Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

8-2009

Data Mining for Software Engineering

Tao XIE
North Carolina State University

Suresh Thummalapenta
North Carolina State University

David LO
Singapore Management University, davidlo@smu.edu.sg

Chao LIU
Microsoft Research

DOI: https://doi.org/10.1109/MC.2009.256

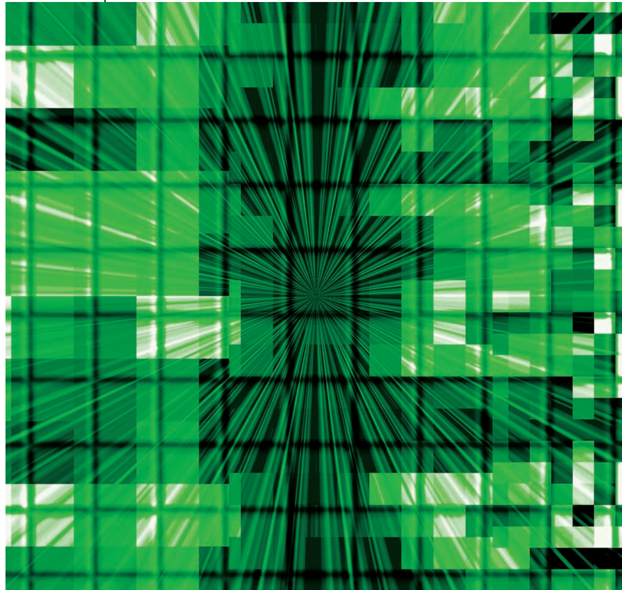
Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

XIE, Tao; Thummalapenta, Suresh; LO, David; and LIU, Chao. Data Mining for Software Engineering. (2009). *Computer*. 42, (8), 55-62. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/763

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.



DATA MINING FOR SOFTWARE ENGINEERING

Tao Xie and Suresh Thummalapenta, *North Carolina State University*
David Lo, *Singapore Management University*
Chao Liu, *Microsoft Research*

To improve software productivity and quality, software engineers are increasingly applying data mining algorithms to various software engineering tasks. However, mining SE data poses several challenges. The authors present various algorithms to effectively mine sequences, graphs, and text from such data.

Because software plays a critical role in businesses, governments, and societies, improving software productivity and quality is an important goal of software engineering. Mining SE data has recently emerged as a promising means to meet this goal due to two main trends: the increasing abundance of such data and its demonstrated helpfulness in solving numerous real-world problems.

Popular software version control systems such as the Concurrent Version System and Subversion let engineers not only capture current snapshots of a project code base but also maintain full version histories. Complete life-cycle bug management is also possible through systems such as Bugzilla. Moreover, rich execution data is available thanks to powerful instrumentation tools such as Microsoft's Dr. Watson technology, used by software developers and lightweight monitoring tools optimized for end users.

SE data concerns the 3Ps: people, processes, and products. People include software developers, testers, project managers, and users. Processes include various development phases and activities such as requirements, design, implementation, testing, debugging, maintenance, and deployment. Products can be structured, such as source code (including production and test code), or nonstructured, such as documentation and bug reports.

As the first column of Table 1 shows, SE data can be broadly categorized into

- *sequences* such as execution traces collected at runtime, static traces extracted from source code, and co-changed code locations;
- *graphs* such as dynamic call graphs collected at runtime and static call graphs extracted from source code; and
- *text* such as bug reports, e-mails, code comments, and documentation.

To improve both software productivity and quality, software engineers are increasingly applying data mining algorithms to various SE tasks. For example, such algorithms can help engineers figure out how to invoke API methods provided by a complex library or framework with insufficient documentation. In terms of maintenance, such algorithms can assist in determining what code locations

Table 1. Example software engineering data, mining algorithms, and SE tasks.

