Nazar N, Hu Y, Jiang H. Summarizing software artifacts: A literature review. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(5): 883–909 Sept. 2016. DOI 10.1007/s11390-016-1671-1

# Summarizing Software Artifacts: A Literature Review

Najam Nazar[1], Yan Hu[1], *Member, CCF, ACM*, and He Jiang[1,2,*], *Member, CCF, ACM*

[1]*Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, School of Software Dalian University of Technology, Dalian 116621, China*

[2]*State Key Laboratory of Software Engineering, Wuhan University, Wuhan 430072, China*

E-mail: najamnazar@mail.dlut.edu.cn; {huyan, jianghe}@dlut.edu.cn

Received November 20, 2015; revised July 30, 2016.

**Abstract** This paper presents a literature review in the field of summarizing software artifacts, focusing on bug reports, source code, mailing lists and developer discussions artifacts. From Jan. 2010 to Apr. 2016, numerous summarization techniques, approaches, and tools have been proposed to satisfy the ongoing demand of improving software performance and quality and facilitating developers in understanding the problems at hand. Since aforementioned artifacts contain both structured and unstructured data at the same time, researchers have applied different machine learning and data mining techniques to generate summaries. Therefore, this paper first intends to provide a general perspective on the state of the art, describing the type of artifacts, approaches for summarization, as well as the common portions of experimental procedures shared among these artifacts. Moreover, we discuss the applications of summarization, i.e., what tasks at hand have been achieved through summarization. Next, this paper presents tools that are generated for summarization tasks or employed during summarization tasks. In addition, we present different summarization evaluation methods employed in selected studies as well as other important factors that are used for the evaluation of generated summaries such as adequacy and quality. Moreover, we briefly present modern communication channels and complementarities with commonalities among different software artifacts. Finally, some thoughts about the challenges applicable to the existing studies in general as well as future research directions are also discussed. The survey of existing studies will allow future researchers to have a wide and useful background knowledge on the main and important aspects of this research field.

**Keywords** mining software repositories, mining software engineering data, machine learning, summarizing software artifacts, summarizing source code

## 1 Introduction

During software maintenance, developers aim to improve the quality of information captured in different forms of software artifacts such as requirement documents, bug reports, source code. These software artifacts may contain excessive information, which is difficult to comprehend. A developer often ends up with glancing or skimming through the details of an artifact to retrieve the desired information, which in turn is tedious and time-consuming. Consequently, summarization systems are proposed based on a set of diverse techniques including data mining and machine learning. Summarization aims to obtain a reductive transformation from a source text to a summary text through different techniques[1]. This summarization task is essential for developers as it helps in saving time, resources, and efficiently managing the information contained in artifacts. It also assists developers in finding the specific information they sought for in an artifact rapidly[1].

Summarization systems have experienced a great development and in recent years, a wide variety of

Survey

Special Section on Software Systems 2016

This work was supported in part by the National Basic Research 973 Program of China under Grant No. 2013CB035906, the Fundamental Research Funds for the Central Universities of China under Grant No. DUT13RC(3)53, and in part by the New Century Excellent Talents in University of China under Grant No. NCET-13-0073 and the National Natural Science Foundation of China under Grant No. 61300017.

*Corresponding Author

techniques (ranging from simple text retrieval to complex heuristics), and paradigms have been proposed to tackle summarization tasks. However, producing an automatic summary is a challenging task. For instance, Murphy[2] in her Ph.D. dissertation, addressed the structural summarization of source code based on software reflection and lexical source model extraction, which are complex processes. Her techniques set standards for studies afterwards and are complemented by future studies, e.g.,[3-4]. Summarization also aims to achieve different objectives. For instance, Rastkar *et al.*[5] investigated if generating automatic summaries of bug reports could help in detecting duplicate bug reports. Another important aspect of summarization is its evaluation. This is very challenging because it is unclear what type of information a summary should contain. Moreover, summarization is a subjective process and there does not exist the best summary for a given task. In general, statistical methods such as precision, recall, and $F$-score are employed to evaluate the effectiveness of generated summaries along with the human evaluation. However, issues such as redundancy, consistency, sentences ordering, conciseness, adequacy, while constructing summaries, have made this field more difficult.

In this paper, we perform a literature review of the state-of-the-art studies in software artifact summarization, focusing on bug reports, source code, mailing lists and developer discussions artifacts. It is not a comprehensive review of all systems and techniques that have been developed since the advent of this research area, because we only target the latest ongoing trend from Jan. 2010 to Apr. 2016. Therefore, the first dimension of this paper is to provide a general overview of existing software artifact summarization techniques or systems, which can be of great help for developers during software maintenance. We also discuss the applications of summarization, i.e., what tasks are achieved through the summarization process (the second dimension). The third dimension details tools that have been employed while developing summarization systems or generated as a result of summarization task. The fourth dimension concerns the evaluation of generated summaries. The last dimension collects and distributes studies over the recent years that are selected for review. It also defines the information sources and the methodology for collecting studies. Finally, we discuss some important challenges pertaining to bug reports, source code, mailing lists, and developer discussions summarization techniques in order to facilitate future researchers in understanding this field thoroughly.

The structure of the paper is organized as follows. Section 2 provides an unabridged overview of software artifact summarization, discussing types of artifacts, how summarization systems are built and the existing approaches for generating summaries. Section 3 discusses the applications of summarizing software artifacts with respect to the selected studies. Summarization tools are listed in Section 4 and Section 5 distinguishes different summary evaluation methods employed in selected studies. Section 6 provides the list of information sources and distribution of selected studies along with the methodology of selecting studies used in this review. In Section 7, we present modern communication channels, discuss complementarities and commonalities among different artifacts, and provide challenges concerning software artifact summarization along with the future directions. Section 8 concludes our paper.

## 2 Summarizing Software Artifacts: An Overview

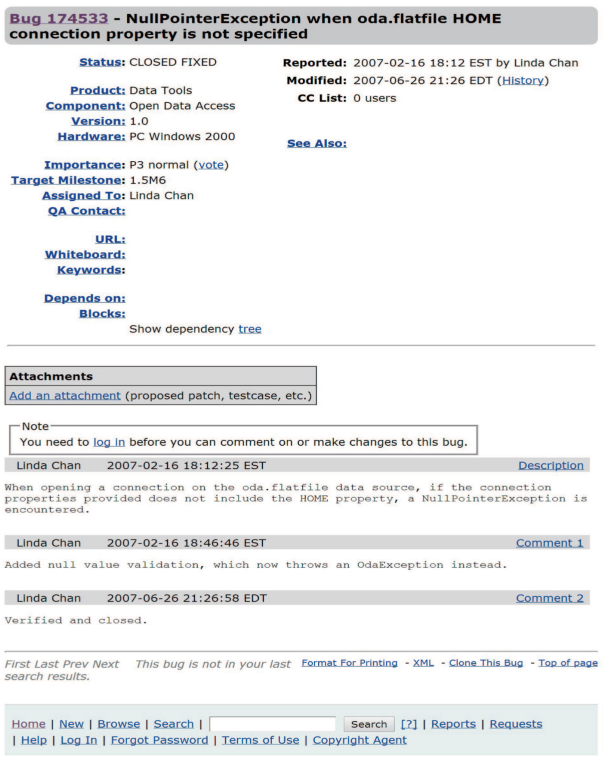
In this section, we first provide an overview of software artifact summarization, starting from the common types of software artifacts (Subsection 2.1), moving to the steps for summarization process (Subsection 2.2) and ending at the brief overview of existing approaches (Subsection 2.3). The existing approaches employed in selected studies are categorized based on data mining and machine learning approaches.

