Intelligent Data Analysis 15 (2011) 413–441
DOI 10.3233/IDA-2010-0475
IOS Press

Data mining in software engineering

M. Halkidi[a], D. Spinellis[b], G. Tsatsaronis[c] and M. Vazirgiannis[c,*]
[a]*Department of Digital Systems, University of Piraeus, Piraeus, Greece*
[b]*Department of Management Science and Technology, Athens University of Economics and Business, Athens, Greece*
[c]*Department of Informatics, Athens University of Economics and Business, Athens, Greece*

Abstract. The increased availability of data created as part of the software development process allows us to apply novel analysis techniques on the data and use the results to guide the process's optimization. In this paper we describe various data sources and discuss the principles and techniques of data mining as applied on software engineering data. Data that can be mined is generated by most parts of the development process: requirements elicitation, development analysis, testing, debugging, and maintenance. Based on this classification we survey the mining approaches that have been used and categorize them according to the corresponding parts of the development process and the task they assist. Thus the survey provides researchers with a concise overview of data mining techniques applied to software engineering data, and aids practitioners on the selection of appropriate data mining techniques for their work.

Keywords: Data mining techniques, KDD methods, mining software engineering data

1. Introduction

The recent advances in data management technology provide us with the tools and methods for efficient collection, storage and indexing of data. Large amount of data is produced in software development that software organizations collect in hope of extracting useful information and obtain better understanding of their processes and products. Also a significant amount of Open Source Software (OSS) project metadata is collected and maintained in software repositories. For instance, the FLOSSmole[1] project integrates data from several OSS projects and freely provides them in several formats. Then there is an increasing abundance of data stored in software engineering (SE) repositories which can play a significant role in improving software productivity and quality. Specifically, data in software development can refer to programs versions, execution traces, error/bug reports and open source packages. Mailing lists, discussion forums and newsletters also provide useful information about a piece of software. This explosive growth of software engineers' ability to collect and store SE data has created a need for new, scalable and efficient, tools for data analysis.

Data mining provides the techniques to analyze and extract novel, interesting patterns from data. Formally, it has been defined as the process of inducing previously unknown and potentially useful information from data collections. Thus mining of software engineering data has recently attracted the interest

*Corresponding author. E-mail: mvazirg@aueb.gr.
[1]http://flossmole.org/.

of researchers, emerging as a promising means to meet the goal of software improvement [58]. The extracted patterns of knowledge can assist software engineers in predicting, planning, and understanding various aspects of a project so that they can more efficiently support future development and project management activities. There are several challenges that emerge in mining software repositories [57]:

– *Complex Data*. Software engineers usually use individual data types to perform a software engineering task. However, software engineering domain can be complex and thus software engineering tasks increasingly demands the mining of multiple correlated data types to achieve the most effective results. Moreover, there are cases that significant information is not only associated with individual data items but with the linkage among them. Thus the requirement of techniques that analyze complex data types taking also into account the links among SE data is stronger than ever in software engineering.
– *Large-scale data*. Huge amount of data are collected and stored in software engineering repositories. In most of the cases software engineers analyze only a few local repositories to accomplish their tasks. However, there may be too few relevant data points in the local software engineering repositories to support the mining of desirable patterns (e.g. extract patterns among API methods of interests). This problem can be addressed if we are able to mine Internet-scale software repositories. Also mining techniques can be applied to many software repositories within an organization or across organization or even to the entire open source world. Thus the requirement for techniques that enable efficient mining of large scale data arises.

In this paper, we present an overview of approaches that aim to connect the research areas of data mining and software engineering leading to more efficient techniques for processing software. In Section 2, we provide an introduction to the main data mining concepts and approaches while in Section 3 we describe the different types of software engineering data that can be mined. In the follow up, a discussion takes place in Section 4 concerning the methods that have applied data mining techniques in the context of software engineering. It surveys the current research that incorporates data mining in software engineering while it discusses on the main characteristics of the respective approaches. Specifically, our study is based on the following main features of the approaches: i) data mining technique used (Section 2), ii) the software engineering data to which they are applied (Section 3), and iii) the software engineering tasks that they can help (Section 4). Thus, this paper aims to introduce practitioners to the fundamental concepts and techniques that can use in order to obtain a better understanding of the software engineering processes and potentially perform them more efficiently by applying data mining. In parallel, researchers can exploit the recent techniques to better understand the project data, to identify the limitations of the current processes and define methodologies that facilitate software engineering tasks.

To summarize, the key contributions of this work are:

– A classification of the approaches used to mine software engineering data, according to the software engineering areas that assist.
– Matrix based analysis framework bridging software engineering with data mining approaches.
– Bringing together data mining and software engineering research areas. A number of approaches that use data mining in software engineering tasks are presented providing new work directions to both researchers and practitioners in software engineering.

2. Data mining and knowledge discovery

The main goals of data mining are *prediction* and *description*. *Prediction* aims at estimating the future value or predicting the course of target variables based on study of other variables. *Description* is focused on patterns discovery in order to aid data representation towards a more comprehensible and exploitable manner. A good *description* suggests a good explanation of data behavior. The relevant importance of *prediction* and *description* varies for different data mining applications. However, as regards the knowledge discovery, *description* tends to be more important than *prediction*, contrary to the pattern recognition and machine learning application, for which *prediction* is more important.

Data mining assists with software engineering tasks by explaining and analyzing in depth software artifacts and processes. Based on data mining techniques we can extract relations among software projects. Data mining can exploit the extracted information to evaluate the software projects and/or predict software behavior. A number of data mining methods have been proposed to satisfy the requirements of different applications. All of them accomplish a set of data mining functionalities to identify and describe interesting patterns of knowledge extracted from a data set. Below we briefly describe the main data mining tasks and how they can be used in software engineering.

– *Clustering – Unsupervised Learning Techniques*
Clustering is one of the most useful tasks in data mining, applied for discovering groups and identifying interesting distributions and patterns in the underlying data. The clustering problem is about partitioning a given data set into groups (clusters) such that the data points in a cluster are more similar to each other than points in different clusters [28,33]. In the clustering process, there are no predefined classes and no examples that foretell the kind of desirable relations being valid among the data. Thus it is perceived as an unsupervised learning process [5]. Clustering can be used to produce a view of the underlying data distribution as well as automatically identify data outliers. In software engineering, clustering can be used to define groups of similar modules based on the number of modifications and cyclomatic number metrics (the number of linearly independent paths through a program's source code).
– *Classification – Supervised Learning Techniques*
The classification problem has been studied extensively in the statistics, pattern recognition and machine learning areas of research as a possible solution to the knowledge acquisition or knowledge extraction problem [15,55]. Classification is one of the main tasks in data mining for assigning a data item to a predefined set of classes. Classification can be described as a function that maps (classifies) a data item into one of the several predefined classes [18]. A well-defined set of classes and a training set of pre-classified examples characterize the classification. On the contrary, the clustering process does not rely on predefined classes or examples [5]. The goal in the classification process is to induce a model that can be used to classify future data items whose classification is unknown. One of the widely used classification techniques is the construction of *decision trees*. They can be used to discover classification rules for a chosen attribute of a data set by systematically subdividing the information contained in this data set. *Decision trees* are also one of the tools that have been chosen for building classification models in the software engineering field. Figure 1 shows a classification tree that has been constructed to provide a mechanism for identifying risky software modules based on attributes of the module and its system. Thus, based on the given decision tree, we can extract the following rule that assists with making decision on errors in a module:

IF(*# of data bindings* $>$ *10*) *AND* (*it is part of a non real-time system*) **THEN**
the module is unlikely to have errors

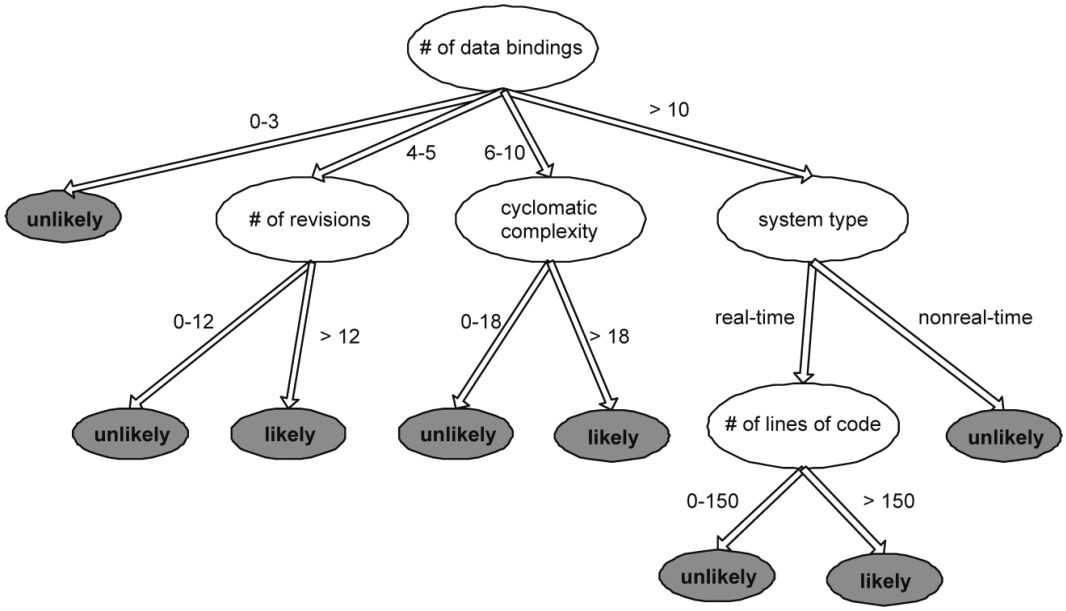

Fig. 1. Classification tree for identifying risky software modules.

– *Frequent Pattern Mining and Association Rules.* Association rules mining has attracted considerable interest because the rules provide a concise way to state potentially useful information that is easily understood by the end-users. Association rules reveal underlying "correlations" between the attributes in the data set. These correlations are presented in the following form: $A \rightarrow B$, where A, B refer to attribute sets in underlying data. Therefore, they are used to extract information based on coincidences in a data set. For instance, analyzing system's error logs discovered at software modules we can extract relations between inducing events based on the software module features and errors' categories. Such a rule would have the following form:
(*large/small size, large/small complexity, number of revisions*) $\rightarrow$ (*interface error, missing or wrong functionality, algorithms or data structure error etc.*)
– *Data Characterization and Summarization.* Data characterization [23] is the summarization of the data characteristics of a specific class of data. The data are collected based on user-specified requirements. These techniques can be used to discover a set of patterns from software engineering repositories that satisfy specific characteristics.
– *Change and Deviation Detection* focuses on discovering the most significant changes in the data from previously measured values. Thus these techniques can assist with identifying source code changes or identifying differences among extracted patterns from software engineering repositories.