Example SE data	Example mining algorithms	Example SE tasks
Sequences: execution/static traces, co-changes	Frequent itemset/sequence/ partial-order mining, sequence matching/ clustering/classification	Programming, maintenance, bug detection, debugging
Graphs: dynamic/ static call graphs, program dependence graphs	Frequent subgraph mining, graph matching/clustering/classification	Bug detection, debugging
Text: bug reports, e-mails, code comments, documentation	Text matching/clustering/ classification	Maintenance, bug detection, debugging

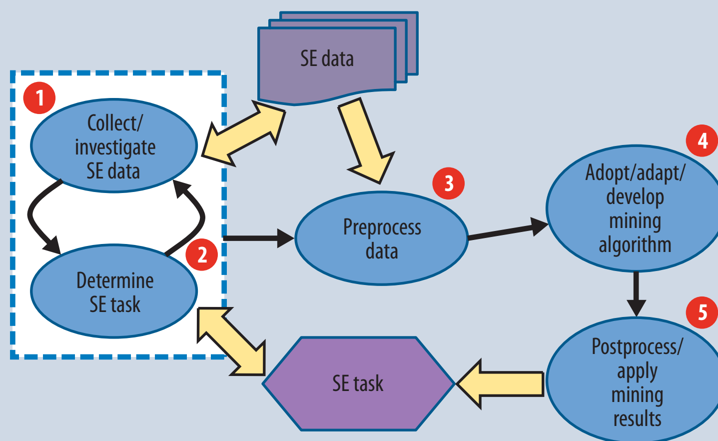


Figure 1. The methodology for mining software engineering data involves five basic steps. In practice, software engineers adopt a mixture of the first two steps.

must be changed when another code location is changed. Software engineers can also use data mining algorithms to hunt for potential bugs that can cause future in-field failures as well as identify buggy lines of code (LOC) responsible for already-known failures. The second and third columns of Table 1 list several example data mining algorithms and the SE tasks to which engineers apply them.

MINING METHODOLOGY

Figure 1 shows an overview of the five main steps in mining SE data. Software engineers can start with either a problem-driven approach (knowing what SE task to assist) or a data-driven approach (knowing what SE data to mine), but in practice they commonly adopt a mixture of the first two steps: collecting/investigating data to mine and determining the SE task to assist. The three remaining steps are, in order, preprocessing data, adopting/adapting/developing a mining algorithm, and postprocessing/applying mining results.

Preprocessing data involves first extracting relevant data from the raw SE data—for example, static method-call sequences or call graphs from source code, dynamic method-call sequences or call graphs from execution traces, or word sequences from bug report summaries. This data is further preprocessed by cleaning and properly formatting it for the mining algorithm. For example, the input format for sequence data can be a sequence database where each sequence is a series of events.

The next step produces a mining algorithm and its supporting tool, based on the mining requirements derived in the first two steps. In general, mining algorithms[1] fall into four main categories:

- *frequent pattern mining*—finding commonly occurring patterns;
- *pattern matching*—finding data instances for given patterns;
- *clustering*—grouping data into clusters; and
- *classification*—predicting labels of data based on already-labeled data.

The final step transforms the mining algorithm results into an appropriate format required to assist the SE task. For example, in the preprocessing step, a software engineer replaces each distinct method call with a unique symbol in the sequence database being fed to the mining algorithm. The mining algorithm then characterizes a frequent pattern with these symbols. In postprocessing, the engineer changes each symbol back to the corresponding method call. When applying frequent pattern mining, this step also includes finding locations that match a mined pattern—for example, to assist in programming or maintenance—and finding locations that violate a mined pattern—for example, to assist in bug detection.

MINING CHALLENGES

Mining SE data presents several challenges.

Requirements unique to SE

Most SE data mining studies rely on well-known, publicly available tools such as association-rule mining and clustering. Such black-box reuse of mining tools may compromise the requirements unique to SE by fitting them to the tools' undesirable features. Further, many such tools are general purpose and should be adapted to assist the particular task at hand. However, SE researchers may lack the expertise to adapt or develop mining algorithms or tools, while data mining researchers may lack

the background to understand mining requirements in the SE domain. One promising way to reduce this gap is to foster close collaborations between the SE community (requirement providers) and the data mining community (solution providers). Our research efforts represent one such instance.

Complex data and patterns

SE researchers typically mine individual data types alone to accomplish a certain SE task. However, SE tasks increasingly demand the mining of multiple correlated data types, including both sequence and text data, together to achieve the most effective result. Even for a single data type, rich information is commonly associated not only with an individual data item but also with the linkage among multiple data items.