### 2.1 Types of Software Artifacts

There are many different types of software artifacts. However, for this survey, we select bug reports, source code, mailing lists and developer discussions artifacts only. Other software artifacts, such as chat logs, execution logs, requirement documents, are not discussed in the survey as they are out of the scope of this paper. In the subsections below we briefly describe these artifacts.

#### *2.1.1 Bug Reports*

A bug tracking system maintains information about software bugs in the form of bug reports. It contains the information about the creation and the resolution of bugs, feature enhancements, and other maintenance tasks. A typical bug report contains the title of a problem, bug fields providing metainformation about the bug report, a description in natural language text written by a reporter, and the comments by other users and developers. It also contains source code snippets, patches for corrections, enumerations, and stack traces[6]. In bug report summarization, a bug report is considered as a natural language text where different contributors discuss problems in a conversational way. Therefore, components such as code snippets and stack traces are generally ignored during bug report summarization. However, these components may help in improving the goodness or usefulness of the generated summaries. Popular bug database systems include Bugzilla[1] and JIRA[2]. Fig.1 exhibits a typical bug report, bug #174533 extracted from the Eclipse bug repository[3].


**Bug 174533 - NullPointerException when oda.flatfile HOME connection property is not specified**

**Status:** CLOSED FIXED

**Product:** Data Tools
**Component:** Open Data Access
**Version:** 1.0
**Hardware:** PC Windows 2000

**Importance:** P3 normal (vote)
**Target Milestone:** 1.5M6
**Assigned To:** Linda Chan
**QA Contact:**

**URL:**
**Whiteboard:**
**Keywords:**

**Depends on:**
**Blocks:**
Show dependency tree

**Reported:** 2007-02-16 18:12 EST by Linda Chan
**Modified:** 2007-06-26 21:26 EDT (History)
**CC List:** 0 users

**See Also:**

**Attachments**
Add an attachment (proposed patch, testcase, etc.)

Note
You need to log in before you can comment on or make changes to this bug.

Linda Chan 2007-02-16 18:12:25 EST Description

```
When opening a connection on the oda.flatfile data source, if the connection
properties provided does not include the HOME property, a NullPointerException is
encountered.
```

Linda Chan 2007-02-16 18:46:46 EST Comment 1

```
Added null value validation, which now throws an OdaException instead.
```

Linda Chan 2007-06-26 21:26:58 EDT Comment 2

```
Verified and closed.
```

*First Last Prev Next This bug is not in your last search results.* Format For Printing - XML - Clone This Bug - Top of page

Home | New | Browse | Search | Search [?] | Reports | Requests
| Help | Log In | Forgot Password | Terms of Use | Copyright Agent

Fig.1. Example of a typical bug report (bug #174533) from the Eclipse bug repository.

## 2.1.2 Source Code

A source code artifact is an executable specification of a software system's behaviour. It consists of a number of files written in one or more programming languages and grouped into logical entities called packages or modules. It is a mixed artifact that contains both structured (e.g., semantics, syntax) and unstructured (e.g., comments, identifiers) data. It also carries the information for communicating with both humans and compilers. The unstructured portion of source code has shown to help developers with various tasks at hand, e.g., program comprehension. However, creating a source code summary is often considered to be difficult because of the complex nature of source code. Fig.2 illustrates an example of source code taken from the NetBeans Wiki, "Can I dynamically change the contents of the System Filesystem at runtime?"[4]

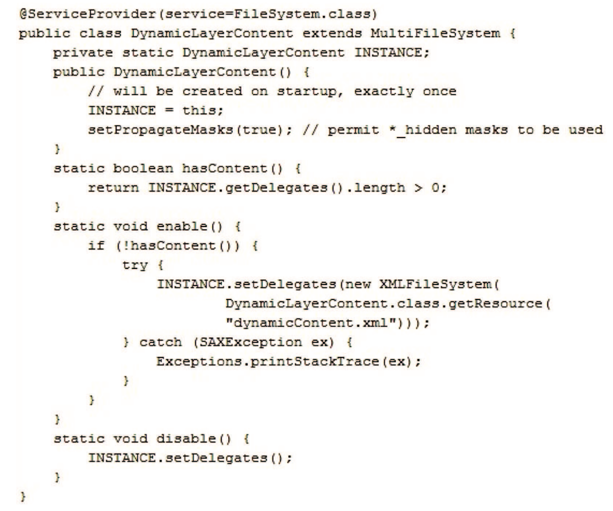

```
@ServiceProvider(service=FileSystem.class)
public class DynamicLayerContent extends MultiFileSystem {
    private static DynamicLayerContent INSTANCE;
    public DynamicLayerContent() {
        // will be created on startup, exactly once
        INSTANCE = this;
        setPropagateMasks(true); // permit *_hidden masks to be used
    }
    static boolean hasContent() {
        return INSTANCE.getDelegates().length > 0;
    }
    static void enable() {
        if (!hasContent()) {
            try {
                INSTANCE.setDelegates(new XMLFileSystem(
                        DynamicLayerContent.class.getResource(
                        "dynamicContent.xml")));
            } catch (SAXException ex) {
                Exceptions.printStackTrace(ex);
            }
        }
    }
    static void disable() {
        INSTANCE.setDelegates();
    }
}
```

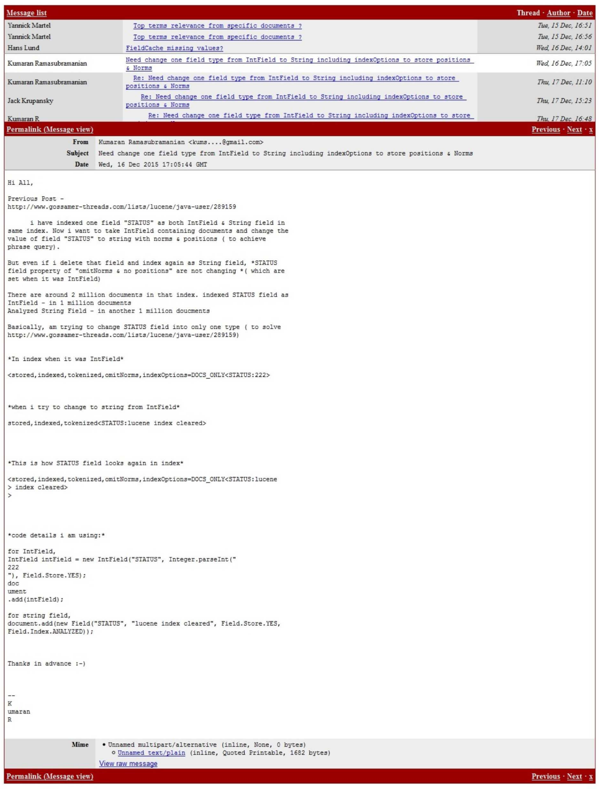
Fig.2. Example of a code fragment taken from the NetBeans Official FAQ.

