Vladimir Kovalenko

Data-Driven Software Engineering

Data-Driven Software Engineering

Data-Driven Software Engineering

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op woensdag 24 maart 2021 om 17.30 uur

door

Vladimir Vladimirovich KOVALENKO

Master of Science in Applied Mathematics and Physics,
Academisch Universiteit van de Russische Academie van Wetenschappen, Rusland,
geboren te Sint-Petersburg, Rusland.

Dit proefschrift is goedgekeurd door de

promotor: Prof. dr. A. van Deursen
promotor: Prof. dr. A. Bacchelli

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof. dr. A. van Deursen,	TU Delft, promotor
Prof. dr. A. Bacchelli,	Universität Zürich, Zwitserland, promotor

Onafhankelijke leden:

Prof. dr. J. Krinke,	University College London, Verenigd Koninkrijk
Prof. dr. N. Nagappan,	Microsoft Research en IIT Delhi, India
Prof. dr. A. Serebrenik,	TU Eindhoven
Prof. dr. D. Shepherd,	Virginia Commonwealth University, Verenigde Staten
Prof. dr. A.E. Zaidman,	TU Delft
Prof.dr.ir. G.J.P.M. Houben,	TU Delft, reservelid

Keywords: Data-Driven Software Engineering

Printed by: Printenbind

Cover image: Ministry of Agriculture and Food of the Moscow Region
Processing and layout by Margarita Kovalenko

Style: TU Delft House Style, with modifications by Moritz Beller
`https://github.com/Inventitech/phd-thesis-template`

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

Contents

Summary

Specialized tools, such as IDEs, issue trackers, and code review tools, are an indispensable part of the modern software engineering process. These tools are constantly evolving. Besides enabling tools to support a wider range of technologies and frameworks, we are learning to provide additional features in completely new ways. One prominent stream of innovation in software engineering tools is dedicated to utilizing historical data to enable data-driven features, such as defect prediction engines and recommender systems, which leverage records of prior activity to assist with decision making. Many data-driven *features* in software engineering tools initially get born out of the context of real-world tools as *techniques* devised and evaluated in synthetic settings by researchers. While convenient, synthetic evaluation of approaches that are ultimately aimed at bringing improvement to real world problems involves a number of simplifications and assumptions.

In this dissertation, we highlight several aspects that, while vital for bringing innovative methods to software engineering tools, are often discarded in existing research. We closely explore several topics specific to artificial evaluation environments, such as simplifications in mining file modification histories, use of synthetic datasets for source code authorship attribution, and a gap between accuracy of reviewer recommendation models and their perception by users. Moreover, we make a case for sharing technical artifacts by converting data mining pipelines into reusable tools, and propose a novel approach to modeling expertise transfer from code modification by capturing individual contribution style of developers.

Key contributions of this dissertation include a high-level model of the lifecycle of a data-driven software engineering technique, a discussion of dangerous assumptions and simplifications that are made on every step in this lifecycle, a demonstration of importance of a careful approach to mining software repositories, and a demonstration of serious misalignment between artificial evaluation and realistic environments for the problems of code reviewer recommendation and code authorship attribution.

We conclude the dissertation by discussing underlying reasons for misalignment between research environments and real-world tools, and propose potential steps to narrow it down and ultimately accelerate innovation in software engineering tooling.

Samenvatting

Gespecialiseerd gereedschap vormt een onmisbaar onderdeel van het moderne software-ontwikklingsproces, bijvoorbeeld voor het schrijven van code (in de geïntegreerde ontwikkelomgeving, de IDE), voor het rapporteren van problemen of gebruikerswensen, of voor het beoordelen van code. Dergelijk gereedschap evolueert voortdurend. Deze evolutie maakt het enerzijds mogelijk een breder aanbod van technologieën te ondersteunen. Daarnaast is er een ontwikkeling gaande waarin dergelijk gereedschap op een radicaal nieuwe manier wordt voorzien van nieuwe functies.

Een prominente stroom van innovatie in software-ontwikkelgereedschap betreft het benutten van historische data om datagedreven functionaliteit mogelijk te maken, zoals het voorspellen van foutieve code en aanbevelingssystemen, op basis van eerder verzamelde gegevens over ontwikkelingsactiviteiten.

Veel datagedreven functionaliteit in ontwikkelgereedschap ontstaat in eerste instantie vanuit een praktische behoefte, maar worden desalniettemin door wetenschappers ontworpen en geëvalueerd in een kunstmatige omgeving. Enerzijds vergemakkelijkt dit het experimenteren voor de onderzoekers. Maar anderzijds wordt kunstmatige evaluatie van methoden en technieken die uiteindelijk bedoeld zijn om verbeteringen te realiseren voor problemen uit de praktijk gekenmerkt door een aantal oversimplificaties en veronderstellingen.

In dit proefschrift ligt de nadruk op aspecten die weliswaar essentieel zijn om ontwikkelgereedschap daadwerkelijk te vernieuwen, maar die toch vaak genegeerd worden in het bestaande onderzoek. We bestuderen diverse thema's die kenmerkend zijn voor kunstmatige evaluatieomgevingen, zoals al te eenvoudige veronderstellingen bij het analyseren van programmawijzigingen zoals vastgelegd in een versiebeheersysteem, het gebruik van kunstmatige data voor het identificeren van de auteurs van code fragmenten, en de nauwkeurigheid van modellen voor het aanbevelen van code-beoordelaars ten opzichte van de perceptie van gebruikers van deze modellen in de praktijk. Daarnaast benadrukken we het belang van het delen van technische artefacten, middels het omzetten van data-anlayse productielijnen in herbruikbaar ontwikkelgereedschap. Bovendien stellen we een nieuwe aanpak voor om de overdracht van expertise te modelleren op basis van programmawijzigingen door de individuele programmerstijl van ontwikkelaars te karakteriseren.

De belangrijkste bijdragen van dit proefschrift omvatten (1) een hoog-niveau model van de levenscyclus van data-gedreven software-ontwikkelingstechnieken, (2) een discussie van gevaarlijke aannames en vereenvoudigingen die gemaakt worden in elke stap van deze levenscyclus, (3) het beargumenteren van het belang van een zorgvuldige aanpak voor het ontginnen van historische software-data, (4) het aantonen van ernstige discrepanties tussen enerzijds kunstmatige evaluatie en anderzijds realistische omgevingen voor de problemen van het aanbevelen van code-beoordelaars en het automatisch vaststellen van auteurschap van programmatuur.

We sluiten het proefschrift af met een bespreking van de onderliggende redenenen voor de kloof tussen onderzoeksomgevingen en gereedschap uit de praktijk, en stellen stappen voor om deze te overbruggen ten einde innovatie in software-ontwikkelgereedschap te versnellen.

Acknowledgments

The last five years have been quite a journey. In this section, I would like to give credit to those who made it worth it. First, I would like to say thank you to everyone I have met during the PhD. Getting to know so many incredible people was the greatest part of it all.

Alberto, thank you for the opportunity to work with you. Thank you for always being around for your students, for your professionalism, for your patience, and for always keeping your cool. While your decision to move to Zurich in our first year added a bit of turbulence to our journeys, thank you for that too — it also made things much more fun. I have learned a lot from you as a scientist and as a person, and I keep learning more.

Arie, since our very first meeting I have been amazed by your ability to ask razor-sharp questions within seconds of getting into context. Thank you for all the advice, support, positive attitude, and for making SERG such a great team. I wish we worked together more.

Dear Jens, Nachi, Alexander, David, Andy, Geert-Jan, thank you for kindly agreeing to join the committee, and for your deep comments on the draft. Dear colleagues and friends who volunteered to give feedback on the earlier versions, thank you.

Anand, Davide, Luca, Marco, it was great sharing the office with you. Having heated discussions and countless beers, too. I applaud you for starting the new incredible chapters in your lives in these uncertain times. Andy, Georgios, Mauricio, Minaksie, Annibale, Sebastian, Cynthia, and many others at SERG and around, thank you for making me feel welcome at the TU even after "leaving". Dear people at ZEST and s.e.a.l, thank you for making me feel like home in Zurich. I would be happy to come visit you again.

Delft would not have been a great place to live if not for the amazing people I've met. The Arubastraat crew, the DK team, the Oude Langendijk family, I can't wait until we can all meet again.

My dear colleagues at ICTL and ML4SE, thank you for your support during the final stages of the PhD. Timofey and Andrey, thank you for the opportunity to rejoin the JB team in a new role, and then take up yet another new role, and all that while finishing the thesis. My dear friends at JetBrains Amsterdam, it's funny how five years later many of us still live an hour away from each other, even after moving countries. I can't complain here, though. Thank you for being so amazing.

Dear people of The Netherlands, thank you for making your country what it is. Saint Petersburg still feels like home every time I'm around. My dear friends who make coming back feel so great, thank you for that.

Mom and dad, thank you for your unconditional support and for simply believing that every adventure I start is a good idea. Margo, thank you for your support and for sharing all the inconvenience of living this far apart, especially now that taking this short flight is not an ordinary thing.

Vladimir
KL1395, February 2021

1 Introduction

1.1 Motivation

1.1.1 Software Engineering

The software industry is exciting these days. Since its birth in the middle of the last century, it has been booming to the day, with every year bringing a number of technological solutions to wide adoption, and some more to retirement. Lately, software had been taking over the world: In 2000, only few households possessed a software-powered device. Today, only two decades later, a layman carries several such devices around. Everything from a laptop to a public transport pass relies on software to function. This incredible rate of innovation, among other things, makes the field of software engineering so challenging and rewarding to work in, to study, and to advance.

A large stream of research and innovation in software engineering is dedicated to enabling people to build complex software products more efficiently, and to collaborate on software in large and diverse teams. A prominent driver for innovation in software engineering is software engineering tooling, which is an indispensable part of the modern software engineering process. In fact, every step in the software engineering lifecycle, from requirements engineering to runtime monitoring, is supported by tools these days. While general-purpose tools, like collaborative document editors and messengers, are widely used for non-technical tasks in the development process, most of the core activities specific to software engineering are supported by dedicated tailored tools like IDEs, issue trackers, code review tools, continuous integration tools, and version control systems.

Specialized software development tools are constantly evolving. The most prominent direction of improvement of software engineering tools is in measurable characteristics: build tools are becoming more efficient, team collaboration tools can cater larger teams, IDEs support more and more languages and frameworks. Progress in this direction can be attributed to advances in hardware capabilities, to improved engineering quality of the tools, to better understanding of needs of tools' users by tool vendors, and to the transition of major engineering tools to the open source platform model, where contributions from the community help to further expand capabilities of the tools.

Besides enabling tools to support a wider range of technologies and frameworks, we are learning to provide additional features in completely new ways. IDEs evolved to

1

provide interactive navigation through code entities and automatically refactor code, static analysis systems provide more and more detailed insight into codebase quality, code hosting platforms automatically suggest security bug fixes by analyzing vast dependency networks. Innovation of this sort, enabling tools to assist in qualitatively new tasks, depends on building, implementing, and using richer models of the medium that the tools manipulate.

IDEs are arguably the most complex software engineering tools. In fact, version control repositories of major IDEs such as Eclipse and IntelliJ IDEA are among the largest open source repositories in existence, with hundreds of contributors involved in development and maintenance. This complexity can in part be attributed to the *integrated* quality of the IDEs: providing support for each language, framework, or external tool involves significant engineering work. Core features, such as code navigation, refactoring, and inspections, owe their existence to elaborate layers that build rich internal representations of code and to processing techniques that operate with these representations.

Internal representations of code, enabling IDEs to provide code insight features, are quasi-static: while they are dynamically recalculated when developers edit code, they only represent the current state of the underlying project. Static snapshots of code are not the only data available in the software engineering context. Version control systems preserve most intermediate states of software, making it possible to trace back its evolution path. Moreover, the processes besides code manipulation and corresponding tools, such as code review systems, issue trackers, continuous integration pipelines, and messengers, all generate and store vast amounts of historical records of developers' activity through everyday use.

The most straightforward way to use such historical data is enabling developers to refer to history. Besides direct use for reference, historical data can as well be utilized to empower tools with new functionality. One simple example of such functionality is enabling the tools to present an aggregated overview of software evolution, which may highlight technical debt or guide refactorings. More sophisticated examples of potential features based on historical data, involving complex processing of data, are defect prediction systems and recommender systems, which leverage records of prior activity to assist with decision making.

The idea of using records of historical development activity to empower software engineering tools has long been around in the academic research community. However, most of the proposed approaches to this idea are yet to be adopted in engineering tools widely used in industry. Luckily, there are several exceptions. Some modern code review tools feature reviewer recommendation systems that model expertise of individual developers in different parts of the codebase by tracking back history of prior development. In a similar way, some build tools can automatically assign investigation of build failures.

A fundamental ambition in the software industry is to produce higher quality software with less effort. While there exist examples of leveraging historical data in tools to assist with this goal, there is still a great potential for further adoption of data-driven techniques in tools. Solely by adopting promising techniques proposed in academic research, software engineering tools could automate or render unnecessary some of the demanding work that is done by developers today. The tools of tomorrow could predict and localize bugs, ensure optimal workload of developers, ensure a healthy knowledge distribution within teams, facilitate efficient code reuse, and even generate code from natural language specifications.

The road to this goal — producing better software with less effort involved — has several distinct lanes. They include refining core techniques for data collection and utilization, discovering new sources of data, adopting existing data in new contexts, lowering the technical barrier to integration of techniques with tools, figuring out users' needs, and setting up an efficient high-level feedback loop to make sure that iterative improvements in all these directions can be enjoyed and embraced by the end users as quickly as possible. This thesis seeks to contribute to the high-level goal by proposing concrete improvements in these areas, and getting them into perspective by presenting them along a high-level model of evolution and adoption of data-driven techniques and features in software engineering tools.

1.2 Research in Data-Driven Software Engineering

1.2.1 Intelligent Software Engineering

Over the recent years, the craft of putting software data to use has matured into a standalone field of Mining Software Repositories (MSR), with a growing community of researchers around the world and a dedicated conference. The broad area of interest for the MSR community is retrieval and processing of data from software repositories, such as version control systems, communication tools, issue trackers, and runtime log containers, and extraction of actionable knowledge from this data [1]. The MSR community seeks to use the software data and extracted knowledge to provide insight into the nature of software engineering processes [2, 3] and to develop better methods of mining and representation of software data [4, 5]. Another, more practice-oriented, stream of MSR research is dedicated to devising approaches for tool-based assistance to software engineers [6–10]. All these tracks contribute to the high-level goal of building better software, either through deeper understanding of the engineering process and its challenges or through innovation in tooling.

One of the most widely used sources of software data in MSR is version control repositories [11]. Version control tools of today, such as *Git*, store intermediate states of systems' source code and enable efficient, collaboration-friendly development workflows. History of states and modifications of source code, stored in version control systems, can be processed and leveraged in engineering tools in several ways. Software visualization techniques [12], consisting in displaying aggregated historical data, can help facilitate decision making — for example, by revealing potentially problematic parts of codebases [13]. Defect prediction techniques [7, 14] aim to pinpoint potentially defective source code entities through statistical analysis of code that introduced prior defects. This knowledge potentially helps with resource allocation by reducing effort for quality control. In addition, defect prediction models help researchers gain deeper insights into software quality and processes by understanding which factors contribute to defects. The idea of change prediction [8] stands in applying association rule mining to identify source code files frequently changed together, which helps to avoid unintended incomplete changes and reveals implicit logical coupling between components, which in turn can facilitate refactorings.

While data from version control alone can be leveraged in multiple ways, using other sources of data and their combinations opens more opportunities for support of development process [15–17]. One family of data-driven techniques, utilizing data from code

1

review archives and issue trackers, aims to support decision making in collaborative software engineering. There exist approaches to automatic assignment of code reviews [9, 18], and bug reports [10, 19]. One benefit of these techniques is reduced cognitive load on those involved in decision making. Another benefit is improved quality of software and reduced iteration time: assignee recommendation helps ensure that the work that requires deep expert knowledge is done by the most qualified people in the given context. These benefits are possible thanks to the underlying idea of the techniques being modeling of contributors' expertise, either by directly linking contributors to data entities [20], or by extracting topics [21]. Some of these approaches are adopted in practice [22].

Progress in methods for retrieval and processing of data from software archives, growth of available data, and major advances in methods, tooling, and hardware for statistical learning in the recent years have triggered a growing trend of application of machine learning methods to software engineering problems. Approaches based on machine learning, often based on concepts originating from the field of natural language processing, have yielded significant improvement over state of the art in tasks such as code summarization [23] and method name suggestion [24]. While clearly being of potential value in software engineering tools, many of these techniques currently primarily serve a purpose of providing a benchmark for modeling of source code semantics [5, 23] and are yet to mature for widespread use. However, machine learning is starting to find its applications in core features of industry-grade tools, such as code completion in IDEs [25].

1.2.2 Data-Driven Tool Development Cycle

Data-Driven Techniques and Features

Central to this thesis are concepts of a *data-driven feature* and a *data-driven technique*. The primary purpose of the definitions for this dissertation is to provide a broader framework for concrete issues investigated in this thesis, to help us underline contributions of work comprising this thesis to the broader area of knowledge in software engineering research. The most suitable for this dissertation are pragmatic, informal, and limited definitions of a data-driven feature and technique.

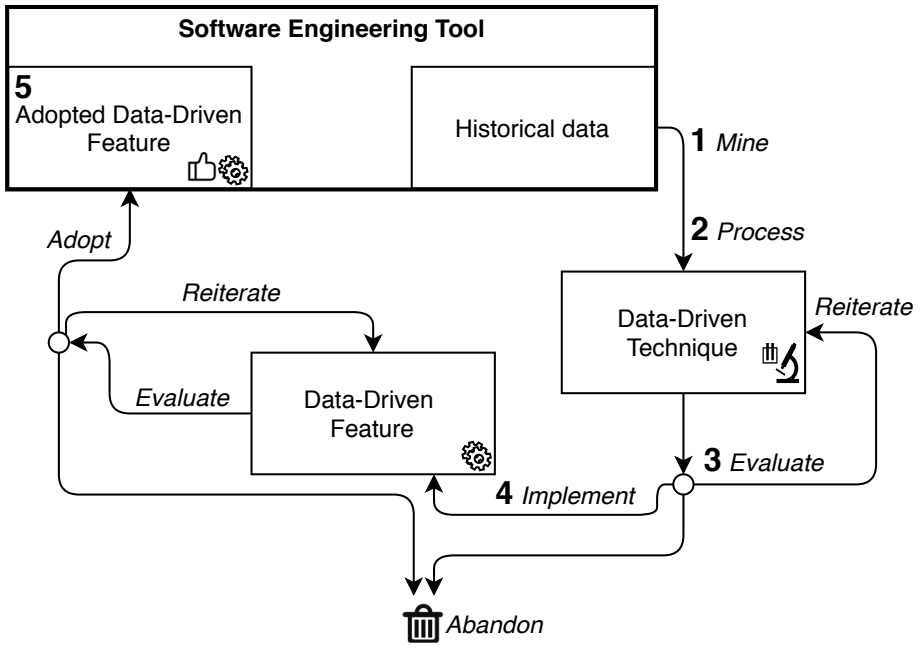
A **data-driven feature** is *a unit of functionality of a software engineering tool that relies on processed data of historical development activity.*

A **data-driven technique** is *the underlying method of utilizing and processing historical development activity data for a data-driven feature.*

Processing is key in the definition of a data-driven feature. In fact, any piece of software operates with data. Some software engineering tools, such as version control systems, are designed to store and present records of historical data, thus utilizing it. While we appreciate the value, complexity, and beauty of these tools from the engineering perspective, we exclude them from the definition of a data-driven software engineering technique, to focus on the techniques that extract additional value from data by processing it.

Our definitions are centered on usage of longitudinal historical data, rather than single snapshots of projects. Some of the most advanced, feature-rich, and complex software engineering tools — IDEs — offer uncountable features for code manipulation, navigation, and reuse, thanks to very rich internal models of code, which are products of processing. However, we exclude IDEs, as well as features of other tools relying on static snapshots

Figure 1.1: The Design Loop



of data, from the scope of this dissertation to center on the more innovative ideas around features that process historical data.

The Design Loop

We present a conceptual model for the process of design and adoption of data-driven techniques and features. In simple terms, the model consists of the following steps.

- Software engineering tools generate and store traces of user activity during everyday use.
- The data is extracted and used as a basis for a data-driven software engineering *technique*, aimed at improving efficiency of an engineering process.
- Efficiency of the technique is evaluated in an isolated environment and improved if needed and possible. As the result of this iterative process, the technique is either abandoned or implemented to serve as a *feature* of a software engineering tool.
- Once implemented, the feature faces evaluation in the context of a real-world tool.
- If deemed viable, the feature may be adopted.

Figure 1.1 presents a scheme of the lifecycle of a data-driven technique and feature described above, from mining data to validate the first prototype all the way to evaluating the feature by users in action. This diagram, like the description of the design loop as

1

presented above, does not represent any particular technique or feature, but is rather generic. It illustrates lifecycle stages common for many data-driven techniques and, where applicable, features they become. Often, not every stage is present in the lifecycle of a particular feature: for example, it may be implemented without prior prototyping and evaluation; a technique may only be evaluated outside the tool context and never be implemented for use.

Every stage in the lifecycle has its challenges. In the rest of this section, we describe every stage of the lifecycle and highlight challenges.

1. Mining the historical data of development activity.
The first step in the lifecycle of a data-driven technique is to extract historical data from engineering tools that store this data, such as version control repositories, issue trackers, or code review systems. Since many modern tools provide convenient APIs for data extraction, this is often a rather straightforward step.

The devil is in the detail. Where several data sources are involved, merely combining multimodal data into a single dataset is in itself a demanding task [17]. Version control tools, such as Git, are not very straightforward sources of data for data-driven techniques. Mutable history as well as limited capability of reproducing the underlying development process pose challenges at the stage of mining data from git [26]. Non-linear history requires a thorough approach to mining to get the most of the data: A limited or overly simplified approach to data mining may lead to suboptimal performance of techniques relying on the data down the line [27].

2. Processing data for use by techniques.
Once data is extracted from its origin, it is usually necessary to process it to model it for the corresponding technique. Examples of such processing are numerous. For classification-based techniques, such as defect prediction, training data has to be labeled. Labeling, either automatic [28] or manual, may introduce noise in the data [29]. Techniques based on social connections between actors [30, 31] require a processing step to build technical collaboration networks from history of contributions. All techniques treating code other than as text require defining models of code [5, 32].

The processing step can constitute a significant technical challenge, in terms of both of the work required to get it done and additional threats to validity of studies from possible bugs in processing and added difficulty of reproduction. While processing software is often provided along with papers describing new techniques, this is not omnipresent and, sometimes, merely sharing processing code does not ensure that the technique is reusable or the study is reproducible [33]. A possible solution is to build and distribute processing software that can be reused in a wide variety of tasks.

3. Evaluating the techniques.
Once a technique is implemented and data is available, the next step is to evaluate the technique's performance. While techniques are meant to provide assistance to users of tools, at the evaluation stage they normally exist outside of any tool, with no straightforward way to test their applicability and added value in a real tool context. Instead, performance of techniques is often approximated with one or several metrics [34]. In some cases, human

judgement is used to assess quality of techniques' output, in form of user studies, surveys, or interviews [35, 36].

Metrics enable researchers to quantify performance of techniques, and, where a common dataset is used, to demonstrate improvement over existing techniques. The weaker side of metrics is the fact that they do not allow for a universal — or, often, even reliable — interpretation, because numerous dimensions of potential added value of the technique and its perceived usefulness are boiled down to a single or several numbers. Assessment of techniques with user feedback is a more direct and flexible approach, but it is also often hard to interpret and generalize [37]. Such assessment is based on existing assumptions of researchers, which may not perfectly align with practitioners' perception of the same problems. Moreover, projects, teams, organizations, and communities differ a lot in views and needs, which makes every attempt on human-based evaluation, that is reasonably limited in scale, prone to bias and limited.

As the result of evaluation, the technique may be either abandoned, if its performance is below reasonable expectations of a production system or if there is no goal of integrating it into a real-world tool, or proceed into the next stage, where it is implemented in a production environment.

4. Implementing the feature prototypes

A data-driven feature is an implementation of a data-driven technique in the context of a software engineering tool. There are numerous difficulties with integrating techniques into tools. First and foremost, adapting the existing environment and the technique to each other is a significant engineering job, as the tool context imposes additional limitations on techniques, compared to a standalone prototype implementation. More complex techniques — and resulting features — are hard to scale in practice. Techniques relying on representations of code different from plain text, integrating in build pipeline, or using data from multiple sources require multiple dedicated language- and tool-specific integration modules to be implemented and maintained. Many tools have strict performance requirements. Some techniques, notably those based on deep learning, are simply not feasible to use in tool contexts: consuming several gigabytes of RAM on a developer machine or taking significant time to produce output for a code completion model is not feasible.

The overarching challenge with bringing techniques to life by implementing features is that techniques are often not initially designed with a particular tool context in mind. Moreover, the end goal of researchers working on techniques is often not to bring new features to software engineering tools, rather to invent new ways to use historical data for potentially functional applications or to improve over state of the art in established problems. Combined, these challenges make turning techniques into features a separate and often, arguably, more demanding job in terms of engineering effort than coming up with new techniques and improving existing ones.

5. Use in tool context

The final stage in the lifecycle of a data-driven software engineering technique is when the technique is implemented as a feature in a tool and is used by people. Testing the features in a live environment is crucial for reasoning about their utility, because performance metrics used in the previous steps of the loop may only serve as a proxy to estimate added

1

value for users. Thus, when features are deployed in production systems, it is common for the last phases of evaluation to take place, either through user feedback or through investigation of changes in target metrics, such as time taken for code review [35].

1.3 Problem Statement

The previous section outlined the distinct stages in the development of data-driven software engineering techniques and features. Many existing contributions to data-driven software engineering either exclude aspects of practical use and are focused on mechanics of turning data into potentially useful output, or focus on specific aspects of user interface design. This is particularly pronounced in academic contributions in the field of software engineering, where the primary output is new knowledge, rather than ready-to-use tools or improvement in industry practices. Moreover, published research is required to demonstrate novelty compared to prior work. For approaches that may ultimately be of use in tools, progress is clearly measurable by comparing evaluation scores on existing datasets to prior work. In such settings, the requirement of demonstrable novelty is likely to incentivize focus on mechanics of the techniques. With less attention targeted to other parts of the loop, such focus may lead to imbalance and may ultimately hinder conversion of data-driven techniques into data-driven features and their adoption.

1.3.1 Core theses

This dissertation explores issues surrounding design and development of techniques based on processing of historical data, or *data-driven techniques*, and their adoption in software engineering tools. It presents diverse research results revolving around two high-level theses:

- **T1. Data preparation and evaluation are no less important than core algorithms for efficient data-driven software engineering techniques.**
- **T2. Evaluation of data-driven techniques in synthetic settings is not sufficient to reason about the added value these techniques provide to the users.**

1.3.2 Research questions

To logically separate individual focus areas of this dissertation, we center its structure around five high-level research questions. Each research question corresponds to a separate chapter, which presents results that answer it.

RQ1. What is the effect of simplifications in data mining procedures on (i) resulting data and (ii) performance of techniques that rely on this data? Data mining at times involves deliberate simplifications, for example, to simplify the processing pipeline or to focus on parts of technical pipeline. However, even seemingly small simplifications in mining may at times cause a significant difference in data retrieval results and impact performance of techniques that consume the data.

In Chapter 2 we explore the effect of the assumption of linear file history during retrieval of histories for individual files from Git. We investigate the effect of this generally incorrect assumption on properties of resulting datasets and the effect of these differences on output of two data-driven techniques based on file histories.

RQ2. How can we convert mining pipelines into reusable tools? Data mining often involves significant technical work. As there are only a few data sources widely used during design of data-driven techniques, some of this work can be eliminated by building reusable flexible pipelines instead of throwaway mining code.

In Chapter 3 we describe a flexible library for mining of path-based representations of code. The library is based on our existing data mining pipeline, can be used in arbitrary settings, and can be extended to support new programming languages with a few lines of code.

RQ3. How closely do synthetic datasets resemble data from real projects? Data mining techniques, of which many aim to improve real-world software engineering, are often evaluated on preexisting datasets. While this approach is convenient for comparison to prior baselines, it is not always clear (1) to what extent synthetic datasets resemble real-world data and (2) how well performance of a technique on a synthetic dataset translates to performance in a real-world context.

In Chapter 4, after proposing a state-of-the-art language-agnostic model for authorship attribution of source code, we investigate performance of authorship attribution techniques on a dataset designed to closely resemble a large-scale industrial project, and seek to capture effect of individual dimensions of difference between synthetic and real-world data on performance of these techniques.

RQ4. Are there cases when evaluation of techniques in the lab does not reflect their usefulness for end users? Techniques, initially motivated by the need to provide added value for engineers, are commonly judged based on their performance estimates obtained in synthetic environments, for example by comparing recommendations to actual choices made by users in hitorical data. However, it is not clear how synthetic evaluation translates to perception of techniques' output by users and added value the techniques provide. Moreover, some aspects of real environments, such as feedback from recommendations on choices of users, is impossible to capture in laboratory settings. Thus, for a deeper understanding of aspects specific to real world and users' perception, it is vital to focus on the late stages in the loop where a technique is already implemented as a feature and deployed.

In Chapter 5 we perform the first to our knowledge *in vivo* evaluation of a reviewer recommendation system and extensively survey users' perception of quality and usefulness of its output.

RQ5. Can concrete data from code repositories provide insight into fuzzy processes, such as knowledge transfer? This research question is related to the stage of coming up with a technique. Most of existing data-driven techniques in software engineering are based on technical data that is relatively easy to aggregate. When proposed as a means of providing assistance to engineers, techniques are very closely tied to concrete steps of the engineering process and the context of their (potential) use in a tool. While pragmatic, this approach limits the potential of data-driven techniques to enable qualitatively new kind of software tools, such as tools that model development process at the level of a team and provide support in collaboration processes based on rich models of these processes.

In Chapter 6, inspired by existing research that suggests that technical repositories reflect social structure of the team, we set out to detect learning between developers from

1

history of changes in a version control repository.

1.4 Methodology

Studies that comprise this thesis employed a wide range of methods for obtaining data and reasoning about it. In this section we provide a brief overview of related methods and link them to the respective chapters.

1.4.1 Obtaining data

Obtaining technical data

Software engineering studies heavily use historical data mined from software repositories. Software repositories contain a variety of traces of development activity, such as snapshots of code, records of developer communication, and history of builds and test runs. This data is well suited for quantitative analysis, such as finding patterns or building descriptive models [38, 39], and can directly fuel many data-driven techniques, such as recommendation engines [9, 40] or prediction models [34].

In some cases, retrieving historical data from a software repository is a trivial task thanks to support from tools that store this data. This is the case, for example, for obtaining history of commits from Git or retrieving code reviews from Gerrit. Throughout this thesis, we used standard capabilities of existing tools where possible. Moreover, there exist services that provide processed data from large-scale repositories, such as GHTorrent [41] for Github and TravisTorrent [42] for Travis CI. In Chapter 2, we used GHTorrent to create samples of repositories for further analysis.

However, when there is no straightforward way to retrieve the necessary data, seemingly simple mining tasks may require quite some engineering work. In Chapter 5, to reproduce actual reviewer recommendations given by Upsource [43] – a code review tool used in one of the industrial environments in the study – we had to implement a custom build of the tool and connect it to a production database.

To reason about users' behavior, which is critical when a study is practice-oriented, i.e., motivated by improvement of experience of users with a particular tool, researchers need to collect records of user actions. This requires building dedicated testing setups, including user interface mockups and event logging. Real-world tools, however, rarely provide features to capture rich data of user behaviour. In Chapter 5, to retrieve records of reviewer selection in prior reviews, we had to rely on a custom build of Upsource as well.

In some cases, mining constitutes a significant technical challenge because non-trivial processing is required. In Chapter 2 we use a custom setup, based on a graph database, to enable fast file history queries and be able to process millions of files. By precalculating connections between snapshots of individual files, we are able to avoid expensive traversal of the commit tree on every query, which takes place in *git log*. In Chapter 4 and Chapter 6, to prepare code snapshots for use as input for machine learning models, we use a rather complex pipeline that involves parsing code in multiple languages and processing syntax trees. Complexity of these pipelines was the primary inspiration for the tool described in Chapter 3. Finally, collected data may require some manipulation to suit the needs of the experimental setup. For example, in Chapter 4, to study the impact of work context on performance of authorship attribution algorithms, we had to additionally process the

dataset.

Obtaining human data

Some research questions in software engineering involve description and interpretation of human interactions, experiences, and opinions. This may be required to understand the problem domain and how people involved perceive it, assess perceived usability of techniques and tools, or gain deep insights from personal experience of experts. Information of this sort usually cannot be adequately expressed in quantitative data and explained by statistics. Moreover, it is often not available for mining from technical repositories. Answers to this class of research questions require input from people through qualitative data collection methods such as participant observation, interviews, and questionnaires.

In this thesis, we rely on qualitative methods for some of the research questions. In Chapter 5, we use semi-structured interviews and a large-scale questionnaire to evaluate perceived relevance and helpfulness of reviewer recommendation engines, and to explore information needs of software engineers during reviewer selection. In Chapter 6, we use a survey to get reports of peer-to-peer learning from developers. These reports later serve as ground truth for evaluation of the code style vectorization method.

One more qualitative research method is literature reviews, which consist in retrieval and summarization of existing published research on the topic of interest. While none of the chapters of this thesis presents a standalone literature review, every chapter includes a review of related research work to help us position and motivate our research questions.

1.4.2 Reasoning about data

Once the data is obtained, the next step is to process it to derive useful information. One straightforward method of data analysis is descriptive statistics, used to quantitatively summarize a data sample or visualize it. In this thesis, we use descriptive statistics in all chapters to describe datasets in use, provide information about survey participants, or to visualize distributions. Notable examples of use are in Chapter 2, where we describe the degree of presence of difference in file histories between mining approaches and visualize differences in distribution of repository characteristics between ecosystems.

Methods of statistical inference provide a way to use a data sample to reason about a broader population. One method widely used (and sometimes misused [44]) in software engineering research is statistical hypothesis testing. It can be used to judge whether observed differences between data samples represent a difference in underlying populations. With empirical data, where often no assumptions can be made about underlying distributions, non-parametric statistical tests such as Kolmogorov-Smirnov [45], Mann-Whitney [46], and Wilcoxon [47] are oftentimes the only reasonably applicable options. In Chapter 6 we use the Kolmogorov-Smirnov test to compare relative positions of coding style representations for pairs of developers who reported learning from each other to movement of those who reported not learning from each other.

Performance metrics are used to quantitatively evaluate relevance of output produced by data-driven techniques. Basic information retrieval metrics such as precision and recall, as well as derived metrics such as F-measure and ROC-AUC, are particularly useful to evaluate classification models [48], while metrics such as mean reciprocal rank and top-k precision are suitable for recommender systems [49].

1

Performance metrics are widely used in this thesis. In Chapter 5 we use precision and mean reciprocal rank to assess quality of recommender's output and to detect influence of the recommender on choices made by the users. In Chapter 4 we use accuracy as a measure of performance of the authorship attribution model. In Chapter 2 we assess impact of the branch mining strategy on performance of a reviewer recommendation model in terms of top-k accuracy and mean reciprocal rank. In the same chapter, we use an ad-hoc metric for a similar assessment in the case of change prediction.

Analysis of qualitative data involves less formal techniques. Analysis of human responses generally requires additional processing to introduce structure to the data. Sometimes it is possible to map qualitative responses to numerical data (for example, using a rating scale such as Likert-type scales [50], or using sentiment analysis methods [51]). Coding techniques, such as tagging or card sorting [52], allow to reveal structure or hierarchy in a spectrum of human responses or mentioned phenomena, and to extract structured knowledge from a set of responses. In Chapter 5 we use tagging by multiple experts to classify respondents' experience with a reviewer recommender and their information needs.

1.4.3 Open-Source Software

While not a method in itself, one of the pillars of software engineering research is open-source software. Collaboratively developed and widely available, open-source software plays several important roles in research related to data-driven software engineering tools.

Open-source data analysis software allows individual researchers to use pre-built software to build data mining and analysis pipelines. This allows researchers to spend less time and effort on implementation of custom software. In addition, thanks to open-source code and wide use, quality and reliability of popular open-source software is at the level hard to achieve in solo development. Chapter 3 presents our contribution to open-source data analysis tools, primarily meant to minimize effort of other researchers in processing code for machine learning models. Moreover, open-source data analysis pipelines help increase reproducibility of studies. Source code of processing, and data, for all chapters is publicly available to ensure reproducibility. The only exception is Chapter 5, where data processing is made in a closed-source system and data cannot be made public due to industrial secrecy concerns.

Apart from saving implementation efforts, open source ecosystems present a rich source of data for development of data-driven software engineering techniques. Github is the largest collection of software repositories ever available. We use history of code changes from Github and other public repositories in Chapters 2, 4, and 6. Code review tools used by open-source ecosystems enable us to evaluate reviewer recommendation techniques in Chapter 2.

1.5 Thesis Outline

1.5.1 Chapter 2: Importance of branching in Git data

In Chapter 2 we study impact of the assumption of linearity of file change history, that is made, either explicitly or without any discussion, in some studies. Using a large stratified sample of 1,405 open source projects containing over 2 million files in total, we find that

this assumption is generally incorrect and leads to difference in retrieved file histories for 19% of files; 71% of projects in our sample contain at least one such file. Further, we investigate the effect of omitting changes in branches on performance of two data-driven techniques that rely on file histories as a main source of data: change recommendation and reviewer recommendation. We find that use of simplified history consistently leads to slightly lower performance of both techniques across all sample projects. While the drop in performance is not dramatic, it might be particularly important in some cases, such as when a demonstrable improvement of a technique's performance over a prior baseline is crucial.

1.5.2 Chapter 3: A library for mining of path-based representations of code

Chapter 3 describes motivation and implementation of an early version of *astminer* — a library for mining of path-based representations of code. With data-driven techniques based on machine learning growing in popularity lately, data transformation and its preparation to use in ML pipelines require significant additional effort from researchers. We hypothesised that other researchers could benefit from a tool which facilitates extraction of path-based representation and have converted our existing parsing and path extraction code into a reusable library. Within a year of publication, *astminer* has been used by several other researchers in their pipelines, which confirms our hypothesis of its potential usefulness.

1.5.3 Chapter 4: Code authorship attribution on realistic datasets

In Chapter 4 we address the problem of authorship attribution on code. First, we propose a state-of-the-art model for authorship attribution of source code. After that, we discuss limitations of existing synthetic datasets for authorship attribution, and propose a data collection approach that delivers datasets that better reflect aspects important for potential practical use in software engineering. In particular, we discuss the concept of work context and its importance for the whole task of authorship attribution and for performance of existing models for this task. Finally, we demonstrate that high accuracy of authorship attribution models on existing datasets drastically drops when they are evaluated on more realistic data. We conclude the chapter by outlining next steps in design and evaluation of authorship attribution models that could bring the research efforts closer to practical use.

1.5.4 Chapter 5: Does reviewer recommendation help developers?

Chapter 5 presents the first to our knowledge evaluation of a reviewer recommender in practice. First, we set out to find evidence of influence of a reviewer recommender on choices made by the users, by looking for signs of feedback of a recommender on users' choices. Having found no trace of such influence, we turn to the users of the recommender. Through interviews and a survey we find that, though perceived as relevant, reviewer recommendations rarely provide additional value for the respondents. We confirm this finding with a larger study at another company. Confirmation of this finding brings up a case for more user-centric approaches to designing and evaluating the recommenders.

1

Finally, we investigate information needs of developers during reviewer selection and discuss promising directions for the next generation of reviewer recommendation tools.

1.5.5 Chapter 6: Building implicit representations of coding style

In Chapter 6 we propose a new approach to building vector representations of individual developers by capturing their individual contribution style, or coding style. Such representations can find use in the next generation of software development team collaboration tools, for example by enabling the tools to track knowledge transfer in teams. The key idea of our approach is to avoid using explicitly defined metrics of coding style and instead build the representations through training a model for authorship recognition and extracting the representations of individual developers from the trained model. By empirically evaluating the output of our approach, we find that implicitly built individual representations reflect some properties of team structure: developers who report learning from each other are represented closer to each other.

1.6 Origins of Chapters

- **Chapter 2** is based on the paper "Mining File Histories: Should We Consider Branches?" by Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli, presented at the Automatic Software Engineering conference in 2018 in Montpellier, France. The first author designed the study, implemented a data collection tool, obtained all results featured in this thesis, and wrote the paper. Section 5.3 from the paper is not included in this thesis, because source code for its reproduction could not be made available to the first author.

- **Chapter 3** is based on the paper "PathMiner: A Library for Mining of Path-Based Representations of Code" by Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli, presented at the Mining Software Repositories conference in 2019 in Montreal, Canada. The first author envisioned the library, implemented the submitted version, and wrote the paper.

- **Chapter 4** is based on the paper "Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering" by Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. The paper is currently (October 2020) under a major revision during submission to the IEEE Transactions on Software Engineering journal. The author of this thesis, as the second author of the paper, implemented the data collection and processing pipeline and assisted with writing. The first two authors agreed to consider their contributions equal.

- **Chapter 5** is based on the paper "Does Reviewer Recommendation Help Developers?" by Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli, published in the IEEE Transactions on Software Engineering journal in 2018 and presented at the International Conference in Software Engineering in 2019 in Montreal, Canada. The first author came up with the idea for the quantitative

experiment, arranged one of the two industrial collaborations, implemented and ran the experiments in the quantitative part, conducted interviews and took part in survey design in the qualitative part, analyzed the results, and wrote the paper.

- **Chapter 6** is based on the paper "Building Implicit Vector Representations of Individual Coding Style" by Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli, presented at the Workshop on Cooperative and Human Aspects of Software Engineering in 2020 from the first author's bedroom in Delft, The Netherlands. The first author came up with the idea of the approach and the study, implemented the data collection and processing pipeline, and wrote the paper. The first two authors agreed to consider their contributions equal.

1.7 Reproducibility

We publish source code and data for verification and reproduction of the results presented in every chapter, where legally possible. Source code for Chapters 2, 3, 4, and 6 is available on Zenodo[1]. For industrial secrecy reasons, we cannot publish any source code or data for replication of results in Chapter 5.

[1] https://zenodo.org/record/4268742

2

Branches

This chapter is based on the paper "Mining File Histories: Should We Consider Branches?" by Vladimir Kovalenko, Fabio Palomba, and Alberto Bacchelli, presented at the Automatic Software Engineering conference in 2018. [1]

[1] Section 5.3 from the original paper is not included in this thesis, as it could not be reproduced while compiling the thesis due to unavailable source code. The removed section covered one of several application examples, and its exclusion does not affect validity of the included results.

2.1 Introduction

The workflow of modern version control systems (VCS), such as Git, extensively relies on branching. Branching support allows developers to manage multiple isolated versions of the working tree, which can be modified independently of each other. Branch-related operations in Git are by design extremely lightweight compared to older VCSs [53]. Low cost of branching allows branches to be used for development of individual features, for experimenting with design solutions, and for preparing releases [54]. In all these examples use of branches allows teams to keep the main working tree free of questionable code and reduces development overhead related to version conflicts [55].

While being a popular version control system today, Git is quite unfriendly for data mining [53]. In particular, branching features introduce issues for miners: branches can be removed and overwritten, and synchronization with the remote repository can introduce implicit branches with no logical meaning [53].

Despite the difficulties with analysis of Git history, mining of historical data from VCS is still the basis for a variety of studies, which quantitatively explore the development process and suggest approaches to facilitate it [1, 17, 56–62].

History of individual files is a particularly important source of information for prominent practical applications, like (i) defect prediction algorithms, where metrics based on file history are important features [57, 63–65], (ii) code ownership heuristics [66–68], which are based on aggregation of individual contributions of all authors of the file, and (iii) code reviewer recommendation [20, 69–71], where history of prior changes to files serves as a basis for automatic selection of the expert reviewers.

Pitfalls of Git from the data mining perspective pose common threats to validity of every of such studies. Some of these threats, such as mutability of history, are commonly acknowledged by researchers (e.g., [61, 72–75]). Nevertheless, there is no widespread approach to handling of merge commits and branches during mining. Moreover, MSR studies often do not provide a detailed description of mining algorithms, and handling of the branches in particular, or explicitly focus the analyses on the main branch of the repository [72].

