2012-03-02

Analysis and Characterization of Author Contribution Patterns in Open Source Software Development

Quinn Carlson Taylor
Brigham Young University - Provo

Analysis and Characterization of Author Contribution Patterns in Open Source Software Development

Quinn C. Taylor

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Charles D. Knutson, Chair
Daniel Zappala
Bryan S. Morse

Department of Computer Science

Brigham Young University

April 2012

ABSTRACT

Analysis and Characterization of Author Contribution Patterns in
Open Source Software Development

Quinn C. Taylor
Department of Computer Science, BYU
Master of Science

Software development is a process fraught with unpredictability, in part because software is created by people. Human interactions add complexity to development processes, and collaborative development can become a liability if not properly understood and managed. Recent years have seen an increase in the use of data mining techniques on publicly-available repository data with the goal of improving software development processes, and by extension, software quality. In this thesis, we introduce the concept of *author entropy* as a metric for quantifying interaction and collaboration (both within individual files and across projects), present results from two empirical observational studies of open-source projects, identify and analyze authorship and collaboration patterns within source code, demonstrate techniques for visualizing authorship patterns, and propose avenues for further research.

Keywords: software engineering, open source, data mining, collaboration, authorship patterns, author entropy, SourceForge, Subversion, Eclipse, Git

Contents

List of Figures

Chapter 1

Introduction

Software is created by humans. This obvious truth is the core reason that software engineering is such an imprecise and nebulous activity, for practitioners and researchers alike. It stands in stark contrast with the deterministic nature of computers, which bring software to life by following instructions with unwavering exactness. Although computer science is a "hard science" ruled by theory and certainty, software development is decidedly a "soft science" with strong dependencies on human behavior and social structures. As an example, one need look no further than Conway's Law [Conway, 1968] to see such principles at play.

Despite more than 40 years of history and collective experience, software engineering is, on average, still mired in mediocrity. Few people can explain why some projects flourish and others fail. Fewer still can accurately predict timelines, completeness, or quality. Even with improved tools, languages, and methodologies freely available to all, there are still fabulous successes and shocking failures. Fred Brooks' observation, made 25 years ago, is as true as ever: "The gap between the best software engineering practice and the average practice is very wide—perhaps wider than in any other engineering discipline." [Brooks, 1987]

The "practice" Brooks refers to consists largely of software development processes. Since software is developed by multiple people within the context of social structures, successful processes must consider human tendencies. People-oriented software development is an old concept, pioneered in *The Psychology of Computer Programming* [Weinberg, 1971]; recently it has been called the most important key to successful software projects [McConnell, 2009].

Software engineering research has traditionally focused on solutions derived from "hard" data, such as source code analysis, defect identification and triage, static and dynamic testing, verification, etc. However, in the past few years there has been an increased focus on understanding and improving how software (and particularly its cost and quality) is shaped by humans and social behaviors. This shift is driven in part by the increase in public data made available by the proliferation of open source communities. The resulting abundance of software artifacts (including source code, defect records, mailing list communications, etc.) has removed many barriers that previously made large-scale analysis of multiple projects unfeasible for most researchers.

In recent years, a new research community focused on data mining and analysis of open source software has sprung up. Since 2004, the International Conference on Software Engineering (ICSE) has held a Working Conference on Mining Software Repositories (MSR). The original call for papers stated that MSR's purpose was "to use data stored in software repositories to further understanding of software development practices ... [and enable repositories to be] used by researchers to gain empirically based understanding of software development, and by software practitioners to predict and plan various aspects of their project" [Hassan et al., 2004]. Several other venues, including the International Conference on Open Source Systems (OSS, since 2005), the Workshop on Public Data about Software Development (WoPDaSD, 2006–2010, subsequently subsumed by OSS), and the International Workshop on Emerging Trends in FLOSS Research (FLOSS, 2007 and 2010) have also played an important role in shaping and advancing this new research domain.

The BYU SEQuOIA[1] lab, established in 2006 by Dr. Charles Knutson, is deeply involved in this research community. Our goal is to extract and distill insights about software development processes, and ultimately better understand the nature of organizations that produce truly exceptional software. Repository data is a critical piece of this puzzle which

[1]Software Engineering Quality: Observation, Insight, and Analysis — `http://sequoia.cs.byu.edu`

can provide a wealth of information about the software itself, the organization that built it, and the development processes used.

When we began research in this space, our first efforts were focused on discovery and evaluation of previous related work. Researchers have examined repository data in a variety of ways, many of them involving data mining in some capacity. We compiled our findings in a paper about data mining applications in software engineering. Chapter 2 is the full version of this paper, which was subsequently published in the International Journal of Data Analysis Techniques and Strategies as "Applications of Data Mining in Software Engineering" [Taylor et al., 2010]. This paper has two major parts: a survey of the ways in which data mining techniques are (or have been) used in the field of software engineering, and an articulation of the ways in which data mining can be effectively leveraged to make significant contributions to understanding and improving software development efforts.

Very few of the repository mining tools we identified supported either numerical or visual analysis of author collaboration patterns, and those that did (see Section 3.4) were fairly limited in their approach. However, human-centric data is a critical part of accurately understanding software development processes. In addition, our lab's research is also shaped by three closely related ideas: 1) "software is not a product, but rather a medium for the storage of knowledge" [Armour, 2000b]; 2) "the product is the knowledge that goes into the software" [Armour, 2000a]; and 3) "software development is not a product-producing activity—it is a knowledge-acquiring activity" [Armour, 2000a]. Software artifacts and repository data, then, can be viewed as byproducts of knowledge acquisition, and hence as artifacts of an intrinsically human activity.

Studying how people contribute knowledge to software (and manipulate it thereafter) can improve our understanding of how the knowledge stored in software artifacts appears and evolves. However, it is non-trivial to measure or quantify software-borne knowledge, and virtually impossible to determine who "knows" what. Software repositories do allow

us to determine who contributed what, but in its raw form, this information is difficult to understand or correlate with other data.

Our solution is to create a new metric called *author entropy*, which blends concepts drawn from machine learning and data mining. This metric enables us to characterize the degree of collaboration between multiple authors. We performed a proof-of-concept study (on 28,955 files from 33 projects) which used author attribution data mined from SourceForge repositories to analyze authorship patterns over time. We observed an exponential decay in the number of files with a given number of authors as well as several recurrent patterns. Chapter 3 was published and presented at the 3rd International Workshop on Public Data about Software Development as "Author Entropy: A Metric for Characterization of Software Authorship Patterns" [Taylor et al., 2008]. This paper contributed directly to related SEQuOIA lab publications, which both expanded on this metric [Casebolt et al., 2009] and derived a related metric called *language entropy*, or the "distribution of an individual's development efforts across multiple programming languages" [Krein et al., 2009, 2010, MacLean et al., 2010].

This foundational research led to additional questions related to author contribution patterns, such as the relationships between collaboration activity and file size, author count, and project size. We performed a larger-scale study (on 251,633 files from 592 projects) to answer these questions, as well as to replicate our earlier work on author entropy. This study both revisited author entropy and added detailed statistical analysis of file sizes, collaboration within files and projects, and common authorship patterns. Chapter 4 contains the results of this research, published and presented at the 7th International Conference on Open Source Systems as "An Analysis of Author Contribution Patterns in Eclipse Foundation Project Source Code" [Taylor et al., 2011].

Key findings from this second paper include: 1) an approximately normal distribution of file sizes; 2) an exponential (log-linear) decay in the frequency of files with n authors; 3) a positive correlation between file size and number of authors; 4) a majority of source files that have a clearly dominant author who controls most of the file; and 5) a disproportionate number

of files which have only two authors, one of whom contributed only 1 or 2 of the lines. Some of these results were in line with our expectations, and others suggest potentially interesting new patterns we had not previously considered. As researchers, we seek to understand how people work together to develop complex systems, and to explain success or failure based on the data at our disposal. The results of these studies have suggested several new avenues of research which can help enhance our understanding.

The work contained in this thesis addresses an important aspect of software engineering, for both researchers and practitioners: leveraging software authorship data to learn from and improve software processes. We identify promising veins of future research, particularly opportunities for correlating authorship patterns (including author entropy) with other measurements of code ownership, analyses of quality, etc. We believe that future research, building on the findings presented here, has the potential to enhance our collective ability to understand and account for human factors that affect software development, and in turn, raise software quality and lower costs.

Chapter 2

Applications of Data Mining in Software Engineering

2.1 Introduction

Software systems are inherently complex and difficult to conceptualize. This complexity, compounded by intricate dependencies and disparate programming paradigms, slows development and maintenance activities, leads to faults and defects, and ultimately increases the cost of software. Most software development organizations develop some sort of processes to manage software development activities. However, as in most other areas of business, software processes are often based only on hunches or anecdotal experience, rather than on empirical data.

Consequently, many organizations are "flying blind" without fully understanding the impact of their process on the quality of the software that they produce. This is generally not due to apathy about quality, but rather to the difficulty inherent in discovery and measurement. Software quality is not simply a function of lines of code, bug count, number of developers, man-hours, money, or previous experience—although it involves all those things—and it is never the same for any two organizations.

Software metrics have long been a standard tool for assessing quality of software systems and the processes that produce them. However, there are pitfalls associated with the use of metrics. Managers often rely on metrics that they can easily obtain and understand, which may be worse than using no metrics at all. Metrics can seem interesting, yet be uninformative, irrelevant, invalid, or not actionable. Truly valuable metrics may be unavailable or difficult to

obtain. Metrics can be difficult to conceptualize, and changes in metrics can appear unrelated to changes in process.

Alternatively, software engineering activities generate a vast amount of data that, if harnessed properly through data mining techniques, can help provide insight into many parts of software development processes. Although many processes are domain- and organization-specific, there are many common tasks which can benefit from such insight, and many common types of data which can be mined. Our purpose here is to bring software engineering to the attention of our community as an attractive testbed for data mining applications and to show how data mining can significantly contribute to software engineering research.

The paper is organized as follows. In section 2.2, we briefly discuss related work, pointing to surveys and venues dedicated to recent applications of data mining to software engineering. Section 2.3 describes the sources of software data available for mining and section 2.4 provides a brief, but broad, survey of current practices in this domain. Section 2.5 discusses issues specific to mining software engineering data and prerequisites for success. Finally, section 2.6 concludes the paper.

2.2 Related Work

Although the application of data mining to software engineering artifacts is relatively new, there are specific venues in which related papers are published, and authors that have created resources similar to this survey.

Perhaps the earliest survey of the use of data mining in software engineering is the 1999 Data & Analysis Center for Software (DACS) state-of-the-art report [Mendonca and Sunderhaft, 1999]. It consists of a thorough survey of data mining techniques, with emphasis on applications to software engineering, including a list of 55 data mining products with detailed descriptions of each product and summary information along a number of technical as well as process-dependent features.

Since then, and over the years, Xie has been compiling and maintaining an (almost exhaustive) online bibliography on mining software engineering data. He also presented tutorials on that subject at the *International Conference on Knowledge Discovery in Databases* in 2006, and at the *International Conference on Software Engineering* in 2007, 2008, and 2009 [e.g., see Xie et al., 2007]. Many of the publications we cite here are also included in Xie's bibliography and tutorials.

The *Mining Software Repositories (MSR) Workshop*, co-located with the *International Conference on Software Engineering*, was originally established in 2004. Papers published in MSR focus on many of the same issues we have discussed in this survey, and the goal of the workshops is to increase understanding of software development practices through data mining. Beyond tools and applications, topics include assessment of mining quality, models and meta-models, exchange formats, replicability and reusability, data integration, and visualization techniques.

Finally, Kagdi et al. [2007] have recently published a comprehensive survey of approaches for mining software repositories in the context of software evolution. Although their survey is narrower in scope than the overview given here, it has greater depth of analysis, presents a detailed taxonomy of software evolution data mining methodologies, and identifies a number of related research issues that require further investigation.

2.3 Software Engineering Data

The first step in the knowledge discovery process is to gain understanding about the data that is available and the business goals that drive the process. This is essential for software engineering data mining endeavors, because unavailability of data for mining is a factor that limits the questions which can be effectively answered.

In this section, we describe software engineering data that are available for data mining and analysis. Current software development processes involve several types of resources from which software-related artifacts can be obtained. Software "artifacts" are a product of

software development processes. Artifacts are generally lossy and thus cannot provide a full history or context, but they can help piece together understanding and provide further insight. There are many data sources in software engineering. In this paper, we focus only on four major groups and describe how they may be used for mining software engineering data.

First, the vast majority of collaborative software development organizations utilize revision control software[1] (e.g., CVS, Subversion, Git, etc.) to manage the ongoing development of digital assets that may be worked on by a team of people. Such systems maintain a historical record of each revision and allow users to access and revert to previous versions. By extension, this provides a way to analyze historical artifacts produced during software development, such as number of lines written, authors which wrote particular lines, or any number of common software metrics.

Second, most large organizations (and many smaller ones) also use a system for tracking software defects. Bug tracking software (such as Bugzilla, JIRA, FogBugz, etc.) associates bugs with meta-information (status, assignee, comments, dates and milestones, etc.) that can be mined to discover patterns in software development processes, including the time-to-fix, defect-prone components, problematic authors, etc. Some bug trackers are able to correlate defects with source code in a revision system.

Third, virtually all software development teams use some form of electronic communication (email, instant messaging, etc.) as part of collaborative development. (Communication in small teams may be primarily or exclusively verbal, but such cases are inconsequential from a data mining perspective.) Text mining techniques can be applied to archives of such communication to gain insight into development processes, bugs, and design decisions.

Fourth, software documentation and knowledge bases can be mined to provide further insight into software development processes. This approach is useful to organizations that use the same processes across multiple projects and want to examine a process in terms of overall

[1]Revision control is sometimes also identified by the acronyms VCS for version control system, and SCM for source control management.

effectiveness or fitness for a given project. Although knowledge bases may contain source code, this approach focuses primarily on retrieval of information from natural languages.