## 2.1.3 Mailing Lists

Mailing lists usually constitute a set of time-stamped email messages. These messages consist of a header (that includes the sender, receiver(s), and time stamp), a message body (i.e., the text content of the email), and a set of attachments (additional documents sent with the email). Sometimes mailing lists aim to suggest different software engineering tasks such as the documentation of a source code. Bacchelli *et al.*[7] classified email contents into five levels, namely, text, junk, code fragment, patch, and stack trace. Apache mailing list[5] is a popular online mailing bundle for developers'

[1]bugzilla.mozilla.org/, Jan. 2016.
[2]www.atlassian.com/software/jira, Jan. 2016.
[3]bugs.eclipse.org/bugs/show_bug.cgi?id=174533, Jan. 2016.
[4]wiki.netbeans.org/DevFaqDynamicSystemFilesystem, Jan. 2016.
[5]wiki.apache.org/lucene-java/MailingListArchives, Jan. 2016.

emails. Fig.3 provides an example of email communications from Apache Lucene mail archive[6].



**Message list** Thread · Author · Date

| | | |
|---|---|---|
| Yannick Martel | Top terms relevance from specific documents ? | Tue, 15 Dec, 16:51 |
| Yannick Martel | Top terms relevance from specific documents ? | Tue, 15 Dec, 16:56 |
| Hans Lund | FieldCache missing values? | Wed, 16 Dec, 14:01 |
| Kumaran Ramasubramanian | Need change one field type from IntField to String including indexOptions to store positions & Norms | Wed, 16 Dec, 17:05 |
| Kumaran Ramasubramanian | Re: Need change one field type from IntField to String including indexOptions to store positions & Norms | Thu, 17 Dec, 11:10 |
| Jack Krupansky | Re: Need change one field type from IntField to String including indexOptions to store positions & Norms | Thu, 17 Dec, 15:23 |
| Kumaran R | Re: Need change one field type from IntField to String including indexOptions to store | Thu, 17 Dec, 16:48 |

**Permalink (Message view)** Previous · Next · x

| | |
|---|---|
| From | Kumaran Ramasubramanian <kums....@gmail.com> |
| Subject | Need change one field type from IntField to String including indexOptions to store positions & Norms |
| Date | Wed, 16 Dec 2015 17:05:44 GMT |

```
Hi All,

Previous Post -
http://www.gossamer-threads.com/lists/lucene/java-user/289159

      i have indexed one field "STATUS" as both IntField & String field in
same index. Now i want to take IntField containing documents and change the
value of field "STATUS" to string with norms & positions ( to achieve
phrase query).

But even if i delete that field and index again as String field, *STATUS
field property of "omitNorms & no positions" are not changing *( which are
set when it was IntField)

There are around 2 million documents in that index. indexed STATUS field as
IntField - in 1 million documents
Analyzed String Field - in another 1 million doucments

Basically, am trying to change STATUS field into only one type ( to solve
http://www.gossamer-threads.com/lists/lucene/java-user/289159)


*In index when it was IntField*

<stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<STATUS:222>



*when i try to change to string from IntField*

stored,indexed,tokenized<STATUS:lucene index cleared>




*This is how STATUS field looks again in index*

<stored,indexed,tokenized,omitNorms,indexOptions=DOCS_ONLY<STATUS:lucene
> index cleared>
>




*code details i am using:*

for IntField,
IntField intField = new IntField("STATUS", Integer.parseInt("
222
"), Field.Store.YES);
doc
ument
.add(intField);

for string field,
document.add(new Field("STATUS", "lucene index cleared", Field.Store.YES,
Field.Index.ANALYZED));



Thanks in advance :-)



--
K
umaran
R
```

Mime
- Unnamed multipart/alternative (inline, None, 0 bytes)
  - Unnamed text/plain (inline, Quoted Printable, 1682 bytes)

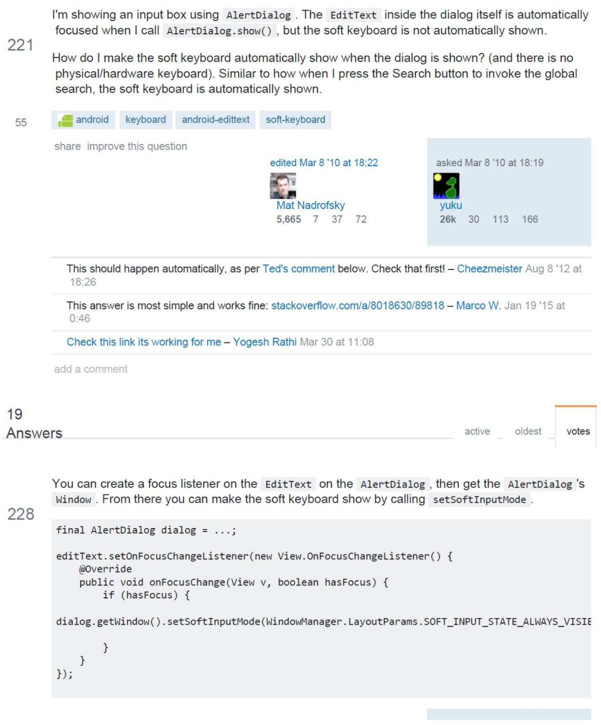
View raw message

**Permalink (Message view)** Previous · Next · x

Fig.3. Example of an email conversation from Apache Lucene mail archive.

### 2.1.4 *Developer Discussions*

Developer discussions (also called developer forums) are the online place to post questions and share comments with fellow engineers, developers, novice users, and general public that are related to the same specialized field. These forums are used to discuss a variety of development topics related to a given software, programming problem, feature request, or project management discussion[8]. These forums are heterogeneous artifacts that contain natural language text, code, XML configurations, images and many other things. Stack Overflow[7] and Apple Developers Forum[8] are popular developer forums. Fig.4 provides an example of developer communications on a heterogeneous artifact, i.e., StackOverflow[9].



221

I'm showing an input box using `AlertDialog`. The `EditText` inside the dialog itself is automatically focused when I call `AlertDialog.show()`, but the soft keyboard is not automatically shown.

How do I make the soft keyboard automatically show when the dialog is shown? (and there is no physical/hardware keyboard). Similar to how when I press the Search button to invoke the global search, the soft keyboard is automatically shown.