In this study, we aim at making a first step toward the assessment of the threats arising from not considering full information about branching in mining software repository studies. Specifically, we focus on the impact of the branch handling strategy on extraction of a file modification history. This task requires a traversal of a repository graph to collect individual commits affecting the file. We first perform a preliminary analysis on how the mining of file histories is impacted by branches, by measuring how much *first-parent* (i.e., history extractable when only considering the first parent of each commit when traversing the repository) and *full* (i.e., the history extractable when considering changes in all branches) file histories differ from each other. Then, we study how performance of two MSR applications (code reviewer recommendation and change recommendation), that use file modification histories as input data, varies when considering branches.

Our results show that the first-parent and full mining strategies consistently result in different file histories, even though the scale of the difference varies across software ecosystems and repositories within each ecosystem. We find that considering the full file histories leads to an increase in the considered MSR-based techniques' performance that is rather modest. This marginal increase indicates that our findings do not raise any serious

questions on the validity of studies that simplify the mining approach. Nevertheless, in our work we devised a method and a tool for efficient mining of full file histories at scale, which we make publicly available [76].

2.2 Background

Several prior studies focus on the use of branching and its added value for developers. Combined, these studies provide strong evidence of importance of branching in modern software development. Appleton et al. [77] explore an extensive set of branching patterns and propose a number of best practices and antipatterns. Buffenbarger and Gruell [78] devise practices and patterns to facilitate efficient parallel development, mitigating the complexity of branching operations in early VCSs. Bird et al. [79] conclude that developers working in a branch represent a virtual team. Barr et al. [54] claim that lightweight branching support is the primary factor in rapid adoption of modern distributed VCSs in OSS projects. Bird and Zimmermann [80] identify common problems from improper branch usage and propose an analysis to assess more efficient alternative branch structures. Shihab et al. [81] find that the excessive use of branches is associated with a decrease in software quality.

Today's most popular version control system — Git — was not designed to preserve a precise history of modifications [82], which implies difficulties with the analysis of these histories [53, 83]. Analysis of software version histories is not only used to study the development practices, but also to facilitate development with data-driven tools. Prominent examples of applications for tools heavily relying on histories of changes of individual files are defect prediction [84–86], code reviewer recommendation [20, 69], and change prediction [8, 87]. Notably, the complexity of Git, the mutability of its data structure, and the difficulty of figuring out the parent relationships between revisions complicate the work of researchers and prevent some practitioners from using it as their version control system [88]. Being able to accurately retrieve histories of prior changes is vital for efficient use of techniques that are based on histories. Moreover, in some cases histories need to be processed to achieve optimal performance of the techniques: For instance, Kawrykow and Robillard [89] show that removing non-essential changes from modification histories improves the performance of co-change recommendation [8].

2.2.1 Motivation

Version control repositories are the key data source for a wide variety of software engineering studies [57, 61, 64, 73, 90]. With no widespread high-level mining tool in use, the common way for the researchers to mine the histories of repositories is to use homegrown tools based on low-level libraries, such as JGit [91]. While low-level operations provide greater flexibility of mining, they also undermine the reproducibility of studies, as details of mining are usually not elaborated on in the papers. Reproduction packages, where available, commonly contain information obtained after mining, but not the repository mining scripts.

Restoring the actual change history from a Git repository is challenging and error-prone [53]. To come around the difficulties, some studies (e.g. [72]) focus on the development activity in the main branch, thus omitting part of the changes in the repository. This

approach may be sufficiently precise for some applications, because (i) in some repositories most of development activity takes place in the master branch, and (ii) the `rebase` operation is often used to integrate changes from branches into the main branch. However, consideration of branches and careful handling of merge commits might be important for precise calculation of individual file histories, which are the primary source of input data in some contexts, such as code reviewer recommendation [69] and change recommendation [8].

The difference in quality of data between different mining approaches, and the impact of the chosen mining approach on performance of analysis methods driven by historical data, are not clear and have not yet been explored. We conduct this study to quantify the effect of considering the graph structure of the repository (importance of such consideration is reported as one of the perils of mining Git [53]) and to investigate how the difference in the results from different mining approaches impacts performance of MSR applications.

While the file histories are the main input data for a variety of MSR-based techniques, there is no guarantee that more complex and precise mining methods ensure an increase in performance of the techniques notable enough to justify the extra mining effort. With no prior research existing on this topic, with this study we seek not only to identify the impact of branch handling strategy on performance of file-history-based methods, but also to compare the scale of this impact between different techniques. This knowledge could help make a step towards ensuring that MSR studies and their practical applications employ optimal mining strategies to get the most value out of the repository data.

2.2.2 Challenges of Mining the File Histories

Mining of histories of individual files at large scale is a non-trivial task. Git provides a toolkit for repository operations, including `git log`, which facilitates retrieval of logs of commits. However, Git was not designed to support careful storage and retrieval of history of changes [82, 92–94], which implies several complications with using `git log` for mining file histories. Specifically:

Performance With no specialized index for file histories in place, retrieval of histories of changes for an individual file requires traversal of the commit graph. Retrieval of histories for every file in the repository tree is very expensive.

Handling of renames A Git repository does not contain any records of renames and moves of files. Such events are detected based on similarity of contents of consecutive versions of a file, with thresholds defined by the settings of the Git client. As a result, calculated history of the same file in the same repository might appear different on different clients.

Handling of merge commits and branches The `git log` tool, which is often used for analysis of software histories, supports an overwhelming variety of settings, with over 100 argument options [95]. Unless the tool is thoroughly set up, some potentially interesting changes might be implicitly omitted: for example, by default `git log` prunes some side branches. A default approach might not be suitable for some applications.

Difficulty of use A wide variety of settings makes the user experience with `git log` quite complicated. Certain scenarios of retrieval of repository history are even harder: for

example, traversing the commits graph forward in time, which might be useful in some contexts, is more difficult. For example, all descendants of a commit can be retrieved with the following command:

```
git rev-list --all --parents | grep ".\{40\}.*SHA.*" |
awk '{print $1}'
```

This command [96], which only lists the commits without providing any information on the structure, is already not trivial.

2.2.3 Retrieval of File Histories

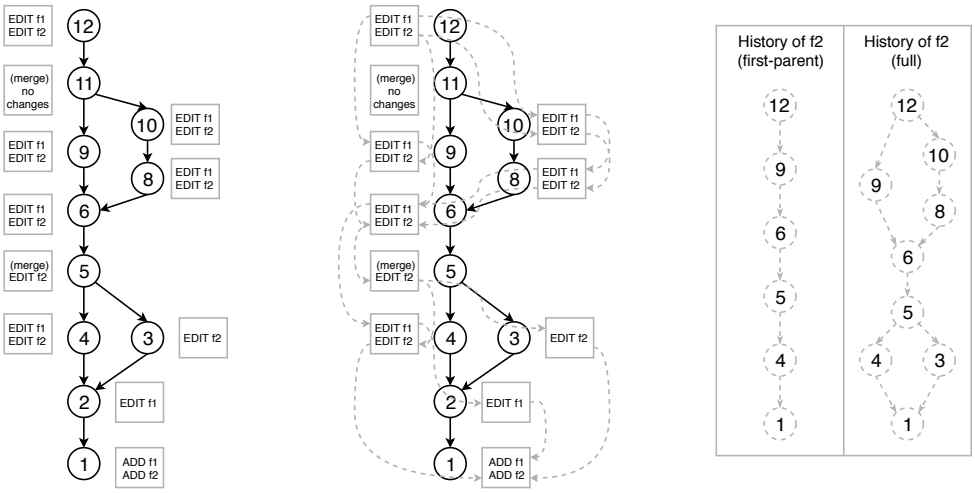


Figure 2.1: Construction of file parent connections and resulting file histories

History of commits in a Git repository can be represented as a directed acyclic graph. Each commit logically represents a state of the repository's file tree. Each version of the state is based on one or several prior *(parent)* version, and only the difference in the state between the parent and the current version is actually stored.[2] In Figure 2.1 (left), which presents a hypothetical commit tree, the commit parent relationships are represented by black arrows: for example, commit 5 is the parent of commit 6. Merge commits, which integrate changes from multiple branches, have more than one parent. In Figure 2.1, commit 5 has two parents: 3 and 4. In a merge commit with multiple parents, the list of parents is sorted: if branch *A* is merged into branch *B*, the first parent of the merge commit would be the one that branch *B* was pointing to before the merge. In Figure 2.1, parent commits are sorted left to right: commit 4 is the first parent of commit 5.

[2]Described is a simplified scheme of the storage model of Git, which is in reality more complex.

Each commit affects a set of files[3] and defines their new content relative to their content in the parent commit. It is possible to say that a commit *affects* a file, if its content in the revision which is represented by the current commit is different from its content in the parent commit, or if the file was created/removed in the current commit. It is common to think of a commit as simply a set of changes in one or multiple files. This simple model is convenient and is used by Git itself, e.g., in the `git diff` command. In Figure 2.1, affected files are shown in boxes next to nodes of the commit tree.

Retrieval of a history of changes for a given file — i.e., list of the commits that affect this file — one needs to traverse the commit graph to identify such commits. A traversal and handling of commits one by one is necessary because Git does not store any auxiliary data that would allow to perform this operation faster. During the traversal, it is possible to either follow all parents of a merge commit, which ensures visiting every transitive parent of the starting point and including all of these commits in the history, or only follow the first parent.

We refer to the traversal strategy that follows all parents and to resulting histories as *full*.

Considering the example repository graph in Figure 2.1, full traversal starting from the latest commit in the repository (12) would include all 12 commits in the repository. A full history for the file *f2* would contain all 9 of these commits that affect *f2*. An alternative strategy is to only follow the first parent of every merge commit during the traversal. We refer to this strategy and resulting file histories as *first-parent*. For the example in Figure 2.1, such traversal starting from commit (12) would only include commits (12), (11), (9), (6), (5), (4), (2), and (1). A first-parent history of *f2* would only include 6 commits from this traversal that affect *f2*. It is important to note that the first-parent strategy does not omit merge commits that contain changes to the file relative to its first parent: an example of such commit in Figure 2.1 is (5), and it is included in the first-parent history as well.

The histories for the file *f2*, as retrieved with both strategies, are presented in Figure 2.1 (right). As the first-parent history contains less commits, some changes to the file are omitted. Thus, using the first-parent strategy to calculate quantitative properties of file histories, such as number of changes or number of contributors, leads to incomplete results.

The two cases represented by the two strategies are rather extreme: the simplified strategy omits all changes that were made outside the main branch and did not end up in the main branch after a rebase. There are less radical ways of simplified handling of branches than to omit traversing the branch completely. For example, one way to see the summary of changes in a branch is to inspect the output of `git diff` between two parents of a merge commit. This approach is adopted by some GUI applications on top of Git, such as Sourcetree[4]. While this approach allows to retrieve a summary of changes in a branch, the individual changes, possibly made by different authors, are presented together and are not distinguishable, which makes the approach less applicable for mining tasks: it is impossible to identify individual contributions by count, sizes, or dates of changes per author.

We find the extreme case of comparing full and first-parent histories an appropriate setting to study the impact of the mining strategy on the results of mining and performance

[3] The set can be empty, e.g., in most of merge commits without conflicts.
[4] https://www.sourcetreeapp.com/

of the methods based on these results. With the two extreme cases, we have the highest chance of identifying effects not present in less extreme settings. As this study is the first to explore such effects, we consider this setting fruitful to highlight the directions for future work.

2.3 Methodology

2.3.1 Research Questions

We center our investigation around two research questions. First, we set out to quantify the importance of careful handling of branches during mining. In particular, we (i) explore the repository structure to calculate numbers of commits reachable with and without considering branches and (ii) analyze differences on a lower level of history of individual files. Thus our first research question is:

RQ$_1$. *How does the branch handling strategy impact the results of mining?*

Having identified the magnitude of the difference between mining approaches that consider and do not consider branches, we investigate the impact of branch handling strategy on the performance of algorithms relying on history of files, such as reviewer recommendation and change recommendation. Thus, our second research question is:

RQ$_2$. *How important is the branch handling strategy for applications?*

2.3.2 Mining Histories at Large Scale

Traversal of the commit tree can take a significant amount of time: in some repositories the tree contains hundreds of thousands of commits. If this operation has to be repeated for many or even all files in the repository — which is often the case during mining the repository for file histories — it can take a lot of time for larger repositories. Slowness of mining the file histories is a limitation of the storage model of Git, which does not link individual versions of a file to each other, and does not include indices of changes to individual files. To overcome this limitation, we devised an alternative representation of changes in Git, aimed at fast retrieval of file histories. We use a graph database engine[5] to store a representation of the commit graph. To keep the database compact, we only store the commit nodes and records of affected files, excluding their content.

In addition to the commit parent relationship, which is the basis of the commit graph in Git, we introduce the concept of *parent file versions*. A parent version of a file can be defined as a change to the same file in some other commit, which can be reached from the current commit with a unique path over the commit parent graph. In Figure 2.1 (center), the file parent connections are represented by gray arrows. By processing the repository, we build the graph of parent relationships between file versions, and store it in the graph database aside the commit graph. Once the file parent graph is built, retrieval of prior changes to the file is as simple as retrieving all transitive parents of the current version, which only requires traversing the file parent graph, which is much faster than traversing the whole repository thanks to direct links to parent file versions. For large-scale mining

[5]Neo4J: https://neo4j.com/

tasks this approach saves time: after processing the repository and building the file parent connections, it is possible to retrieve full modification histories for several thousand files per second. Figure 2.1 (right) presents the resulting histories from traversal of the file parent graph.

2.3.3 Target Systems

We take a number of steps to ensure diversity in our target systems [97]. Our dataset consists of 260 repositories from Github, selected in a stratified manner to include projects of different scale. Using GHTorrent [41], we randomly sample 100 repositories with over 10,000 commits recorded in GHTorrent database for each, and 200 repositories with 1,000 to 10,000 commits. We included more medium-sized repositories than larger ones in the sample because they are more frequent on GitHub. 40 of the 300 repositories turned out not to be publicly available anymore; the rest comprise our Github sample. Counts of commits as retrieved from GHTorrent are not completely accurate, so actual distribution of sizes of repositories in the sample is slightly more disperse. We have attempted to download and mine all repositories from the Apache open source ecosystem, of which we succeeded with 441 repositories of 532. The rest of the repositories were not available, empty, or failed to process with our toolkit.

Finally, we include 395 and 309 repositories from Eclipse and OpenStack respectively, which use Gerrit for code review. The repositories in these two samples belong to the projects concerned by the latest 100,000 reviews in each Gerrit instance, which we have mined to evaluate the performance of a reviewer recommendation algorithm (Section 2.5.1). We use smaller subsets of repositories for parts of RQ2. For change recommendation we use samples of 10 repositories from each of Eclipse and Apache ecosystems. To evaluate reviewer recommendation performance, we use the repositories of the 20 most active projects from each of Eclipse and OpenStack ecosystems (Section 2.5.1). For the quantitative analysis of repository structure and file histories (RQ1), we use all 1,405 repositories from the four samples. We report the entire list of repositories in our online appendix [76].

2.4 RQ1: Difference in mining results

Our first research question seeks to quantify the differences resulting from the application of two different mining approaches to retrieval of the history of files in Git.

2.4.1 Methodology

We devised a set of metrics that quantify the effect of strategy of branch handling on the results of mining. Afterwards, we compared two approaches in terms of these metrics,

Table 2.1: Overview of the target ecosystems

Ecosystem	Projects			Commits			Files				
	total	with difference		total	in projects with difference		total	with difference		in projects with difference	
Github	260	173	(67%)	872,833	257,156	(29%)	322,783	85,170	(26%)	225,861	(70%)
Apache	441	305	(69%)	998,910	938,032	(93%)	861,196	136,692	(16%)	525,997	(61%)
OpenStack	309	301	(97%)	1,317,165	1,317,004	(100%)	87,382	47,634	(55%)	87,166	(100%)
Eclipse	395	220	(56%)	1,000,997	883,318	(88%)	874,044	127,833	(14%)	712,315	(82%)
Total	**1,405**	**999**	**(71%)**	**4,189,905**	**3,395,510**	**(81%)**	**2,145,405**	**397,329**	**(19%)**	**1,551,339**	**(72%)**

i.e., the *first-parent* one (which extracts history only considering the first parent of each commit when traversing the repository) and the *full* one (which extracts all the commits by exploiting the approach described in Section 2.2.3).

For each repository, we first compute descriptive measures of its structure, such as number of commits that are reachable from HEAD (the latest commit in the main branch), number of merge commits (with more than one parent), number of files in the repository, and number of unique contributors to the repository. We use these metrics to compare the ecosystems between each other, and to explore the variation in branching activity within and between them, which is important for mining: for example, differences between the numbers of commits reachable depending on the traversal strategy denote the importance of the strategy for mining: if only the first parent is considered and traversed, some commits are left out, while still contributing to the state of the repository at HEAD. Number of merge commits can be used as a proxy measure of branching activity in the repository.

Beyond repository-wide metrics, the way in which branches are handled also impacts the calculation of histories of individual files: if only the main branch is considered, some changes from file history are omitted. This effect might impact various applications of file histories, ranging from identification of contributors to a file to more complex scenarios such as reviewer recommendation.

To quantify this effect, we calculate histories of every file in the repository, using both the *first parent* and the *full* approaches to retrieve all commits that contribute to a given version of a file. For every repository (all of which contain over 2 million files in total), we calculate repository-wide average length of the history of a file in its tree, when retrieved via first-parent and full method. We compute the ratio of these averages, and fraction of files for which the two methods deliver different histories among all files in the repository. For files with non-linear history, we count the total number of changes to this file as its history length. In addition, we calculate numbers of contributors to each file for both methods of file history mining, and compare their repository-wide averages.

2.4.2 Results

Descriptive Metrics

To display the natural differences between the ecosystems, which are not associated with different mining strategies, we first present the comparison of the ecosystems in terms of natural activity metrics, such as sizes of repositories and number of contributors. The top two rows of Figure 2.2 present a comparison of descriptive metrics between the repositories in the four subject ecosystems. The numbers of commits in the repositories within each ecosystem greatly vary. A median number of commits in a repository ranges from 172 for Github to 1,039 for OpenStack. These numbers include merge commits. An average repository contains several hundred files in the file tree at HEAD and this number varies in all ecosystems, with a lower variation for OpenStack. Projects from GitHub are typically developed by only a few authors – the median number of contributors for a Github repository is 4. This value for OpenStack is 61, with Eclipse and Apache falling in the middle.

For measures of branching activities, repositories from three of the four ecosystems display moderate values: The majority of commits in a typical repository from every ecosystem except OpenStack is reachable from HEAD. OpenStack also stands out in numbers

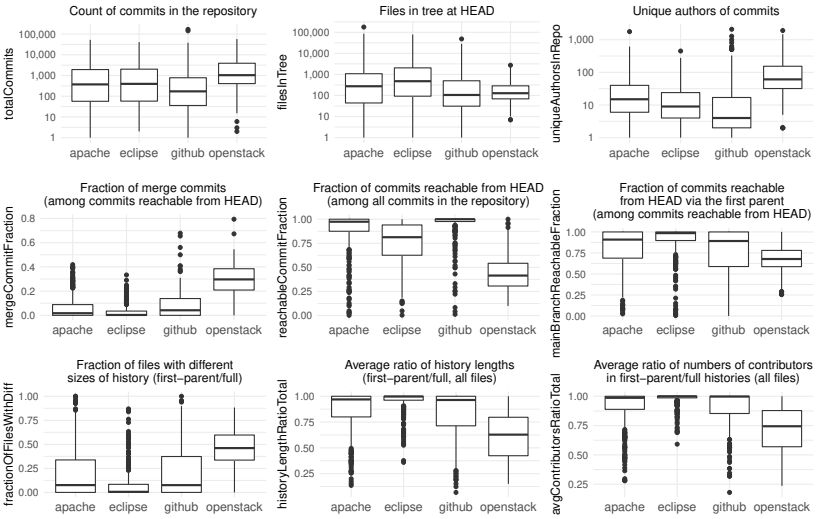


Figure 2.2: Comparison of descriptive metrics between repositories in different ecosystems

of merge commits. In a median project, almost 30% of commits in the repository that are reachable from HEAD are merges (Figure 2.2). Along with higher branching activity, OpenStack repositories show the highest difference between numbers of commits reachable from HEAD via the first parent and via all parents.

Notably, repositories from Github are much more diverse in terms of branching activity metrics. We attribute this diversity to the fact that projects in other ecosystems are logically connected, with possibly common engineering guidelines and intersecting development teams. In addition, the branching structure of the repositories is possibly impacted by the strategy of integrating the pull requests, which are a common part of the workflow at Github and can be either merged or rebased.

File History Metrics

Table 2.1 presents the statistics on the four ecosystems with regard to difference in first-parent and full file histories. Over the four ecosystems, 19% of files display difference in histories retrieved via first parent and full methods. Ecosystem-wide fractions vary from 14% in Eclipse to 55% in OpenStack. 71% of the repositories in our samples contain at least one file with difference in the history. Fraction of such repositories varies between ecosystems from 56% in Eclipse to 97% in OpenStack. 81% of all commits and 72% of all files belong to such repositories, which indicates that the difference between first-parent and full histories is significant in most of the repositories and cannot be ignored as a rare effect.

Distributions of the metrics related to difference in the results of mining file histories are presented in the bottom row of Figure 2.2. One metric that indicates the importance of the strategy of file history mining for a given repository is number of files for which histories retrieved via the first parent and via the full traversal have different lengths. Such

files exist in 305 of 441 repositories (69%) from Apache, 173 out of 260 (67%) from the Github, 220 of 395 (56%) in Eclipse, and in 301 of 309 (97%) OpenStack repositories.

Similarly to the merge activity metrics, fraction of files with difference in history greatly varies within every ecosystem. Median proportion of such files is 8% in Apache and Github, under 1% in Eclipse, and 46% in OpenStack repositories. Distribution of ratio of the length of the two histories across repositories that contain files with the difference in histories (naturally, this metric is only defined for such projects), displays a similar behaviour across ecosystems to fraction of files with the difference in history. This ratio for Eclipse repositories is the highest of the four (the histories are the most similar; median ratio is 0.997) and the lowest for OpenStack (0.63). If we omit the files with identical first-parent and full histories and only consider the files with the difference, an ecosystem-wide median value for average ratio of history lengths in a repository ranges from 0.54 to 0.82 for OpenStack and Eclipse, respectively.

The values for the aforementioned metrics indicate that—in most of the repositories from all four ecosystems—the first-parent strategy leaves out a significant number of changes. The proportion of files affected by the difference and the scale of this difference are the lowest for Eclipse and highest for OpenStack.

One straightforward practical application of file histories is retrieval of contributors to a file. Developer tools with code viewing features, such as Github, display contributors to the current file in the user interface of file content display. Figure 2.2 displays the ratios of counts of version control user records in first-parent and full histories. While the ratio is close to 1 for most of the repositories in Eclipse, Github and Apache, there are quite a few repositories with large difference in each of these ecosystems. For most of the repositories in OpenStack, retrieval of histories via the first parent of the commit leaves out over 25% of contributor records (median value of the ratio is 0.74). Only considering the files with different histories naturally leads to an even stronger effect.

Summary

The analysis of the results of mining across over 1,400 repositories in the four subject ecosystems reveals that—in most of the considered repositories—the two strategies of mining file histories deliver different results (Figure 2.2, Table 2.1). The size of the difference for a typical repository varies across the four ecosystems and is the lowest for Eclipse and the highest for OpenStack. The size of the effect aligns with the proportion of the merge commits in repositories and reachability of commits (Figure 2.2). The difference between the histories does not only impact their quantitative metrics, but also distorts the observed numbers of contributors to a file. We explore the impact of the mining technique on the more complex repository analysis techniques relying on file histories in the next research question (RQ2, Section 2.5).

2.5 RQ2: Importance for applications

To evaluate importance of the mining approach for practice, we compare the performance of two prominent techniques based on histories of changes: (i) recommendation of code reviewers [69] and (ii) change recommendation [8].

It is important to note that, while we compare the numbers of performance of the techniques, we do not perform statistical tests to assess the significance of the difference across

Table 2.2: Results of reviewer recommendation evaluation on projects from Eclipse and OpenStack

			MRR			Top 1 precision			Top 2 precision			Top 3 precision			Top 5 precision			Top 10 precision			Changes per review		
Project (OpenStack)	**Reviews**	**Reviews w/ diff**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**ratio**
openstack/tripleo-heat-templates	2,588	2,477	0.16	0.19	**0.03**	0.13	0.10	**-0.02**	0.17	0.18	**0.01**	0.19	0.23	**0.03**	0.21	0.30	**0.09**	0.22	0.38	**0.16**	7.75	76.46	**9.87**
openstack/releases	2,107	1,990	0.16	0.21	**0.05**	0.10	0.10	**0.00**	0.19	0.28	**0.09**	0.22	0.32	**0.11**	0.24	0.35	**0.11**	0.24	0.36	**0.12**	9.81	26.56	**2.71**
openstack/cinder	2,073	2,049	0.02	0.02	**0.00**	0.01	0.01	**0.00**	0.02	0.02	**0.00**	0.02	0.02	**0.00**	0.02	0.03	**0.01**	0.03	0.04	**0.02**	17.41	157.62	**9.05**
openstack/requirements	1,786	1,771	0.05	0.07	**0.03**	0.01	0.01	**0.01**	0.02	0.02	**0.00**	0.02	0.04	**0.02**	0.08	0.12	**0.04**	0.14	0.29	**0.15**	255.40	1,534.32	**6.01**
openstack-infra/zuul	1,367	1,329	0.10	0.13	**0.03**	0.05	0.04	**-0.01**	0.08	0.10	**0.02**	0.11	0.14	**0.03**	0.15	0.23	**0.08**	0.24	0.35	**0.12**	65.85	178.63	**2.71**
Total (OpenStack top 5)	**9,921**	**9,616**	**0.10**	**0.13**	**0.03**	**0.07**	**0.06**	**-0.01**	**0.10**	**0.13**	**0.03**	**0.12**	**0.16**	**0.04**	**0.15**	**0.21**	**0.06**	**0.17**	**0.29**	**0.11**	**62.79**	**359.34**	**5.72**
Total (OpenStack top 20)	**25,179**	**24,297**	**0.13**	**0.15**	**0.02**	**0.09**	**0.08**	**-0.01**	**0.14**	**0.15**	**0.01**	**0.16**	**0.19**	**0.03**	**0.19**	**0.24**	**0.05**	**0.21**	**0.31**	**0.10**	**40.05**	**210.50**	**5.26**

Project (Eclipse)	**Reviews**	**Reviews w/ diff**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**delta**	**first parent**	**full**	**ratio**
papyrus/org.eclipse.papyrus	4,884	3,775	0.26	0.26	**0.00**	0.18	0.18	**0.00**	0.27	0.26	**0.00**	0.31	0.31	**0.00**	0.35	0.36	**0.00**	0.41	0.42	**0.02**	48.10	68.46	**1.42**
jgit/jgit	4,842	4,197	0.42	0.42	**0.01**	0.30	0.31	**0.00**	0.43	0.43	**0.00**	0.50	0.51	**0.00**	0.58	0.59	**0.01**	0.64	0.67	**0.03**	44.74	78.38	**1.75**
linuxtools/org.eclipse.linuxtools	4,616	2,910	0.42	0.41	**-0.01**	0.29	0.28	**-0.02**	0.46	0.44	**-0.02**	0.55	0.53	**-0.02**	0.59	0.59	**0.00**	0.62	0.62	**0.01**	42.46	56.11	**1.32**
egit/egit	4,587	4,028	0.35	0.36	**0.01**	0.23	0.23	**-0.01**	0.33	0.33	**0.00**	0.40	0.41	**0.01**	0.51	0.56	**0.05**	0.64	0.67	**0.03**	98.74	140.66	**1.42**
platform/eclipse.platform.ui	4,083	2,486	0.26	0.23	**-0.03**	0.15	0.12	**-0.03**	0.25	0.21	**-0.04**	0.32	0.27	**-0.04**	0.40	0.36	**-0.04**	0.50	0.48	**-0.02**	117.53	145.60	**1.24**
Total (Eclipse top 5)	**23,012**	**17,396**	**0.34**	**0.34**	**0.00**	**0.24**	**0.22**	**-0.01**	**0.35**	**0.43**	**-0.01**	**0.42**	**0.41**	**-0.01**	**0.49**	**0.50**	**0.01**	**0.56**	**0.57**	**0.01**	**68.67**	**96.15**	**1.40**
Total (Eclipse top 20)	**55,620**	**36,795**	**0.37**	**0.37**	**0.00**	**0.25**	**0.24**	**-0.01**	**0.38**	**0.37**	**0.00**	**0.46**	**0.46**	**0.00**	**0.55**	**0.55**	**0.01**	**0.62**	**0.63**	**0.01**	**103.37**	**122.57**	**1.19**

Total (Eclipse top 20 + OpenStack top 20)	**80,799**	**61,092**	**0.30**	**0.30**	**0.01**	**0.20**	**0.19**	**-0.01**	**0.30**	**0.30**	**0.00**	**0.37**	**0.37**	**0.01**	**0.43**	**0.45**	**0.02**	**0.49**	**0.53**	**0.04**	**83.63**	**149.97**	**2.45**

the broader population of projects, because we are primarily interested in comparing the performance of two methods within each individual project, as this is the most reasonable way to assess the practical importance of the effect.

2.5.1 Code Reviewer Recommendation

Methodology

Recommendation of reviewers for code review has the goal of finding the most qualified reviewer for a new code change committed on a repository [69]. Such recommendation tools usually mine the change history information to identify the developer that is more expert on the piece of code impacted by the change under review. Thus, it represents a prominent example of usage of file histories to assist developers in routine tasks. In the last few years the reviewer recommendation algorithms have been adopted by industrial code review tools, such as Github [98], Gerrit [99] and Upsource [43].

We focus on two open source ecosystems that use Gerrit for code review — Eclipse [100] and OpenStack [101]. For each of the two Gerrit instances [102, 103], we extract the 100,000 most recent code reviews. We choose this number to have a sufficiently large dataset, which would, however, not include all of the reviews in the corresponding instances, but only two sets, similar in size, of the most recent reviews from each instance. We assess the impact of the file history retrieval method on accuracy of recommendations of code reviewers based on the history of changes. We perform the evaluation to find out whether the accuracy of recommendations changes depending on method of file history retrieval, but not to achieve maximum possible accuracy. Thus, we resort to a trivial reviewer recommendation algorithm, based on counts of developers' prior contributions to the files under review. We evaluate the recommendations by comparing a list of recommendations with actual reviewers of a changeset, as recorded in Gerrit. To assess the accuracy of recommendations, we use two commonwise metrics: Mean Reciprocal Rank (MRR) and top-k precision [104].

Results

Table 2.2 presents the results of evaluation of reviewer recommendation algorithm based on authors of past changes to files under review. We compare the accuracy of the algorithm between the two variations of input data: first-parent and full file histories. For each of the two ecosystems — Eclipse and OpenStack — we compare the accuracy numbers for 20 projects in each ecosystem. The selected projects are the most represented among 100,000 latest code reviews in the corresponding Gerrit instance. We report values of mean reciprocal rank and top-k accuracy for k in $\{1,2,3,5,10\}$ (average for all reviews in the project) for the recommendation lists based on both first-parent and full histories of files under review, and explore the difference between these values. To keep the table compact, we only report the 5 most active projects from each ecosystem individually, and aggregated values for the top 5 and top 20 projects. To illustrate the scale of difference between file histories, we also report average counts of commits in the union of histories of files under review, for both methods of retrieval of file histories, and the ratio of these numbers for the two methods.

For the Eclipse ecosystem, the difference in recommendation accuracy is subtle: MRR and all 5 top-k precision values only vary very slightly between first-parent and full histories consistently across all projects. For the OpenStack ecosystem the difference is slightly more pronounced. While MRR values only differ slightly, top-k precision values differ increasingly for higher values of k. This difference indicates that the full histories of files may include changes, made by future reviewers of a file, that are not present in the first-parent histories. The increase in the size of the effect with the increase of k suggests that authors of such changes are typically not the main contributors of the file, as they end up around k-th position in the list of past contributors sorted by numbers of contributions, thus starting to affect the top-k precision value for the corresponding k and higher.

The difference in the size of the effect betwen the two ecosystems can be explained by the fact that the difference between the full and first-parent histories is much higher for OpenStack repositories than for Eclipse. In OpenStack, the full histories of commits for all files under review, in union, contain on average 5.26 times as many changes as the first-parent histories. In Eclipse, the average difference between sizes of histories is under 20%. Such a low difference is unlikely to cause a large deviation of the two sorted contributor lists, and is thus not critical for the accuracy of reviewer recommendation based on history of changes. A large difference in OpenStack, however, noticeably impacts the accuracy of reviewer recommendation.

2.5.2 Change Recommendation

Methodology

Another example of usage of historical data to improve the user experience of development process is recommendation of changes, based on mining of association rules for changes to individual files. A practical application for this technique was described by Zimmermann *et al.* [8]: in particular, given a new code change as input, their technique suggests related changes that the developer might want to apply based on the files that frequently change with the modified file. We perform an experiment to assess the effect of using full histories of changes to a file, compared to using the first-parent histories.

The original design by Zimmermann [8] uses a set of changes from version control

to infer file association rules. To apply the approach to our area of focus — difference in results of mining of individual files — we adapt the design of the original tool. We use two different approaches to infer the association rules from past changes, and evaluate their performance in predicting a change of a given file in the commit. The common part of the two algorithms is their context and input data.

Below we use the notation $F \in C$ to denote that the commit C affects the file F, i.e., the file is *modified* in it. Both algorithms try to predict the change to the file $F_{current}$ in the current commit $C_{current}$.

In the first algorithm (*"single-file"*), we infer the association rules from the commits that affected this file in the past: $\{C : C_i \textit{ affects } F_{current}\}$.

In the second algorithm (*"other-files"*) we infer the rules from the commits that affected every of the other files in the commit C: $\{C : C_i \textit{ affects } F_k, F_k \in \{F : F_i \in C_{current}\} \setminus \{F_{current}\}\}$

We evaluate both algorithms and compare their performance depending on the type of file histories in use: first-parent or full. Intuitively, a full history contains more information about past changes, which allows one to infer more association rules, some of which are more likely to match the current change. However, since we are interested in difference between the two methods of file history mining in terms of their capability to provide information to infer the association rules from past changes (quantitative difference is explored by $\mathbf{RQ}_1$), we make adjustments to account for the difference in sizes of the two histories: (1) We only include the predictions where the histories are different (otherwise they perform equally); (2) We trim the full history to the size of the first-parent history, taking the most recent commits into account (to infer rules from the same number of commits); (3) We sort the rules by *support* and trim the larger ruleset to match size with smaller (to account for the possible difference in the number of rules).

In addition, to bring the algorithm closer to a practical approach, we only generate the predictions when the following (empirically derived) criteria are met: (i) We do not consider large commits with more than 10 changes when infering the rules (they rarely represent meaningful changes); (ii) We only execute the algorithm when the smaller of two histories contains at least 5 commits (otherwise the history is too trivial to learn meaningful rules from); (iii) We use at most 100 most recent commits from the history to infer the rules (to capture the current state of logical coupling between files); (iv) We use at most 10 rules with the highest support values (recommendation lists are finite and small in practical contexts). We use an open source implementation [105] of the Apriori algorithm [106] to infer the association rules. A formal definition of the concept of an association rule is available in literature [8].

Imitating a real-life context of recommendation of changes, we derive a recommendation set from the set of rules as a union of all one-item sets of *heads* of the rules, *bodies* of which are fully contained among the other files changed in the commit C.

We use a random sample of 10 projects from each of Apache and Eclipse ecosystems for evaluation.

Results

Table 2.3 presents the results of evaluation of change recommendation. An "event" corresponds to a single case when association rules have been successfully generated using both

Table 2.3: Comparison of performance metrics for change recommendation, by mining approach and ecosystem

Apache

Algorithm	History type	Events count	Recommendations (average)	Rules (average)	Success rate	Failure rate	No prediction rate
Single file	Full	8780	0.908	8.848	0.496	0.029	0.474
	First parent	8780	0.858	8.848	0.474	0.031	0.496
Other files	Full	19072	1.565	8.003	0.184	0.528	0.288
	First parent	19072	1.485	8.003	0.170	0.527	0.303

Eclipse

Algorithm	History type	Events count	Recommendations (average)	Rules (average)	Success rate	Failure rate	No prediction rate
Single file	Full	2721	0.772	7.922	0.491	0.016	0.493
	First parent	2721	0.729	7.922	0.483	0.016	0.501
Other files	Full	6661	1.324	7.540	0.163	0.514	0.323
	First parent	6661	1.299	7.540	0.152	0.526	0.322

first-parent and full histories. In some events, none of the rules match the set of changes in the commit, so no recommendations can be produced. Rate of such events is presented in the last column of Table 2.3. The setup and the algorithms are described in detail in Section 2.5.2. In the context of this study, we are interested in comparing the performance of full and first-parent histories as the input data for each of the two algorithms.

The "other files" algorithm, which uses histories of the other files in the commit to produce the association rules, produces more events and generates more recommendations on sampled repositories from both Apache and Eclipse. However, only under 20% of these recommendations are successful. The "single file" algorithm, using rules inferred from the history of a single file, generates less recommendations, of which around a half are successful. The full histories perform slightly better as the input data for both algorithms in both ecosystems: depending on the ecosystem and algorithm, they yield 5 – 8% more recommendations, which are 2 – 8% more likely to be successful (i.e., to match an actual change).

2.6 Limitations

A number of limitations affect the results of our study.

1. When designing the experiments to compare performance, we tried to follow the original approaches as closely as possible, but it was not always possible completely (e.g. with change prediction), thus results on the original approaches may differ. This is a threat to internal validity of our study.

2. While we used a diverse population of projects from four independent ecosystems, it is not clear whether and how our results can be generalized further. This is a threat

to external validity of our study.

3. Our analysis included an extensive technical work, although we tested it carefully and under several scenarios, we cannot guarantee that code is bug-free. However, we make the code available in the online appendix [76]. This is a threat to credibility of our study.

2.7 Discussion and implications

2.7.1 Discussion

Our results highlight several important aspects regarding the choice of the mining strategy.

The technical details of mining can significantly impact the quantitative properties of the retrieved data. In case of our study, such properties are the sizes of file histories and numbers of contributors: In over 2 million files from our composite sample of four ecosystems, the resulting histories differ for 19% of files.

The size of the difference varies across projects and ecosystems. In OpenStack projects, the first-parent and full histories are different for 55% of files, while in Eclipse projects we observe the difference for only 14% of files.

The size of the difference is associated with other properties of a repository. In our case, the ecosystem with the highest difference (i.e., OpenStack) also demonstrates the highest merging activity, most of which can be attributed to code review: after a successful code review in OpenStack, changes are merged back into the trunk by a bot. Thus, high proportion of merges and their impact on file histories can be considered a consequence of their development workflow – namely, code review. Notably, while a similar workflow with code reviews performed with Gerrit is also present in Eclipse, this ecosystem displays the lowest degree of difference: changes in Eclipse are typically rebased, but not merged, after code review. The impact of this particular factor on the results of repository mining deserves a deeper analysis in future work.

The difference in results of mining can influence the performance of techniques based on file histories: reviewer recommendation and change recommendation. For all of the tested approaches, full histories, when used as input data, perform at least not worse than first-parent histories, in most cases yielding a slight increase in performance.

Some applications are more sensitive to quality of input data than others. In our case, for reviewer recommendation in OpenStack full histories provide better accuracy, especially when longer recommendation lists are evaluated (top-5 and top-10 accuracy). OpenStack also happens to be the ecosystem where the difference between the first-parent and full histories is the highest. At the same time, Eclipse projects show the smallest difference between the first-parent and full histories among all four target ecosystems, and this small difference appears insufficient to influence accuracy of reviewer recommendation.

2.7.2 Implications

Considering the points above, we see several important implications of our results.

1. **Software engineering researchers should be aware of the possible impact of the mining technique on the results.** Our study demonstrates that omitting

changes outside the main branch during mining of file histories significantly impacts the results of mining, which often leads to a slight decrease in performance of methods that use file histories as input data.

2. **The choice of the mining technique should account for the context of the mining task.** While using full file histories ensures a better performance, in most cases the difference is only marginal. In many contexts, a small increase in performance may not justify dedicating the extra effort to more precise mining. While we suggest using precise mining methods where possible—and provide a tool for doing that—in many contexts it is not essential.

3. **Researchers should report the technical details of mining.** We suggest that techniques of repository mining should be described in more detail by authors of MSR studies, as not providing details complicates reproducibility of studies, and oversimplifying the mining potentially undermines performance of methods and validity of studies.

2.8 Conclusion

With the study presented in this paper, we make the following main contributions:

- The first demonstration of the importance of careful handling of merge commits and changes from outside the main branch for calculation of file histories;
- Analysis of impact of a strategy of mining file histories on performance of two techniques relying on them as input data;
- A tool for efficient mining of precise file histories in Git [76].

Our results cover the underrepresented topic of technical details of mining the repositories for file histories, and open opportunities for deeper analysis of associated factors, such as topology of change histories. We hope that this study will inspire other researchers in MSR to apply a more detailed approach to mining, where it is feasible, and to report the technical details of mining to ensure clarity and reproducibility of the studies.

3

Mining Tool

This chapter is based on the tool paper "PathMiner: A Library for Mining of Path-Based Representations of Code" by Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli, presented at the Mining Software Repositories conference in 2019 in Montreal, Canada.

3.1 Introduction

Recent achievements in methods of statistical learning have been lately making their way to the field of software engineering. Approaches based on machine learning have improved the state of the art in various software engineering contexts, such as program synthesis [107, 108], language modeling [109, 110], code summarization [111, 112], optimization [113, 114], code completion [115], and automatic code review [116].

The first step in applying machine learning algorithms to source code is to represent the subject code in a way that both captures aspects that are relevant to the task at hand and is digestible by the algorithms. Examples of such representations include a vector of tokens [109], a set of explicitly defined AST metrics [117], and a traversal sequence of the syntax tree [118].

The recently introduced *path-based representation* [119] is a particularly interesting modeling approach, because—even though it was initially introduced for prediction of program properties—it had since proven applicable to several tasks, such as learning embeddings of snippets of code [5], mining of error-handling specifications [120], authorship attribution of code (Chapter 4), and building representations of coding style (Chapter 6).

The extraction of code representations requires substantial technical work, but, to date, we have no well-known and reusable tooling for this task. Conducting this technical work has several implications: (1) It diverts researchers' effort and focus away from the essential part of the work – designing the models and evaluating their applicability in practice, (2) it implies effort duplication within the research community, and (3) it poses a barrier for newcomers in the research area of machine learning on code. As a result, both the development and the adoption of machine learning on code are impaired.

In this paper, we present PathMiner – a library for mining of path-based representations of code. By providing a convenient API for retrieval of the path-based representations and efficient storage of the extracted data, PathMiner strives to provide value for both SE and ML communities by allowing the researchers to skip the time-consuming step of writing custom mining pipelines for the models that utilize the path-based representations of code.

PathMiner supports mining of code in Java as well as Python and is designed to be easily extensible to support other programming languages. We achieve this extensibility by providing a convenient extension point for parsers generated by ANTLR [121]. In addition, in the distribution of PathMiner we provide a Python library to read and process the output of PathMiner, and an example of usage of its output as a dataset for a machine learning task.

3.2 Path-based Representations

The idea of the path-based representation is to model a snippet of code as a collection of paths between the nodes in its syntax tree. In this section, we describe the related concepts.