2.4 Mining Software Engineering Data: A Brief Survey

In this section, we give a technique-oriented overview of how traditional data mining techniques have been applied in the context of software engineering, followed by a more task-oriented view in which we show how software tasks in three broad groups can benefit from data mining.

2.4.1 Data Mining Techniques in Software Engineering

In this section, we discuss several data mining techniques and provide examples of ways they have been applied to software engineering data. Many of these techniques may be applied to software process improvement. We attempt to emphasize innovative and promising approaches and how they can benefit software organizations.

2.4.1.1 Association Rules and Frequent Patterns

Zimmermann et al. [2005] have developed the ROSE tool (Reengineering of Software Evolution) to help guide programmers in performing maintenance tasks. The goals of ROSE are to 1) suggest and predict likely changes, 2) prevent errors due to incomplete changes, and 3) detect coupling undetectable by program analysis. Similar to Amazon's system for recommending related items, they aim to provide guidance akin to "Programmers who changed these functions also changed. . . ." They use association rules to distinguish between change types in CVS and try to predict the most likely classification of a change-in-progress.

Livshits and Zimmermann [2005] collaborated to create DynaMine, an automated tool that analyzes code check-ins to discover application-specific coding patterns and identify violations which are likely to be errors. Their approach is based on a classic *a priori* algorithm, combined with pattern categorization and dynamic analysis. Their tool has been able to

detect previously unseen patterns and several pattern violations in studies of the Eclipse and jEdit projects.

Śliwerski et al. [2005] have used association rules to study the link between changes and fixes in CVS and Bugzilla data for Eclipse and Mozilla. Their approach is to identify *fix-inducing changes*, or those changes which cause a problem that must later be fixed. (Closely related are *fix-inducing fixes*, or bug "fixes" which require a subsequent fix-on-fix.) They identify several applications, including: characterization and filtering of problematic change properties, analysis of error-proneness, and prevention of fix-inducing changes by guiding programmers. Interestingly, they also find that the likelihood of a change being fix-inducing (problematic) is greatest on Fridays.

Wasylkowski et al. [2007] have done work in automated detection of anomalies in *object usage models*, which are collections of typical or "correct" usage composed of sequences of method calls, such as calling `hasNext()` before `next()` on an `Iterator` object. Their JADET tool learns and checks method call sequences from Java code patterns to deduce correct usage and identify anomalies. They test their approach on five large open-source programs and successfully identify previously unknown defects, as well as "code smells" that are subject to further scrutiny.

Weimer and Necula [2005] focus on improving the effectiveness of detecting software errors. They note that most verification tools require software specifications, the creation of which is difficult, time-consuming, and error-prone. Their algorithm learns specifications from observations of error handling, based on the premise that programs often make mistakes along exceptional control-flow paths even when they normally behave correctly. Tests which force a program into error control flows have proven effective. The focus is on learning rules of temporal safety (similar to [Wasylkowski et al., 2007]) and infer correct API usage. They test several existing Java programs and demonstrate improvements in discovery of specifications versus existing data mining techniques.

Christodorescu et al. [2007] explore a related technique: automatic construction of specifications consistent with malware by mining of execution patterns which are present in known malware and absent in benign programs. They seek to improve the current process of manually creating specifications that identify malevolent behavior from observations of known malware. Not only is the output of this technique usable by malware detection software, but also by security analysts seeking to understand malware.

2.4.1.2 Classification

Large software organizations frequently use bug tracking software to manage defects and correlate them with fixes. Bugs are assigned a severity and assigned to someone within the organization. Classification and assignment can sometimes be automated, but are often done by humans, especially when a bug is incorrectly filed by the reporter or the bug database. Anvik et al. [2006, 2005, 2006] have researched automatic classification of defects by severity ("triage"), and Čubranić and Murphy [2004] have studied methods for determining who should fix a bug. Both approaches use data mining and learning algorithms to determine which bugs are similar and how a specific bug should be classified.

Work by Kim and Ernst [2007] has focused on classification of warnings and errors, and specifically the ability to suggest to programmers which should be fixed first. Their motivations include the high false-positive rates and spurious warnings typical of automatic bug-finding tools. They present a history-based prioritization scheme that mines software change history data that tells if and when certain types of errors were fixed. The intuition is that categories of warnings that were fixed in previous software changes are likely to be important. They report significant improvements in prioritization accuracy over three existing tools.

Nainar et al. [2007] use statistical debugging methods together with dynamic code instrumentation and examination of the execution state of software. They expand on the use of simple predicates (such as branch choices and function return values) by adding compound

boolean predicates. They describe such predicates, how they may be measured, evaluation of predicate "interestingness", and pruning of uninteresting predicates. They show how their approach is robust to sparse random sampling typical of post-deployment statistical debugging, and provide empirical results to substantiate their research.

2.4.1.3 Clustering

Most applications of data mining clustering techniques to software engineering data relate to the discovery and localization of program failures.

Dickinson et al. [2001] examine data obtained from random execution sampling of instrumented code and focus on comparing procedures for filtering and selecting data, each of which involves a choice of a sampling strategy and a clustering metric. They find that for identifying failures in groups of execution traces, clustering procedures are more effective than simple random sampling; adaptive sampling from clusters was found to be the most effective sampling strategy. They also found that clustering metrics that give extra weight to unusual profile features were most effective.

Liu and Han [2006] present R-Proximity, a new failure proximity metric which pairs failing execution traces and regards them as similar if they suggest roughly the same fault location. They apply this new metric to failure traces for software systems that include an automated failure reporting component, such as Windows and Mozilla. These traces (which include related information like the stack trace) are created when a crash is detected, and (with the user's permission) are sent back to the developers of the software. Their approach improves on previous methods that group traces which exhibit similar behaviors (such as similar branch coverage) although the same fault may be triggered by different sets of conditions. They use an existing statistical debugging tool to automatically localize faults and better determine failure proximity.

2.4.1.4 Text Mining

Text mining is an area of data mining with extremely broad applicability. Rather than requiring data in a very specific format (e.g., numerical data, database entries, etc.), text mining seeks to discover previously unknown information from textual data. Because many artifacts in software engineering are text-based, there are many rich sources of data from which information may be extracted. We examine several current applications of text mining and their implications for software development processes.

Code duplication is a chronic problem which complicates maintenance and evolution of software systems. Ducasse et al. [1999] propose a visual approach which is language-independent, overcoming a major stumbling block of virtually all existing code duplication techniques. Although their approach requires no language-specific parsing, it is able to detect significant amounts of code duplication. This and other similar approaches help alleviate the established problems of code duplication—such as unsynchronized fixes, code bloat, architectural decay, and flawed inheritance and abstraction—which frequently contribute to diminished functionality or performance.

Duplication of bug reports is also common, especially in organizations with widespread or public-facing test and development activities. Runeson et al. [2007] have applied Natural Language Processing and text mining to bug databases to detect duplicates. They use standard methods such as tokenization, stemming, removal of stop words, and measures of set similarity to evaluate whether bug reports are in fact duplicates. Because text mining is computationally expensive, they also use temporal windowing to detect duplicates only within a certain period of time of the "master" record. A case study of Sony Ericsson bug data has yielded success rates between 40% and 66%.

Tan et al. [2007] have presented preliminary work that addresses an extremely common occurrence: inconsistencies between source code and inline comments. The authors observe that out-of-sync comments and code point to one of two problems: 1) bad code inconsistent with correct comments, or 2) bad comments inconsistent with correct code. The former

indicates existing bugs; the latter can "mislead programmers to introduce bugs in subsequent versions." However, differences between intent and implementation are difficult to detect automatically. The authors have created a tool (iComment) which combines natural language processing, machine learning, statistics and program analysis to automatically analyze comments and detect inconsistencies. Their tests on 4 large code bases achieved accuracy of 90.8%–100% and successfully detected a variety of such inconsistencies, due to both bad code and bad comments.

Locating code which implements specific functionality is important in software maintenance, but can be difficult, especially if the comments do not contain words relevant to the functionality. Chen et al. [2001] propose a novel approach for locating code segments by examining CVS comments, which they claim often describe the changed lines and functionality, and generally apply for many future versions. The comments can then be associated with the lines known to have been changed, enabling users to search for specific functionality based on occurrences of search terms. Obviously, the outcome depends on CVS comment quality.

Large software projects require a high degree of communication through both direct and indirect mediums. Bird et al. [2006] mine the text of email communications between contributors to open-source software. This approach allows them to detect and represent social networks that exist in the open-source community, characterize interactions between contributors, and identify roles such as "chatterers" and "changers". The in-degree and out-degree of email responses are analyzed, and communication is correlated with repository commit activity. These techniques were applied to the Apache mailing lists and were able to successfully construct networks of major contributors.

A very recent application of text mining is analysis of the lexicon (vocabulary) which programmers use in source code. While identifier names are meaningless to a compiler, they can be an important source of information for humans. Effective and accurate identifiers can reduce the time and effort required to understand and maintain code.

Antoniol et al. [2007] have examined the lexicon used during software evolution. Their research studies not only the objective quality of identifier choices, but also how the lexicon evolves over time. Evidence has been found to indicate that evolution of the lexicon is more constrained than overall program evolution, which they attribute to factors such as lack of advanced tool support for lexicon-related tasks.

2.4.2 Software Engineering Tasks That Benefit From Data Mining

In this section, we survey existing approaches which focus on improving effectiveness of tasks in three aspects of software engineering: 1) development, 2) management, and 3) research. Although not all of these approaches use techniques specific to data mining, outlining domain-specific theoretical and empirical research can help develop understanding of which tasks can be effectively targeted by data mining tools.

2.4.2.1 Development Tasks

Software development is inherently a creative process, and no two programs are the same. During the initial programming phase of a software project, it is difficult to accumulate enough relevant data to provide insights that can help guide development. However, as development progresses, programming effort transitions to maintenance and refactoring, which we discuss separately in this section. Debugging and software evolution are also discussed here.

Mens and Demeyer [2001] seek to identify effective ways of applying metrics to evolving software artifacts. They cite evolution as a key aspect of software development, and differentiate between *predictive* analysis and *retrospective* analysis, of which the latter is most common. They propose a taxonomy to classify code segments with respect to evolution: 1) *evolution-critical* (parts which must be evolved to improve software quality and structure, or refactored to counter the effects of software aging); 2) *evolution-prone* (unstable parts that are likely to be evolved, often because they correspond to highly volatile software requirements); and 3) *evolution-sensitive* (highly-coupled parts that can cause ripple effects when evolved).

Livshits and Zimmermann [2005] present a methodology for discovering common error patterns in software, which combines mining of revision histories with dynamic analysis, including correlation of method calls and bug fixes with revision check-ins. When applied to large systems with substantial histories, they have been able to uncover errors and discover new application-specific patterns. Often, the errors found with this approach were previously unknown.

A similar testing approach was proposed by Liblit et al. [2005] which uses a dynamic analysis algorithm to isolate defects through sampling of predicates during program execution. They explore how to simplify redundant predicates, deal with predicates that indicate more than one bug, and isolating multiple bugs at once. This work is contrasted with static analysis of software quality, an approach which is currently very popular in software engineering.

Shirabad et al. [2001] propose the use of inductive methods to extract relations to create Maintenance Relevance Relations, which indicate which files are relevant to each other; this is helpful in the context of program maintenance, and especially for legacy systems, in which it is often difficult to know what other pieces of code may be affected by a change. They show how this approach can reveal existing complex interconnections among files in a system, useful for comprehending both the files and their connections.

Zimmermann et al. [2005] propose a predictive variant of this approach; they elaborate a tool for detecting coupling and predicting likely further changes. Their goal is to infer and suggest likely changes based on changes made by a programmer, but also to prevent errors due to incomplete changes. They use association rules to create linkage between changes, and in some cases are able to reveal coupling that is undetectable with program analysis. Predictive power increases with historical context for existing software, although it is known that not all suggestions are valid even in the best case; they report potential changes for the user to evaluate rather than omitting valid change linkages.

Mockus et al. [1999] take an approach closest to pure data mining: analyzing changes to legacy code to promote good business decisions. They state that understanding and

quantification are vital since "[e]ach change to legacy software is expensive and risky but it also has potential for generating revenues [sic] because of desired new functionality or cost savings in future maintenance." They study a large software system at Lucent technologies, highlight driving forces of change (related to both cost and quality), and discuss how to make inferences using measures of change obtained from version control and change management systems.

2.4.2.2 Management Tasks

Hassan [2006] discusses ways in which software artifacts and historical data can be used to assist managers. He states: "Managers of large projects need to prevent the introduction of faults, ensure their quick discovery, and their immediate repair while ensuring that the software can evolve gracefully to handle new requirements by customers." Their summary paper addresses some challenges commonly faced by software managers (including bug prediction and resource allocation) and provides several possible solutions.

These issues tie closely with research from Mockus et al. [2003] that deals with predicting the amount and distribution of effort remaining to complete a project. They propose a predictive model based on the concept that each software modification may cause repairs at some later time, then use the model to predict and successfully plan development resource allocation for existing projects. This model is a novel way to investigate and predict effort and schedules, and the results they present also empirically confirm a relationship between new features and bug fixes.

Canfora and Cerulo [2005] discuss impact analysis, "the identification of the work products affected by a proposed change request, either a bug fix or a new feature request." They study open source project and extract change requests and related data from bug tracking systems and versioning systems to discover which source files would be impacted by a change request. Links from changes to impacted files in historical data and information retrieval algorithms are used in combination to derive sets of impacted files.

Atkins et al. [1999] attempt to quantify the effects of a software tool on developer effort. Software tools can improve software quality, but are expensive to acquire, deploy and maintain, especially in large organizations. They present a method for tool evaluation that correlates tool usage statistics with estimates of developer effort. Their approach is inexpensive, observational, non-intrusive in nature, and includes controls for confounding variables; the subsequent analysis allows managers to accurately quantify the impact of a tool on developer effort. Cost-benefit analyses provide empirical data (although possibly from dissimilar domains) that can influence decisions about investing in specific tools.

