Applications of data mining in software engineering

Quinn Taylor*

Department of Computer Science,
Brigham Young University,
Provo, UT 84602, USA
E-mail: quinntaylor@byu.edu
*Corresponding author

Christophe Giraud-Carrier

Department of Computer Science,
Brigham Young University,
Provo, UT 84602, USA
E-mail: cgc@cs.byu.edu

Abstract: Software engineering processes are complex, and the related activities often produce a large number and variety of artefacts, making them well-suited to data mining. Recent years have seen an increase in the use of data mining techniques on such artefacts with the goal of analysing and improving software processes for a given organisation or project. After a brief survey of current uses, we offer insight into how data mining can make a significant contribution to the success of current software engineering efforts.

Keywords: data mining; software engineering; applications.


Biographical notes: Quinn Taylor is a student in the MS degree programme in Computer Science at Brigham Young University and a Researcher in the SEQuOIA Lab where he focuses on understanding and visualising software structure and development processes, including through the use of data mining techniques. His research interests include software architectures, software evolution, code maintenance and decay, software reverse engineering and refactoring.

Christophe Giraud-Carrier is an Associate Professor and the Director of the Data Mining Laboratory in the Department of Computer Science at Brigham Young University. His research interests include metalearning, social network analysis, medical informatics and applications of data mining. He received his BS, MS and PhD in Computer Science at BYU in 1991, 1993 and 1994, respectively.

Copyright © 2010 Inderscience Enterprises Ltd.
1 Introduction

Software systems are inherently complex and difficult to conceptualise. 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 organisations 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 organisations 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 organisations.

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 conceptualise 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 organisation – 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 organised as follows. In Section 2, we briefly discuss related work, pointing to surveys and venues dedicated to recent applications of data mining to software engineering. Section 3 describes the sources of software data available for mining and Section 4 provides a brief, but broad, survey of current practices in this domain. Section 5 discusses issues specific to mining software engineering data and prerequisites for success. Finally, Section 6 concludes the paper.

2 Related work

Although the application of data mining to software engineering artefacts 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 and 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 (2010) 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 visualisation techniques.

Finally, Kagdi et al. (2007) have recently published a comprehensive survey of approaches for MSR 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.

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 endeavours, 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 artefacts can be obtained. Software ‘artefacts’ are a product of software development processes. Artefacts 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 organisations utilise 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 analyse historical artefacts 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 organisations (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 (e-mail, 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 organisations that use the same processes across multiple projects and want to examine a process in terms of overall 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.

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.

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 emphasise innovative and promising approaches and how they can benefit software organisations.

4.1.1 Association rules and frequent patterns

Zimmermann et al. (2005) have developed the Reengineering of Software Evolution (ROSE) tool 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
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 analyses 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 \textit{a priori} algorithm, combined with pattern categorisation 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.
Applications of data mining in software engineering

Sliwerski 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: characterisation 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 behaviour 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.

4.1.2 Classification

Large software organisations frequently use bug tracking software to manage defects and correlate them with fixes. Bugs are assigned a severity and assigned to someone within the organisation. 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) and Anvik (2006) have researched automatic classification of defects by severity (‘triage’), and Cubranic 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) 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 prioritisation 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 prioritisation 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.

4.1.3 Clustering

Most applications of data mining clustering techniques to software engineering data relate to the discovery and localisation 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 behaviours (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 localise faults and better determine failure proximity.

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 artefacts 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 unsynchronised 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 organisations 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 tokenisation, 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
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 analyse comments and detect inconsistencies. Their tests on four 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 e-mail communications between contributors to open-source software (OSS). This approach allows them to detect and represent social networks that exist in the open-source community, characterise interactions between contributors and identify roles such as ‘chatterers’ and ‘changers’. The in-degree and out-degree of e-mail responses are analysed 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.

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
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.

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 artefacts. 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)
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
Applications of data mining in software engineering

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) proposed 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: analysing 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.

4.2.2 Management tasks

Hassan (2006) discusses ways in which software artefacts and historical data can be used to assist managers. He states that: “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 as “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 organisations. 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.

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 characterise patterns in software development, rather than understanding about a specific project to guide its development.

Researchers frequently analyse data from open-source projects, but as Howison and Crowston (2004) explain, mining data from organisations 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 generalise 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 visualisation 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. (2006) characterise 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 (1926) Law (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 artefacts 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 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.
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.

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 endeavour.

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 organisation. 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 organisations 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 localisation 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 analysed and properly implemented if it is to change anything. Data mining can only answer questions that are effectively articulated and implemented and good intentions cannot rescue bad data (or no data).

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

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 3. 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 generalised.

Because many data mining tasks (e.g., text mining) are extremely computationally expensive, replication of effort is a major concern. Tools that help simplify centralised extraction and caching of results will make widespread data mining more appealing to large software organisations; the same tools can make collaborative data mining research more effective. The ability to share data among colleagues or collaborators without replication amortises 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 visualisation 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 organisation’s perception of value) of the data itself.

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 artefacts and data, no two organisations 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 analyse it.

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 ‘artefact’ data:

1 version control systems
2 bug trackers
3 electronic developer communication
Applications of data mining in software engineering 255

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.

References


Notes

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