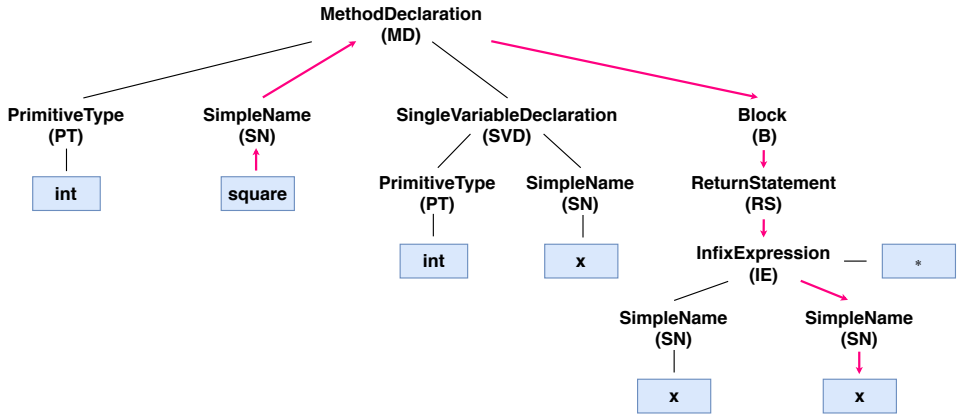
3.2.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree-based structure that represents the syntactic structure of a program. ASTs do not represent the complete contents of a program's source code – some information (such as formatting of the code, parentheses, and the exact form of syntactic constructs) is omitted. Each node in the AST represents a syntactic unit of the

program – such as a variable, an operation, or a logical operator. The children of the node represent the lower-level units associated with the current one.

```
int square(int x) {
  return x * x;
}
```

(a) An example code snippet



(b) The snippet's syntax tree. An example of a path is highlighted in pink.

Figure 3.1: An example code snippet and its syntax tree

Figure 3.1 features a simple code snippet and its AST. While omitting some information, an AST represents the essential structural information about the code, which, along with its own strict tree-based structure, makes it a good intermediate form of code representation in a wide variety of tasks, such as prediction of variable names [119], code summarization [122], and authorship attribution [117].

3.2.2 Abstract Syntax Tree Paths

An *AST path* is a sequence of connected vertices in the AST, logically representing a path from one vertex to another. While, in theory, a path can connect arbitrary nodes of the AST, existing approaches operate with paths between two leaves, which can be linked to concrete tokens in the code.

In practical contexts, a path is denoted by a sequence of types of its nodes and marks of traversal direction. Moreover, the concrete tokens denoting the start and the end vertex of the path are added to the representation, forming a *path-context*.

In Figure 3.1, we highlight an example of an AST path in pink. This path can be denoted as follows (using the abbreviated labels for node types and arrows for direction):

$$SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN$$

Along with the tokens of the end nodes, it forms the following path-context:

$$(square, SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN, x)$$

Formal definitions of path and path-context are available in existing literature [5, 119].

Semantically, a single AST path denotes a logical connection between two concrete elements of code. Representing the whole tree by a set of the contained paths allows to efficiently capture the semantics of code. This is demonstrated by high performance of path-based representations in tasks such as prediction of names for variables as well as methods, and prediction of variable types [119].

Mining of path-based representations involves parsing the code and extracting paths from its syntax tree. With the variety of programming languages, each requiring a specific toolkit for analysis, mining pipelines are generally not reusable. With PathMiner, by providing a reusable and extensible mining library, we strive to help the researchers to dedicate less effort to mining, thus allowing our community to focus on the essence of research problems.

3.3 PathMiner: An Overview

PathMiner is designed to do one thing: extracting path-based representations from code. The core of PathMiner is the `PathMiner` class, whose only method extracts all the paths meeting the height and width limitations from a provided AST.

Practically, it often makes sense to require the extracted paths not to exceed certain *width* (*i.e.,* the distance between the end nodes in the ordered list of AST leaves) and *height* (*i.e.,* the distance between the topmost node in the path and the lower of its end nodes). In fact, this step helps to reduce the amount of generated data and to only capture connections between the nearby code elements [5]. For this reason, the `PathMiner` class is instantiated with a `PathRetrievalSettings` object denoting the limits for path extraction.

ASTs for code in various languages are generated by various `Parser` implementations. The path extraction results are handled by `PathStorage` classes, which also produce the output. Listing 3.1 demonstrates an example of usage of the core components of PathMiner.

The value of PathMiner stands in reduction of development effort for implementation of the extraction of paths – instead of implementing the complete mining pipeline, the code written by the users of PathMiner should only provide integration of PathMiner into their own mining pipelines. Another strength of PathMiner is the ease in supporting arbitrary languages, which is achieved through integration with ANTLR (Section 3.3.3).

The distribution of PathMiner contains several usage examples (Section 3.3.5). In the following, we describe PathMiner's inner workings in more detail.

```
1  val file = File("Example.java")
2  // instantiate the PathMiner
3  val miner = PathMiner(
4  PathRetrievalSettings(maxHeight = 5, maxWidth = 5))
5  // generate the AST from the file
6  val astRoot = Java8Parser().parse(file.inputStream())
7  // retrieve the paths from the AST
8  val paths = miner.retrievePaths(astRoot)
9  // convert the paths into path-contexts and store
10 storage.store(paths.map { toPathContext(it) },
11 entityId = file.path)
12 // produce the output in the specified folder
```

```
storage.save("out_examples/single_java_file")
```

Example 3.1: An example of usage of PathMiner in Kotlin

3.3.1 An Overview Of The Internals

Converting a piece of code into its path-based representation includes several steps, which define the workflow and architecture of PathMiner. Figure 3.2 presents an overview of path extraction workflow and the key components of PathMiner.

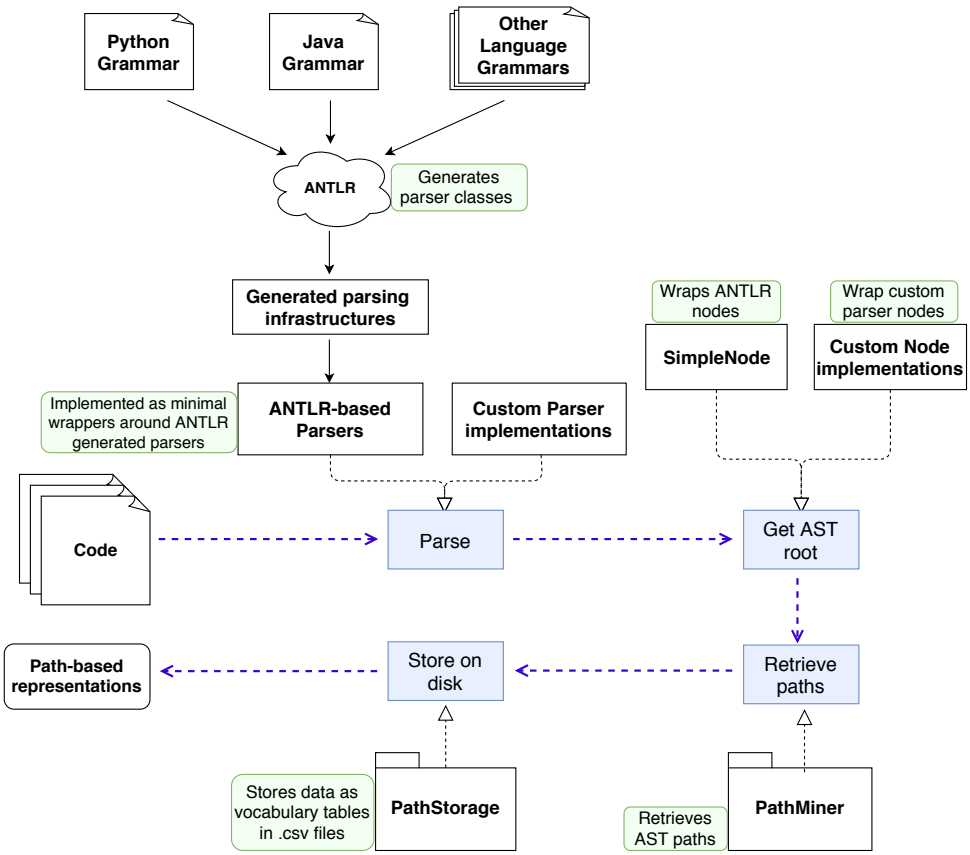


Figure 3.2: An overview of PathMiner's workflow and components.

Parsing. The first step towards a path-based representation is to build an AST from code. All AST operations in PathMiner operate with `Node` – a simple interface representing a node of an AST. The implementations of `Node` are required to support a limited set of operations that are absolutely essential for path extraction, such as retrieval of node's type label and access to children and parent nodes, as well as storage of arbitrary metadata in the node object.

The code in the text form is converted to tree-shaped hierarchies of `Node` implementations by implementations of `Parser` – an interface defining a single method that takes an `InputStream` as an argument and returns an instance of a class implementing `Node`. The concrete implementations of `Node` and the corresponding `Parser` are defined by the programming language and the concrete underlying parser implementation that actually processes the source code. A parser can be implemented either from scratch, which is a hefty task, or, in a simpler way, as an adapter of an external parsing library.PathMiner supports two practical ways to implement a `Parser`, thus supporting a new programming language: (1) conversion from an existing standalone library with parsing capabilities (see `GumTreeJavaParser` class, which wraps the AST representation created by GumTree [123] into `Node`, for an example) and (2) generation of a parser from a predefined grammar with ANTLR (see Section 3.3.3).

Extracting paths. Extracting the paths from the tree is the task of the `PathWorker` class. Given a `Node` object representing the root node of an AST and an object that defines limitations on width and height of the paths, `PathWorker` traverses the tree bottom-up starting from the leaves, memorizing sequences of nodes encountered during the traversal up to the given node. For each non-leaf node, `PathWorker` matches the sequences from each of the node's children and collides the pairs of these sequences to complete paths. Matching of the sequences is performed in a way that ensures that all generated paths are trivial, *i.e.,* each path features a node at most once, and that the paths do not exceed the limits on height and width.

Output. Upon extraction, paths are passed to an implementation of `PathStorage` interface. These entities take care of storing the paths and producing the output. An AST of even a simple program may potentially yield thousands of paths, if the limits are not strict enough. Due to a potentially high amount of produced data, the default implementation of `PathStorage` (*i.e., VocabularyPathStorage*) uses a deduplication technique to ensure an efficient usage of the memory. The output format (Section 3.3.4) is also designed to avoid storing duplicate information in the output.

PathMiner features multiple extension points to ensure its applicability to a wide range of path extraction tasks. Section 3.3.3 provides details on possible extension scenarios.

3.3.2 Technologies In Use

PathMiner is implemented in Kotlin [124], but its output is designated for use in machine learning pipelines, which are commonly implemented in Python. While implementing PathMiner in Python would help ensure easier interoperability, implementation of PathMiner in a statically typed language was our prerequisite to ensure better code quality, null-safety, maintainability, and ease of debugging. To compensate for the extra effort of using the output of PathMiner in Python pipelines, the distribution of PathMiner includes a Python library to handle its output format, as well as an example of its usage for a machine learning task.

PathMiner depends on GumTree for one of the parser implementations, on the ANTLR runtime for generated parsers, and (only in compile time) on JUnit for unit testing.

3.3.3 Extensibility

PathMiner includes several extension points, which help ensure its applicability in a wide range of contexts.

Custom language support

One can easily extend PathMiner to process code in languages that are not supported out of the box. We achieve this by integrating with ANTLR [121].

```
class PythonParser : Parser<SimpleNode> {
  override fun parse(content: InputStream): SimpleNode? {
    // Instantiate the lexer and parser generated by ANTLR
    val lexer = Py3Lexer(ANTLRInputStream(content))
    val tokens = CommonTokenStream(lexer)
    val parser = Py3Parser(tokens)
    // Retrieve the root node from the tree by ANTLR
    val context = parser.file_input()
    // Convert the tree to PathMiner format
    return convertAntlrTree(context, Py3Parser.ruleNames)
  }
}
```

Example 3.2: A complete implementation of Python support for PathMiner

Given a grammar in a predefined format, ANTLR generates a lexer and a parser for the language described by the grammar. Resulting classes transform the source code input into a syntax tree. With many existing grammars for ANTLR [125], PathMiner supports a wide variety of languages. In fact, any language supported by ANTLR 4 can be supported by PathMiner through the implementation of a simple wrapper around the lexer and parser classes generated by ANTLR. Implementing the wrapper is necessary due to variation in methods to retrieve the root node of the parse tree across various generated ANTLR parsers. However, the coding effort is very minimal: Listing 3.2 features a complete example of an implementation of support for a new language in PathMiner, containing only 4 lines of meaningful code.

AST splitting

Parsers in PathMiner generate a single AST for the whole input – the basic unit of input is a file. Depending on the task, one can operate with smaller units of code, such as method definitions. To support possible scenarios of use for extracting paths from smaller units of code, PathMiner features the `TreeSplitter` interface. Its implementations extract finer units from the tree, returning a collection of AST nodes, each representing such unit.

```
interface TreeSplitter<T : Node> {
  fun split(root: T): Collection<T>
}
```

Due to variability of AST node types across languages and corresponding parsers, the implementations of `TreeSplitter` are not reusable across languages and have to be implemented individually. Our distribution of PathMiner includes an example implementation (`GumTreeMethodSplitter`) that extracts method definition nodes from the ASTs generated by `GumTreeJavaParser`.

Output

Storage of extracted path-contexts and generation of output are handled by the ancestors of the `PathStorage` class. PathMiner includes a default implementation

(`VocabularyPathStorage`) that handles the storage and output in a memory-efficient manner. However, mining tasks that require alternative representations of the data can be supported by implementing a custom `PathStorage`.

3.3.4 Output format

By default, PathMiner uses a custom output format, introduced by the authors of the path-based representation [5]. The format is designed to present the data in numerical form, while storing it in a memory-efficient manner. Figure 3.3 presents an overview of the format, which is based on vocabulary tables. The format exploits the fact that, due to the structured nature of code and limited number of unique node types and tokens, many identifiers and paths are likely to occur more than once over mining tasks of significant size. The storage algorithm associates every token, node type, and path with a unique identifier, thus avoiding storing duplicate data. Each vocabulary table is stored on disk in a separate `.csv` file.

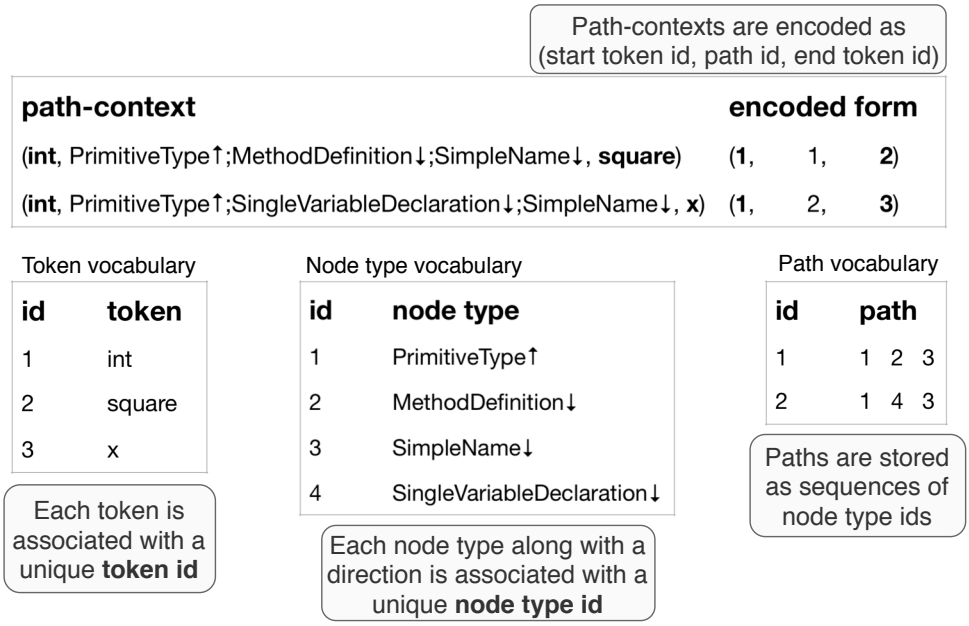


Figure 3.3: An example of vocabulary encoding for two path-contexts

3.3.5 Distribution And Usage Examples

The distribution of PathMiner [126] includes several examples of its usage. The examples of path retrieval tasks, implemented in Kotlin and Java, are located in the `examples` package.

To ease further use of the vocabulary-based output of PathMiner in real-world machine learning tasks, we include a small Python library that reads the output and wraps it into

wrapper classes for fast access to tokens, paths, node types and path-contexts. Data in this form can later be used with any machine learning framework.

To demonstrate the usage of the Python library,[1] we also provide an example of use of the path-based representations for an actual machine learning task. We implement a linear classifier that is trained to distinguish between two possible projects of origin for a file based on its path-based representation. When evaluated on two projects of several thousand Java files each, the classifier achieves over 99% accuracy, precision, and recall after a few minutes of training.

While we only provide the classifier as an example of usage of PathMiner's output for a new task, high accuracy achieved with a very simple model suggests that the range of applications for path-based representations stretches beyond existing research. This point highlights the potential value of PathMiner for the research community.

3.4 Quality and performance

The critical components of PathMiner—path extraction pipeline and parsers—are covered by unit tests, which are run on every change by a continuous integration system. To ensure quality and readability of PathMiner's sources, we employed code reviews during the development process: All of the code in PathMiner was reviewed by, at least, a second developer.

On a mid-range developer machine, PathMiner is capable of processing 300-500 Java files per second, with the most of CPU time being spent on I/O and parsing. This suggests that PathMiner is efficient and is not going to be a bottleneck in existing pipelines. Our own experience with using PathMiner for ongoing research projects confirms its high performance.

[1] The Python library and its usage example are located in the `/py_example` folder in the distribution. The folder contains a `README.md` file with instructions to run the example.

4

Authorship Attribution

This chapter is based on the paper "Authorship Attribution of Source Code: A Language-Agnostic Approach and Applicability in Software Engineering" by Egor Bogomolov, Vladimir Kovalenko, Yurii Rebryk, Alberto Bacchelli, and Timofey Bryksin. The paper is currently (October 2020) under a major revision during submission to the IEEE Transactions on Software Engineering journal.

4.1 Introduction

The task of source code authorship attribution can be formulated as follows: given a piece of code and a predefined set of authors, to attribute this piece to one of these authors, or judge that it was written by someone else. This problem has been of interest for researchers for at least three decades [127].

Prior research has shown that software engineering tasks, such as software maintenance [128–130] and software quality analysis [131–134], benefit from authorship information. Since authorship information in software repositories may be missing or inaccurate (*e.g.*, due to pair programming, co-authored commits, and changes made after code review suggestions), authorship attribution techniques could help in these tasks.

Source code authorship attribution is also useful for plagiarism detection, either to directly determine the author of plagiarized code [135–137] or to ensure that several fragments of code were written by a single author [138]. Plagiarism detection, in turn, is important in software engineering: Software companies need to pay extra attention to copyright and licensing issues, as they can become liable to lawsuits [139]. For example, developers often use Stack Overflow[1] to copy and paste code snippets to their projects. However, if developers do not use caution, code borrowed from Stack Overflow can induce licence conflicts on top of complicating maintenance [140, 141].

Recently, several pieces of work improved the state of the art in authorship attribution on datasets for three popular programming languages: C++, Python, and Java. For C++, Caliskan *et al.* reported the accuracy value of 92.8% when distinguishing among 1,600 potential code authors [142]. For Python, Alsulami *et al.* attributed code of 70 programmers with 88.9% accuracy [143]. Yang *et al.* developed a neural network model that achieved 91.1% accuracy for a dataset of Java code by 40 authors [144].

Some of the existing approaches use language-specific features. While language-specific features can improve accuracy of authorship attribution for a particular language, designing a set of such features is a complex manual task. To transfer an approach that targets a specific language to another one (*e.g.*, from C++ to Python), one either needs to come up with a new set of features, or otherwise suffer from a major accuracy drop [142].

In this study, we improve over the existing results in authorship attribution accuracy by suggesting two *language-agnostic* models. Both models work with path-based representations of code [145], which can be built based on an abstract syntax tree (AST) for any programming language. The first model, called PbRF (Path-based Random Forest), is a random forest trained on term frequencies of tokens and paths in ASTs. The second model, named PbNN (Path-based Neural Network), is an adapted version of the *code2vec* neural network [145].

We evaluate both models on the datasets of code in three programming languages that were used in previous work. PbRF matches or improves the state of the art for Java, C++, and Python datasets, even with few available samples per author. PbNN outperforms PbRF when the number of available samples per author is large. Both models improve state-of-the-art results for Java on a dataset of Yang *et al.* [144], with 97% and 98% accuracy respectively. Later, we evaluate how both models perform on our new collected Java datasets.

[1] `https://stackoverflow.com/`

Another aspect that we target in this study is data used for evaluation. Existing work on authorship attribution operates with sources of data different from regular software projects: examples from books [127, 146], students' assignments [147–149], solutions of programming competitions [142, 143, 150, 151], and open-source projects with a single author [135, 137, 144, 152]. This data is different from code that can be found in real-world software projects. In this study, we deeply investigate this difference. Based on the results of this investigation, we propose a new data collection technique that can reduce these differences and generate more realistic datasets. To formalize the differences, we suggest the concept of *work context*: *i.e.,* aspects that can affect developer's coding practices and are specific to the concrete project, such as project's domain, team, internal coding conventions, and more. Also, we discuss another source of data differences: the evolution of programmers' individual coding practices over time and changing context of their contributions.

We propose a method to quantitatively measure the impact of work context and evolution of individual coding practices on the accuracy of authorship attribution models. Our evaluation shows that accuracy of authorship attribution models plunges when models are tested with more realistic data than what is offered by existing datasets. In particular, the model that can distinguish between 40 authors with 98% accuracy in one setup, reaches only 22% for 21 developers in another. This result suggests that—before their practical adoption for software engineering—existing results in the field of source code authorship attribution should be revisited to evaluate robustness of the models and their applicability to more realistic datasets, which aligns with the existing understanding of the issues of reusing models on other datasets in other tasks [153, 154].

With this work we make the following contributions:

- Two language-agnostic models that work with path-based representation of code. These models match or outperform language-specific state-of-the-art models on existing datasets.
- An in-depth discussion on the limitations of existing datasets, supported by quantitative evaluation of effects of these limitations, particularly when applied to the software engineering domain.
- A novel, scalable approach to data collection for evaluation of source code authorship attribution models.
- Formalization of the concept of *developer's work context* and a novel methodology to evaluate its influence on the accuracy of authorship attribution models.
- Empirical evidence on how the evolution of developers' coding practices impacts accuracy of current authorship attribution models.

We structure this paper around four research questions:

- **RQ1.** To what extent can models based on path-based representation of code improve existing results in authorship attribution task?
- **RQ2.** Do existing datasets reflect characteristics of the real-world projects?

- **RQ3.** If we collect a dataset from a real-world open-source project with many authors, how will accuracy of authorship attribution models change compared to the existing datasets?

- **RQ4.** To what extent does evolution of developers' coding practices affect quality of authorship attribution models?

4.2 Background

To the best of our knowledge, the first work on source code authorship attribution dates back to Oman *et al.* [127] in 1989. Although the results and approaches for the authorship attribution have changed and improved since then, the formulation of the problem did not change, and the underlying idea remains to use machine learning based on the features extracted from source code.

In the task of source code authorship attribution, a model is given a piece of code, and it should attribute the piece of code to one of the known developers, or state that it was written by someone else. The input code can differ by its form (the code can be unchanged, obfuscated, or even decompiled) and source of origin. In previous work, four sources of data have been used:

- **Code examples from books** ([127, 146]). They were used before the era of easily available open-source projects, for the lack of other sources.

- **Students' assignments** ([147–149, 155]). Often, researchers are not allowed to publish these datasets (mostly from university courses) due to privacy or intellectual property issues. Lack of published data causes problems for comparison of different methods.

- **Solutions to programming competitions** ([142, 143, 150, 151]). This mostly refers to data from Google Code Jam[2] (GCJ), an annual competition held by Google since 2008.

- **Single-author open-source projects** ([135, 137, 144, 152]). With the increasing popularity of hosting platforms for open source projects (*e.g.*, GitHub), they have become a major source of data. In this case, the dataset consists of multiple repositories, each developed by a single programmer. For evaluation, the researchers use unseen code snippets from the same repositories. Researchers avoid repositories with multiple authors because in this case authorship of even small fragments of code might be shared.

According to the recent survey by Kalgutkar *et al.* [156], the following are the best results per programming language:

- **C++**: Caliskan *et al.* [142] reported the best results using a random forest trained on syntactical features. They achieved 93% and 98% accuracy for datasets with 1,600 and 250 developers, respectively.

[2] `https://codingcompetitions.withgoogle.com/codejam`

- **Python**: Alsulami *et al.* [143] suggested to use tree-based LSTM to derive implicit features from ASTs, achieving 88.9% accuracy in distinguishing among 70 authors.
- **Java**: Yang *et al.* [144] reported 91% accuracy for a dataset of 40 authors using neural networks. Instead of a commonly used stochastic gradient descent optimizer, the authors trained the network with particle swarm optimization [157] improving the accuracy by 15 percent points.

Syntactic features, derived from AST of code, are known to improve the results for authorship attribution [142, 143] as well as for other software engineering tasks, such as code summarization [23], method name prediction [145], and clone detection [158].

Compared to real-world data, where a programmer often works on multiple projects and using multiple languages, existing datasets are limited to a single language and one project per author. To overcome this limitation, models applicable in real-world environment should work with different programming languages in a consistent manner. Following this idea, we decided to build a language-independent model, that is based on syntactic features and works on par with prior studies.

4.3 Language-Agnostic Models

Our first goal is to develop an authorship attribution solution that is language-agnostic and achieves accuracy comparable to state-of-the-art approaches.

To apply machine learning methods to code, one should transform code into a numerical form called *representation*. While some work uses explicitly designed language-specific features [142, 144], we represent code using *path-based representation* [24] as to work with code in various programming languages in a uniform way. Section 3.2 explains the path-based representation and related concepts in detail.

A common way to use path-based representation is the *code2vec* neural network [145], suggested by Alon *et al.* for the task of method name prediction. However, code2vec requires a significant number of samples for each author to infer meaningful information, due to the large number of trainable parameters. Thus, alongside with the neural network, we also employ a random forest model, trained on similar features. The random forest model shows better accuracy for small datasets, but generalizes worse for the larger ones.

4.3.1 PbRF (random forest model)

The random forest model is designed to work even when the number of samples for each author is rather small (starting from a few samples per author), where the neural network cannot capture enough information to generalize. Random forest has already proved to be effective in this setup in previous work [142, 151]. Random forest does not allow training an embedding of path-contexts, thus, instead of combining paths and tokens into path-contexts, we use raw term frequencies of tokens and paths as features. Tokens and AST-based features has already proved to be effective for authorship attribution in the work by Caliskan *et al.* [142]. Compared to their work, we use more complex AST features – AST paths.

If a set of documents contains T tokens and P paths, the random forest model takes a sparse vector of size $F = P + T$ as an input. The size of such a vector might be significant (up

to millions), with some features being unimportant for identifying the author. We employ feature filtering to reduce the effect of this dimensionality problem. As in previous work on authorship attribution [142, 159, 160], we use filtering based on *mutual information* (MI) [161]. The mutual information of a source code feature f and an author A can be expressed as:

$$I(A,f) = H(A) - H(A|f),$$

where $H(X)$ is Shannon entropy [162]. For the task of authorship identification, we interpret it as follows: the higher the MI is (*i.e.,* the lower $H(A|f)$ is), the better one can recognize the author based on the value of the given feature.

Feature selection based on mutual information criteria ranks all the features by their MI with the author label and takes $N\%$ with the highest MI value. N is a hyperparameter determined empirically during the evaluation process by trying various values. This approach does not account for dependencies among features; for example, if there are two identical high-ranked features, we take both and miss some other feature. To avoid this problem, one could add features one by one and compute mutual information after each step, but on the scale of millions of features, this procedure is too costly.

4.3.2 PbNN (neural network model)

To achieve better accuracy for larger datasets, we adopted the neural network called code2vec [24]. It takes a bag of path-contexts from a code snippet as an input, transforms them into a single numerical vector, and predicts a probability for each known developer to be an author of the snippet. Compared to classical machine learning methods, neural networks can derive more complex concepts and relationships from structured data, when given enough training samples.

Figure 4.1 shows the architecture of the network. The network takes a bag of path-contexts from a code snippet as input. The number of path-contexts, even with restrictions on path length and width in place, might be tremendous. To speed up the computation, at each training iteration we only take up to 500 random path-contexts for each sample. Then, the network embeds paths and tokens into $\mathbb{R}^d$, where d is a hyperparameter. Finally, the model aggregates the information from up to 500 path-contexts into a single d-dimensional vector and predicts an author of the code snippet based on it. The detailed description of the model is available in the original *code2vec* paper by Alon *et al.* [24]

The number of the PbNN's parameters is $O((T+P)d)$. Since the value of $(T+P)$ is usually large, from tens of thousands to millions, the number of required samples for the model to train is also significant.

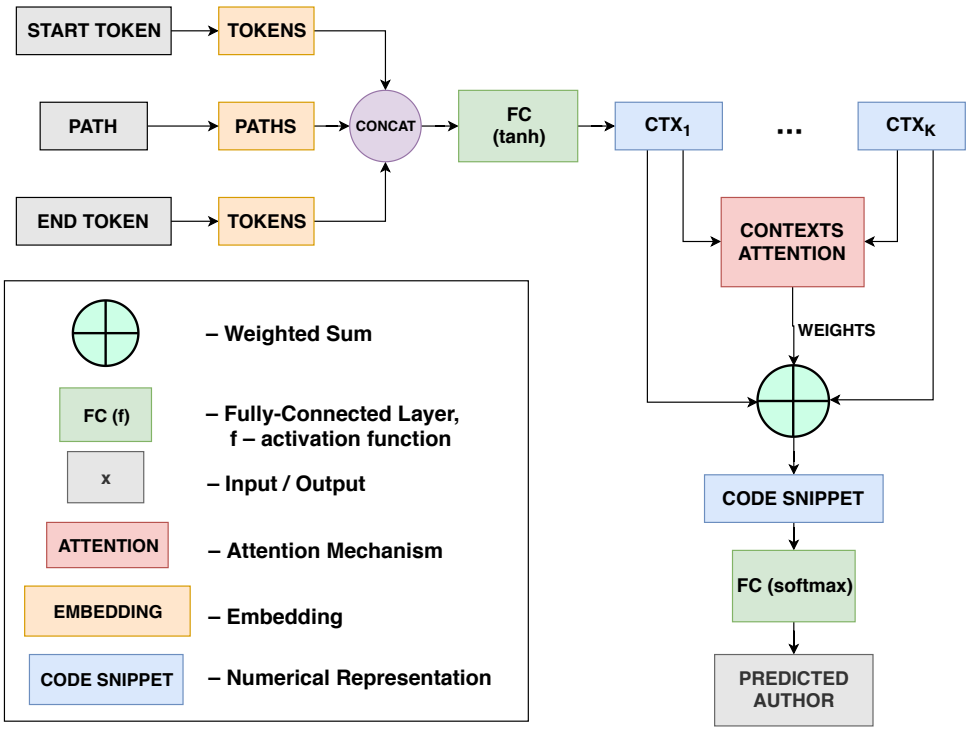


Figure 4.1: Architecture of the PbNN

4.4 Evaluation on Existing Datasets

We evaluated the two models presented in the previous section on publicly available datasets for Java, C++, and Python, used in recent work [142–144], and compared the accuracy of PbRF and PbNN to the results reported in these papers. Table 4.1 shows statistical information about the datasets. Table 4.2 presents results of accuracy evaluation.

To compare the results of different models when these results are obtained through multiple runs (*i.e.,* folds in cross-validation), we apply the Wilcoxon signed-rank test [163] to various accuracy values per run. When only the mean accuracy is available (which is the case for previous work), or the number of runs is too small, we directly compare the mean values.

Table 4.1: Datasets used in previous works. The number of paths is provided for $width <= 3$ and $length <= 8$

	C++ [142]	Java [144]	Python [143]
Number of authors	1,600	40	70
Number of files	14,400	3,021	700
Samples per author	9	11 to 712	10
Source	GCJ	GitHub	GCJ
Unique tokens	30,200	36,700	4,300
Unique paths	169,900	7,900	46,300

Table 4.2: Mean accuracy by approach and dataset

	C++	Python	Java
Caliskan *et al.* [142]	**0.928**	0.729	N/A
Alsulami *et al.* [143]	N/A	0.889	N/A
SGD [144]	N/A	N/A	0.760
PSO [144]	N/A	N/A	0.911
This work, PbNN	0.415	0.617	**0.981**
This work, PbRF	**0.927**	**0.937**	**0.97**

4.4.1 Hyperparameters

Both our models have parameters that should be fixed before the training phase, *i.e.,* hyperparameters. These hyperparameters are (1) number of trees, (2) percentage of features left after feature selection (feature ratio) for the random forest, (Section 4.3.1) and (3) size of the embedding vector for the neural network (Section 4.3.2).

We tuned these hyperparameters using grid search [164]. We found that the optimal values are the same across the datasets. For the random forest model, increasing the number of trees improves accuracy until this number reaches 300, after that accuracy reaches a plateau. The optimal feature ratio for all datasets lays between 5% and 10%.

For the neural network, increasing the embedding dimensionality results in a significant growth in accuracy until the size of the vector reaches the value of 64.

4.4.2 Evaluation on C++

The C++ dataset was introduced by Caliskan *et al.* [142]. It contains solutions by each of 1600 participants to each of 9 problems from Google Code Jam 2012, making it one of the largest experiments with respect to the number of authors. Each author has 9 samples in the dataset, corresponding to the solutions of 9 problems. Following the original paper, we run 9-fold cross-validation. At each fold, the held-out set contains solutions to a single problem, while training set contains the rest 8 problems. For comparison, we use the average models' accuracy over all folds.

Caliskan *et al.* [142] reported 92.8% mean accuracy after 9-fold cross-validation. PbNN and PbRF achieve 60% and 92.7% average accuracy, respectively. The neural model performs poorly because of overfitting: The number of available data points is too small to train a much larger set of the network's internal parameters. The PbRF's accuracy is marginally lower compared to the Caliskan's work (92.7% vs 92.8%), but the difference is smaller than the standard deviation computed based on the cross-validation (which is 0.8%), thus the difference is indistinguishable from random noise. We can conclude that *PbRF is on par with the previous best result.*

4.4.3 Evaluation on Python

The Python dataset also contains Google Code Jam solutions. It was collected and introduced by Alsulami *et al.* [143] and consists of solutions to 10 problems implemented by 70 authors. As in the C++ dataset, this one contains the same number of samples for each author. During cross-validation, the model is trained on 8 problems and validated on 2 other problems that are initially held out. The best reported average accuracy is 88.9%.

For comparison, we use average accuracy reported by Alsulami *et al.* [143] for two models. The first one is a novel model introduced by the authors. The second one is an adopted version of the model by Caliskan *et al.* [142], with C++-specific features removed.

On this dataset, our models achieve 61.7% (PbNN) and 93.7% (PbRF). Similarly to the C++ dataset, PbRF shows better accuracy compared to PbNN because the number of available samples is too small for efficient training of the neural network. Also in this case, *PbRF achieves an accuracy that is at least as good as the previous best result.*

4.4.4 Evaluation on Java

The Java dataset, introduced by Yang *et al.* [144], consists of 40 open source projects, each authored exclusively by a single developer. Each project contains from 11 to 712 files with a median value of 54, totaling 3,021 files overall. The number of samples per author varies.

For evaluation, we split the dataset into 10 folds and perform cross-validation, similarly to Yang *et al.* [144]. Ideally, to compare the accuracy of our model to theirs in the most precise manner, we would use an identical split of the dataset into folds for our evaluation. However, the original split into folds is not available and we created our own with a fixed random seed.

The model by Yang *et al.* is a neural network, trained either by stochastic gradient descent (SGD) or particle swarm optimization (PSO) [157]. When trained by PSO, the

model achieves an average accuracy of 91.1% using 10-fold cross-validation. Both our models reach an accuracy of more than 97%. The previous result lies out of the standard deviation range in both cases, thus, indicating that the difference is statistically significant. Although the median number of samples per author is only 54, the neural network shows high accuracy. Even though the average accuracy of the neural network model is slightly higher than that of the random forest, the statistical test yields a p-value of 0.07. Also in this case, both *PbRF and PbNN improve the previous best result.*

4.5 Limitations of Current Evaluations

During the analysis presented in Section 4.4, we have realized a number of limitations posed by this evaluation technique. Particularly, using a single accuracy metric to compare complex approaches to problems motivated by practical needs (such as authorship attribution in software engineering) is a one-sided solution. In this section, we discuss the limitations of such evaluation.

Academic work on authorship attribution is motivated by the practical needs such as detection of plagiarism [152, 165, 166], detection of ghostwriting [137, 147], and attribution of malware [142, 148, 149, 167]. In a perfect world, introduced authorship attribution approaches should be evaluated on real-world data. However, such real-world data is privacy-sensitive and seldom publicly available. For this reason, to show how models behave and compare against each other, researchers create datasets from available data sources. Even though these datasets try to mimic the data found in practical applications, there exist major differences. To illustrate them, we introduce the concept of developer's *work context, i.e.,* the environment that surrounds programmers when they write code. The work context includes but not limited to the following:

- **Parts of a codebase**: A codebase of a software project usually contains logically interconnected code that is organized into packages and modules. This logical connection often implies a lower-level connection observed in code: calls of methods, creation of objects, similar names of entities.

- **Project domains**: The domain of the task (*e.g.*, an Android application) influences names, used libraries, implemented features, and architectural patterns that are used more commonly.

- **Projects**: The project itself might have internal naming conventions and utility components that are called from different parts of the codebase. Moreover, some companies have their own style guides for programming languages that affect naming conventions, formatting, and preferred use of specific language constructions. An example of this is Google Style Guides [168].

- **Set of tools**: Integrated development environments (IDE) or text editors, version control systems, as well as build and deployment tools may influence the way how developers write code. For example, a recent survey with programmers concluded that developers affect their development practices by simply using GitHub in their projects [169].

This list is not complete and could easily be extended, but the effect of even a single of the aforementioned individual examples of work context might be significant for the task of authorship attribution. When collecting data for evaluation, we should take work context into consideration: practical applications of authorship attribution often imply that the model should be trained on code written in one context and tested in another, or distinguish between developers working in the same context. However, datasets used in prior works do the opposite: There is difference between authors' work contexts (*e.g.*, different single-authored projects) and no difference between work contexts of the same developer in training and evaluation sets.

Another concern is that existing datasets do not consider the impact of collaboration on the code. All of them consist of projects developed by a single programmer. However, projects studied in the software engineering domain are usually developed by teams. Collaborative work is not reflected in prior work, but it may introduce additional complexity for the authorship attribution task: developers integrate their code into codebases written in collaboration with their colleagues, which might make their code harder to distinguish.

Developers' individual coding practices may change over time [170]. It is reasonable to think that changes in coding practices (*e.g.*, programmers style, used libraries, naming conventions, process of development) can influence the accuracy of the models significantly. For existing datasets, all code written by a single author belongs to roughly the same period of time (*e.g.*, one project, one competition, and assignments belonging to one course) and there is no temporal division between evaluation and training sets. Though, for practical problems, one might need to train a model on the historical data and apply it to new samples. This temporal aspect may introduce potential significant difference in individual coding practices between code used for training and testing.

Two prior studies consider evolution of programmers' style as a challenge for authorship attribution [142, 170]. Burrows *et al.* evaluated the difference between six student assignments, showing that their coding style changes over time [170]. Caliskan *et al.* trained a model on solutions of Google Code Jam (GCJ) 2012 and evaluated it on a single problem from GCJ 2014. Their experiment did not reveal any major differences in accuracy compared to evaluating on a problem from the same GCJ 2012 [142]. These results are contradictory, but both studies operated with small datasets and in domains different from real-world projects. Thus, further research on this topic is required.

We conclude that there is a gap (at least theoretical) between the existing datasets and what can be collected and used in the context of real-world applications. In particular, there is a difference in terms of work context, effects of developer collaboration, and changes over time. In the following sections, we suggest a novel approach to data collection that allows to quantitatively evaluate the impact of both temporal and contextual issues on accuracy of authorship attribution models.

4.6 Collecting Realistic Data

To quantitatively evaluate the impact of dissimilarities between existing datasets and real software projects on the accuracy of authorship attribution techniques, we developed a new approach to data collection. It uses Git[3] repositories as data source and, unlike existing

[3] https://git-scm.com/

datasets from open-source data [135, 143, 144, 171], overcomes the limitation of a single author per project.

4.6.1 Method of data collection

We suggest a new approach to collecting evaluation data for authorship attribution models. The approach works with any Git project without restrictions on the number of developers. In particular, using Git as the main data source allows taking data from GitHub — the world's largest repository hosting platform with more than 100 million repositories and 30 million users [172]. Git repositories, and GitHub in particular, are a uniquely rich source of data for modern software engineering research efforts [26, 173–175]. The atomic unit of contribution in Git projects is a commit. Usually, a commit has a single author. This authorship information, associated with every change recorded in a Git repository, makes Git a particularly rich source of data for authorship attribution studies.

The first step of our data extraction method is to traverse the history of a repository to gather individual commits. Then, we need to identify commits authored by the same developer. It is not a trivial task, because a developer can work within one repository under multiple aliases using different emails. Even though there are prior studies on the problem of alias merging [176–181], these methods either make assumptions or are probabilistic to some extent. The main benefit of the existing alias merging methods is their complete autonomy that allows working with arbitrary amounts of data. In this study, we processed IntelliJ IDEA Community [182], and accessed an internal system to avoid any mistakes at this step. For future researchers who may want to process larger amounts of data autonomously, our data preparation pipeline supports using existing entity merging methods that not require external information [178].

In the second step, we split each commit into changes of fixed granularity (*e.g.*, a change to a single class, method, or field). In this study, we use a *method change* as the granularity unit: it is hard to precisely track changes of smaller fragments of code (*e.g.*, lines or statements), while using class-level or higher granularity would leave us with less data points. Also, authorship attribution at the method level can be a valuable task in the domain of clone detection, where method is a commonly studied unit [141].

Within the scope of a single commit, methods can be renamed or moved to other files. We use GumTree [123] to precisely track such changes in Java code as well as simple changes to a method's body. As a result, we get a set of all method changes made during the project development. Afterwards, the extracted data can be grouped into datasets with different properties.

4.6.2 Collected datasets

We implemented an open-source tool supporting the described approach to data collection [183]. We used the tool to extract data from the IntelliJ IDEA Community Edition [182] project, the second largest public Java project on GitHub. At the time of processing (March 10, 2020), the project contained about 270,000 commits authored by 500 developers. These commits comprise about two million individual method changes. Each change is of one of three types: creation of a method, deletion, or a modification of its body or signature. The latter, unlike method creations, cannot be processed directly by authorship identification models: newly added code fragments might be incomplete, and the concrete modifications

might be scattered across the method body. Moreover, the author of the original code might be different from the one who modifies the method, which makes it impossible to define a sole author of the method. In the datasets designed in this work, we only use method creations, because they contain new code fragments implemented by a single person and can be labeled accordingly. However, attributing authorship of method modifications is an interesting task for future research. Chapter 6 presents a related algorithm that operates with method changes.

In the IntelliJ Community repository, out of all the changes, 680,000 are method creations. The developers highly differ by the amount of contributions: 20 most active developers created 58% of all methods, 50 most active — 88%. To reduce imbalance of the datasets, we split the authors in two groups: 21 authors with at least 10,000 created samples (*i.e.,* created methods) and 44 authors who have between 2,000 and 10,000 samples. In total, the groups have about 400,000 and 200,000 samples, respectively.

To quantitatively evaluate the impact of work context and evolution of coding practices on the quality of authorship attribution, we further create several datasets from the collected data. To make evaluation conditions as close to practical tasks as possible, we should have processed several projects and split *projects* between training and testing sets. However, at this point it is unclear how to define similarity between work contexts of different projects, and we would not be able to run several experiments with an increasing degree of context difference to perform a quantitative evaluation. Thus, this left us with one project and multiple datasets.

Datasets with gradual separation of work context

The purpose of these datasets is to measure the influence of variation in developers' work context on the quality of authorship attribution. To achieve this, we need multiple pairs of training and evaluation samples that differ only in their work context. More specifically, the pairs should contain the same code fragments but be split differently between the training and testing part.