2.4.2.3 Research Tasks

Data mining from the perspective of a software engineering researcher is unique in that the goal is generally to gain understanding about a variety of projects in order to characterize patterns in software development, rather than understanding about a specific project to guide its development.

Researchers frequently analyze data from open-source projects, but as Howison and Crowston [2004] explain, mining data from organizations like Sourceforge.net is fraught with fundamental pitfalls such as dirty data and defunct projects. In addition, screening to control for potential problems introduces bias and skew, and the similarities of software in the open-source "ecosystem" can tempt researchers to create models which fit the training data but do not generalize to other development patterns or ecosystems.

Software evolution is a popular topic for software data miners. Ball et al. [1997] examine ways to better understand a program's development history through partitioning and clustering of version data. Gall and Lanza [2006] explores avenues for analysis, filtering, and visualization of software processes evolution. Identification of architectural decay and trends of logical coupling between unrelated files are also shown. Kagdi et al. [2006] take a similar approach that focuses on identifying sequences of changed files by imposing partial temporal

ordering on atomically-committed files, using heuristics such as time interval, committer, and change-sets.

Extraction and correlation of software contributors is another area of active research. Alonso et al. [2004] characterize the role of project participants based on rights to contribute. Newby et al. [2003] study contributions of open-source authors in the context of Lotka's Law [Lotka, 1926] (which relates to predicting the proportion of authors at different levels of productivity), while Zhang et al. [2007] focus on understanding individual developer performance.

Several research groups have worked to create tools to simplify collection and analysis of software artifacts and metrics, although some are more reusable than others.

One such available tool is GlueTheos, written by Robles et al. [2004], which is an all-in-one tool for collecting data from open-source software (OSS). Currently, its analysis and presentation options are somewhat limited, but its data input and storage architecture is designed for extensibility.

Scotto et al. [2006] have proposed an architecture which focuses on providing a non-invasive method for collection of metrics. Their approach leverages distributed and web-based metrics collection tools to aggregate information automatically with minimal interaction from users.

2.5 Mining Software Engineering Data: The Road from Here

Applications of data mining to various areas of software engineering—several of which have been discussed in this paper—will certainly continue to develop and provide new insights and benefits for software development processes. Regardless of the specific techniques, there are aspects of data mining that are increasingly important in the domain of software engineering.

In this section we discuss a few issues that can help increase the effectiveness and adoption of data mining, both in software engineering and in general.

2.5.1 Targeting Software Tasks Intelligently

Data mining is only as good as the results it produces. Its effectiveness may be constrained by the quantity or quality of available data, computational cost, stakeholder buy-in, or return on investment. Some data or tasks are difficult to mine, and "mining common sense" is a waste of effort, so choosing battles wisely is critical to the success of any data mining endeavor.

Automatable tasks are potentially valuable targets for data mining. Because software development is so human-oriented, people are generally the most valuable resources in a software organization. Anything that reduces menial workload requiring human interaction can free up those resources to perform other tasks which only humans can do.

For example, large organizations may benefit substantially from automation of bug report triage and assignment. Automatic analysis and reporting of defect detection, error patterns, and exception testing can be highly beneficial, and the costs of computing resources to accomplish these tasks are very reasonable. Text analysis of source code for duplication, out-of-sync comments and code, and localization of specific functionality could also be extremely valuable to maintenance engineers.

Data mining is most effective at finding new information in large amounts of data. Complex software processes will generally benefit more from data mining techniques than simpler, more lightweight processes that are already well-understood. However, information gained from large processes will also have more confounding factors and be more difficult to interpret and put into action. Changes to software process are not trivial, and the effects that result from changes are not always what one might expect.

Equally important to remember is the fact that data mining is not a panacea or "silver bullet" that improves software all by itself. Information gleaned from mining activities must be correctly analyzed and properly implemented if it is to change anything. Data mining can only answer questions that are effectively articulated and implemented, and good intentions can't rescue bad data (or no data).

Data miners and software development organizations wishing to employ data mining techniques should carefully consider the costs and benefits of mining their data. The cost to an organization—whether in man-hours, computing resources, or data preparation—must be low enough to be effective for a given application.

2.5.2 Lowering the Barrier of Entry

In order to make a difference in more areas of software engineering, data mining needs to be more accessible and easier to adapt to tasks of interest. There is a great need for tools which can automatically clean or filter data, a problem which is intractable in the general case but possible for specific domains where data is in a known format.

In addition to automated "software-aware" data mining tools, we see a need for research and tools aimed at simplifying the process of connecting data mining tools to common sources of software data, as discussed in Section 2.4. Currently, it is common for each new tool to re-implement problems which have already been solved by another tool, perhaps only because the solutions have not been published or generalized.

Because many data mining tasks (e.g., text mining) are extremely computationally expensive, replication of effort is a major concern. Tools that help simplify centralized extraction and caching of results will make widespread data mining more appealing to large software organizations; the same tools can make collaborative data mining research more effective. The ability to share data among colleagues or collaborators without replication amortizes the cost of even the most time-intensive operations. Removing the "do-it-all-yourself" requirement will open many possibilities.

Intuitive client-side analysis and visualization tools can help spur adoption among those responsible for applying newly-discovered information. Most current tools, although extremely powerful, are targeted at individuals with strong fundamental understanding of machine learning, statistics, databases, etc. A greater emphasis on creating approachable

tools for the layperson with interest in using mined data will increase the value (or at least an organization's perception of value) of the data itself.

2.5.3 A Word of Caution

Just as with any tool, data mining techniques can be used either well or poorly. As data mining techniques become more popular and widespread, there is a tendency to treat data mining as a hammer and any available data as a nail. If unchecked, this can be a significant drain on resources.

Software practitioners must carefully consider which, if any, data mining technique is appropriate for a given task. Despite the many commonalities in software development artifacts and data, no two organizations or software systems are identical. Because improvement depends on intelligent interpretation of information, and the information that can be obtained depends on the available data, knowledge of one's data is just as crucial in software development as it is in other domains. Thus, we reiterate that the first step is to understand what data is available, then decide whether that can provide useful insights, and if so, how to analyze it.

2.6 Summary

We have identified reasons why software engineering is a good fit for data mining, including the inherent complexity of development, pitfalls of raw metrics, and the difficulties of understanding software processes.

We discussed four main sources of software "artifact" data: 1) version control systems, 2) bug trackers, 3) electronic developer communication, and 4) documentation and knowledge bases. We presented three areas of software engineering tasks (development, management, and research) and provided examples of how tasks in each area have been addressed by software engineering researchers, both with data mining and other techniques.

We also discussed four broad data mining techniques (association rules and frequent patterns, classification, clustering, and text mining) and several instances of how each has been applied to software engineering data.

Finally, we have presented some suggestions for future directions in mining of software engineering data, and suggested that future research in this domain is likely to focus on increased automation and greater simplicity.

Chapter 3

Author Entropy: A Metric for Characterization of Software Authorship Patterns

3.1 Introduction

Software development is a process fraught with complexity and unpredictability because software is designed and written by people. Human interactions add complexity to development processes, although some software engineering authorities disagree about the implications [Brooks, 1975, Conway, 1968, Crowston and Howison, 2005].

Contributor interactions critically affect software development, and it follows that characterizing contributor interaction is an important task. Studies of developer interactions have generally focused on bug tracking, mailing list analysis, or studies of developer productivity; few consider developer interaction within source code.

In this paper, we introduce *author entropy*, a metric that quantifies the mixture of author contributions to a file. Just as code-level metrics—including file length, number of function points, complexity, cohesion, and coupling—quantify properties of source code, author entropy characterizes properties of author interactions within source files using a simple summary statistic.

This paper describes the author entropy metric, presents a proof of concept empirical study, and proposes topics for future research relating to author entropy and authorship patterns.

3.2 Author Entropy

In this section, we discuss entropy, define how entropy is calculated, and describe how entropy applies to authorship.

3.2.1 Definitions of Entropy

Entropy is a measure of chaos or disorder in a system. Thermodynamics defines entropy as a measure of randomness of molecules in a system; an increase in entropy leads to greater spontaneity. Entropy is often understood as the "useless energy" in a system: energy which is not available to perform work. The second law of thermodynamics states that the entropy of an isolated system will tend to increase over time, approaching a maximum value at equilibrium.

Information theory borrows the idea of entropy, defines it in terms of probability theory, and uses it to analyze communication, compression, information content, and uncertainty [Shannon, 1948]. In machine learning, entropy "characterizes the (im)purity of an arbitrary collection of examples", or the degree to which members of a collection can be split into groups based on a given attribute [Mitchell, 1997].

Entropy may also be applied to software engineering as a measure of collaboration. Specifically, we consider entropy of source code, which can be broken into smaller segments (e.g., lines, functions, statements, identifiers, etc.) and classified by author. This definition of entropy allows us to quantify the mixture of author contributions to a file. We discuss why this matters to software in Section 3.2.3.

3.2.2 Calculating Entropy

Entropy is a summary statistic[1] calculated from the relative sizes of the groups or classifications present in a system. Entropy formulae are nearly identical across domains, varying only in constant multipliers and symbolic representation.

3.2.2.1 The Special Case: Binary Classification

We first consider entropy in the special case of a Bernoulli distribution with proportion p of positive outcomes and proportion q of negative outcomes, where $0 \leq p \leq 1$. The entropy of system S (shown in Figure 3.1) is defined as:

$$E(S) \equiv -p \cdot log_2\ p\ - q \cdot log_2\ q \tag{3.1}$$

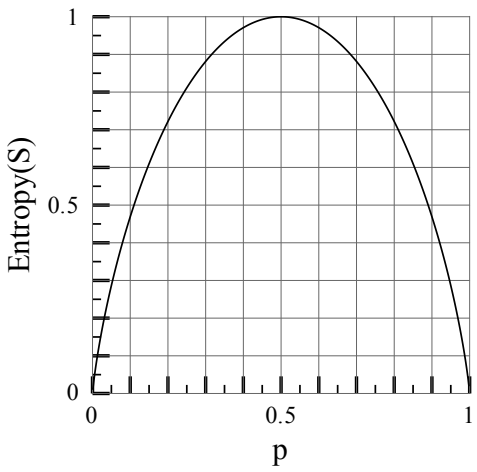


Figure 3.1: Entropy of a Bernoulli distribution.

Note that entropy is maximized when p and q are equal, and minimized when either proportion reaches 1.

[1]Summary statistics are lossy summaries of observations, such as mean, median, variance, skewness, and kurtosis.

3.2.2.2 The General Case: Any Number of Groups

Entropy also generalizes to an arbitrary number of groups. If elements of a system S belong to c different classes, and p_i is the proportion of elements in S belonging to class i, then the entropy of S is:

$$E(S) \equiv -\sum_{i=1}^{c}(p_i \cdot log_2\, p_i) \tag{3.2}$$

$E(S)$ is maximized when the proportions of classes in S are equal ($\forall\ i, p_i = \frac{1}{c}$). Equation 3.2 is a non-normalized summation, so the limit of $E(S)$ is a function of c. As shown in Figure 3.2, if the elements of a system S belong to c possible classes, the entropy can be as large as:

$$E_{max}(S) \equiv log_2\, c \tag{3.3}$$

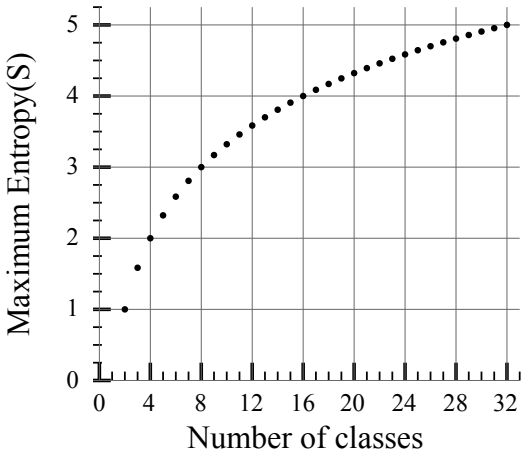


Figure 3.2: Maximum possible entropy for a system S as a discrete function of the total number of groups.

Because the maximum possible entropy for a system is a function of c, intuitive understanding of an entropy value can be difficult. For example, an entropy of 1 is the maximum entropy for a system with 2 classes, but comparatively low entropy for a system with 10 classes. Dividing $E(S)$ by $log_2\, c$ produces a value in the range [0,1]; this normalized

entropy value represents the percentage of maximum entropy, which may be more intuitive than non-normalized entropy. (See section Section 3.5.3 for normalization strategies.)

By definition, a system with only one class has zero entropy, so we define $log\,0$ to be 0. We also note that the logarithmic base is unrelated to the number of classifications; we use log_2 for historical reasons rooted in information theory.

3.2.2.3 Entropy Applied to Text Authorship

If system S is a file and c is the number of authors, then each p_i is the proportion of the text written by author i, and $E(S)$ is the entropy of the file. The values of p_i are the proportions of text segments attributed to each author. (Text segments may be of any size, although entropy may be more meaningful when the segments are roughly equivalent in size or information content.) Entropy increases as all authors' contributions (p_i) approach equality.

3.2.3 Interpretation of Entropy in Software

Entropy in source code is not inherently good or bad; it merely indicates that multiple people are contributing in a fairly balanced way. Although low entropy could be an indicator of modular team structure and well-architected software, it could also reflect poorly structured code that few contributors are willing to work on. Similarly, high entropy could be the result of poor communication or code that is in dire need of refactoring, or it may indicate excellent organization that makes it easy for many authors to contribute to the same code. As with any metric, context is essential.

Correlating entropy with other metrics and observations can provide valuable new insights. For example, a file with high entropy written by several experts may be of higher quality than a file written by one novice author; combining entropy with a metric of quality can help distinguish between "good entropy" and "bad entropy". Several ways to leverage the author entropy metric are discussed more in Section 3.5.1.

