and covariance matrices of emissions.

KNN5

For KNN5, the transitions and means of emissions were already mentioned. Thus, we start with covariance matrices Σ_k belonging to the means μ_k in Table 6.6. Together, this specifies the emission distributions $\mathscr{N}(k;\mu_k,\Sigma_k)$.

We start with the general model for core developers:

$$\Sigma_{low} = \begin{pmatrix} 16.060 & 1.573 & -5.583 & -1.568 \\ 1.572 & 0.906 & -0.969 & 0.096 \\ -5.583 & -0.969 & 19.531 & 4.663 \\ -1.568 & 0.096 & 4.663 & 1.927 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 65.503 & 21.913 & -43.114 & -38.509 \\ 21.913 & 18.879 & 9.771 & -14.291 \\ -43.114 & 9.771 & 149.776 & 13.233 \\ -38.509 & -14.291 & 13.233 & 73.591 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 59.347 & 34.465 & 56.746 & -37.852 \\ 34.465 & 35.317 & 29.253 & -39.568 \\ 56.746 & 29.253 & 88.749 & -62.445 \\ -37.852 & -39.568 & -62.445 & 339.419 \end{pmatrix}.$$

Figure B.1.: Covariance matrices of the core developer general model.

For the major developers, we get the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.183 & 0.064 & 0.340 & 0.199 \\ 0.064 & 0.097 & 0.136 & 0.042 \\ 0.340 & 0.136 & 1.902 & 0.506 \\ 0.199 & 0.042 & 0.506 & 3.965 \end{pmatrix},$$
$$\Sigma_{medium} = \begin{pmatrix} 5.682 & 1.380 & 1.069 & -0.842 \\ 1.380 & 1.314 & -0.102 & -0.366 \\ 1.069 & -0.102 & 7.957 & -10.388 \\ -0.842 & -0.366 & -10.388 & 48.673 \end{pmatrix},$$
$$\Sigma_{high} = \begin{pmatrix} 0.843 & -0.045 & 0.188 & -0.189 \\ -0.045 & 0.498 & -0.237 & 1.581 \\ 0.188 & -0.237 & 0.376 & -0.623 \\ -0.189 & 1.581 & -0.623 & 7.537-0.623 \end{pmatrix}.$$

Figure B.2.: Covariance matrices of the major developer general model.

Finally, the covariances of minor developers can be expressed by these matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.120 & 0.018 & 0.029 & 0.036 \\ 0.018 & 0.031 & -0.010 & -0.027 \\ 0.029 & -0.010 & 0.287 & -0.077 \\ 0.036 & -0.027 & -0.077 & 0.294 \end{pmatrix},$$
$$\Sigma_{medium} = \begin{pmatrix} 1.174 & 0.093 & 0.116 & 6.432 \\ 0.093 & 0.076 & 0.029 & 0.655 \\ 0.116 & 0.029 & 1.017 & 0.858 \\ 6.432 & 0.655 & 0.858 & 48.106 \end{pmatrix},$$
$$\Sigma_{high} = \begin{pmatrix} 0.020 & 0.006 & 0.001 & 0.015 \\ 0.006 & 0.003 & 0.000 & 0.009 \\ 0.001 & 0.000 & 0.000 & -0.000 \\ 0.015 & 0.009 & -0.000 & 0.064 \end{pmatrix}.$$

Figure B.3.: Covariance matrices of the minor developer general model.

Thresholds

$$A_{core} = \begin{array}{c} \\ low \\ med. \\ high \end{array}\begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.65 & 0.28 & 0.07 \\ 0.39 & 0.48 & 0.13 \\ 0.09 & 0.22 & 0.69 \end{pmatrix} \end{array}, \quad A_{major} = \begin{array}{c} \\ low \\ med. \\ high \end{array}\begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.27 & 0.06 \\ 0.37 & 0.53 & 0.10 \\ 0.07 & 0.17 & 0.76 \end{pmatrix} \end{array},$$

$$A_{minor} = \begin{array}{c} \\ low \\ med. \\ high \end{array}\begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.70 & 0.24 & 0.06 \\ 0.30 & 0.51 & 0.19 \\ 0.09 & 0.16 & 0.75 \end{pmatrix} \end{array}.$$

Figure B.4.: General transition matrices for developer roles over all projects.

State	Core	Major	Minor
low	$(1,0,5,1)$	$(1,0,2,1)$	$(1,0,1,1)$
medium	$(22,6,27,13)$	$(12,3,6,11)$	$(7,3,2,12)$
high	$(33,11,50,45)$	$(13,4,11,26)$	$(3,1,4,5)$

Table B.1.: Means μ_k of emissions for the general model.

We start with the general model for core developers:

$$\Sigma_{low} = \begin{pmatrix} 0.216 & 0.031 & 0.447 & 0.141 \\ 0.031 & 0.018 & 0.082 & 0.009 \\ 0.447 & 0.082 & 8.074 & 0.784 \\ 0.141 & 0.009 & 0.784 & 0.740 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 22.525 & 1.980 & -5.432 & -1.370 \\ 1.980 & 2.578 & -13.656 & -0.361 \\ -5.432 & -13.656 & 997.357 & -20.015 \\ -1.370 & -0.361 & -20.015 & 5.054 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 86.289 & 10.060 & 94.972 & 18.876 \\ 10.060 & 8.928 & 10.462 & 14.923 \\ 94.972 & 10.462 & 230.876 & 10.453 \\ 18.876 & 14.923 & 10.453 & 110.327 \end{pmatrix}.$$

Figure B.5.: Covariance matrices of the core developer general model.