In addition, pattern representation in the SE domain can be complex. There might be no existing mining algorithms that produce desired pattern representations, and developing new algorithms for such representations can be difficult. Overall, ensuring a scalable yet expressive mining solution is difficult.

Large-scale data

SE researchers often mine only a few local repositories. However, there may be too few relevant data points in these repositories to support the mining of desirable patterns, such as ones among API methods of interest. One way to address this problem is to mine Internet-scale software repositories—for example, via a code search engine. Mining can then be applied to the entire open source world or to many software repositories within an organization or across organizations. Further, execution traces collected from even an average-sized program can be very long, and dynamically or statically extracted call graphs can be enormous. Analyzing such large-scale data poses a challenge to existing mining algorithms.

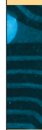
Just-in-time mining

SE researchers usually conduct offline mining of data already collected and stored. However, in modern integrated SE environments, especially collaborative environments, software engineers must be able to collect and mine SE data on the fly to provide rapid just-in-time feedback. Stream data mining algorithms and tools could be adapted or developed to satisfy such challenging mining requirements.

MINING SEQUENCES

Example SE sequence data include method-call sequences, either dynamically collected during program execution or statically extracted from program source code. Dynamically collected method-call sequences are in the form of execution traces.

Common types of mining algorithms include frequent pattern mining—for example, frequent itemset, sequence, or partial-order mining—as well as sequence matching, clustering, and classification. Frequent pattern mining can be used to mine for API call usage patterns to help in programming or for specifications to help in bug detection.[2] Different algorithms produce patterns that reflect different levels of information, and which algorithm to choose depends on the specific SE task's mining requirements. For example, association rules or frequent itemsets do not reflect sequential order information among the elements in the mined patterns, whereas frequent sequence or partial orders do. Sequence matching can be used to detect potential bugs by finding locations for almost, but not exactly, matching mined sequence patterns.



In modern integrated SE environments, software engineers must be able to collect and mine SE data on the fly to provide rapid just-in-time feedback.

We have developed new mining algorithms that address the unique characteristics of SE data. These algorithms include mining iterative patterns, temporal rules, sequence diagrams, finite-state machines (FSMs), and sequence association rules.

Iterative pattern mining

Existing sequential-pattern-mining algorithms ignore repetitions of items (method calls) in a trace (sequence) and are agnostic to the rich semantics of property specification languages in SE. To address this problem, we have developed an algorithm that captures repetitive occurrences of the patterns within each trace and across multiple traces.[3]

Consider the following sequence database:

1 A B C D E A X B C
2 A G X B C
3 A X B C

Our algorithm mines the set of frequent closed iterative subsequence patterns, at the minimum support threshold of 4, which is {⟨A B C⟩}. Current frequent subsequence mining algorithms would ignore the second occurrence or repetition of this pattern in sequence 1 and thus cannot mine this pattern.

We have applied iterative pattern mining to traces collected from the transaction component of the JBoss application server. The mined patterns range from small, frequently occurring patterns such as "lock must be

Connection setup	TransactionManagerLocator.getInstance TransactionManagerLocator.locate TransactionManagerLocator.tryJNDI TransactionManagerLocator.usePrivateAPI	Transaction commit	TxManager.commit TransactionImpl.commit TransactionImpl.beforePrepare TransactionImpl.checkIntegrity TransactionImpl.checkBeforeStatus TransactionImpl.endResources TransactionImpl.completeTransaction TransactionImpl.cancelTimeout TransactionImpl.doAfterCompletion TransactionImpl.instanceDone
TxManager setup	TxManager.begin XidFactory.newXid XidFactory.getNextId XidImpl.getTrulyGlobalId		
Transaction setup	TransactionImpl.associateCurrentThread TransactionImpl.getLocalId XidImpl.getLocalId LocalId.hashCode TransactionImpl.equals TransactionImpl.getLocalIdValue XidImpl.getLocalIdValue TransactionImpl.getLocalIdValue XidImpl.getLocalIdValue	Transaction disposal	TxManager.releaseTransactionImpl TransactionImpl.getLocalId XidImpl.getLocalId LocalId.hashCode LocalId.equals

(a)