55

android keyboard android-edittext soft-keyboard

share improve this question

edited Mar 8 '10 at 18:22
Mat Nadrofsky
5,665 7 37 72

asked Mar 8 '10 at 18:19
yuku
26k 30 113 166

This should happen automatically, as per Ted's comment below. Check that first! – Cheezmeister Aug 8 '12 at 18:26

This answer is most simple and works fine: stackoverflow.com/a/8018630/89818 – Marco W. Jan 19 '15 at 0:46

Check this link its working for me – Yogesh Rathi Mar 30 at 11:08

add a comment

19 Answers

active oldest votes

228

You can create a focus listener on the `EditText` on the `AlertDialog`, then get the `AlertDialog`'s `Window`. From there you can make the soft keyboard show by calling `setSoftInputMode`.

```
final AlertDialog dialog = ...;

editText.setOnFocusChangeListener(new View.OnFocusChangeListener() {
    @Override
    public void onFocusChange(View v, boolean hasFocus) {
        if (hasFocus) {

dialog.getWindow().setSoftInputMode(WindowManager.LayoutParams.SOFT_INPUT_STATE_ALWAYS_VISIE

        }
    }
});
```

Fig.4. Example of a Stack Overflow discussion.

## 2.2 Typical Summarization Process

A typical summarization task can be fragmented into following steps, namely, corpus creation, normalization, experimentation based on either machine learning or data mining methods (sometimes it is called indexing), and evaluation of results. Generally, in a supervised learning approach, annotation process is required to compare and evaluate generated summaries. These steps are discussed in the following subsections.

### 2.2.1 *Corpus Creation and Normalization*

A corpus creation of selection is always the first step while building a summarization system. A corpus defines the collection of documents (a software artifact as in our case). It can be extracted at different granularities, e.g., a corpus may contain a set of classes or methods or fragments in the source code, or words, or sentences, or paragraphs in bug reports. The granularity should be decided up-front according to the needs of the required task as it influences the results of a given task significantly. After the corpus is created, a few op-

[6]http://mail-archives.apache.org/mod_mbox/lucene-java-user/201512.mbox/browser, Jan. 2016.
[7]http://stackoverflow.com/, Jan. 2016.
[8]developer.apple.com/devforums/, Jan. 2016.
[9]This example is taken from the following link: stackoverflow.com/questions/2403632/, Jan. 2016.

tional steps, often known as pre-processing steps, i.e., tokenization, stop word removal or filtering, and stemming are applied to reduce noise. For example, in the source code, programming language keywords (if, else) and character related syntax ("&&") are removed from the source code.

#### *2.2.2 Experimentation and Summary*

After the corpus is created and normalized, different techniques are applied to produce results (also known as indexing). These techniques could be machine learning based such as supervised or unsupervised learning or data mining based such as IR (information retrieval) and NLP (natural language processing). Existing techniques are explained with more details in Subsection 2.3.

Regarding the output, a summary can be extractive, i.e., a subset of sentences are selected to build a summary, or abstractive when it substitutes an original text with a new vocabulary. It is also possible to distinguish summaries as generic, informative, indicative, and lead. The generic summaries can serve as the surrogate of the original text as they may try to represent all relevant facts of a source text[1]. The indicative summaries are used to indicate what topics are addressed in the source text; they can give a brief idea of what original text is about. The informative summaries are intended to cover the topics in the source text and provide more detailed information. Lead summaries are based on the notion that the first term or terms that appear in the main document, i.e., leading terms, are the most relevant to that document[9]. As abstractive summaries are difficult to generate, most systems attempt to generate either indicative or informative summaries, or the combination of different types of summaries in an extractive manner.

#### *2.2.3 Evaluation of Summaries*

Evaluation methods are employed to verify the quality of the effectiveness of generated summaries. Annotation process is also applied to evaluate summaries. Annotation is a manual or automated process applied separately from the experimentation on the selected corpus for summary comparison or evaluation purposes. On a given corpus, human annotators are hired to select candidate sentences for summary creation. The likelihood of a sentence to belong to a summary is the score of the sentence[10]. For each sentence, the score is 0 when it has not been selected by any annotator, and maximum, when all annotators have selected sentences as candidate sentences. Generally, the set of sentences with a score 2 or more, i.e., positive sentences, or the manually created summaries by humans, are known as the gold standard summary (GSS).

### 2.3 Existing Approaches for Software Artifact Summarization

Here we intend to investigate the state-of-the-art studies in summarizing bug reports, source code, mailing lists, and developer discussions. The mining approaches contain studies based on information retrieval, natural language processing, stereotype identification, and program analysis methods. The machine learning techniques contain supervised, unsupervised and semi-supervised learning. Bug report summarization studies have employed machine learning based techniques while the other three artifacts have utilized both data mining and machine learning based techniques, or the combination of both approaches. In the following subsections, we discuss the state-of-the-art studies related to software artifact summarization based on aforementioned techniques.

#### *2.3.1 Information Retrieval Based Studies*

The information retrieval (IR) approaches aim to find materials of structured or unstructured nature, which satisfy the information need among large collections[11]. Previously, IR approaches have been applied to many software engineering problems such as traceability link recovery, program comprehension (summary) and software reuse[12]. IR techniques generally employ the methods based on vector space model (VSM), latent semantic indexing (LSI), latent Dirichlet indexing (LDI) or simple tf-idf (term frequency-inverse document frequency) methods. Several studies in software artifact summarization have employed IR methods. Here, we briefly survey these studies.

VSM represents the query and the artifact in the corpus as terms or term vectors[13]. Researchers have proposed many forms of weighting in VSM, but the most widely recognized weighting method is called tf-idf. LSI uses singular value decomposition (SVD) to identify patterns between terms and concepts. It can additionally extract the conceptual content of a text.

Haiduc *et al.*[9,14] applied VSM and LSI models to generate term-based summaries for classes and methods. They investigated the suitability of several techniques based on text retrieval methods to capture source code semantics in a way developers understand. They further evaluated the impact of their technique on the quality of summaries via a study of four developers. Using the same approach, Moreno and Aponte[15] discovered that the length of term-based summaries should range from 10 to 20 words, 10 words for methods and 20 words for classes, to get the real gist of source code. Similarly, Rodeghero *et al.*[16-17] utilized the eye movements and gaze fixations of programmers to identify keywords and built a tool on these findings using IR methods.

Rastkar and Murphy[18] presented a multi-document summarization technique based on IR methods to describe the motivation behind code change. In a pilot study, Binkley *et al.*[19] developed a tool for generating task-driven summaries using VSM and suggested that such summaries contain different levels of details. Bacchelli *et al.*[13] established benchmarks for finding traceability links between emails and source code. This benchmark was based on LSI and VSM models. They further manually inspected statistically significant emails for six unrelated software. Panichella *et al.*[20] employed VSM as an IR approach to extract method signatures' descriptions from bug reports and mailing lists. They assumed that there was an explicit traceability link between a source code and an email, and the extracted method descriptions could help developers understand and redocument the source code.

Latent Dirichlet allocation (LDA)[21] is an IR model that fits a generative probabilistic model from the term occurrences in a corpus of documents[22]. Panichella *et al.*[22] proposed a novel approach based on LDA — GA (genetic algorithm) to adapt, configure, and achieve acceptable performance across various software engineering tasks, especially source code summarization. In a couple of more studies, De Lucia *et al.*[23-24] investigated the overlap between automatic and human generated summaries — "to what extent an IR-based source code labeling would identify relevant words in the source code, compared to the words a human would manually select during a program comprehension task". They analyzed their technique based on VSMs, LSI, LDA, and a simple heuristic[10] — overlapped with those identified by humans, in the end.