For the major developers, we get the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.382 & 0.024 & 0.613 & -0.028 \\ 0.024 & 0.016 & 0.010 & 0.021 \\ 0.613 & 0.010 & 1.568 & -0.074 \\ -0.028 & 0.021 & -0.074 & 0.357 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 2.257 & 0.275 & 0.995 & 0.048 \\ 0.275 & 0.099 & 0.323 & 0.026 \\ 0.995 & 0.323 & 3.728 & 0.617 \\ 0.048 & 0.026 & 0.617 & 0.718 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 0.372 & 0.049 & -0.029 & 0.196 \\ 0.049 & 0.026 & -0.049 & 0.096 \\ -0.029 & -0.049 & 0.605 & -0.209 \\ 0.196 & 0.096 & -0.209 & 0.691 \end{pmatrix}.$$

Figure B.6.: Covariance matrices of the major developer general model.

Finally, the covariances of minor developers can be expressed by these matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.092 & 0.006 & 0.009 & 0.009 \\ 0.006 & 0.015 & -0.009 & -0.002 \\ 0.009 & -0.009 & 0.300 & -0.015 \\ 0.009 & -0.002 & -0.015 & 0.075 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 0.737 & 0.320 & -0.076 & 0.702 \\ 0.320 & 0.358 & -0.097 & 0.116 \\ -0.076 & -0.097 & 0.619 & 0.431 \\ 0.702 & 0.116 & 0.431 & 9.003 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 1.428 & 0.126 & 0.031 & 8.585 \\ 0.126 & 0.028 & 0.016 & 0.790 \\ 0.031 & 0.016 & 0.603 & 0.345 \\ 8.585 & 0.790 & 0.345 & 64.314 \end{pmatrix}.$$

Figure B.7.: Covariance matrices of the minor developer general model.

KNN

$$A_{core} = \begin{array}{c c} & \begin{array}{ccc} low & med. & high \end{array} \\ \begin{array}{c} low \\ med. \\ high \end{array} & \begin{pmatrix} 0.67 & 0.26 & 0.07 \\ 0.39 & 0.46 & 0.15 \\ 0.10 & 0.23 & 0.67 \end{pmatrix} \end{array}, \quad A_{major} = \begin{array}{c c} & \begin{array}{ccc} low & med. & high \end{array} \\ \begin{array}{c} low \\ med. \\ high \end{array} & \begin{pmatrix} 0.64 & 0.30 & 0.06 \\ 0.39 & 0.48 & 0.13 \\ 0.09 & 0.19 & 0.72 \end{pmatrix} \end{array},$$

$$A_{minor} = \begin{array}{c c} & \begin{array}{ccc} low & med. & high \end{array} \\ \begin{array}{c} low \\ med. \\ high \end{array} & \begin{pmatrix} 0.74 & 0.25 & 0.01 \\ 0.29 & 0.70 & 0.01 \\ 0.02 & 0.03 & 0.95 \end{pmatrix} \end{array}.$$

Figure B.8.: General transition matrices for developer roles over all projects.

State	Core	Major	Minor
low	$(4,1,5,2)$	$(1,0,2,2)$	$(1,0,1,1)$
medium	$(20,8,32,16)$	$(12,4,8,22)$	$(10,3,2,20)$
high	$(32,16,47,50)$	$(17,5,13,30)$	$(0,0,0,1)$

Table B.2.: Means μ_k of emissions for the general model.

We start with the general model for core developers:

$$\Sigma_{low} = \begin{pmatrix} 14.877 & 1.237 & -2.692 & -0.850 \\ 1.237 & 0.890 & 0.374 & 0.488 \\ -2.692 & 0.374 & 6.355 & 2.962 \\ -0.850 & 0.488 & 2.962 & 2.282 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 59.545 & 21.424 & -23.843 & -45.184 \\ 21.424 & 16.644 & -8.199 & -17.000 \\ -23.843 & -8.199 & 62.687 & -3.350 \\ -45.184 & -17.000 & -3.350 & 93.667 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 33.824 & 22.523 & 25.361 & -20.830 \\ 22.523 & 25.093 & 20.369 & -30.849 \\ 25.361 & 20.369 & 89.609 & -112.080 \\ -20.830 & -30.849 & -112.080 & 332.444 \end{pmatrix}.$$

Figure B.9.: Covariance matrices of the core developer general model.

For the major developers, we get the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.197 & 0.066 & 0.330 & 0.232 \\ 0.066 & 0.086 & 0.100 & 0.046 \\ 0.330 & 0.100 & 1.640 & 0.526 \\ 0.232 & 0.046 & 0.526 & 4.431 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 0.569 & 0.195 & 0.037 & -0.265 \\ 0.195 & 0.318 & -0.163 & -0.496 \\ 0.037 & -0.163 & 7.094 & -10.258 \\ -0.265 & -0.496 & -10.258 & 54.044 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 1.302 & 0.362 & 0.018 & 0.758 \\ 0.362 & 0.350 & -0.030 & 0.716 \\ 0.018 & -0.030 & 0.928 & -0.068 \\ 0.758 & 0.716 & -0.068 & 4.148 \end{pmatrix}.$$

Figure B.10.: Covariance matrices of the major developer general model.