Various approaches have been developed to accomplish the above mentioned data mining tasks and deal with different types of data. They exploit techniques from different aspects of data management and data analysis, including pattern recognition, machine learning, statistics, information retrieval, concept and text analysis.

Text Mining is introduced as a specific case of data mining and refers to the process of deriving information from text. Software engineering repositories, among others, include textual information like source code, mailing lists, bug reports and execution logs. The mining of textual artifacts is requisite for many important activities in software engineering: tracing of requirements; retrieval of components

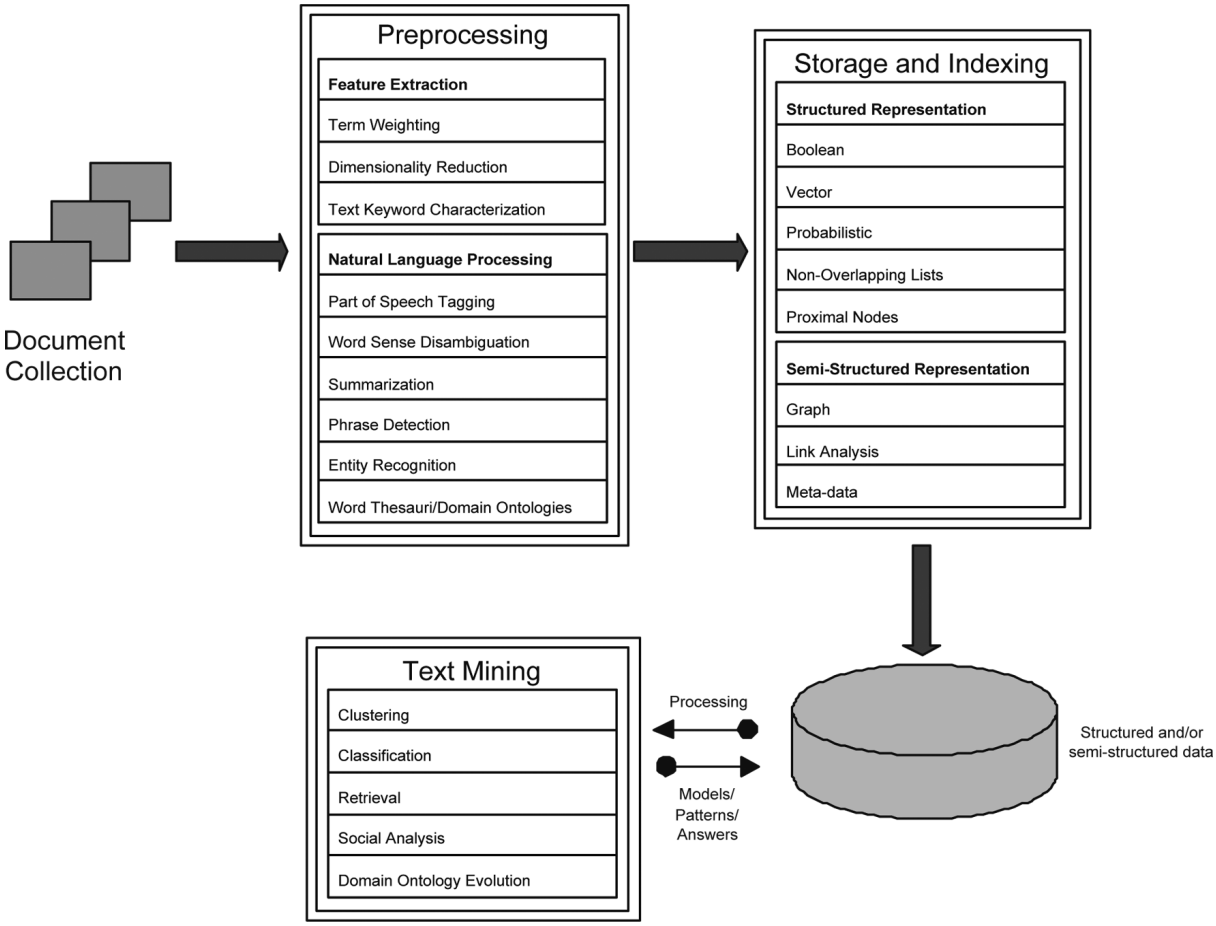

Fig. 2. Preprocessing, Storage and Processing of Texts in Text Mining.

from a repository; identify and predict software failures; software maintenance; testing etc. The methods deployed in text mining, depending on the application, usually require the transformation of the texts into an intermediate structured representation, which can be for example the storage of the texts into a database management system, according to a specific schema. In many approaches though, there is gain into also keeping a semi-structured intermediate form of the texts, like for example the representation of documents in a graph, where social analysis and graph techniques can be applied. Independently from the task objective, text mining requires preprocessing techniques, usually levying qualitative and quantitative analysis of the documents' features. In [10,2], several preprocessing and feature analysis techniques are discussed for text mining. In Fig. 2, the diagram depicts the most important phases of the preprocessing analysis, as well as the most important text mining techniques.

3. Software engineering data

The nature of the data being used by data mining techniques in software engineering can act as distinguishing means of the underlying methods, since it affects the preprocessing as well as the post

analysis. Below we present the various sources of software engineering data to which data mining has been applied. The presentation also tries to reflect the difficulty of preparing the data for processing.

3.1. Documentation

Software documentation data are of high importance but in tandem of high complexity for being processed by data mining techniques. Application, system administration and source code documentation constitute a large buffer of documents and free text for software analysis and mining. Among the pieces of text information that can be considered of great value for use in mining techniques are the software description, start up and usage configuration, user guide, file management issues, logging, license and compatibility issues. Besides external software documentation, internal documentation might also play the role of important data source. Most of the documentation though lies in several different types of documents, like portable document format, html, text only and typesetting system files. An analytical reference of all possible types of software documentation data can be found in [53]. Due to the large variety of document types and text data used, it is necessary that a preprocessing module for documentation data is able to use parsers of all the aforementioned types of documents. Another important source of information that lies in software documentation are the multimedia data. Figures, as well as audio and video instructions, can all be considered of added information value. In such cases, multimedia mining techniques must be incorporated to tackle with the pre- and post-processing, raising the overall processing overhead.

3.2. Software configuration management data

Data rising from software configuration management systems (SCMs) among others may include software code, documents, design models, status accounting, defect tracking as well as revision control data (documentation and comments escorting software versions in the adopted CVS). In [17] the evolution of SCMs from the early days of software development to present is discussed, where additionally the impact of research in the field of SCMs is depicted. Independently of the underlying version control system (centralized or distributed) the amount of data available from SCMs is large and thus a careful study and clean understanding of the software domain is needed, so thus the most valuable data are kept. The majority of the SCMs data is structured text.

Decentralized source code management (DSCM) systems show a significant growth the last years. Bird et al. have studied the main properties of theses systems in [7] and discussed the advantages and risks that decentralization brings in mining software engineering data. DSCMs can provide software engineering researchers with new and useful data which enable them to better understand software processes. However the DSCM data should be mined with care. Bird et al. in their work have noted potential pitfalls that one may encounter when analyzing this data since there are differences in semantics of terms used in centralized and decentralized source code management systems.

3.3. Source code

Source code for data mining in software engineering can be proved an important source of data. Various data mining applications in software engineering have employed source code to aid software maintenance, program comprehension and software components' analysis. The details of these approaches are discussed in section 4. An initial preprocessing of the available source code is always a caveat, since a parser for the respective source code language must be available. Once parsed, the source

code can be seen as structured text. Central aspects of applying data mining techniques in source code among others include prediction of future changes through mining change history, predicting change propagation, faults from cached history, as well as predicting defect densities in source code files.

3.4. Compiled code and execution traces

Compiled code constitutes in its form of object code one of the alternative data sources for applying static analysis in software engineering. Compiled code has also been used as a data source from data mining techniques in order to assist malicious software detection. Furthermore, web mining principles have been widely used in object-oriented executables to assist program comprehension for means of reverse engineering. When the software modules and components are tested, a chain of events occurs which are recorded in an execution trace. Execution pattern mining has also been used in execution traces under the framework of dynamic analysis to assist with the extraction of software systems' functionalities.

3.5. Issue-tracking and bug databases

Issue-tracking or bug reporting databases constitute the most important cesspool of issue reporting in software systems. Structured data (database tuples) containing the description of an issue, the reporter's details and date/time are the standard three types of information that can be found in issue-tracking databases. Machine learning techniques have been successfully used in the past to predict correct assignments of developers to bugs, cleaning the database from manifestations of the same error, or even predicting software modules that are affected at the same time from reported bugs.

3.6. Mailing lists

Large software systems, and especially open source software, offer mailing lists as a means of bridging users and developers. Mailing lists constitute hard data since they contain a lot of free text. Message and author graphs can be easily pulled up from the data, but content analysis is hard since probably messages constituting replies need to consider initial and/or previous discussions in the mailing lists. Data mining applications in mailing lists among others include but are not limited to text analysis, text clustering of subjects discussed, and linguistic analysis of messages to highlight the developers personalities and profiles.

4. Data mining for software engineering

Due to its capability to deal with large volumes of data and its efficiency to identify hidden patterns of knowledge, data mining has been proposed in a number of research work as mean to support industrial scale software maintenance, debugging, testing. The mining results can help software engineers to predict software failures, extract and classify common bugs, identify relations among classes in a libraries, analyze defect data, discover reused patterns in source code and thus automate the development procedure. In general terms, using data mining practitioners and researchers can explore the potential of software engineering data and use the mining results in order to better manage their projects and to produce higher quality software systems that are delivered on time and on budget. In the following sections we discuss the main features of mining approaches that have been used in software engineering and how the results can be used in the software engineering life cycle. We classify the approaches according to the software engineering tasks that they help and the mining techniques that they use.

