Mining Unstructured Data in Software Repositories: Current and Future Trends

Gabriele Bavota
Free University of Bozen-Bolzano, Bolzano, Italy
gabriele.bavota@unibz.it

***Abstract*—The amount of unstructured data available to software engineering researchers in versioning systems, issue trackers, achieved communications, and many other repositories is continuously growing over time. The mining of such data represents an unprecedented opportunity for researchers to investigate new research questions and to build a new generation of recommender systems supporting development and maintenance activities.**

This paper describes works on the application of Mining Unstructured Data (MUD) in software engineering. The paper briefly reviews the types of unstructured data available to researchers providing pointers to basic mining techniques to exploit them. Then, an overview of the existing applications of MUD in software engineering is provided with a specific focus on textual data present in software repositories and code components.

The paper also discusses perils the "miner" should avoid while mining unstructured data and lists possible future trends for the field.

Keywords*—*mining unstructured data; future trends;

I. Introduction

Unstructured data refer to information that is not organised by following a precise schema or structure. Such data often include text (*e.g.,* email messages, software documentation, code comments) and multimedia (*e.g.,* video tutorials, presentations) contents and are estimated to represent 80% of the overall information created and used in enterprises [1]. The situation is not different in software projects: Archived communications like projects' mailing lists[1] are by nature fully unstructured and even when turning the attention to software repositories one would expect to be highly structured (*e.g.,* versioning systems, issue trackers), the amount of unstructured data is simply overwhelming. For example, let us have a look to the repository of the `Ruby on Rails` project,[2] just one of the almost 22 million `git` repositories hosted on GitHub.[3] This repository contains information about 53K commits performed by 3K contributors. Each commit, besides containing information related to the code changes and the authors of those changes, is accompanied by a commit note "describing" in free (unstructured) textual form the goal of the change (*e.g.,* "*Remove wrong doc line about AC::Parameters*"). Also, the GitHub's `Ruby on Rails` repository contains information about 8K issues (*e.g.,* bug reports, new features to implement, *etc.*) that are characterised by a textual description and heavily discussed by the project's contributors: It is not uncommon to encounter issue's discussions with dozens of comments mixing together code snippets and free textual paragraphs.

The availability of unstructured data for software engineering researchers has grown exponentially in the past few years with the high spread of software repositories. Just to report an example, GitHub hosted 46K projects in 2009, 1M in 2010, 10M in 2013, and 27M in 2015 (*i.e.,* 586 times more projects in just six years) [2]. Besides the widely "mined" software repositories, precious (unstructured) information for the software engineering community is also available in Questions & Answers (Q&A) websites (*e.g.,* discussions on Stack Overflow[4]), video-sharing websites (*e.g.,* programming tutorials hosted on YouTube[5]), slide hosting services (*e.g.,* technical presentations hosted on SlideShare[6]), *etc.* As already observed for GitHub, the growth rate of all these repositories is simply impressive: **Every hour** 102K users look for help on StackExchange[7] and 6K hours of new video contents are posted on YouTube.[8]

This steep growth of data available over the internet has pushed the Mining of Unstructured Data (MUD) to become a popular research area in the software engineering community, where MUD techniques have been applied to automatically generate documentation [3], [4], [5], [6], [7], [8], to summarise bug reports [9], [10], [11] and code changes [12], [13], [14], to assess [15], [16], [17] and improve [18] software quality, to build recommenders supporting project managers in tasks related to mentoring [19] and bug triaging [20], [21], [22] and developers in their daily coding activities [23], [24], [25], [26], [27], [28], [29] as well as in the identification of duplicated bug reports [30]. Also, MUD techniques have been employed to identify malicious mobile apps from their textual description [31] and to automatically categorise reviews left by users in mobile apps' stores [32], [33]. Those are just some examples of the wide applicability that MUD techniques have found in the software engineering research field.

This paper aims at providing an overview of the techniques used to mine unstructured data, of the type of relevant repositories containing unstructured data, and of (part of) the existing literature applying MUD to software repositories/artefacts in order to highlight what are the current trends in the field. Note that the primary focus of the paper is on textual information, representing the vast majority of the unstructured data used by software engineering researchers. The paper also sets out some directions for future work in the MUD field.

[1] See *e.g.,* http://apache.org/foundation/mailinglists.html.
[2] https://github.com/rails/rails
[3] https://github.com/
[4] http://stackoverflow.com
[5] https://www.youtube.com
[6] http://www.slideshare.net
[7] http://stackexchange.com/about
[8] https://www.youtube.com/yt/press/statistics.htm

Paper structure. The rest of the paper is organised as follows: Section II overviews some of the basic techniques to mine unstructured data. Section III describes the most relevant unstructured data for software engineering researchers together with perils to avoid when mining such information. Section IV describes (part of) the works performed in five research areas where the MUD has been widely applied. Finally, Section VI concludes the paper after a discussion of possible future work directions in the MUD field (Section V).

II. Techniques for Mining Unstructured Data

This section describes three techniques that are widely applied for the mining of unstructured data and, in particular, of textual information: Pattern Matching, Information Retrieval (IR), and Natural Language Processing (NLP).

A. Pattern Matching

In the MUD context, pattern matching mainly refers to the checking of the existence of a specific pattern (*i.e.,* a sequence of textual tokens) inside a string. Pattern matching is often implemented by using regular expressions describing the sequence of tokens we are interested in identifying inside a text, and represents a cheap yet powerful solution to deal with a number of problems, including:

1) Identifying in a corpus of documents those likely related to a specific topic. For example, given a set of commit messages (documents' corpus) we could filter out those likely referring to changes applied in order to reduce the energy consumption of a software project (*e.g.,* a mobile app) with the following regular expression:

```
(reduce[s,ed]?)*\s+((energy
(consumption)?)|(battery drain))
```

2) Trace related artefacts containing an explicit link between them, as done by Bacchelli *et al.* [34] to identify mail messages explicitly referring to a specific code component (*e.g.,* a Java class):

```
(.*) (\s|\.|\\|/) <packageTail>
     (\.|\\|/)<EntityName>
((\.(java|class|as|php|h|c))|(\s))+ (.*)
```

The main limitation of pattern matching techniques is represented by their low flexibility (*i.e.,* the match usually has to be exact). For example, while the regular expression defined at point one is able to match the commit message "*Reduced energy consumption*", it is not able to identify "*Reduced battery consumption*" as a relevant commit message. Clearly, it is possible to improve such a regular expression in order to also match this pattern. However, there will likely be other patterns (*e.g.,* "*Improved battery life*") that are still left uncovered by our regular expression. More sophisticated techniques described in the next subsections, while being more complex and computationally expensive, help in overcoming such a limitation.

B. Information Retrieval

van Rijsbergen has defined Information Retrieval (IR) as *the process of actively seeking out information relevant to a topic of interest*. Indeed, IR is generally applied to look for documents in a corpus relevant to a given query. For example, in the context of concept location, IR-techniques have been employed to look for code components (the documents' corpus) relevant to a bug report (the query) [35], [36].