Author entropy cannot directly indicate attributes of the subject text. For example, file length is obscured since files of different size but equal proportions of contribution have the same entropy. Entropy also does not consider quality or the relative importance of contributions, such as new functionality, bug fixes, comments, whitespace, or formatting.

3.3 Proof of Concept Study

In order to give the reader a better understanding of how author entropy can be useful, we have conducted a small empirical study as a proof of concept that demonstrates possible applications of the metric. We begin by describing the methods and tools used to gather data and calculate author entropy. As part of that discussion we present the criteria we used to select projects for the study and the threats to the validity of our results. We also present some results of our preliminary study, including observations and analysis of authorship patterns manifest in the data.

3.3.1 Extraction and Calculation

Author entropy calculations require data that attributes text fragments to authors. Software authorship information can be gleaned from revision control systems that record snapshots of development history [Ball et al., 1997]. Our exploratory study considers only projects stored in Subversion.

We created a Python script to collect author data. We use Subversion's `log` command to identify the files modified in each revision, and record the revision number and path for each file. We then use Subversion's `blame` command to determine authorship for each line in each changed file. Author counts are divided by the total number of lines in the file to obtain p_i values, and author entropy for each file is calculated as shown in Equation 3.2. Entropy for each file is also normalized to the range [0,1] as in Equation 3.3.

3.3.2 Project Selection

Due to the amount of data which must be analyzed, we identified a subset of SourceForge projects with favorable characteristics. (We selected SourceForge.net because it hosts thousands of projects with multiple years of development history and various development platforms.) Howison and Crowston [2004] has identified potential weaknesses in this approach.

Many projects were not suitable for our analysis because they 1) were immature, abandoned, or not very active, 2) didn't use Subversion exclusively, 3) had very few developers, or 4) contained many non-source text files. We addressed these issues by limiting our sample to projects that meet the following criteria:

1. Projects categorized as "Production/Stable".
2. Projects registered since 2006 (higher SVN usage).
3. Projects with 5 or more committing developers.[2]
4. Projects in Java with easily identifiable source files.

We queried FLOSSmole [2004] data with these four criteria and identified 33 candidate projects, with the following distributions of revisions and authors:

	Min	Q1	Median	Q3	Max
Revisions	41	373	723	994	11576
Authors	5	6	8	13	23

Table 3.1: Distributions of revisions and authors for 33 projects selected from SourceForge.

3.3.3 Threats to Validity

One significant concern is the limited number of projects and the criteria used to select them. Although many open-source projects share similar development patterns, by no means should our results be construed as representative of all open-source projects, or even of all projects

[2]We scraped Subversion logs to determine the actual number of committing authors, instead of relying on the number of registered developers.

hosted on SourceForge. Many projects that did not fit our criteria would undoubtedly exhibit interesting authorship patterns.

Hidden factors which we have not addressed include irregularities in the historical data. For example, some projects contained anonymous commits, and many had a majority of commits from a single author. Without more in-depth study of specific projects, we cannot ascertain whether one developer in fact wrote all the code, or whether other contributors submitted patches that a single developer then committed. We also did not examine any specific changes to see whether changes in entropy were caused by source code reformatting, which artificially attributes lines to the committing author.

Our study is limited to line-level granularity provided by Subversion, and does not examine how much of a line changes.

With these threats to validity, however, it is important to reiterate that the focus of this paper is the author entropy metric itself. Our study is intended as a proof of concept, and should not be interpreted as exhaustive or complete. We describe potential avenues for future research in Section 3.5.

3.3.4 Results

In this section, we identify and offer possible explanations for patterns we observed in our study. These observations place author entropy in a real-world context; we demonstrate how changes to individual files affect entropy, characterize relationships between number of authors and entropy distribution, and identify project-wide entropy patterns.

3.3.4.1 Degree of Collaboration Within Files

For the projects in our sample, the maximum number of authors contributing to a project was 23, but there were no individual files with more than 9 authors. Figure 3.3 shows the counts of file revisions and unique files we observed with each number of authors, plotted on a logarithmic scale.

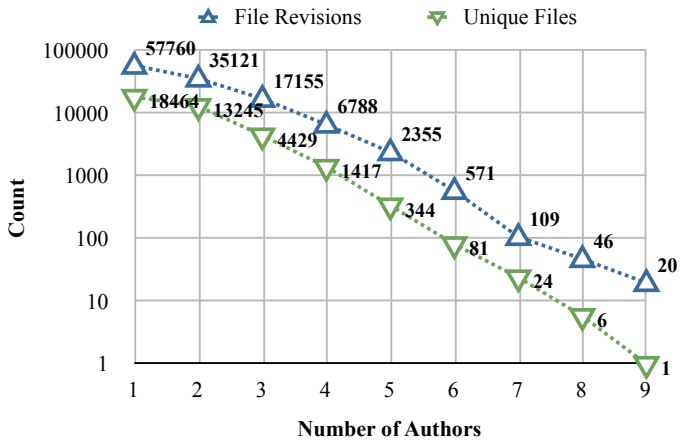


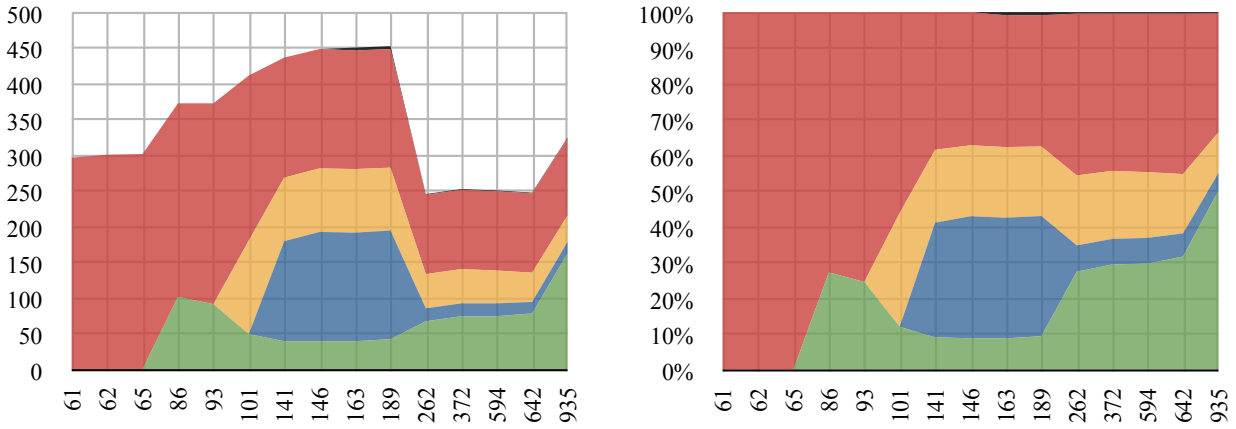
Figure 3.3: Counts of file revisions and unique files plotted against number of authors.

We found it noteworthy that the counts of file revisions and unique files with n authors are inversely proportional to n and exhibit near-perfect exponential decay. We hypothesize that communication and coordination become prohibitively expensive as the number of authors increases, and that these costs naturally discourage many authors from contributing on a file.

3.3.4.2 Entropy Patterns Within Files

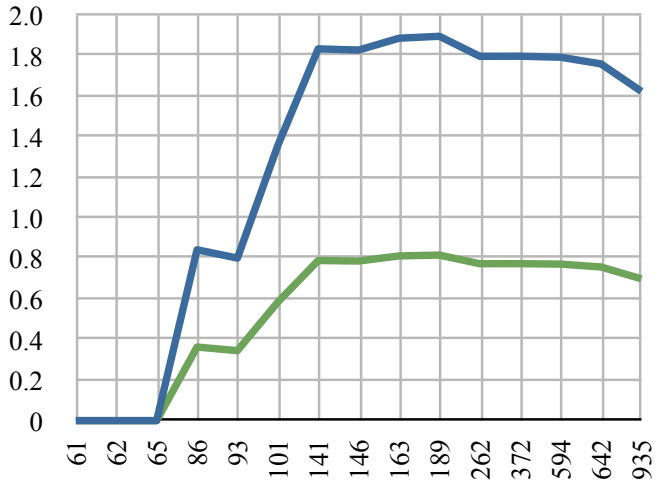
We also focused on fine-grained analysis of individual files to identify potentially interesting entropy patterns. We chose files that had high standard deviation of normalized entropy over multiple revisions (an indicator of significant changes) and compared author contributions (both the number and percent of total lines) to entropy and normalized entropy. The results for one such file are shown in Figure 3.4.

In Figure 3.4(c), the upper line is raw entropy and the lower line is normalized by $log_2 5$, since 5 is the maximum number of authors that ever contributed to the file. Normalized entropy plateaus at approximately 0.8 before decreasing slowly. Note that, despite a significant reduction in number of total lines at revision 262, author entropy does not drop rapidly. However, the addition of new authors at revisions 86, 101, and 141 does cause a significant increase in entropy.



(a) Number of total lines by author.

(b) Percentage of total lines by author.



(c) Entropy and normalized entropy.

Figure 3.4: Data for 15 revisions of `BookmarksHelper.java` in S3B. The x axis shows consecutive revisions; actual time periods between revisions is not represented here.

Because entropy calculations include logarithmic factors, entropy is very sensitive to small segments of text added by additional authors, but less sensitive to changes once an author is "established." Consider the two author case in Figure 3.1: 50% of maximum entropy is reached when one author contributes approximately 10% of the text. This bias makes entropy highly sensitive to initial changes by new authors.

3.3.4.3 Entropy Distributions Within Projects

Entropy is difficult to visualize for projects with many file revisions, so we created a histogram-based plot to display entropy distributions over a project's life. We found that using color rather than a 3D height map improved scale determination and trend exploration for large projects.

Although non-zero entropy often approximated a uniform distribution as projects progressed, several projects had patterns of generally high or low entropy, dramatic changes in entropy, and even "flip-flops" between high and low entropy.

(a) StoryTestIQ trends towards high entropy.

(b) Xendra trends towards low entropy.

(c) NakedObjects increases in entropy.

(d) SweetDEV RIA "flip-flops" between high and low entropy.

Figure 3.5: Excerpts of entropy distributions for several projects, excluding zero-entropy values. The darkness of each (x, y) point represents the percentage of files at revision x that map to normalized entropy y. These plots have 20 bins over the range of entropy values and have been contrast-adjusted for better readability.

Because the plots in Figure 3.5 are histograms, many files need to change before the histogram changes significantly. Dramatic shifts in entropy can occur when: 1) entropy shifts in a significant number of files or 2) a large number of files are added or removed. Development activities that may cause these shifts include: new authors contributing to existing files, refactoring, code formatting, or bug fixes.

3.3.4.4 Entropy Distributions Across Projects

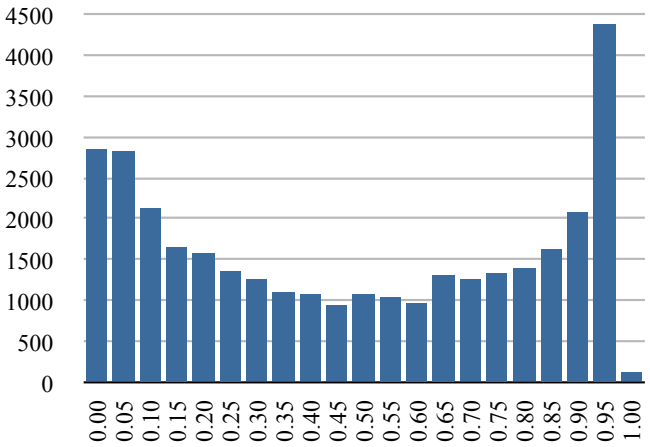
We examined the distribution of entropy as the number of authors for a file increases, shown in Figure 3.6. For $n = 2...9$ authors, we calculated univariate Gaussian kernel density estimators (a form of histogram smoothing) for normalized and non-normalized entropy values. We then combined each density function into a single 3D plot.

The entropy distribution for files with two authors was bimodal. Files were most likely to have either: 1) very low entropy, indicating that one author contributed only a very small portion of the file, or 2) very high entropy, indicating that both authors contributed almost equally. However, the entropy distributions for more than two authors were unimodal with a mean that increased with the number of authors.

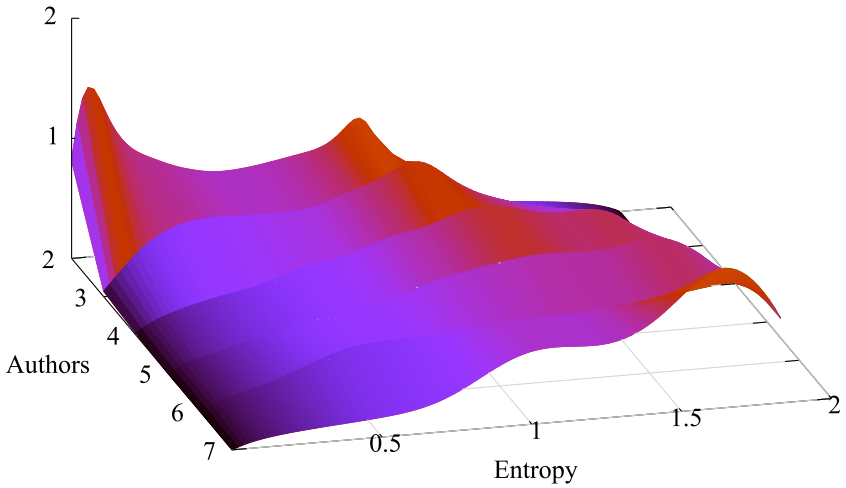
Normalized entropy for three or more authors displayed an interesting trend. As the number of authors increased, the distribution of normalized entropy remained fairly constant with a peak around 0.6. Although entropy increases as more authors are added, it remains proportional to maximum possible entropy. This may indicate hidden communication or social factors that naturally keep entropy around 60% of its maximum when more than two people contribute to a file.