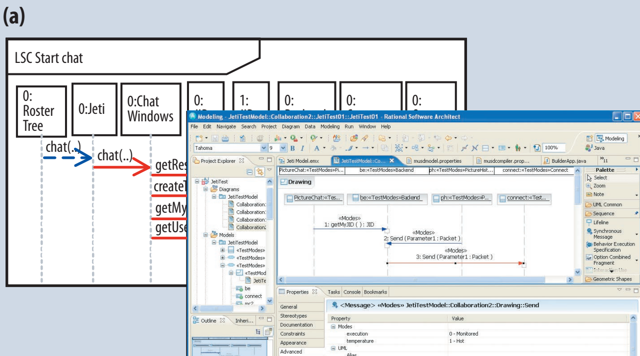
Resource allocation:

"Whenever a resource is allocated, eventually it needs to be released."

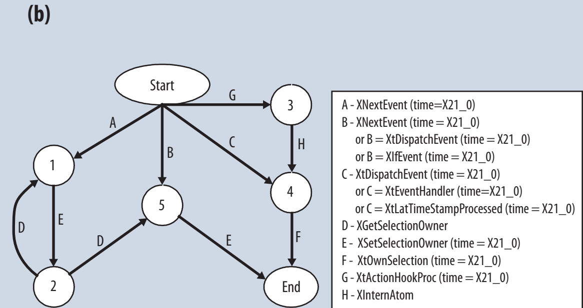
Windows WDK CancelSpinLock rule:

"A device driver needs to call IoAcquireCancelSpinLock before calling IoReleaseCancelSpinLock and it needs to call IoReleaseCancelSpinLock before any subsequent calls to IoAcquireCancelSpinLock."

(b)



(c)



Learner	k-len = 1			k-len = 3		
	Recall	Precs.	PS	Recall	Precs.	PS
k-tails	1.000	0.000	N/A	0.998	0.313	N/A
sk-strings	1.000	0.654	0.692	0.998	0.883	0.758
SMArTIC	1.000	0.820	0.910	0.998	0.987	0.956

(d)

Figure 2. Examples of mined API specifications: (a) A frequent pattern from the transaction component of the JBoss application server; (b) temporal rules; (c) sequence diagrams from the Jeti instant messaging application; (d) a finite state machine from the Xlib and XToolkit routines of the X11 windowing system.

followed by unlock" to longer patterns exceeding 30 method calls, as shown in Figure 2a. We have also shown that mined iterative patterns could be used as high-level features to classify program behaviors.[4]

Temporal rule mining

A rule captures a constraint between its precondition and postcondition. To find common temporal rules such as "Whenever a series of events (for example, method calls) occurs, eventually another series of events will occur," we developed an algorithm to mine rules of arbitrary lengths (pre- and postconditions of the rules could be composed of multiple events).[5] We apply the algorithm on traces to find candidate temporal invariants, which could later be used to detect bugs.

The algorithm first locates frequent preconditions and then, for each one, mines a set of significant rules obeying the minimum confidence threshold. An example of a significant rule mined from the sequence database is $\langle A \rangle \rightarrow \langle B\ C \rangle$, with *min_sup* set to 4 and *min_conf* set to 100 percent. Using an extension, the algorithm finds rules of the format "Whenever a series of events occurs, another series of events must have happened before"—known as past-time temporal rules.[6]

We have applied the algorithm on traces from multiple open source applications to mine both short and long rules characterizing the applications' behaviors. Figure 2b shows some examples of temporal rules.

Sequence-diagram and FSM mining

We have developed algorithms to mine other forms of API specifications, including sequence diagrams and FSMs (or automata)[7,8] from sequences. These specifications capture constraints that are difficult to express using patterns and rules. For example, a UML sequence diagram contains not only method calls but also caller and callee information shown as lifelines. Moreover, rules and patterns do not express disjunctions and thus cannot capture the branching and loop behaviors expressible with FSMs. Rules and patterns are often used to express strongly observed properties in the traces—those that appear with high support and confidence—while FSMs tend to be used to capture the overall view of event transitions in the traces, ignoring both support and confidence.

Figure 2c shows examples of sequence diagrams mined from the Jeti instant messaging application, while Figure 2d shows a mined FSM corresponding to a protocol of the Xlib and XToolkit routines of the X11 windowing system.