As opposed to the pattern matching techniques, IR approaches do not look for exact string matching and (some of them) are even able to infer word relationships without the use of a thesaurus (*e.g.,* the words *reduced* and *improved* have the same meaning when followed by the bigram *energy consumption*).

The relevance of a document to a query[9] is based on their textual similarity measured by looking at the occurrences of terms (words) within the query and the document. The extraction of the terms from the corpus is generally preceded by a text normalisation phase aimed at removing most non-textual tokens (*e.g.,* operators, special symbols, some numbers) and splitting into separate words source code identifiers composed of two or more words separated by using the under_score, CamelCase notation, or by exploiting more advanced techniques [38]. Common terms (*e.g.,* articles, adverbs) that are not useful to capture semantics of the documents are also discarded using a stop word function, to prune out all the words having a length less than a fixed threshold, and a stop word list, to cut-off all the words contained in a given word list.

Natural Language Processing (NLP) techniques (Section II-C) like stemming and abbreviations expansions can also be applied. Stemming [39] removes suffixes of words and extracts their stems (*e.g., stemming* and *stemmed* are both represented with *stem*) while abbreviations can be expanded into their full words (*e.g., regex* becomes *regular expression*) [40].

The extracted information is generally stored in a $m \times n$ matrix (called *term-by-document* matrix), where m is the number of all terms that occur in all the documents, and n is the number of documents in the corpus. A generic entry $w_{i,j}$ of this matrix denotes a measure of the weight (*i.e.,* relevance) of the i^{th} term in the j^{th} document [41]. A widely adopted term weighting schema is the *term frequency – inverse document frequency (td-idf)* [41]. Term frequency awards terms appearing in a document with a high frequency, while inverse document frequency penalises terms appearing in too many artefacts, *i.e.,* non-discriminating terms. Thus, a term is considered relevant for representing the document content if it occurs many times in the document, and is contained in a small number of documents.

Based on the term-by-document matrix representation, different IR methods can be used to identify relevant documents for a given query. In the following three of them are overviewed.

[9]Note that a query could also be a second document, as usual in IR-based traceability recovery where the goal is to identify pairs of related documents [37].

The first, is the Vector Space Model (VSM), in which each document is represented as a vector of terms that occur within the corpus [41]. In particular, each column of the *term-by-document* matrix can be considered as a document vector in the m-space of the terms. Thus, the similarity between a query and a document is measured by the cosine of the angle between their vectors in the m-space of the terms. Such a similarity measure increases as more terms are shared between the two vectors. VSM does not take into account relations between terms and it suffers of the synonymy and the polysemy problems (*e.g.,* it is not able to infer that words like *car* and *automobile* refer to the same concept).

Latent Semantic Indexing (LSI) [42] is an extension of the VSM developed to overcome these limitations. In LSI the dependencies between terms and between documents, in addition to the associations between terms and documents, are explicitly taken into account. For example, the terms *car* and *automobile* are likely to co-occur in different documents with terms like *motor* and *wheel*. To exploit information about co-occurrences of terms, LSI applies Singular Value Decomposition (SVD) [43] to project the original term-by-document matrix into a reduced space of concepts, thus limiting the noise that terms may cause. Basically, given a term-by-document matrix A, it is decomposed into:

$$A = T \cdot S \cdot D^T$$

where T is the term-by-concept matrix, D the document-by-concept matrix, and S a diagonal matrix composed of the concept eigenvalues. After reducing the number of concepts to k, the matrix A is approximated with:

$$A_k = T_k \cdot S_k \cdot D_k^T$$

Latent Dirichlet Allocation (LDA) [44] fits a generative probabilistic model from the term occurrences in a corpus of documents. The fitted model is able to capture an additional layer of latent variables which are referred as topics. Basically, a document can be considered as a probability distribution of topics—fitting the Dirichlet prior distribution—and each topic consists in a distribution of words that, in some sense, represent the topic. In particular, each document has a corresponding multinomial distribution over T topics and each topic has a corresponding multinomial distribution over the set of words in the vocabulary of the corpus. LDA assumes the following generative process for each document d_i in a corpus D:

1) Choose $N \sim$ Poisson distribution (ξ)
2) Choose $\theta \sim$ Dirichlet distribution (α)
3) For each of the N words w_n:
   a) Choose a topic $t_n \sim$ Multinomial (θ).
   b) Choose a word w_n from $p(w_n|t_n, \beta)$, a multinomial probability conditioned on topic t_n.

While applying IR techniques is quite easy and straightforward also thanks to the availability of open source implementations (see *e.g.,* the APACHE LUCENE search engine[10]), empirical studies have shown how suboptimal configuration of their parameters (*e.g.,* the LDA's α and β) might led to suboptimal performances when applied to software engineering related tasks [45], [46], [47], [48]. These works, also provide hints on how to automatically identify an appropriate configuration of IR techniques.

C. Natural Language Processing

NLP techniques allow to automatically extract meaning from pieces of text written in natural language. Besides the already mentioned stemming and abbreviation expansions, exemplar applications of NLP include (but are not limited to):

- *Text summarisation*: generate a natural language summary from a longer text. *E.g.,* summarise code components.
- *Autocompletion*: predicting which word (token) is likely to follow a given sequence of words (tokens) (*e.g.,* "blu" is more likely to follow the trigram "the sky is" with respect to "yellow"). Useful to support code autocompletion in IDEs.
- *Part-of-speech tagging*: determine the part of speech for each word in a given sentence (*i.e.,* tagging each word as noun, verb, adjective, adverb, *etc.*).
- *Sentiment analysis*: determine the polarity of sentences (*e.g.,* automatically tagging users' reviews as positive or negative).
- *Topic segmentation*: separate a given text into cohesive fragments talking about a specific topic (*e.g.,* segmenting a discussion in an issue tracker based on the different solutions proposed in it).
- *Optical Character Recognition (OCR)*: extracting the textual information represented in an image (*e.g.,* the source code snippets reported in a frame of a video tutorial).

NLP techniques are often built on top of statistical machine learners, able to derive linguistic rules from a large documents corpora. Taking the autocompletion task as example, it is possible to build a language model able to estimate the probability that a specific word (*e.g.,* w_m) follows a given n-gram (*e.g.,* w_1, w_2, ..., w_n) as:

$$P(w_1, w_2, \ldots, w_n, w_m) = \prod_{i=1}^{m} P(w_i|w_1, \ldots w_{i-1})$$

Also in the context of sentiment analysis statistical techniques are used to estimate the probability that a specific word in a sentence is related to particular emotions [49]. Simpler techniques assign a fixed weight (positive or negative) to each word in a sentence, or build upon available knowledge bases (see *e.g.,* WordNet[11]) to estimate the sentiment of sentences.

The reader interested in applying NLP techniques can find several software tools made available by the Stanford Natural Language Processing Group.[12]

[10] https://lucene.apache.org/core/

[11] https://wordnet.princeton.edu

[12] http://nlp.stanford.edu/software/

III. Unstructured Data in Software Repositories and Artefacts

This section overviews the unstructured data present in software artefacts and repositories by also highlighting perils (indicated with ⚠) to avoid while mining such information.