3.3.5 Summary

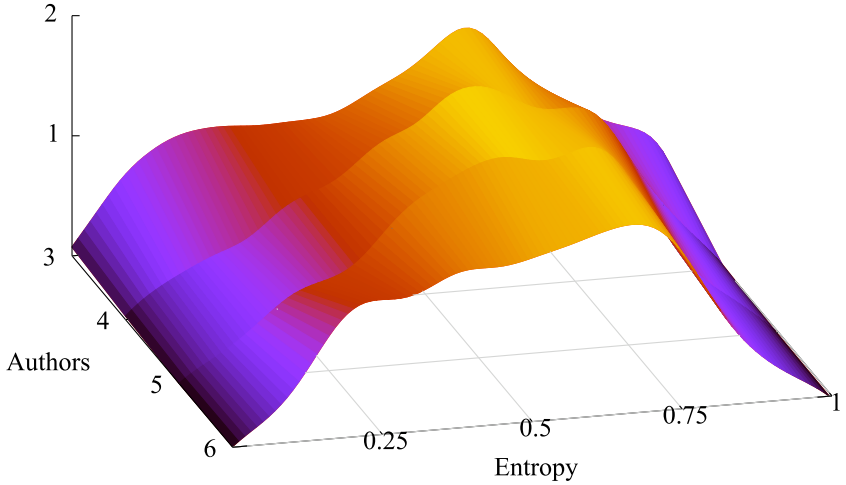
In this preliminary empirical study, we have identified several fine- and coarse-grained authorship patterns present in the projects we selected. We have observed a bimodal distribution of entropy for files with only two authors, but noted that the entropy distribution



(a) Entropy distribution, 2 authors.



(b) Entropy distributions, 2–7 authors.



(c) Normalized entropy, 3–6 authors.

Figure 3.6: Plots of entropy distributions for 28,955 files from 33 open-source projects.

for files with three or more authors is unimodal. We have also noticed significant shifts in project-wide entropy for some projects.

While explanation of the causes of these observations is beyond the scope of this paper, these patterns (and the questions they raise) suggest that author entropy is a potentially valuable metric for software engineering researchers.

3.4 Related Work

Several existing tools include functionality for visual or numerical analysis of authorship patterns.

CVSscan [Voinea et al., 2005] is a visualization tool for observing source code structure and evolution during software maintenance. It extracts data from CVS repositories and represents source code lines of one file at a time as a sequence of stacked lines that are colored according to author, age, or code construct.

StatSVN [2006] retrieves information from Subversion repositories and generates charts, tables and statistics which describe project development. A few of the statistics address authorship patterns (e.g. number of source lines contributed per author, commit activity) at the directory level.

CodeSaw [Gilbert and Karahalios, 2007] is a tool for visualizing author activity in distributed software development. It combines code activity and developer communication to reveal group dynamics. Data for up to eight authors may be visualized together on a timeline. Hovering over the timeline displays detailed information for the selected developer and time period.

Author entropy can be coupled with the functionality in these and other tools to provide additional context for understanding established software metrics and patterns. We explore several possible options and benefits in Section 3.5.1.

3.5 Future Work

Our initial research generated many questions about the implications of author entropy in software engineering and other domains. Although we cannot fully address these topics in this paper, we list several avenues for future research.

3.5.1 Empirical Evaluation of Applicability

We hypothesize that high author entropy may be correlated with existing software metrics, such as lines of code, high complexity, high coupling, low cohesion, high bug count, etc. However, no work has yet been done to empirically test such theories. Studies of author entropy in the context of Lotka's Law [Lotka, 1926] and exponential decay (similar to work done by Newby et al. [2003], but in the context of files rather than across projects) may also provide significant new insights and allow characterization of developer productivity, both in terms of quantity and significance of work.

Author entropy may also lead to new insights if combined with similar metrics in other fields, such as Gini's inequality coefficient [Gini, 1912]. For example, balanced project contribution can mitigate "bus factor" risks, but could also be detrimental to a team's efficiency and agility if taken to extremes.

A topic of special interest to us is analysis and visualization of relationships between author entropy and program structure. Mapping entropy values to a representation of a program's structure may reveal valuable information about its evolution, similar to the findings of Gall and Lanza [2006]. Author entropy could also be an effective indicator of design and behaviors which substantiate Conway's Law [Conway, 1968].

3.5.2 Aggregating Entropy for Groups of Files

In this paper we have calculated author entropy only for individual files. Entropy can also be calculated and analyzed for groups of files—such as packages, directories, modules, and projects—at any level of depth in a project hierarchy.

However, care must be taken as to how aggregated entropy is calculated. Two possible techniques we have identified are to 1) create an average or linear combination of file entropies, or 2) calculate entropy from a sum of author counts. Both techniques can pre-compute data to be used in calculations for later revisions, but the results can differ significantly between the two. Specifically, author entropy for a group of files with multiple authors should be non-zero, but the first approach does not always yield non-zero values.

For example, consider n files, each of which is written exclusively by a different author. The entropy for each individual file is zero, so any combination or average will always be zero. (In fact, when any of the files in a group have zero entropy, results from the first approach will be inaccurate.) The second approach represents overall author contributions more accurately, but does require that author counts for the most recent revision of each file in the group be stored.

Software developers are often unaware of exactly how much their programming efforts overlap with others. The ability to aggregate entropy can help more effectively evaluate and react to collaboration patterns at any level of granularity.

3.5.3 Normalizing Author Entropy

Comparison of files or revisions with different numbers of authors (and thus different maximum entropies) can be difficult or unintuitive. Normalization facilitates comparison between files by dividing observed entropy by maximum possible entropy, scaling entropy to the range [0,1]. However, maximum possible entropy can vary according to context, and normalization factors must be chosen carefully. For example, consider the following possible options for normalizing author entropy for a set of three files with authors {A,B}, {B,C,D}, and {E,F,G}:

1. Normalize each file's entropy by log_2 of the number of authors in that file; scales all values to the range [0,1].
2. Normalize all entropies by $log_2 3$, since the maximum number of authors in any file is 3.

3. Normalize all entropies by $log_2 7$, since there are a total of 7 unique authors between all the files.

4. Do not normalize at all; define normalized entropy as ambiguous for sets with unknown maximum entropy.

Each of these strategies has advantages and drawbacks which depend on context and the question being asked. For example, the third approach produces deceptively low entropy values when there are many unique authors and few authors per file, while the first two approaches can distort the fact that files with more authors arguably have more complex collaboration. In the first three normalization techniques, adding more files with common or unique authors can change the normalization factor.

For example, we found files with a near-even split between two authors and near-maximum entropy. The addition of a few lines from a third author raised entropy slightly, but dividing by $log_2 3$ reduced normalized entropy significantly. When the new lines were changed by one of the original authors, entropy rebounded. In such cases, examining the percent of possible entropy may detract from accurate understanding of entropy trends.

3.5.4 Parallels with Social Network Studies

Social network analysis is an important corollary to author entropy. It is quite likely that underling social structure influences code collaboration.

Crowston and Howison [2005] have studied communication patterns in FLOSS (Free/Libre and Open Source Software) projects by examining developer interaction in bug tracking systems. They define and examine "centrality"—the degree to which communication pathways flow through a single developer. Centrality could augment author entropy data by providing social explanations for high or low entropy.

Bird et al. [2008] examine hidden social structures in open-source projects. They extract latent structure from email data, show that sub-communities form within projects,

and demonstrate that sub-communities are correlated with collaboration behavior. Additionally, they discuss parallels with Conway's Law [Conway, 1968] and Brooks' assertion that communication channels increase as the square of group size [Brooks, 1975]. Identification of sub-communities, organizational structure, and communication channels may strengthen our hypothesis that author entropy is influenced by social structure.

Alonso et al. [2004] study distinctions between open-source developers and contributors, and characterize roles of project participants based on rights to contribute. They mine CVS data for code authors and use email data to correlate coding productivity and mailing list activity, then construct interactions between contributors and committing developers. Their results could extend the author entropy metric; instead of counting only committing developers, indirect email contributors could be included in the entropy calculation.

3.6 Summary

Author entropy is a summary statistic that characterizes contribution patterns in source code. Entropy is easy to calculate, and can be calculated for different levels of granularity (e.g., lines, methods, files, modules). While author entropy does not directly imply a level of code quality, it can be used in conjunction with other software metrics to identify potential areas of concern within the source code of a project.

In a proof of concept study, we calculated author entropy and analyzed authorship patterns for a selection of open source data. Our exploratory research revealed interesting patterns in entropy distributions which may be indicators of significant development activities.

A potentially promising area of future research is to examine author entropy in the context of social network factors such as sub-communities and communication patterns. Crowston and Howison [2005] assert, "it is wrong to assume that FLOSS projects are distinguished by a particular social structure merely because they are FLOSS." The analysis of author contribution patterns in source code can help identify latent interactions and implicit social structures.

Because author entropy is a new metric, there are many unanswered questions about its utility and applicability. The vast amount of publicly available software data makes open source software research an especially suitable avenue for discovering the answers to these questions and expanding our current understanding of software development patterns.

Chapter 4

An Analysis of Author Contribution Patterns in Eclipse Foundation Project Source Code

4.1 Introduction

Software development is an inherently complex activity, often involving a high degree of collaboration between multiple individuals and teams, particularly when creating large software systems. Interactions between individual contributors can affect virtually all aspects of software development, including design, implementation, testing, maintenance, complexity, and quality.

Collaboration involves cooperation, communication, and coordination, and generally implies some governing organizational structure. The organization has an effect on the structure of the software being developed, as per "Conway's Law" [Conway, 1968]; presumably applying equally to proprietary and open source software. Brooks [1975] noted that potential communication channels increase as the square of the number of contributors. Thus, there is benefit to understanding and managing collaboration so it does not become a liability.

Analyzing collaboration data can help explain how people work together to develop software. Studies by Bird et al. [2008], Ducheneaut [2005], Gilbert and Karahalios [2007], Mockus et al. [2002], Dinh-Trong and Bieman [2005], and others have examined interactions between open source developers by correlating communication records (such as email) with source code changes. Such approaches can expose patterns which reinforce contributor roles and module boundaries, but may not be feasible for all projects (particularly if email archives are unavailable) and can be difficult to compare or aggregate across disparate projects.

In addition to examining collaboration across projects and modules, there is value in understanding how contributors collaborate *within* files. Having a sense of what constitutes "typical" collaboration for a project can provide valuable context. For example, if most files in a project have one or two authors, a file with 10 authors may merit additional scrutiny. In open source projects, unorganized and organic contributions may be evidence of the bazaar rather than the cathedral [Raymond, 2001]. In any case, simply knowing can help set expectations.

This paper both replicates and extends earlier results [Taylor et al., 2008]. Our research goals center around detecting, characterizing, and understanding patterns of collaboration within source code files. Our primary research questions are:

1. *How often do* n *authors contribute to a given file?*

   We anticipate that most files have a single author, and as the number of authors increases, the count of files with that many authors will decrease.

2. *Is there a higher degree of collaboration in small or large files?*

   We anticipate that there will be a positive correlation between file size and author count, partially because larger files have more code, and the potential for more distinct functionalities and individual responsibilities.

3. *Do files contain similar proportions of contributions from each author, or is there a dominant author who is the clear "owner" of a given file, and if so, how dominant is that author?*

   We anticipate that most source files will have one author who contributes significantly more code than any other single author, and that this author's dominance will be inversely related to the number of contributing authors.

4. *Is there a uniform or uneven distribution of collaboration across projects?*

   We anticipate that there will be a few "core" projects which are highly collaborative, and many ancillary projects which are less collaborative.

4.2 Methodology

We conducted an observational study on existing Eclipse projects by extracting author attribution data for Java source code files from git repositories. In this section we describe the process we used to select and obtain the data.

4.2.1 Project and File Selection

We chose to analyze Eclipse Foundation projects for several reasons, including:

- the number and variety of Eclipse-related projects,
- use of easily-recognizable programming languages,
- the ability to locally clone remote git repositories,
- a track record of sustained development activity,
- the existence of corporate-sponsored open source development projects.

We selected Java source files for our analysis, since over 92% of the source files in the repositories are Java, and Eclipse is so closely aligned with Java. We mined data from 251,633 files in 592 projects. We included auto-generated code in our analysis, since the inclusion of such files allows us to accurately characterize the state of the project to which they belong.

4.2.2 Extraction and Calculation

The first step in calculating author collaboration is to count how many authors have contributed to a file and the number of lines contributed by each one. Summarizing raw line counts with a single representative statistic per file allows for detailed statistical analysis of collaboration trends. In this paper, we use: (1) the percentage of lines attributed to the most dominant author in each file, and (2) author entropy (see Section 4.3 for details). These numbers can help characterize some aspects of author contribution patterns.

We created a bash script to locally clone each remote git repository and use 'git blame' to count the number of lines attributed to each author for each matching file. For each file in a project, the file path and line counts attributed to each author were recorded.

We then wrote a simple CLI tool to process this data and calculate the percentage of lines written by each author. Author entropy for each file was calculated using Equation 4.1. We also normalized entropy by dividing by the maximum possible entropy for each file, shown in Equation 4.2.

4.2.3 Limitations of the Data

We draw data only from git, a source control management (SCM) system that preserves snapshots of file state over time. We do not consider other collaboration mechanisms, such as email archives, forums, etc., although this could be a very interesting extension of this work.

It it important to note that the SCM record of who "owns" a line of code only identifies the individual who committed the most recent change affecting that line. It does not guarantee that the contributor actually conceived of, wrote, or even understands the code. By itself, it also does not tell us the genesis of a line; it could be new, a minor tweak, or a formatting change.

Because we consider only the latest revision of each file, this data cannot be used to make any inferences about collaboration over time. Without historical data, we can see the result of collaboration, but not the nature of the evolution of such collaboration.

Lastly, because we record author counts but not relative ordering of contributions from various authors, this data does not fully capture or express the amount of disorder. Because only percentages by each author are considered, the data makes no distinction between files with orderly, segregated blocks of contributions and files in which authors' contributions are all mixed together.