Finally, the covariances of minor developers can be expressed by these matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.0252 & 0.036 & 0.055 & 0.055 \\ 0.036 & 0.097 & -0.009 & 0.038 \\ 0.055 & -0.009 & 0.400 & -0.130 \\ 0.055 & 0.038 & -0.130 & 0.665 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 2.230 & 0.056 & 0.144 & 7.339 \\ 0.056 & 0.228 & 0.040 & 0.729 \\ 0.144 & 0.040 & 1.410 & 0.465 \\ 7.339 & 0.729 & 0.465 & 61.310 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 0.163 & 0.046 & 0.009 & 0.120 \\ 0.046 & 0.020 & 0.002 & 0.062 \\ 0.009 & 0.002 & 0.001 & -0.001 \\ 0.120 & 0.062 & -0.001 & 0.455 \end{pmatrix}.$$

Figure B.11.: Covariance matrices of the minor developer general model.

KNN3

$$A_{core} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.65 & 0.27 & 0.08 \\ med. & 0.38 & 0.48 & 0.14 \\ high & 0.11 & 0.22 & 0.67 \end{array}, \quad A_{major} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.65 & 0.28 & 0.07 \\ med. & 0.39 & 0.51 & 0.10 \\ high & 0.07 & 0.16 & 0.77 \end{array},$$

$$A_{minor} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.74 & 0.25 & 0.01 \\ med. & 0.28 & 0.71 & 0.01 \\ high & 0.01 & 0.01 & 0.98 \end{array}.$$

Figure B.12.: General transition matrices for developer roles over all projects.

State	Core	Major	Minor
low	$(5,1,6,2)$	$(1,0,2,1)$	$(1,0,0,1)$
medium	$(24,8,27,15)$	$(19,5,9,19)$	$(9,2,2,14)$
high	$(41,15,52,52)$	$(17,6,11,30)$	$(0,0,0,0)$

Table B.3.: Means μ_k of emissions for the general model.

We start with the general model for core developers:

$$\Sigma_{low} = \begin{pmatrix} 17.045 & 1.425 & -3.051 & -1.075 \\ 1.425 & 0.952 & 0.332 & 0.398 \\ -3.051 & 0.332 & 7.111 & 2.713 \\ -1.075 & 0.398 & 2.713 & 1.780 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 72.075 & 26.101 & -30.528 & -31.626 \\ 26.101 & 20.590 & -9.736 & -11.246 \\ -30.528 & -9.736 & 51.270 & 0.568 \\ -31.626 & -11.246 & 0.568 & 33.832 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 50.915 & 27.544 & 44.978 & -29.978 \\ 27.544 & 29.515 & 26.661 & -35.730 \\ 44.978 & 26.661 & 133.822 & -139.170 \\ -29.978 & -35.730 & -139.170 & 403.406 \end{pmatrix}.$$

Figure B.13.: Covariance matrices of the core developer general model.

For the major developers, we get the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.153 & 0.053 & 0.266 & 0.105 \\ 0.053 & 0.078 & 0.089 & 0.048 \\ 0.266 & 0.089 & 1.381 & 0.273 \\ 0.105 & 0.048 & 0.273 & 2.159 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 3.900 & 0.918 & 0.643 & -0.522 \\ 0.918 & 0.870 & 0.020 & -0.400 \\ 0.643 & 0.020 & 6.416 & -7.407 \\ -0.522 & -0.400 & -7.407 & 41.816 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 0.496 & 0.011 & -0.072 & -0.327 \\ 0.011 & 0.268 & -0.099 & 0.764 \\ -0.072 & -0.099 & 0.189 & -0.322 \\ 0.096 & 0.764 & -0.322 & 3.237 \end{pmatrix}.$$

Figure B.14.: Covariance matrices of the major developer general model.

Finally, the covariances of minor developers can be expressed by these matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.133 & 0.023 & 0.030 & 0.049 \\ 0.023 & 0.045 & -0.008 & 0.035 \\ 0.030 & -0.008 & 0.276 & -0.061 \\ 0.049 & 0.035 & -0.061 & 0.265 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 1.291 & 0.047 & 0.137 & 6.390 \\ 0.047 & 0.165 & 0.033 & 0.708 \\ 0.137 & 0.033 & 0.977 & 0.987 \\ 6.390 & 0.708 & 0.987 & 47.705 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 0.051 & 0.008 & 0.004 & 0.013 \\ 0.008 & 0.002 & 0.001 & 0.002 \\ 0.004 & 0.001 & 0.001 & 0.001 \\ 0.013 & 0.002 & 0.001 & 0.0110.002 \end{pmatrix}.$$

Figure B.15.: Covariance matrices of the minor developer general model.

Random Forests

$$A_{core} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.65 & 0.26 & 0.09 \\ 0.39 & 0.47 & 0.14 \\ 0.10 & 0.23 & 0.67 \end{pmatrix} \end{array}, \quad A_{major} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.64 & 0.30 & 0.06 \\ 0.40 & 0.49 & 0.11 \\ 0.08 & 0.15 & 0.77 \end{pmatrix} \end{array},$$

$$A_{minor} = \begin{array}{c} \\ low \\ med. \\ high \end{array} \begin{array}{c} \begin{array}{ccc} low & med. & high \end{array} \\ \begin{pmatrix} 0.67 & 0.27 & 0.06 \\ 0.33 & 0.61 & 0.06 \\ 0.15 & 0.07 & 0.78 \end{pmatrix} \end{array}.$$

Figure B.16.: General transition matrices for developer roles over all projects.

State	Core	Major	Minor
low	$(4,1,5,1)$	$(1,0,2,1)$	$(1,0,1,2)$
medium	$(21,7,25,16)$	$(11,4,7,18)$	$(10,2,5,21)$
high	$(35,13,51,47)$	$(16,5,15,26)$	$(4,2,6,6)$

Table B.4.: Means μ_k of emissions for the general model.