Versioning Systems (*e.g.,*`git`, `svn`, `cvs`) are probably the most mined repositories by MSR (Mining Software Repository) researchers. The basic unit of information here is represented by commit activities performed by developers. By mining such commits researchers can gather insights about the complete change history of a software system, knowing what (*i.e.,* which code components) has been modified at a given time, by who, and *why*. When talking about the *why*, unstructured data comes in.

Developers usually describe the "reason" why a certain commit has been performed in a commit message written in natural language. The analysis of commit messages can provide precious insights to researchers and find wide application in empirical studies as well as in the building of "golden sets" to evaluate software engineering recommenders. For example, as previously mentioned pattern matching techniques (Section II-A) can be applied to identify (and study) commits related to specific development activities, *e.g.,* commits implementing refactoring operations can be identified by looking for commit messages containing specific keywords, like *refactoring*, *refactored*, and *cleanup* [50].[13] The set of identified commits can be exploited (possibly, after a manual validation) to (i) study refactoring activities performed by developers, and (ii) evaluate a (semi-)automated refactoring recommender by comparing its recommendations against the "golden set" containing actual refactoring operations manually performed by developers over the change history of a software system.

While representing a very lightweight approach, the use of pattern matching techniques to identify commits related to specific activities (*e.g.,* refactoring) presents some perils. ⚠ **Commit messages are not always detailed enough to actually reflect what has been changed in a commit**. For instance, refactoring operations are often performed *while* fixing a bug and the commit message is likely to only describe the main activity performed in the commit (*e.g.,* "*Fixed bug NA-85*"). This problem is strictly related to what have been defined by Herzig and Zeller [52] as "tangled changes", *i.e.,* commit activities grouping together different types of changes (*e.g.,* a bug fix and the implementation of new features).

The mining of commit messages, again performed by using pattern matching techniques, can also be used to establish links between commits and issues (*e.g.,* bug fixes activities) present in the project *Issue Tracker*: A commit message reporting "*Throw exception on negative values; Fixes IO-375*" provides an explicit link toward the issue having IO-375 as id. The peril here are the ⚠ **missing explicit links between commits and issues** [53]. A commit message "*Throw exception on negative values*" does not provide any link to the issue it is related to. This problem can be overcome by adopting more sophisticated mining techniques like the RELINK approach by Wu *et al.* [54]. RELINK exploits a set of heuristics to link commits and issues, including the computation of IR-based textual similarity (Section II-B) between the issue description and the commit message as well as the analysis of information present in the issue's comments left by developers.

Establishing links between commits and issues is particularly useful, among other things, when performing qualitative analyses. Indeed, by reading the issue's description it is possible to complement the commit message and gather a better understanding of the changes that are applied in a given commit. Also, researchers could "tag" the commits based on their main purpose, as indicated by the type of issue (*e.g.,* bug fix, new feature, enhancement, *etc.*) they are linked to. However, ⚠ **empirical studies have highlighted the presence of issue misclassifications in issue trackers** [55], [56]. Finally, by mining the comments posted by developers in the related discussion it is possible to extract the rationale behind specific implementation choices.

Archived Communications, like mailing lists and chat logs, report information about messages exchanged by developers and other project's stakeholders. Here, the vast majority of the information is unstructured and written in natural language. These repositories can be mined to identify communication networks representing developers who are likely to cooperate in a project [57]. Note that ⚠ **resolving ambiguities (*e.g.,* developers using different names/emails on different communication channels) might be needed when mining archived communications** [58]. For example, a developer could use the nickname *gbavota* in a chat and the email address *gabriele.bavota@unibz.it* in the project's mailing list. The analysis of his social network requires the disambiguation of his different identifiers [58]. This also holds when applying topic modelling techniques (*e.g.,* LDA) to identify the topics characterising the communications of a specific developer. Such information could be exploited to identify people knowledgeable about a specific topic (*e.g.,* the developer more indicated to fix a bug—bug triaging). Also, the MUD techniques can be used to link developers' communications to other software artefacts (*e.g.,* source code classes), enriching the knowledge available about those artefacts (see *e.g.,* Bacchelli *et al.* [34]).

Online Forums refer to both specialised discussion forums (see *e.g.,* Apache Lounge[14]) as well as to general purpose forums like Q&A websites. Among those, Stack Overflow is by far the most mined by researchers in the software engineering community. While discussions are generally organised by topics via tagging mechanisms, the amount of unstructured information in such websites is prevalent. Also, Stack Overflow's discussions are rich of code snippets. For this reason, ⚠ **MUD techniques must often be exploited in combination with island parsers [59] to effectively mine information from these repositories** (*e.g.,* to identify relevant discussions for a given piece of code [23]).

[13] ⚠ **Note that the best way to identify refactoring operations is to analyse the source code modified in a specific commit**. However, tools identifying refactoring operations by code analysis (*e.g.,* Ref-Finder [51]) often require the compiled source code that might not be available.

[14] http://www.apachelounge.com

Another repository in which mining unstructured data is represented by the *Mobile App Stores* [60]. Apps on the stores are characterised, among other things, by (i) a textual description reporting its main features/updates with respect to the previous releases, and (ii) a set of users' reviews, written in natural language and accompanied by a rating indicating the user's satisfaction. MUD techniques, and in particular NLP, can be applied in such a context to extract useful pieces of information from users' reviews (*e.g.,* to identify bugs experienced by the user base [33]), or to estimate the apps' success based on their reviews [61]. **A peril to avoid when mining information from apps' reviews is to ⚠ consider all reviews as equally important and informative**. Indeed, as recently shown by Chen *et al.* [32], only a small percentage of reviews are informative (*i.e.,* contain useful information for the apps' developers).

The mining of the apps' textual description could instead help in identifying similar apps (competitors) or in checking the consistency of the app description with what is actually implemented [31].

Finally, it is worth remembering as the *Source Code* itself and the *Software Artefacts* like high- and low- level documentation are themselves a rich source of unstructured data. For example, comments left inside the code can be treated with NLP techniques to summarise code components [62] and the terms present in code comments and identifiers can be exploited via IR techniques to assess the quality of the code itself [16] or to link it to other software artefacts [37]. Clearly, **⚠ techniques exploiting textual information present in the source code assume the use of meaningful terms in the code comments and identifiers, which is not always the case** [17], [63].

IV. Existing Applications of MUD Techniques

MUD techniques have been applied in software engineering to support a wide range of tasks and as an instrument to perform dozens of empirical studies. The goal of this section is not to provide a complete review of all these works, but instead to discuss some of the research areas in which MUD techniques play a major role.

A. Generating Documentation

NLP techniques have been used to summarise changes occurred to code components [12], [13], [14]. Buse and Weimer proposed *DeltaDoc* [12], a technique to generate a human-readable text describing the behavioural changes caused by modifications to the body of a method. To this aim, the modified statements are statically located and the control flow paths conducting to them are symbolically executed to obtain the changes in the method's behaviour. This information is embedded into templates that constitute the description of what changed in a method.

Summarisation techniques have been applied by Cortes-Coy *et al.* [13], who proposed *ChangeScribe*, an Eclipse plug-in[15] that describes a given set of source code changes based on its stereotype, type of changes, and the impact set of the changes. Similarly, Rastkar and Murphy [14] exploit multi-document summarisation to automatically infer the rationale behind a code change from a set of textual documents (*e.g.,* bug reports or commit messages).