To control the degree of difference in work contexts, we use the project's file tree. Figure 4.2 shows an example of such a tree. It consists of folders with edges between a folder and its content. Leaves in the tree correspond to files. For Java code, we are interested only in Java source files identified by the '.java' extension. To reduce the depth of the tree, we compress paths of folders with a single sub-folder into nodes: In Figure 4.2, folders "plugins", "src", and "main" are compressed into a single node "plugins/src/main". This operation preserves the structure of the tree.

Table 4.3: Correct splits at different levels of similarity with equal size for the training and the evaluation sets

Similarity	Training	Evaluation
1	plugins/src/main	platform
2	P-A, P-B, platform-api	P-C, P-D, platform-impl
3	P-A, P-B, API-A, Impl-A	P-C, P-D, API-B, Impl-B

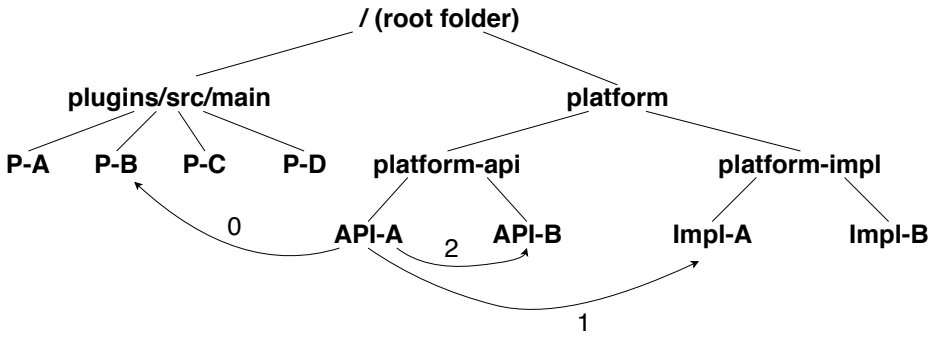


Figure 4.2: An example of a project's file tree with similarities between files

4

A file tree for a Java codebase resembles the structure of packages. Usually, classes in one package are logically connected and refer to each other; thus, they have similar work contexts. At a higher level of abstraction, this also applies to classes in different sub-packages of the same package. In Figure 4.2, the tree class 'API-A' has a work context that is similar to 'API-B', because they are in the same package, but a less similar context to 'Impl-A' from 'platform-impl'. Nevertheless, they are still much closer to each other than to any file from the 'plugins' package since both are used to implement some platform features and may even depend on one another.

From this observation, we derive a way to measure similarity of work contexts of two files: it is *the depth of the lowest common ancestor in the file tree*. In Figure 4.2, similarities between 'API-A' and other classes are shown with arrows (depth of the root is considered to be 0). By doing so, for a training-validation split, we can define the similarity of work contexts of these parts as the highest value of pairwise similarity between files in them. Usage of the maximum value might seem over-cautious, but in real-world applications it is natural to assume that the model will not see any samples from another context during training, before receiving them as input.

The subsequent step is to create a sequence of data splits with increasing similarity values, or depth of split. To preserve the distribution of authors at each level, we pick splits for different authors independently and merge them afterwards. We fix a fraction of training samples t, depth of split d, and an author A. Then, we collect all folders at depth d and files at depth d or less. Afterward, we greedily divide them into training and evaluation parts, trying to get as close to t as possible. When a folder F is put into training/testing part, all the methods created by A in the subtree of F go into this part.

Table 4.3 shows an example of correct splits at different depths. Splits with smaller values of similarity are valid for the subsequent levels. If we split the parts randomly without any restrictions, it can result in the same split at all levels, in contrast to our goal of obtaining splits with different values of similarity. To avoid this, at each depth level we create several splits randomly and take the one with the minimal mutual information with the previous one. The mutual information shows the degree of randomness of the split with respect to the previous one. In Table 4.3, the mutual information between consequent splits is 0, because every time the training and evaluation parts are split into halves.

We created two datasets by applying the described algorithm to the method creations by the developers in two aforementioned groups. The file tree in the IntelliJ IDEA project has a depth of 12. Figure 4.3 shows the distribution of files by depth. Since the increase in similarity value from $d-1$ to d affects only files with depth d and higher, we vary d only in the range $[1;9]$, which includes over 95% of the files.

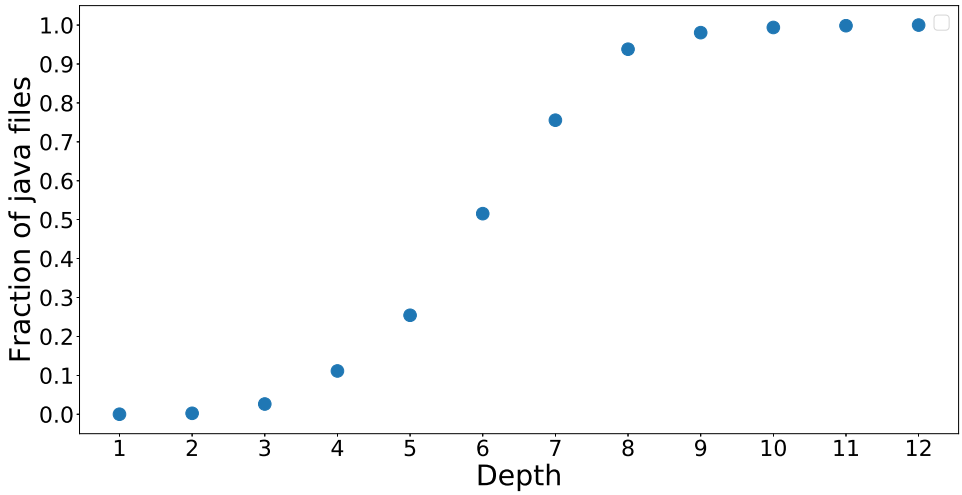


Figure 4.3: Fraction of Java files with depth not exceeding d

Dataset with separation in time

These datasets are designed to investigate whether developers' coding practices change over time. The high-level idea is as follows: we pick a set of method creations from a project (IntelliJ IDEA in our case), sort them by time, split into ten folds, then train a model on one fold and evaluate it on the others.

More specifically, we gather all events of method creation generated by the developers in groups with different amount of samples per developer. Then, we sort all the methods written by each author by creation time and divide them into ten buckets of equal size. To preserve the same distribution of the authors across the buckets, we do the division independently for each programmer. We also evaluated an alternative approach of splitting all the methods simultaneously, but this approach ended up adding too much noise to the data (in fact, some programmers joined the project later and the model trained on earlier folds did not have any information about them).

The resulting datasets consist of 400,000 and 200,000 events of method creation split into ten equal folds. The events are sorted in time and the difference between the indices of their folds can be used as the temporal distance between them. The distribution of the authors and the number of samples is uniform for every fold.

4.6.3 Benefits of the data collection technique

The proposed data collection technique enables collection of datasets that have several major benefits over existing datasets and capture some important effects that are specific to real-world data:

- *Smaller gap between work context of code written by different authors.* While different developers still tend to work in different parts of the codebase, naming conventions, internal utility libraries, and the general domain are the same for everyone, since all code originates from the same codebase.

- *Large number of samples available per author.* Existing datasets mostly work with up to several hundred code fragments per author. In the IntelliJ IDEA dataset collected with our technique, 21 developers have created more than ten thousand methods each. The ability to collect multiple contributions for a single author makes the resulting data suitable for studying more fine-grained aspects of authorship attribution, such as the effect of the changes in coding practices over time on attribution accuracy.

- *Broader domain of application.* Since our data collection technique allows one to collect data from any Git project, it is possible to investigate cross-project or cross-domain authorship attribution.

4.7 Evaluation on Collected Datasets

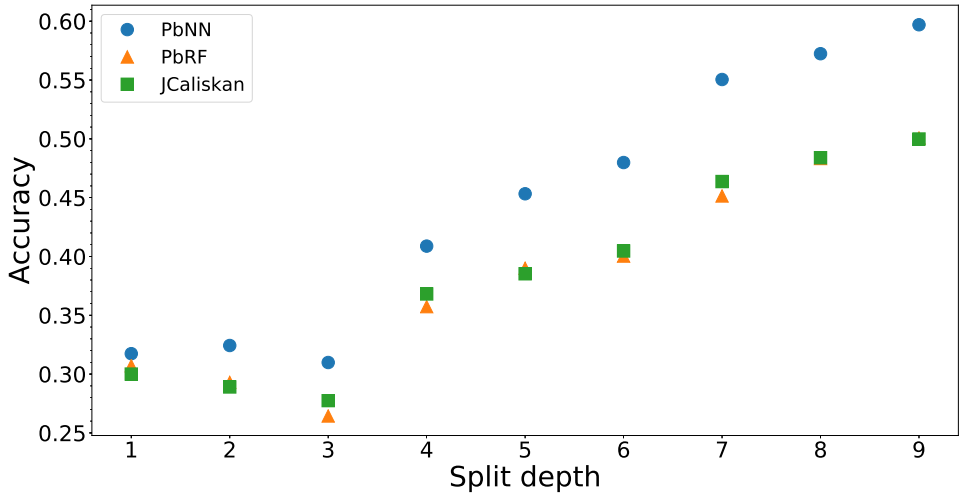
We evaluate both our models (described in Section 4.3) on the datasets that we created from IntelliJ IDEA source code using the technique described in Section 4.6. Also, we adapt the model by Caliskan *et al.* [142] to work with Java code and refer to it as *JCaliskan*. We compare *JCaliskan* to our models on the new datasets.

We also tried to reimplement the model by Yang *et al.* [144], since the work lacks an open-source implementation. However, we failed to reproduce the results mentioned in the paper [184]. In our case, usage of particle swarm optimization to train the model led to no improvements and strong overfitting. Since features used in their paper resemble the work by Caliskan *et al.* [142], we used JCaliskan for comparison.

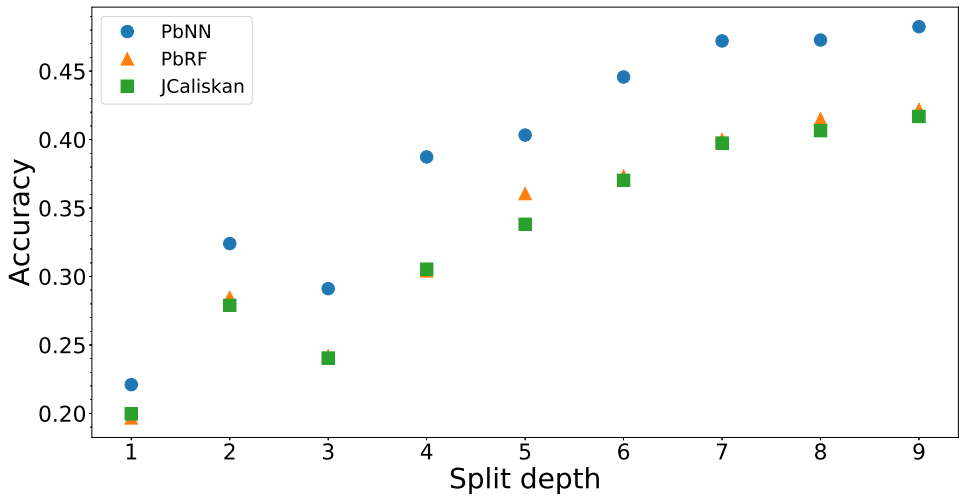
4.7.1 Separated work contexts

First, we work with separated work contexts (Section 4.6.2). These datasets contain nine different training/evaluation splits of the same large pool of method creation events, labeled with the method's author. Each split is parameterized with a depth value, which indicates the maximum possible depth of the lowest common ancestor of file changes in training and evaluation set.

If the model is sensitive to the influence of work context, it should perform better for higher split depths, because with the growth of the split depth training and testing sets become more and more similar. Figure 4.4 shows the dependency of the accuracy values of all three models on the split depth. Since the number of available samples per author is high and sufficient for proper training, the neural network model (PbNN) outperforms both random forest models (PbRF and JCaliskan) for each depth split and for both developer groups.



(a) 44 authors, 2,000 to 10,000 samples per author



(b) 21 authors, at least 10,000 samples per author

Figure 4.4: Models' accuracy on the datasets with separation of work context

Figure 4.4 shows that the accuracy values increase with depth, and at small depths the accuracy is much lower than in previous experiments (see Table 4.2). We tried to eliminate possible reasons for that, except for the difference in work context: the experiments were held on the same data points, sizes of the training sets vary by less than 3%, and we train models until convergence.

Thus, we conclude that *work context strongly affects accuracy of authorship attribution.*

4.7.2 Time-separated dataset

To see if developers' coding practices change in time, which might affect the accuracy of authorship attribution models, we evaluate our models on two collected datasets with folds separated in time (Section 4.6.2). The datasets contain samples of method creation from developers who did between 2,000 and 10,000 method creations and developers who did at least 10,000 method creations in the IntelliJ IDEA project. For each of the developers, the data has been divided by time into ten folds of equal size. This way we preserved the distribution of authors across folds, eliminating all differences between folds, except for the time when they were written.

We train a separate model on data from each fold except for the last. Then, the models are tested on code fragments from subsequent folds. Thus, for ten folds we get nine trained models and 45 fold predictions. We expect lower accuracy for more distant folds, if the developers' practices change in time.

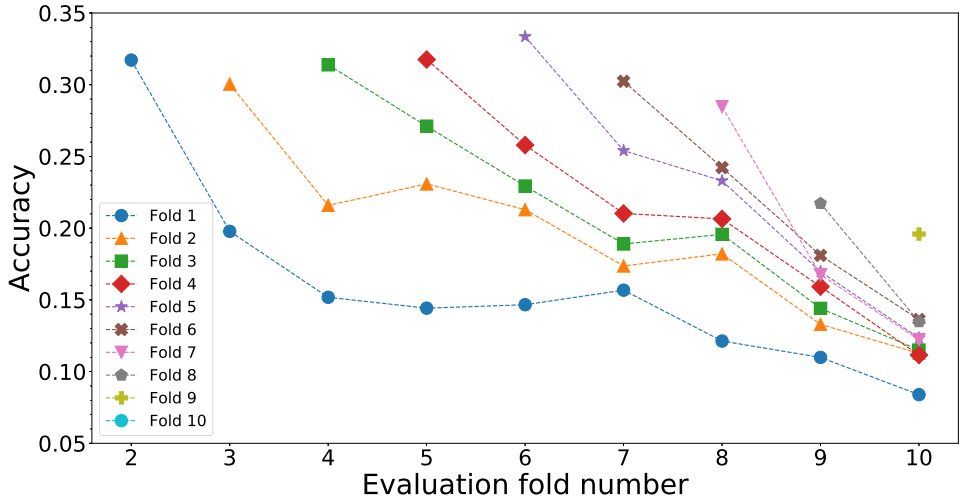
The results for the models are presented in Figure 4.5 and Figure 4.6. The neural network (PbNN) performs on par with random forest models (PbRF and JCaliskan) for the developers with at least 10,000 samples, but falls behind when the number of samples per author decreases. The graphs show that the accuracy of all models drops as distance in time grows, which confirms our hypothesis: *evolution of coding practices affects accuracy of authorship attribution.*

4.7.3 Evaluation on other projects

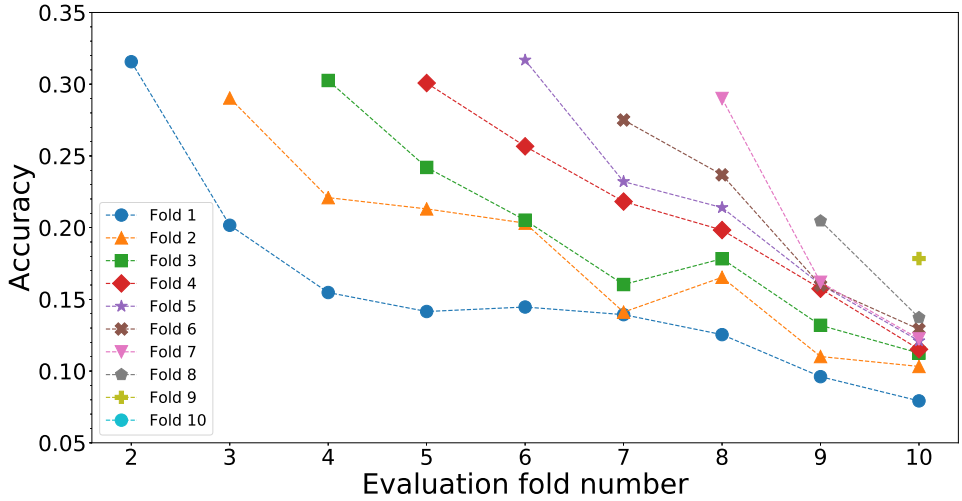
We also held a preliminary evaluation on two other large Java projects: Gradle [185] and mule [186]. We applied the proposed data collection technique and created context-separated and time-separated datasets from each project. The results are consistent with the evaluation on IntelliJ IDEA:

- Models' accuracy rapidly drops as we gradually separate contexts.
- Models' accuracy strongly depends on the distance in time between training and evaluation folds.
- PbNN outperforms both PbRF and JCaliskan when the number of available samples per author is high.

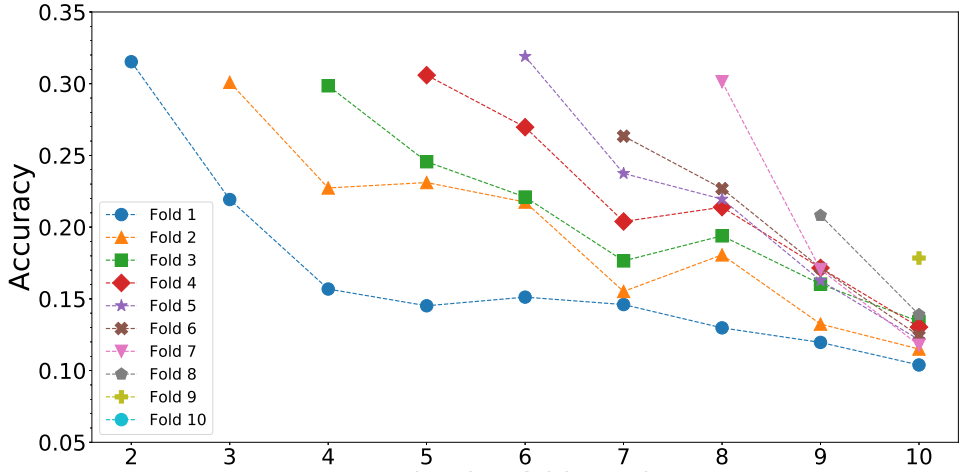
The detailed results and graphs are available on GitHub [183].

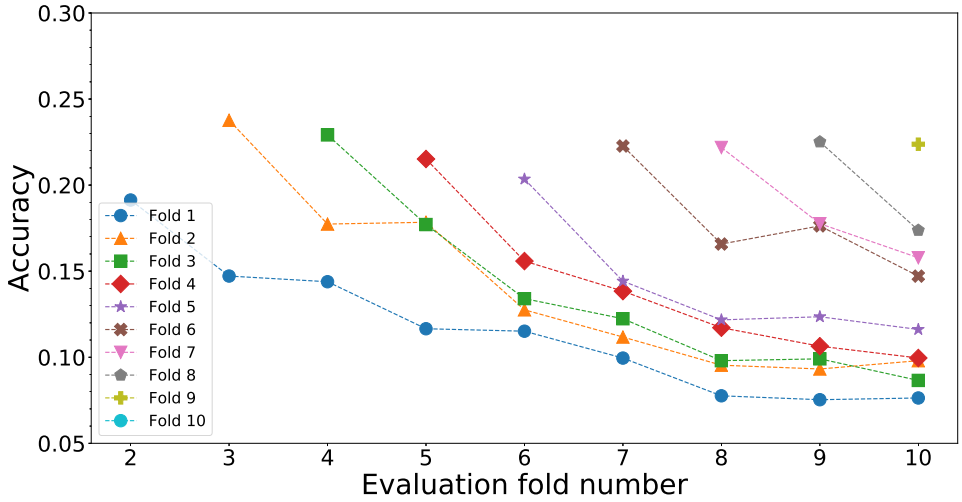


(a) PbNN's accuracy

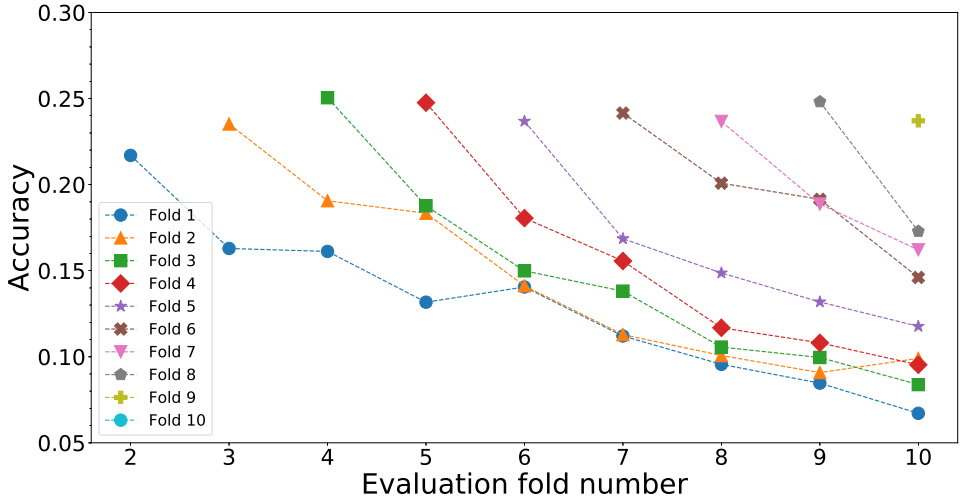


(b) PbRF's accuracy

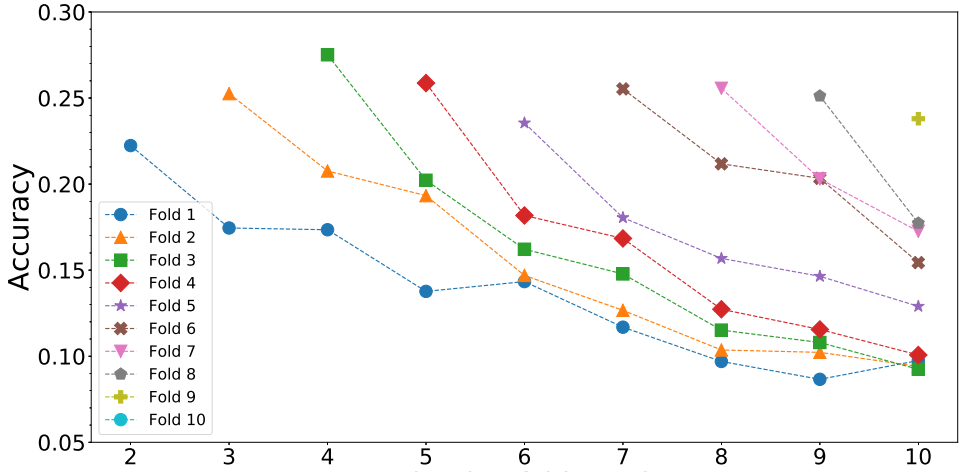




(a) PbNN's accuracy



(b) PbRF's accuracy



4.8 Discussion

Based on the evaluation results on the collected datasets, we conclude that both the difference in work contexts between training and testing data and the evolution of developers' coding practices have strong influence on the accuracy of authorship attribution models. However, our results are limited to Java language and several large open-source projects and need further research.

4.8.1 Influence of the work context

In Section 4.7, we described an experiment with separation of the work context between training and evaluation sets. We demonstrated that the model's accuracy decreases as we train and evaluate it on more distant (in terms of the codebase's file tree) pieces of code for each author. Specifically, the accuracy might vary by almost two times as we divide the same samples differently (Figure 4.4).

We conclude that the gap between datasets used in previous work and the data observed in practical tasks is not negligible. Specifically, a model's accuracy can drop from 97-98% in one setting (Table 4.2) to 22% in another (Figure 4.4). This suggests that, for evaluation to provide realistic information about potential accuracy of a solution on data from conventional software engineering projects, researchers should use datasets where training and testing part for each author belong to different environments. This supports the broader notion of the need to evaluate data-driven models in environments as realistic as possible.

The proposed dataset with gradual separation of training and evaluation sets can be used by researchers in future to measure their models' tendency to rely on context-related features rather than individual developers' properties. The models evaluated in this work turned out to have a strong dependency on work context as well, with the accuracy dropping from 48% to 22% (PbNN) and from 42.2% to 19.6% (PbRF) for splits at depth 9 and 1. To lower the influence of work context on the model's accuracy, researchers could design context-independent features or add regularization terms.

4.8.2 Evolution of developers' coding practices

Evaluation on the dataset with samples of code from each developer split in time showed that, as programmers' coding practices evolve over time, learning on older contributions to attribute authorship of the new code leads to lower accuracy of attribution. Moreover, coding styles of individual developers might even converge over time (Chapter 6) To maximize the accuracy in potential real-world scenarios, we should use training data that is as relevant as possible, and re-train or fine-tune the models as we gather new data samples.

4.8.3 Threats to validity

The proposed technique of data collection does not take into account automatically generated code, boilerplate code, or code embedded by an IDE. Its presence may bias the generated dataset.

The experiment with gradual separation of work context relies on the proposed method to measure work context similarity as the depth of the lowest common ancestor in a file tree. Despite the provided rationale on why it is reasonable, for some projects the similarity

of code might be weakly related to the location in the codebase due to specific codebase organization practices.

We found that the evolution of developers' coding practices strongly affects accuracy of authorship attribution models. However, the observed drop in models' accuracy can be caused in part by the evolution of the whole project instead of the individual programmers. Also, the observed results are limited to IntelliJ IDEA data. To extend them to a general case, further research is needed.

While the datasets collected from the IntelliJ IDEA repository suit the goal of measuring influence of work context and evolution of developers coding practices on authorship attribution, additional work is needed to create a dataset for proper evaluation of the models in a practical setting. It should comprise several projects with overlapping sets of developers, with projects divided between training and testing sets.

4.9 Conclusion

Authorship attribution of source code has applications in software engineering in tasks related to software maintenance, software quality analysis, and plagiarism detection. While recent studies of authorship attribution report high accuracy values, they use language-dependent models and do not assess whether their datasets resemble data from real-world software projects.

We propose two models for authorship attribution of source code: PbNN (a neural network) and PbRF (a random forest model). Both models are language-agnostic and work with path-based representations of code. Our evaluation on datasets for C++, Python, and Java used in recent work shows that the suggested models are competitive with state-of-the-art solutions in terms of accuracy. In particular, they improve attribution accuracy on the Java dataset from 91.1% to 97% (PbRF) and 98% (PbNN).

While demonstrating high accuracy, existing work in authorship attribution is evaluated on datasets that might be inaccurate in modeling real-world conditions. This may hinder their adoption in software engineering methods and tools. To formalize the differences, we suggest a concept of *work context* — the environment that influences the process of writing code — such as surrounding files, broader codebase, or team conventions. Taking work context into account, there is significant dissimilarity with previous results. Another concern investigated in this work is the evolution of developers' coding practices and its potential impact on accuracy of authorship attribution.

We suggest a novel approach to creation of authorship attribution datasets. In contrast to prior studies that are limited to projects with a single author, our approach can work with any Git project. We use our approach to process history of changes in a large Java project (IntelliJ IDEA Community repository on GitHub) and create several datasets to study the influence of work context and coding practices evolution on the accuracy of authorship attribution.

Evaluation of three models on the dataset with separation of work context shows that the accuracy goes down as similarity values decrease. As we gradually change the similarity level from maximal to minimal, the accuracy dramatically drops to the low of 22%, which is much lower than 98% achieved on the existing dataset of 40 single-authored projects. For the experiment with folds divided in time, the accuracy drops as the time difference between training and evaluation folds increases, and the drop might also be significant —

over 3 times for the most distant folds. We conclude that programmers' coding practices evolve over time, at least in large projects, and their evolution negatively affects quality of authorship attribution methods.

Our study demonstrates that existing solutions to authorship attribution can perform very differently when existing datasets are put into conditions close to real-world. This should be taken into account when evaluating authorship attribution approaches, especially during stages of data collection and training/testing division.

5

Reviewer Recommendation

This chapter is based on the paper "Does Reviewer Recommendation Help Developers?" by Vladimir Kovalenko, Nava Tintarev, Evgeny Pasynkov, Christian Bird, and Alberto Bacchelli, published in the IEEE Transactions on Software Engineering journal in 2018 and presented at the International Conference in Software Engineering in 2019 in Montreal, Canada.

5.1 Introduction

Code review, *i.e.,* manual inspection of source code changes, has become a standard step in software engineering [187, 188]. The inspection approaches have evolved over the last decades and these days developers commonly conduct change-based code reviews using dedicated tools [188, 189]. This lightweight, change- and tool-based approach to code review, commonly used in the software industry, is also referred to as Modern Code Review (MCR) in literature [9, 188, 190, 191].

Code review tools provide developers with a convenient environment to read and discuss code changes. The tools have evolved to support the reviewers with more features, such as integration with bug trackers and continuous integration tools [192–194]. The research community has proposed techniques that utilize historical data about development activity to optimize the code review process and tools further. A notable example of such technique is *automatic reviewer recommendation* — the focus of this study.

Automatic reviewer recommendation consists in having an algorithm that identifies the optimal reviewer(s) for a given changeset and provides a suggestion accordingly. Selecting the right reviewers for a changeset, as previous studies reported [188, 195], is a critical step in the code review process, because the knowledge and ability of reviewers can dramatically impact the quality of a review [188]. The common idea behind the automatic reviewer recommendation is modeling developers' experience to identify those developers who are the most experienced with the code under review. This expertise is thought to ensure their capability of providing good feedback [188], and it is commonly identified by analyzing the history of developers' code changes as well as participation in prior code reviews.

Academic researchers have proposed several approaches and models for automatic reviewer assignment and reviewer recommendation. Examples include recommendations based on prior reviewers of files with similar paths in the same project [9], on cross-project work experience of potential reviewers and estimation of their expertise in specific technologies [70], and on analysis of the history of file changes at line level [69]. Most approaches demonstrate high accuracy, sometimes as high as 92% for top-5 [69].

The analysis and the comparison of the performance of reviewer recommendation approaches have been largely based on evaluating how well these approaches can produce recommendations that match the historical records of actual reviewers. In practice, the evaluation consists in measuring how precisely the reviewer recommendation approach would have recommended the developers who actually did the review for a given changeset in the past, given the information available in the moment the review was requested. This evaluation is based on the assumption that reviewers who did review the code under the change before are (among) the best candidates to review it. Such an *offline evaluation* [196], performed on a historical dataset of reviews, is convenient because it enables the parallel comparison of multiple algorithms on the same data, does not require human input, and *does not interfere* with the observed phenomenon.

Reflecting on the primary goal of a reviewer recommendation system, we see that such a system should help developers making their choice of a reviewer for a changeset. This help is the most valuable in scenarios where this choice is not completely clear. Consequently, the effect of a reviewer recommender system can be described as *positively influencing* the users' behavior by mitigating their difficulties with making an informed choice.

Offline evaluation leaves this critical aspect out of the picture. Influence from the recommender system on user's decision is particularly likely to occur when the user does not have an intention to select a particular person as a reviewer beforehand; in such cases recommendations can serve as hints, directing user's choices towards recommended options. Evaluating recommendation algorithms against actual reviewers from historical data does not allow to account for this effect, because the users do not interact with the recommender in this case. This limitation is not specific to reviewer recommendation, but is typical for all recommender systems [196].

Another effect that is not taken into account by existing evaluation techniques for reviewer recommendation is whether and how the recommendations play a different role for different users. For example, novice developers or newcomers to a team may find a recommendation more helpful, since these users are known to benefit from guidance and mentoring the most [197]. In contrast, for experienced members of a small team, where codebase ownership is clearly split between developers, reviewer recommendation may be a less useful or even redundant feature, as also hypothesized by Baum *et al.* [198]. The ability of recommendation models to take the individual user's needs into account may be no less important for the real-world tool context than how suitable the recommended reviewers are for corresponding changes. However, existing evaluation methods omit these aspects, focusing solely on comparing the alignment of recommendations with reviewers from historical data.

Acknowledging these aspects, which are specific for scenarios where a recommender is *deployed*, looks like the next important step in the evolution of reviewer recommendation. Moreover, these arguments fall in line with state of the art in the research field of recommender systems, where the idea of considering a broader set of metrics beyond accuracy of algorithms has recently been gaining traction [199, 200].

The increasing adoption of reviewer recommendation in industrial tools (*e.g.*, [22, 201, 202]) brings an unprecedented opportunity to bridge the gap between offline evaluation of reviewer recommendation algorithms and their actual value for the code review process. This study is our take on this opportunity. In collaboration with two companies (JetBrains [203] and Microsoft [204]), we conduct, for the first time, a longitudinal, *in vivo* study to explore the experience of users with reviewer recommendation in the setting of commercial software companies.

In our study, we use a mixed quantitative/qualitative approach. In the first, quantitative stage at JetBrains, we analyze the history of over 21,000 code reviews that were performed in the company's internal instance of Upsource.[1] By reproducing the historical recommendations from Upsource, we set out to measure the accuracy of a *deployed* reviewer recommender and identify the impact of recommendations on users' choices. Thanks to a change of the recommendation model amid the longitudinal data period, we have an opportunity to seek evidence of such impact by observing the trend of recommendation accuracy, relative to choices of users, around the point of the model change. Unexpectedly, we find no evidence of such impact: the accuracy does not noticeably change with the change of a deployed model.

[1] Upsource is a code review tool developed by JetBrains, which is available as a commercial product and also used for code review by the developers at JetBrains.

In the second stage, to gain a deeper understanding of reviewer recommendation beyond its accuracy, we turn to the users. Through four interviews and a survey of 16 respondents at JetBrains, we explore how the developers perceive and use the reviewer recommendations. We find that, despite being generally perceived as relevant, *automatic reviewer recommendations are often not helpful* for the users at JetBrains. To validate this unexpected finding, we conduct a more extensive survey at Microsoft (508 responses to a survey consisting of 22 questions, both Likert scale and open), which generally confirms the result in another company.

Overall, our results suggest that accepted evaluation measurements are misaligned with the needs of most developers in the company settings we investigated. This misalignment highlights the importance of carefully considering the context when developing reviewer recommender mechanisms and when selecting the corresponding evaluation techniques. Indeed, our setting is an example of environments where the established means of evaluation do not match well the value of the recommendations for users.

Finally, we use the responses from Microsoft to identify scenarios of demand for reviewer recommendation and propose a new, more user-centric and context-aware take on this problem.

Our study makes the following contributions:

- The first *in vivo* evaluation of reviewer recommendation as a code review tool feature, in the context of two commercial software companies, investigating empirical accuracy (RQ1), influence on reviewer choices (RQ1), and added value for users (RQ2);
- Empirical evidence on the importance of metrics beyond accuracy for the evaluation of reviewer recommendation systems (RQ1);
- Analysis of users' perception of reviewer recommendation features, which challenges the universality of the use case for reviewer recommendation in commercial teams and underlines the importance of context (RQ2);
- An investigation of the information needs of developers when selecting reviewers, suggesting directions for further evolution of reviewer recommendation approaches (RQ3);
- Empirical evidence on categories of developers with more difficulties selecting reviewers than others (RQ3).

5.2 Background and motivation

5.2.1 Code Review

Code review is a practice of manual examination of source code changes. Its primary purpose is early detection of defects and code quality improvement [188]; other goals include distribution of knowledge and increase team awareness, as well as promotion of shared code ownership [188, 205].

In modern development environments, code review is typically performed on code changes, before these changes are put into production, and is done with dedicated tools [205]. Code review tools mostly provide logistic support: packaging of changes, textual diffing

(for reading the changes), inline commenting (to facilitate discussions among authors and reviewers), and accept/reject decisions. A few tools provide additional features to extend user experience. Examples of such features include code navigation (*e.g.*, [193]), integration of static analysis results (*e.g.*, [193, 206]), and code repository analytics (*e.g.*, [193, 207]).

5.2.2 Recommender systems

The research field of recommender systems investigates how to provide users with personalized advice in various information access environments. Prominent applications for these techniques are online marketing [208], web search [209], social media content filtering [210], and entertainment services [211–213]. Another line of research is dedicated to recommending experts for various applications in knowledge-heavy contexts, such as academic research and software development [214, 215]. In particular, recommender systems are proposed in a variety of software engineering related scopes and are targeted towards improving the efficiency of development and quality assurance activities. Examples of problems tackled with recommendation approaches include bug triaging [216], defect localization [217], identification of related Q&A threads [218], and recommendation of code reviewers [9].

Most of the research in the broader field of recommender systems had focused on devising core recommendation algorithms able to predict the choices of users — for example, predicting the books that were eventually bought or rated as high quality by a given user. Design of such algorithms and their evaluation are typically conducted on historical datasets (*e.g.*, by splitting the data in temporal order for training and evaluation [219]), and do not require actual interaction of users with the evaluated algorithm. Hence, such evaluation techniques are called *offline experiments*. Such experiments do not allow to capture the factors influencing user satisfaction, or what happens with the quality or perception of the predictions over time [220–223], or aspects of user interaction with the recommender.

A more powerful alternative is *live user experiments*, which are essential to evaluate finer aspects of recommendation quality, user experience, and business metrics based on outcomes of the interactions [224]. However, live user studies of recommender systems are rare. Large-scale live user experiments, that should involve interaction of real users with the recommender, are costly: an experiment requires a long-running infrastructure to support the data collection for high-quality predictions, and poses the risk that some interventions may lead to worse recommendations for sub-groups of users (*e.g.*, in the case of A/B testing).

A common measure for evaluation of recommendation engines, which is typically a focus of offline evaluation, is accuracy — the measure of recommender's ability to model actual choices of users, thus providing an output that is relevant to them. Thanks to the rising adoption of recommendation engines in consumer services and tools, researchers could start moving beyond accuracy as they consider complementary metrics for evaluation. This includes considering factors such as *diversity* [225–227], *novelty* [224, 228], and *serendipity* [199, 229] alongside accuracy. Along with the expansion of the spectrum of evaluation metrics, the nature of interactions between users and recommender systems, and the influence that user interface and interaction style have on user behaviour and overall recommendation experience [230–232] have also been attracting more attention.

One particular gap in this literature is lack of investigations of change of the quality

of recommendations over time, and how this change influences interaction of the users with the recommenders [233]. This gap can be attributed to the high cost of obtaining longitudinal data at large scale. Nguyen *et al.* found that recommender systems decrease the diversity of content that users consume over time [233], and Bakshy *et al.* found that both algorithms and users contribute to over-tailoring of recommendations [234].

Recommendations based on historical data (which also includes several models of code reviewer recommendation) are subject to feedback once deployed — the recommended items that were chosen by the user have a higher chance to be recommended in the future. Impact of this effect on the value of the recommender systems is double-sided. Learning user's preferences through interaction history can reduce user effort [235]; however, relying too heavily on recommender systems may result in a negative effect for other factors, such as sales diversity [236]. In more complicated information retrieval contexts, the long-term impact of feedback is also controversial. For example, in online social media content filtering, feedback can undermine the diversity of users' interaction scopes [237].

5.2.3 Reviewer recommendation

Researchers provided evidence that inappropriate selection of reviewers can slow down the review process [9]. As a consequence, recent work in software engineering research is dedicated to building reviewer recommendation approaches to support developers during the critical step of reviewer selection. The common idea behind these approaches is to automatically identify potential reviewers who are the most suitable for a given change. The main proxy for suitability estimation is expertise (or familiarity) of candidates with code under review, which is estimated through analysis of artifacts of developers' prior work, such as histories of code changes and review participation [69, 71, 190].

The exact mechanics of reviewer recommendation vary between approaches. Some techniques are based on scoring of candidates, either based on changes history at line level [69] or on analysis of historical reviewers for files with similar paths [71]. Another approach is machine learning on change features [238]. Other studies incorporate additional information, such as socio-technical relationships [191], reviewer activity information from past reviews [190], social interactions between developers [40], and expertise of potential reviewers with similar contexts in other projects [18, 70].

5.2.4 Practical motivation

Existing research in a broader scope of recommender systems suggests that evaluation of recommendation algorithms should go beyond offline evaluations and accuracy measures: there is demand for methods that consider the real-world impact of recommender systems. Such methods are essential to gain a deeper understanding of long-term effects of recommendation systems, and to facilitate their adoption. In this work, we set out to shed light on the value of reviewer recommendation for users of code review tools, by conducting the first live user evaluation and taking a more user-centric approach to this increasingly popular topic. By using the records of development activity and interviewing developers at two software companies, we are particularly focusing on the accuracy and perception of a reviewer recommender in commercial teams.

A particularly interesting effect in the context of reviewer recommendation is the influence that the recommendations may have on choices of the users exposed to a recommender.

A similar effect was described by Cosley *et al.* [239]: users of a movie recommendation tool, when asked to rate movies, displayed a small but significant bias towards a predicted rating. Presence of such effect in the interaction of users with a recommender system could lay the foundation for the collaboration tools to help with controlling large-scale characteristics of software projects, such as the distribution of code ownership.

5.3 Research Questions and Setting

In this section, we present the research questions, the research settings, and an overview of the research method.

5.3.1 Research questions

We organize our results along three research questions (and corresponding sub-questions), which we have iteratively refined during the investigation.

RQ1: How does a reviewer recommendation system perform in practice? (Section 5.4)

RQ1.1 Do the recommendations influence the choice of reviewers? Investigating the performance of a reviewer recommender system in a deployed tool is interesting from the perspective of identifying potential effects that are specific to an online scenario. In RQ1.1, we are looking for evidence of the most important of such effects: influence of recommendations on choices of users.

RQ1.2 How accurate are the recommendations of a deployed recommender? In RQ1.2, we focus on the accuracy of reviewer recommendations. While a number of previous studies cover the accuracy aspect, it is important to evaluate it in our online scenario separately: feedback from recommendations to choices can possibly inflate observed accuracy of a deployed recommender.

RQ1.3 What are other performance properties of the deployed recommender? RQ1.3 is dedicated to performance properties of the recommender apart from accuracy. We find it a worthwhile question to formulate, because common metrics for evaluation of reviewer recommenders are limited to accuracy figures. Accuracy-centric approach is obsolete with regard to recent achievements in the Recommender Systems research field, where it is now established that other properties of a recommender are no less critical for a real-world system than its accuracy.

Afterwards, we investigate the perception of the reviewer recommender by users. Through interviews and surveys, we aim to understand if developers perceive the recommendations as accurate, relevant, and helpful:

RQ2. How do developers perceive and use reviewer recommendations? (Section 5.5)

RQ2.1 Do developers need assistance with reviewer selection? With this question, we investigate to what extent the reviewer selection process is challenging for developers.

RQ2.2 Are the reviewer recommendations perceived as relevant? With this question, rather than comparing recommendations against choices, we ask users about their perception of recommendation quality — in particular, whether the recommendations appear relevant.

RQ2.3 Do the recommendations help with reviewer selection? This question addresses the role of reviewer recommendations in the process of reviewer selection. To provide additional value, a recommender system does not only have to be accurate, but it should also be helpful with regard to the information needs of the users.

The information needs during reviewer assignment may (1) be different for different users and (2) be not satisfied by current reviewer recommender systems. To provide suggestions for further improvement of reviewer recommendation approaches, we investigate the information needs of developers who select reviewers for a change.

5

RQ3. What are the information needs of developers during reviewer assignment? (Section 5.6)

Figure 5.1: Main interface of Upsource — the code review tool used at JetBrains

RQ3.1 What kinds of information do developers consider when selecting reviewers? This question aims to better understand the reviewer selection process by figuring out the most relevant types and sources of information.

RQ3.2 How difficult to obtain are the different kinds of information needed for reviewer selection? Some of the important information may be more difficult to obtain for the user. It is an important factor for the design of recommendation systems, as they are capable of obtaining and aggregating information that is harder for users to get otherwise, such as modification history of files. With this question, we aim at identifying such types of information for reviewer selection.

RQ3.3 When is it more difficult to choose a reviewer? The task of reviewer selection may be more challenging in some scenarios, such as when changing the legacy code, or for a new team member. Future reviewer recommendation approaches could also consider the context of changes — for example, by only offering recommendations when there is a clear demand for them. With this question, we aim to identify such situations, in which a recommender could be more helpful.