4.3 Author Entropy

Author entropy is a summary statistic that quantifies the mixture of authors' contributions to a file. Contribution percentages are weighted using logarithms and summed; the resulting value conveys more information about the distribution than a simple average, and can expose interesting authorship patterns more readily than raw line counts. Taylor et al. [2008] introduced author entropy and examined distributions in a proof-of-concept study with SourceForge data. A follow-on paper [Casebolt et al., 2009] examined author entropy in GNOME application source.

Entropy originated in the field of thermodynamics, which defines it as the disorder or randomness of molecules in a system. Entropy has also been defined in terms of probability theory and used in the fields of information theory [Shannon, 1948] and machine learning [Mitchell, 1997].

We apply entropy as a measure of collaboration between individual contributors. Specifically, we consider entropy of source code by counting the number of lines attributed to each author. This definition of entropy allows us to quantify the mixture of author contributions to a file.

4.3.1 Calculating Entropy

Entropy formulae are nearly identical across domains, and generally vary only in symbolic representation and constant multipliers. We use a formulation very similar to that used in machine learning.

If F is a file, A is the number of authors, and p_i is the proportion of the text attributed to author i, then the entropy of the file is defined as:

$$E(F) \equiv -\sum_{i=1}^{A}(p_i \cdot log_2\ p_i) \tag{4.1}$$

$E(F)$ is maximized when all authors contributed equal proportions of text in a file ($\forall\ i, p_i = \frac{1}{A}$). The upper limit of $E(F)$ is a function of A:

$$E_{max}(F) \equiv log_2\ A \tag{4.2}$$

We use log_2 for historical reasons tied to information theory (essentially, calculating the number of bits required to encode information). Although any logarithmic base would suffice, it is convenient that using log_2 results in entropy values in the range (0,1] for a binary classification.

4.3.2 Normalizing Entropy

Because the maximum possible entropy for a file is a function of the number of authors, intuitive understanding of entropy can be difficult. For example, an entropy value of 1.0 is the maximum possible for a file with 2 authors, but comparatively low for a file with 10 authors. Dividing E by E_{max} produces a normalized value in the range (0,1] which represents the percentage of maximum entropy. Normalized entropy can be easier to understand, and in some cases more appropriate for comparisons between disparate files.

4.4 Interpreting Collaboration

A high degree of collaboration within a given source file is not inherently good or bad; as with any metric, context is key. Without knowledge about additional factors such as a project's state, organization, and development conditions, interpreting collaboration is purely speculative. To illustrate this point, we list below a variety of factors that could influence author entropy.

Low entropy could result from factors as varied as:

- Well-architected and modular software.
- Excellent communication and coordination.

- Lack of involvement from potential contributors.
- A disciplined team in which each person "owns" a module.
- A gatekeeper who gets credit for code written by others.
- Code that few people understand.
- Code that was reformatted and old attributions lost.
- Code with exhaustive unit tests, known to be correct.
- Code with negligible unit tests and unknown defects.
- Auto-generated code that no human actually "wrote."
- Critical code that few people are allowed to modify.
- Mature code with little or no need for maintenance.
- Stale code that isn't touched, even if it needs fixing.
- Dead code which is no longer used or modified.

High entropy could result from factors as varied as:

- Code with high coupling or many inter-dependencies.
- Unrelated code entities being stored in a single file.
- Adding manpower to a late project (Brooks' law).
- Extremely buggy code that is constantly patched.
- Extremely stable code that is well-maintained.
- Enhancements or fixes that touch several files.
- Contributors joining or leaving a project team.
- Actively evolving code or refactoring activity.
- Miscommunication or lack of clear direction.
- Healthy collaboration between contributors.

- Overlapping responsibilities of contributors.
- Agile development or team programming.
- Potential for integration-stage problems.
- Continuous integration testing and fixes.

Such a menagerie of disparate factors is not a flaw in the metric itself, but rather suggests that any metric can easily be misinterpreted without proper context. For example, a file with high entropy written by several experts is likely of higher quality than a file written by one novice author. Two files may have similar entropies despite a large size difference. A recent contributor may understand a file better than the original author who wrote it years ago. Correlating author entropy with other metrics and observations can help distinguish between "good" and "bad" entropy and provide valuable new insights.

Author entropy cannot directly indicate other attributes of the source code. For example, file length is obscured since files of different size but equal proportions of contribution have the same entropy. Entropy also does not reflect quality or the relative importance of contributions, such as new functionality, bug fixes, comments, whitespace, or formatting. Although different entropy calculation techniques could opt to account for such factors, there is no way to deduce the weighting of such factors from a single number.

4.5 Results

The line count of the source files we examined ranged from 1 to 228,089, with a median of 89. The extreme right-tail skew (97.5% have 1,000 lines or fewer, 92.5% have 500 or fewer) suggests that the data may have an exponential distribution. Plotting the data with a log_{10} transformation produces a histogram (Figure 4.1) that closely resembles a normal distribution. A Q-Q plot showed that the population fits a log-normal distribution quite well, although the long tail caused higher residual deviations in the upper range. We also examined the files with 10 lines or fewer and found that nearly all of them were related to unit tests; several

projects have extensive tests with Java source files that specify in and out conditions, but have little or no code. Excluding these left-tail outliers greatly improved the fit of the Q-Q plot in the low range.

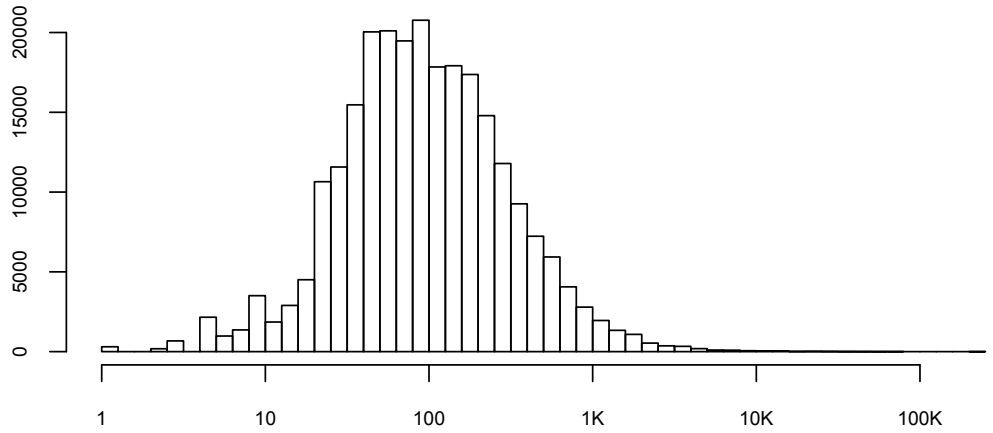


Figure 4.1: Frequency of file sizes (in number of lines).

To answer our first research question, we plotted the frequencies of files with n authors. The resulting histogram was an exponential decay curve, and when plotted with a logarithmic scale, a near-perfect log-linear decay is evident (see Figure 4.2). This confirms our hypothesis that most files have a single author, and that the number of files with n authors decreases as n increases. It is also strikingly similar to Lotka's Law [Lotka, 1926], which states that the number of authors who contribute n scientific publications is about $1/n^a$ of those with one publication, where a is nearly always 2. Lotka's law predicts about 60% of authors publish only once; in our data, 58.22% of the files have one author.

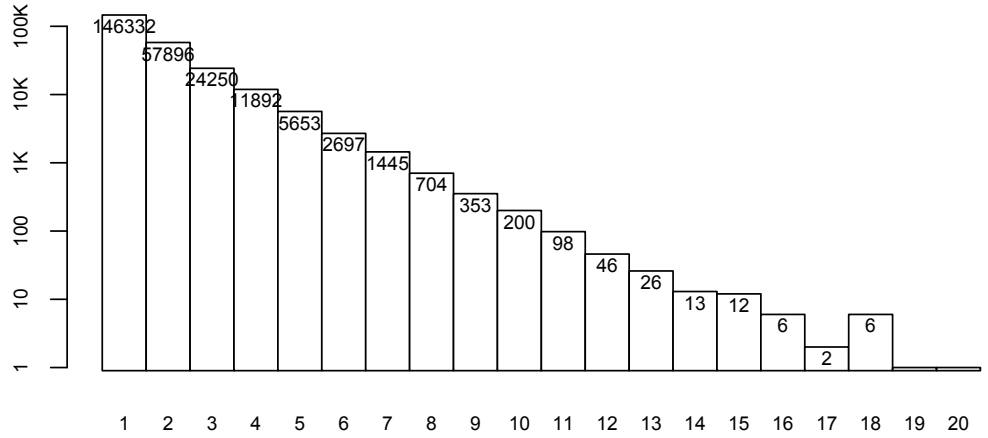


Figure 4.2: Frequency of number of authors contributing to a given file.

To answer our second research question, we plotted file size distributions grouped by author count (see Figure 4.3). The log-linear increase in average file size as the number of authors increases confirms our hypothesis that, on average, there is more collaboration (i.e., more authors) in large files. However, we must note that there is a degree of uncertainty due to the decreasing sample sizes for higher author counts and the extreme outliers for lower author counts.

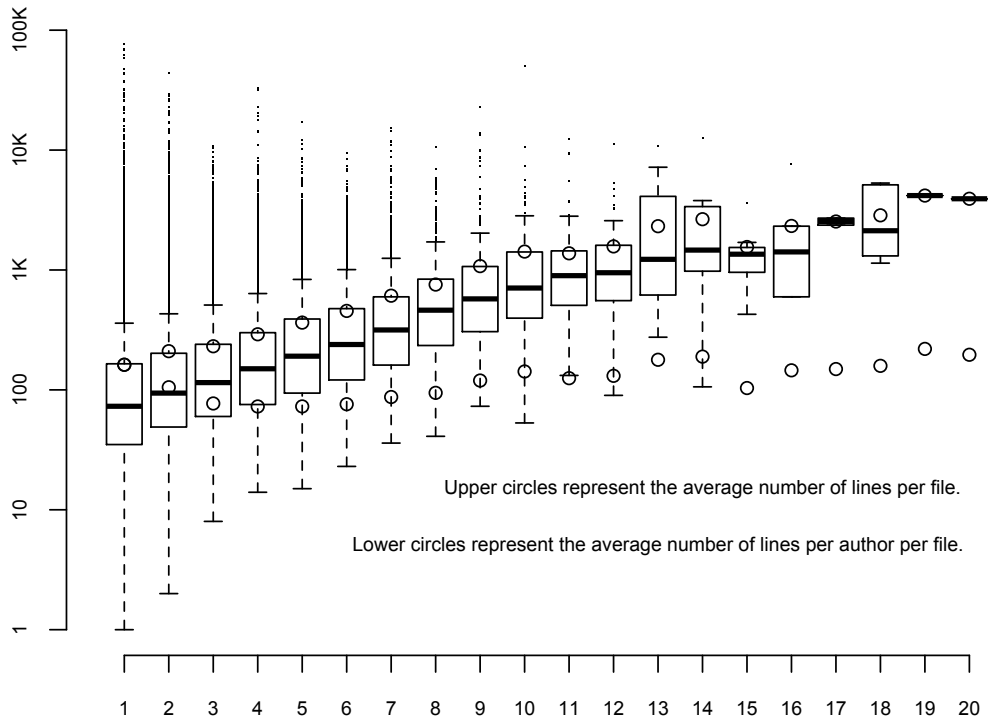


Figure 4.3: Author count vs. file size (in number of lines).

We augmented Figure 4.3 with two additional data series: (1) the average number of lines in a file, and (2) the average number of lines contributed *per author* to a file. Note that there is a pronounced dip between 1 and 10 authors, but a fairly consistent average throughout. Although evaluating the causes and ramifications of this trend are beyond the scope of this paper, we find this to be an interesting topic for future work.

To answer our third research question, we plotted the number of lines in files with two or more authors against the percentage of lines attributed to the most dominant author

in each file (see Figure 4.4). We also plotted the distributions of author dominance for all files with a given author count (see Figure 4.5).

These plots confirm our hypothesis that most files have a dominant author, and that the percentage of lines attributed to that author generally decreases as the count of contributing authors increases. We find it noteworthy that author dominance diverges increasingly from the lower bound ($\frac{1}{x}$). This suggests that in Eclipse projects, more authors contributing to a file does not imply balanced contributions; rather, a single author usually dominates the others.

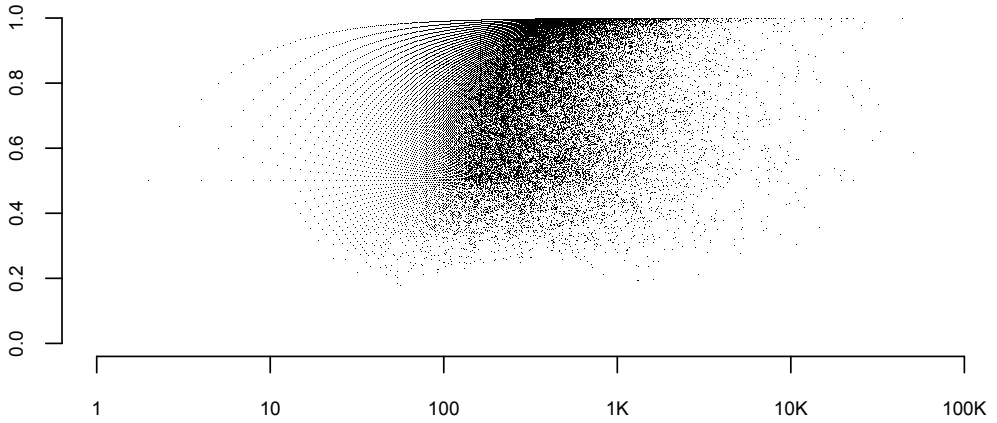


Figure 4.4: Line count vs. percent written by dominant author for files with 2+ authors.