MUD techniques have also been used to summarise bug reports [9], [10], [11]. The focus of such approaches is on identifying and extracting the most relevant sentences of bug reports by using supervised learning techniques [9], network analysis [10], [11], and information retrieval [11].

Recently, Moreno *et al.* [3] presented ARENA,[16] a tool to automatically generate complete release notes. ARENA mines the project versioning system and issue tracker to identify changes performed between two releases. Then, it uses NLP techniques, and in particular text summarisation [64], to remove redundancy and create a compact yet complete release note.

At the source code level, automated summarisation research has focused on describing Object Oriented artefacts, such as classes and methods, by generating either term-based (*i.e.,* bag of words) [65], [66], [67], [68], [69] or text-based [62], [64] summaries. In both cases, the information to be included in the summaries is selected through structural heuristics and transformed into human-readable phrases by using NLP techniques. In such a context, interesting are the results reported by De Lucia *et al.* [66], who investigated how source code artefact labelling that is performed by IR techniques would overlap (and differ) with labelling performed by humans. They asked 17 undergraduate students to manually label 20 Java classes by using up to ten keywords for each of them. Then, the authors exploited four automatic labelling approaches to extract summaries from the same 20 classes. Three of them where IR techniques, and in particular VSM, LDA, and LSI. The fourth one, was a simple automatic labelling heuristic in which each class was labeled by considering only words composing (i) the class name, (ii) the signature of methods, and (iii) the attribute names. Among this set of words, the authors selected the most representative ones by ranking them using their *tf-idf*, however always considering the words contained in the class name as part of the summary. The effectiveness of each labelling technique has been evaluated in terms of words overlap it achieved with respect to the manual labelling. Surprisingly, the highest overlap was obtained by using the simplest heuristic, while the most sophisticated techniques, *i.e.,* LSI and LDA, provided the worst accuracy. The results of this study highlight as not always more complex techniques represent the panacea to achieve the best results when mining unstructured data.

MUD techniques have also been successfully used in conjunction with structural code analysis to automatically generate code examples [5], [6], [7], [70], [8]. MAPO, proposed by Xie and Pei [5] and extended by Zhong *et al.* [6], mines abstract usage examples for a given API method. MAPO analyses code snippets retrieved by code search engines to extract all the call sequences involving the desired API method. A subset of sequences covering all method calls is identified and then clustered into similar usage scenarios, according to heuristics based on method names, class names, and called API methods composing each sequence. For each cluster, MAPO identifies

[15] https://github.com/SEMERU-WM/ChangeScribe

[16] https://seers.utdallas.edu/ARENA/

usage patterns based on frequent call sequences, and, finally, it ranks the patterns based on their similarity with the developer's code context. The similarity here includes textual information extracted from the terms present in the code identifiers.

UP-MINER [7] is a variation of MAPO that reduces the redundancy in the resulting example list. To this end, UP-MINER clusters the extracted method sequences based on n-grams and discovers a pattern for each cluster by applying a frequent sequence mining algorithm. As these patterns might be similar, UP-MINER executes another clustering round on them. The resulting patterns are ranked according to their frequency and presented as probabilistic graphs.

Buse and Weimer [70] proposed to generate documented abstract API usages by extracting and synthesising code examples of a particular API data type. Their approach mines examples by identifying and ordering (i) code instantiations of the given data type and (ii) the statements relevant to those instantiations as defined by previously extracted path predicates, computed from intra-procedural static traces. The examples are then clustered based on their statement ordering and data type usage. For each cluster, an abstract example (*i.e.,* a usage pattern) is formed by merging its code examples, and finally documented according to predefined heuristics that depend on the kind of statement and most frequent names in the mined code.

More recently, Moreno *et al.* [8] presented MUSE, a technique for mining and ranking actual code examples that show how to use a specific method. Here, MUD techniques are employed to automatically comment the generated code examples. Given a method m_i for which MUSE generated an usage example, MUSE automatically extracts information from the m_i's Javadoc documentation and includes it in the code examples. In particular, MUSE extracts the textual descriptions of m_i's parameters (identified with the tag `@param`) and includes them as inline comments to explain the arguments passed to m_i invocation right where it occurs.

B. Identifying Related Software Artefacts

IR techniques have been applied to the problem of recovering traceability links between different categories of high- (*e.g.,* requirements) and low-level (*e.g.,* source code) software artefacts. The conjecture here is that artefacts containing similar terms (*i.e.,* having a high textual similarity as captured by the IR technique) have a higher likelihood of being related with respect to artefacts characterised by different vocabularies. A complete treatment of the IR-based traceability topic can be found in [37]. In a similar fashion IR techniques have also been applied to the concept location task with the goal of identifying relevant code components for a specific bug report [35].

Wang *et al.* [30] used both natural language information and execution information to detect duplicate bug reports. Given a newly submitted bug report b_n their approach exploits IR techniques to compute the similarity between b_n and all already existing bug reports. Given two bug reports under analysis (*i.e.,* b_n and one of the already existing reports) the IR-based similarity is computed at two different levels: First, between the title and the description of the two reports. Second, between the terms of the execution traces reported for the two bugs. Pairs of reports exhibiting high values of similarity are candidate to be marked as duplicated.

Bacchelli *et al.* [34] exploited pattern matching techniques to link e-mails and source code artefacts. Their approach, based on simple regular expressions (see Section II-A) matching the name of the code artefacts in the email body, outperformed more complex IR techniques in an evaluation conducted over six software systems. This finding supports what we previously observed while discussing the work by De Lucia *et al.* [66]: Not always employing more sophisticated techniques leads to better results in the mining of unstructured data. However, as also noticed by the authors, the pattern matching technique implemented via regular expression immediately showed its limitations (*i.e.,* its low flexibility) when moving from java based systems toward different programming languages. Indeed, the authors had to adapt their technique (*i.e.,* to extend the exploited regular expression) in order to effectively deal with syntactic features of the new programming languages. Also, while IR techniques provide a ranked list of relevant results indicating the relevance of each document (here, a code artefact) for a given query (here, an e-mail), for pattern matching methods a document (the e-mail) either matches or not the regular expression. Thus, if n e-mails match a regular expression, the developers have to analyse all of them to identify the ones actually relevant for the code artefact.

C. Generating Recommendations for Software Developers and Managers

Several works apply MUD techniques to identify documents, discussions, and code samples relevant for a given (development) task. Chatterjee *et al.* [71] and Keivanloo *et al.* [72] use textual similarity to return a ranked list of abstract examples relevant to a natural language (NL) query formulated by the user and expressing her task at hand. Keivanloo *et al.*'s approach combines textual similarity and clone detection techniques to find relevant code examples, and ranks them according to (i) their textual similarity to the query and (ii) the completeness and popularity of their encoded patterns. In a similar token, but generating a ranked list of concrete API usage examples, the Structural Semantic Indexing (SSI), proposed by Bajracharya *et al.* [73], combines heuristics based on structural and textual aspects of the code, based on the assumption that code entities containing similar API usages are also similar from a functional point of view.