Vassallo *et al.*[25] proposed an IR-based approach, CODES, that maps developer discussions on Stack-Overflow to source code segments in order to generate summaries of Java systems. They further developed an Eclipse plugin and tested results on Lucene[11] and Hibernate[12] systems. Similarly, Rahman *et al.*[26] proposed a mining approach, which recommends insightful comments about the quality, deficiency, and scope to improve the source code using LDA. They developed a recommender-heuristic based technique and utilized the crowdsourcing knowledge from Stack Overflow discussions for constructing code comments.

Hierarchial PAM (hPAM) is a topic modelling technique, which is built on LDA. It connects words and topics with a directed graph in a bag-of-words representation. It is extended over the PAM (Pachinko Allocation Model) and associated with a distribution of vocabulary. Eddy *et al.*[4] replicated the same work proposed earlier by Haiduc *et al.*[9] and expanded it using the HPAM topic modeling algorithm. They further evaluated and compared the impact of summaries using 14 developers. They found that the VSM technique was simpler and more efficient than HPAM for extracting source code keywords.

### 2.3.2 *Natural Language Processing Based Studies*

NLP, in general, deals with the natural language generation of text through artificial intelligence and computer linguistics. In this subsection, we discuss studies that have adopted NLP methods in constructing textual summaries.

Sridhara *et al.*[3,27] first employed NLP methods to generate automatic summary comments for Java methods. Given the signature and body of a method, their automatic comment generator identifies the content for the summary and generates descriptive comments that outline the method's overall actions. It also provides the high-level overview of the role of a parameter in achieving the computational intent of the method. In another effort[28], the authors presented an automatic technique for identifying code fragments that implement high-level abstractions of actions and expressing them as a natural language description. They identified code fragments that implement high-level actions. A high-level action means a high-level abstract algorithmic step of a method. Rastkar *et al.*[29-30] generated light abstractive natural language summaries of source code concern — what it is and how it is implemented.

[10]Picking terms from class, attribute, and method names only.

[11]lucene.apache.org/, Jan. 2016.

[12]hibernate.org/, Jan. 2016.

They used RDF graph[13], Verb-DO[14], and SEON java ontology[15] for this purpose.

In the same way, Moreno *et al.*[31-32] produced natural language summaries by incorporating NLP and class stereotypes for Java classes. Their summaries focus on class responsibilities rather than on class relationships. They further extended this study and generated an Eclipse plugin called JSummarizer. McBurney and McMillan[33-35] applied PageRank algorithm along with SWUM (Software Word Usage Model) and natural language generation system to extract the contextual meanings hidden behind these classes by measuring the importance of classes and methods. They argued that existing documentation generators would be more effective if they included information from the context of the methods. In a different study[36], they presented a topic modeling approach based on HDTM (Hierarchical Document-Topic Model) algorithm, which selects keywords and topics as summaries for source code. They further organized the topics in source code into a hierarchy, with more general topics near the top of the hierarchy, thus, presenting software's highest-level functionality before lower-level details.

In another effort Moreno *et al.*[37] introduced an automatic release generator, ARENA (Automatic RElease Notes GenerAtor) that extracts changes from the source code, summarizes and integrates them with the information from release trackers. Kulkarni and Varma[38] used natural language lexicons to generate source code summaries. Wong *et al.*[39] generated a tool, AutoComment, aiming at mapping code fragments to their descriptions in developer communications, relying on the clone detection between source code and code fragment contained in the discussion, and then automatically generating comments using NLP.

Zhang and Hou[40], using natural language processing and sentiment analysis techniques applied on online forums, investigated how to extract problematic API features, i.e., features that cause difficulties for API users and often are discussed in a forum. In particular, they extracted phrases from online threads and realized that meaningful problematic features mostly appeared in the phrases that contain negative sentences or the neighbors of negative sentences. Meaningful features are API features that help support teams finding out the problems users have. It also helps support teams in improving the API contents effectively. As an extension to Kamimura and Murphy's work[41] (discussed in Subsection 2.3.4), Panichella *et al.*[42] generated unit test case summaries using NLP methods, and evaluated their performance as well as their impact on bug fixing. Similarly, Li *et al.*[43] used NLP with static analysis to automatically document unit test case and generated a tool UnitTestScribe.

### 2.3.3 Stereotype Identification Based Studies

Stereotypes are simple abstractions of a class' or method's roles and responsibilities in a system's design, e.g., an accessor is a method stereotype that returns information[44].

We found four studies[31-32,45-46] that employ stereotype identification in conjunction with different heuristics to produce human readable summaries for source code artifacts. As described in Subsection 2.3.2, Moreno *et al.*[31-32] focused on summaries regarding class contents and responsibilities, rather than the relationships using stereotypes. Cortés-Coy *et al.*[46] presented an approach designed to generate commit messages automatically from code change sets. They identified method stereotypes developed by Moreno and Marcus[47] and inserted them in pre-defined templates to generate commit messages. Abid *et al.*[45] generated template-based summaries[16] of C++ methods using stereotype identification, where required roles (stereotype) are extracted from the method's signature and inserted into the sentence with a pre-defined format.

### 2.3.4 Program Analysis Based Studies

Nielson[49] defined program analysis as a process of automatically analyzing (statically or dynamically) the behaviour of computer programs in the context of properties such as correctness and robustness. Program analysis focuses on two major areas: program optimization and program correctness. The former focuses on improving the program's performance while reducing the resource usage. The latter focuses on ensuring what the program does and what it is supposed to do.

Buse and Weimer[48] designed an automatic technique, DeltaDoc, to describe source code modifications using symbolic execution and summarization tech-

[13]Resource Description Graph. It represents triple (subject, predicate, and object) in the form of a graph.

[14]Verb-DO makes direct object verb based pairs for a given artifact (a class, method, or document).

[15]SEON is a Java ontology schema which represents facts by subject, predicate and object triples.

[16]www.sdml.info/method_summarization/, Jan. 2016.

niques. The documentation describes the effect of a change in the runtime behaviour of a program, including the conditions under which program behaviour changed and what the new behaviour was. It summarizes the runtime conditions necessary for control flow to reach the changed statements, the effect of the change on functional behaviour and program state, and what the program used to do under these conditions. At a high-level it produces structured and hierarchical documentation of the form: *When calling A(),If X,do Y Instead of Z*.

Kamimura and Murphy[41] generated template-based unit test cases summaries using static program analysis. They determined how unique a particular method invocation was relative to other test cases, and identified the focus of the test case. It helps in improving human's ability to quickly comprehend unit test cases so that appropriate decisions could be made about where to place effort when dealing with large unit test suites.

Table 1 lists studies that employ IR, NLP, SI and PA based methods.