We start with the general model for core developers:

$$\Sigma_{low} = \begin{pmatrix} 14.822 & 1.212 & -2.706 & -0.970 \\ 1.212 & 0.840 & 0.329 & 0.381 \\ -2.706 & 0.329 & 5.881 & 2.382 \\ -0.970 & 0.381 & 2.382 & 1.669 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 59.034 & 21.561 & -23.572 & -44.758 \\ 21.561 & 16.751 & -7.763 & -16.946 \\ -23.572 & -7.763 & 40.146 & -2.557 \\ -44.75 & -16.946 & -2.557 & 93.036 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 37.598 & 22.701 & 32.255 & -21.391 \\ 22.701 & 24.927 & 20.772 & -30.993 \\ 32.255 & 20.772 & 103.350 & -113.049 \\ -21.39 & -30.993 & -113.049 & 332.874 \end{pmatrix}.$$

Figure B.17.: Covariance matrices of the core developer general model.

For the major developers, we get the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.197 & 0.063 & 0.302 & 0.204 \\ 0.063 & 0.076 & 0.088 & 0.039 \\ 0.302 & 0.088 & 1.511 & 0.470 \\ 0.204 & 0.039 & 0.470 & 3.955 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 0.484 & 0.172 & 0.149 & -0.217 \\ 0.172 & 0.249 & -0.121 & -0.492 \\ 0.149 & -0.121 & 6.154 & -9.083 \\ -0.217 & -0.492 & -9.083 & 47.912 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 0.882 & 0.239 & 0.177 & 0.423 \\ 0.239 & 0.259 & 0.001 & 0.540 \\ 0.177 & 0.001 & 0.479 & -0.063 \\ 0.423 & 0.540 & -0.063 & 47.912 \end{pmatrix}.$$

Figure B.18.: Covariance matrices of the major developer general model.

Finally, the covariances of minor developers can be expressed by these matrices:

$$\Sigma_{low} = \begin{pmatrix} 0.313 & 0.076 & 0.042 & 0.032 \\ 0.076 & 0.045 & 0.004 & -0.059 \\ 0.042 & 0.004 & 0.617 & -0.272 \\ 0.032 & -0.059 & -0.272 & 1.605 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 2.882 & 0.196 & 0.250 & 7.858 \\ 0.196 & 0.270 & 0.029 & 0.875 \\ 0.250 & 0.029 & 2.032 & 1.043 \\ 7.858 & 0.875 & 1.043 & 63.613 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 1.443 & 0.278 & 0.787 & 0.476 \\ 0.278 & 0.305 & 0.090 & 0.938 \\ 0.787 & 0.090 & 1.854 & -2.494 \\ 0.476 & 0.938 & -2.494 & 10.430 \end{pmatrix}.$$

Figure B.19.: Covariance matrices of the minor developer general model.

C. Correlations of Transition Matrices of Project Activity

Table C.1.: Correlations of transition matrices for project activity.

Activity	Project	Accumulo	Cayenne	Kafka	Storm	Ant ivy	Archiva	Deltaspike	Mahout	Nutch	Opennlp	Struts	Tez	Tika	Zookeeper	Pig	Xerces
Active	Accumulo	–	0.94	0.91	0.98	0.81	0.95	0.98	0.97	0.94	0.99	0.96	0.94	0.95	0.94	0.99	0.97
	Cayenne	0.94	–	0.89	0.94	0.87	0.89	0.88	0.96	0.97	0.91	0.96	0.97	0.96	0.80	0.93	0.94
	Kafka	0.91	0.89	–	0.95	0.95	0.75	0.87	0.95	0.91	0.91	0.94	0.93	0.80	0.84	0.86	0.96
	Storm	0.98	0.94	0.95	–	0.89	0.91	0.96	0.98	0.96	0.98	0.97	0.96	0.93	0.92	0.97	0.99
Maintenance	Ant ivy	0.81	0.87	0.95	0.89	–	0.68	0.74	0.87	0.91	0.79	0.91	0.92	0.76	0.67	0.78	0.92
	Archiva	0.95	0.89	0.75	0.91	0.68	–	0.92	0.87	0.89	0.92	0.90	0.88	0.98	0.87	0.98	0.89
	Deltaspike	0.98	0.88	0.87	0.96	0.74	0.92	–	0.95	0.89	0.99	0.92	0.89	0.91	0.97	0.96	0.93
	Mahout	0.97	0.96	0.95	0.98	0.87	0.87	0.95	–	0.93	0.97	0.95	0.94	0.91	0.92	0.94	0.96
	Nutch	0.94	0.97	0.91	0.96	0.91	0.89	0.89	0.93	–	0.92	0.99	0.99	0.95	0.79	0.95	0.98
	Opennlp	0.99	0.91	0.91	0.98	0.79	0.92	0.99	0.97	0.92	–	0.95	0.92	0.92	0.96	0.97	0.96
	Struts	0.96	0.96	0.94	0.97	0.91	0.90	0.92	0.95	0.99	0.95	–	0.99	0.95	0.83	0.96	0.99
	Tez	0.94	0.97	0.93	0.96	0.92	0.88	0.89	0.94	0.99	0.92	0.99	–	0.94	0.79	0.94	0.98
	Tika	0.95	0.96	0.80	0.93	0.76	0.98	0.91	0.91	0.95	0.92	0.95	0.94	–	0.82	0.98	0.93
	Zookeeper	0.94	0.80	0.84	0.92	0.67	0.87	0.97	0.92	0.79	0.96	0.83	0.79	0.82	–	0.91	0.87
Inactive	Pig	0.99	0.93	0.86	0.97	0.78	0.98	0.96	0.94	0.95	0.97	0.96	0.94	0.98	0.91	–	0.96
	Xerces	0.97	0.94	0.96	0.99	0.92	0.89	0.93	0.96	0.98	0.96	0.99	0.98	0.93	0.87	0.96	–

D. Hidden Markov Models for Project Activity

In the following, we list all produced project activity models that are not taken as an example within the thesis. This means, we report for each classifier the transition matrices, means of the emissions, and covariance matrices of emissions.