Other work focused on suggesting relevant documents, discussions and code samples from the web to fill the gap between the IDE and the Web browser. Examples are CODETRAIL [27], MICA [28], FISHTAIL [74], and DORA [75]. Subramanian *et al.* [29] presented an approach to link webpages of different nature (*e.g.,* javadoc, source code, Stack Overflow) by harnessing code identifiers. They recommend augmented webpages by injecting code that modifies the original web page.

Among the various sources available on the Web, Q&A Websites and in particular Stack Overflow, have been the target of many recommender systems [24], [25], [26], [23]. Ponzanelli *et al.* proposed PROMPTER[17] [23], an Eclipse plug-in continuously tracking the Eclipse's code context every time a change in the source code occurs. The extracted code context is

[17] http://prompter.inf.usi.ch

treated with NLP techniques to extract the most discriminating terms representing it and automatically generate a textual query which is sent to search engines (Google, Bing) to perform a Web search on the Stack Overflow's website. Then, every retrieved discussion is ranked according to a *Ranking Model* taking into account textual and structural information extracted from the developer's code context. The top ranked discussion is pushed directly in the IDE if its similarity with the code context is higher than a given threshold.

Canfora *et al.* presented YODA[18] [19], an approach to identify likely mentors in software projects by mining data from software repositories, and to support the project manager in recommending possible mentors when a newcomer joins a project. YODA adopts a two-step process. First, it laverages Social Network Analysis methods to identify candidate mentors in the past history of a software project (*e.g.,* a good mentor should be quite active in the project). Once identified the set of project's members who are good candidates to be mentors, MUD techniques are exploited in the second YODA's step to select the most appropriate mentor for a given newcomer p. Suppose that p joins the project at time t_x and that a set of n mentors $M = \{m_1, \ldots, m_n\}$ has been identified, in the first step, the period before t_x. An IR process is used to rank the available mentors, where each document d_i with $i = 1, \ldots, n$ in the corpus consists of the union of the text of all emails exchanged by the mentor m_i before t_x, while the query q_p is represented by a request for help submitted by the newcomer p. Note that, the IR-based methodology exploited by YODA is very similar to those applied in the context of bug triaging [20], [21], [22], where the goal is to assign a bug to the most appropriate fixer.

D. Classifying Textual Artefacts

MUD techniques have been applied to help mobile apps developers in identifying useful requirements (*e.g.,* bugs to fix, suggestions for new features) from reviews left by their users. Galvis and Winbladh [76] extract the main topics in app store reviews and the sentences representative of those topics with the goal of capturing the mood and feelings of the apps' users.

Iacob and Harrison [77] provided empirical evidence of the extent users of mobile apps rely on reviews to describe feature requests, and the topics that represent the requests. Among 3,279 reviews manually analyzed, 763 (23%) expressed feature requests. Then, linguistic rules were exploited to define an approach, coined as MARA, to automatically identify feature requests.

Linguistic rules have also been recently exploited by Panichella *et al.* [33] in conjunction with sentiment analysis to classify sentences in app reviews into four categories: Feature Request, Problem Discovery, Information Seeking, Information Giving. Through a manual inspection of 500 reviews, the authors identified 246 recurrent linguistic patterns in users' reviews, and defined NLP heuristics to automatically recognise them. For example, by using NLP is possible to catch patterns in the form:

```
[someone] should add [something]
```

These patterns clearly indicate a suggestion for new features in the user's review. As for the sentiment analysis, Panichella *et al.* exploited Naive Bayes to classify the sentiment of a review in a range between -1 (negative review) and 1 (positive review). Similar techniques for labelling app reviews have also been proposed by McIlroy *et al.* [78] and by Villarroel *et al.* [79].

Chen *et al.* [32] pioneered the prioritisation of user reviews with AR-MINER, a tool to automatically filter and rank informative reviews. Informative reviews are identified by using a semi supervised learning-based approach exploiting textual features, and in particular the terms present in the reviews. Reviews like "*This is a crap app*" are categorised as non-informative, while those containing potentially useful feedbacks (*e.g.,* "*The app crashes when touching the home button*") are tagged as informative. Once discriminated informative from non-informative reviews, AR-MINER groups them into topics and ranks the groups of reviews by priority. The grouping step is realised by exploiting topic modelling techniques (LDA) and the Aspect and Sentiment Unification Model (ASUM) [80].

Bacchelli *et al.* [81] presented an approach to classify the text present in email messages at line level granularity. Each line is classified into one of the following five categories: text, junk, code, patch, or stack trace. The approach uses machine learners to perform a term based classification of the email's lines and island parsers to deal with specific cases and improve the overall classification accuracy. An important lesson from this work is that while simple IR techniques can be easily used to mine unstructured data (and in the specific case, to obtain a first classification), adopting customised solutions for the problem at hand can lead to much better results.

Gorla *et al.* [31] exploited MUD techniques in CHABADA, a tool aimed at verifying if the *advertised* behaviour of Android apps correctly reflects the *implemented* behaviour. The advertised behaviour is extracted from the natural language apps' descriptions available in the Google Play store (see Section III), while the implemented behaviour is represented by the invoked Android Application Programming Interfaces (APIs). CHABADA uses LDA to assign each app (*i.e.,* each app description) to a set of topics and then cluster together apps by related topics (*e.g.,* it is reasonable to think that messaging apps share similar topics characterised by words like "message", "send", "reply", *etc.*). Then, CHABADA extracts the sensitive[19] APIs each app invokes and identifies "outliers behaviours" inside each cluster of apps. For example, it would be suspicious if a sms messaging app requires the device's location. CHABADA can thus be used to identify malicious apps' behaviours (*i.e.,* to classify the app's description as suspicious or not).

[18] http://www.ifi.uzh.ch/seal/people/panichella/tools.html

[19] Sensitive APIs are those requiring an user permission, like the ones providing access to the device's location.

E. Assessing and Improving Code Quality

IR techniques have been used to assess the quality of object oriented classes in terms of cohesion and coupling. Poshyvanyk *et al.* [15] proposed the Conceptual Coupling Between Classes (CCBC). CCBC is based on the textual information (terms) present in code comments and identifiers. Two classes are conceptually related if their (domain) semantics are similar, *i.e.,* their terms indicate similar responsibilities. CCBC has been shown to capture new dimension of coupling, which are not captured by the conventional structural metrics (*e.g.,* those measuring coupling as the number of structural dependencies between two classes). Similarly, Marcus *et al.* [16] presented the Conceptual Cohesion of Classes (C3), assessing the cohesion of a class as the average textual similarity between all unordered pairs of methods in it.

Still in the context of assessing code components' quality by analysing the textual information they contain, Arnaoudova *et al.* [17], [63] introduced the concept of linguistic anti-patterns, related to inconsistencies (i) between method signatures, documentation, and behaviour and (ii) between attribute names, types, and comments.