5.3.2 Research Settings

The study we conducted to answer the research questions took place with professional developers, managers, and data from two commercial software companies.

JetBrains: The first subject company is a vendor of software tools for a niche area of professional software developers. The company has over 700 employees, most of whom are located in several development centers across Europe. Upsource, a code review tool, is one of the products of the company and includes a recommender for reviewers. Different teams at JetBrains have been using Upsource for code review since its early releases in 2014 and, subsequently, have used the reviewer recommender since it was implemented in Upsource. However, with no centralized code review policy in place, adoption of Upsource inside the company and within individual teams is underway.

Microsoft: The second subject company is a large corporation that develops software in diverse domains. Each team has its own development culture and code review policies. Over the past eight years, CodeFlow — a homegrown code review tool at Microsoft — has achieved company-wide adoption. As it represents a standard solution for code review at the company (over 50,000 developers have used it so far) and offers an integrated reviewer recommendation engine, we focused on developers who use this tool for code review.

Code review tools. The functioning and features of code review tools, including Upsource and CodeFlow, are substantially the same. Here we explain the functioning, by considering Upsource as an example.

Upsource is a commercially available code review tool. It provides code discussion facilities, code highlighting and navigation, and repository browsing features. Figure 5.1 is a screenshot of the code review interface in Upsource.

Apart from these standard features, Upsource is capable of recommending reviewers for code changes. This feature is central for this work. When a new review is created from a set of commits, the tool analyzes the history of changes and reviews of changed files and ranks the potential reviewers according to their relevance. Then Upsource presents a list of relevant developers to be quickly selected as reviewers with one click. (Figure 5.2). The user can opt to use a search form to add reviewers manually. In such case, the history-based recommendations are presented in the search results as well (Figure 5.3). We detail the internal structure of the recommendation algorithm in Section 5.4.2.

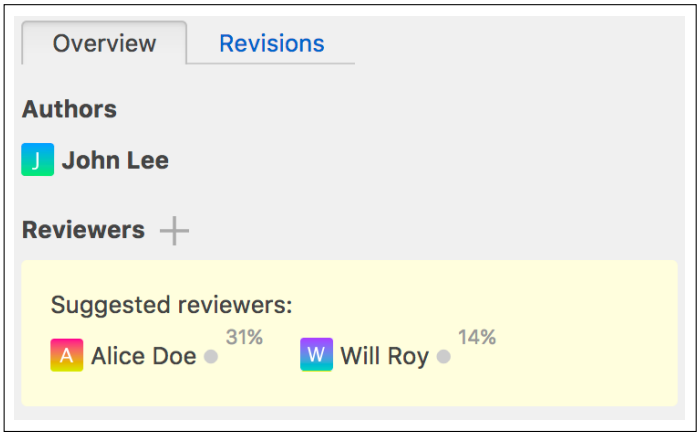


Figure 5.2: Instant reviewer suggestions in Upsource

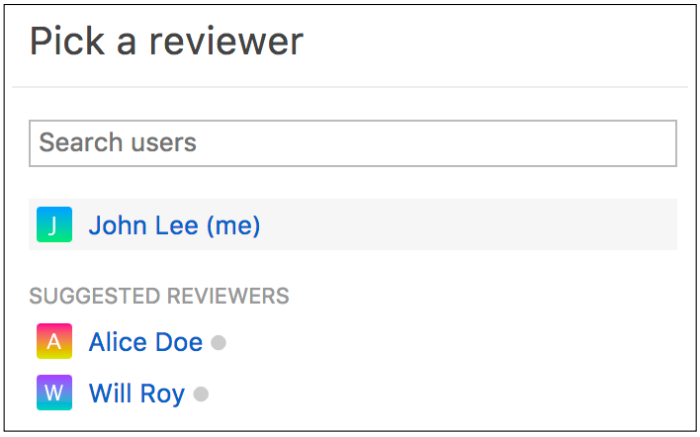


Figure 5.3: Reviewer suggestions in search form in Upsource

Both in Upsource and CodeFlow, the reviewers can be added to a review by any user with corresponding rights, which are typically held by all team members. The standard scenario in the code review workflow both at Microsoft and JetBrains is that it is the author of the change who initiates the review and selects the colleagues whom they prefer to invite as reviewers.

A distinguishing feature of CodeFlow (the code review tool used at Microsoft) is the option to configure the recommendations according to the policy of a team. For example, one team can decide that all the reviews for certain files are sent to an alias visible by all developers in a specific team.

5.3.3 Study overview

Figure 5.4 presents a schematic view of the research method employed for investigating the research questions. We briefly describe our method in the following and provide details by research question in the next sections.

Our study followed a mixed-method (qualitative and quantitative) approach [240]. We collected and analyzed data from different sources for triangulation and validation. We conducted our study in three phases.

In the first two phases, we teamed up with JetBrains. In the first of these two phases, with the help of Upsource developers and with a team of Upsource users at JetBrains, we have reproduced reviewer recommendations that were given to the users in over 21,000 reviews that took place across the period of 2.5 years. To evaluate recommendation accuracy, we also collected the records of actual reviewers in those reviews. In the second of these two phases, we conducted interviews and sent a survey to JetBrains developers to collect data on developers' perception and usage of recommendations. In the third phase we turned to Microsoft: We expanded the scope of the investigation and validated our outcomes from the first two phases through a separate structured survey, by targeting the developers working at Microsoft.

We used the quantitative data from the deployed recommender system at JetBrains to answer RQ1. Responses from interviews at JetBrains and surveys at both companies were the primary data sources for RQ2. RQ3 was based on the responses to a large-scale survey at Microsoft.

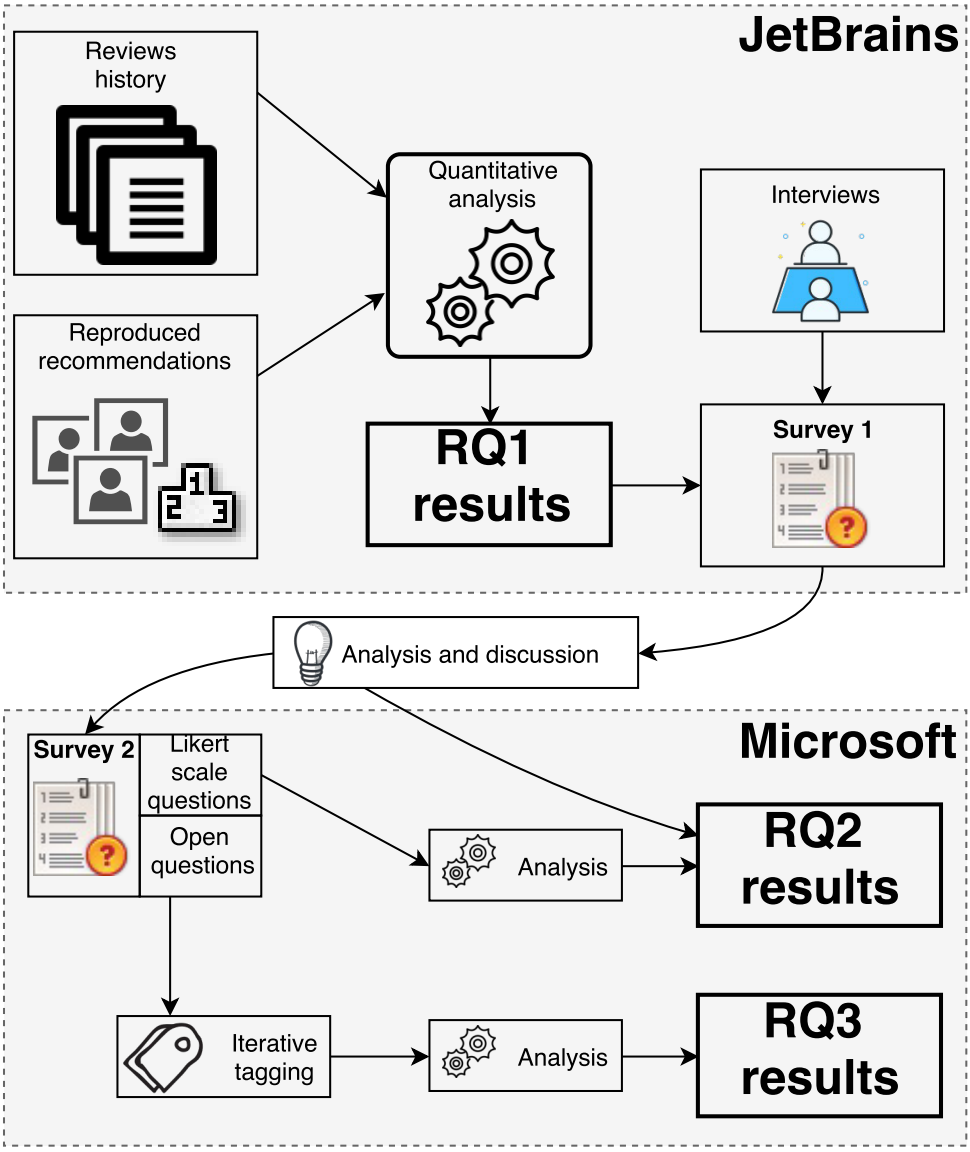


Figure 5.4: Overview of the research method

5.4 RQ1: Performance of the deployed reviewer recommender system

Our first research question seeks to empirically investigate the performance of a deployed reviewer recommender.

5.4.1 Data collection

To answer this research question, we have reproduced the recommendations for reviews in the codebase of JetBrains' flagship product — IntelliJ IDEA Ultimate. To extract the necessary dataset from the backup files of the internal Upsource instance at JetBrains, the first author of this article devised a custom build of Upsource, which included a custom module for reproducing the recommendations and dumping the data.

For every completed review, we identify the events of a reviewer being manually added to a review. For each of these events, we reproduce the recommendations that were given to the user who added the reviewer. We identify historical recommendations by sandboxing the components of the actual recommender system and reproducing its output.

Each observation in the dataset represents an event of manual selection of a reviewer. For each of these events, the dataset contains records of the selected user, the user who made the selection, and the recommendations made by two different models. In contrast with studies where the list of recommendations is usually compared against a list of actual reviewers in the whole review, our observations are more fine-grained because a single review can contain multiple addition events. This data structure is mostly dictated by the recommendation algorithm (described in Section 5.4.2), and it also imposes limitations on the metrics that can be used to evaluate recommendation accuracy (as we explain in Section 5.4.3).

5.4.2 Reviewer recommender internals

Figure 5.5 presents the scheme of the reviewer recommendation system in Upsource.

For every review, (1) recommendations are calculated based on the changes that are included in this review. For every modified file in the change set, (2) Upsource retrieves the history of all the previous commits affecting these files. For each of these commits, the recommender gathers the (3) VCS meta-data, such as the author and timestamp of the commit and the list of developers who reviewed them. This information is compiled into (4) the input data for the recommendation model. To disambiguate several versioning system aliases of the same user, we associate the aliases with user profiles in an external user management tool.

Based on this input data, for every author and reviewer of the past versions of the files, the recommender model computes a relevance score, based on recency, count, and magnitude of developers' prior contributions (both as authors and as reviewers) to the files under review. The score is designed to represent the degree of familiarity of each developer with the code under review. This approach is aligned with state of the art in reviewer recommendation [241], [190].

The recommender system filters the list of potential recommendations (5) to remove irrelevant candidates: users who already participate in the review as reviewers (such as 'Jack Shaw' in Figure 5.5), users who have no review access (*e.g.*, because they left the

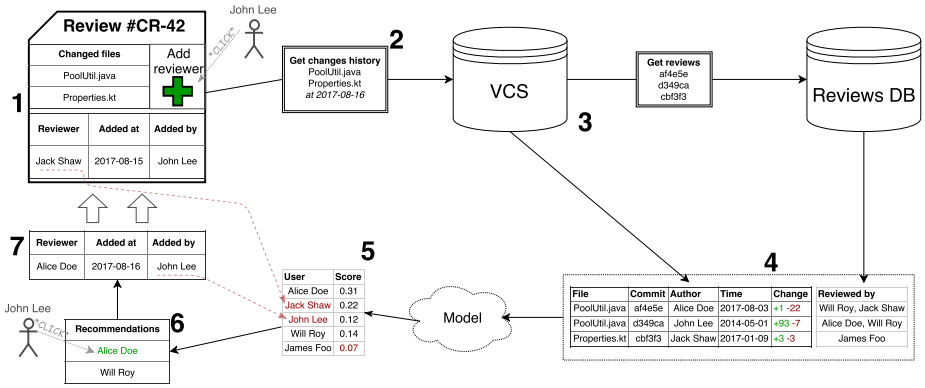


Figure 5.5: Recommendation system workflow. On 2017-08-16, John Lee wants to add a second reviewer to their change (1) in addition to Jack Shaw, who is already a reviewer. Upsource collects history of changes for files under review and history of reviews for these changes (4). Based on this data, the recommendation model scores potential reviewers (5). The scored list is filtered, leaving out the current reviewers (Jack), the author (John), and users with low scores (James). The remaining users are converted into a list of recommendations (6). Here a recommended user (Alice) is selected from a list of 2 recommendations, yielding precision of 0.5, recall of 1, and MRR of 1.

company), users with a score that is too low (such as 'James Foo'), and the author of the change to be reviewed. Finally, (6) the recommender presents at most three of the remaining candidates to the user, who may select one (as in this case) or more, or add someone else through manual search.

Two recommendation models

The scoring algorithm, which is at the heart of the recommender system in Upsource, was changed one year and a half after it was deployed. The change in the scoring algorithm, made along with a refactoring and a performance optimization of the recommendation backend, was triggered by user feedback indicating occasional irrelevant recommendations: *"It is better not to recommend anyone than to recommend a random person" — Upsource dev lead*, thus the change focused on reducing the number of recommendations.

For our study, this change of the scoring algorithm amid the longitudinal data period is a ripe opportunity to observe the effects of this change on the overall performance of the recommender system.

We refer to the first and the second versions as *Recency* model and *Recency+Size* model, respectively. The *Recency* model weights individual contributions of a user to every file, based on their recency: The more recent changes are given more priority to account for the temporal decay of user's expertise [35]. The size of a change does not influence the weight of a contribution in *Recency* model, and reviewing a change is considered an equally strong contribution as authoring it. The *Recency+Size* model, in contrast, takes the sizes of contributions into account; furthermore, authoring a change is considered a stronger contribution than reviewing it, and a different temporal decay function is used.

5.4.3 RQ1.1 — Do the recommendations influence the choice of reviewers?

Detecting the influence on choosing the reviewers

In an online setup, without a controlled experiment, it is impossible to directly measure the impact of recommendations on choices made by users. However, the change of recommendation model amid the data period (Section 5.4.2) gives us a chance to seek evidence of such impact.

If the recommendations played a significant role in determining the choice, the set of selected reviewers (also, the output against which the model is evaluated) would be partially defined by the recommendations. As a consequence, the influence of the recommender would lead to an increase in the observed accuracy of the recommendations.

We illustrate the nature of this effect with an exaggerated hypothetical example. Consider Alice, who always decides whether to ask Bob or Charlie to review her changes by tossing a coin. Also consider an isolated reviewer recommender system, that is as simple as tossing another coin and recommending the corresponding reviewer. If we evaluated such a recommender system offline on the history of reviewers of Alice's changes, in the long run, its precision would converge to 0.5 — the odds of two coins landing on the same side. However, if Alice indeed used that recommender, and followed its recommendations (rather than her own coin) at least once in a while, evaluation of the output of the recommender on historical data would yield higher precision values, because the recommended and the chosen reviewer would match beyond random occasions.

We have reproduced the recommendations provided by both models for the whole period. Given that the outputs of the models are different, if we consider the moment in which the deployed model changes, a change of accuracy at this moment would indicate an influence of the recommendations on choices. For example, if choices are biased towards recommendations of the *Recency* model, its observed accuracy would experience a drop at the moment the *Recency+Size* model gets deployed instead. A similar argument works for the *Recency+Size* model — its accuracy should increase once it is deployed, if the recommender influences the choices of users.

Considering the explanation above, we explore the accuracy trend around the model change date to conclude whether we can observe the influence.

Adjusted accuracy metrics

The commonly used metrics to quantify the accuracy of recommendations are top-k accuracy and MRR. These are reasonable metrics of list similarity, which makes them a good choice when the task is to compare two lists — recommended and selected items. However, these metrics are not a good fit for our event-based data. In our case, at every event of reviewer addition, a list of recommendations should be evaluated against exactly one selected user. Because the scope of recommendations in our target system is one review, it might seem reasonable to merge all observations for a given review into one list and use the conventional metrics. It is, however, not feasible for a thorough evaluation: the recommendation output is influenced both by the user who adds a new reviewer and by a set of previously added reviewers. Thus, if the recommendations for a given review were compiled in a list, every item in that list would be defined, among other things, by order of

the previous elements in this list. Thus, the only feasible option is to define and calculate accuracy per individual event, and then aggregate these numbers over time periods.

In line with previous investigations on recommender systems for software engineering [190, 216], we calculate the two widely used accuracy metrics — precision and recall, adjusted to our specific case of calculating the match between a set of recommendations and exactly one chosen user. In addition, we calculate adjusted MRR and use it to complement the recall values: unlike recall, MRR considers ranking of recommendations in the list.

Specifically, precision for a recommender system is defined as the fraction of relevant items among all recommended items. In our specific case, as the recommendations are calculated from scratch for every new added reviewer, exactly one item is selected at a time. Thus, for each event, given a non-empty recommendation set $Recs$ for an event where a user U was added as a reviewer, adjusted precision P is defined as

$$\text{Precision } P = \begin{cases} \frac{1}{|Recs|} & \text{if } U \in Recs \\ 0 & \text{otherwise} \end{cases}$$

5

Adjusted recall R, the measure of how fully the recommendations cover the relevant items, is defined in a similar way to P:

$$\text{Recall } R = \begin{cases} 1 & \text{if } U \in Recs \\ 0 & \text{otherwise} \end{cases}$$

While, as described above, MRR would not be a good primary metric for the recommendation accuracy, it complements the recall by penalizing the recommender for placing the correct recommendation below the top position in the recommendation list. Therefore, alongside precision and recall, we calculate the MRR, adjusting it to our scenario of a single recommendation:

$$MRR = \begin{cases} \frac{1}{\text{rank of } U \text{ in } Recs} & \text{if } U \in Recs \\ 0 & \text{otherwise} \end{cases}$$

Results

As a first step, we compare the output of the two models to ensure that they are dissimilar for the same input, so that we could see a difference in case of influence of the model. The outputs of the two models are indeed dissimilar: The mean value of Jaccard similarity index [242] between the recommendation lists provided by the two models for the same event is 0.502.[2]

If the recommendations had a significant influence on the choice of reviewers, we would expect the models to demonstrate higher precision when evaluated during the period of their deployment, than during the period when another recommendation model was in place, as an effect of the influence of recommendations on the choices (Section 5.4.3). Figures 5.6 and 5.7 suggest lack of such effect: we do not see any increase in precision in

[2]For calculating the Jaccard index, we only consider the events where at least one of the two models provided a non-empty recommendation list, because (1) we only consider such events for calculating the accuracy metrics and (2) computing the Jaccard index for two empty sets would imply division by zero.

Figure 5.7 at the moment when the *Recency+Size* model was deployed. On the one hand, the increase in precision could be dampened by the increase of the pool of potential reviewers: it is harder for the model to select a few relevant users from a bigger pool, which could cause precision to degrade. However, coverage figures (Section 5.4.5) suggest that it is unlikely to be the case, and the recommendations from *Recency+Size* model is actually more focused: *Recency+Size* model recommends the smaller proportion of the pool (only two in three active users ever got recommended by *Recency+Size* model), and these recommendations are picked more often: "intersection/recommended" and "match/recommended" ratios are higher than those of *Recency* model. Another argument towards the lack of influence is that we do not see a decrease of precision of similar nature in Figure 5.6 at the point when the *Recency* model went out of use. From this observation, we conclude that the increase in the number of active users does not directly decrease precision, and lack of shift in the precision of one model at the point where the deployed model has changed can be interpreted as an indication of the weakness of influence of model's recommendations on users' choices.

We expected to see a noticeable shift in precision values at the moment of change of the deployed model, as a sign of the influence of recommendations on choices. Figures 5.6 and 5.7 display no such shift.

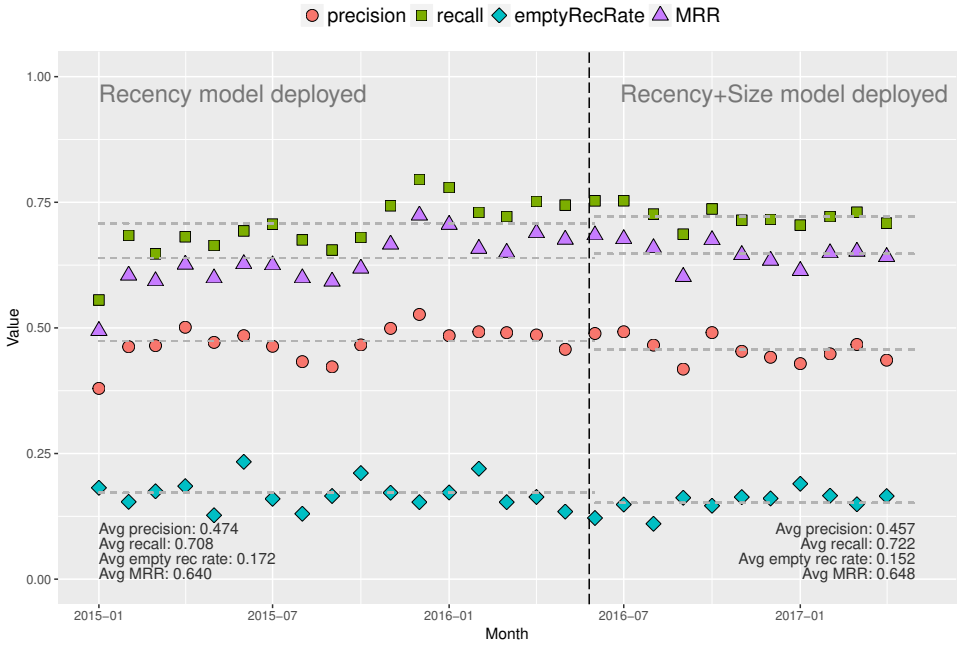


Figure 5.6: Accuracy of the *Recency* model relative to user choices. During the second period, a different recommendation model (*Recency+Size*) was in use. However, difference in accuracy values between the two periods is well within monthly variance.

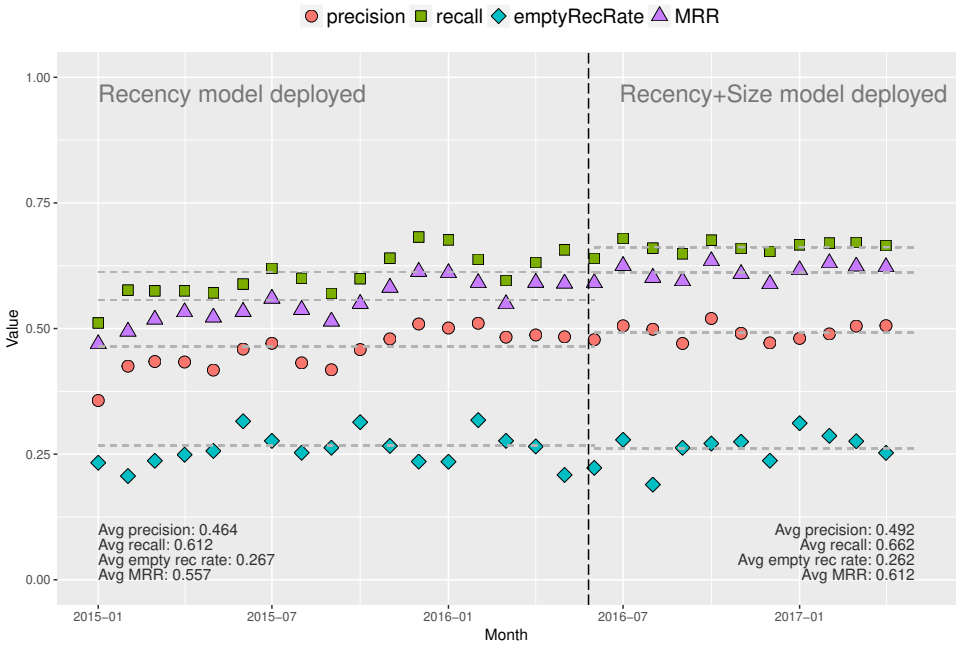


Figure 5.7: Accuracy of the *Recency+Size* model relative to user choices. During the first period, a different recommendation model (*Recency*) was in use. However, difference in accuracy values between the two periods is well within monthly variance.

5.4.4 RQ1.2 — How accurate are the recommendations of a deployed recommender?

Figure 5.8 presents average values of the three accuracy metrics for non-empty recommendations, as well as the frequency of empty recommendations, over the monthly periods. Evaluated during the deployment period, both models demonstrate accuracy values within the known range of prototypes. The *Recency+Size* model, due to a different scaling formula, is more conservative with recommendations — compared to the *Recency* model, it is 52% more likely not to produce any recommendations. It leads to a 6.5% lower mean recall and a 4% higher mean precision than for the *Recency* model. The mean MRR value for the *Recency+Size* model is 4% lower. Notably, the difference in average accuracy metrics between the two models is within the range of variance of these metrics between consecutive monthly periods for each of the models.

5.4.5 RQ1.3 — What are other performance properties of the recommender?

Recommendation count and coverage metrics

Precision and recall are only defined when the set of recommendations is not empty. Thanks to internal filtering in the recommender (described in Section 5.4.2), it is possible that in some cases the model gives no recommendations. To account for events with empty

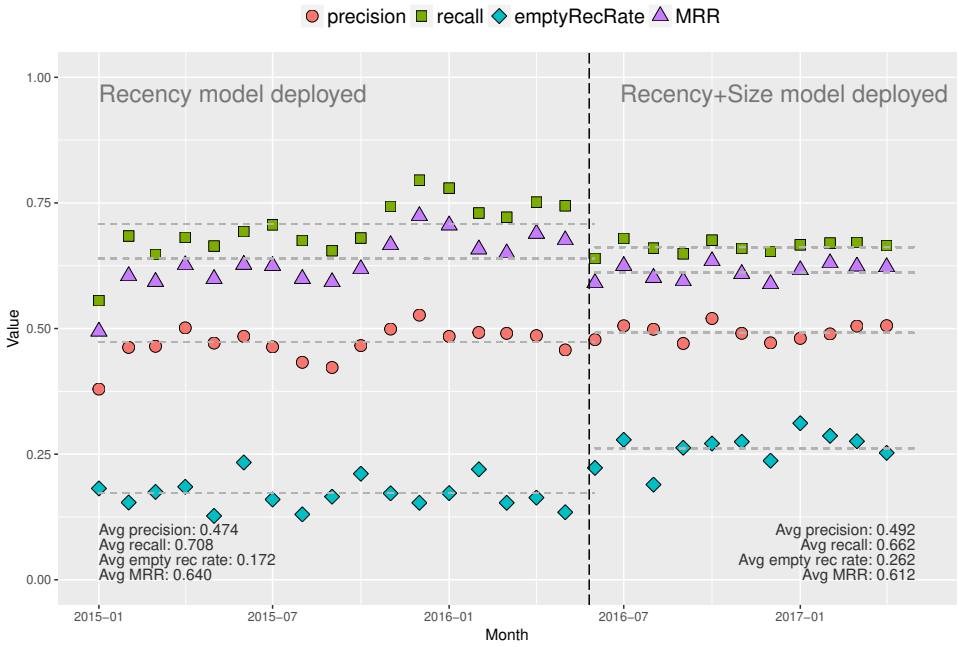


Figure 5.8: Accuracy metrics for non-empty recommendations and rate of empty recommendations over 1-month periods

recommendation lists, where precision and recall cannot be defined, we use frequency of empty recommendations and count of recommendations as auxiliary metrics.

In addition to recommendations count, we augment the accuracy numbers with numbers of catalog coverage — a measure of how many of the users who can hypothetically be recommended do get recommended. In the absence of other studies that consider this parameter of a reviewer recommender, and of a steady definition of the catalog in this context, we use several metrics related to the catalog coverage. For the periods of deployment of each of the two recommendation models, we calculate the following:

- Number of developers who made the code changes in the project *in and before* the period. This number represents the pool of users who can potentially be recommended.
- Number of developers who were recommended as a reviewer at least once during the period.
- Number of developers who were selected as a reviewer at least once.
- Size of the intersection of the previous two sets.
- Number of developers who have been selected as a reviewer in at least one event where they have also been recommended.

Comparing these numbers adds to the understanding of the difference in behavior of the two recommendation models.

Results

Figure 5.9 presents the recommendation list sizes of the two models. The average count of recommendations from the *Recency+Size* model is 21% less than for the *Recency* model. A lower ratio of average lengths of non-empty recommendation lists between two models suggests that a higher rate of empty recommendations largely defines the difference.

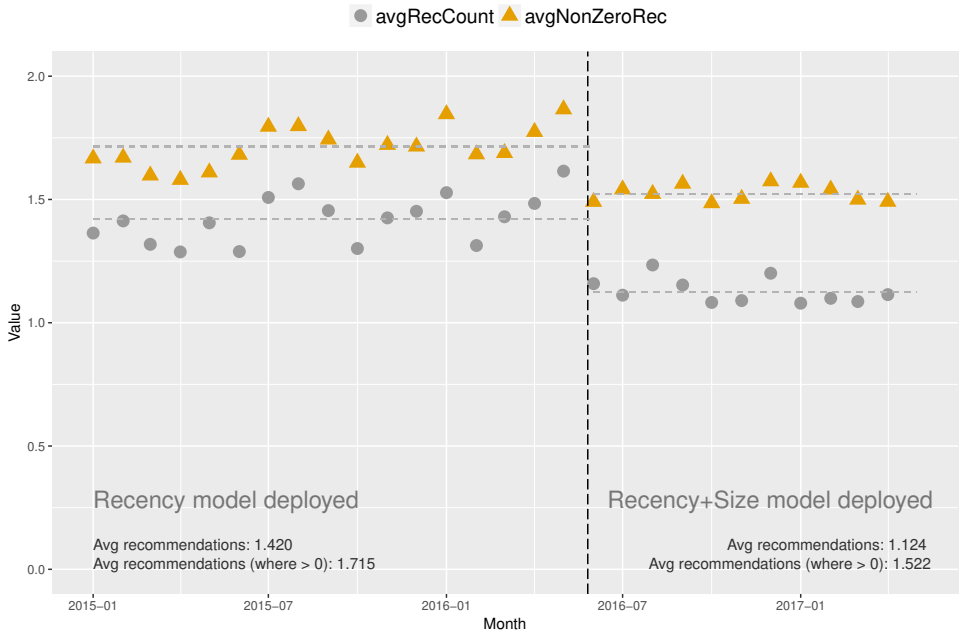
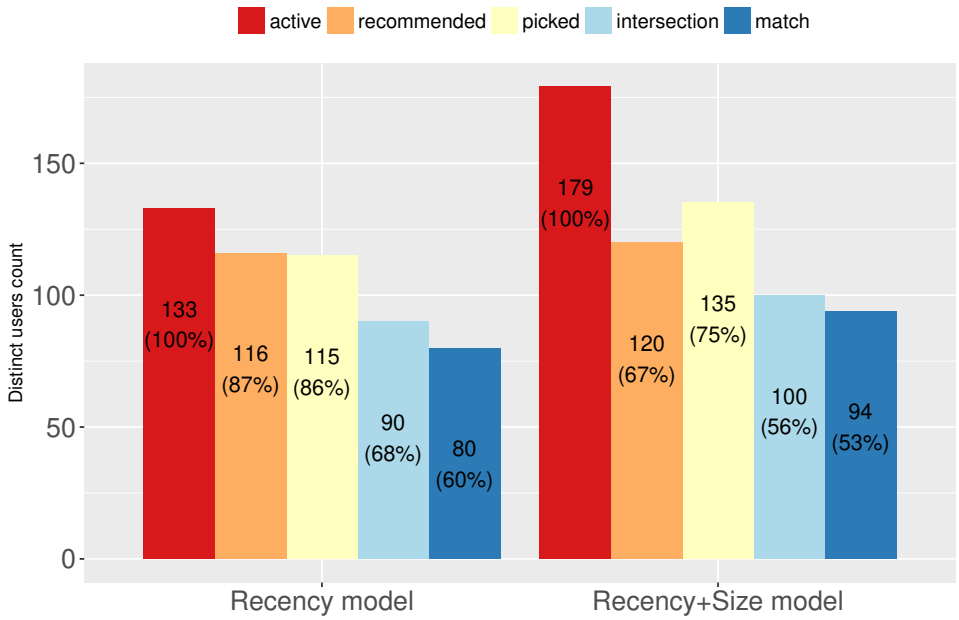


Figure 5.9: Average recommendation list size over 1-month periods

Figure 5.10 presents coverage of recommendation and selection of users, relative to all active users. The numbers demonstrate the value of this metric in addition to accuracy metrics: with analysis based strictly on accuracy metrics, the lower value of recall that *Recency+Size* model demonstrates, along with only marginal change in precision, would be interpreted as degraded performance compared to *Recency* model. However, the higher "intersection/recommended" and "match/recommended" ratios that *Recency+Size* model demonstrates despite a lower "picked/active" ratio, suggest that the smaller subset of active users whom *Recency+Size* model presents as recommendations, appears more relevant to the users, making the recommendations of *Recency+Size* model more likely to be followed.

5.4.6 RQ1 - Summary

The deployed models' accuracy is in line with existing results obtained through offline evaluation. The models are slightly different in terms of accuracy metrics. *Recency+Size*



"active": number of developers who made code changes in and before the corresponding period;
"recommended": number of developers who were recommended as reviewers at least once during the period;
"picked": number of developers who were picked as reviewers at least once during the period;
"intersection": size of the intersection of "recommended" and "picked" sets;
"match": number of developers who were selected in at least one event where they were also recommended.

Figure 5.10: Recommendations coverage for the two periods.

model on average gives less recommendations and reaches a slightly lower average recall. There are no noticeable changes in precision, evaluated for one model, at the moment of the deployed model change. A possible reason for lack of this effect is lack of influence of recommendations on choices of users, which contradicts our expectations about a deployed reviewer recommendation model. To shed light on how the users in our considered setting perceive the recommendations, we turn to a qualitative investigation of this aspect, which we describe in the next section.

5.5 RQ2: Developers' perception and use of reviewer recommendations

We dedicate our second research question to understanding the perception of relevance and helpfulness of recommendations by developers. To do so, we turn to the developers with interviews and surveys.

5.5.1 Data Collection and Analysis

First, we conducted semi-structured interviews at JetBrains with four developers who routinely use the recommender system. To further explore preliminary themes that emerged during the interviews, we ran an online survey among JetBrains developers. Finally, we sent another, large-scale online survey—augmented with questions addressing the themes that emerged at JetBrains—to developers at Microsoft who perform code review and possibly use the available reviewer recommender system.

Interviews. We conducted a series of online one-to-one interviews with professional developers at JetBrains, each taking approximately 20 minutes. To select participants, we focused on developers from the IntelliJ IDEA team, whose review activity was the subject for quantitative investigation in RQ1. The first author of this paper, who used to work at JetBrains before conducting this work, reached out to several developers from his past professional network. To mitigate the risk of *moderator acceptance bias* [243], the author selected only developers who provided him with frank feedback on his work at the company on past occasions.

The same author conducted the interviews [244] in a *semi-structured* form. This form of interviews makes use of an *interview guide* that contains general groupings of topics and questions rather than a pre-determined fixed set and order of questions. Such interviews are often used in an exploratory context to "*find out what is happening [and] to seek new insights*" [245]. The guideline was initially based on the main topics that emerged from the analysis of the recommender system's behavior; then it was iteratively refined after each interview. With consent, we recorded the audio, assuring the participants of anonymity. Since the interviewing author had both backgrounds in software development and practices at JetBrains, he could quickly understand the context and focus on the relevant topics, thus reducing the length of the interviews. We have transcribed the audio recording of each interview for subsequent analysis.

After the first four developers, the interviews started reaching a *saturation* point [246]: interviewees were providing insights very similar to the earlier ones. For this, we decided to design the first survey that we ran at JetBrains.

Surveys. The first survey, deployed at JetBrains, was aimed at further exploring the concepts emerged from the data analysis and the interviews. We sent the first survey to all 62 developers working on the product for which we collected the quantitative data at JetBrains. All these developers are using Upsource and are exposed to the recommendation system. This was a short, 5-minute survey, comprising 5 demographic information questions and 4 open-ended questions intermingled with 5 multiple choice or Likert scale questions.[3] The questions were focused on perceived relevance and usefulness of the recommendations. We received 16 valid answers (26% response rate).

The second survey, deployed at Microsoft, was aimed at validating and generalizing our conclusions, by reaching a large number of respondents working in different teams, products, and contexts. For the design of the surveys, we followed Kitchenham and Pfleeger's guidelines for personal opinion surveys [247]. Both surveys were anonymous to increase response rates [248]. The second survey was split into 4 sections: (1) demographic information about the respondent, (2) demographic information about the current team

[3] Available at https://doi.org/10.5281/zenodo.1404755

of the respondent, multiple choice and Likert scale questions (intermingled with open-ended fields for optional elaboration on the answers) on (3) reviewer selection and (4) relevance as well as helpfulness of the reviewer recommendation. Excluding demographic questions, the second survey consisted of 4 Likert scale questions with several items to scale (complemented by one or two optional open-ended responses) and 3 open-ended questions. The survey could be completed in less than 15 minutes.

We have sent the second survey to 2,500 randomly chosen developers who sign off on least three code reviews per week on average. We used the time frame of January 1, 2017 to August 1, 2017 to minimize the amount of organizational churn during the time period and identify employees' activity in their current role and team. As we have found that incentives can increase participation [249], survey participants were entered into a raffle for four $25 gift cards. We received 507 valid answers (20% response rate). The response rates for both surveys are in line with other online surveys in software engineering, which have reported response rates ranging from 14% to 20% [250].

In the rest of this section, when quoting interviewees' responses, we refer to interviewees from JetBrains as (I#) and to respondents to the JetBrains survey as (S#).

5.5.2 RQ2.1 — Do developers need assistance with reviewer selection?

In the interviews, we have asked developers at JetBrains about their criteria of reviewer selection. The answers indicate that the primary characteristic of the desired reviewers is their familiarity with the context of change: *"[desired reviewer] is the person who usually works with this [changed] part"(I1)*, *"the person who wrote a lot of code in the subsystem I am changing, or has recently been "digging" into this subsystem – there are fresh non-trivial changes by them"(I3)*. Along the responses, interviewees refer to the codebase as divided between the developers, each responsible for their subsystem: *"someone else's subsystem and [...] it's not my subsystem", "most of the work I do is in my subsystems"(I2), "there is the part of the codebase that I'm responsible for"(I1)*. This detail suggests the presence of strong code ownership practices at JetBrains.

The answers also indicate that the respondents are usually well aware of who is responsible for the code they are changing, thus often knowing the reviewers in advance. Developers say: *"I always know who will be reviewing my changes because I know which subsystem I'm changing and who owns this part"(I1)*, *"it is "from the experience" established who does [code review for my changes]"(I2)*, *"Almost always I know [the future reviewer]"(I3)*.

Results of the survey at JetBrains (presented in Figure 5.11) confirm this point: To the question "How often do you know whom to pick as a reviewer before even creating a review?" 63% of developers replied that they "Always" know the future reviewer, and 31% answered "Usually (90%)". The one remaining response was "Frequently (70%)". The Microsoft survey reveals a similar picture that is only slightly less extreme: 92% of respondents at Microsoft reported that they "Always", "Usually", or "Frequently" knew whom to pick as a reviewer before creating a review.

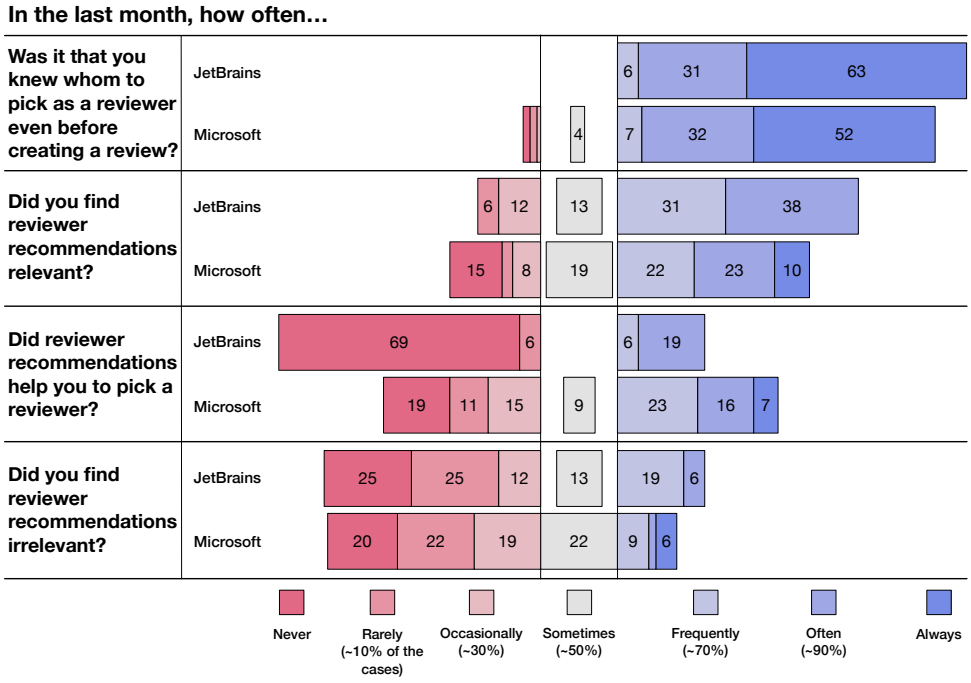


Figure 5.11: Distributions of answers to Likert scale questions about relevance and helpfulness of recommendations from Microsoft and JetBrains. Numbers represent the distribution of the responses in percent, rounded to nearest integer.

5.5.3 RQ2.2 — Are the reviewer recommendations perceived as relevant?

In the interviews, we have asked JetBrains developers whether they consider the recommendations given by Upsource relevant. The answers indicate that the recommendations are often relevant, and, moreover, it is usually clear why a particular person is recommended. *"in about 80% of the cases [recommendations] are relevant", "I understand why it suggests these or those people"(I3) "he's a code owner in many places, so that's why it happens"(I2).* On the other hand, all interviewees also report the scenarios of irrelevant recommendations. Such cases reportedly occur after changes in subsystem ownership, that are not yet widely reflected in changes history, or after bulk code modifications (such as API migrations), that have a short-term effect on the recommendation relevance score. *"there is a lot of code that's written by people who don't maintain that code anymore, so people who don't even work in the project anymore are sometimes recommended"(I2), "sometimes it happens so that it's absolutely not the right fit – say, someone who left for a different team", "[apart from the 80% of relevant recommendations] the rest — 20% — are [events] touching old code or code with no "bright" owner."(I3).*

In the JetBrains survey, to a question "How often do you find recommendations by Upsource relevant?", 6 users replied "Usually (90%)", 5 replied "Frequently (70%)", with the rest of answers spread between "Occasionally (30%)" and "Sometimes (50%)". The

answers to an open-ended question, that invited the respondents to elaborate on irrelevant recommendations, are confirming that users perceive the recommender as not capable of quickly capturing changes in subsystem ownership, which is indeed its Achilles' heel: *"suggested people sometimes don't even work for the team (or in this subsystem)"(S3)*, *"sometimes suggested person does not work on this code anymore"(S10)*, *"person is not actively responsible for subsystem anymore"(S16)*.

The responses from Microsoft, presented in Figure 5.11, are well aligned with these numbers: Most respondents find the recommendations mostly relevant. However, only 10% of respondents find them relevant in all cases.