Table 1
Mining approaches used in Requirement Elicitation and Tracing

Requirement Elicitation		
MINING APPROACH	INPUT DATA	DATA ANALYSIS RESULTS
Classification ([26],[27])	Documentation	requirements
Information [21] retrieval, Data Summarization	SCM, mailing lists CVS logs	requirements

4.1. Requirement elicitation and tracing

In this section we discuss how data analysis techniques can contribute to educe or trace system requirements. The works for requirement analysis refers to data mining in its broadest sense, including certain related activities and methodologies from statistics, machine learning and information retrieval. Table 1 summarizes the main features of the techniques discussed below.

4.1.1. Classification

A recent approach, presented in [26], has focused on improving the extraction of high level and low level requirements using information retrieval. More specifically, they consider the documents' universe as being the union of the design elements and the individual requirements and they map the problem of requirements tracing into finding the similarities between the vector-space representations of high level and low level requirements, thus reducing it into an IR task. As an expansion of this study, in [27], the authors focused on discovering the factors that affect an analysts' behavior when working with results from data mining tools in software engineering. The whole study was based on the verified hypothesis that *the accuracy of computer-generated candidate traces affects the accuracy of traces produced by the analyst*. The study presents how the performance of tools that extract high level and low level requirements through the use of information retrieval, affects the time consumed by an analyst to submit feedback, as well as her performance. Results reveal that data mining systems exhibiting low recall result in a time consuming feedback from the analyst. In parallel, high recall leads to a large number of false positive thus prompting the analyst cut down large number of requirements, dimming recall. Overall reported results reveal that the analyst tends to balance precision and recall at the same levels.

4.1.2. Data summarization

From another perspective, text mining has been used in software engineering to validate the data from mailing lists, CVS logs, and change log files of open source software. In [21] they created a set of tools, namely SoftChange[2], that implements data validation from the aforementioned text sources of open source software. Their tools retrieve, summarize and validate these types of data of open source projects. Part of their analysis can mark out the most active developers of an open source project. The statistics and knowledge gathered by SoftChange analysis has not been exploited fully though, since further predictive methods can be applied with regard to fragments of code that may change in the future, or associative analysis between the changes' importance and the individuals (i.e. were all the changes committed by the most active developer as important as the rest, in scale and in practice?).

[2]Publicly available at http://sourcechange.sourceforge.net/.

Table 2
Mining approaches used in Software Development

Development		
MINING APPROACH	INPUT DATA	DATA ANALYSIS RESULTS
Clustering [29] (Social network analysis)	source code	software processes
Classification [56] text retrieval	SCM, source code	track of bugs
Frequent pattern mining [44] and Association Rules	defect data	defect correction effort rules
Frequent pattern mining [11] and Association Rules	program dependence graph	neglected conditions

4.2. Development analysis

This section provides an overview of mining approaches used to assist with development process. We summarize the main features of these approaches in Table 2.

4.2.1. Clustering

Text mining has also been used in software engineering for discovering development processes. Software processes are composed of events such as relations of agents, tools, resources, and activities organized by control flow structures dictating that sets of events execute in serial, parallel, iteratively, or that one of the set is selectively performed. Software process discovery takes as input artifacts of development (e.g. source code, communication transcripts, etc.) and aims to elicit the sequence of events characterizing the tasks that led to their development. In [29] an innovative method of discovering software processes from open source software Web repositories is presented. Their method contains text extraction techniques, entity resolution and social network analysis, and it is based on the process of entity taxonomies. Automatic means of evolving the taxonomy using text mining tasks could have been levied, so that the method lacks strict dependency on the taxonomy's actions, tools, resources and agents. An example could be the use of text clustering on the open software text resources and extraction of new candidate items for the taxonomy arising from the clusters' labels.

In [6], they used as text input the Apache developer mailing list. Entity resolution was essential, since many individuals used more than one alias. After constructing the social graph occurring from the interconnections between poster and replier, they made a social network analysis and came to really important findings, like the strong relationship between email activity and source code level activity. Furthermore, social network analysis in that level revealed the important nodes (individuals) in the discussions. Though graph and link analysis were engaged in the method, the use of node ranking techniques, like PageRank, or other graph processing techniques like Spreading Activation, did not take place.

4.2.2. Classification

Source code repositories stores a wealth of information that is not only useful for managing and building source code, but also provide a detailed log how the source code has evolved during development. Information regarding the evidence of source code refactoring will be stored in the repository. Also as bugs are fixed, the changes made to correct the problem are recorded. As new APIs are added to the source code, the proper way to use them is implicitly explained in the source code. Then, one of the challenges is to develop tools and techniques to automatically extract and use this useful information.

In [56], a method is proposed which uses data describing bug fixes mined from the source code repository to improve static analysis techniques used to find bugs. It is a two step approach that uses the source code change history of a software project to assist with refining the search for bugs.

The first step in the process is to *identify the types of bugs* that are being fixed in the software. The goal is to review the historical data stored for the software project, in order to gain an understanding of what data exists and how useful it may be in the task of bug findings. Many of the bugs found in the CVS history are good candidates for being detected by statistic analysis, NULL pointer checks and function return value checks.

The second step is to *build a bug detector* driven by these findings. The idea is to develop a function return value checker based on the knowledge that a specific type of bug has been fixed many times in the past. Briefly, this checker looks for instances where the return value from a function is used in the source code before being tested. Using a return value could mean passing it as an argument to a function, using it as part of calculation, de-referencing the value if it is a pointer or overwriting the value before it is tested. Also, cases that return values are never stored by the calling function are checked. Testing a return value means that some control flow decision relies on the value.

The checker does a data flow analysis on the variable holding the returned value only to the point of determining if the value is used before being tested. It simply identifies the original variable the returned value is stored into and determines the next use of that variable.

Moreover, the checker categorizes the warnings it finds into one of the following categories:

- Warnings are flagged for return values that are completely ignored or if the return value is stored but never used.
- Warnings are also flagged for return values that are used in a calculation before being tested in a control flow statement.

Any return value passed as an argument to a function before being tested is flagged, as well as any pointer return value that is de-referenced without being tested.

However there are types of functions that lead the static analysis procedure to produce false positive warnings. If there is no previous knowledge, it is difficult to tell which function does not need their return value checked. Mining techniques for source code repository can assist with improving static analysis results. Specifically, the data we mine from the source code repository and from the current version of the software is used to determine the actual usage pattern for each function.

In general terms, it has been observed that the bugs cataloged in bug databases and those found by inspecting source code change histories differ in type and level of abstraction. Software repositories record all the bug fixed, from every step in development process and thus they provide much useful information. Therefore, a system for bug finding techniques is proved to be more effective when it automatically mines data from source code repositories.

4.2.3. Frequent pattern mining and association rules

An approach is proposed in [44] that exploits association rules extraction techniques to analyze defect data. Software defects include bugs, specification and design changes. The collected defect data under analysis are nominal scale variables such as description of defect, priority to fix a defect and its status as well as interval and ratio scale variable regarding defect correction effort and duration. An extended association rule mining method is applied to extract useful information and reveal rules associated with defect correction effort.

The problem of discovering neglected conditions (missing paths, missing conditions, and missing cases) in software is studied in [11]. Chang et al. proposed the use of graph mining techniques to

Table 3
Mining approaches used in software testing

Testing		
MINING APPROACH	INPUT DATA	DATA ANALYSIS RESULTS
Classification[35]	I/O variables of software system	a network producing sets for function testing
Clustering [13]	execution profiles	clusters of execution profiles
Clustering, Classification [8]	program executions	software behavior classifiers

discover implicit conditional rules in a code base and to discover rule violations that indicate neglected condition. They represent programs and conditional programming rules in terms of dependence graphs. Then they use frequent item set mining and frequent subgraph mining techniques to discover conditional rules involving preconditions and postconditions of function calls as well as discover violations of those rules.

4.3. Testing

The evaluation of software is based on tests that are designed by software testers. Thus the evaluation of test outputs is associated with a considerable effort by human testers who often have imperfect knowledge of the requirements specification.

Data mining approaches can be used for extracting useful information from the tested software which can assist with the software testing. Specifically, the induced data mining models of tested software can be used for recovering missing and incomplete specifications, designing a set of regression tests and evaluating the correctness of software outputs when testing new releases of the system. A regression test library should include a minimal number of tests that cover all possible aspects of system functionality. To ensure effective design of new regression test cases, one has to recover the actual requirements of an existing system. Thus, a tester has to analyze system specifications, perform structural analysis of the system's source code and observe the results of system execution in order to define input-output relationships in tested software.

Table 3 summarizes the main data mining techniques that are used in the context of software testing.

4.3.1. Clustering

In [13] a method is proposed that exploits the cluster analysis methods to select the set of executions that will be evaluated for conformance to requirements. The proposed approach assumes a set of execution profiles that have been defined executing the software version under test on a given set of program inputs. A clustering algorithm is used to filter profiles based on their similar characteristics. Then execution profiles are selected from the resulting clusters.

An approach that aims to analyze a collection of programs' executions and define classifiers of software behavior is proposed in [8]. According to this work, Markov models are used to encode the execution of profiles of projects. Then the Markov models of individual program executions are clustered using an agglomerative clustering algorithm. The clustering procedure aims to aggregate the similar program execution and thus define effective classifiers of program behavior. Also a bootstrapping is used as an active learning technique so that the learning classifiers is trained in a incremental fashion. Specifically, it assists with identifying new training instances for the classifiers and then the classifiers can be retrained using the expanded set of training instances.

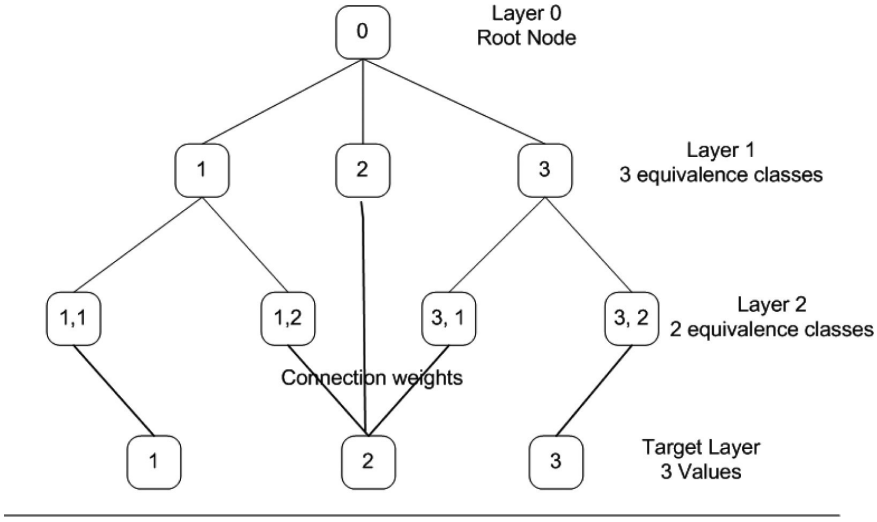

Fig. 3. An example of Info-Fuzzy Network structure.