**Table 1**. Summary of Studies Utilizing IR, NLP, SI and PA Techniques

| Author | Artifact | Method | Corpus | Summary Type |
|---|---|---|---|---|
| Haiduc *et al.*[9,14] | C + M | IR → VSM, LSI | ATunes, Art of Illusions | Lead, extractive, light abstractive |
| Rastkar *et al.*[29-30] | CC | PA → RDF, SEON + NLP → Verb-Do | JHotDraw, Drupal, Jex, JFreeChart | Abstractive |
| Sridhara *et al.*[3,27] | M | NLP → AST, CFG, SWUM | Megamek, SweetHome3D, JHotDraw, Jajuk, JBidWatcher | Abstractive |
| Eddy *et al.*[4] | C + M | IR → VSM, LSI, hPAM | ATunes, Art of Illusions | Extractive |
| Moreno and Aponte[15] | C + M | IR → VSM, LSI | ATunes | Extractive, abstractive |
| Moreno *et al.*[31-32] | C | NLP + SI | ATunes, AgroUML | Indicative, abstractive, generic |
| Buse and Weimer[48] | CC | PA | FreeCol, jFreeChart, iText, Phex, Jabref | Abstractive |
| Rastkar and Murphy[18] | CC | IR → SVM | Connent | Extractive |
| Binkley *et al.*[19] | C + M | IR → VSM | JEdit | Abstractive |
| McBurney and McMillan[33-35] | M | IR → PR, CG + NLP → SWUM, Verb-DO | NanoXML, Siena, JTopas, Jedit, Jajuk, JHotDraw | Abstractive |
| McBurney *et al.*[36] | M | NLP → HTDM | NanoXML | Extractive |
| Cortés-Coy *et al.*[46] | CC | SI + NLP | Elastic Search, Spring Social, JFreeChart, Apache Solr, Apache Felix, and Retrot | |
| Panichella *et al.*[20] | M + BR + E | IR → VSM | Lucene, Eclipse | |
| Kamimura and Murphy[41] | UT | PA | Junit, JFreeChart, CodePro | Abstractive |
| Kulkarni and Varma[38] | M | IR → VSM + NLP → lexicons, clues | JEdit | Extractive |
| Moreno *et al.*[37] | C + CC | NLP + PA | 1 000 release notes from 58 projects | Abstractive |
| Rodeghero *et al.*[16-17] | M | IR → VSM + manual summaries | NanoXML, Siena, JTopas, Jajuk, JEdit, and Jhotdraw | Extractive |
| Abid *et al.*[45] | M | SI | HippoDraw | Light abstractive |
| Panichella *et al.*[22] | C | IR → LDA-GA | JHotDraw, exVantage | Extractive |
| Lucia *et al.*[23-24] | C | IR → VSM, LSI, LDA | JHotDraw, eXVantage | Extractive |
| Vassallo *et al.*[25] | M + DD | IR → VSM | Lucene, Hibernate | |
| Rahman *et al.*[26] | C + DD | IR → LDA, PageRank | Stack OverFlow discussions and Comments | |
| Wong *et al.*[39] | C + DD | NLP + clone detection | 23 projects from Stack Overflow discussions | |
| Zhang and Hou[40] | DD | NLP + PA | Oracle Java API | Extractive |
| Sridhara *et al.*[28] | M + CF | NLP → AST, CFG, SWUM | Freecol, GanttProject, HsqlDB, Jajuk, JBidwatcher, JHotDraw, PlanetaMessenger, SweetHome3D | Abstractive |
| Panichella *et al.*[42] | UT | NLP → SWUM | Math4J, Commons Primitives | Light abstractive |

Note: C stands for class, M for method, CC for code change, UT for unit test, CF for code fragment, DD for developer discussion, VSM for vector space model, LSI for latent semantic indexing, NLP for natural language processing, PA for program analysis and SI for stereotype identification.

### 2.3.5 *Machine Learning Based Approaches*

Machine learning based studies can be categorized into supervised, unsupervised and semi-supervised learning. In supervised summarization, the task of selecting important sentences can be represented as a binary classification problem, where it classifies all sentences into summary and non-summary sentences[10]. A corpus of human annotation of sentences is used to train a statistical classifier, where sentences are represented based on the importance[10]. Kupiec *et al.*[50] first proposed the use of supervised machine learning for summarization. They argued that the supervised learning approach provides freedom to use and combine any desired number of features. A supervised approach relies on the availability of a summary corpus, which is trained and predicted using predefined procedures. However, building corpus, which is usually large, and annotated data require substantial amount of manual effort[51].

Rastkar *et al.*[52] developed a supervised learning classifier based on the pre-existing technique[53] for producing extractive summaries of bug reports. They trained a classifier, Bug Report Classifier (BRC), on a manually created reference set called Gold Set. They selected 36 bug reports from four open source projects, Mozilla[17], Eclipse[18], KDE[19] and Gnome[20]. The authors[5] further performed a task based analysis for evaluating how summaries helped in detecting duplicate bugs without degrading the accuracy.

Jiang *et al.*[54] targeted mining authorization characteristics using bug report summaries. They employed byte level $N$-Grams to find the similarity between Normalized Simplified Profile Interactions (NSPI) and utilized it for generating summaries of bug reports. Ying and Robillard[55] developed a supervised source to source summarization of code fragments. They extracted 49 syntactic and 5 query features from the 70 code fragments, which were taken from the Eclipse official FAQ[21]. By training and testing the data on these features, they developed a support vector machine (SVM) and Naive Bayes (NB) models to generate summaries. Recently, Nazar *et al.*[56] incorporated crowdsourcing on a smaller scale with SVM and NB in a supervised manner. They achieved the precision of 82% and outperformed the existing classifiers[55] as well.

Petrosyan *et al.*[57] investigated the use of text classification to discover tutorial sections explaining how to use a given API type using supervised learning. They considered that a tutorial section explains an API type if it would help a reader unfamiliar with the corresponding API to decide when or how to use the API type to complete a programming task. They further considered API types (classes and interfaces) as the best level of granularity for finding usage information because a single section of an API tutorial usually describes a solution for a programming task by using a set of methods.

Contrary to supervised techniques, unsupervised techniques do not require manually annotated corpus, and training and test sets, and they focus on finding hidden data in an unlabeled dataset. In an attempt to produce unsupervised summaries of bug reports, Mani *et al.*[58] evaluated the quality of summaries generated by four well known unsupervised classifiers, namely, Maximum Marginal Relevance (MMR), Grasshopper, DivRank and Centroid. They introduced a heuristic based noise reducer for bug sentences on a corpus of 19 bug reports from IBM DB2 Bind that automatically classifies sentences into questions, investigations, and code snippets. When using the noise reducer, they found that each of the four unsupervised algorithms produced a summary of slightly better quality if compared with the supervised approach[52].

Similarly, Lotufo *et al.*[51,59], proposed heuristic and graph based unsupervised bug report summarization technique. They posed three hypothesis questions which were “sentence relevance”, “frequency discussion topics” and “bug report’s titles”. Furthermore, they compared results with the existing corpus[52] and found 12% improvement. They applied graph-based methods such as Markov Chain and PageRank for summarizing bug reports. Next, Yeasmin *et al.*[60] proposed a prototype that assists developers to review a project’s bug report by interactively visualizing insightful information regarding bug reports. They applied the unsupervised topic modeling techniques to visualize bug reports. They found that their approach outperformed existing unsupervised techniques statistically.

[17] www.mozilla.org, Jan. 2016.

[18] www.eclipse.org, Jan. 2016.

[19] www.kde.org, Jan. 2016.