KNN3

For KNN3, the transitions and means of emissions were already mentioned. Thus, we start with covariance matrices Σ_k belonging to the means μ_k in Table 6.10. Together, this specifies the emission distributions $\mathcal{N}(k;\mu_k,\Sigma_k)$.

We start with the covariance matrices for active projects:

$$\Sigma_{low} = \begin{pmatrix} 217.83 & -15.08 & 60.92 \\ -15.08 & 115.43 & 77.85 \\ 60.92 & 77.85 & 494.23 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 462.87 & -84.63 & -20.28 \\ -84.63 & 2232.02 & 496.23 \\ -20.28 & 496.23 & 3654.10 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 1409.69 & 215.84 & -783.13 \\ 215.84 & 924.86 & 382.31 \\ -783.13 & 382.31 & 3676.08 \end{pmatrix}.$$

Figure D.1.: Covariance matrices of active models.

For maintenance projects we received the following matrices:

$$\Sigma_{low} = \begin{pmatrix} 5.41 & 10.30 & -0.97 \\ 10.30 & 40.64 & 3.44 \\ -0.97 & 3.44 & 27.06 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 22.82 & -28.59 & -9.73 \\ -28.59 & 454.65 & -0.03 \\ -9.73 & -0.03 & 82.21 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 52.94 & 116.14 & 6.01 \\ 116.14 & 570.83 & 46.89 \\ 6.01 & 46.89 & 171.93 \end{pmatrix}.$$

Figure D.2.: Covariance matrices of maintenance models.

Finally, for inactive projects we get:

$$\Sigma_{low} = \begin{pmatrix} 49.01 & 14.21 & 0.74 \\ 14.21 & 29.02 & 15.40 \\ 0.74 & 15.40 & 163.35 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 490.79 & -148.40 & 21.85 \\ -148.40 & 192.88 & -15.65 \\ 21.85 & -15.65 & 825.36 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 725.25 & 1137.02 & -425.60 \\ 1137.02 & 6411.84 & 34.14 \\ -425.60 & 34.14 & 5986.13 \end{pmatrix}.$$

Figure D.3.: Covariance matrices of inactive models.

Random Forests

For the transitions with Random Forests as classifier, we received the following:

$$A_{active} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.67 & 0.25 & 0.08 \\ med. & 0.30 & 0.46 & 0.24 \\ high & 0.12 & 0.30 & 0.58 \end{array}, \quad A_{maintenance} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.67 & 0.25 & 0.08 \\ med. & 0.35 & 0.43 & 0.22 \\ high & 0.08 & 0.33 & 0.59 \end{array},$$

$$A_{inactive} = \begin{array}{c|ccc} & low & med. & high \\ low & 0.67 & 0.26 & 0.07 \\ med. & 0.37 & 0.47 & 0.16 \\ high & 0.05 & 0.34 & 0.61 \end{array}.$$

Figure D.4.: General transition matrices for project activity over all projects.

For the means of emission distributions we get the following:

State	Active	Maintenance	Inactive
low	$(27, 57, 104)$	$(14, 37, 50)$	$(14, 40, 66)$
medium	$(33, 126, 255)$	$(17, 60, 122)$	$(8, 32, 137)$
high	$(90, 183, 263)$	$(30, 99, 215)$	$(13, 70, 195)$

Table D.1.: Means μ_k of emissions for the general project activity model.

We now report all covariance matrices for Random Forests:

$$\Sigma_{low} = \begin{pmatrix} 50.83 & 40.55 & -46.40 \\ 40.55 & 172.17 & -211.10 \\ -46.40 & -211.10 & 2847.52 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 484.20 & -581.10 & -170.71 \\ -581.10 & 3927.29 & 1644.29 \\ -170.71 & 1644.29 & 3133.87 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 1341.98 & -138.67 & -382.68 \\ -138.67 & 4493.06 & 1952.04 \\ -382.68 & 1952.04 & 3752.65 \end{pmatrix}.$$

Figure D.5.: Covariance matrices of active models.

$$\Sigma_{low} = \begin{pmatrix} 5.71 & 11.48 & -0.95 \\ 11.48 & 42.25 & 0.02 \\ -0.95 & 0.02 & 22.18 \end{pmatrix},$$

$$\Sigma_{medium} = \begin{pmatrix} 26.50 & -18.05 & -9.08 \\ -18.05 & 402.10 & -0.51 \\ -9.08 & -0.51 & 62.61 \end{pmatrix},$$

$$\Sigma_{high} = \begin{pmatrix} 41.53 & 57.03 & 13.38 \\ 57.03 & 360.59 & 79.52 \\ 13.38 & 79.52 & 176.76 \end{pmatrix}.$$

Figure D.6.: Covariance matrices of maintenance models.

$$
\begin{aligned}
\Sigma_{low} &= \begin{pmatrix} 49.64 & 12.37 & -2.12 \\ 12.37 & 27.72 & 15.75 \\ -2.12 & 15.75 & 95.01 \end{pmatrix}, \\
\Sigma_{medium} &= \begin{pmatrix} 407.33 & -116.06 & 30.88 \\ -116.06 & 181.69 & -28.04 \\ 30.88 & -28.04 & 703.03 \end{pmatrix}, \\
\Sigma_{high} &= \begin{pmatrix} 731.86 & 1112.20 & -462.54 \\ 1112.20 & 6339.42 & 64.59 \\ -462.54 & 64.59 & 5929.46 \end{pmatrix}.
\end{aligned}
$$

Figure D.7.: Covariance matrices of inactive models.