5.5.4 RQ2.3 — Do the recommendations help with reviewer selection?

Given that developers are often already familiar with the information that the recommender provides, it is essential to assess the added value of the recommendation system. In the interviews, we have asked whether the recommendation feature was perceived as useful by the developers. Interviewees report that, in some cases, even while being relevant, recommendations did not provide any useful information. *"When I open Upsource, I know who I'll assign, so Upsource doesn't really help me to choose"(I3)*, *"I don't change anything in other parts of codebase — [other subsystems]. If I would change code there, I wouldn't know whose code it is, and would rely on Upsource. But I don't!"(I1)*. The interviews bring another important aspect to the surface: because some of the recommendations are irrelevant, it is harder to rely on the recommender in general. *"It is not very useful. Moreover, it [is wrong] often."*, *"It is a useful feature [in general], but in my case it [not always] worked correctly"(I1)*.

Some interviewees mention a scenario when the recommendation is useful, even though they already know the reviewer: when the recommendation matches their intention, they would add the desired reviewer in one click from the recommendations popup (Figure 5.2), instead of using the search form. *"Sometimes I use [recommendations]. Usually I know who will review, so I just click on their icon not to make multiple clicks and search"(I1)*, *"the most convenient part is when the suggested person is already "kind of" in reviewers list and you just have to click them, I think it's cool"(I2)*, *"Upsource doesn't really help me to choose, but it helps me to click! [...] instead of looking for a user, just click the suggestion and you're done — that's how it's convenient."(I3)*.

In the survey at JetBrains, 69% of respondents reported that reviewer recommendation "Never" helps them to find a reviewer. On the other hand, 19% of respondents reported that it "Always" helped. This polarity in perceived helpfulness may be attributed to the ambiguity of question: the "shortcut" scenario may not be considered a case of helpful recommendation by some of the respondents. One respondent to the survey explicitly mentioned the "shortcut" case: *"I use it as a quick way of adding a person I have in mind when it's on the list. If it isn't, I just ignore suggestions and use [search]."(S4)*.

The responses on usefulness from Microsoft (Figure 5.11) are more smoothly distributed across the frequency scale, with a slight incline towards lower frequencies: The recommendations are "never" or "rarely" helpful for 30% of respondents, 23% find them "usually" or "always" helpful.

5.5.5 RQ2 - Summary

This research question was dedicated to a qualitative investigation of how software developers in the two considered commercial software companies perceive and use code reviewer recommendations. The results of the investigation indicate that developers at both companies very often do not experience problems with reviewer selection. In fact, the vast majority of respondents at both Microsoft and JetBrains report to usually know the future reviewer even before creating a review for new changes. Most survey respondents (56% at Microsoft and 69% at JetBrains) find the recommendations more often relevant than not. However, reviewer recommendation features for reviewer selection were reported to be more often helpful than not by only 46% of respondents at Microsoft and by only 25% at JetBrains. These results suggest that—in the setting of the two considered commercial software companies—reviewer recommendation features in their current state are not perceived as essential for code review tools, since most of the developers usually do not experience any valuable support from recommenders.

The results also call for a deeper investigation on information needs of developers during reviewer selection, to identify other data sources that may be more helpful for future recommendation approaches. The results from RQ2 also call to identify whether recommendations can provide more value and be more helpful in certain situations than on average. We set out to explore these aspects in RQ3. We describe the methodology and results in Section 5.6.

5.6 RQ3: Information needs during reviewer selection

Our third research question aims to better understand the reviewer selection process by figuring developers' information needs when selecting reviewers for a code change.

5.6.1 Data Collection and Analysis

We have dedicated a part of the survey at Microsoft to questions regarding information needs of developers during the selection of reviewers. For each of 13 different kinds and sources of information (defined by two authors through brainstorming based on interview transcripts, existing studies on reviewer recommendation, and general knowledge of modern software development environment), we have asked developers (1) whether they consider it when selecting reviewers; (2) whether they find it relevant for selection of reviewers; and (3) how difficult it is for them to obtain when selecting a reviewer.

To identify other information needs of developers beyond the fixed list, we have included three open-ended questions in the Microsoft survey:

- Please describe other information that you consider when selecting reviewers.
- If there is information that you would like to consider that you aren't able to obtain, please tell us what information you would like.
- When do you find it most difficult to determine the right reviewers for a changeset that you send out for review?

To structure and quantify the answers to open-ended questions, we have used iterative *content analysis sessions* [251]. In the first iteration, the author of this thesis and his collaborator have independently assigned one to three tags to each of the answers. The tags were derived from the answers during the process. After the first iteration, through comparison of the sets of tags and discussion, the researchers agreed on the set of categories and fine-grained tags for the final iteration. Finally, the first author repeated the tagging with the new tags for all answers. The collaborator repeated the process on a random sample of answers for each of the questions (in total for 149 of 617 answers, or 24%).

To estimate consistency of tagging between the researchers, we calculated Cohen's Kappa [252] as a measure of inter-rater agreement. Keeping in mind that the number of tags per response could differ, we only took into account the first tag each author marked the response with, as it represents the strongest point in the response. For the fine-grained tags, the Kappa values for the samples are 0.852, 0.747 and 0.814 for respective questions. Calculating agreement measure for categories of tags—thus allowing fine-grained tags to differ if categories match—yields even higher Kappa values of 0.916, 0.851 and 0.859, respectively. Thus, agreement of researchers about tag categories can be interpreted as "almost perfect" according to Cohen, and only for fine-grained tags in one of the questions as slightly lower "substantial." As we do not make any quantitative statements based on exact proportions of different tags in responses, but only rely on these numbers for understanding needs and concerns of developers, a high degree of agreement between the researchers suggests that the results of tagging are strong and reliable.

5.6.2 RQ3.1 — What kinds of information do developers consider when selecting reviewers?

The answers to the multiple choice question are presented in Figure 5.12. The most important factor considered by developers during the selection of reviewers is the involvement of the candidate reviewer with the code under change. Three related categories of information, from generic "the person is still involved with the code" through ownership of files to authorship of recent changes, are reported as considered at least sometimes by 82–91% of the respondents. Another important aspect is whether the potential reviewer works in the area dependent on the changed code: 79% of the respondents report considering it. Other factors that are often (yet slightly less) considered include: the history of past reviews; working on code that the changed code depends on; opting in for reviews in the code area; availability of the person; prior review requests for code under change; and swiftness of response to code review requests. These factors are considered by 56–69% of the respondents. The three least popular categories are: working in directory surrounding files in change (38%); physical proximity of workplace (34%); and, surprisingly reported as irrelevant by nearly half of the respondents, current activity and load level of potential reviewers, which is taken into account at least sometimes by only 32% of the respondents.

The second question about considered information is open-ended, inviting the respondents to describe other information that they consider in free form. The answers were processed with iterative tagging (described in Section 5.6.1). The categories and tags of the answers are presented in Table 5.1. While being invited to describe *other* sources of information in relation to the previous question with predefined options, many of the responses correspond to one of these options. We did not filter out such responses. Respondents

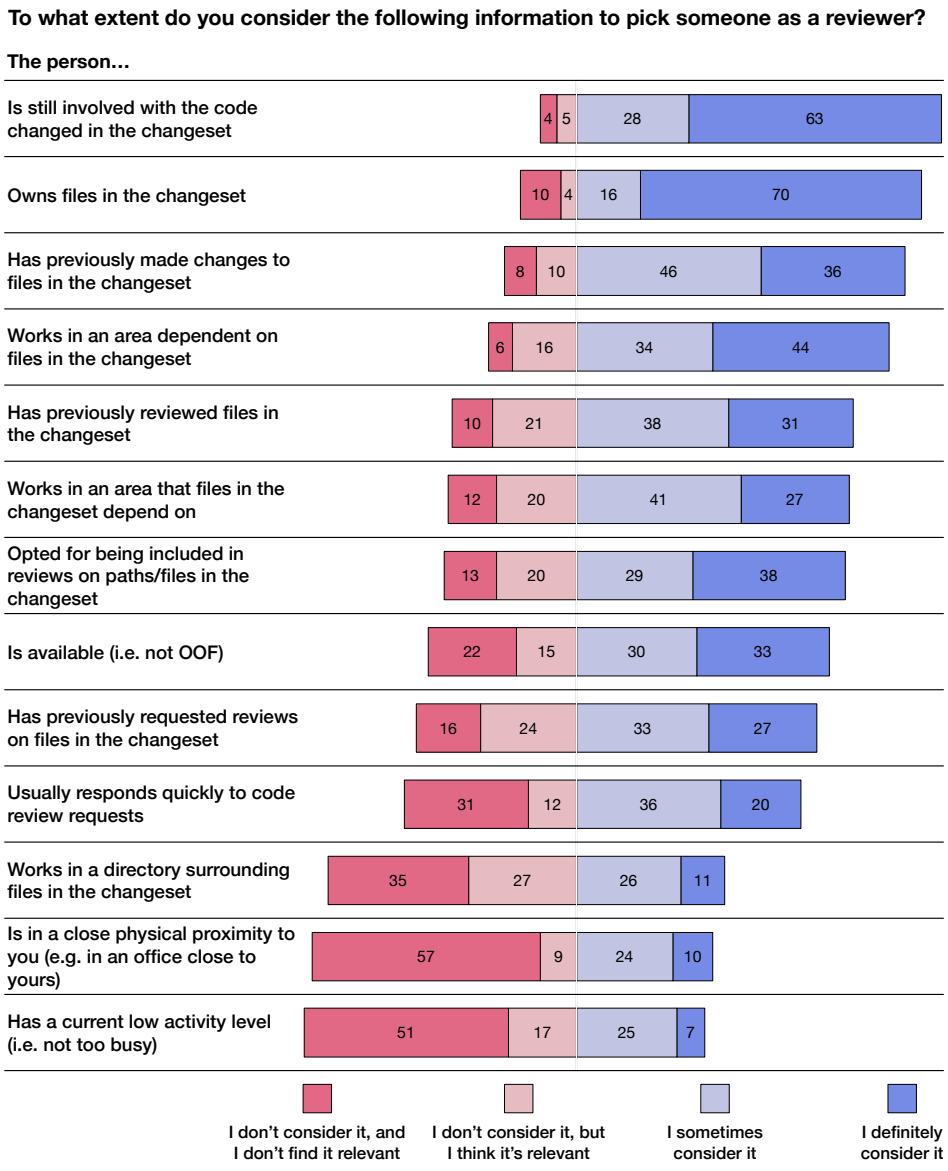


Figure 5.12: Information considered during reviewer selection, as reported by Microsoft developers. Numbers represent the distribution of the responses in percent, rounded to the nearest integer.

often make several different points in one response. Such responses are impossible to put in a single category, so we assigned 1 to 3 category tags to each response. In total, we have tagged 270 valid responses with 373 tags, yielding 1.38 tags per answer, on average. We report both the relative frequency of tags and the fraction of responses marked with a tag, while giving priority to the latter when explaining and discussing the results. We refer to

individual responses by their ID, e.g. *(#329)*.

Knowledge. Almost 30% of responses indicate the importance of potential reviewer's *knowledge* of one kind or another. These responses refer to knowledge of the area of code *"Person has expertise in the area, so can understand the algorithm changes. I work in [an area] where a lot of background is usually needed"(#54)*, to high-level or general knowledge *"I'll seek out opinion of known smart folks for some changes."(#498)*, knowledge of context *"Experts who have solved similar problems."(#192)*, and specific technical knowledge *"Area of expertise. If person is good in SQL, those types of changes will be reviewed better by that person."(#406)*.

None. 25% of the respondents said that they do not consider any specific information. A common assignment strategy described in such responses is to broadcast a review request to the immediate team, or to another mailing list: *"I send [the review] to the whole team and whoever is available and sees the email/PR first completes it."(#158)*. 6 respondents of 270 reported that they follow some policy to select a reviewer: *"Team lead and two other team members are mandatory. Others are optional."(#480)*.

Seniority. 23% of the respondents mention seniority of a potential reviewer. The categories of reported selection strategies comprise of preferring a person higher in hierarchy *"manager of person who owns the file or recently did changes"(#445)*, a more experienced or skilled person *"I tend to select who I believe are better developers."(#358)*, selecting a less experienced person to provide a learning opportunity *"I sometimes include new members of the team to review my change and to learn from it."(#349)*, and delegating the selection of reviewer to a colleague, often a more senior one *"Sometimes I add based on suggestions from other code reviewers, or from my manager."(#26)*.

Stakeholder. 22% of responses describe the potential reviewer as a stakeholder of code. Some of these responses vaguely mention involvement with project or feature *"I consider who wrote the files i'm changing, and who is currently working with me on my project."(#382)*, or relation to change code *"People working on deliverables for the same slice."(#466)*. More concrete answers mention the authorship of recent changes to code under review *"Generally it's who I see in git blame in that area of the code I'm changing."(#279)*, requesting the change under current review *"The (internal) customer who requested the change being made."(#491)*, authoring the code or helping with the current change *"If that person was involved in any way during investigation of the problem or he/she was involved in developing this fix."(#164)*, and working with code that depends on the changed code *"people that depend on that change or are impacted."(#191)*.

Reviewer qualities. 18.5% of responses refer to qualities of the reviewer without mentioning their relation to code. The most important quality, mentioned by 34 respondents, is thoroughness of the reviewer, usually known from the track record of quality reviews from their side *"Mostly I look for who can provide the best feedback on the set of changes"(#273)*. Other qualities include availability *"If the change is simple, I try to load balance based on other work they are doing."(#261)* and swiftness of their responses *"I think area interest and responsiveness is most important. Sometimes people can be knowledgeable about an area but fail to respond in a timely way"(#356)*. 2 people mention the physical proximity of reviewer's workplace, and 2 more people prefer their changes reviewed by someone whom they see as a nice person *"Someone that i trust and that is not a jerk:)"(#13)*.

Ownership. Despite code ownership being mentioned in the multiple-option question, 12.6% of responses refer to this concept at different levels. Most of such responses refer to ownership of area under change *"Ownership of the area if you know is helpful in deciding whom to add for review."(#152).* A few other respondents mention ownership of feature *"Its mostly the person who is owning the functionality"(#325)*, predefined ownership of a file or component *"We have owners.txt files in all of our service repositories which identify a base set of reviewers."(#503).* Several responses also mention ownership of service or a repository.

In total, nearly 75% of developers at Microsoft reported relying on specific kinds of information when selecting a reviewer for their changes. Various types of information serve to ensure three distinct properties of the potential reviewer: they are qualified to review this change (Knowledge and Seniority categories), are interested in reviewing the changes (Stakeholder and Ownership), and are capable of providing a quality review (Reviewer Qualities).

5.6.3 RQ3.2 — How difficult to obtain are the different kinds of information needed for reviewer selection?

Similarly to the previous research question, which investigated different kinds of information that developers rely on when selecting reviewers, ease of access to different kinds of information was targeted by two questions in the survey at Microsoft: a Likert scale question, which invited respondents to rate each of 13 categories of information by ease of access during reviewer selection, and an open-ended question, which invited respondents to describe what information they miss when selecting reviewers.

Figure 5.13 presents the aggregated answers to the Likert scale question. The information that is reported as the most difficult to obtain is the history of past reviews of the files in the changeset, both in terms of performed reviews (reported as difficult to some extent by 48% of the respondents) and of review requests (47%). However, responses display notable diversity – these types of information were classified as easy to obtain by other 38% and 31% of the respondents, accordingly. Another kind of information that is often reported as difficult to obtain is the connection of potential reviewer's area to the code in the changeset in terms of codebase proximity or dependency, either as a consumer or a producer. This information is reported as difficult to obtain by 43–46% of the respondents, and as easy by other 30–33%. Other kinds of information, including code ownership, involvement with code under change, and personal qualities of reviewers, are rather easy to obtain for most of the respondents. Notably, history of changes is hard to obtain for only 11% of the respondents, while reviews history is ranked as the hardest. This inequality might be caused by imbalance in tool support for retrieval of histories for changes and reviews — open text responses about considered information often mention usage of *git blame* to identify reviewers, with no similar tool existing for the history of reviews.

The second, open-ended question was inviting developers to name other kinds of information that they would like to consider, but are not able to obtain during reviewer selection. We received 74 valid answers and the results are presented in Table 5.2.

Past contributions. One in three developers (25 of 74) reported missing information about past contributions. History of past changes demonstrates the highest demand with 25% respondents mentioning it. *"File history from before the review started (just the git commit comments would be great)"(#45).* Some respondents also mentioned missing information

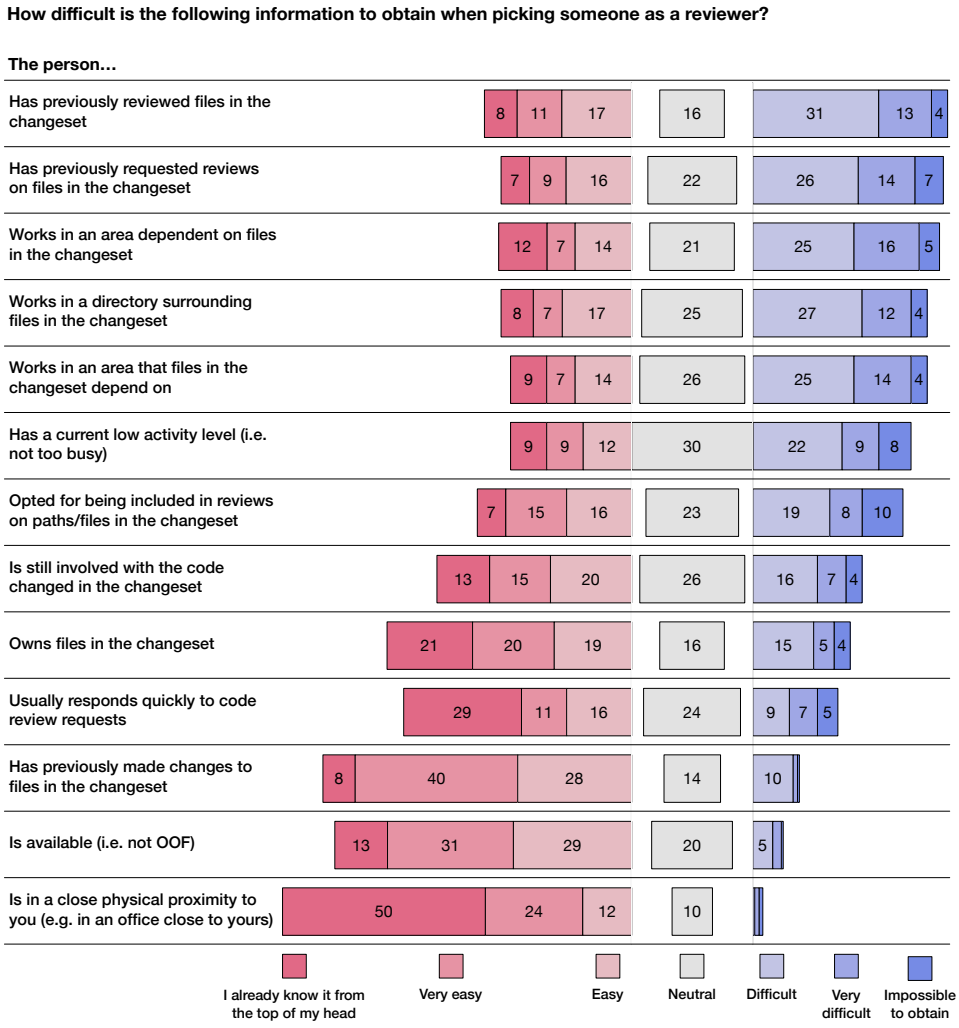


Figure 5.13: Difficulty in obtaining different kinds of information during reviewer selection, as reported by Microsoft developers. Numbers represent the distribution of the responses in percent, rounded to nearest integer.

about the history of reviews.

Reviewer qualities. 32% of the respondents miss information about reviewer qualities, with the most critical aspect being their thoroughness and helpfulness *"It would be nice to get some sense of whether other developers consider a reviewer's input valuable"(#120).* This result is in line with the results of the question about considered information (RQ 3.1), where reviewer thoroughness is as well reported as the most important quality. Multiple respondents would also like to be aware of reviewers opting in for code areas *"I really want to be able to opt in to say, "I want to be included on any review that touches a specific tree/directory/file""(#260)*, reviewer availability and their swiftness to respond *"how many*

other reviews are pending them and how quickly they tend to turn around code reviews."(#321).

Ownership. 28% of the respondents report missing ownership information, with 4 people explicitly mentioning ownership data derived from changes history *"Owners of past changes related to the change set (most important to consider) is tedious to get even though history does help. If there an auto selection of such owners, that would be great."(#175)*, and two respondents mentioning ownership information in context of selecting a reviewer for legacy code *"File ownership is very nebulous. [...] many of those developers have moved onto other teams and there are enough reorg's that it's not clear who is responsible for the code. The hardest part of that information to obtain is transfer of ownership when people move on (be it leave company or team, or reorg to a new project)"(#262).*

Knowledge. 9 of 74 respondents (12%) mentioned missing information about different aspects of knowledge *"I would like to include people who have significant knowledge about the code being changed, but that's not always easy to figure out."(#179).*

Dependency. 5 responses (7%) mention information about code dependencies, both *of* changed code and *on* it. *"who are the dependent people working on the same or dependent files at that time"(#473)*; *"I would like to know all authors who have touched the code on which the changeset depends or code that depends on the changeset – automatically."(#138)*

5

5.6.4 RQ3.3 — When is it more difficult to choose a reviewer?

We identified the more challenging scenarios for reviewer selection with an open-ended question in Microsoft survey. The results are presented in Table 5.3.

Uncommon context. Over a half of the respondents (146 out of 273) reported having a harder time selecting reviewers for changes done in unusual contexts. 92 of these responses (34%) mention changes in areas outside of typical work scope. *"When working in feature areas that you do not own, that your team does not own, and that you usually don't spend time in"(#235).* 40 respondents (15%) specifically mention having issues with selecting reviewers for legacy code: *"When the code is in a "legacy" area with prior contributors or owners who have changed roles or no longer work at the company. Some code is not actively "owned" at all [...]"(#40).* 31 respondents (11%) report problems with selecting reviewers as newcomers to a new team or code area. *"When I am new to a team and have not yet built up a mental map of area experts and dependencies between feature areas."(#46).*

Special ownership. 100 of 273 responses (37%) specifically mention special cases of code ownership distribution as a trigger for more difficult reviewer selection. 47 responses (17%) describe situations when ownership of code under the change is unclear due to technical issues or transfer of component ownership between teams: *"When the code being changed is old and ownership has changed many times since"(#206).* 21 responses (8%) mention parts of code shared between teams as a challenging scope to find a reviewer for: *"When I make a change on a unusual part of the codebase that is shared with other teams."(#465)*; *"There are multiple shared files, which gets altered by multiple folks(like app initialization order ...). It is very difficult to find out who is the prime owner of these files."(#436).* 20 respondents (7%) reported having issues with reviewer selection when developing a new feature or component from scratch: *"When it is a new logic or code being started and not many in the team are aware of the design and process."(#423).* 12 respondents mentioned difficulties finding a reviewer for changes in code that only they are familiar with: *"When*

its a component that I pretty much exclusively own [...] People don't have the bandwidth to deeply learn a new component for the purposes of code review."(#265).

Reviewer availability. 41 of 273 respondents (15%) mentioned issues related to the availability of a potential reviewer as factors for a more challenging selection process. 19 respondents (7%) describe situations when the potential reviewer is not available at the moment, if they are very busy or are on vacation: *"When the only other backend developer on the project is out on vacation or otherwise priority tasked on something different."(#22).* 14 respondents (5%) report a harder choice of reviewer when the person typically responsible for a specific piece of the codebase has left the team. *"When all the people who worked on the code/reviewed it in the past have left the team/company"(#77).* 5 people (2%) reported situations when no one was willing to review code: *"When no one else wants to take ownership of the review. But usually some does or the team takes a call on who it should be."(#116).* 3 of 273 respondents mentioned that it is harder to select a reviewer when they do not know the candidates personally: *"If I haven't personally met everyone on the dev team yet"(#147).*

Uncommon impact. 22 respondents (8%) mention unusually high potential impact of their changes, due to presence of dependent code, as a challenging factor for selection: *"when changing files in public APIs that are consumed by external teams, its difficult to know all the scenarios and the persons to cover for the code changes."(#125)*; *"when I make changes to code that many people depend on, I send the review to everyone."(#138).*

Uncommon content. Only 18 of 273 respondents (7%) reported cases in which the content of the change makes it more difficult to find a reviewer. 10 respondents (4%) mentioned that the choice is more challenging if the review of the change requires specific expertise that few of the colleagues possess: *"When I need an expert to figure out if I'm doing the right thing and they aren't among the folks I know of"(#504).* 6 respondents (2%) talk about unusually large changes. *"When the change set is large in terms of number of files or amount of change"(#175)* 2 respondents (1%) said it is more difficult to find a reviewer for changes with complex logic: *"Usually when a change is extremely complex."(#190).*

5.6.5 RQ3 - Summary

With this research question, we explored the types of information that software developers need and rely on when selecting reviewers for their changeset.

In the first part (Section 5.6.2), we asked developers about information that they consider for selection of reviewers. The most commonly considered information is involvement of potential reviewers with code under the changeset when selecting reviewers. Such involvement, defined either through formal ownership of code or by regular changes and reviews of the code, is often considered during reviewer selection by the vast majority (69 – 91%) of the respondents. Such high demand suggests that the standard mechanism of reviewer recommendation approaches (*i.e.,* the identification of involved developers through an analysis of the history stored in software repositories) is indeed targeted to maximize the most important qualities of potential reviewers. Many developers also reported relying on other kinds of information, such as code dependencies and availability of reviewers, which are not yet considered by existing approaches to reviewer recommendation. The open-ended question revealed more categories of relevant information for reviewer selection, such as seniority and personal qualities of potential reviewers.

The second part of this research question (Section 5.6.3) was dedicated to the difficulty in obtaining the different kinds of information. The most difficult information to obtain is related to the history of reviews and code dependency: 43 – 48% of developers find this information rather difficult to obtain. The easiest information is code authorship history, availability of colleagues, and physical proximity of their workplaces. Responding to the open-ended question regarding the information developers would like to consider but are not able to obtain, respondents reported missing history of past contributions, information on reviewer personal qualities, knowledge, and code ownership.

In the third part (Section 5.6.4), we analyzed the most difficult situations for selection of reviewers, as reported by developers in open text responses. The most prominent category of responses mentions reviews for changes in an unknown codebase, including modifying legacy code and being new to a team. Other difficult scenarios include changes in code with an unclear owner, development of new code from scratch, changing code with external dependencies, and situations when the usual reviewer is not available.

The insights from this research question indicate directions for improvement of future reviewer recommendation algorithms. We discuss these results in Section 5.7.3.

Table 5.1: Other information considered during reviewer selection, as reported by Microsoft developers in responses. Data from responses to an open question. Counts of tags and tagged responses are reported separately: each valid response was assigned one to three tags.

Please describe other information that you consider when selecting reviewers.

Category / Tag	Tags	% of all tags	Responses	% of all responses
Knowledge	**85**	**22.8%**	**80**	**29.6%**
Knowledge of area	42	11.3%	42	15.6%
High-level knowledge	17	4.6%	17	6.3%
Knowledge of context	13	3.5%	13	4.8%
Knowledge of technology	13	3.5%	13	4.8%
None	**69**	**18.5%**	**68**	**25.2%**
Send request to team	50	13.4%	50	18.5%
Broadcast	13	3.5%	13	4.8%
Policy	6	1.6%	6	2.2%
Seniority	**66**	**17.7%**	**62**	**23.0%**
Reviewer is higher in hierarchy	24	6.4%	24	8.9%
Reviewer is more experienced	23	6.2%	23	8.5%
Choice is delegated to someone else	10	2.7%	10	3.7%
Reviewer is less experienced	9	2.4%	9	3.3%
Stakeholder	**63**	**16.9%**	**60**	**22.2%**
Person is involved in project	16	4.3%	16	5.9%
Person recently made changes to code	14	3.8%	14	5.2%
Person is related to change code	11	2.9%	11	4.1%
Person requested this change	8	2.1%	8	3.0%
Person authored or helped with change	7	1.9%	7	2.6%
Person's area depends on this code	7	1.9%	7	2.6%
Reviewer qualities	**54**	**14.5%**	**50**	**18.5%**
Reviewer is thorough	34	9.1%	34	12.6%
Reviewer is available	9	2.4%	9	3.3%
Reviewer is swift	7	1.9%	7	2.6%
Reviewer is close	2	0.5%	2	0.7%
Reviewer is nice	2	0.5%	2	0.7%
Ownership	**36**	**9.7%**	**34**	**12.6%**
Area ownership	18	4.8%	18	6.7%
Feature ownership	8	2.1%	8	3.0%
Predefined ownership	7	1.9%	7	2.6%
Service ownership	2	0.5%	2	0.7%
Repository ownership	1	0.3%	1	0.4%
Total	**373**		**270**	

Table 5.2: Information missing during reviewer selection, as reported by Microsoft developers in responses to open questions. Counts of tags and tagged responses are reported separately: each response was assigned one to three tags.

Is there information that you would like to consider when selecting reviewers, but are not able to obtain?

Category / Tag	Tags	% of all tags	Responses	% of all responses
Past contributions	**27**	**30.0%**	**25**	**33.8%**
Past authors of changes to code	18	20.0%	18	24.3%
Past reviewers of code	7	7.8%	7	9.5%
People requesting reviews to code before	2	2.2%	2	2.7%
Reviewer qualities	**26**	**28.9%**	**24**	**32.4%**
Reviewer thoroughness and helpfulness	9	10.0%	9	12.2%
Availability or workload of reviewer	6	6.7%	6	8.1%
Who opted in or is responsible for reviews	5	5.6%	5	6.8%
Review speed	4	4.4%	4	5.4%
Access of reviewer to code	1	1.1%	1	1.4%
Interest or willingness to review	1	1.1%	1	1.4%
Ownership	**21**	**23.3%**	**21**	**28.4%**
General code ownership information	15	16.7%	15	20.3%
Ownership derived from changes history	4	4.4%	4	5.4%
Ownership of legacy code	2	2.2%	2	2.7%
Knowledge	**10**	**11.1%**	**9**	**12.2%**
Knowledge of area	5	5.6%	5	6.8%
Knowledge of code itself	3	3.3%	3	4.1%
Knowledge of technology	1	1.1%	1	1.4%
Knowledge of type of code	1	1.1%	1	1.4%
Dependency	**6**	**6.7%**	**5**	**6.8%**
Dependencies on change (consumers)	4	4.4%	4	5.4%
Dependencies of change (producers)	2	2.2%	2	2.7%
Total	**90**		**74**	

Table 5.3: Most difficult situations to choose a reviewer, as reported by Microsoft developers in responses to open questions. Counts of tags and tagged responses are reported separately: each response was assigned one to three tags.

When do you find it most difficult to determine the right reviewers?

Category / Tag	Tags	% of all tags	Responses	% of all responses
Uncommon context	**163**	**47.4%**	**146**	**53.5%**
Not usual area	92	26.7%	92	33.7%
Legacy code	40	11.6%	40	14.7%
Newbie	31	9.0%	31	11.4%
Special ownership	**100**	**29.1%**	**100**	**36.6%**
Unclear owner	47	13.7%	47	17.2%
Shared code	21	6.1%	21	7.7%
Development from scratch	20	5.8%	20	7.3%
Self-owned code	12	3.5%	12	4.4%
Reviewer availability	**41**	**11.9%**	**41**	**15.0%**
Reviewer is not available	19	5.5%	19	7.0%
Reviewer left the team	14	4.1%	14	5.1%
No one willing to review	5	1.5%	5	1.8%
Author does not know the reviewer	3	0.9%	3	1.1%
Uncommon impact	**22**	**6.4%**	**22**	8.1%
Dependent code	22	6.4%	22	8.1%
Uncommon content	**18**	**5.2%**	**18**	**6.6%**
Code requires exotic expertise	10	2.9%	10	3.7%
Change is large	6	1.7%	6	2.2%
Change contains complex logic	2	0.6%	2	0.7%
Total	**344**		**273**	

5.7 Discussion

5.7.1 RQ1: Performance of a deployed reviewer recommender

The initial result of this study is the first ever evaluation of the performance of a reviewer recommender in action, as opposed to the customary approach of benchmarking isolated prototypes on historical data. Due to limitations imposed by the practical context, we were unable to use the commonwise metrics for recommender evaluation, such as top-k accuracy and MRR, as the primary measures of recommendation performance. Instead, we were looking at each individual event of reviewer addition, and aggregated adjusted precision, recall, and MRR over monthly periods.

The values of the accuracy metrics slightly change throughout the longitudinal data period, which can be attributed to the growth of the user base of Upsource at JetBrains. At the same time, comparison of coverage metrics (Figure 5.10) reveals that there is a substantial difference between the two recommendation models, which, however, does not result in a drastic change of precision values as the deployed model is changed (Figure 5.8). This supports the notion (increasingly popular in the Recommender Systems field) of the importance of metrics beyond accuracy for evaluating recommender systems, and demonstrates its applicability to the problem of reviewer recommendation. It also suggests that future efforts in reviewer recommendation should carefully consider the specific, real-world software engineering context, and not count out the potential effects specific to deployed recommenders.

One of such effects is the influence of recommendations on choices of users. While our attempt on identifying it was not straightforward from a methodological point of view (Section 5.4.3) and yielded a negative result, this result was the main inspiration for the other two research questions in this study. While reliably detecting such effects in practice is indeed a hard task due to a need of longitudinal monitoring and A/B testing, our example demonstrates that, in some cases, key insights are possible to gain without expensive experiments.

5.7.2 RQ2: Perception of the recommender by users

In the second research question, we investigated the perception of relevance and usefulness of recommendations by collecting user feedback in two different commercial environments. Developers generally perceive recommendations as relevant. However, developers report that recommendations are not always helpful. It is explained by the fact that developers report to quite often know the future to select reviewer in advance.

Evidence of this imbalance is, in our opinion, one of the most important outcomes of this study for researchers. Quite a few studies focus on building new approaches for reviewer recommendation. Researchers strive to improve accuracy over existing algorithms, and in recent work, their efforts go beyond straightforward scoring techniques based on history to building expertise models, which involves more sophisticated methodology [191]. Existing studies on reviewer recommendation argue that tool assistance during the stage of reviewer selection can improve the efficiency of code review process, which implies that a reviewer recommender is a valuable tool in practical contexts. For example, Thongtanunam *et al.* [9] found that "[in selected open source projects] 4%–30% of reviews have code-reviewer assignment problem" and concluded: "A code-reviewer recommendation tool is necessary

in distributed software development to speed up a code review process". Our results suggest that code reviewer assignment is indeed problematic in certain contexts (Section 5.6.4) such as for developers not familiar with code under the change. It is therefore vital to first understand and study the context of application of the recommendation, and then select the appropriate set of evaluation measurements that align with that context, in order to develop helpful recommendation algorithms.

Industrial code review tools, like other work instruments, generally develop in a very pragmatic way by first offering support for actually existing issues. While recent adoption of reviewer recommendation features in several popular code review tools supports the notion of the importance of recommendation as a tool feature, our results strongly challenge the assumption that it is a universally valuable and helpful feature for the users. We believe that research efforts in reviewer recommendation would have a stronger practical impact if they focused on user experience rather than accuracy. A particularly important direction of future work would be to investigate the added value of a reviewer recommender in open source environments: different patterns of contribution frequency and degree of involvement with the project and the team could cause a recommender to be perceived differently from the company settings in our study.

5.7.3 RQ3: Information needs for reviewer selection

Results from our third research question provide insights on reviewer selection process, along with strong indications on the further design of data-driven support in review tools.

Information considered when selecting reviewers. The types of information most commonly taken into account during reviewer selection by developers at Microsoft (Figure 5.12, Table 5.1) are related to scopes of responsibility and recent contribution of developers, code ownership, and knowledge of code and involved technologies by individual contributors. Existing approaches to reviewer recommendation estimate the relevance of potential reviewers based on history of changes and prior reviews for files in the current changeset. The history is either used directly to identify prior authors and reviewers of changes similar to the current [69, 71, 190, 238], or as a basis for more complex methods to estimate reviewer relevance. Examples of such methods include using a search-based approach to identify the optimal set of expert reviewers [191], and extracting additional data, such as a social graph of prior developer interactions from comments in prior reviews [40], or records of developers' experience with technologies and libraries specific to the current changeset, from previous pull requests in the current project [18, 70]. Thus, the results from the survey confirm that prior approaches to reviewer recommendation are well aligned with the most prominent information needs of developers.

However, our results from RQ3 also indicate that, apart from knowledge and prior involvement with code under review, developers consider a more extensive range of factors including codebase dependency information, hierarchy in organizational structure, personal qualities of colleagues, and others. To be more helpful for users and for a broader set of users, future reviewer recommendation approaches could incorporate a broader spectrum of information, beyond histories of changes and reviews and data derived from these histories, into their underlying models as well. Examples of such information could include graphs of code dependencies within the project, records of organizational structure and workplace proximity from HR information systems, or traces of developer communication

beyond code review comments, such as emails and messengers.

Prior research [188] established that expectations and outcomes of code review process go beyond elimination of defects and codebase quality control in general. Developers and managers report that benefits of code review also include ensuring knowledge transfer and team awareness of changes in the codebase. Responses from our survey at Microsoft support this point. Some respondents mentioned that they sometimes look for less experienced peers as reviewers to provide a learning opportunity. Concerning awareness, some respondents also mentioned that they are looking for people using their code as a dependency to perform the review, thus ensuring their awareness of changes. Reviewer recommendation systems could promote knowledge transfer and team awareness by not solely focusing on finding developers who already are the most familiar with the code, but also promoting knowledge transfer by recommending less experienced people as reviewers. In our vision, this idea suggests a particularly interesting direction for future research.

We found that personal qualities and hierarchical position of potential reviewers are often considered important factors for the choice. The most important of these factors is the track record of quality reviews from an individual. This highlights the importance of personal qualities and reputation of the engineers for the collaborative activity of software engineering in teams.

Availability of information for reviewer selection. We found that developers find some kinds of information, that is required for reviewer selection, more difficult to obtain. These results suggest opportunities for meaningful tool support of this process. History of prior changes and reviews are reported among the most commonly considered information for reviewer selection (Figure 5.12). At the same time, options corresponding to the history of reviews are named as the hardest kinds of information to obtain, while change history is among the easiest (Figure 5.13). As respondents often directly mention using *git blame* to identify best reviewers, this imbalance in ease of access to information is likely to be caused by inequality in tool support for retrieval of historical data. Efforts of researchers and practitioners can be targeted at mitigating this inequality. Expanding this notion further, many of other difficult-to-obtain information types, such as dependency information, workload level of colleagues, swiftness of review activities of an individual, and proximity of their working area to a given change, can potentially be aggregated by a code review tool. While it might not be feasible to compile all these data into a single universal model for reviewer suitability, merely presenting some of this information to the user during reviewer selection will make it not necessary for users to acquire this information by themselves, thus being a valuable feature to improve the efficiency of code review process.

More difficult situations for reviewer selection. Our results reveal that in some situations it is harder for developers to find a suitable reviewer. Examples of such situations (Table 5.3) include making changes in code outside the normal work area, being new to the company or a team, and having external code depending on the area of change. An implication of this for researchers and developers is the possibility to tailor recommendation tools to users' needs and reduce noise by making the recommender only trigger when a given user needs assistance. In such situations, the user is more likely to find a recommendation helpful. Helpfulness of the recommendations in some of these scenarios, such as reviewing code with complex dependencies or large amounts of new code, could be related to confusion to some extent [253].

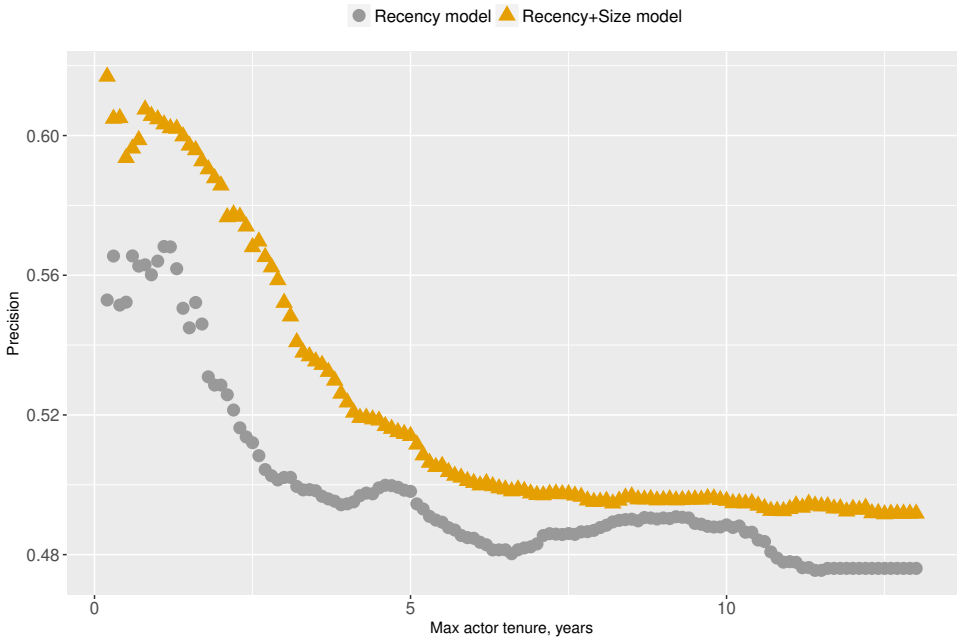


Figure 5.14: Precision of recommendations as function of maximum tenure of the developer receiving the reviewer recommendation (*i.e.,* the author of the changeset).

To support this point, we have attempted to discover the potential difference in perception of recommendations by new developers and by others through quantitative analysis. In the data that we used for RQ1, for each event we have additionally estimated the tenure of the receiver of the recommendation (*i.e.,* the author of the changeset), by calculating the time since the first trace of their activity in the historical data, be it a contribution to codebase or participation in a review.

For threshold values varying from zero to lifetime of the project, we have filtered out the events by receivers of the recommendation with tenure above the threshold and calculated average precision of the model across the whole longitudinal period for the rest of the events. Figure 5.14 presents the picture of precision as a function of maximum tenure of the event actor: for each time threshold, we discard the events where the receiver of the recommendation is more experienced than the threshold, and calculate the average precision across the remaining events. The chart shows that the precision is higher for actors with low experience. Considering that the model does not take characteristics of the receivers of the recommendation into account and that the picture is similar for both models, the effect of high precision for lower-experienced users might be explained by their higher likelihood to follow the recommendations.

Unfortunately, we cannot claim from this observation alone that the less tenured developers are more likely to follow the recommendations: developers report that the responsibility scopes of developers at JetBrains rarely intersect, so the difference in the precision might as well be rooted in the difference of the codebase that they work on. The

unstable precision trend, in the left part of Figure 5.14, suggests that variance in precision between individual actors is high: the trend stabilizes as events by more and more actors contribute to the overall picture. Moreover, a further analysis of the survey responses along with demographic data of the respondents revealed no connection between developers' tenure in the current team and perceived relevance or usefulness of the recommendations.

For these reasons, a stronger evidence of connection between actors' tenure and alignment of their choices with recommendations requires designing and running a controlled experiment to be able to control for the confounding factor of difference in their work scopes: we would need to observe many developers with different levels of experience interacting with recommendations given for the same code, which goes beyond the scope of this study where we only use the historical data but do not set up an experiment. Nevertheless, the trend in Figure 5.14 suggests that conducting such an experiment could be a fruitful direction for future work.

Triangulating the signal above on the potential connection between the experience of the actor and the empirical accuracy with the responses on the most difficult situations for choosing a reviewer (Section 5.6.4), it seems reasonable to conclude that existing recommendation models and evaluation metrics may be a good fit for scenarios in which a project receives a lot of contributions from external people, such as open source projects and large teams where newcomer onboarding is a frequent procedure. This reasoning would also be in line with the recommendation by Thongtanunam *et al.*, which are based on data from large open source projects [9].

5.7.4 Overview

The results from RQ1 indicate that the choices of reviewers are not strongly influenced by the recommendations. RQ2 reveals that the recommendations are commonly perceived as relevant, yet do not play a minor role for most of the developers during reviewer selection. With RQ3 we find that, while the most prominent information needs of developers during reviewer selection — namely, involvement and experience of the colleagues with code under review — are already targeted by the recommendation algorithms, there are other information needs that the future recommendation models could cover, such as information on code dependency and team hierarchy.