Finally, textual information extracted from code components have been used to semi-automatise refactoring activities [82], [18], [83]. Such information is particularly useful when building tools aimed at recommending refactoring solutions moving code components inside the software system to better organise the implemented responsibilities (*e.g.,* extract class, move method, move class, *etc.*). Indeed, by analysing textual relationships between code components (*e.g.,* the conceptual coupling between classes previously described) it is possible, for example, to spot classes placed in the wrong package as the ones having a low CCBC with the classes in the package they belong to and a high CCBC with classes in another package.

V. Future Trends in MUD

This section describes a set of possible future trends the author expects for the MUD research in the software engineering context. Clearly, the listed future trends are the results of a very personal (and partial) view.

A. MUD to Support Qualitative Analysis

Nowadays the empirical research in software engineering is pushing toward the combination of quantitative and qualitative findings. Indeed, while quantitative data that is collected through carefully designed experiments can provide us with a statistically significant piece of evidence about a phenomenon (*e.g.,* classes exhibiting specific characteristics are more bug-prone than other classes) they hardly tell *why* we obtained such a result (*e.g.,* why developers introduce more bugs while working on such classes). To properly investigate the "why side" of an experiment, complementing quantitative and qualitative findings can be the way to go.

In this context, the mining of unstructured data can play a major role. Let us consider the example of a researcher who performs a quantitative study aimed at understanding the relationship between the code readability (as assessed by a specific metric, see *e.g.,* Buse and Weimer [84]) and the code bug-proneness. In other words, the researcher is interested in investigating if developers tend to introduce more bugs when working on code components having a low readability. Let us suppose that the quantitative results collected by the researcher highlights that there is a strong, statistically significant negative correlation between code readability and bug-proneness (*i.e.,* the higher the code readability, the lower the code bug-proneness). The problem here is that correlation $\neq$ causation and the reason behind such a result (the "why side") might be less obvious than it appears. Indeed, it could be that: (i) low code readability hampers code comprehension, thus favouring the bug introduction, **or** (ii) frequent bug fixes possibly applied in a rush negatively affect the code quality, thus decreasing its readability. In these cases, MUD can help in extracting qualitative data aimed at better understanding the phenomenon and avoid unsupported claims.

For instance, pattern matching analysis or more sophisticated NLP techniques can be used to automatically isolate developers' discussions (*e.g.,* those archived in the projects' mailing list, issue trackers, *etc.*) related to the code components exhibiting low readability and/or high bug-proneness, in order to understand what are the issues discussed by the developers, *e.g.,* are they complaining about difficulties in comprehending the code or about the few hours they have to fix the bug? Note that most of the times MUD techniques can only provide a first filtering of the qualitative data (in this case, the developers' discussions) that then must be manually analysed.

A strongly recommended paper highlighting the need for qualitative analysis to complement the quantitative findings is: "*Failure is a four-letter word: a parody in empirical research*" by Zeller *et al.* [85].

B. Crosscutting Analysis

When mining (unstructured) data from a software repository, researchers should be aware of the very partial view it could give about a specific phenomenon. An example is provided in the work by Panichella *et al.* [86], where the authors analysed developers' communications over different channels (mailing lists, issue trackers, IRC chat) to understand to what extent findings drawn by looking in isolation in a specific communication channel also generalise to the other channels.

Results of the study highlighted that analysing developers collaboration/communication through specific channels would only provide a partial view of the reality, and that different channels may provide different perspectives of developers' communication. In particular, (i) not all developers use all communication channels; and (ii) people mainly interact through two out of three communication channels, whereas the third one is only used sporadically. Therefore, if using specific collaboration/communication networks for various purposes—*e.g.,* identifying experts or mentors—one should be careful as different channels may lead to more or less accurate—and in any case different—results.

The take-away here is that, when possible, MUD researchers should look at different sources of information and integrate the pieces of evidence in each of them to strength claimed findings or gather a better knowledge of the studied phenomenon.

C. Inter-domain Recommendations

As discussed in Section IV-C, MUD techniques have been exploited to build several different types of recommenders. Some of these recommenders could be defined as "intra-domain": they look into a specific fenced gardens for things to recommend, without ever climb over the fence. Let us consider the example of the recommenders built to extract requirements from mobile apps' reviews (see *e.g.,* references [32], [33], [77], [78], [79], discussed in Section IV-D). Given a mobile app App_i, the goal of these tools is to provide App_i's developers with useful information to plan App_i's next release: Which features should be implemented? Which bugs should be fixed? *etc.* These recommenders are intra-domain since they look for requirements (things to recommend) only among App_i's reviews (the fenced garden).

Suppose that App_i is a word processor app. An *inter-domain recommender* in this context could look on the app store for all reviews belonging to word processor apps, thus climbing over the fence and recommending, for example, features that have been suggested not only by App_i's users, but also by those of all word processors in the app store. Similarly, one could spot features that are particularly appreciated by users' of competitive apps. Clearly, there would be a number of non-trivial problems to overcome: (i) how to detect similar apps among the millions available in the app store, (ii) how to discard features recommended for a word processor App_j that are already implemented in App_i, (iii) how to discriminate between successful and unsuccessful features discussed in the users' comments, *etc.*

MUD techniques really fit in the building of this type of inter-domain recommenders, since the MUD infrastructure developed to extract data in a specific context can generally be easily extended to wider contexts, increasing the possibilities such recommenders offer.

D. Mining Multimedia Contents

When talking about mining unstructured data in the software engineering context most of people think to textual information spread in software repositories. This is also clear by the (partial) analysis of the MUD literature presented in Section IV. However, unstructured data are also, by definition, multimedia contents, like images and videos. These unstructured data are nowadays *almost* ignored in the software engineering community but, as also confirmed by a very recent study conducted by MacLeod *et al.* [87], they embed very precious information.

MacLeod *et al.* performed an empirical study aimed at investigating how and why developers create video tutorials and host them on YouTube. They found that they share, among other things, demonstrations of code, personal development experiences, implementation approaches, and also provide the audience with live demonstrations showing the execution of the implemented code. The study also highlighted key advantages of video tutorials compared to other resources. In particular, the authors highlight that [87]:

> "*Unlike other documentation methods that are primarily text based, screencasts allow developers to share their coding practices through live interactions. These video captured interactions allow developers to present tacit knowledge would be otherwise difficult to present through textual methods.*"

These observations, and in particular the complementarity of multimedia contents with respect to textual information, highlight the strong potential of mining more unconventional unstructured data, like videos (*e.g.,* the mentioned tutorials), and images (*e.g.,* slides projected during a recorded University's lesson). The only work in this direction is the CODETUBE tool developed by Ponzanelli *et al.* [88], and able to mine video tutorials found on the web enabling developers to query their contents. The video tutorials are processed and split into coherent fragments, to return only fragments related to the query.

E. Consumer-related and Task-related Customisation of MUD Techniques

MUD techniques are often applied out of the box to support software engineering tasks. For instance, summarisation algorithms have been exploited to summarise object oriented code artefacts (see Section IV-A). The content of the generated summary generally represents an overview of the main responsibilities implemented in the code component *e.g.,* in a class.