E. Questionnaire for AB/BA Crossover Study

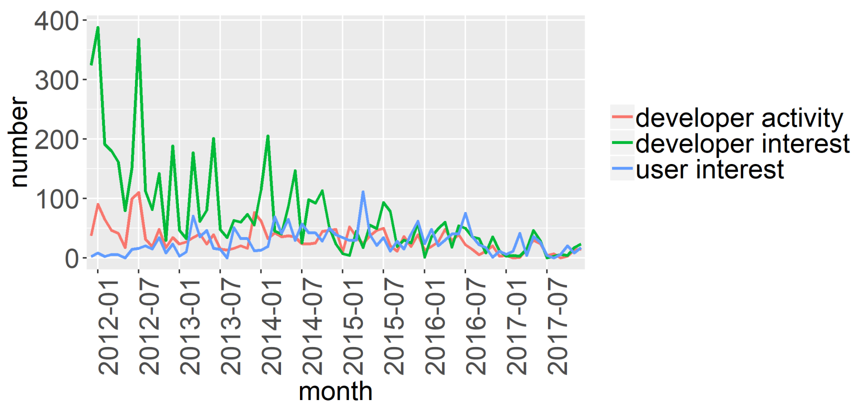
Task Description

On the following pages, you will find a picture displaying the **activity of an open source software project** over the time on each.

Please judge for each project whether you would consider it as **active** or **inactive** and mark your decision.

Please do not compare the projects. Assess each project individually and intuitively without referring back or forth to other projects.