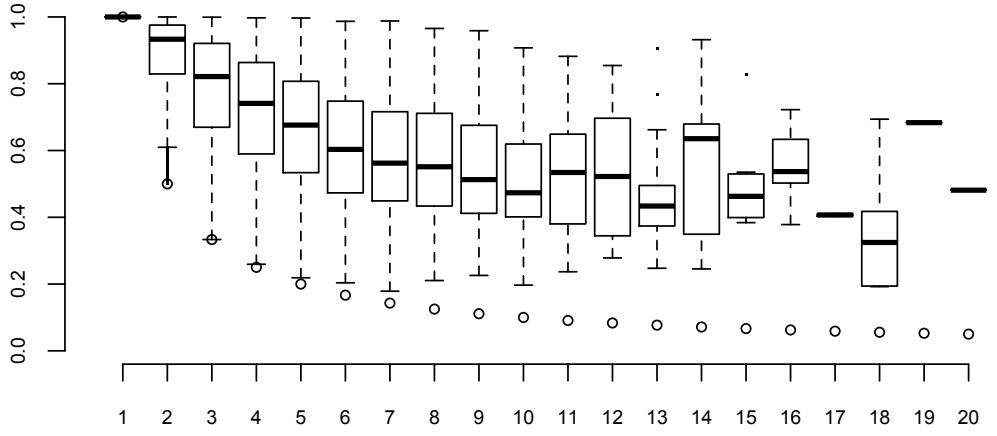


Figure 4.5: Author count vs. author dominance. Circles represent the curve $\frac{1}{x}$.

To answer our fourth research question, we plotted the number of lines in a project against the number of unique authors contributing to it for all 592 projects (see Figure 4.6).

Over 83% of the projects have 10 or fewer unique authors, and some significant outliers have much larger numbers of authors.

We also manually examined the 211 files with 11 or more authors. Nearly all of these files came from a handful of projects, all of which were among the top 25 projects with the most authors. These projects include:

- org.eclipse.ui.workbench (Eclipse IDE interface)
- org.eclipse.jdt.core (Java Development Tools)
- org.eclipse.cdt (C/C++ Development Tooling)
- org.eclipse.pdt (PHP Development Tools)
- org.eclipse.birt.report (Business Intelligence and Reporting Tools)
- org.eclipse.jface (UI application framework/toolkit based on SWT)

The nature and role of these highly-collaborative projects confirms our hypothesis that collaboration is not distributed uniformly, but is concentrated in a few core projects. This phenomenon is also related to our second research question, about the relationship between collaboration and file size.

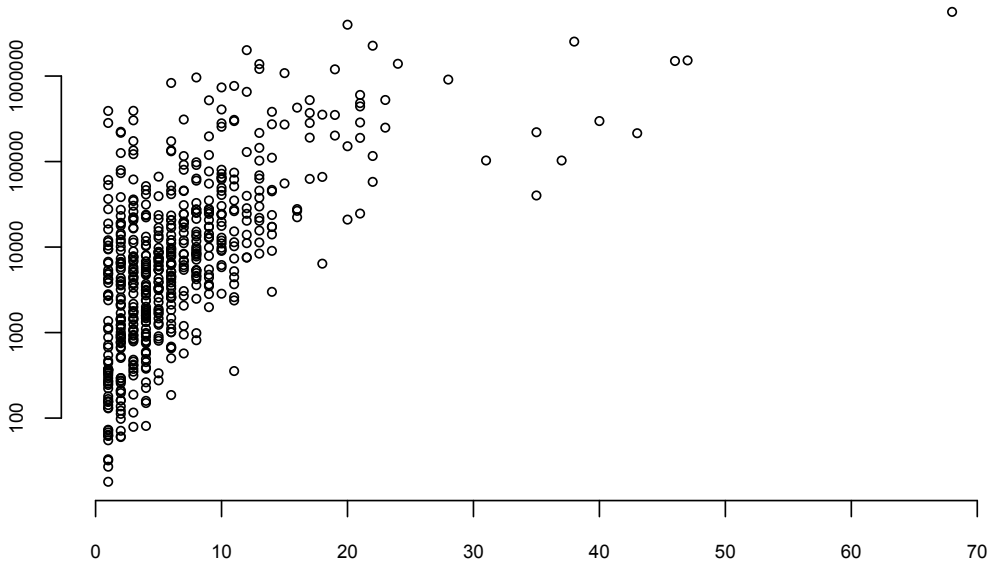


Figure 4.6: Author count vs. total number of lines for all 592 projects.

4.5.1 Additional Questions

In addition to our primary research questions, we also replicated some results from prior related work to verify whether the assertions made therein still hold for broader data. These results are related to distributions of author entropy (see Section 4.3) over varying file sizes and author counts.

In [Taylor et al., 2008] we found a positive relationship between author count and entropy (entropy rises as the number of authors increases). We found the same trend in Eclipse source code, although it breaks down somewhat for 11 or more authors due to sparseness of data (see Figure 4.7).

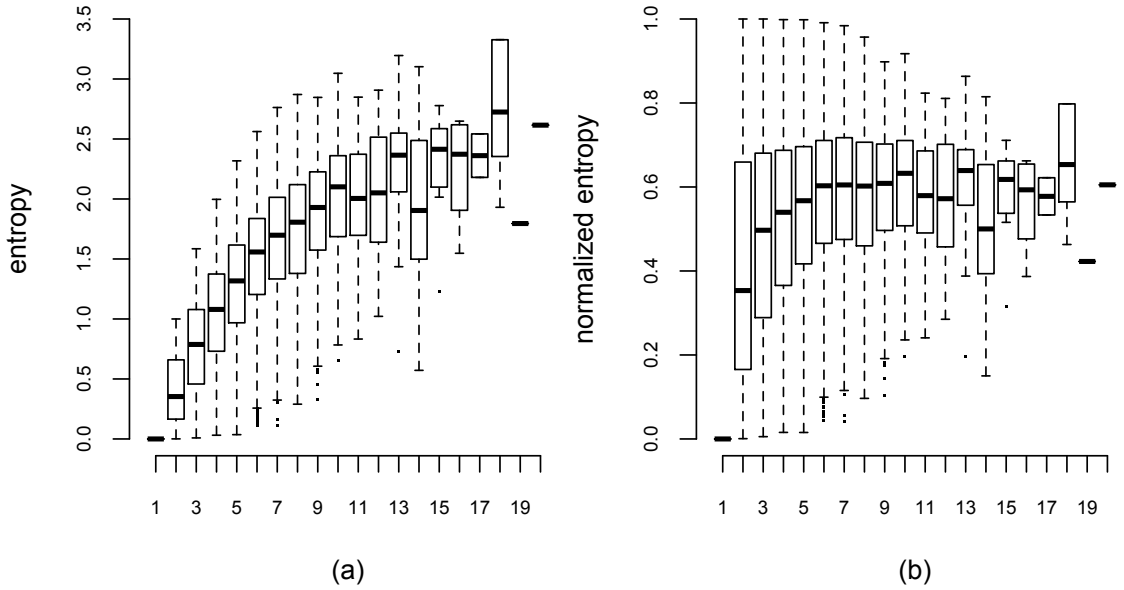


Figure 4.7: Author count vs. (a) entropy and (b) normalized entropy.

Although the entropy metric is inherently biased toward higher values (particularly when there are more authors), any file can have low entropy when one of the authors is extremely dominant. However, because the maximum possible entropy for a given file is a function of the number of authors, it can be difficult to compare entropies for files with different number of authors. For this reason, we use normalized entropy, which always falls in the range [0,1] regardless of author count, and thus represents the percentage of maximum possible entropy.

Interestingly, the data exhibits a trend previously observed [Taylor et al., 2008] in a very constrained set of SourceForge data: distributions of normalized entropy tend to center around 0.6 (or 60% of maximum possible entropy) as the author count increases. Even as the data becomes more sparse for higher author counts, the distributions tend to converge on the same range.

Casebolt et al. [2009] examined two-author source code files and observed an inverse relationship between file size and entropy (small files have high entropy and vice versa). A similar pattern occurs in our data, as shown in Figure 4.8(a). Unfortunately, it is impossible to discern how many data points share the same location. The task is even more hopeless when all files (not just those with two authors) are included in the same plot, as in Figure 4.8(b). To better understand the distribution and density of these data, we borrow a tool used by Krein et al. [2010] to visualize language entropy: 3D height maps. This technique generates an image in which densely-populated regions appear as elevated terrain (see Figure 4.9).

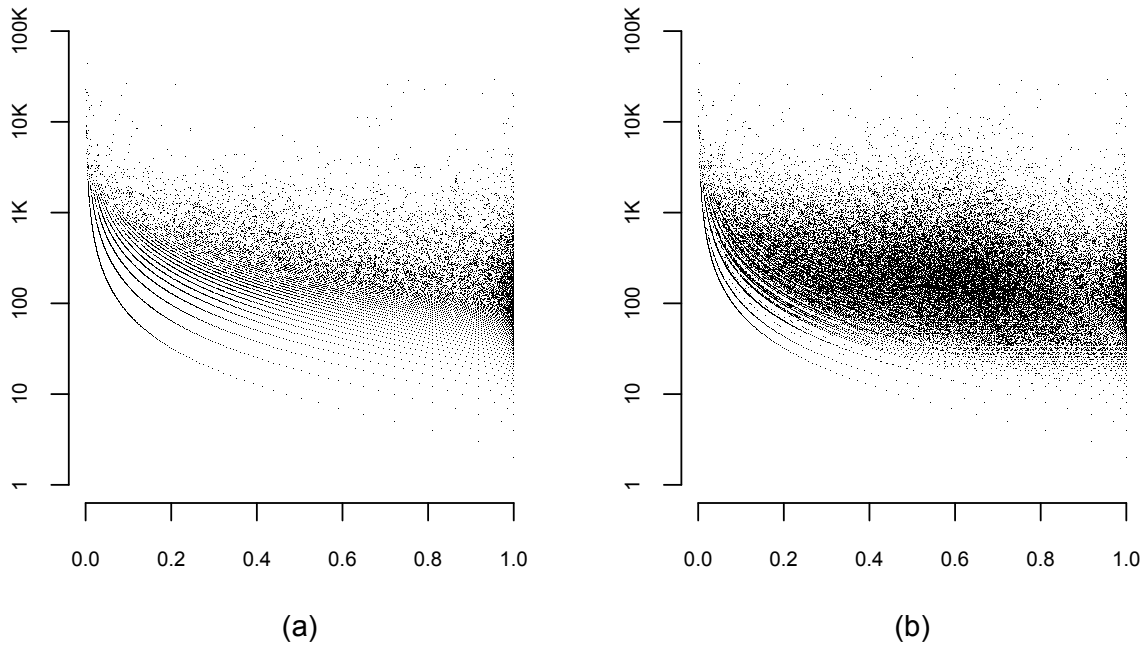


Figure 4.8: Normalized entropy vs. line count for (a) two authors and (b) all files.

Figure 4.9 is an adaptation of both Figure 4.8(b) (adding height) and Figure 4.1 (adding depth). Starting from the back/left, the curves represent files in which one author

"owns" all lines but one, two, etc. The disproportionate distribution of files on the furthest curves suggests that one- and two-line edits are probably extremely common occurrences in the projects we examined. This may be a manifestation of many small bug fixes, refactorings, interface changes, etc.

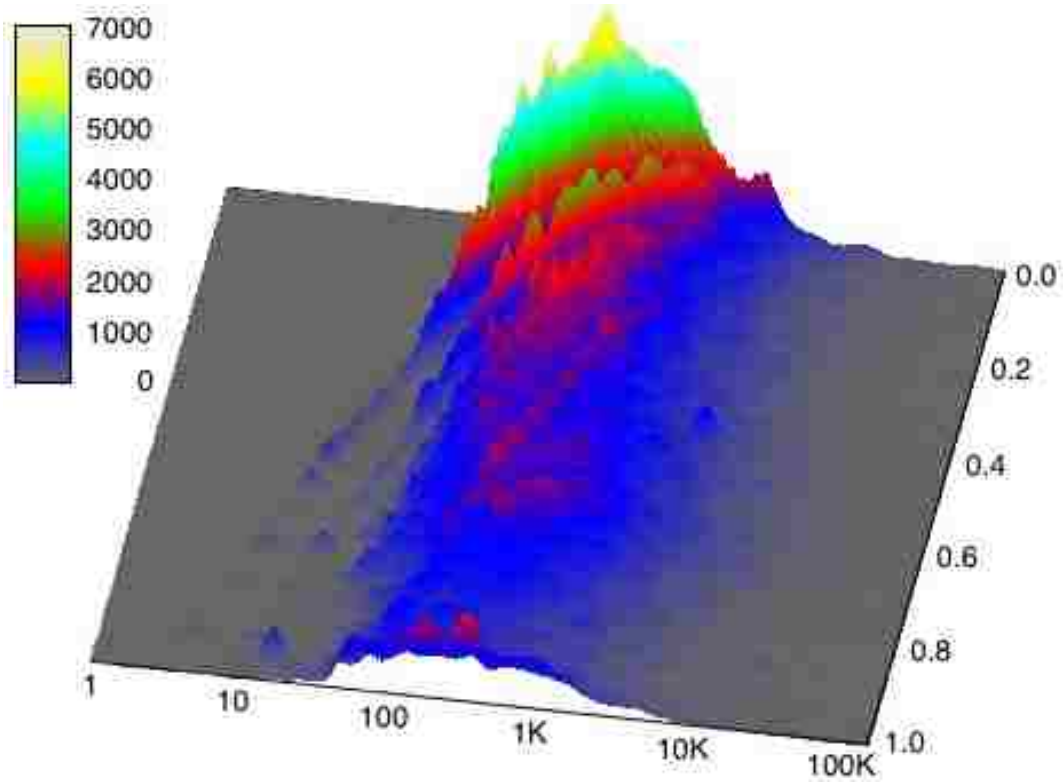


Figure 4.9: Height map of line count vs. normalized entropy (same data as Figure 4.8b).