4.3.2. Classification

An approach that aims to automate the input-output analysis of execution data based on a data mining methodology is proposed in [35]. This methodology relies on the *info-fuzzy network* (*IFN*) which has an 'oblivious' tree-like structure. The network components include the root node, a changeable number of hidden layers (one layer for each selected input) and the target (output) layer representing the possible output values. The same input attribute is used across all nodes of a given layer (level) while each target node is associated with a value (class) in the domain of a target attribute. If the IFN model is aimed at predicting the values of a continuous target attribute, the target nodes represent disjoint intervals in the attribute range.

A hidden layer *l*, consists of nodes representing conjunctions of values of the first *l* input attributes, which is similar to the definition of an internal node in a standard decision tree. The final (terminal) nodes of the network represent non-redundant conjunctions of input values that produce distinct outputs. Considering that the network is induced from execution data of a software system, each interconnection between a terminal and target node represents a possible output of a test case. Figure 3 presents an IFN structure.

A separate info-fuzzy network is constructed to represent each output variable.

The main modules of the IFN-based environment presented in [35] are:

- *Legacy system* (*LS*). This module represents a program, a component or a system to be tested in subsequent versions of the software.
- *Specification of Application Inputs and Outputs* (*SAIO*). Basic data on each input and output variable in the Legacy System.
- *Random test generator* (*RTG*). This module generates random combinations of values in the range of each input variable.
- *Test bed* (*TB*). This module feeds training cases generated by the RTG module to the LS.

The IFN algorithm is trained on inputs provided by RTG and outputs obtained from a legacy system by means of the Test Bed module. A separate IFN module is built for each output variable.

The IFN algorithm takes as input the training cases that are randomly generated by the RTG module and the outputs produced by LS for each test case. The IFN algorithm repeatedly runs to find a subset

Table 4
Mining approaches used in Debugging

Debugging		
MINING APPROACH	INPUT DATA	DEBUGGING RESULTS
Probabilistic classification [36]	test suite of program input and desired output	detection of logical errors
SVM classification [37]	program executions	logical bugs
Classification, Decision trees [19,46]	execution profiles & result(success/failure)	decision tree of failed executions
Frequent pattern mining and Association Rules [31]	source code	patterns of call-usage

of input variables relevant to each output and the corresponding set of non-redundant test cases. Actual test cases are generated from the automatically detected equivalence classes by using an existing testing policy.

4.4. Debugging

Program logic errors rarely incur memory access violations but generate incorrect outputs. A number of mining techniques have been used to identify logic error and assist with software debugging (see Table 4).

4.4.1. Classification

An approach that aims to investigate program logic errors is proposed in [36]. Liu et al. develop a data mining algorithm that can assist programmers' manual debugging. They introduce a statistical approach to quantify the bug relevance of each condition statement and then develop two algorithms to locate the possible buggy functions.

The proposed model considers a test suite $T_{i=1}^{n}$ for a program P, where each test case $t_i = (d_i, o_i)$ has an input d_i and the desired output o_i. We say that P passes the test case t_i if and only if the output of execution P on t_i is identical to o_i. Then we can partition the test suite T into two disjoint subsets T_p and T_f, corresponding to the passing and failing cases respectively. In this work, they instrument each condition statement in P to collect the evaluation frequencies at runtime. Specifically, they consider the boolean expression in each condition statement as one distinct *boolean feature*. Also assuming that X is the random variable for the boolean bias of a boolean feature B, we use $f(X/\theta_p)$ and $f\,(X/\theta_f)$ to denote the underlying probability model that generates the boolean bias of B for cases from T_p and T_f respectively. Then we claim that a boolean feature B is relevant to the program error if its underlying probability model $f(X/\theta_f)$ diverges from $f(X/\theta_p)$. If $L(B)$ is a similarity function, $L(B) = Sim(f(X/\theta_f), f(X/\theta_p))$, the bug relevance score of B can be defined as $s(B) = -\log(L(B))$. An open issue is the definition of a suitable similarity function. In [36], they introduce an approach based on a probabilistic model to approximate the values of $f(X/\theta_f)$ and $f(X/\theta_p)$ and define the bug relevance score for a boolean feature B. Moreover they propose two algorithms (*CombineRank* and *UpperRank*) to combine individual bug scores of B ($s(B)$) in order to define a global score s(F) for a function F.

Another method that exploits data mining methods to analyze logical bugs is proposed in [37]. In this work, they treat program executions as software behavior graphs and develop a method to integrate closed graph mining and SVM classification in order to isolate suspicious regions of non-crashing bugs. They consider that each execution of a program is summarized as a behavior graph. Then, given a set

of behavior graphs that are labeled either positive (incorrect runs) or negative (correct runs), the goal is to train a classifier to identify new behavior graphs with unknown labels. The proposed classification model consists of three steps:

- define the training dataset extracting features from behavior graphs
- learn an SVM classifier using these features
- classify new behavior graphs.

The graphs are represented as vectors in a feature space in order to apply SVM in behavior graph classification. A naive representation is to consider the edges as features and a graph as a vector of edges. The vector is $\{0, 1\}$ valued. It takes the value '1' in the dimension that corresponds to the feature(edges) that the graph has, and '0', otherwise. The similarity between two graphs is defined as the dot product of their edges. According to the above representation of graphs, the dot product of two feature vectors is the number of common edges that two graphs have. The hyperplane learned in this way will be a linear combination of edges. Thus it may not achieve good accuracy when a bug is characterized by multiple connected call and transition structures. Liu et al. observed that functions in well-designed programs usually exhibit strong modularity in source code and in dynamic executions. Also these functions are often grouped together to perform a specific task. The calls and transitions of these functions will be tightly related in the whole behavior graph. The buggy code may disturb the local structure of a run and then have an effect on its global structure. Based on this observations they propose to use recurrent local structures as features. They introduce the concept of frequent graphs and define a classification process that is based on them. Each frequent graph is treated as a separate feature in the feature vector. Hence, a behavior graph G is transformed into a feature vector whose i-th dimension is set to be 1 if G contains the i-th frequent graph or 0 otherwise. The authors in [37] proposed an approach for mining closed frequent graphs from a set of behavior graphs and then used them as features. Based on these features a classification model is trained so that assists programmers with debugging non-crashing bugs. Moreover, an approach that measures incrementally the classification accuracy changes aiming to identify suspicious regions in a software program.

4.4.1.1. Software failures classification

A semi-automated strategy for classifying software failures is presented in [46]. This approach is based on the idea that if m failures are observed over some period during which the software is executed, it is likely that these failures are due to a substantially smaller number of distinct defects. Assume that $F = \{f_1, f_2, \ldots, f_m\}$ is the set of reported failures and that each failure is caused by just one defect. Then F can be partitioned into $k < m$ subsets $F_1, F_2, \ldots, F_k$ such that all of the failures in F_i are caused by the same defect d_i, $1 \leqslant i \leqslant k$. This partitioning is called the *true failure classification*. In the sequel, we describe the main phases of the strategy for approximating the true failure classification:

1. The software is implemented to collect and transmit to the development either execution profiles or captured executions and then it is deployed.
2. Execution profiles corresponding to reported failures are combined with a random sample of profiles of operational executions for which no failures were reported. This set of profiles is analyzed to select a subset of all profile features to use in grouping related failures. A feature of an execution profile corresponds to an attribute or element of it. For instance, a function call profile contains an execution count for each function in a program and each count is a feature of the profile. Then the feature selection strategy is as follows:

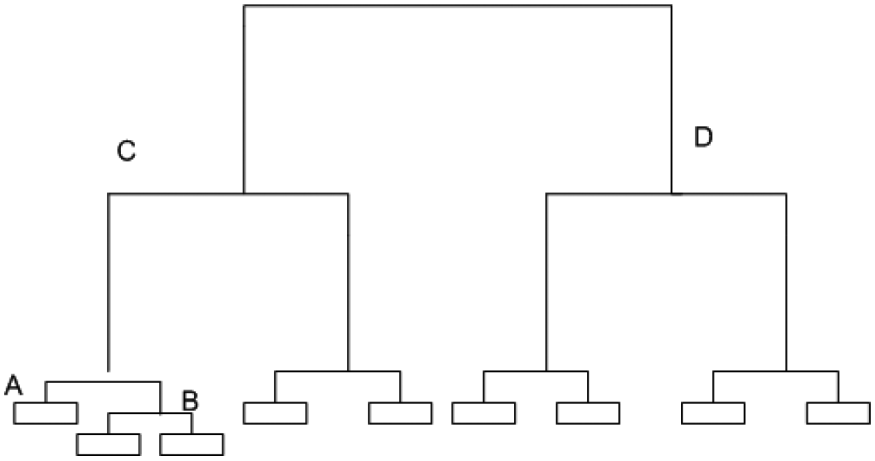

Fig. 4. A clusters' hierarchy.

- Generate candidate feature-sets and use each one to create and train a pattern classifier to distinguish failures from the successful executions.
- Select the features of the classifier that give the best results.

3. The profiles of reported failures are analyzed using cluster analysis, in order to group together failures whose profiles are similar with respect to the features selected in phase 2.
4. The resulting classification of failures into groups is explored in order to confirm it or refine it.

The above described strategy provides an initial classification of software failures. Depending on the application and the user requirements these initial classes can be merged or split so that the software failure are identified in an appropriate fashion.

In [19], two tree-based techniques for refining an initial classification of failures are proposed. Below we present the main idea of these approaches.

4.4.1.2. Refining failures classification using dendograms

One of the strategies that has been proposed for refining initial failure classification relies on tree-like diagram (known as dendrograms). Specifically, it uses them to decide how non-homogeneous clusters should be split into two or more sub-clusters and to decide which clusters should be considered for merging. A cluster in a dendrogram corresponds to a subtree that represents relationships among its sub-clusters. The more similar two clusters are to each other, the farther away from the dendrogram root their nearest common ancestor is. For instance, based on the dendrogram presented in Fig. 4 we can observe that the clusters A and B are more similar than the clusters C and D. A cluster's largest homogeneous subtree is the largest subtree consisting of failures with the same cause. If a clustering is too coarse, some clusters may have two or more large homogeneous subtrees containing failures with different causes. Such a cluster should be split at the level where its large homogeneous subtrees are connected, so that these subtrees become siblings as Fig. 6 shows. If it is too fine, siblings may be clusters containing failures with the same causes. Such siblings (clusters) should be merged at the level of their parent as Fig. 5 depicts.