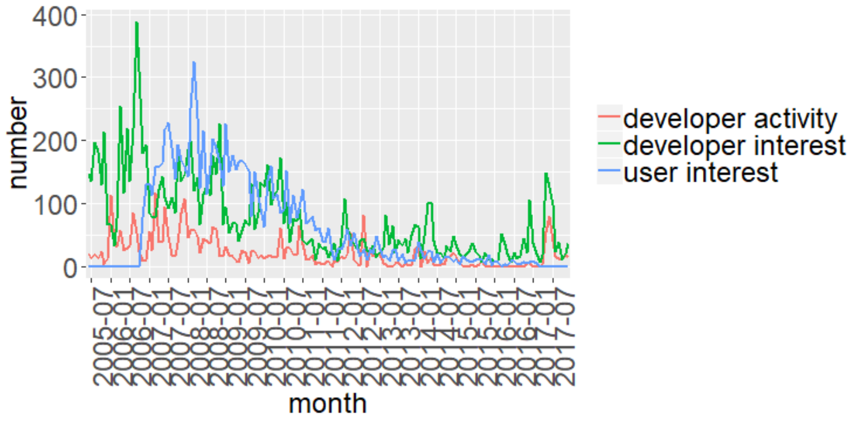
1



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

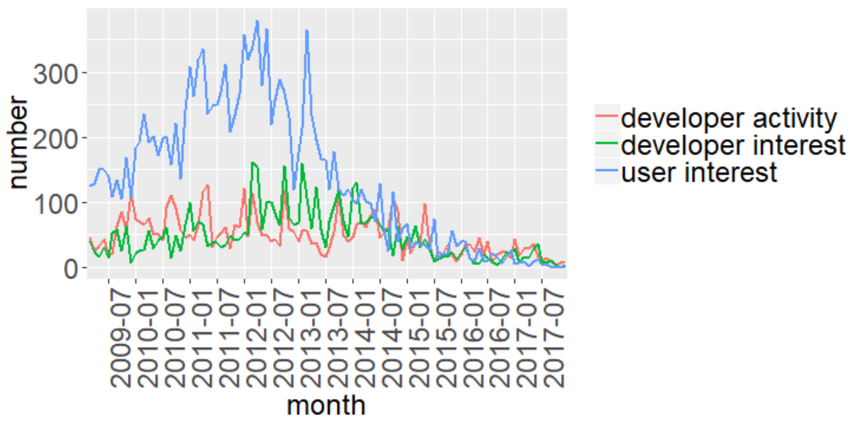
2



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

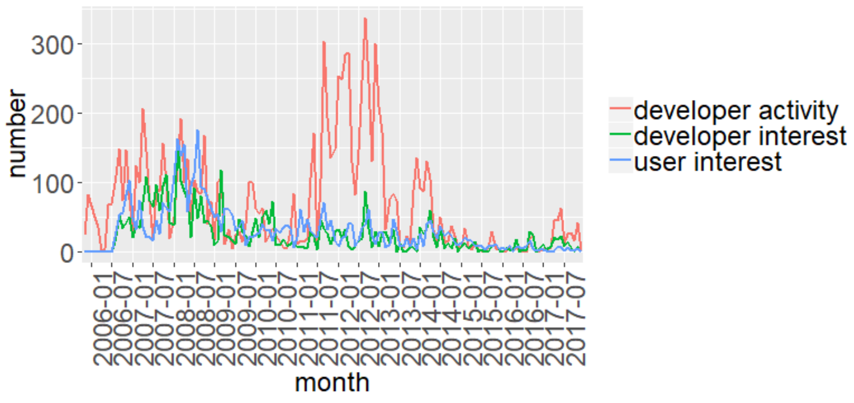
3



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

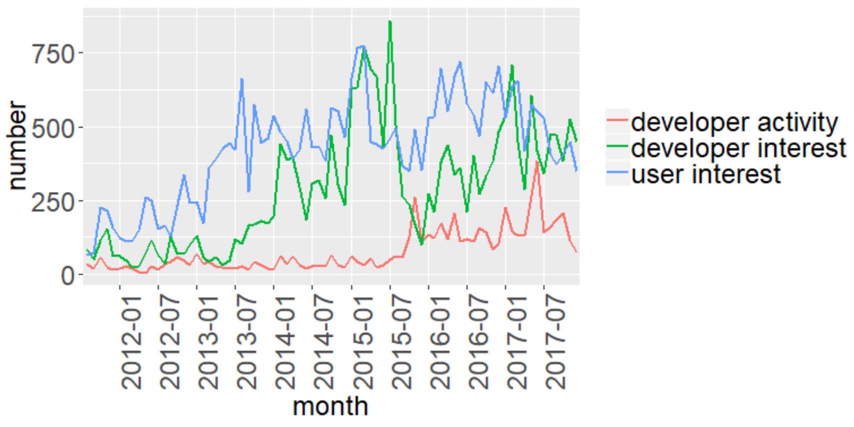
4



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

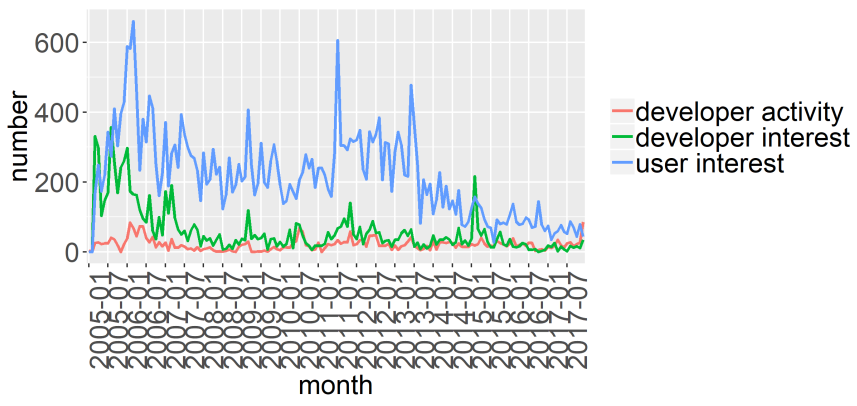
5



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

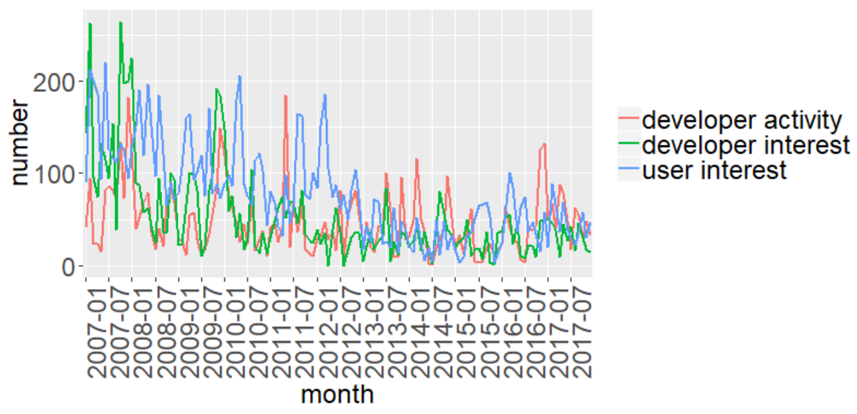
6



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

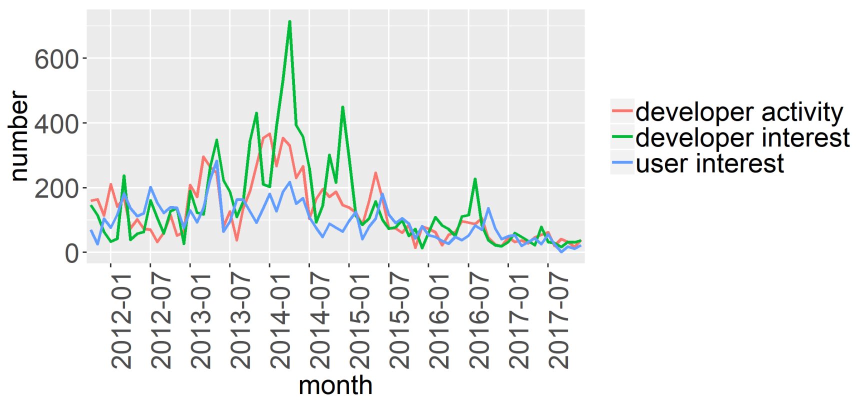
7



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

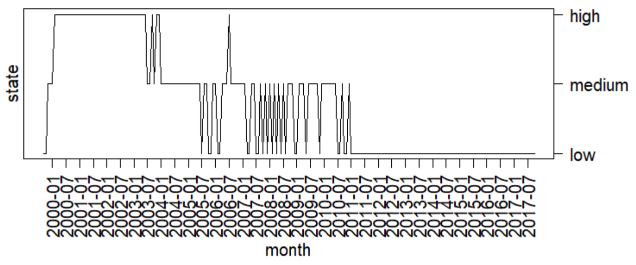
8



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

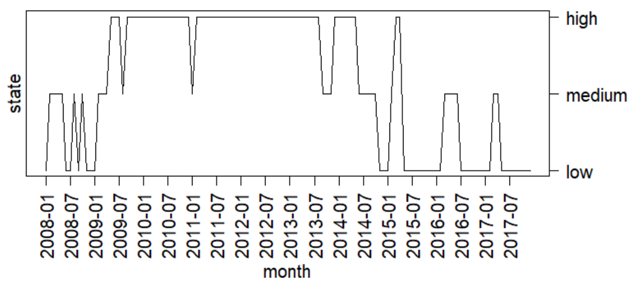
9



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

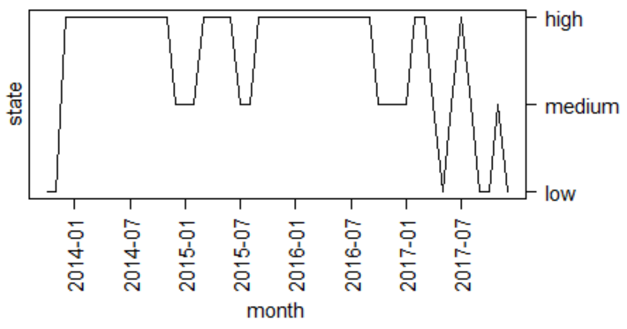
10



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

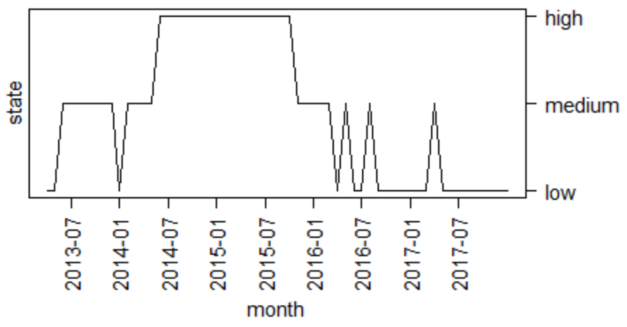
11



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

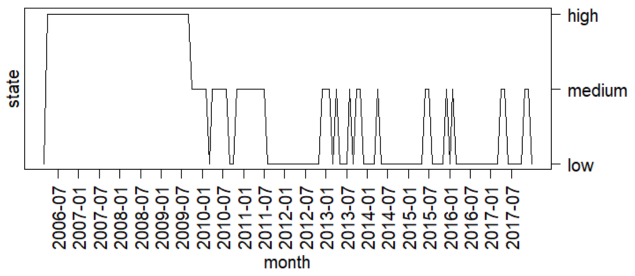
12



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

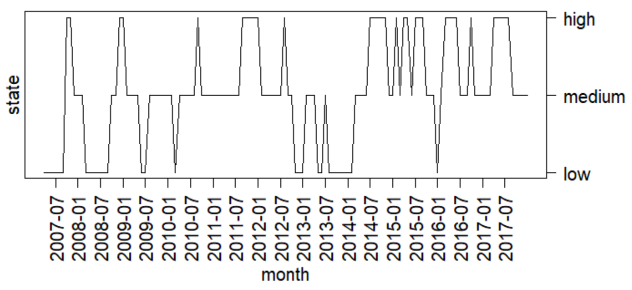
13



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