While these approaches are certainly valuable, they do not customise the generated summaries on the basis of *who* (consumer) will read them and to support *which task*. Indeed, it is reasonable to think that a developer newcomer needs different information in the summary with respect to an experienced developer. Also, a developer in charge of writing the unit tests for a given class is likely interested in different things (*e.g.,* the covered statements) with respect to a developer in charge of writing the class documentation for third-party usage of the class's methods. For these reasons, consumer-related (who will consume the summary) and task-related (for what the summary will be used) summarisation techniques could led to the generation of more useful pieces of documentation.

The summarisation is just an example of the MUD techniques that could be exploited in a consumer-related and/or task-related customisation fashion.

VI. SUMMARY

This paper has provided an overview of the mining of unstructured data in the software engineering field, by focusing on the analysis of textual information in software repositories. Widely adopted MUD techniques have been illustrated and a description of the software repositories containing unstructured data has been provided.

The survey about the current trends in applying MUD techniques in software engineering had the goal of providing the reader with a general idea of what can be done with such techniques, while the listed future trends reflect the personal author's view of five promising research areas. The take away of the paper is summarised in Fig. 1.

ACKNOWLEDGMENT

The author would like to thank Rocco Oliveto for his feedbacks on a preliminary version of this paper.

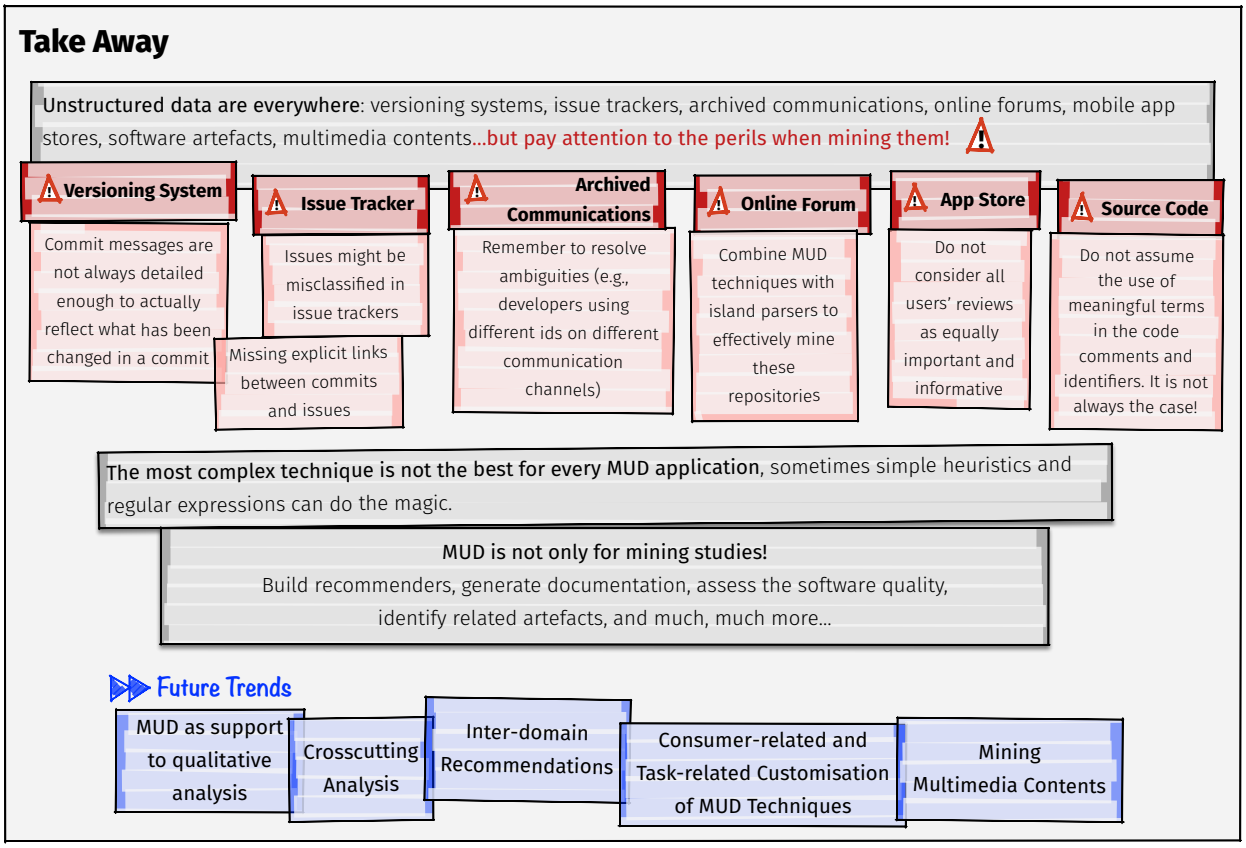

Fig. 1. Paper's Take Away.

REFERENCES

[1] A. Holzinger, C. Stocker, B. Ofner, G. Prohaska, A. Brabenetz, and R. Hofmann-Wellenhof, "Combining hci, natural language processing, and knowledge discovery - potential of ibm content analytics as an assistive technology in the biomedical field," in *Human-Computer Interaction and Knowledge Discovery in Complex, Unstructured, Big Data*, ser. Lecture Notes in Computer Science, 2013, vol. 7947, pp. 13–24.

[2] "Github press," 2015. [Online]. Available: https://github.com/about/press

[3] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 484–495.

[4] S. Abebe, N. Ali, and A. Hassan, "An empirical study of software release notes," *Empirical Software Engineering*, pp. 1–36, 2015.

[5] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 54–57.

[6] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proceedings of the 23rd European Conference on ECOOP 2009*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[7] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 319–328.

[8] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?" in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, 2015, pp. 880–890.

[9] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 505–514.

[10] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried' bug report reading process to summarize bug reports," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Riva del Garda, Trento, Italy, September 23-28, 2012*. IEEE Computer Society, 2012, pp. 430–439.

[11] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 11:1–11:11.

[12] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 33–42.

[13] L. Cortes-Coy, M. Linares-Vasquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, Sept 2014, pp. 275–284.

[14] S. Rastkar and G. C. Murphy, "Why did this code change?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 1193–1196.

[15] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proc. of 22nd IEEE ICSM*. Philadelphia, Pennsylvania, USA: IEEE CS Press, 2006, pp. 469 – 478.

[16] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual

cohesion of classes for fault prediction in object-oriented systems," *IEEE Trans. Software Eng.*, vol. 34, no. 2, pp. 287–300, 2008.

[17] V. Arnaoudova, M. D. Penta, G. Antoniol, and Y. Guéhéneuc, "A new family of software anti-patterns: Linguistic anti-patterns," in *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*, 2013, pp. 187–196.

[18] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, p. 1, 2013.

[19] G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?" in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 44.

[20] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 365–375.

[21] J. Anvik and G. C. Murphy, "Reducing the effort of bug report triage: Recommenders for development-oriented decisions," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000791.2000794

[22] G. Canfora and L. Cerulo, "Supporting change request assignment in open source development," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06, 2006, pp. 1767–1772.

[23] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to turn the IDE into a self-confident programming prompter," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. ACM, 2014, pp. 102–111.

[24] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *Proceedings of RSSE 2012*. IEEE Press, 2012, pp. 85–89.

[25] P. Rigby and M. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of ICSE 2013*, 2013, pp. 832–841.