Sequence association rule mining

Applying existing association-rule mining algorithms can help find rules of the form "$FC_a \Rightarrow FC_e$" as specifications, where both FC_a and FC_e are method calls that share the same receiver object in object-oriented programs. These specifications can be used to find exception-handling bugs: if FC_e does not follow FC_a in all exception paths. However, association rules of this form are often insufficient to capture common exception-handling rules. In some situations, FC_a is not necessarily followed by FC_e when FC_a raises exceptions, although both method calls share the same receiver object.

```
1.1:  ...
1.2:  OracleDataSource ods = null; Session session = null;
      Connection conn = null; Statement statement = null;
1.3:  logger.debug("Starting update");
1.4:  try {
1.5:      ods = new OracleDataSource();
1.6:      ods.setURL("jdbc:oracle:thin:scott/tiger@192.168.1.2:1521:catfish");
1.7:      conn = ods.getConnection();
1.8:      statement = conn.createStatement();
1.9:      statement.executeUpdate("DELETE FROM table1");
1.10:     connection.commit(); }
1.11:     catch (SQLException se) {
1.12:             if (conn != null) {conn.rollback(); }
1.13:             logger.error("Exception occurred"); }
1.14:  finally {
1.15:     if(statement != null) statement.close();
1.16:     if(conn != null) conn.close();
1.17:     if(ods != null) ods.close();
1.18: }
```

(a)

```
2.1:   Connection conn = null;
2.2:   Statement stmt = null;
2.3:   BufferedWriter bw = null; FileWriter fw = null;
       try {
2.4:       fw = new FileWriter("output.txt");
2.5:       bw = BufferedWriter(fw);
2.6:       conn = DriverManager.getConnection("jdbc:pl:db", "ps", "ps");
2.7:       Statement stmt = conn.createStatement();
2.8:       ResultSet res = stmt.executeQuery("SELECT Path FROM Files");
2.9:       while (res.next()) {
2.10:              bw.write(res.getString(1));
2.11:      }
2.12:      res.close();
2.13: } catch (IOException ex) { logger.error("IOException occurred");
2.14: } finally {
2.15:     if(stmt != null) stmt.close();
2.16:     if(conn != null) conn.close();
2.17:     if(bw != null) bw.close();
2.18: }
```

(b)

Figure 3. Two code samples from real applications. (a) This code attempts to modify the contents of a database through the method call Statement.executeUpdate (line 1.9). (b) This code attempts to read the contents of a database through the method call Statement.executeQuery (line 2.8).

Consider the two code samples shown in Figure 3, which are extracted from real applications. The code example in Figure 3a attempts to *modify* the contents of a database through the method call Statement.executeUpdate (line 1.9), whereas the code example in Figure 3b attempts to *read* the contents of a database through the method call Statement.executeQuery (line 2.8). A simple specification in the form of an association rule "Connection creation ⇒ Connection rollback" indicates that a rollback method call should occur in exception paths whenever an object of Connection is created. However, this form of specification is not a real rule since the rollback method call should be invoked only when changes are made to the database.

We propose *sequence association rules*[9] of the form "$(FC_c^1 \ldots FC_c^n) \wedge FC_a \Rightarrow (FC_e^1 \ldots FC_e^m)$," which prescribes that method-call sequence $FC_e^1 \ldots FC_e^m$ should follow FC_a in exception paths only when method-call sequence $FC_c^1 \ldots FC_c^n$ precedes FC_a. Thus, the sequence association rule for the code samples shown in Figure 3 is expressed as "$(FC_c^1\ FC_c^2) \wedge FC_a \Rightarrow (FC_e^1)$," where FC_c^1 is OracleDataSource.getConnection, FC_c^2 is Connection.createStatement, FC_a is Statement.executeUpdate, and FC_e^1 is Connection.rollback.

We have developed an algorithm that annotates items in sequences, mines the frequent subsequences of these sequences, and postprocesses the mined frequent subsequences to produce sequence association rules. We applied the new algorithm on static sequences extracted from code examples (found on the Internet) that use third-party APIs found in five real-world applications (including 285 KLOC), mining 294 real exception-handling rules and detecting 160 exception-handling defects.