The mechanics of the current reviewer recommendation models are well aligned with the information most commonly considered by developers. In fact, the recommendations are perceived as relevant by the majority of the developers (RQ2.2). This means that the developers could benefit from the recommendations as a way to confirm their judgments with data. Moreover, some developers report that they find the recommendations widget useful to quickly add a reviewer they already have in mind (RQ2.3).

At the same time, the results of the surveys demonstrate that the recommendations are not helpful for many developers, as it is common for them to know the reviewers in advance. This calls for development of recommendation techniques that take the development context into account and try to satisfy the information needs of a broader range of developers during reviewer selection that are not covered by existing recommendation approaches, such as code dependency information and team hierarchy. Such more advanced recommenders could be helpful in the complex reviewer selection scenarios. Moreover, future recommenders could not only target the selection of the most relevant reviewers

for the given changes, but also optimize for other goals of code review, such as transfer of knowledge within teams and promotion of shared code ownership and responsibility.

5.8 Limitations and threats to validity

The scope of our analysis is limited to the two subject companies, which, due to the exceptional diversity of software engineering environments, cannot be considered representative of the whole range of practices. Despite the difference in scale of the companies, policies and organizational structure, and cultural origin of the respondents, results, where comparable, are highly consistent in the two subject companies.

However, it is essential to underline the role of the study context and its impact on the generalizability of our results beyond the considered development environments. Our results reveal that the conventional means of evaluation of reviewer recommendation systems do not completely align with the needs of developers in the commercial *company* settings that we studied. An average respondent to the survey at Microsoft works in a team of 12 people, and has been in the team for 2.25 years. Average tenure of a developer in our target team at JetBrains is 5 years. In community-driven open source projects, which accept external contributions, the results may be different, as may be the needs for reviewer recommendation. In fact, the demand for reviewer recommendation might be higher in the projects that often receive one-time or infrequent contributions from people who are less familiar with the codebase and the scopes of responsibility of fellow developers. Such developers may represent a significant number of contributors, compared to the group of core developers. We can expect only core members of open source communities to be familiar with the codebase to the degree that is typical for a developer in a commercial team, as the peripheral contributors commonly devote less time and effort to the project, thus having little chance to gain such familiarity. In larger commercial teams current reviewer recommender models might as well be perceived as a more valuable tool, as for a member of a large team it is harder to be aware of responsibility scopes of the colleagues than for someone whose team is small. In addition, larger teams more often have newcomers, who may especially benefit from the recommender during onboarding. Thus, in different contexts the misalignment between the evaluation techniques for reviewer recommendation and its value for users may be less pronounced, if at all. For these reasons, the value of reviewer recommendation in such contexts deserves a separate dedicated study. Moreover, codebases of some open source ecosystems consist of multiple interconnected projects. In such settings, optimal performance of recommendations might be ensured by other algorithms than in single-repository settings (for example, by algorithms that use history of multiple repositories to recommend reviewers for code in a given repository). We find this another interesting direction for future work.

The authors who tagged the open-ended answers work as researchers, but not as software engineers. Occupation might impose some bias on the interpretation of responses.

We used a weakly formalized method to identify feedback from recommendations on users. Thus, we cannot claim that no feedback is present, but only that it is too subtle to be detected without a controlled experiment; setting up such experiment would require a lot of resources. However, lack of evidence of feedback inspired the other two research questions.

5.9 Conclusion

In this study, we have explored multiple aspects of reviewer recommendation algorithms as features of code review tools. We have conducted the first *in vivo* performance evaluation of a reviewer recommender, explored the perception of relevance and helpfulness of recommendations by users, and investigated the information needs of developers in the process of reviewer selection, in a company setting. The results of this study suggest directions for the future evolution of reviewer recommendation approaches by bringing out the most common information needs of developers in two commercial teams. Our results also provide insights that are valuable in a broader context of the evaluation of data-driven features in developer tools. We further separate the two characteristics: accuracy of an isolated algorithm and its value for the users when deployed, and demonstrate that the two are, to an extent, misaligned in our setting. We interpret this misalignment as a signal of importance of selecting the evaluation techniques with the practical context in mind: in our setting, the common recommendation accuracy measures did not represent the value of the tool for users well. However, in other contexts these techniques may still work well. Our findings emphasizes the importance of deeply investigating the context before designing and evaluating reviewer recommender systems.

We hope that the example of this study could serve as an inspiration for other researchers to employ more user- and context-centric methodology when evaluating prototypes of tools that are ultimately motivated by the need to optimize software developers' routine tasks. We believe that studies that are more focused on practical aspects ultimately bring the academic research closer to the software engineering industry.

5.10 Acknowledgements

The authors are grateful to all participants of surveys and interviews at JetBrains and Microsoft for their input. Vladimir thanks the amazing people at JetBrains for making the data mining possible. Special thanks to Ekaterina Stepanova for arranging the process from the legal side, and to Upsource team for their help with the technical aspects. We thank Arie van Deursen for his comments on the drafts of this paper.

Bacchelli gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PP00P2_170529.

6

Representation of Coding Style

This chapter is based on the paper "Building Implicit Vector Representations of Individual Coding Style" by Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli, presented online at the Workshop on Cooperative and Human Aspects of Software Engineering in 2020.

6.1 Introduction

Machine learning (ML) has lately been having more impact in software engineering by improving over state of the art in problems such as code summarization [254, 255] and program synthesis [256]. Many of the methods that apply ML techniques to code are aimed at enhancing software development tools by offering engineers assistance in routine tasks. Examples of such enhancements include code completion engines, static analysis, and automated code review systems [115]. Most of these methods are designed to assist in a task that is relevant within a short time scope, such as to insert the right code snippet or to fix problems in a single changeset.

While this assistance to developers promises a significant improvement in the daily experience with developer tools, it does not cover the complete scope of potential tooling support for engineering teams, particularly when in relation to socio-technical aspects. Indeed, developers report that their use of developer tools is not only related to technical artifacts, but is also a vital part of interpersonal communication in teams (e.g., as it is in the case of code review tools [257]).

Social aspects in teams are manifested in technical artifacts and records of these processes can be extracted with software repository mining techniques [258]. Still, modern software team collaboration tools make little to no use of the data available in software repositories to assist with social aspects of the engineering process. To enable tools to assist with interpersonal processes at the scale of a software team, one first needs reliable and transparent models of these processes as well as methods to retrieve corresponding data from software repositories. As a step in this direction, in this work, we propose a new approach to building representations of developers' individual coding fingerprints – or their *coding style*. Such representations can find use in the next generation of team collaboration tools, which could, for example, track the process of knowledge transfer in teams and provide assistance. Other potential applications include searching for similar developers and profiling of individual coding habits for tasks related to the management of human resources.

Existing work on code stylometry typically relies on explicitly defined features to represent code style [32]. We take a different direction: Instead of using explicit measures of code style, we implicitly extract the distinguishing features of individual developers by training a model to recognize authorship of a batch of code changes and processing the model's internal representations. The input of the model consists of changes made by a developer to individual methods, and its output is a label for the predicted author. To maximize transparency of the model, we use an attention mechanism – a technique widely used in neural machine translation [259], which allows us to point out particular code constructs that are more important for authorship attribution. After training the model to recognize the author of code changes, we extract representations of contributions of individual developers from it. We do so by combining the vector representations of individual method changes made by a developer over a time period, using the weights for vectors of individual method changes that are learned by the attention mechanism.

Finally, we assess the capability of representations of reflecting a practical social aspect, namely, learning within teams. For this, we produce multiple snapshots of individual representations of developers in a large open source project maintained by an enterprise, with each snapshot corresponding to a specific time period. Thanks to a localized develop-

ment team, we were able to also collect reports of mutual learning from the developers of this project. Finally, we look for a connection between reported learning and relative distances between developer representations. While we find no connection between reported learning and relative movement of developers' representations between consecutive time buckets, we see that reported learning is associated with lower distance between representations of two developers.

The primary contribution of this work is a novel method to extract representations of the contribution style of individual developers. The method is designed to be suitable for use with raw data from software repositories and to not require any additional labeling or explicit feature engineering. Another contribution is an empirical assessment of how the retrieved representations match learning as perceived by software developers.

6.2 Background and motivation

6.2.1 The need for developer representations

In the following, we reason about the importance of representations of individual developers' style, focusing on another class of tools (i.e., IDEs) as an example of a class of advanced tools successfully making use of a comprehensive model of the main medium they are designed to manipulate: code.

Despite the important role of team collaboration tools in software engineering, existing approaches aimed at improving software engineering tools with data have been mostly targeting coding environments and IDEs. Modern industry-grade IDEs, such as IntelliJ IDEA [260] and Eclipse [261], provide rich toolkits for code manipulation and maintenance. The IDEs feature automated code refactoring, code inspections pointing at potential issues, are able to provide high-level overview of large codebases, and enable deep integration of external tools, e.g. debuggers, with the code editor. Capability of IDEs to provide such rich code manipulation features is based on comprehensive internal models of software projects. In particular, IntelliJ relies on PSI [262] that represents a rich internal code model. Language-specific features in Eclipse IDEs are based on language support packages like JDT [263] and CDT [264] that as well manipulate with comprehensive language-specific program structure data. Outside the industrial IDE realm, academic methods aimed at coding manipulation and improvement also operate with code models and in some cases partially rely on language support initially designed for use in IDEs [265].

Modern team collaboration tools, such as code review tools, repository hosting engines, and bug tracking systems, are vital mediums for collaborative software engineering. In fact, these tools do not simply provide an environment to perform short-term tasks, like reviewing changes or communicating an issue, but also play a crucial role in supporting knowledge transfer in teams [257] and serve as a knowledge base [266].

In contrast to comprehensive code manipulation and problem detection features in IDEs, most team collaboration tools, despite their vital role in team-wide processes, do not maintain a comparably complex and detailed model of teams' communication structure, nor do these tools routinely analyze records of prior communication in teams. While there are exceptions to this rule, such as data-driven techniques like reviewer or assignee recommendation systems [20, 36, 216] and repository analytics features that are present in some collaboration tools, these tools are yet to evolve to feature and utilize a more compre-

hensive model of team communication and to assist in maintenance and improvement of communication at a larger time scale.

Enabling assistive features in team collaboration tools requires enabling tools to model social processes internally. While existing research suggests that social processes are to an extent reflected in technical artifacts [267] and can be extracted with data mining techniques [258], it is important to focus on extracting representations of individual properties of contributors from records of their collaborative work.

This work is dedicated to extracting representations of individual properties of engineer's coding that distinguish their contributions from their peers'. This could provide the tools with a sense of proximity of individual properties of their users' work, which could be use to detect learning in teams or provide onboarding assistance. We require that extraction of representations should not rely on any explicitly defined set of features, as opposed to existing code stylometry approaches. Using a neural code change embedding technique, we avoid feature engineering and utilize the ability of the model to capture optimal distinguishing features implicitly.

6.2.2 Existing work

Despite a solid track record of academic efforts and lack of widespread usage in modern team collaboration tools, we believe individual developer representations to be a promising ground for the evolution of collaboration tools.

The idea of building representations of individual developers' style has been around in the research community for several decades. The need for such representations is mostly motivated by the demand for code authorship attribution, which is deemed important for a variety of real-world applications, such as malware detection [32] and plagiarism elimination [268].

Some recent work is closely related to ours. Azcona *et al.* [269] propose building vector representations for individual computer science students, based on source code of their assignment submissions. As opposed to our work, they do not use any information on structure of code. Alsulami *et al.* [118] use a deep learning model to attribute authorship of source code, based on traversal sequences of the AST. Authorship attribution, however, is the sole task of their approach. Moreover, the model they propose works on code snippets and not code modifications, thus making it very hard to apply to data from software repositories.

6.3 Method

In contrast to explicitly defined feature sets for developers' coding style, commonly used in existing literature, we define individual coding style in the scope of a single repository as vaguely as *anything that distinguishes a developer's contribution to the codebase from their peers' contributions* and focus our method on capturing this.

We propose a two-step method for building the embeddings – essentially, vector representations – of individual code style. The overarching idea of our approach is to first learn to vectorize individual method changes in a way that best represents individual contribution style of each developer, then combine representations of multiple changes made by a single developer into this developer's individual contribution fingerprint.

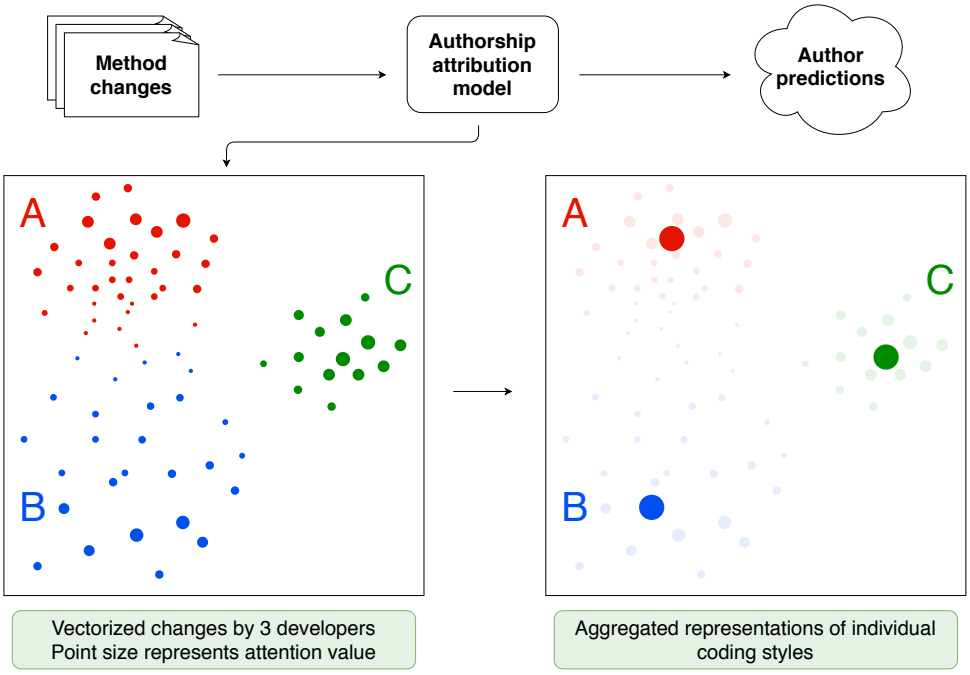


Figure 6.1: A high-level overview of our approach to building representations of individual developers

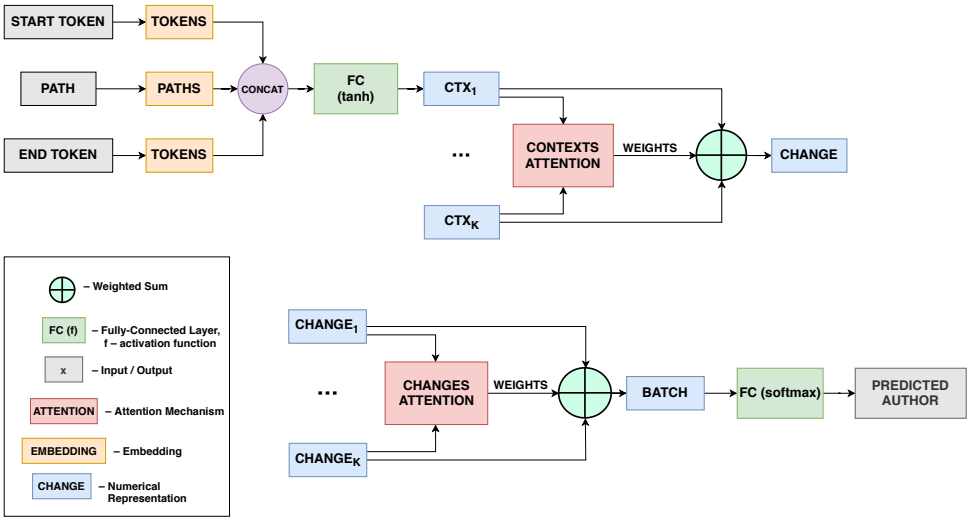


Figure 6.2: Overview of the authorship attribution pipeline that we use to obtain authorship-based embeddings of method changes and importance of individual method changes for attribution of authorship. The method nodes and their attention weights are later used to produce developer representations.

In the first step, we extract individual changes in Java methods from the project's VCS history and randomly group them into batches of changes authored by the same person. Then we train a neural network to vectorize individual changes and their batches so to distinguish between contributions as efficiently as possible.

At this step, the machine learning model essentially learns a function that maps a code change to a vector. The primary requirement for this function, that defines the learning process, is to represent method changes in a way that groups multiple changes made by one person close to each other and far from changes made by other developers. In addition, we use an attention mechanism by training the model on batches on multiple code changes instead of single changes. Thanks to attention, the model is also capable of assigning a weight to each method change, defining its "importance" for attributing authorship of each change batch. The inner workings of the authorship attribution model are explained in more detail in Section 6.3.1.

In the second step, we combine representations of changes made by every individual developer into a representation for that developer. In this step, we use the trained model for authorship attribution to produce vectors for individual code changes made by the developer, and calculate the representation of a person as a weighed sum of the changes they have made, using attention values from the model as weights.

In the rest of this section we provide a more detailed technical overview of the extraction pipeline. In the first part, we discuss the inner workings of the authorship attribution model. In the second part, we describe extraction of representations of individual developers from the trained model for authorship recognition.

6.3.1 Vectorizing code changes to represent authorship

The first step in building the representations of code style is to train a model to distinguish among the contributions of individual developers. During training for authorship attribution, the model implicitly learns to extract information that distinguishes method changes made by each developer from those made by their teammates.

We operate with method changes, rather than static snapshots of code snippets, to use authorship labels from version control: a method change can always be attributed to a single person who performs the change, and this data is already present in the version control system. Moreover, we use batches of randomly selected method changes as a unit of input for authorship attribution. This allows us to use the whole history of a project for training. For each developer, we shuffle their history of changes and split it into batches. A batch consists of 16 methods authored by the same developer, sampled uniformly randomly from their development history. We chose this number empirically as a good balance point between having too small batches and too little data points per developer. While making the authorship attribution task easier by letting the model focus on more important pieces of input, the use of batches and attention forces the model to estimate importance of specific changes before making a prediction. We further use these attention values when constructing individual style vectors for developers, so to let the code changes that are more representative of a developer contribute more to their fingerprint.

Our authorship recognition model is a neural network. Figure 6.2 presents an overview of the model. Its architecture is based on *code2vec* [5] – a state-of-the-art code embedding model. Similarly to *code2vec*, it uses path-based representations [119] of versions of each method before and after a change. Path-based representations are explained in detail in Section 3.2. Chapter 4 presents a deep evaluation of the authorship attribution technique and discusses its applicability on real software projects. In this chapter, we use the attribution model as an intermediate step.

Mechanics of the authorship recognition model.

The first step in the authorship recognition task is to convert a set of individual method changes into a vector form. For each changed method, we parse both the old method version and the new one to retrieve their ASTs. We extract path-contexts from both versions of the AST. We impose limitations on maximum length and width of the paths, to only include path-contexts representing local relations in code. To distill the concrete effect of each code change, we only use the difference in sets of path-contexts representing the old and the new versions of each method: we only include the path-contexts that were introduced or removed in the new version of the method after the change. We convert path-contexts representing the difference into a numerical form that can be passed to the neural network. First this, we apply vocabulary-based encoding to paths and tokens. Vocabulary-based encoding consists in representing every path or token with a unique integer number.

After that we learn the embeddings for tokens and paths. Essentially, at this step the model learns to convert them into a vector form in an optimal way that is most meaningful for the ultimate objective of attributing authorship of method change representations that they comprise. Initially represented by a random matrix, stacked embeddings of paths or tokens eventually converge to optimal values during training.

Further down the pipeline, the model concatenates the embedding vector for a path with embeddings of its start and end tokens to build a *path-context vector*. This vector is a

combined representation of the path and its end tokens. We transform the path-context vectors with a fully connected layer and aggregate them into *method change* vectors, using weights from an *attention* layer. The attention mechanism essentially attributes a "relevance" weight to each path-context in the batch corresponding to the code change. Path-contexts with higher attention values are more important for distinguishing between developers, i.e., capture more individual information. By highlighting the relevant path-contexts, the attention mechanism improves the accuracy of the model and improves its interpretability: it is possible to pinpoint the concrete path-contexts in the input.

As depicted in the bottom part of Figure 6.2, we use another attention mechanism to combine a batch of method change vectors, each corresponding to a change made to a single method, into a *change batch vector*. Combining changes in batches, rather than using a single change for every prediction, allows to attribute an attention weight, representing its importance for authorship attribution, to each individual method change. The size of this vector is a hyperparameter of the model, but we choose it to be much less than the number of possible developer labels to ensure that representations of developers are dense. The next fully connected layer with softmax activation solves the classification problem by learning to attribute a change batch vector to a concrete developer.

6.3.2 From authorship recognition to developer embeddings

The classifier in the authorship recognition model is learning to attribute a change batch vector (which is a weighted sum of change method vectors for methods in the batch) to an individual developer. Essentially, the whole model is learning to map batches of individual method changes into a vector space so that the sets of contributions of individual developers can be separated as well as possible. High accuracy in the authorship recognition task suggests that the learned model separates the space of method change vectors into areas that correspond to individual developers, thus capturing individual characteristics of a sample of a developer's contributions.

The pivotal idea of our approach is to extract a representation of a developer from the trained model. In addition, the attention mechanism learns to evaluate importance of representations of single method changes in the combined method batch vector. To extract a representation of contributions of a single developer over a period of time, we consider all of changes made by that developer during the period. We feed the representations of individual changes into the model and retrieve a vector representing each method change as well as the corresponding attention value. Finally, we combine these vectors and weights into a representation of a developer as a weighted sum.

This representation is used in further analysis: we calculate the representations for multiple time buckets, and retrieve multiple representation vectors for various team members, each corresponding to a certain time period. Finally, we explore whether positions and relative movement of representations are connected to learning from peers, as reported by developers.

6.3.3 Threats to validity

The curse of context. The authorship recognition model distinguishes contributors based on both the *structure* of code (which is represented by sequences of node types from AST paths) and the *context* of their changes (which is represented by the tokens). Tokens include

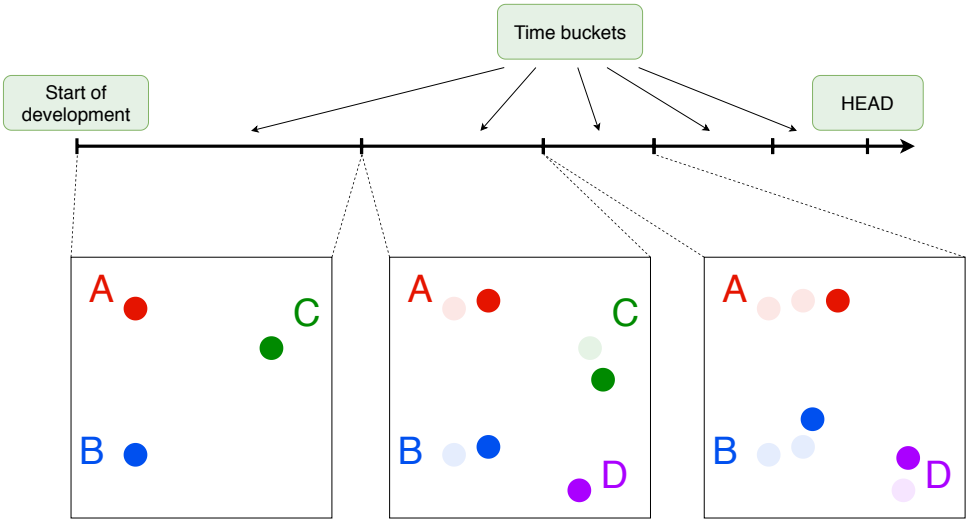


Figure 6.3: Change of developers' vectors between time buckets. A developer may be only active in some of the buckets.

variable names and names of declared and invoked methods. These names may be highly specific for a concrete narrow area of code. It is reasonable to think that in projects with practices of individual code ownership this context information alone can be enough to recognize the author of a method change. Given our ultimate goal of capturing individual characteristics of developers, including the context information in the model is not always desirable. While we perform a separate evaluation on data with excluded tokens, there is a chance that context information may as well be reflected in characteristic unique code patterns that are captured in sequences of AST node types.

Performance. We must note that resource consumption of our approach is very high, mostly due to the need to repeat training multiple times to reduce noise in the data. While not a crucial property for a proof-of-concept tool, reasonable performance of an approach is a requirement for practical applicability of an approach to this task. Producing a slow pipeline for a far-fetched, yet practical, goal impacts strength of the motivation.

6.4 Evaluation Setup

One critical design choice for our approach to building individual developer representations, or embeddings, is to take a step away of explicitly defined code style. In essence, as we use the authorship attribution task for building the representations, we implicitly define code style as *"anything that distinguishes between individuals' code"*. While avoiding explicit definition of code style gives potential to include characteristics of the style that are otherwise left out, this makes the evaluation of embeddings' quality challenging, as there is no ground truth to compare against.

To get a realistic estimate of quality of code style embeddings, we decided to evaluate them in the context of a possible application. As discussed in Section 6.2.1, one promising

application of code style representations is enabling team collaboration tools to make sense of proximity of individual contribution styles, capture the process of knowledge transfer in teams, and potentially provide aid by aligning this process to be more efficient. Essentially, we formulate the task of embeddings evaluation as evaluating their ability to *capture learning between individuals*. In the rest of this section, we elaborate on the evaluation setup and technique.

6.4.1 Dataset preparation

As an evaluation dataset, we use the source code and development history of IntelliJ Community.[1]

Merging, splitting and filtering. In the first step, we merge name-email pairs accounted in the VCS history and belonging to the same developer into a single entity. For this purpose, we use a separate user management tool, internally used by developers of IntelliJ and accessible to us, containing merged records of VCS entities for developers in the project. To facilitate running our pipeline on other projects, we implemented a simple algorithm for entities merging: it builds a bipartite graph of names and emails, where pairs that appear together are connected by an edge. Connected components in the graph correspond to merged entities. While the algorithm is not perfect, it gives an approximate merging which can be further improved manually. The algorithm cannot handle situations where a developer uses a nickname unrelated to their name as their VCS alias, or spells their name differently across several aliases. We include the implementation in the reproduction package.

Afterward, we split the history of the repository into multiple time chunks, each containing the same number of commits. Learning representations of developers over a small time chunk, rather than over complete history of a repository, allows to produce multiple representations of a developer, each corresponding to a relatively short time bucket. This accounts for potential changes in developers' coding traits over time and allows us to look at changes in distances between representations in consecutive time buckets – in other words, to track relative movement of representations.

As explained in Section 6.3, we focus on Java method changes for training and producing the representations. The repository contains contributions from about 500 developers, with a long tail of developers with only a few contributions. To ensure that we have sufficient data for every developer, we exclude the developers who made less than 1,000 method changes over the whole history of the repository. This left us with 124 active developers in the dataset. Having significant amounts of data for every developer, we increase stability of representations and reduce noise.

Noise reduction. The weights in the authorship attribution model, which ultimately represent the resulting change embedding function, are randomly initialized before learning and may converge to very different configurations depending on the initial random seed. Moreover, density of representations in each snapshot for multiple developers differs.

To account for the varying density, we calculate (for each time bucket) the average distance between every pair of developer representations, and divide the actual distances by this value. This allows us to compare distances between two given representations in

[1] `https://github.com/jetbrains/intellij-community`

consecutive time buckets and is necessary because density of representations may differ between two buckets.

On top of normalization, to account for the random nature of representations, we repeat the whole learning and style representation extraction 30 times and calculate the average normalized distance between every two developers in every time bucket across all runs. While making the process of obtaining representations from a large repository computationally demanding, repeating the learning and using the average make the resulting data less noisy and the comparison of distances between two different pairs more reliable.

The resulting data consists of 20 lists of relative distances between every two representations, with each snapshot corresponding to a certain time bucket. The number of resulting representations in a bucket varies between 23 and 87, depending on the bucket, displaying an upward trend, representing the growth of the team.

6.4.2 Team survey

To get a baseline to check to what extent proximity and relative movement of individual representations, as extracted by the model, reflect actual learning in the team, we circulated a short online survey with the development team of the project in the dataset via a post in the project's internal communication channel.

The goal of the survey was to collect information on mutual learning between some developers in the team, so that this data could be used as ground truth regarding actual learning taking place, which could let us see whether relative movement of developer representation reflected learning reported by developers. In the survey, we ask each respondents the questions depicted in Figure 6.4 three times to get information about three different colleagues they have learned from. We also included an option to mention one or two colleagues in a similar way. To avoid hinting the respondents with any particular definition of coding elements their learning may relate to, we explicitly stated that we would like them to define it for themselves and consider anything they may have learned during collaborative work, mentoring, code review, or other team activities. Responses to this question provide us with examples of positive pairs in terms of reported learning.

In addition, we ask the respondents to mention several developers who contribute to the same project but from whom they are sure they have not learned any elements of coding: *"Please name a few colleagues from the IntelliJ team you think you did not learn any coding elements from at all"*. This question provides us with a set of negative examples.

- **Name a colleague in your team from whom you have learned some elements of coding**
- **To what extent have you learned from the person above?**
  1 (Very little: I can barely identify any particulars)

  7 (Very much: I owe several elements of my coding to learning from this person)
- **When do you think you have been learning from them the most actively?**

Figure 6.4: Excerpt from the survey

6.4.3 Survey results and model output

The actual evaluation consisted in mapping the results of the developer survey onto data of relative movement of developers' representations between different time buckets.

While we asked the survey participants to indicate the degree of perceived learning and the time period when such a learning took place, we believe that the data for these two questions is not reliable enough. 3 respondents simply reported a similar neutral or extreme score for every person they mention and noted that they were confused by the request to indicate the degree of learning. Regarding the time period when learning took place, only one in three respondents provided any meaningful information, which in most cases was still too vague to attribute to a concrete time bucket. Thus, we only use the fact of a participant naming someone they have learned from, or certainly have not learned from, as extracted from the answers, to map the results of the survey onto the output of the model. The final data from the survey consists of 23 *positive pairs* – pairs of developers one of whom reporting learning from the other, and 13 *negative pairs*, where one of the developers named the other as someone they certainly have not learned any elements of coding from.

To see whether distances between representations reflect reported learning, we compare distributions of distances in all positive pairs, taken across all buckets where both developers in the pair displayed activity and were present in the data, to a similarly defined distribution across all negative pairs. The sample of distances for positive pairs consists of 229 values, and a similar sample for negative pairs has 113 values.

To extract dynamics of relative distances, we obtain a distribution of *differences of distances between two consecutive time buckets* for all pairs in the positive and negative groups, using every time bucket where these distances are defined both in the given and the previous bucket. The sample of distance differences for positive pairs contains 204 values, and the negative pairs yielded 99 values.

Finally, we use a 2-sample Kolmogorov-Smirnov test[2] to compare distributions of distances, as well as distributions of their differences, between the positive and negative groups.

6.5 Evaluation results

Relative distances. When tokens are present in the input data, the distribution of 229 distance values in pairs with reported learning has a mean of 0.85 and variance of 0.54. For pairs with reported lack of learning, 113 distance values display a mean of 1.19 with variance of 0.60. The p-value from the 2-sample Kolmogorov-Smirnov test comparing distributions of these two samples is under 10^{-7}. When tokens are removed from the input data (to minimize context information available to the model), the mean distance between positive pairs is 0.81 with a variance of 1.37. For negative pairs the mean distance is 0.98 and variance is 1.22. The KS test yields a p-value < 0.001.

These results suggest that the *representations of developers in pairs with reported learning are located closer to each other, both in cases with included and excluded tokens.*

[2] We use this test because we cannot make any assumptions about distribution of these values due to the obscure stochastic process of learning and the fact that this data may reflect some social processes in teams that are impossible to completely quantify.

Relative movement. We perform a similar comparison for the distributions of distances in pairs of representations between two consequent time buckets. With tokens in data, 204 values for positive pairs are distributed with mean of −0.005 and variance of 0.86. 99 values for negative pairs display a mean of 0.026 and variance of 0.59. The KS test yields a p-value of 0.94, suggesting that samples of distance differences for positive and negative pairs are likely from similar distributions. When no tokens are present, values for positive pairs display a distribution with mean of 0.046 and variance of 1.97. Values for negative pairs are distributed with mean of −0.045 and variance of 2.47. The p-value in the KS test is 0.16.

These results suggest that *distributions of differences in relative distances between consecutive time buckets are no different between pairs with reported learning and with reported lack of learning, regardless of whether tokens are included.*

Summary. Overall, learning is to an extent reflected in the distances between representations: distances between developers who report learning from each other are lower than between developers who report not learning from each other. This result persists when tokens are removed from the data, thus importantly suggesting how learning is not only captured in context of developers' contributions represented by tokens. However, a similar comparison for distributions of distance differences between consecutive time buckets suggests that distributions are similar for both groups of developer pairs.

6.6 Discussion

The core idea of our approach to building coding style representations is to step aside from explicitly defined feature sets for developer representations, rather build the representations of style by aggregating embeddings of code changes, which are produced to distinguish between developers as good as possible by the authorship attribution model.

We evaluated the computed developer embeddings through reports of peer learning in a large development team finding that inter-peer learning is indeed reflected in the embeddings: representations of developers who a given developer reports having learned from are closer to their own representation than representations of those who they reported not having learned from. Importantly, this effect persists even when the context information (reflected in concrete code tokens) is removed from the training data. This result suggests that implicitly built developer representations reflect the fuzzy process of learning in teams, by capturing individual characteristic patterns of code constructs and their proximity for people who report learning from each other and not just capturing the context information. However, the learning reflected in coding style might as well be an effect of shared guidelines or training. Our evaluation technique does not account for that and could be improved further.

Moreover, we do not see any connection between reported learning and relative movement of developer representations. In our opinion, the most important reason for the lack of such a connection is the low stability of the resulting representations. While reflecting learning in distributions of relative positions of individual representations, the representations are too noisy to reason about learning in side-by-side comparison of team snapshots in consecutive time buckets. We discuss possible ways to mitigate this in the next subsection.

6.6.1 Future work

Evaluation. A more elaborate evaluation of our approach, involving multiple software projects and more feedback from developers, could help clarify quality of the representations.

Other scopes. Apart from the learning detection task, the representations produced by our approach could find use in other contexts, including tasks already supported by team collaboration tools, e.g., recommendation of code reviewers [36] or issue assignment [216].

Alternative embedding techniques. We use a modified version of code2vec for method change embeddings. Using other embedding techniques could improve the performance of our approach.

Transparency. The authorship attribution model uses two attention layers: one to learn importance of individual changes in a batch and the other for individual path-contexts. While we use weights from the former to produce developer representations, a careful consideration of values from both layers could provide more insights on what makes a certain change characteristic for a developer.

Stability of representations. Noise and jitter in representation snapshots between time buckets make extracting representations harder. Additional constraints could help increase representations' stability. Figuring out a way to increase stability without compromising ability of representations to capture important social properties is another promising direction of future work.

6.7 Conclusion

We introduced an approach to building representations of individual coding style of developers relative to their peers in the team. The most important feature of our approach is that it does not require explicit feature engineering, rather relies on implicit vectorization of code changes via a code embedding model, trained to distinguish between changes made by individual developers and the aggregation of individual changes.

We demonstrate that it is possible to build representation of individual developers' coding style without defining style formally. The resulting representations reflect learning between peers in the team to a certain degree.

Reproducibility. The technical artifacts of this work – the pipeline to build the representations and data analysis code that we used to map the survey results to the data – are available online: `https://zenodo.org/record/3647645`

7

Conclusion

In this concluding chapter, we list the contributions of this dissertation, return to the research questions and to particular results that concern them, and reflect on implications of this dissertation for the process of research and innovation in software engineering.

7.1 Contributions

This dissertation makes several contributions to software engineering research.

- An evaluation of the effect of ignoring the non-linear nature of file modification histories in Git on file histories data and on performance of two data-driven techniques that rely on such data (Chapter 2);
- An open-source toolkit for mining of path-based representations, which are required for many machine learning models operating with code, and other data derived from abstract syntax trees of code in arbitrary languages (Chapter 3);
- A language-agnostic technique for authorship attribution of source code, performing on par or better than language-specific state-of-the-art models (Chapter 4);
- The first ever evaluation of a code reviewer recommendation model in practice (Chapter 5);
- A study of perception of code reviewer recommendation features by users and an investigation of their information needs during reviewer selection(Chapter 5);
- A novel approach to unsupervised extraction of individual coding style without relying on explicit features (Chapter 6);
- A conceptual model for the lifecycle of data-driven features in software engineering tools, and a discussion of potential directions to rationalize innovation in this direction (Chapter 1 and Chapter 7).

7.2 Revisiting the Research Questions

This dissertation explores several aspects of design, evaluation, and implementation of data-driven techniques for software engineering tools. The focus of the majority of recent research in our field, ultimately motivated by the need to improve existing software engineering tools by utilizing historical data, is technical essence of data processing techniques. The broad message we are looking to convey with this thesis is that stages of the broad design process of data-driven techniques, namely data collection and evaluation, are no less important for the final goal – improvement of software engineering tools and processes.

RQ1. What is the effect of simplifications in data mining procedures on (i) resulting data and (ii) performance of techniques that rely on this data? In Chapter 2 we explored how the assumption on linearity of Git history, sometimes implicitly made at the stage of data collection, impacts the raw data on modification history of files, which in turn negatively affects performance of two data-driven techniques that rely on this data: code reviewer recommendation and change recommendation. In particular, in a large stratified sample of over 1400 repositories, in over 70% of repositories this assumption would lead to a difference in retrieved file history data. In fact, the histories appear different for 19% of more than 2,000,000 individual files in the sample. This difference is particularly prominent in projects that consistently prefer merging to rebasing, such as most projects in the OpenStack ecosystem.

The difference in retrieved data may pose a threat to performance of methods that rely on the data. In the same chapter we demonstrate that the difference in data impairs performance of reviewer recommendation and change recommendation – two data-driven techniques that, in their basic variations, use file histories as a sole source of data. While the performance drop may appear small, especially in Eclipse projects where the divergence in data is not very high, the simplified mining method may jeopardize a technique or a feature that relies on the data, especially when achieving state-of-the-art performance is a goal. Moreover, calling a seemingly small effect "small enough to ignore" is dangerous, because interpretation of formally measured quality of techniques in terms of (potential) end-user value is hard.

RQ2. How can we convert mining pipelines into reusable tools? Mining data for data-driven techniques is often a demanding task. With the growing variety of techniques and data sources, the research community is still nowhere near having an established toolkit for mining of multivariate software data. A lot of effort of researchers working on data-driven techniques thus has to be spent on building and debugging data mining pipelines, if existing datasets are not sufficient. Oftentimes, mining code is custom-tailored for a particular task and is not reused in other studies. In Chapter 3 we presented *astminer*, a flexible tool for mining of path-based representations of code, which is an emerging method to represent code used by several state-of-the-art data-driven techniques based on modern machine learning. *Astminer* is based on our own mining pipeline. Its development, which mostly consisted in separating the reusable part from our problem-specific code, adding convenient user interfaces, and writing some documentation, is an example of how it is possible, with relatively little effort, to provide value for the community in the form of a reusable tool that allows others not to implement complex analysis pipelines from scratch. By the time of writing of this conclusion, *astminer* had already proven useful in several studies not related to our work [270, 271].

RQ3. How closely do synthetic datasets resemble data from real projects? In Chapter 4, among other contributions, we propose a novel approach to generation of datasets for evaluation of authorship attribution techniques. Unlike prior studies that take a synthetic approach with limiting datasets to collections of code snippets authored by a single person, such as programming contest solutions and single-author projects, our approach supports generation of datasets from any Git repository. Use of arbitrary repositories brings the evaluation data closer to data that occurs in potential real-world applications of authorship attribution techniques. We find that aspects specific to collaboratively developed software, namely difference of contributions of individual developers in terms of change context and time, dramatically decrease accuracy of authorship attribution. This finding supports the statement that synthetic datasets can be different from data that naturally occurs in collaborative software projects in ways that greatly influence performance of data-driven techniques.

RQ4. Are there cases when evaluation of techniques in the lab does not reflect their usefulness for end users? One implication of the previous research question, which concerns fitness of datasets for evaluation of potential practical value, is that in general it is not possible to assess potential practical value of data-driven techniques by performance metrics alone. In Chapter 5 we study perception of reviewer recommendation systems by their users. We find that, while demonstrating reasonable formal performance metrics and being generally perceived as providing relevant results, reviewer recommendation engines rather rarely actually assist users in selecting reviewers. Also, we could not detect traces of interaction of users with the recommender in historical data. These findings further reinforce the notion of the gap between formal performance of data-driven techniques and their value for end users.

RQ5. Can concrete data from code repositories provide insight into fuzzy processes, such as knowledge transfer? The final Chapter 6 presents a rather ambitious attempt to extract traces of the intangible process of learning from source code changes: we find that it is be possible at least to some extent. This result supports the notion of extracting social information from collaboration artifacts being a promising direction for further research. This is further supported by the fact that most of existing data-driven techniques both in research and in practice are aimed at providing assistance with small and rather technical tasks, often related to code, while team collaboration tools that we use today still have a lot of potential for further improvement.

7.3 Outlook

This dissertation, its diverse research results, and their interpretations repeatedly point out the proverbial gap between existing points for improvement of tools and processes in software engineering practice and academic approaches to improving them. In a broad sense, this gap is manifested in work dedicated to building new data-driven techniques for software engineering tools, even those explicitly motivated by the demand for better tooling, consistently relying on assumptions that do not completely align with realities of software engineering practice.

Each individual chapter in this dissertation can be viewed as addressing a particular narrow aspect of this broad gap. Data mining and processing constitutes a significant technical challenge (Chapter 3). Simplifications in this process lead to suboptimal perfor-

mance of potentially useful techniques (Chapter 2). Many techniques are evaluated with artificial metrics on dedicated datasets used across multiple studies; Good performance of techniques on such datasets may not mean a similarly good performance on more realistic data (Chapter 4) or high perceived usefulness (Chapter 5). We suggest interpreting these results, combined, as giving a case for action in amending the misalignment between research and practice.

It is close to impossible to point out one single cause for existence of this gap between research and practice. It is rather caused by a whole spectrum of various misalignments in motivation, incentives, planning horizons, and traditions between academic environments, where most of research originates from, and industry, where most software is developed [272]. Instead of discussing these individual aspects, we will focus on suggesting concrete steps for making the process of development of data-driven software engineering tools more efficient.

Software practitioners, particularly those responsible for setting up development processes and building tools, could follow related research more closely, and actively collaborate with the research community by sharing relevant experience and actual pain points of practitioners. Practice tracks of software engineering research conferences, where practitioners, ironically, largely remain a minority, could serve as one of the platforms for such collaboration.

Some steps from the academic side of the gap could help establish closer collaboration with practitioners. One prominent example is the format of presentation of research results: research papers and formal presentations are, arguably, a suboptimal format for efficiently spreading new ideas, and should give way to more flexible and efficient formats, such as interactive prototypes of techniques, as a primary mode of demonstrating results to practitioners.

With numerous studies dedicated to understanding and describing daily routines of software engineers, their coding practices, and use of tools, not so many explicitly seek to build a deep understanding of their needs, or to systematically identify specific points for improvement in existing engineering processes and tools from the perspective of a practitioner. A deeper understanding of users' interaction with their tools, shared across the community, could help researchers tackle problems that are more relevant in practice. The field of Human-Computer Interaction research, closely related to ours, could be a great source of inspiration and expertise here.

While essentially working towards the same goal, research and industry present too different career tracks. While this may not be a big problem in less practice-oriented fields, it certainly is in ours, where state of the practice changes so fast that lifetimes of some technologies are comparable to the duration of an average journal review. Collaborations between academic researchers and practitioners have a great potential of turning the misunderstandings, often caused by the differences in models and approaches between academia and industry, into advantages. This is possible when roles of each side in such collaborations are clearly defined to benefit from the unique perspective of the respective sides. Industrial projects provide a good setup for academic software engineering researchers to do some hands-on work and exchange experience and perspectives with their colleagues from industry. The industry partner should normally be responsible for formulating desired outcomes, handling the technical complexity of joint projects, and

sharing data that is often not available to academic researchers. At the same time, deep understanding of the research landscape and knowledge of related work, usually possessed by experienced academic researchers, should help ensure that real-world problems are addressed reasonably and no wheels are being reinvented. Companies whose business relies on innovative methods could benefit from defining dedicated job roles and sponsorship programs to foster collaborations with academia. While this is not uncommon in industry giants, such measures could perhaps prove fruitful in smaller companies as well.