Based on these definitions, the strategy that has been proposed for refining an initial classification of failures using dendrograms has three phases:

1. Select the number of clusters into which the dendrogram will be divided.

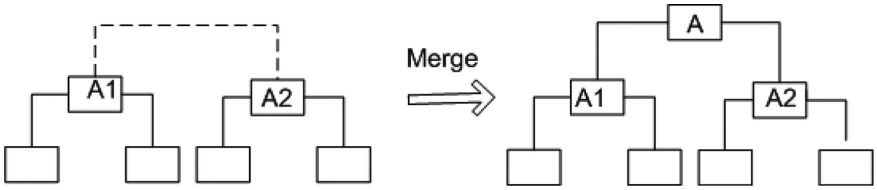

Fig. 5. Merging two clusters. The new cluster A contains the clusters represented by the two homogeneous sub-trees A1 and A2.

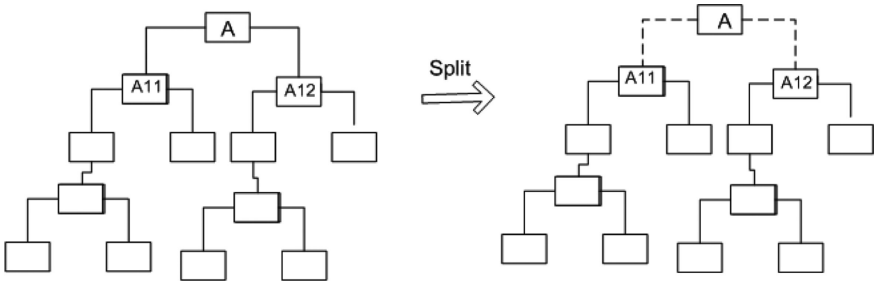

Fig. 6. Splitting a cluster: The two new clusters (subtrees with roots A11 and A12) correspond to the large homogeneous subtrees in the old cluster.

2. Examine the individual clusters for homogeneity by choosing the two executions in the cluster with maximally dissimilar profiles. If the selected executions have the same or related causes, it is likely that all of the other failures in the cluster do as well. If the selected executions do not have the same or related causes, the cluster is not homogeneous and should be split.
3. If neither the cluster nor its sibling is split by step 2, and the failures were examined have the same cause then we merge them.

Clusters that have been generated from merging or splitting should be analyzed in the same way, which allow for recursive splitting or merging.

4.4.1.3. Refinement using classification trees

The second technique proposed by Francis et al., relies on building a classification tree to recognize failed executions. A classification tree is a type of pattern classifier that takes the form of binary decision tree. Each internal node in the tree is labeled with a relational expression that compares a numeric feature of the object being classified to a constant splitting value. On the other hand, each leaf of the tree is labeled with a predicted value, which is the class of interest the leaf represents.

Given the classification tree, we have to traverse the tree from the root to a leaf in order to classify an object. At each step of the traversal prior to reach a leaf, we evaluate the expression at the current node. When the object reaches a leaf, the predicted value of that leaf is taken as the predicted class for that object.

In case of software failure classification problem, we consider two classes, that is *success* and *failure*. The **C**lassification **A**nd **R**egression **T**ree (CART) algorithms was used in order to build the classification tree corresponding of software failures. Assume a training set of execution profiles

$$L = \{(x_1, j_1), \ldots, (x_N, j_N)\}$$

where each x_i represents an execution profile and j_i is the result (success/failure) associated with it. The steps of building the classification tree based on L are as follows:

– The deviance of a node $t \subseteq L$ is defined as

$$d(t) = \frac{1}{N_t} \sum \left(j_i - \overline{j}(t)\right)^2$$

where N_t is the size of t and $\overline{j}(t)$ is the average value of j in t.
– Each node t is split into two children t_R and t_L. The split is chosen that maximizes the reduction in deviance. That is, from the set of possible splits S, the optimal split is found by:

$$s^* = argmin_{s \in S} \left(d(t) - \frac{N_{t_L}}{N_t} d(t_R) - \frac{N_{t_L}}{N_t} d(t_L) \right)$$

– A node is declared a leaf node if $d(t) \leqslant \beta$, for some threshold β.
– The predicted value for a leaf is the average value of j among the executions in that leaf.

4.4.2. Frequent pattern mining and association rules

Two approaches for mining call-usage patterns from source code are presented in [31]. The first approach is based on the idea of itemset mining. It identifies frequent subsets of items that satisfy at least a user-defined minimum support. The results of applying this approach to source code are unordered patterns related to the function calls. On the other hand, sequential pattern mining approach produces a set of ordered patterns with a specified minimum support. In general terms these approaches can assist with mining patterns of call-usage and thus identifying potential bugs in a software system.

4.5. Maintenance

A problem that we have to tackle in software engineering is the corrective maintenance of software. It would be desirable to identify software defects before they cause failures. It is likely that many of the failures fall into small groups, each consisting of failures caused by the same software defect. Recent research has focused on data mining techniques which can simplify the problem of classifying failures according to their causes. Specifically, these approaches requires that three types of information about executions are recorded and analyzed: i)*execution profiles* reflecting the causes of the failures, ii) *auditing information* that can be used to confirm reported failures and iii) *diagnostic information* that can be used in determining their causes. Below we present the various data mining approaches used to facilitate software maintenance (see also Table 5).

4.5.1. Clustering

In [32] a framework is presented for knowledge acquisition from source code in order to comprehend an object-oriented system and evaluate its maintainability. Specifically, clustering techniques are used to assist engineers with understanding the structure of source code and assessing its maintainability. The proposed approach is applied to a set of elements collected from source code, including:

– Entities that belong either to behavioral (classes, member methods) or structural domain (member data).
– Attributes that describe the entities (such class name, superclass, method name etc).
– Metrics used as additional attributes that facilitate the software maintainer to comprehend more thoroughly the system under maintenance.

The above elements specifies the data input model of the framework. Another part of the framework is an extraction process which aim to extract elements and metrics from source code. Then the extracted

Table 5
Mining approaches used in software maintenance

Maintenance		
MINING APPROACH	INPUT DATA	DATA ANALYSIS RESULTS
Searching/ matching [52,34]	SCM (bug reports, bug fixes)	identification of bug-introducing changes
Clustering [32]	source code	Extract significant patterns from the system source code groups of similar classes, methods, data
Clustering [54]	SCM, source code	patterns in the history and the development process
Clustering [38]	source code	System modules
Clustering [4]	source code	structure clone
Frequent pattern mining and Association Rules		and their categorization
Classification [25]	commits (SCRs)	classes of commits
Classification [42]	source code	FP & NFP modules Classifier
Classification [16]	CVS and Bugzilla	stability of prediction models
Frequent pattern mining [59] and Association Rules	source code	Prediction of failures, correlations between entities identification of additions, modifications,deletions of syntactic entities
Frequent pattern mining and association rules [41]	software libraries	Reused patterns
Frequent pattern mining and Association rules [50]	source code of legacy system	design alternatives
Frequent pattern mining and Association rules [51]	instantiation code of software	usage changes
Regression, Classification[49]	SCM issue-tracking database	functionality analysis
Classification based on Statistics	source code	syntactic and semantic changes
Differencing [48]		semantic changes
Mining based on Statistics, CVS annotations [20]	version history of source code, classes	syntax & semantic – hidden dependencies
Mining based on Statistics CVS annotations [22]	bug & comments modification request	syntax & semantic – file coupling
Mining via Heuristic [24]	CVS annotation heuristics	candidate entities for change

information is stored in a relational database so that the data mining techniques can be applied. In the specific approach, clustering techniques are used to analyze the input data and provide a rough grasp of the software system to the maintenance engineer. Clustering produces overviews of systems by creating mutually exclusive groups of classes, member data, methods based on their similarities. Moreover, it

can assist with discovering programming patterns and outlier cases (unusual cases) which may require attention.

Text clustering has also been used in software engineering, in order to discover patterns in the history and the development process of large software projects. In [54] they have used CVSgrab to analyze the ArgoUML and PostgreSQL repositories. By clustering the related resources, they generated the evolution of the projects based on the clustered file types. Useful conclusions can be drawn by careful manual analysis of the generated visualized project development histories. For example, they discovered that in both projects there was only one author for each major initial contribution. Furthermore, they came to the conclusion that PostgreSQL did not start from scratch, but was built atop of some previous project. An interesting evolution of this work could be a more automated way of drawing conclusions from the development history, like for example extracting clusters labels, map them to taxonomy of development processes and automatically extract the development phases with comments emerging from taxonomy concepts.

Mancroridis et al. [38] proposes the use of clustering techniques in order to assist with software development and maintenance. Introducing the concepts of inter-connectivity and intra-connectivity, they develop a clustering algorithm that aims to partition the components of a system into compact and well-separated clusters. Specifically, they aim to apply clustering to the module dependency graph in order to identify significant connection among the system modules. The goal is to partition the software system so that it maximizes the connections between the components of the same cluster and minimizes the connections between the components of distinct clusters).

Basit et al. [4] introduced the concept of structure clone and proposed the use of mining techniques in order to detect them in software. The procedure of detecting structural clones can assist with understanding the design of the system for better maintenance and with re-engineering for reuse. According to their approach, they extract simple clones form the source code (similar code fragments). Then they use techniques of finding frequent closed item sets to detect recurring groups of simple clones in different files or methods. Also clustering techniques are applied to identify significant groups of similar clones. Also Basit et al. implement their structural clone detection technique in a tool called *Clone Miner*.

4.5.2. Classification

In [49] they use the data coming from more than 100.000 open source software projects lying in the SourceForge portal, in order to build a predictive model for software maintenance using data and text mining techniques. Using SAS Enterprise Miner and SAS Text Miner, they focused on collecting values for variables concerning maintenance costs and effort from OSS projects, like Mean Time to Recover (MTTR) an error. The task also entailed the removal of projects that were under development, thus considering exclusively operational projects, as well as the removal of projects that did not have a bug reports database since the absence of such prohibited the measurement of variables like MTTR. Furthermore, they clustered the remaining projects based on their descriptions, in order to discover the most important categories of OSS projects lying in the SourceForge database. Finally, they used the SAS Enterprise Miner to build classifiers on the MTTR class variable, after having transformed the later into a binary one (High or Low) using its values' distribution. The reported results highlight interesting correlations between features like number of downloads, use of mail messages and project age and the class variable. For example, projects with increased age have higher MTTR than younger projects.