[26] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of SUITE 2011*. ACM, 2011, pp. 17–20.

[27] M. Goldman and R. Miller, "Codetrail: Connecting source code and web resources," *Journal of Visual Languages & Computing*, pp. 223–235, 2009.

[28] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding api components and examples," in *Proceedings of VL/HCC 2006*, 2006, pp. 195–202.

[29] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of ICSE 2014 (36th International Conference on Software Engineering)*, ser. ICSE 2014. ACM, 2014, pp. 643–652.

[30] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, pp. 461–470.

[31] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 1025–1035.

[32] N. Chen, J. Lin, S. Hoi, X. Xiao, and B. Zhang, "AR-Miner: Mining informative reviews for developers from mobile app marketplace," in *36th International Conference on Software Engineering (ICSE'14)*, 2014, p. To appear.

[33] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *Proceedings of the 31st International Conference on Software Maintenance and Evolution*, ser. ICSME 2015, 2015, p. To appear.

[34] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. ACM, 2010, pp. 375–384.

[35] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, p. 23, 2012.

[36] T.-H. Chen, S. Thomas, and A. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Software Engineering*, pp. 1–77, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9402-8

[37] A. D. Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, "Information retrieval methods for automated traceability recovery," in *Software and Systems Traceability.*, 2012, pp. 71–98.

[38] E. Hill, D. Binkley, D. J. Lawrie, L. L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, 2014.

[39] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[40] D. Lawrie and D. Binkley, "Expanding identifiers to normalize source code vocabulary," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, 2011, pp. 113–122.

[41] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[42] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[43] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*. Boston: Birkhauser, 1998, vol. 1, ch. Real rectangular matrices.

[44] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *The Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[45] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 522–531.

[46] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang, "Improving trace accuracy through data-driven configuration and composition of tracing features," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, 2013, pp. 378–388.

[47] S. Thomas, M. Nagappan, D. Blostein, and A. Hassan, "The impact of classifier configuration and classifier combination on bug localization," *Software Engineering, IEEE Transactions on*, vol. 39, no. 10, pp. 1427–1443, 2013.

[48] L. Moreno, G. Bavota, S. Haiduc, M. D. Penta, R. Oliveto, B. Russo, and A. Marcus, "Query-based configuration of text retrieval solutions for software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 567–578.

[49] R. Stevenson, J. Mikels, and T. James, "Characterization of the affective norms for english words by discrete emotional categories," *Behavior Research Methods*, vol. 39, no. 4, pp. 1020–1024, 2007.

[50] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. ACM, 2008, pp. 35–38.

[51] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, 2010, pp. 1–10.

[52] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 121–130.

[53] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, 2010, pp. 97–106.

[54] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: recovering links between bugs and changes," in *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 15–25.

[55] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: a text-based approach to classify change requests," in *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. IBM, 2008, p. 23.

[56] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: how misclassification impacts bug prediction," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE / ACM, 2013, pp. 392–401.

[57] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006*, 2006, pp. 137–143.

[58] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "Social interactions around cross-system bug fixings: the case of FreeBSD and OpenBSD," in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, 2011, pp. 143–152.

[59] L. Moonen, "Generating robust parsers using island grammars," in *8th IEEE Working Conference on Reverse Engineering (WCRE)*, 2001, pp. 13–22.

[60] M. Harman, Y. Jia, and Y. Zhang, "App store mining and analysis: Msr for app stores," in *9th IEEE Working Conference on Mining Software Repositories (MSR'12)*, 2012, pp. 108–112.

[61] G. Bavota, M. L. Vásquez, C. E. Bernal-Cárdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of API change- and fault-proneness on the user ratings of android apps," *IEEE Trans. Software Eng.*, vol. 41, no. 4, pp. 384–407, 2015.

[62] G. Sridhara, E. Hill, D. Muppaneni, L. L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 43–52.

[63] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Guéhéneuc, "REPENT: analyzing the nature of identifier renamings," *IEEE Trans. Software Eng.*, vol. 40, no. 5, pp. 502–532, 2014.

[64] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Proceedings of the IEEE International Conference on Program Comprehension*, ser. ICPC '13. IEEE, 2013, pp. 23–32.

[65] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of 17th IEEE Working Conference on Reverese Engineering*. Beverly, MA: IEEE CS Press, 2010, pp. 35–44.

[66] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using ir methods for labeling source code artifacts: Is it worthwhile?" in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, June 2012, pp. 193–202.

[67] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 279–290.

[68] P. W. McBurney, C. Liu, C. McMillan, and T. Weninger, "Improving topic model source code summarization," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 291–294.

[69] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 390–401.

[70] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE, 2012, pp. 782–792.

[71] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A search engine for java using free-form queries," in *Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2009, pp. 385–400.

[72] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *ICSE*. ACM, 2014, pp. 664–675.

[73] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 157–166.

[74] N. Sawadsky and G. Murphy, "Fishtail: from task context to source code examples," in *Proceedings of TOPI 2011*. ACM, 2011, pp. 48–51.

[75] O. Kononenko, D. Dietrich, R. Sharma, and R. Holmes, "Automatically locating relevant programming help online," in *Proceedings of VL/HCC 2012*, 2012, pp. 127–134.

[76] L. V. G. Carreno and K. Winbladh, "Analysis of user comments: An approach for software requirements evolution," in *35th International Conference on Software Engineering (ICSE'13)*, 2013, pp. 582–591.

[77] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *10th Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 41–44.

[78] S. McIlroy, N. Ali, H. Khalid, and A. E. Hassan, "Analyzing and automatically labelling the types of user issues that are raised in mobile app reviews," *Empirical Software Engineering*, pp. 1–40, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10664-015-9375-7

[79] L. Villarroel, G. Bavota, B. Russo, R. Oliveto, and M. Di Penta, "Release planning of mobile apps based on user reviews," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE 2016, 2016, p. To appear.

[80] Y. Jo and A. H. Oh, "Aspect and sentiment unification model for online review analysis," in *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, ser. WSDM '11. New York, NY, USA: ACM, 2011, pp. 815–824. [Online]. Available: http://doi.acm.org/10.1145/1935826.1935932

[81] A. Bacchelli, T. D. Sasso, M. D'Ambros, and M. Lanza, "Content classification of development emails," in *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 375–385.

[82] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.

[83] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Improving software modularization via automated analysis of latent topics and dependencies," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, p. 4, 2014.

[84] R. P. Buse and W. R. Weimer, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010.

[85] A. Zeller, T. Zimmermann, and C. Bird, "Failure is a four-letter word: A parody in empirical research," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, ser. Promise '11, 2011, pp. 5:1–5:7.

[86] S. Panichella, G. Bavota, M. D. Penta, G. Canfora, and G. Antoniol, "How developers' collaborations identified from different sources tell us about code changes," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 251–260.

[87] L. MacLeod, M.-A. Storey, and A. Bergen, "Code, camera, action: How software developers document and share program knowledge using YouTube," in *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*, 2015.

[88] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza, "Too long; didnt watch! extracting relevant fragments from software development video tutorials," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE 2016, 2016, p. To appear.