MINING GRAPHS

Example SE graph data includes static or dynamic call graphs as well as program dependence graphs,[10] where edges represent data or control dependence, and nodes represent statements. Program executions are directed by the evaluations of various predicates throughout a program (for example, in if and while statements), so the executions can be modeled as traversals on static call graphs, leading to dynamic call graphs.

Common types of graph mining algorithms include frequent subgraph mining, graph matching, graph classification, and graph clustering. Frequent subgraph mining can be used to find programming rules, which manifest as frequent subgraphs in program dependence graphs that are extracted from code bases, while graph matching can be used to find locations for almost, but not exactly, matching mined subgraphs corresponding to potential bugs.[10]

We have developed new graph mining algorithms including discriminative graph mining and graph classification to assist in debugging.

Discriminative graph mining

Running a set of test cases over an instrumented program would produce two sets of traces: one corresponding to correct executions of passing test cases and the other to erroneous executions of failing test cases. Each of the traces could be "coiled" to form dynamic call graphs. A node in the call graph corresponds to method calls, and a transition in the graph corresponds to the various relationships among the method calls—for example, a method is immediately called after another method returns, and one method invokes another. Such coiling would produce two sets of dynamic call graphs. We could also build a similar graph from executions of basic blocks—a sequence of statements without any jump—in a program. Figure 4

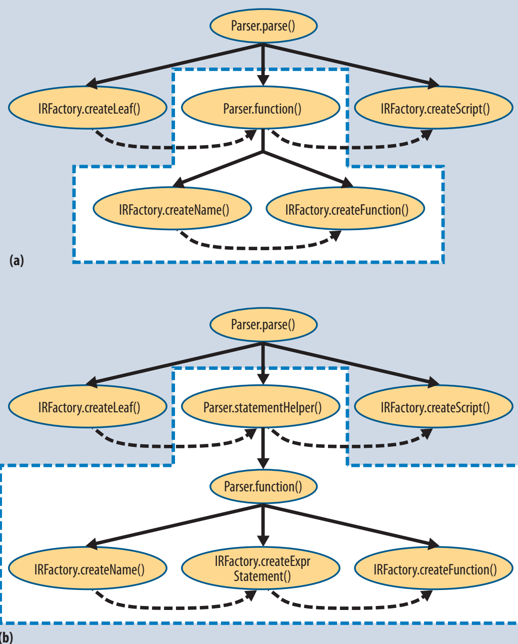


Figure 4. Examples of dynamic call graphs from executions of Mozilla Rhino: (a) partial graph for a correct execution; (b) partial graph for an erroneous execution.

shows examples of such graphs recovered from executions of Mozilla Rhino.

To extract features, in the form of subgraphs, that distinguish the two sets of dynamic call graphs, we have developed an algorithm that mines the top-k most discriminating graphs.[11] These graphs serve as signatures and context where bugs occur and could potentially help software engineers to locate and fix them. Our empirical results on benchmark datasets show an average of 74 percent precision and 91 percent recall at method level and an average of 59 percent precision and 73 percent recall at basic block level.

Graph classification

The dynamic call graph of an erroneous execution can sometimes diverge from those of correct executions.[12] Knowing at what time the divergence occurs would shed light on the potential bug location. Given that call graphs look roughly the same between correct and erroneous executions until the buggy place is executed, we use incremental classification to detect when the buggy place is triggered. Specifically, we build a support vector machine classifier at different execution points based on the dynamic call graphs accumulated so far from correct and erroneous executions; intuitively, the classification accuracy will stay low until the buggy point has been executed. Therefore, by detecting the accuracy boost, we can determine the bug location.

To classify dynamic call graphs, we have developed a closed graph mining algorithm to represent graphs by a series of essential subgraphs.

MINING TEXT

Example SE text data include bug reports, e-mails, code comments, and documentation for API methods. A unit of common concern is a text document similar to a sequence in the sequence database for sequence data.

Common types of text mining algorithms include text clustering, classification, and matching. Example text clustering applications include clustering bug reports to detect duplicate bug reports and thereby reduce inspection efforts, and assigning reports to specific developers to fix the bugs. Example text classification applications include recommending assignment of a new bug report to a specific developer based on the past assignment of old bug reports. Example text matching applications include searching keywords in code comments, API documentation, or bug reports, and detecting duplicates of a given bug report among old reports.