An approach that exploits the idea of spam filtering techniques to identify fault-prone software modules is presented in [42]. The proposed framework is based on the fact that faulty software modules have similar pattern of words or sentences. Mizuno et al. proposed the implementation of a tool that extracts

fault-probe (FP) modules and non fault-prone (NPF) modules from source code repositories. Then these set of modules are used to learn a classifier that is used to classify new modules as FP or NFP.

Also an approach for classifying large commits so that understand the rationale behind them is proposed in [25]. Though large commits are usually considered as outliers when we study source control repositories (SCRs), they may contain useful information about the projects and their evolution. Hindle etal. decided to exploit classification techniques in order to classify commits and thus identify different types of software changes. This study shows that in many cases the large commits refer to modification of the system architecture while small commits are more often corrective.

Ekanayake et al. [16] propose a method to evaluate the stability of a prediction model. They explore four open source projects and extract features from their CVS and Bugzilla repositories. Then they build defect prediction models using Weka's decision tree learner and evaluate the prediction quality over time. This study conclude that there are significant changes over time and thus it should be used cautiously.

4.5.3. Frequent pattern mining and association rules

The work proposed by Zimmerman et al. [59] exploits the association rules extraction technique to identify co-occurring changes in a software system. For instance, we want to discover relation between the modification of software entities. Then we aim to answer the question when a particular source-code entity (e.g. a function A) is modified, what other entities are also modified (e.g. the functions with names B and C)? Specifically, a tool is proposed that parses the source code and maps the line numbers to the syntactic or physical-level entities. These entities are represented as a triple (*filename, type, id*). The subsequent entity changes in the repository are grouped as a transaction. An association rule mining techniques is then applied to determine rules of the form $B, C \rightarrow A$.

Sartipi et al. [50] proposes the use of clustering and association rules techniques in order to recover the architectural design of legacy software systems according to user defined plans. The source code of a legacy system is analyzed and a set of frequent itemsets is extracted from it. Using clustering and pattern matching techniques, the proposed algorithm defined the components of the legacy system. Given a user query, the best matching component of the system is selected. Also a score can be associated with each possible answer (match) to the user query and thus a ranking of design alternatives can be presented to the user for further evaluation.

An approach for identifying library reuse patterns is presented in [41]. The proposed approach exploits association rules techniques to identify relations among classes in a library. The authors extend the concept of traditional association rules to generalized rules so that the inheritance relationships are taken into account. Thus an automated technique is developed for discovering reused patterns in libraries and identifying characteristic usages of a library.

An approach for analyzing instantiation code to find usage changes in evolving frameworks is proposed in [51]. The mining process takes as input two versions of instantiation code and exploiting frequent pattern mining techniques aims to find patterns describing a changed usage of the framework. At the first step, it extracts information about how the instantiation code uses the framework (which methods are called, which framework classes are sub-classed). Then transactions are built by combining usage information from the two versions of each instantiation class. Finally, an association rule mining algorithm is applied to those transactions to extract all possible change rules.

4.5.4. Change and deviation detection

The identification and fixing of bugs is one of the most common and costly tasks of software development. The software projects manage the flow of the bugs using software configuration management

(SCM) systems to control the bug changes, bug tracking software (such as Bugzilla) to capture bug reports and then they record the SCM system that fixes a specific bug in the tracking system. Generally, a bug is introduced into the software when a programmer makes a change to the system, that is, to add a new functionality, to reconstruct the code or to repair an existing bug. When the bug is identified, it is recorded in a bug tracking system. Subsequently, a developer could repair the bug by modifying the project's source code and commit the change to the SCM system. This modification is widely called bug-fix change. The bug tracking and SCM systems are widely used, the most readily available data concerning bugs are the bug-fix changes. There are some approaches that deals with mining a SCM system to find those changes that have repaired a bug. There are two categories of approaches that search for changes in the log messages: i) approaches [43] that searches for keywords such as 'Fixed' and 'Bug', ii) approaches that look for references to the bug reports (e.g $\#9843$). Bug-fixing information is useful for determining the location of a bug. This permits useful analysis, such as determining per-file bug counts, predicting bugs, finding risky parts of software or visually revealing the relationship between bugs and software evolution. One of the main problems with the bug-fix data is that it does not give an indication when a bug was injected into the code and who injected it. Also bug-fix data provide imprecise data on where a bug occurred. In order to deeply understand the phenomena related to the introduction of bugs into the code, we need access to the actual moment and point the bug was introduced. The *algorithm* proposed in [52] (further referred to as SZZ algorithm) is the first effort to identify bug-introducing changes from bug-fix changes. The main steps of *SZZ* can be summarized as follows: i) it finds bug-fix changes by locating bug identifiers or relevant keywords in change log text or following a recorded linkage between a bug tracking system and a specific SCM commit. ii) it runs a *diff* tool to determine what changed in the bug-fixes. The *diff tool* returns a list of regions further, referred to as *hunks*, which are different in the two files. In each hunk the deleted or modified source code is considered as a location of a bug. iii) it tracks down the origins of the deleted or modified source code in hunks. For this purpose it uses the built-in annotate feature of a SCM system, which computes the most recent revision in which a line was changed and the developer who made the change. The discovered origins are identified as bug-introducing changes. However there some limitations of the SZZ algorithm which can be summarized as follows:

- SCM annotation does not provide enough information to identify bug-introducing changes. Also we have to trace the evolution of individual lines across revisions in order that the functions/methods containment can be determined.
- All modifications are not fixes: There might be changes that are not bug-fixes. For instance, changes to comments, black line and formatting are not bug-fixes, even though based on SCM are flagged as such.

An approach proposed in [34] aims to tackle the above discussed problems of the SZZ algorithm. The proposed approach exploits annotation graphs which contain information on the cross-revision mappings of the individual lines. This allow us to associate a bug with its containing function or method. The proposed bug-introducing identification algorithm can be employed as an initial clean-up step to obtain high quality data sets for further analysis on causes and patterns of bug formation. The accuracy of the automatic approach is determined using a manual approach. This implies that two human manually verified all hunks in a series of bug-fix changes to ensure the corresponding hunks are real bug-fixes. The main steps of the approach introduced in [34], which aims to remove false positive and false negatives in identifying bug-introducing changes are the followings:

- Use annotation graphs that provide more detailed annotation information

– Ignore comments, black line, format changes, outlier bug-fix revisions in which too many files were changed
– Manually verify all hunks in the bug-fix changes

4.5.5. Mining approaches based on statistics

Many of source code version repositories repositories are examined and managed by tools such as CVS (Concurrent Versions System) and (increasingly) its successor *Subversion*. These tools store difference information access across document(s) versions, identifies and express changes in terms of physical attributes, i.e., file and line numbers. However, CVS does not identify, maintain or provide any change-control information such as grouping several changes in multiple files as a single logical change. Moreover, it does not provide high-level semantics of the nature of corrective maintenance(e.g. bug-fixes). Recently, the interest of researchers has been focused on techniques that aim to identify relationships and trends at a syntactic-level of granularity and further associate high-level semantics from the information available in repositories. Thus a wide array of approaches that perform mining of software repositories (MSR) have been emerged. They are based on statistical methods and differencing techniques, and aim to extract relevant information from the repositories, analyze it and derive conclusions within the context of a particular interest.

4.5.5.1. Mining via CVS annotations

One approach is to utilize CVS annotation information. Gall et al. [20] propose an approach for detecting common semantic (logical and hidden) dependencies between classes on account of addition or modification of particular class. This approach is based on the version history of the source code where a sequence of release numbers is maintained for each class in which its changes are recorded. Classes that have been changed in the same release are compared in order to identify common change patterns based on *author name* and *time stamp* from the CVS annotations. Classes that are changed with the same time stamp are inferred to have dependencies.

Specifically, this approach can assist with answering questions such as which classes change together, how many times was a particular class changed, how many class changes occurred in a subsystem (files in a particular directory). An approach that studies the file-level changes in software is presented in [22]. The CVS annotations are utilized to group subsequent changes into what termed modification request (MR). The proposed approach focus on studying bug-MRs and comment-MRs to address issues regarding the new functionality that may be added or the bugs that may be fixed by MRs, the different stages of evolution to which MRs correspond or identify the relation between the developer and the modification of files.

4.5.5.2. Mining via heuristics

CVS annotation analysis can be extended by applying heuristics that include information from source code or source code models. Hassan et al. [24] proposed a variety of heuristics (developer-based, history-based, code-layout-based (file-based)) which are then used to predict the entities that are candidates for a change on account of a given entity being changed. CVS annotations are lexically analyzed to derive the set of changed entities from the source-code repositories. Also the research in [24,59] use source-code version history to identify and predict software changes. The questions that they answered are quite interesting with respect to testing and impact analysis.

Table 6
Mining approaches used in Software Reuse

Software Reuse		
MINING APPROACH	INPUT DATA	DATA ANALYSIS RESULTS
Frequent pattern mining and Association Rules [40]	set of variables describing projects [40]	rules among features of projects
Frequent pattern mining [3]	source code	software design patterns
Classification [39]	project variables describing projects [40]	value ranges used in successful reuse projects
Classification [30]	project variables describing projects [40]	features affect software reuse

4.5.5.3. Mining via differencing

Source-code repositories contain differences between versions of source code.Thus it would be interesting to mine source code repositories, identify and analyze the actual source-code differences.

An approach that aims to detect syntactic and semantic changes from a version history of C code is presented by Raghavan [48]. According to this approach, each version is converted to an abstract semantic graph (ASG) representation. This graph is a data structure which is used in representing or deriving the semantics of an expression in a programming language. A top-down or bottom-up heuristics-based differencing algorithm is applied to each pair of in-memory ASGs. The differencing algorithm produces an edit script describing the nodes that are added, deleted, modified or moved in order to derive one ASG from another. The edit scripts produced for each pair of ASGs are analyzed to answer questions from entity level changes such as how many functions and functions calls are inserted, added or modified to specific changes such as how many *if* statement conditions are changed. Also in [12] a syntactic-differencing approach, which is called *meta-differencing*, is introduced. It allows us to ask syntax-specific questions about differences. According to this approach the abstract syntax tree (AST) information is directly encoded into the source code via XML format. Then we compute the added, deleted or modified syntactic elements based on the encoded AST. The types and prevalence of syntactic changes can be easily computed. Specifically, the approach supports the following questions: i) Are new methods added to an existing class?, ii) Are there changes to pre-processor directives?, iii) Was the condition in an if-statement modified?