4.6 Future Work

Although this paper both replicates and adds to the results of prior work [Taylor et al., 2008], we also see several promising ways to extend this research.

First, statistical analysis of author entropy over time, including how entropy of files, modules, and projects change over time, and why. One limitation of this paper is that we examine only the most recent version of each file; we do not consider previous versions of existing files, or files which once existed but have since been deleted. We see significant value

in understanding not only code ownership, but the degree of the resulting disorder, and how it is related to and caused by the development processes at play within a project.

Second, correlation of entropy with other code ownership measurements, communication records (such as email), and development roles. This could build on studies such as those by Bird et al. [2008], Mockus et al. [2002], Dinh-Trong and Bieman [2005], Jensen and Scacchi [2007], and von Krogh et al. [2003], among others. Understanding how contributor roles and project organization affect source code entropy could help OSS project administrators (or "core team") to more effectively analyze and organize contributors' efforts.

4.7 Conclusion

We discovered that author attribution data for source code files can provide insight into the nature of collaboration between source code contributors. As researchers, we seek to understand how people work together to develop complex systems, and to explain success or failure based on the data at our disposal. We are fascinated by the patterns of order which seem to naturally fall into place amid the organic chaos of free-form interactions.

Our study revealed similar authorship patterns in a vastly different code base than prior work, and suggested interesting new patterns we had not previously considered. Author entropy continues to be an interesting and useful metric for characterizing contributor interactions. Future research will improve our ability to link collaborative activity with the underlying factors that influence it, and facilitate improvements that enhance the quality of the software we produce.

References

FLOSSmole, 2004. URL `http://ossmole.sourceforge.net/`.

StatSVN, 2006. URL `http://statsvn.org/`.

Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Extraction of Contributor Information from Software Repositories. Submitted to MSR '06, 2004. URL `http://wwwcsif.cs.ucdavis.edu/~alonsoom/contributor_information_adg.pdf`.

Giuliano Antoniol, Yann-Gaël Guéhéneuc, Ettore Merlo, and Paolo Tonella. Mining the Lexicon Used by Programmers during Software Evolution. In *23rd IEEE International Conference on Software Maintenance*, pages 14–23, October 2007. doi:10.1109/ICSM.2007.4362614.

John Anvik. Automating Bug Report Assignment. In *28th International Conference on Software Engineering*, pages 937–940, May 2006. doi:10.1145/1134285.1134457.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an Open Bug Repository. In *3rd OOPSLA Workshop on Eclipse Technology eXchange*, pages 35–39, October 2005. doi:10.1145/1117696.1117704.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Who Should Fix This Bug? In *28th International Conference on Software Engineering*, pages 361–370, May 2006. doi:10.1145/1134285.1134336.

Phillip G. Armour. The Case for a New Business Model. *Communications of the ACM*, 43: 19–22, August 2000a. doi:10.1145/345124.345131.

Phillip G. Armour. The Five Orders of Ignorance. *Communications of the ACM*, 43:17–20, October 2000b. doi:10.1145/352183.352194.

David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using Version Control Data to Evaluate the Impact of Software Tools. In *21st International Conference on Software Engineering*, pages 324–333, May 1999. doi:10.1145/302405.302649.

Thomas Ball, Adam A. Porter, and Harvey P. Siy. If Your Version Control System Could Talk.... In *Workshop on Process Modeling and Empirical Studies of Software Engineering*, May 1997.

Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining Email Social Networks. In *3rd International Workshop on Mining Software Repositories*, pages 137–143, May 2006. doi:10.1145/1137983.1138016.

Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent Social Structure in Open Source Projects. In *16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 24–35, November 2008. doi:10.1145/1453101.1453107.

Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., January 1975. ISBN 0201006502.

Frederick P. Brooks, Jr. No Silver Bullets—Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987. doi:10.1109/MC.1987.1663532.

Gerardo Canfora and Luigi Cerulo. Impact Analysis by Mining Software and Change Request Repositories. In *11th IEEE International Software Metrics Symposium*, pages 9–29, September 2005. doi:10.1109/METRICS.2005.28.

Jason R. Casebolt, Jonathan L. Krein, Alexander C. MacLean, Charles D. Knutson, and Daniel P. Delorey. Author Entropy vs. File Size in the GNOME Suite of Applications. In *6th Working Conference on Mining Software Repositories*, pages 91–94, May 2009. doi:10.1109/MSR.2009.5069484.

Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. CVSSearch: Searching through Source Code using CVS Comments. In *17th IEEE International Conference on Software Maintenance*, pages 364–373, November 2001. doi:10.1109/ICSM.2001.972749.

Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining Specifications of Malicious Behavior. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 5–14, September 2007. doi:10.1145/1342211.1342215.

Melvin E. Conway. How Do Committees Invent? *Datamation*, 14(4):28–31, April 1968. URL `http://www.melconway.com/research/committees.html`.

Kevin Crowston and James Howison. The Social Structure of Free and Open Source Software Development. *First Monday*, 10(2), February 2005.

Davor Čubranić and Gail C. Murphy. Automatic Bug Triage Using Text Categorization. In *16th International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, June 2004.

William Dickinson, David Leon, and Andy Podgurski. Finding Failures by Cluster Analysis of Execution Profiles. In *23rd International Conference on Software Engineering*, pages 339–348, May 2001. doi:10.1109/ICSE.2001.919107.

Trung T. Dinh-Trong and James M. Bieman. The FreeBSD Project: A Replication Case Study of Open Source Development. *IEEE Transactions of Software Engineering*, 31(6): 481–494, June 2005. doi:10.1109/TSE.2005.73.

Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *15th IEEE International Conference on Software Maintenance*, pages 109–118, August 1999. doi:10.1109/ICSM.1999.792593.

Nicolas Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work*, 14(4):323–368, August 2005. doi:10.1007/s10606-005-9000-1.

Harald C. Gall and Michele Lanza. Software Evolution: Analysis and Visualization. In *28th International Conference on Software Engineering*, pages 1055–1056, May 2006. doi:10.1145/1134285.1134502.

Eric Gilbert and Karrie Karahalios. CodeSaw: A Social Visualization of Distributed Software Development. In *11th IFIP International Conference on Human-Computer Interaction*, pages 303–316, September 2007. doi:10.1007/978-3-540-74800-7_25.

Corrado Gini. *Variabilità e mutabilità*. Studi Economico Giuridici della Reale Università di Cagliari, 1912. Reprinted in *Memorie di metodologia statistica* (Ed. E. Pizetti and T. Salvemini.) Rome: Libreria Eredi Virgilio Veschi, 1955.

Ahmed E. Hassan. Mining Software Repositories to Assist Developers and Support Managers. In *22nd IEEE International Conference on Software Maintenance*, pages 339–342, September 2006. doi:10.1109/ICSM.2006.38.

Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. Call for Papers, 1st International Workshop on Mining Software Repositories, 2004. URL `http://msr.uwaterloo.ca/MSR2004_CallForPapers.pdf`.

James Howison and Kevin Crowston. The Perils and Pitfalls of Mining SourceForge. In *1st International Workshop on Mining Software Repositories*, pages 7–11, May 2004.

Chris Jensen and Walt Scacchi. Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. In *29th International Conference on Software Engineering*, pages 364–374, May 2007. doi:10.1109/ICSE.2007.74.

Huzefa Kagdi, Shehnaaz Yusuf, and Jonathan I. Maletic. Mining Sequences of Changed-files from Version Histories. In *3rd International Workshop on Mining Software Repositories*, pages 47–53, May 2006. doi:10.1145/1137983.1137996.

Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, March 2007. doi:10.1002/smr.344.

Sunghun Kim and Michael D. Ernst. Which Warnings Should I Fix First? In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 45–54, September 2007. doi:10.1145/1287624.1287633.

Jonathan L. Krein, Alexander C. MacLean, Daniel P. Delorey, Charles D. Knutson, and Dennis L. Eggett. Language Entropy: A Metric for Characterization of Author Programming Language Distribution. In *4th International Workshop on Public Data about Software Development*, page 6, June 2009.

Jonathan L. Krein, Alexander C. MacLean, Charles D. Knutson, Daniel P. Delorey, and Dennis L. Eggett. Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study. *International Journal of Open Source Software and Processes*, 2(2):41–61, June 2010. doi:10.4018/jossp.2010040104.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable Statistical Bug Isolation. In *26th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, June 2005. doi:10.1145/1065010.1065014.

Chao Liu and Jiawei Han. Failure Proximity: A Fault Localization-Based Approach. In *14th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 46–56, November 2006. doi:10.1145/1181775.1181782.

Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories. In *5th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 296–305, September 2005. doi:10.1145/1095430.1081754.

Alfred J. Lotka. The Frequency Distribution of Scientific Productivity. *Journal of the Washington Academy of Sciences*, 16(12):317–324, June 1926.

Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Threats to Validity in Analysis of Language Fragmentation on SourceForge Data. In *1st International Workshop on Replication in Empirical Software Engineering Research*, page 6, May 2010.

Steve McConnell. The 10 Most Powerful Ideas in Software Engineering. In *Companion Volume, 31st International Conference on Software Engineering*, page 12, May 2009. doi:10.1109/ICSE-COMPANION.2009.5070958.

Manoel Mendonca and Nancy L. Sunderhaft. Mining Software Engineering Data: A Survey. Technical report, Data & Analysis Center for Software, November 1999.

Tom Mens and Serge Demeyer. Future Trends in Software Evolution Metrics. In *4th International Workshop on Principles of Software Evolution*, pages 83–86, September 2001. doi:10.1145/602461.602476.

Tom M. Mitchell. *Machine Learning*, pages 55–57. McGraw-Hill, 1997. ISBN 0070428077.

Audris Mockus, Stephen G. Eick, Todd L. Graves, and Alan F. Karr. On Measurement and Analysis of Software Changes. Technical report, National Institute of Statistical Sciences, 1999.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002. doi:10.1145/567793.567795.

Audris Mockus, David M. Weiss, and Ping Zhang. Understanding and Predicting Effort in Software Projects. In *25th International Conference on Software Engineering*, pages 274–284, May 2003. doi:10.1109/ICSE.2003.1201207.

Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical Debugging Using Compound Boolean Predicates. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 5–15, July 2007. doi:10.1145/1273463.1273467.

Gregory B. Newby, Jane Greenberg, and Paul Jones. Open source software development and Lotka's Law: Bibliometric patterns in programming. *Journal of the American Society for Information Science and Technology*, 54(2):169–178, January 2003. doi:10.1002/asi.10177.

Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, Inc., 2001. ISBN 0596001088.

Gregorio Robles, Jesus M. González-Barahona, and Rishab A. Ghosh. GlueTheos: Automating the Retrieval and Analysis of Data from Publicly Available Software Repositories. In *1st International Workshop on Mining Software Repositories*, pages 28–31, May 2004. doi:10.1049/ic:20040471.

Per Runeson, Magnus Alexandersson, and Oskar Nyholm. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering*, pages 499–510, May 2007. doi:10.1109/ICSE.2007.32.

Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza. A non-invasive approach to product metrics collection. *Journal of Systems Architecture*, 52(11):668–675, November 2006. doi:10.1016/j.sysarc.2006.06.010.

Claude Elwood Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423/623–656, July/October 1948.

Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Supporting Software Maintenance by Mining Software Update Records. In *17th IEEE International Conference on Software Maintenance*, pages 22–31, November 2001. doi:10.1109/ICSM.2001.972708.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? In *2nd International Workshop on Mining Software Repositories*, pages 1–5, May 2005. doi:10.1145/1083142.1083147.

Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* iComment: Bugs or Bad Comments? */. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158, October 2007. doi:10.1145/1294261.1294276.

Quinn C. Taylor, James E. Stevenson, Daniel P. Delorey, and Charles D. Knutson. Author Entropy: A Metric for Characterization of Software Authorship Patterns. In *3rd International Workshop on Public Data about Software Development*, pages 42–47, September 2008.

Quinn C. Taylor, Christophe Giraud-Carrier, and Charles D. Knutson. Applications of Data Mining in Software Engineering. *International Journal of Data Analysis Techniques and Strategies*, 2(3):243–257, July 2010. doi:10.1504/IJDATS.2010.034058.

Quinn C. Taylor, Jonathan L. Krein, Alexander C. MacLean, and Charles D. Knutson. An Analysis of Author Contribution Patterns in Eclipse Foundation Project Source Code. In *7th International Conference on Open Source Systems*, pages 269–281, October 2011. doi:10.1007/978-3-642-24418-6_19.

Lucian Voinea, Alexandru Telea, and Jarke J. van Wijk. CVSscan: Visualization of Code Evolution. In *2nd ACM Symposium on Software Visualization*, pages 47–56, May 2005. doi:10.1145/1056018.1056025.

Georg von Krogh, Sebastian Spaeth, and Karim R. Lakhani. Community, Joining, and Specialization in Open Source Software Innovation: A Case Study. *Research Policy*, 32(7): 1217–1241, July 2003. doi:10.1016/S0048-7333(03)00050-7.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting Object Usage Anomalies. In *6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 35–44, September 2007. doi:10.1145/1287624.1287632.

Westley Weimer and George C. Necula. Mining Temporal Specifications for Error Detection. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, April 2005. doi:10.1007/978-3-540-31980-1_30.

Gerald M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971. ISBN 0442207646.

Tao Xie. Bibliography on Mining Software Engineering Data. URL `http://ase.csc.ncsu.edu/dmse`.

Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining Software Engineering Data. In *Companion Volume, 29th International Conference on Software Engineering*, pages 172–173, May 2007. doi:10.1109/ICSECOMPANION.2007.50.

Shen Zhang, Yongji Wang, Feng Yuan, and Li Ruan. Mining Software Repositories to Understand the Performance of Individual Developers. In *31st International Computer Software and Applications Conference*, pages 625–626, July 2007. doi:10.1109/COMPSAC.2007.148.

Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6): 429–445, June 2005. doi:10.1109/TSE.2005.72.