We have developed an approach for duplicate-bug-report detection[13] that helps avoid assigning duplicate bug reports to different developers. We illustrate the approach with two examples using reports from the Firefox bug repository.

Example 1

Consider the following bug summary documents:

Bug-219232: random “The Document contains no data.” Alerts
Bug-244372: “Document contains no data” message on continuation page of NY Times article

To detect Bug-244372 to be a duplicate of Bug-219232 among all existing reports, we represent the summary document of each as an n-dimensional vector, where n is the number of unique index terms (words) occurring in all the documents and each index term has a weight, being the vector value of that index, calculated based on the following rationale: If the index term t occurs in more documents, it receives less weight than that given to index terms that occur in fewer documents since t is not a good discriminator across documents.

Before transforming documents to vectors, we use common natural-language processing (NLP) techniques such as removing stop words—words that carry little meaning such as “on” or “of” in the Bug-244372 summary. After transforming documents into vectors, we calculate the similarity of the summary document of each bug report in the existing bug repository with the summary document of Bug-244372 through a formula for defining the similarity of two vectors.

Our approach determines that Bug-219232 has the highest similarity with Bug-244372. Hence, it would mark Bug-244372 as a potential duplicate of Bug-219232.

Example 2

Now consider the following bug summary documents from the Firefox bug repository:

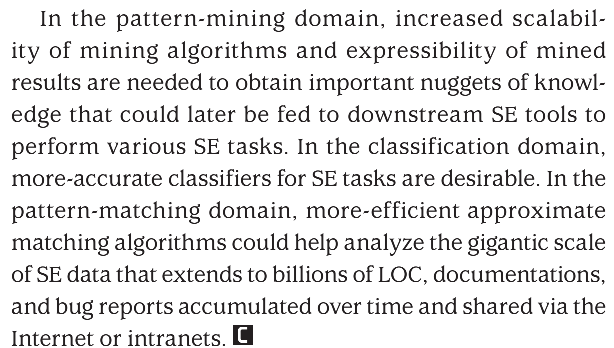
> Bug-260331: After closing Firefox, the process is still running. Cannot reopen Firefox after that, unless the previous process is killed manually.
> Bug-239223: (Ghostproc) - [Meta] Firefox.exe doesn't always exit after closing all windows; session-specific data retained.

Both documents use terms such as "Firefox" and "after closing," but these terms also commonly appear in other unrelated bug reports in the Firefox bug repository. In addition, inherent NLP challenges are evident: "still running" and "retained" should be treated as having the equivalent meaning. Even if common synonym lists are used, however, it is still difficult for NLP techniques to treat these two phrases as equivalent. Thus, using the technique of comparing vectors would fail in this case. Fortunately, by comparing the execution traces for the failing test cases corresponding to these two bug reports, we can detect that the execution traces are highly similar and are dissimilar to the execution traces for other reports in the bug repository.

This observation suggests that we should consider both summary documents and execution traces of bug reports in detecting duplicate reports. Specifically, we first calculate the NLP-based similarities between the summary document of the new bug report and the summary documents of the existing reports. Second, we calculate the execution-trace-based similarities between the new bug report and existing reports—for example, with sequence matching. Finally, we identify potential duplicate bug reports for the new report using the two types of similarities based on some combination heuristics.

Our empirical results show that our approach, which combines the analysis of both textual information and execution traces, can identify 67 to 93 percent of duplicate bug reports in the Firefox bug repository, compared with 43 to 72 percent using information from text alone.

The huge, growing amount of SE data provides many opportunities for future research. Integrating additional, more effective data mining tools that address practical SE problems into popular SE environments such as Eclipse and Microsoft Visual Studio will facilitate widespread adoption of data mining solutions in SE. However, this long-term goal does not come without obstacles. Currently, much work needs to be done to further adapt general-purpose mining algorithms or develop specific algorithms to satisfy the unique requirements of SE data and tasks. Further, the scope of SE tasks that can benefit from data mining must be expanded as well as the range of SE data that can be mined.