4.6. Software reuse

Systematic software reuse has been recognized as one of the most important aspects towards the increase of software productivity, and quality [40,45]. Though software reuse can take many forms (e.g., ad-hoc, systematic), and basic technical issues such as development of software repositories, and search engines for software components in various programming languages are on the frontier of research in the area of software reuse, recently there have been attempts to incorporate data mining techniques in an effort to identify the most important factors affecting the success of software reuse. The motivation behind those approaches stems partially from the fact that previous surveys showed possibility of projects' failure due to the lack of reuse processes introduction, as well as modification of non-reuse processes [45].

In this direction, Morisio et al. [45] attempted to identify the key factors that are crucial to the success of the software reuse process in conducted projects. More specifically, they collected through an interview process data from twenty-four European projects from nineteen companies in the period from 1994 to 1997. In their analysis, they defined 27 variables that are used to formulate the description of each project, which are nicely summarized in [40]. Among the used variables, there are ten state variables

representing attributes over which a company has no control, six high-level control variables representing key high-level management decisions about a reuse program, ten low-level control variables representing specific approaches to the implementation of reuse, and a variable indicating whether the project was successful or not. Currently, this data set, though with few examples, constitutes the largest empirical data set on software reuse at present, and in this data set several data mining algorithms have been applied to identify patterns regarding the factors affecting the success of software reuse. Table summarizes the main feature of mining techniques that have been used in software reuse.

4.6.1. Classification

In the same study ([40]), the authors also attempted the use of decision trees, and more specifically the $J.48$ implementation of Weka, which is essentially the implementation of the $C4.5$ decision tree algorithm [47], in order to analyze the same data. The application of the $C4.5$ decision tree algorithm in this was made in a manner so that the authors were able to identify the most important features from the 27, by conducting attribute removal experiments. More specifically, they studied what would be the root node of the tree in each case, if at each time the most important attribute is removed (i.e., the root node), and the tree is rebuilt without considering that attribute. This methodology allowed them to identify weak attributes (i.e., attributes that appear in any non-root node after several removals of root node attributes), as well as barely supportive attributes (i.e., attributes that, once root nodes, if removed and the tree is rebuilt disregarding them, the classification accuracy remains the same).

In addition to the aforementioned analysis, they also applied a learning algorithm called *treatment learning*, and more specifically they applied the *TAR2* algorithm [39]. The basic idea behind the treatment learner is that it selects a subset D' of the training set D, which contains more preferred classes and less undesired ones. The criterion according to which the subset is selected is based on the used treatment, denoted as R_x, and, thus, D' should not contradict the treatment. However, the *TAR2* treatment learner requires from the user to assign a numeric score to each class that represents how much a user likes that class. In this case, the authors weighted more the successful reuse project than an unsuccessful project, and after the conducted analysis through the application of *TAR2*, they were able to discover the features and the respective value ranges that were mostly used in successful reuse projects. The interesting part from the application of this data mining algorithm is the fact that the algorithm discovered features and value ranges that the empirical study in [45] had failed to uncover, showing how important the application of data mining can be in this case.

In another direction, but still using the same data set for software reuse, Jiang et al. [30] applied an ensemble learning approach (i.e, an approach that combines the decisions of different classifiers and attempts to get the best of all worlds) based on the notion of random forests [9]. The used ensemble algorithm, *RF2TREE* (Random Forest to Tree), introduced in the same paper, has two conditions under which it can be guaranteed to work well: (a) the original training data set is very small, like in the case of the data set produced in [45], and (b) the random forest is more accurate than single decision tree if both of them were directly trained from the original training data set. The algorithm first builds a random forest from the original data set, and then the random forest is used to generate many virtual examples that are used to train a single decision tree. Based on their conducted experiments, they discovered that the most important features that affect the success of software reuse are: Human Factor,Reuse Process Introduced, Type of Software Production, Application domain, Top Management Commitment, and Nonreuse Processes Modified, which vary from the empirical analysis in [45], and the data mining analysis in [40]. The differences, as well as the similarities of the three research works with regards to the most important factors affecting software reuse are summarized nicely in Table 8 in the work of Jiang et al. [30].

4.6.2. Frequent pattern mining and association rules

Menzies and Di Stefano [40] worked on the aforementioned data set in order to examine further what conclusions may be drawn regarding the affecting factors in software reuse, and also to compare the patterns derived from applying automated data mining methodologies with the empirical conclusions formulated in [45]. Among the methodologies they used, association rule mining was employed, in order to extract meaningful associations between the 27 features. The association rule extraction was conducted with the Apriori algorithm [1] implementation offered by the Weka data mining platform[3]. The top 10 association rules derived, setting minimum confidence at 90%, are presented in [40]. Among the association rules derived, there are very interesting associations learned, like for example the fact that when the produced software was embedded in a product (e.g., in contrast to being embedded in a process, or itself being a stand-alone product), the use of reward policy for software reuse was not enabled (i.e., $SoftwareandProduct = product => RewardsPolicy = no$).

Another important aspect of data mining in software engineering is the process of mining design patterns from software, in order to allow for the reuse of software system design expertise. More specifically, the process of mining focuses on extracting patterns by analyzing the code or the design of the software system in order to trace back the design decisions made, which are usually buried inside the source code. Typically, during the software system design, the system components are not tagged with the respective design patterns applied, and, thus, the design decisions are no longer connected with the existent system, often leading to lack of understanding of the software's details. In this direction, a number of techniques and tools have been proposed in the past, which attempt to mine the design patterns from a software system.

In [14] the authors present a thorough overview of these approaches in a comparative study. Depending on the description of each design pattern, i.e., the perspective from which it is described, the approaches of design pattern mining can be widely classified into the ones analyzing the structural aspect only, the behavioral aspect only, or both. There are also some approaches that attempt to analyze a combination of the above aspects including the semantic aspect as well, which refers to the semantic meaning of some entities in the system. From the perspective of our analysis, we focus only on the types of data mining methods to extract those patterns. In this direction, the primal technique used is the utilization of existing tools that transform the source code into some intermediate representation, e.g. Abstract Syntax Trees (AST) or Abstract Semantic Graphs (ASG), and then simple search strategies are applied to the transformed graphs, in order for patterns to be identified. This procedure of course implies that the patterns are somehow already defined, in order for the application of the search strategy to be able to find matches. The problem of design pattern mining from source code is then reduced to identify graph components of the code that match already predefined patterns, which can be expressed for example in a XML-like format [3].

The matching itself can be usually conducted through the use of sub-graph isomorphic comparison between the code and the pattern graphs, and thus it is a form of supervised learning (patterns are already known), using graph comparison or similarity metrics between the examined graphs and the graph patterns.

5. Summary and open issues in mining software repositories

The recent explosive growth of our ability to collect data during the software development process has created a need for new, scalable and efficient, tools for data analysis. Also there is strong requirement for

[3] http://www.cs.waikato.ac.nz/ml/weka/.

mining software repositories and extracting hidden information. This extracted knowledge is expected to assists the software engineers with better understanding the development processes and predict the future of software products. The main focus of the discipline of data mining in software repositories is to address this need. In this paper we review the various data mining methods and techniques used in software engineering. Specifically our objective is to present an overview of the different data sources in software engineering that are interesting to be mined. Also we discuss how the data mining approached can be used in software engineering and what software engineering tasks can be helped by data mining. The main characteristics of data mining approaches used in software engineering are summarized in Tables 1–6.

One of the main issues in software engineering is the evaluation of software project and the definition of metrics and model that give us an indication of the future of a project. Though a number of mining approaches have been used to assist with software engineering tasks, an open issue is if and how data mining techniques can be exploited to define novel quality metrics in software engineering.

Below we discuss *challenging issues in mining software engineering repositories* that are interesting and deserve further work.

- *Supervised learning approaches*, like text classification, based on predictive modeling techniques, for the purposes of predicting future bugs and/or possibly affected parts of code. A measure of future influence of bugs in the source code, associated with a weight and a prediction ranking can show a lot for the software quality.
- *Text clustering of the bug reports*, and cluster's labeling can be used to automatically create a taxonomy of bugs in the software. Metrics in that taxonomy can be defined to show the influence of generated bugs belonging to a specific category, to other categories of bugs. This can also be translated as a metric of bug influence across the software project.
- *Online mining*. The data mining techniques that have recently been developed in software engineering 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 online to provide immediate feedback. Thus a challenging issue is the adaptation or development of stream mining algorithms for software engineering data so that the above mentioned requirement are satisfied.
- *Quality project classification*. A classifier will be built to categorize projects as successful or non-successful based on the data collected about projects. These data provide information about features of projects related to the popularity, ranking of projects. The quality of the classification (i.e. accuracy of classifier) depends on the training set. Then the requirement is to select the appropriate set of data features based on which we will build an accurate classifier of projects.
- *Association rules extraction from OSS project data*. There is useful information provided for Open Source Software projects regarding the number of downloads, the number of developers, the popularity, the vitality of the software, etc. These are considered to be metadata of the OSS project. Analyzing thus information we can extract useful knowledge about OSS projects and new quality metrics could be defined. An interesting direction would be to find correlations between the metadata provided for OSS. We assume that each project can be represented by a vector (*projectid, selected_metadata*), where metadata refers to the OSS development metrics i.e., popularity, activity, number of downloads. The subsequent project's evaluations are stored in the repository as transactions. Then an association rule extraction algorithm can be used to discover correlations or co-occurrences of events in a given OSS environment.