[20] www.gnome.org, Jan. 2016.

[21] http://wiki.eclipse.org/index.php/Eclipse, Jan. 2016.

Fowkes *et al.*[61] presented a novel unsupervised approach called TASSAL (Tree-Based Autofolding Software Summarization ALgorithm), for summarizing source code using autofolding. Autofolding is automatically creating a code summary by folding non-essential code blocks in a source code. Autofolding was presented as a subtree optimization problem making use of a scoped topic model. They showed that their formulation outperformed simpler baselines at meeting human judgments, yielding a 77% accuracy and a 28% error reduction. In another effort, Sorbo *et al.*[8] developed a semi-supervised email content analyzer tool, DECA, that uses captured language patterns from email conversations to generate source code summaries. Their semi-supervised approach was based on island parsing and hidden Markov models. Table 2 provides the list of studies that employ supervised, unsupervised and semi-supervised machine learning methods.

Other efforts, e.g., [13, 62] have discussed summarization techniques to address different problems such as establishing traceability link recovery between different artifacts such as emails and source code or developer discussion and source code. McBurney and McMillan[33] conducted an empirical study examining the summaries written by authors or readers and automatic summaries generated by summarization tools. They used different metrics to examine this hypothesis. Fritz *et al.*[63] and Kevic *et al.*[64] utilized eye-movement tools and human developers to conduct empirical studies for developing models in the context of change task. In the same way, Ying and Robillard[65] employed think-aloud verbalization to find decisions of participants in order to elicit a list of practices for summarization task. These and other applications of software artifact summarization are thoroughly discussed in Section 3.

**Table 2**. Summary of Studies Utilizing Supervised Learning, Unsupervised Learning and Semi-Supervised Learning Approaches (from Jan. 2010 to Apr. 2016)

| Author | Artifact | Method | Corpus | Summary Type |
|---|---|---|---|---|
| Rastkar *et al.*[5,52] | BR | SL → VSM | 36 bug reports from Eclipse, Mozilla, KDE, and Gnome | Extractive |
| Mani *et al.*[58] | BR | UL → MMR, Centroid, GH, DR | 36 bug reports from Study[52] + 19 from IBM DB2-Bind | Extractive |
| Lotufo *et al.*[51,59] | BR | UL → Markov Chain, PageRank, VSM | 55 bug reports from Chrome, Launchpad, Mozilla and Debian | Extractive |
| Ying and Robillard[55] | CF | SL → SVM, NB | 70 code fragments from Eclipse Official FAQs | Extractive |
| Jiang *et al.*[54] | BR | SL → VSM, N-Grams | 96 bug reports from Eclipse, Mozilla, KDE, and Gnome | Extractive |
| Yeasmin *et al.*[60] | BR | UL → VSM, LDA, PageRank | 3 914 bug reports from Eclipse ANT | Extractive |
| Fowkes *et al.*[61] | C | UL → TASSAL | Storm, easticsearch, spring-framework, libgdx, bigbluebutton, netty from Github | Extractive |
| Nazar *et al.*[56] | CF | SL → SVM, NB | 78 code fragments from Eclipse Official FAQs + 49 from Netbeans Official FAQs | Extractive |
| Sorbo *et al.*[8] | E + M | SSL → Hidden Markov | Eclipse, Lucene | Extractive |
| Petrosyan *et al.*[57] | C + I + OF | SL → MaxEnt Classifier | Joda Time, Apache Commons, Java Collections API, Smack API | Extractive |

Note: BR stands for bug report, C for class, M for method, I for interface, E for email, CF for code fragment and OF for online forum.

## 3 Applications

In this section, we review and classify selected studies in context of the purpose of generating summary. What summary is intended for and what is used to achieve? In the following subsections, we discuss some important applications of software artifacts summarization with respect to the selected studies.

### 3.1 Code Change

A developer working as part of a software development team often needs to understand the reason behind a code change. This information is important when multiple developers work on the same code and the changes they individually make may cause a conflict[18]. Such collisions can be avoided if a developer working on the code knows why the particular code is added, deleted or modified. Often, a developer can access a commit message or a bug report with some information on the code change. However, this information often does not provide the exact context of code changes, for example which business objective or feature is being implemented. Automatic summary generation of a code change is a one way to overcome this problem.

Rastkar and Murphy[18] proposed the use of multi-document summarization technique to generate concise descriptions of the motivation behind the code change based on the information present in the related documents. They identified a set of sentence-level features to locate the most relevant sentences in a change set to be included in a summary.

Cortés-Coy *et al.*[46] presented an approach, coined as ChangeScribe[22], which was designed to generate automatic commit messages from change sets. It generated natural language commit messages by taking into account commit stereotype, the type of changes (e.g., files rename, changes done only to property files), as well as the impact set of the underlying changes.

Similarly, Buse and Weimer[48] designed an automatic technique, DeltaDoc, to describe source code modifications using symbolic execution and summarization techniques. DeltaDoc generates textual descriptions of code changes, but when the changeset was very large (i.e., many files or methods), it describes each method separately ignoring possible dependencies of those methods. Primarily, it reduces the human effort in documenting program changes.

## 3.2 Duplicate Bug Detection

If two bugs are same, or they have different reproducible steps with same results, or their cause/reason of defect is same, then such bugs are called duplicate bugs. A common activity when using a bug repository is to detect duplicate bug reports stored in the bug repository. It helps programmers in fixing defects and other bug triage tasks. However, it is quite challenging to manually detect duplicate bug reports since there can be a large number of bug reports in a bug repository. Bug report duplicate detection is performed when a new bug report is filed against a bug repository and has to be triaged. Early determination of duplicate bugs can add information about the context of a problem and can ensure that the same problem does not end up with being assigned to multiple developers for resolution. Developers use different techniques to retrieve a list of potential duplicates from the bug repository, including their memory of bugs they know about in the repository and keyword searches[67-68].

Rastkar *et al.*[5] evaluated the summaries produced earlier in their study[52] for duplicate detection. They evaluated if summaries could help in determining the duplicate bug during bug triage. They hypothesized that the concise summaries of original bug reports could help developers save time in performing duplicate detection tasks without compromising accuracy.

## 3.3 Bug Report Digestion

Bug digestion means the information developers are looking for from an artifact can easily be looked for. Bug reports are not created with this intent in mind and rather they are the result of the collaboration between different reporters, developers, and users' communications. Therefore, it is uneasy to read and comprehend bug reports. Based on this information, Lotufo *et al.*[51,59] proposed a PageRank based summarization technique that focuses on if the bug reports could be digestible, providing a deeper understanding of the information exchange in bug reports and utilizing that information for generating general purpose bug report summarizer. Using 52 bug reports from Chrome, Launchpad[23], Mozilla, and Debian[24] open source projects, they evaluated results statistically and qualitatively, achieving 12% improvement over state-of-the-art supervised summarizers.