In the pattern-mining domain, increased scalability of mining algorithms and expressibility of mined results are needed to obtain important nuggets of knowledge that could later be fed to downstream SE tools to perform various SE tasks. In the classification domain, more-accurate classifiers for SE tasks are desirable. In the pattern-matching domain, more-efficient approximate matching algorithms could help analyze the gigantic scale of SE data that extends to billions of LOC, documentations, and bug reports accumulated over time and shared via the Internet or intranets. C

Acknowledgment

This work is supported in part by NSF grant CCF-0725190 and ARO grant W911NF-08-1-0443.

References

1. J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2000.
2. M. Acharya et al., "Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications," *Proc. 6th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.* (ESEC-FSE 07), ACM Press, 2007, pp. 25-34.
3. D. Lo, S-C. Khoo, and C. Liu, "Efficient Mining of Iterative Patterns for Software Specification Discovery," *Proc. 13th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (KDD 07), ACM Press, 2007, pp. 460-469.
4. D. Lo et al., "Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach," *Proc. 15th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining* (KDD 09), ACM Press, 2009, pp. 557-566.
5. D. Lo, S-C. Khoo, and C. Liu, "Mining Temporal Rules for Software Maintenance," *J. Software Maintenance and Evolution: Research and Practice*, July 2008, pp. 227-247.
6. D. Lo, S-C. Khoo, and C. Liu, "Mining Past-Time Temporal Rules: A Dynamic Analysis Approach," *Artificial Intelligence Applications for Improved Software Eng. Development: New Prospects*, F. Mezaine and S. Vadera, eds., IGI Global, 2009, Chap. 13, pp. 259-277.
7. D. Lo and S. Maoz, "Mining Scenario-Based Triggers and Effects," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 08), IEEE Press, 2008, pp. 109-118.
8. D. Lo and S-C. Khoo, "SMArTIC: Towards Building an Accurate, Robust and Scalable Specification Miner," *Proc. 14th ACM SIGSOFT Int'l Symp. Foundations on Software Eng.* (FSE 06), ACM Press, 2006, pp. 265-275.
9. S. Thummalapenta and T. Xie, "Mining Exception-Handling Rules as Sequence Association Rules," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), ACM Press, 2009, pp. 496-506.

10. R-Y. Chang, A. Podgurski, and J. Yang, "Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software," *Proc. 2007 Int'l Symp. Software Testing and Analysis* (ISSTA 07), ACM Press, 2007, pp. 163-173.
11. H. Cheng et al., "Identifying Bug Signatures Using Discriminative Graph Mining," *Proc. 2009 Int'l Symp. Software Testing and Analysis* (ISSTA 09), ACM Press, 2009, pp. 141-152.
12. C. Liu et al., "Mining Behavior Graphs for 'Backtrace' of Noncrashing Bugs," *Proc. SIAM Int'l Data Mining Conf.* (SDM 05), Soc. for Industrial and Applied Mathematics, 2005, pp. 286-297.
13. X. Wang et al., "An Approach to Detecting Duplicate Bug Reports Using Natural Language and Execution Information," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), ACM Press, 2008, pp. 461-470.

Tao Xie *is an assistant professor in the Department of Computer Science at North Carolina State University. His research interests include software engineering, particularly mining software engineering data and automated software testing. Xie received a PhD in computer science from the University of Washington. He is a member of the IEEE and the ACM. Contact him at xie@csc.ncsu.edu.*

Suresh Thummalapenta *is a PhD student in the Department of Computer Science at North Carolina State University. His research interests include automated software engineering, with an emphasis on mining software engineering data and software verification. Thummalapenta received an MS in computer science from North Carolina State University. He is a member of the ACM. Contact him at sthumma@ncsu.edu.*

David Lo *is an assistant professor in the School of Information Systems at Singapore Management University. His research interests include dynamic program analysis, specification mining, and pattern mining. Lo received a PhD in computer science from the National University of Singapore. He is a member of the IEEE and the ACM. Contact him at davidlo@smu.edu.sg.*

Chao Liu *is a researcher at Microsoft Research in Redmond, Washington. His research interests include data mining, machine learning, statistical methods and their applications to software engineering, and Internet services. Liu received a PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE and the ACM. Contact him at chaoliu@microsoft.com.*

13 magazines—one source • **FREE** articles on hot topics • Blogs, podcasts, & more

computing now
ACCESS | DISCOVER | ENGAGE
http://computingnow.computer.org