– *Graph mining on the mailing lists of OSS projects*. Based on the information provided by the mailing list of the project we could build author and message graphs. Then applying mining techniques to these graphs we can extract useful information regarding message exchange or the users that contributes to projects. An interesting research approach will be to exploit graph processing techniques, like PageRank or Spreading Activation so that we rank nodes in each graph. The extract results can assist with rank users and measure relatedness between important users and important messages.
– *Pattern mining form source code*. Another interesting perspective in the category of design pattern mining approaches in software engineering, could be to apply a graph clustering approach, or in general an unsupervised method, and examine what design patterns would be produced from the analyzed code. This would imply the definition of a graph clustering model, where in this case the graphs could be Abstract Syntax Trees (ASTs) or Abstract Semantic Graphs (ASGs). The model should allow for the computation of the similarity between graphs, as well as for the computation of cluster representatives, i.e., the centroid graph of the graphs included in each cluster. The process would then be able to extract patterns, and which in turn could give an insight, after post-processing, about the design patterns used, as well as the design decisions made.
– *Mining bug reports*. The bug report database contains useful information regarding the quality of the software. Analyzing the data collected from the bug fixing procedure, we could extract information about i) average impact on code change (i.e. % of files or % of lines changed), ii) estimate mean time before bug fixing developers involved in the bug fixing procedure. iii) temporal bug distribution in relation to project release dates.

Acknowledgments

This work is supported by the European Community Framework Programme 6, Information Society Technologies key action, contract number IST-5-033331 (SQO-OSS: Software Quality Observatory for Open Source Software).

References

[1] R. Agrawal, T. Imielinski and A.N. Swami, Mining association rules between sets of items in large databases. In *Proc of the ACM SIGMOD*, 1993, pages 207–216.
[2] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley, 1999.
[3] Z. Balanyi and R. Ferenc, Mining design patterns from c++ source code. In *International Conference on Software Maintenance* (*ICSM*), 2003, pages 305–314.
[4] H.A. Basit and S. Jarzabek, Data mining approach for detecting higher-level clones in software, *IEEE Transactions on Software Engineering*, 2009.
[5] M. Berry and G. Linoff, *Data Mining Techniques For marketing, Sales and Customer Support*, John Willey and Sons Inc., 1996.
[6] C. Bird, A. Gourley, P. Devanbu, M. Gertz and A. Swaminathan, Mining email social networks. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2006.
[7] C. Bird, P. Rigbyy, E. Barr, D. Hamilton, D. Germany and P. Devanbu, The promises and perils of mining git. In *Proceedings of the IEEE Working Conference on Mining Software Repositories*, 2009.
[8] J. Bowring, J. Rehg and M.J. Harrold, Acive learning for automatic classification of software behavior, *International Symposium on Software Testing and Analysis* (*ISSTA*), 2004.
[9] L. Breiman, Random forests, *Machine Learning* **45**(1) (2001), 5–32.
[10] S. Chakrabarti, *Mining theWeb: Analysis of Hypertext and Semi Structured Data*, Morgan Kaufmann, 2002.

[11] R Chang, A. Podgurski and J. Yang, Discovering neglected conditions in software by mining dependence graphs, *IEEE Transactions on Software Engineering*, 2008.
[12] M.L. Collard and J.K. Hollingsworth, *Meta-differencing: An Infrastructure for Source Code Difference Anlysis*, Kent State University, Kent, Ohio USA, Ph.D. Dissertation Thesis, 2004.
[13] W. Dickinson, D. Leon and A. Podgurski, Finding failures by cluster analysis of execution profiles, *International Conference on Software Engineering* (*ICSE*), 2001.
[14] J. Dong, Y. Zhao and T. Peng, A review of design pattern mining techniques, *International Journal of Software Engineering and Knowledge Engineering* **19**(6) (2009), 823–855.
[15] R.O. Duda and P.E. Hart, *Pattern Classification and Scene Analysis*, John Wiley and Sons, 1973.
[16] J. Ekanayake, J. Tappolet, H. Gall and A. Bernstein, Tracking concept drift of software project using defect prediction quality. In *Proceedings of InternationalWorkshop on Mining Software Repositories* (*MSR*), 2009.
[17] J. Estublier, D. Leblang, A. Van Der Hoek, R. Conradi, G. Clemm, W. Tichy and D. Wilborg-Weber, Impact of software engineering research on the practice of software configuration management, *ACM Transactions on Software Engineering and Methodology* **14**(4) (2005), 1–48.
[18] U. M. Fayyad, G. Piatesky-Shapiro, P. Smuth and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.
[19] P. Francis, D. Leon, M. Minch and A. Podguraki, Tree-based method for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, 2004.
[20] H. Gall, K. Hajek and M. Jazayeri, Detection of logical coupling based on product release history. In *Proceedings of the 14th IEEE International Conference in Software Maintainance*, 1998.
[21] D. German and A. Mockus, Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering* (*ICSE-03*), 2003.
[22] D.M. German, An empirical study of fine-grained software modifications. In *Proceedings of 20th IEEE International Conference on Software Maintenance* (*ICSM'04*), 2004.
[23] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publishers, 2nd edition, 2006.
[24] A. Hassan and R.C. Holt, Predicting change propagation in software systems. In *Proceedings of 26th International Conference on Software Maintenance* (*ICSM'04*), 2004.
[25] A. Hindle, D. German and R. Holt, What do large commits tell us? a taxonomical study of large commits. In *Proceedings of the IEEE Working Conference on Mining Software Repositories*, 2008.
[26] J. Huffman Hayes, A. Dekhtyar and J. Osborne, Improving requirements tracing via information retrieval. In *Proceedings of the International Conference on Requirements Engineering*, 2003.
[27] J. Huffman Hayes, A. Dekhtyar and S. Sundaram, Text mining for software engineering: How analyst feedback impacts final results. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2005.
[28] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data*, Prentice-Hall, 1988.
[29] C. Jensen and W. Scacchi, Data mining for software process discovery in open source software development communities. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2004.
[30] Y. Jiang, M. Li and Z.H. Zhou, Mining extremely small data sets with application to software reuse, *Softw, Pract Exper* **39**(4) **2009**, 423–440.
[31] H. Kagdi, M. Collard and J. Maletic, Comparing approaches to mining source code for callusage patterns. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2007.
[32] Y. Kannelopoulos, Y. Dimopoulos, C. Tjortjis and C. Makris, Mining source code elements for comprehensing object-oriented systems and evaluating their maintainability, *SIGKDD Explorations* **8**(1), 2006.
[33] L. Kauffman and P.J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*, John Wiley and Sons, 1990.
[34] S. Kim, T. Zimmermann, K. Pan and J. W Whitechead, Automatic identification of bugintroducing changes. In *International Conference on Automated Software Engineering*, 2006.
[35] M. Last, M. Friedman and A. Kandel, *The Data Dimining Approach to Automated Software Testing*, In Proceeding of the SIGKDD Conference, 2005.
[36] C. Liu, X Yan, and J. Han. Mining control ow abnormality for logical errors. In *Proceedings of SIAM Data Mining Conference* (*SDM*), 2006.
[37] C. Liu, X. Yan, H. Yu, J. Han and P. Yu, Mining behavior graphs for 'backtrace' of non-crasinh bugs. In *SIAM Data Mining Conference* (*SDM*), 2005.
[38] S. Mancoridis, S. Mitchell, C. Rorres, Y. Chen and E.R. Gansner, Using automatic clustering to produce high-level system organisations of source code. In *Proc. 6th Int'l Workshop Program Understanding* (*IWPC 98*), 1998.
[39] T. Menzies, E. Chiang, M. Feather, Y. Hu and J.D. Kiper, Condensing uncertainty via incremental treatment learning. In book chapter in: *Software Engineering with Computational Intelligence*, T.M. Khoshgoftaar, ed., page Volume 731. Kluwer Academic Publishers, 2003.

[40] T. Menzies and J.S. Di Stefano, More success and failure factors in software reuse, *IEEE Trans Software Eng* **29**(5) (2003), 474–477.
[41] A. Michail, Data mining library reuse patterns using generalized association rules. In *Proc Int'l Conf Software Eng*, 2000.
[42] O. Mizuno, S. Ikami, S. Nakaichi and T. Kikuno, Spam filter based approach for finding fault-prone software modules. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2007.
[43] A. Mockus and L.G. Votta, Identifying reasons for software changes using historic databases. In *Proceedings of International Conference on Software Maintenance*, 2000.
[44] S. Morisaki, A. Monden and T. Matsumura, Defect data analysis based on extended association rule mining. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2007.
[45] M. Morisio, M. Ezran and C. Tully, Success and failure factors in software reuse, *IEEE Trans Software Eng* **28**(4) (2002), 340–357.
[46] A. Podgurski, W. Masri, Y. McCleese, M. Minch, J. Sun, B. Wang and W. Masri, Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.
[47] J. Ross Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.
[48] S. Raghavan, R. Rohana and A. Podgurski, Dex: A semantic-graph differencong tool for studying changes in large code bases. In *Proceedings of 20th IEEE International Conference on Software Maintenance* (*ICSM'04*), 2004.
[49] U. Raza and M.J. Tretter, Predicting software outcomes using data mining and text mining. In SAS Global Forum, 2007.
[50] K. Sartipi, K. Kontogiannis and F. Mavaddat, Architectural design recovery using data mining techniques. In *Proc. 2nd European Working Conf. Software Maintenance Reengineering* (*CSMR 2000*), 2000.
[51] T. Schafer, J. Jonas and M. Mezini, Mining framework usage changes from instantiation code. In *International Conference on Software Engineering* (*ICSE*), 2008.
[52] S. Sliwerski, T. Zimmermann and A. Zeller, When do changes induce fixes? In *Proceedings of International Workshop on Mining Software Repositories* (*MSR*), 2005.
[53] Ian Somerville, *Software Engineering*, Addison-Wesley, Chapter 30, 4th edition, 1992.
[54] L. Voinea and A. Telea, Mining software repositories with cvsgrab. In *Proceedings of International Workshop on Mining Software Repositories* (*MSR-06*), 2006.
[55] S.M.Weiss and C. Kulikowski, *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning and Expert Systems*, Morgan Kauffman, 1991.
[56] C.C. Williams and J.K. Hollingsworth, Automating mining of source code reposito- ries to improve bug finding techniques, *IEEE Transactions on Software Engineering* **31**(6) (2005), 466–480.
[57] T. Xie, S. Thummalapenta, D. Lo and C. Liu, *Data mining for Software Engineering*, IEEE Computer, 2009.
[58] T. Xie, S. Thummalapenta, D. Lo and C. Liu, Data mining for software engineering, *Computer* **42** (2009), 55–62.
[59] T. Zimmermann, P. Weibgerber, S. Diehl and A. Zeller, Mining version histories to guide software changes. In *Proceedings of 26th International Conference on Software Engineering* (*ICSE'04*), 2004.