Some of the aspects of the gap should be a responsibility of institutions and policy makers. If agreements between researchers and their institutions could provide greater flexibility in terms of teaching obligations, it would be easier for professors to tap into more practical work. Providing flexible options for industry professionals to participate in research could be a way to incentivize the reverse flow of expertise. In particular, better alignment of compensation in academic software engineering research and software industry could help ensure that switching to academic settings, perhaps temporarily, is at least feasible. Making these two labour markets intersect would require loosening strict regulations and compensation frameworks that are in place in academic institutions. Strict requirements for research-related positions in academia, of which most require a PhD and a strong publication record, also stand in a way of bringing professionals with strong understanding of state of the practice to work shoulder-by-shoulder with academic researchers (observing the social distance, of course).

To finish on a positive note, we should mention existing opportunities for collaboration between researchers and practitioners. Software businesses do invest in research, either through in-house R&D departments or through partnerships with academic institutions. In some countries, including The Netherlands, research investments from private enterprises are incentivized by the government through tax credit and cash grants. There is a possibility for industry sabbaticals for professors, which helps them remain relevant with regard to practice. Finally, it is not uncommon for research directions originating in academia to grow into successful businesses.

Bibliography

References

[1] A. E. Hassan, *The road ahead for mining software repositories,* in *Frontiers of Software Maintenance, 2008. FoSM 2008.* (IEEE, 2008) pp. 48–57.

[2] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, *What is the vocabulary of flaky tests? MSR 2020,* (2020).

[3] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, *Method-level bug prediction,* in *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (IEEE, 2012) pp. 171–180.

[4] R. Compton, E. Frank, P. Patros, and A. Koay, *Embedding java classes with code2vec: improvements from variable obfuscation,* in *MSR 2020* (ACM, 2020).

[5] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *code2vec: Learning distributed representations of code,* Proceedings of the ACM on Programming Languages **3**, 40 (2019).

[6] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, *Using others' tests to identify breaking updates,* MSR. ACM, Seoul, Republic of Korea (2020).

[7] X. Cai, Y. Niu, S. Geng, J. Zhang, Z. Cui, J. Li, and J. Chen, *An under-sampled software defect prediction method based on hybrid multi-objective cuckoo search,* Concurrency and Computation: Practice and Experience **32**, e5478 (2020).

[8] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, *Mining version histories to guide software changes,* IEEE Transactions on Software Engineering **31**, 429 (2005).

[9] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, *Who should review my code? a file location-based code-reviewer recommendation approach for modern code review,* (2015).

[10] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, *Fuzzy set-based automatic bug triaging (nier track),* in *Proceedings of the 33rd International Conference on Software Engineering* (2011) pp. 884–887.

[11] K. K. Chaturvedi, V. Sing, and P. Singh, *Tools in mining software repositories,* in *2013 13th International Conference on Computational Science and Its Applications* (IEEE, 2013) pp. 89–98.

[12] T. Ball and S. G. Eick, *Software visualization in the large,* Computer **29**, 33 (1996).

[13] R. Wettel and M. Lanza, *Codecity: 3d visualization of large-scale software,* in *Companion of the 30th international conference on Software engineering* (2008) pp. 921–922.

[14] H. Remus and S. Zilles, *Prediction and management of program quality,* in *Proceedings of the 4th international conference on Software engineering* (1979) pp. 341–350.

[15] M. Paixao, J. Krinke, D. Han, and M. Harman, *Crop: Linking code reviews to source code changes,* in *Proceedings of the 15th International Conference on Mining Software Repositories* (2018) pp. 46–49.

[16] J. Guo, J. Cheng, and J. Cleland-Huang, *Semantically enhanced software traceability using deep learning techniques,* in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)* (IEEE, 2017) pp. 3–14.

[17] A. Bacchelli, M. Lanza, and R. Robbes, *Linking e-mails and source code artifacts,* in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1* (ACM, 2010) pp. 375–384.

[18] M. M. Rahman, C. K. Roy, J. Redl, and J. A. Collins, *Correct: Code reviewer recommendation at github for vendasta technologies,* in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on* (IEEE, 2016) pp. 792–797.

[19] Y. Tian, D. Wijedasa, D. Lo, and C. Le Goues, *Learning to rank for bug report assignee recommendation,* in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (IEEE, 2016) pp. 1–10.

[20] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, *Who should review my code? a file location-based code-reviewer recommendation approach for modern code review,* in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on* (IEEE, 2015) pp. 141–150.

[21] X. Xia, D. Lo, Y. Ding, J. M. Al-Kofahi, T. N. Nguyen, and X. Wang, *Improving automated bug triaging with specialized topic model,* IEEE Transactions on Software Engineering **43**, 272 (2016).

[22] GitHub Help, *About pull request reviews,* `https://help.github.com/articles/about-pull-request-reviews/` (2017).

[23] U. Alon, O. Levy, and E. Yahav, *code2seq: Generating sequences from structured representations of code,* CoRR **abs/1808.01400** (2018), arXiv:1808.01400 .

[24] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *Code2vec: Learning distributed representations of code,* Proc. ACM Program. Lang. **3**, 40:1 (2019).

[25] *Code completion – help : Intellij idea,* `https://www.jetbrains.com/help/idea/2020.2/auto-completing-code.html#configure-code-completion` (2020), accessed: 2020-09-08.

[26] 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 *2009 6th IEEE International Working Conference on Mining Software Repositories* (2009) pp. 1–10.

[27] V. Kovalenko, F. Palomba, and A. Bacchelli, *Mining file histories: Should we consider branches?* in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018) pp. 202–213.

[28] J. Ś liwerski, T. Zimmermann, and A. Zeller, *When do changes induce fixes?* ACM sigsoft software engineering notes **30**, 1 (2005).

[29] K. Herzig, S. Just, and A. Zeller, *It's not a bug, it's a feature: how misclassification impacts bug prediction,* in *2013 35th International Conference on Software Engineering (ICSE)* (IEEE, 2013) pp. 392–401.

[30] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu, *Putting it all together: Using socio-technical networks to predict failures,* in *2009 20th International Symposium on Software Reliability Engineering* (IEEE, 2009) pp. 109–119.

[31] P. F. Xiang, A. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. Matchen, A. Empere, P. L. Tarr, C. Williams, *et al.*, *Ensemble: a recommendation tool for promoting communication in software teams.* in *RSSE@ SIGSOFT FSE* (2008) p. 2.

[32] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, *De-anonymizing programmers via code stylometry,* in *24th USENIX Security Symposium (USENIX Security 15)* (USENIX Association, Washington, D.C., 2015) pp. 255–270.

[33] J. ú. s. M. Gonz á lez Barahona and G. Robles, *On the reproducibility of empirical software engineering studies based on data retrieved from development repositories,* Empirical Software Engineering **17**, 75 (2012).

[34] M. D'Ambros, M. Lanza, and R. Robbes, *Evaluating defect prediction approaches: a benchmark and an extensive comparison,* Empirical Software Engineering **17**, 531 (2012).

[35] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, *Does bug prediction support human developers? findings from a google case study,* in *Proceedings of the 2013 International Conference on Software Engineering* (IEEE Press, 2013) pp. 372–381.

[36] V. Kovalenko, N. Tintarev, E. Pasynkov, C. Bird, and A. Bacchelli, *Does reviewer recommendation help developers?* IEEE Transactions on Software Engineering (2018).

[37] J. Scholtz, *Usability evaluation,* National Institute of Standards and Technology **1** (2004).

[38] A. Mockus and D. M. Weiss, *Predicting risk of software changes,* Bell Labs Technical Journal **5**, 169 (2000).

[39] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, *The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects,* in *Proceedings of the 11th Working Conference on Mining Software Repositories* (2014) pp. 192–201.

[40] Y. Yu, H. Wang, G. Yin, and T. Wang, *Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?* Information and Software Technology **74**, 204 (2016).

[41] G. Gousios, *The ghtorrent dataset and tool suite,* in *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13 (IEEE Press, Piscataway, NJ, USA, 2013) pp. 233–236.

[42] M. Beller, G. Gousios, and A. Zaidman, *Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration,* in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (IEEE, 2017) pp. 447–450.

[43] *Upsource,* `https://www.jetbrains.com/upsource/` (2018), accessed: 2018-04-22.

[44] R. P. R. Ch, O. Dieste, N. Juristo, *et al.*, *Statistical errors in software engineering experiments: A preliminary literature review,* in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)* (IEEE, 2018) pp. 1195–1206.

[45] F. J. Massey Jr, *The kolmogorov-smirnov test for goodness of fit,* Journal of the American statistical Association **46**, 68 (1951).

[46] H. B. Mann and D. R. Whitney, *On a test of whether one of two random variables is stochastically larger than the other,* The annals of mathematical statistics , 50 (1947).

[47] J. J. Litchfield and F. Wilcoxon, *A simplified method of evaluating dose-effect experiments,* Journal of pharmacology and experimental therapeutics **96**, 99 (1949).

[48] D. M. Powers, *Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation,* (2011).

[49] M. Chen and P. Liu, *Performance evaluation of recommender systems.* International Journal of Performability Engineering **13** (2017).

[50] A. Joshi, S. Kale, S. Chandel, and D. K. Pal, *Likert scale: Explored and explained,* Current Journal of Applied Science and Technology , 396 (2015).

[51] R. K. Bakshi, N. Kaur, R. Kaur, and G. Kaur, *Opinion mining and sentiment analysis,* in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)* (IEEE, 2016) pp. 452–455.

[52] D. Spencer and T. Warfel, *Card sorting: a definitive guide,* Boxes and arrows **2**, 1 (2004).

[53] 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 *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (IEEE, 2009) pp. 1–10.

[54] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, *Cohesive and isolated development with branches,* in *International Conference on Fundamental Approaches to Software Engineering* (Springer, 2012) pp. 316–331.

[55] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development* (" O'Reilly Media, Inc.", 2012).

[56] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, *Modern code reviews in open-source projects: Which problems do they fix?* in *Proceedings of the 11th working conference on mining software repositories* (ACM, 2014) pp. 202–211.

[57] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, *A developer centered bug prediction model,* IEEE Transactions on Software Engineering **44**, 5 (2018).

[58] H. Kagdi, M. L. Collard, and J. I. Maletic, *A survey and taxonomy of approaches for mining software repositories in the context of software evolution,* Journal of Software: Evolution and Process **19**, 77 (2007).

[59] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, *The scent of a smell: An extensive comparison between textual and structural smells,* IEEE Transactions on Software Engineering (2017).

[60] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, *An empirical investigation into the nature of test smells,* in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ACM, 2016) pp. 4–15.

[61] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, *There and back again: Can you compile that snapshot?* Journal of Software: Evolution and Process **29** (2017).

[62] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, *When and why your code starts to smell bad (and whether the smells go away),* IEEE Transactions on Software Engineering **43**, 1063 (2017).

[63] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, *The limited impact of individual developer data on software defect prediction,* Empirical Software Engineering **18**, 478 (2013).

[64] A. E. Hassan, *Predicting faults using the complexity of code changes,* in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (IEEE, 2009) pp. 78–88.

[65] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, *Toward a smell-aware bug prediction model,* IEEE Transactions on Software Engineering (2017).

[66] L. Hattori and M. Lanza, *Mining the history of synchronous changes to refine code ownership,* in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (IEEE, 2009) pp. 141–150.

[67] M. Greiler, K. Herzig, and J. Czerwonka, *Code ownership and software quality: a replication study,* in *Proceedings of the 12th Working Conference on Mining Software Repositories* (IEEE Press, 2015) pp. 2–12.

[68] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, *Revisiting code ownership and its relationship with software quality in the scope of modern code review,* in *Proceedings of the 38th international conference on software engineering* (ACM, 2016) pp. 1039–1050.

[69] V. Balachandran, *Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation,* in *Software Engineering (ICSE), 2013 35th International Conference on* (IEEE, 2013) pp. 931–940.

[70] M. M. Rahman, C. K. Roy, and J. A. Collins, *Correct: code reviewer recommendation in github based on cross-project and technology experience,* in *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on* (IEEE, 2016) pp. 222–231.

[71] P. Thongtanunam, R. G. Kula, A. E. C. Cruz, N. Yoshida, and H. Iida, *Improving code review effectiveness through reviewer recommendations,* in *Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering* (ACM, 2014) pp. 119–122.

[72] M. í. c. Aniche, G. Bavota, C. Treude, M. A. é. l. Gerosa, and A. van Deursen, *Code smells for model-view-controller architectures,* Empirical Software Engineering , 1 (2017).

[73] G. Catolino, F. Palomba, A. De Lucia, F. Ferrucci, and A. Zaidman, *Developer-related factors in change prediction: an empirical assessment,* in *Proceedings of the 25th International Conference on Program Comprehension* (IEEE Press, 2017) pp. 186–195.

[74] B. Turhan, T. Menzies, A. ş. e. B. Bener, and J. Di Stefano, *On the relative value of cross-company and within-company data for defect prediction,* Empirical Software Engineering **14**, 540 (2009).

[75] E. Kocaguneli, T. Menzies, and J. W. Keung, *On the value of ensemble effort estimation,* IEEE Transactions on Software Engineering **38**, 1403 (2012).

[76] *Mining File Histories: Should We Consider Branches? - Online Appendix,* `https://github.com/vovak/branches` (2018), accessed: 2018-07-24.

[77] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, *Streamed lines: Branching patterns for parallel software development,* (1998).

[78] J. Buffenbarger and K. Gruell, *A branching/merging strtegy for parallel software development,* in *International Symposium on Software Configuration Management* (Springer, 1999) pp. 86–99.

[79] C. Bird, T. Zimmermann, and A. Teterev, *A theory of branches as goals and virtual teams,* in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering* (ACM, 2011) pp. 53–56.

[80] C. Bird and T. Zimmermann, *Assessing the value of branches with what-if analysis,* in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (ACM, 2012) p. 45.

[81] E. Shihab, C. Bird, and T. Zimmermann, *The effect of branching strategies on software quality,* in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement* (ACM, 2012) pp. 301–310.

[82] *Lkml: Linus torvalds: Re: git mv (was re: [git pull] acpi & suspend patches for 2.6.29-rc0),* `https://lkml.org/lkml/2009/1/9/323` (2018), accessed: 2018-04-24.

[83] H. M. Michaud, D. T. Guarnera, M. L. Collard, and J. I. Maletic, *Recovering commit branch of origin from github repositories,* in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on* (IEEE, 2016) pp. 290–300.

[84] M. D'Ambros, M. Lanza, and R. Robbes, *An extensive comparison of bug prediction approaches,* in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on* (IEEE, 2010) pp. 31–41.

[85] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, *A systematic literature review on fault prediction performance in software engineering,* IEEE Transactions on Software Engineering **38**, 1276 (2012).

[86] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. ş. e. Bener, *Defect prediction from static code features: current results, limitations, new approaches,* Automated Software Engineering **17**, 375 (2010).

[87] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, *Predicting source code changes by mining change history,* IEEE transactions on Software Engineering **30**, 574 (2004).

[88] *Why sqlite does not use git,* `https://sqlite.org/whynotgit.html` (2018), accessed: 2018-04-18.

[89] D. Kawrykow and M. P. Robillard, *Non-essential changes in version histories,* in *Proceedings of the 33rd International Conference on Software Engineering* (ACM, 2011) pp. 351–360.

[90] S. W. Thomas, *Mining software repositories using topic models,* in *Proceedings of the 33rd International Conference on Software Engineering* (ACM, 2011) pp. 1138–1139.

[91] *Jgit,* `https://www.eclipse.org/jgit/` (2018), accessed: 2018-04-09.

[92] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, *How do centralized and distributed version control systems impact software changes?* in *Proceedings of the 36th International Conference on Software Engineering* (ACM, 2014) pp. 322–333.

[93] A. Mockus, *Amassing and indexing a large sample of version control systems: Towards the census of public source code history,* in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on* (IEEE, 2009) pp. 11–20.

[94] D. Spinellis, *Git,* IEEE software **29**, 100 (2012).

[95] *git-log(1) manual page,* https://mirrors.edge.kernel.org/pub/software/scm/git/docs/git-log.html (2018), accessed: 2018-04-24.

[96] *git - find all the direct descendants of a given commit,* https://stackoverflow.com/questions/27960605/find-all-the-direct-descendants-of-a-given-commit#27962018 (2018), accessed: 2018-04-24.

[97] M. Nagappan, T. Zimmermann, and C. Bird, *Diversity in software engineering research,* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (ACM, 2013) pp. 466–476.

[98] *Github,* https://github.com (2018), accessed: 2018-04-22.

[99] *Gerrit code review,* https://www.gerritcodereview.com/ (2018), accessed: 2018-04-16.

[100] *Eclipse - the eclipse foundation open source community website.* https://www.eclipse.org/ (2018), accessed: 2018-04-16.

[101] *Open source software for creating private and public clouds.* https://www.openstack.org/ (2018), accessed: 2018-04-16.

[102] *Eclipse code review,* https://git.eclipse.org/r/ (2018), accessed: 2018-04-16.

[103] *Openstack code review,* https://review.openstack.org/ (2018), accessed: 2018-04-16.

[104] E. M. Voorhees *et al.*, *The trec-8 question answering track report.* in *Trec*, Vol. 99 (1999) pp. 77–82.

[105] *A java implementation of the apriori algorithm for finding frequent item sets and (optionally) generating association rules,* https://github.com/michael-rapp/Apriori/ (2018), accessed: 2018-04-22.

[106] R. Agarwal, R. Srikant, *et al.*, *Fast algorithms for mining association rules,* in *Proc. of the 20th VLDB Conference* (1994) pp. 487–499.

[107] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, *Program synthesis from natural language using recurrent neural networks,* University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01 (2017).

[108] P. Yin and G. Neubig, *A syntactic neural model for general-purpose code generation,* in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1 (2017) pp. 440–450.

[109] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. é. N. Amaral, *Syntax and sensibility: Using language models to detect and correct syntax errors,* in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (IEEE, 2018) pp. 311–322.

[110] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, *Suggesting accurate method and class names,* in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ACM, 2015) pp. 38–49.

[111] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, *Deep code comment generation,* in *Proceedings of the 26th Conference on Program Comprehension* (ACM, 2018) pp. 200–210.

[112] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, *Summarizing source code using a neural attention model,* in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1 (2016) pp. 2073–2083.

[113] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, *The case for learned index structures,* in *Proceedings of the 2018 International Conference on Management of Data* (ACM, 2018) pp. 489–504.

[114] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. Torr, *Adaptive neural compilation,* in *Advances in Neural Information Processing Systems* (2016) pp. 1444–1452.

[115] V. Raychev, M. Vechev, and E. Yahav, *Code completion with statistical language models,* in *PLDI'14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, 2014) pp. 419–428.

[116] A. Gupta and N. Sundaresan, *Intelligent code reviews using deep learning,* (2018).

[117] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, *De-anonymizing programmers via code stylometry,* in *24th USENIX Security Symposium (USENIX Security), Washington, DC* (2015).

[118] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, *Source code authorship attribution using long short-term memory based networks,* in *European Symposium on Research in Computer Security* (Springer, 2017) pp. 65–82.

[119] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *A general path-based representation for predicting program properties,* in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (ACM, 2018) pp. 404–419.

[120] D. DeFreez, A. V. Thakur, and C. Rubio-Gonz á lez, *Path-based function embedding and its application to error-handling specification mining,* in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ACM, 2018) pp. 423–433.

[121] *ANTLR,* https://antlr.org/ (2019), accessed: 2019-01-16.

[122] A. T. Ying and M. P. Robillard, *Code fragment summarization,* in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (ACM, 2013) pp. 655–658.

[123] J. . R. é my Falleri, F. é al Morandat, X. Blanc, M. Martinez, and M. Monperrus, *Fine-grained and accurate source code differencing,* in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (2014) pp. 313–324.

[124] *Kotlin programming language,* https://kotlinlang.org/ (2019), accessed: 2019-01-21.

[125] *antlr/grammars-v4: Grammars written for antlr v4; expectation that the grammars are free of actions.* https://github.com/antlr/grammars-v4 (2019), accessed: 2019-01-21.

[126] *pathminer,* https://github.com/vovak/astminer/releases/tag/pathminer (2019), accessed: 2019-03-15.

[127] P. W. Oman and C. R. Cook, *Programming style authorship analysis,* in *Proceedings of the 17th Conference on ACM Annual Computer Science Conference*, CSC '89 (ACM, New York, NY, USA, 1989) pp. 320–326.

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

[129] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, *A degree-of-knowledge model to capture source code familiarity,* in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10 (ACM, New York, NY, USA, 2010) pp. 385–394.

[130] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, *How developers drive software evolution,* in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)* (2005) pp. 113–122.

[131] 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*, ESEC/FSE '11 (ACM, New York, NY, USA, 2011) pp. 4–14.

[132] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, *Revisiting code ownership and its relationship with software quality in the scope of modern code review,* in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016) pp. 1039–1050.

[133] F. Rahman and P. Devanbu, *Ownership, experience and defects: A fine-grained study of authorship,* in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11 (ACM, New York, NY, USA, 2011) pp. 491–500.

[134] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, *How do fixes become bugs?* in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11 (ACM, New York, NY, USA, 2011) pp. 26–36.

[135] C. Zhang, S. Wang, J. Wu, and Z. Niu, *Authorship identification of source codes,* in *APWeb/WAIM* (2017).

[136] S. Burrows and S. Tahaghoghi, *Source code authorship attribution using n-grams,* ADCS 2007 - Proceedings of the Twelfth Australasian Document Computing Symposium (2007).

[137] J. Kothari, M. Shevertalov, E. Stehle, and S. Mancoridis, *A probabilistic approach to source code authorship identification,* in *Fourth International Conference on Information Technology (ITNG'07)* (2007) pp. 243–248.

[138] B. Stein, N. Lipka, and P. Prettenhofer, *Intrinsic plagiarism analysis,* Language Resources and Evaluation **45**, 63 (2011).

[139] P. S. Menell, *Api copyrightability bleak house: Unraveling and repairing the oracle v. google jurisdictional mess,* Berkeley Technology Law Journal (2017).

[140] S. Baltes and S. Diehl, *Usage and attribution of stack overflow code snippets in github projects,* Empirical Software Engineering **24**, 1259 (2019).

[141] Y. Golubev, M. Eliseeva, N. Povarov, and T. Bryksin, *A study of potential code borrowing and license violations in java projects on github,* in *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20 (Association for Computing Machinery, New York, NY, USA, 2020) p. 54–64.

[142] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, *De-anonymizing programmers via code stylometry,* in *24th USENIX Security Symposium (USENIX Security 15)* (USENIX Association, Washington, D.C., 2015) pp. 255–270.

[143] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, and R. Greenstadt, *Source code authorship attribution using long short-term memory based networks,* in *Computer Security – ESORICS 2017*, edited by S. N. Foley, D. Gollmann, and E. Snekkenes (Springer International Publishing, Cham, 2017) pp. 65–82.

[144] X. Yang, G. Xu, Q. Li, Y. Guo, and M. Zhang, *Authorship attribution of source code by using back propagation neural network based on particle swarm optimization,* PLOS ONE **12**, 1 (2017).

[145] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, *A general path-based representation for predicting program properties,* SIGPLAN Not. **53**, 404 (2018).

[146] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, *Source code author identification based on n-gram author profiles,* in *Artificial Intelligence Applications and Innovations*, edited by I. Maglogiannis, K. Karpouzis, and M. Bramer (Springer US, Boston, MA, 2006) pp. 508–515.

[147] B. S. Elenbogen and N. Seliya, *Detecting outsourced student programming assignments,* J. Comput. Sci. Coll. **23**, 50 (2008).

[148] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. Chaski, and B. Stephen H., *Identifying authorship by byte-level n-grams: The source code author profile (scap) method.* IJDE **6** (2007).

[149] I. Krsul and E. Spafford, *Authorship analysis: Identifying the author of a program,* Computers and Security **16**, 233 (1997).

[150] N. Rosenblum, B. P. Miller, and X. Zhu, *Recovering the toolchain provenance of binary code,* in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11 (ACM, New York, NY, USA, 2011) pp. 100–110.

[151] L. Simko, L. Zettlemoyer, and T. Kohno, *Recognizing and imitating programmer style: Adversaries in program authorship attribution,* Proceedings on Privacy Enhancing Technologies **2018**, 127 (2018).

[152] M. Shevertalov, J. Kothari, E. Stehle, and S. Mancoridis, *On the use of discretized source code metrics for author identification,* in *2009 1st International Symposium on Search Based Software Engineering* (2009) pp. 69–78.

[153] N. Novielli, D. Girardi, and F. Lanubile, *A benchmark study on sentiment analysis for software engineering research,* in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)* (IEEE, 2018) pp. 364–375.

[154] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, *Detecting code smells using machine learning techniques: are we there yet?* in *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (IEEE, 2018) pp. 612–621.

[155] S. Burrows, A. Uitdenbogerd, and A. Turpin, *Comparing techniques for authorship attribution of source code,* Software: Practice and Experience **44** (2014), 10.1002/spe.2146.

[156] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhanova, and A. Matyukhina, *Code authorship attribution: Methods and challenges,* ACM Comput. Surv. **52**, 3:1 (2019).

[157] J. Kennedy and R. Eberhart, *Particle swarm optimization,* in *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4 (1995) pp. 1942–1948 vol.4.

[158] D. Perez and S. Chiba, *Cross-language clone detection by learning over abstract syntax trees,* in *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19 (IEEE Press, Piscataway, NJ, USA, 2019) pp. 518–528.

[159] N. Rosenblum, X. Zhu, and B. P. Miller, *Who wrote this code? identifying the authors of program binaries,* in *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11 (Springer-Verlag, Berlin, Heidelberg, 2011) pp. 172–189.

[160] X. Meng, *Fine-grained binary code authorship identification,* in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016 (ACM, New York, NY, USA, 2016) pp. 1097–1099.

[161] T. M. Cover and J. A. Thomas, *Elements of Information Theory* (Wiley-Interscience, USA, 1991).

[162] C. E. Shannon, *A mathematical theory of communication,* Bell System Technical Journal **27**, 379 (1948), https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1948.tb01338.x .

[163] F. Wilcoxon, *Individual comparisons by ranking methods,* Biometrics Bulletin **1**, 80 (1945).

[164] F. Pedregosa, G. e. l. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and . E. d. Duchesnay, *Scikit-learn: Machine learning in python,* J. Mach. Learn. Res. **12**, 2825 (2011).

[165] K. J. Ottenstein, *An algorithmic approach to the detection and prevention of plagiarism,* SIGCSE Bull. **8**, 30 (1976).

[166] C. Liu, C. Chen, J. Han, and P. S. Yu, *Gplag: Detection of software plagiarism by program dependence graph analysis,* in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06 (ACM, New York, NY, USA, 2006) pp. 872–881.

[167] R. Layton, P. Watters, and R. Dazeley, *Automatically determining phishing campaigns using the uscap methodology,* in *2010 eCrime Researchers Summit* (2010) pp. 1–8.

[168] *Google style guides,* (2020).

[169] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, *Open source-style collaborative development practices in commercial projects using github,* in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1 (2015) pp. 574–585.

[170] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, *Temporally robust software features for authorship attribution,* in *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 1 (2009) pp. 599–606.

[171] R. C. Lange and S. Mancoridis, *Using code metric histograms and genetic algorithms to perform author identification for software forensics,* in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07 (ACM, New York, NY, USA, 2007) pp. 2082–2089.

[172] .

[173] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, *Learning natural coding conventions,* in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014 (ACM, New York, NY, USA, 2014) pp. 281–293.

[174] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, *The promises and perils of mining github,* in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014 (ACM, New York, NY, USA, 2014) pp. 92–101.

[175] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek, *On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github,* IEEE Transactions on Software Engineering , 1 (2018).

[176] 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*, MSR '06 (ACM, New York, NY, USA, 2006) pp. 137–143.

[177] G. Robles and J. M. Gonzalez-Barahona, *Developer identification methods for integrated data from various sources,* SIGSOFT Softw. Eng. Notes **30**, 1 (2005).

[178] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, *Who's who in gnome: Using lsa to merge software repository identities,* in *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012) pp. 592–595.

[179] I. S. Wiese, J. T. Da Silva, I. Steinmacher, C. Treude, and M. A. Gerosa, *Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant,* in *2016 IEEE international conference on software maintenance and evolution (ICSME)* (IEEE, 2016) pp. 345–355.

[180] T. Fry, T. Dey, A. Karnauch, and A. Mockus, *A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits,* in *Proceedings of the 17th International Conference on Mining Software Repositories* (2020) pp. 518–522.

[181] S. Amreen, A. Mockus, R. Zaretzki, C. Bogart, and Y. Zhang, *Alfaa: Active learning fingerprint based anti-aliasing for correcting developer identity errors in version control systems,* Empirical Software Engineering **25**, 1136 (2020).

[182] *IntelliJ Community,* (2020).

[183] *JetBrains Research GitHub: Authorship detection,* (2020).

[184] *ML in Programming GitHub: PSO authorship detection,* (2020).

[185] *Gradle,* (2020).

[186] *Mule,* (2020).

[187] G. Gousios, M. Pinzger, and A. v. Deursen, *An exploratory study of the pull-based software development model,* in *Proceedings of the 36th International Conference on Software Engineering*, ICSE (ACM, New York, NY, USA, 2014) pp. 345–355.

[188] A. Bacchelli and C. Bird, *Expectations, outcomes, and challenges of modern code review,* in *Proceedings of the 2013 international conference on software engineering* (IEEE Press, 2013) pp. 712–721.

[189] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, *Contemporary peer review in action: Lessons from open source development,* IEEE Software **29**, 56 (2012).

[190] M. B. Zanjani, H. Kagdi, and C. Bird, *Automatically recommending peer reviewers in modern code review,* IEEE Transactions on Software Engineering **42**, 530 (2016).

[191] A. Ouni, R. G. Kula, and K. Inoue, *Search-based peer reviewers recommendation in modern code review,* in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on* (IEEE, 2016) pp. 367–377.

[192] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, *Using static analysis to find bugs,* IEEE software **25** (2008).

[193] *Code review, project analytics, and team collaboration – features : Upsource,* `https://www.jetbrains.com/upsource/features/` (2018), accessed: 2018-02-08.

[194] *Crucible – features : Atlassian,* `https://www.atlassian.com/software/crucible/features` (2018), accessed: 2018-02-08.

[195] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, *Investigating code review quality: Do people and participation matter?* in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on* (IEEE, 2015) pp. 111–120.

[196] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl, *Evaluating collaborative filtering recommender systems,* ACM Transactions on Information Systems (TOIS) **22**, 5 (2004).

[197] A. Begel and B. Simon, *Novice software developers, all over again,* in *Proceedings of the Fourth international Workshop on Computing Education Research* (ACM, 2008) pp. 3–14.

[198] T. Baum and K. Schneider, *On the need for a new generation of code review tools,* in *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway, November 22-24, 2016, Proceedings 17* (Springer, 2016) pp. 301–308.

[199] M. Ge, C. Delgado-Battenfield, and D. Jannach, *Beyond accuracy: Evaluating recommender systems by coverage and serendipity,* in *Recommender Systems* (2010).

[200] S. M. McNee, J. Riedl, and J. A. Konstan, *Being accurate is not enough: how accuracy metrics have hurt recommender systems,* in *CHI'06 extended abstracts on Human factors in computing systems* (ACM, 2006) pp. 1097–1101.

[201] *Choosing reviewers – atlassian documentation,* https://confluence.atlassian.com/crucible/choosing-reviewers-298977465.html (2018), accessed: 2018-02-08.

[202] *Requesting a code review – help : Upsource,* https://www.jetbrains.com/help/upsource/codereview-author.html (2018), accessed: 2018-07-06.

[203] *Jetbrains: Developer tools for professionals and teams,* https://www.jetbrains.com/ (2018), accessed: 2018-02-08.

[204] *Microsoft – official home page,* https://www.microsoft.com/en-us/ (2018), accessed: 2018-02-08.

[205] P. C. Rigby and C. Bird, *Convergent contemporary software peer review practices,* in *Proceedings of ESEC/FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, ESEC/FSE 2013 (ACM, 2013) pp. 202–212.

[206] *Code quality – github marketplace,* https://github.com/marketplace/category/code-quality (2018), accessed: 2018-02-08.

[207] *Repository statistics,* https://developer.github.com/changes/2013-05-06-repository-stats/, accessed: 2018-02-08.

[208] K. Wei, J. Huang, and S. Fu, *A survey of e-commerce recommender systems,* in *Service systems and service management, 2007 international conference on* (IEEE, 2007) pp. 1–5.

[209] K. McNally, M. P. O'Mahony, M. Coyle, P. Briggs, and B. Smyth, *A case study of collaboration and reputation in social web search,* ACM Transactions on Intelligent Systems and Technology (TIST) **3**, 4 (2011).

[210] A.-T. Ji, C. Yeon, H.-N. Kim, and G.-S. Jo, *Collaborative tagging in recommender systems,* AI 2007: Advances in Artificial Intelligence , 377 (2007).

[211] W. Carrer-Neto, M. í. a. L. Hern á ndez Alcaraz, R. Valencia-Garc í a, and F. Garc í a-S á nchez, *Social knowledge-based recommender system. application to the movies domain,* Expert Systems with applications **39**, 10990 (2012).

[212] S. K. Lee, Y. H. Cho, and S. H. Kim, *Collaborative filtering with ordinal scale-based implicit ratings for mobile music recommendations,* Information Sciences **180**, 2142 (2010).

[213] Z. Yu, X. Zhou, Y. Hao, and J. Gu, *Tv program recommendation for multiple viewers based on user profile merging,* User modeling and user-adapted interaction **16**, 63 (2006).

[214] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, *Expert recommendation with usage expertise,* in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* (IEEE, 2009) pp. 535–538.

[215] E. Davoodi, M. Afsharchi, and K. Kianmehr, *A social network-based approach to expert recommendation system,* Hybrid Artificial Intelligent Systems , 91 (2012).

[216] J. Anvik, L. Hiew, and G. C. Murphy, *Who should fix this bug?* in *Proceedings of the 28th international conference on Software engineering* (ACM, 2006) pp. 361–370.

[217] J. Zhou, H. Zhang, and D. Lo, *Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports,* in *Proceedings of the 34th International Conference on Software Engineering* (IEEE Press, 2012) pp. 14–24.

[218] R. Venkataramani, A. Gupta, A. Asadullah, B. Muddu, and V. Bhat, *Discovery of technical expertise from open source code repositories,* in *Proceedings of the 22nd International Conference on World Wide Web* (ACM, 2013) pp. 97–98.

[219] V. W. Anelli, V. Bellini, T. Di Noia, W. La Bruna, P. Tomeo, and E. Di Sciascio, *An analysis on time-and session-aware diversification in recommender systems,* in *Proceedings of the 25th Conference on User Modeling, Adaptation and Personalization* (ACM, 2017) pp. 270–274.

[220] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, *Grouplens: An open architecture for collaborative filtering of netnews,* in *CSCW '94, Proceedings of the Conference on Computer Supported Cooperative Work, Chapel Hill, NC, USA, October 22-26, 1994* (1994) pp. 175–186.

[221] Y. Koren and R. M. Bell, *Advances in collaborative filtering,* in *Recommender Systems Handbook* (Boston, MA, USA: Springer, 2015) pp. 77–118.

[222] B. Smyth, *Case-based recommendation,* in *The Adaptive Web, Methods and Strategies of Web Personalization* (Berlin, Heidelberg: Springer, 2007) pp. 342–376.

[223] S. Chang, F. M. Harper, and L. G. Terveen, *Crowd-based personalized natural language explanations for recommendations,* in *Proceedings of the 10th ACM Conference on Recommender Systems* (ACM, 2016) pp. 175–182.

[224] J. L. Herlocker, J. A. Konstan, L. Terveen, and J. T. Riedl, *Evaluating collaborative filtering recommender systems,* ACM Trans. Inf. Syst. **22**, 5 (2004).

[225] B. Smyth and P. McClave, *Similarity vs. diversity,* Case-Based Reasoning Research and Development , 347 (2001).

[226] L. McGinty and B. Smyth, *On the role of diversity in conversational recommender systems,* in *International Conference on Case-Based Reasoning* (Springer, 2003) pp. 276–290.

[227] K. McCarthy, J. Reilly, L. McGinty, and B. Smyth, *An analysis of critique diversity in case-based recommendation,* in *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference, Clearwater Beach, Florida, USA*, edited by I. Russell and Z. Markov (AAAI Press, 2005) pp. 123–128.

[228] *Workshop on Novelty and Diversity in Recommender Systems (DiveRS 2011)* (2011).

[229] L. Iaquinta, M. de Gemmis, P. Lops, G. Semeraro, M. Filannino, and P. Molino, *Introducing serendipity in a content-based recommender system,* in *Hybrid Intelligent Systems* (2008).

[230] B. P. Knijnenburg, M. C. Willemsen, Z. Gantner, H. Soncu, and C. Newell, *Explaining the user experience of recommender systems,* User Modeling and User-Adapted Interaction **22**, 441 (2012).

[231] P. Pu, B. Faltings, L. Chen, J. Zhang, and P. Viappiani, *Usability guidelines for product recommenders based on example critiquing research,* in *Recommender Systems Handbook* (Springer, 2011) pp. 511–545.

[232] N. Tintarev and J. Masthoff, *Explaining recommendations: Design and evaluation,* in *Recommender Systems Handbook*, edited by F. Ricci, L. Rokach, and B. Shapira (Springer, Berlin, 2015) 2nd ed.

[233] T. T. Nguyen, P.-M. Hui, F. M. Harper, L. Terveen, and J. A. Konstan, *Exploring the filter bubble: the effect of using recommender systems on content diversity,* in *Proceedings of the 23rd international conference on World wide web* (ACM, 2014) pp. 677–686.

[234] E. Bakshy, S. Messing, and L. A. Adamic, *Exposure to ideologically diverse news and opinion on facebook,* Science **348**, 1130 (2015).

[235] A. M. Rashid, I. Albert, D. Cosley, S. K. Lam, S. M. McNee, J. A. Konstan, and J. Riedl, *Getting to know you: learning new user preferences in recommender systems,* in *Proceedings of the 7th international conference on Intelligent user interfaces* (ACM, 2002) pp. 127–134.

[236] D. Fleder and K. Hosanagar, *Blockbuster culture's next rise or fall: The impact of recommender systems on sales diversity,* Management science **55**, 697 (2009).

[237] E. Rader and R. Gray, *Understanding user beliefs about algorithmic curation in the facebook news feed,* in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (ACM, 2015) pp. 173–182.

[238] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, *Improving code review by predicting reviewers and acceptance of patches,* Research on Software Analysis for Error-free Computing Center Tech-Memo **6**, 1 (2009).

[239] D. Cosley, S. K. Lam, I. Albert, J. A. Konstan, and J. Riedl, *Is seeing believing?: how recommender system interfaces affect users' opinions,* in *Proceedings of the SIGCHI conference on Human factors in computing systems* (ACM, 2003) pp. 585–592.

[240] J. W. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*, 3rd ed. (Sage Publications, 2009).

[241] C. Hannebauer, M. Patalas, S. St ü nkelt, and V. Gruhn, *Automatically recommending code reviewers based on their expertise: An empirical comparison,* in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on* (IEEE, 2016) pp. 99–110.

[242] P. Jaccard, *É tude comparative de la distribution florale dans une portion des alpes et des jura,* Bull Soc Vaudoise Sci Nat **37**, 547 (1901).

[243] A. Furnham, *Response bias, social desirability and dissimulation,* Personality and individual differences **7**, 385 (1986).

[244] B. Taylor and T. Lindlof, *Qualitative communication research methods* (Sage Publications, Incorporated, 2010).

[245] R. Weiss, *Learning from strangers: The art and method of qualitative interview studies* (Simon and Schuster, 1995).

[246] B. Glaser, *Doing Grounded Theory: Issues and Discussions* (Sociology Press, 1998).

[247] B. Kitchenham and S. Pfleeger, *Personal opinion surveys,* Guide to Advanced Empirical Software Engineering , 63 (2008).

[248] P. Tyagi, *The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople,* Journal of the Academy of Marketing Science **17**, 235 (1989).

[249] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann, *Improving Developer Participation Rates in Surveys,* in *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering* (IEEE, 2013).

[250] T. Punter, M. Ciolkowski, B. Freimut, and I. John, *Conducting on-line surveys in software engineering,* in *International Symposium on Empirical Software Engineering* (IEEE, 2003).

[251] W. Lidwell, K. Holden, and J. Butler, *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design* (Rockport Pub, 2010).

[252] J. Cohen, *A coefficient of agreement for nominal scales,* Educational and psychological measurement **20**, 37 (1960).

[253] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, *An exploratory study on confusion in code reviews,* Empirical Software Engineering **26**, 1 (2021).

[254] S. Haiduc, J. Aponte, and A. Marcus, *Supporting program comprehension with source code summarization,* in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (ACM, 2010) pp. 223–226.

[255] M. Allamanis, H. Peng, and C. Sutton, *A convolutional attention network for extreme summarization of source code,* in *International Conference on Machine Learning* (2016) pp. 2091–2100.

[256] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, *Robustfill: Neural program learning under noisy i/o,* in *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (JMLR. org, 2017) pp. 990–998.

[257] A. Bacchelli and C. Bird, *Expectations, outcomes, and challenges of modern code review,* in *Proceedings of the 2013 international conference on software engineering* (IEEE Press, 2013) pp. 712–721.

[258] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, *Using software repositories to investigate socio-technical congruence in development projects,* in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)* (IEEE, 2007) pp. 25–25.

[259] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate,* arXiv preprint arXiv:1409.0473 (2014).

[260] *Intellij idea: The java ide for professional developers by jetbrains,* `https://www.jetbrains.com/idea/` (2019), accessed: 2019-12-27.

[261] *Eclipse desktop & web ides : The eclipse foundation,* `https://www.eclipse.org/ide/` (2019), accessed: 2019-12-27.

[262] *Program structure interface (psi),* `https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html` (2019), accessed: 2019-12-26.

[263] *Eclipse java development tools (jdt) : The eclipse foundation,* `https://www.eclipse.org/jdt/` (2019), accessed: 2019-12-27.

[264] *Eclipse cdt : The eclipse foundation,* `https://www.eclipse.org/cdt/` (2019), accessed: 2019-12-27.

[265] J. . R. é my Falleri, F. é al Morandat, X. Blanc, M. Martinez, and M. Monperrus, *Fine-grained and accurate source code differencing,* in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014* (2014) pp. 313–324.

[266] H. M. Tran, G. Chulkov, and J. ü. r. Sch ö nw ä lder, *Crawling bug tracker for semantic bug search,* in *International Workshop on Distributed Systems: Operations and Management* (Springer, 2008) pp. 55–68.

[267] M. Cataldo, J. D. Herbsleb, and K. M. Carley, *Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity,* in *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (ACM, 2008) pp. 2–11.

[268] R. C. Lange and S. Mancoridis, *Using code metric histograms and genetic algorithms to perform author identification for software forensics,* in *Proceedings of the 9th annual conference on Genetic and evolutionary computation* (2007) pp. 2082–2089.

[269] D. Azcona, P. Arora, I.-H. Hsiao, and A. Smeaton, *user2code2vec: Embeddings for profiling students based on distributional representations of source code,* in *Proceedings of the 9th International Conference on Learning Analytics & Knowledge* (ACM, 2019) pp. 86–95.

[270] D. Cui, *Early detection of flawed structural dependencies during software evolution,* IEEE Access (2021).

[271] J. R. Barr, P. Shaw, F. N. Abu-Khzam, S. Yu, H. Yin, and T. Thatcher, *Combinatorial code classification & vulnerability rating,* in *2020 Second International Conference on Transdisciplinary AI (TransAI)* (IEEE, 2020) pp. 80–83.

[272] V. Garousi, K. Petersen, and B. Ozkan, *Challenges and best practices in industry-academia collaborations in software engineering: A systematic literature review,* Information and Software Technology **79**, 106 (2016).