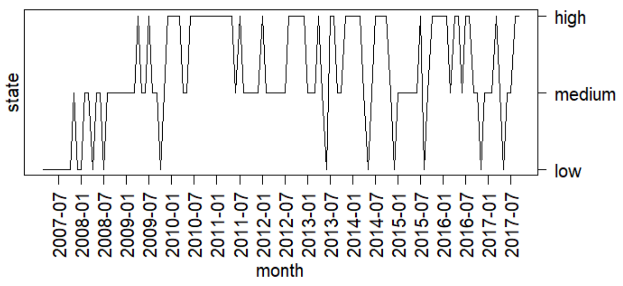
14



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

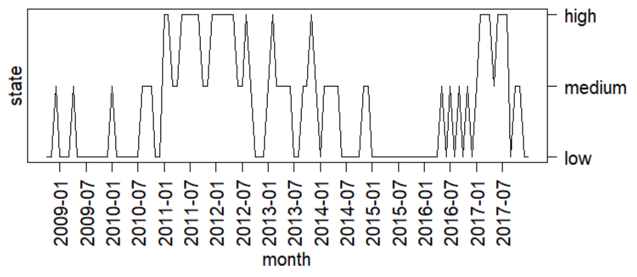
15



active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

16



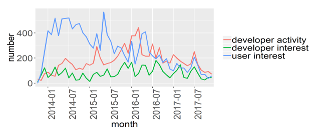
active: ☐ inactive: ☐

How sure are you? very ☐ fairly ☐ halfway ☐ little ☐

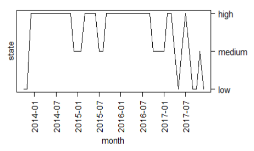
17

Task Evaluation

Please evaluate which method you prefer:



Both equal



☐ ☐ ☐ ☐ ☐

Thank you for your participation!

18

F. R Output for the Calculation of Effect Sizes

output.txt

```
Linear mixed model fit by REML ['lmerMod']
Formula: value ~ TimePeriod + variable + 1  Participant
   Data: scores_long

REML criterion at convergence: -21.5

Scaled residuals:
     Min       1Q   Median       3Q      Max
-1.77505 -0.55080  0.06193  0.57809  1.95494

Random effects:
 Groups      Name        Variance Std.Dev.
 Participant Intercept 0.01863  0.1365
 Residual              0.01819  0.1349
Number of obs: 54, groups:  Participant, 27

Fixed effects:
                      Estimate Std. Error t value
Intercept           0.688823   0.040942  16.824
TimePeriodR2         -0.007555   0.036729  -0.206
variableScore_states -0.046016   0.036729  -1.253

Correlation of Fixed Effects:
            Intr TmPrR2
TimePeridR2 -0.432
vrblScr_stt -0.432 -0.037
```