## 3.4 Summary Visualization

Visualization is the use of a computer-based interactive visual representation of data to amplify cognition. Charts, graphs, and maps are some examples of visualized data. Only one effort in software artifact summarization, proposed by Yeasmin *et al.*[60], produced an interactive visualization of bug reports using extractive summaries and their relationship between different bug reports. One of the main objectives of their proposed visualization prototype was to provide insightful information to developers through software bug reports over time. Their visualization prototype generates and shows the topic evolution of each topic automatically, retrieves all software bug reports associated with a given topic along with their bug report IDs and titles, provides searching option by keywords associated with a topic, and visualizes an extractive summary of each bug report. Furthermore, they produced a set of keyword clouds layout to show the evolution of summary contents over time. Visualizing software artifact information could be a promising research area in future.

[22]www.cs.wm.edu/semeru/data/SCAM14-ChangeScribe/, Jan. 2016.

[23]launchpad.net/, Jan. 2016.

[24]www.debian.org/, Jan. 2016.

### 3.5 Traceability Link Recovery

Establishing a link between two artifacts is a nontrivial problem especially when the artifacts are communicative, e.g., emails and source code or bug reports and source code. Bacchelli *et al.*[13] devised a set of lightweight methods to establish the link between emails and source code. They further defined the benchmark for recovering traceability links and its utilization for summarization. Similarly, Aponte and Marcus[62] utilized text summarization techniques to address traceability link recovery problem. They conducted a pilot study that proposed a hybrid summarization technique that combines textual and structural information to summarize source code. They proposed generating concise descriptions of short methods (ten lines) and offered developers with the tool providing candidate link analysis. These concise summaries could help developers in making proper decisions.

### 3.6 Document Generation

Programmers rely on documentation to understand the source code or an API or a user manual. Document generator is a tool that generates software documentation such as JavaDocs[25]. It can be a complete document that precisely summarizes the contents of an artifact or a template that requires a set of words to be inserted at appropriate places in pre-defined sentences, intending to make meaningful sentences. We have found four studies that targeted generating documents for source code summaries.

Kamimura and Murphy[41] generated textual summaries of test cases using pre-defined templates. First, all operation invocations were described using a *"calls <methodname> on <objectname>"* format. Next, they outputted all verification invocations using templates for various verification operations, such as "*checks the <methodname> of <object> is equal to <value>*". Similarly, Abid *et al.*[45] constructed pre-defined templates for summarizing C++ methods using stereotype identification and static analysis. Their short method descriptions follow *<method> is a <stereotype>*[*that collaborates with the <obj>*] format. The text in "[ ]" is a pre-defined text while the text in "<>" is identified through stereotype identification and inserted in the sentence.

McBurney and McMillan[33] developed a documentation generation technique, which focuses on generating documentation for a given set of source code using NLP methods. They argued that summarization techniques mainly stitched together words to generate summaries, but lacked the real context of a source code. Thus, they presented a page rank based technique that generates a summary document, same as the official JavaDoc, but more contextual in nature. Therefore, the summary is presented in a way that readers can understand it[69]. Recently, Sorbo *et al.*[8] developed an intention mining tool that captures linguistic pattern from emails to define email contents and utilized it for re-documenting Lucene and Eclipse source code.

### 3.7 Source-to-Source Summaries

Source-to-source summarization deals with selecting important code lines from the given code fragment. A code fragment is a partial program that serves the purpose of demonstrating the usage of an API[55]. The code fragment summary is a shorter code fragment consisting of informative lines only in the context of a given task. According to Ying and Robillard[55] current research in software engineering has mostly focused on the retrieval accuracy aspect of code fragments, or developing a tool for summarization, or devising new methods for generating summaries, but little on the presentation aspect of code examples, e.g., how code examples should be presented in a result page? Source-to-source summarization is one way to improve the presentation of code examples.

Ying and Robillard[55] conducted two studies related to source code fragment summarization and their practices. In the first study[55] they used SVM and NB classifiers to find the feasibility of generating source-to-source summaries of code fragments. They extracted syntactic and query features, and trained them to achieve desirable task. In another study[65] they conducted experiments to discover how and why source code could be summarized. They elicited a list of practices, for example none of the participants exclusively extracted code verbatim for the summaries, motivating abstractive summarization. Their results provided a grounded basis for the development of code example summarization and presentation technology. Later, Nazar *et al.*[56] generated source-to-source summaries of code fragments by utilizing crowd enlistment on a small scale to extract code features. They found that their approach produced statistically better summaries than the existing study[55]. From generated results they

[25]www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html, Jan. 2016.

inferred that the query based features might statistically degrade the accuracy of extracting informative lines from the code fragment.

## 3.8 Tool Development

In order to facilitate developers, researchers targeted building summarization tools. Eight studies from surveyed papers explicitly developed summarization tools.

Rodeghero *et al.*[16-17] built a novel summarization tool based on the eye-tracking (eye movement and gaze features) results obtained from humans (developers). They argued that little evidence existed about the statements and keywords that the programmers deemed important while summarizing source code. Therefore, they conducted an eye-tracking study of 10 developers, in which they read java source code and wrote summaries. Later, they applied the findings in building a novel summarization tool. Kevic *et al.*[64] developed an eclipse plugin iTrace for measuring eye interactions by combining user interaction monitoring with very fine granular eye-tracking data that is automatically linked to the underlying source code entities in the IDE.

Later, McBurney and McMillan[33] developed a source code summarization tool, Sumslice[26] that focuses primarily on the interaction of methods. Method interaction intends to describe how methods communicate with each other — the dependencies of the method, and any other method. It uses SWUM to identify keywords and parts of speech messages. SWUM is a software word usage model introduced by Hill *et al.*[70] and it is used for representing program statements as sets of nouns, verbs, and prepositional phrases. SWUM works by making assumptions about different Java naming conventions and using these assumptions to interpret different program statements. In another effort[34], they developed a tool Sumalyze for detecting the similarity between the source code summaries and the source code.

JSummarizer[31] is an Eclipse plug-in for automatically generating natural language summaries of Java classes. It automatically infers design intents of classes through stereotype identification. Stereotypes are simple abstractions of a class's role and responsibility in a system's design[44]. Accessors (getter methods), mutators (setter methods), constructors, and collaborators are some of the examples of method stereotypes.

In other efforts, Aponte and Marcus[62] built a tool for the candidate analysis of source code, Vassallo *et al.*[25] and Cortés-Coy *et al.*[46] built eclipse plugins CODES and ChangeScribe for mining source code from developer discussions and generating commit messages respectively. Sorbo *et al.*[8] developed a tool for the intention mining of emails and utilized it for source code documentation.

## 3.9 Authorship Characteristics

Jiang *et al.*[54] leveraged the authorship characteristics to facilitate the well-known task in software maintenance, i.e., bug report summarization. They presented a framework called Authorship Characteristics Based Summarization (ACS). The reasoning of ACS is that, given a new bug report initialized by contributor $A$, a classifier trained over annotated bug reports by $A$ is highly likely to perform better than a classifier trained over annotated bug reports by other contributors.

## 3.10 Eye-Tracking Interactions

Rodeghero *et al.*[16-17] first conducted an eye-tracking study, focusing on eye movements and gaze features of 10 software developers. They asked developers to read Java source code and wrote summaries. Their eye and gaze movements were recorded to gain insight about what methods or classes they felt were important. They further extended it to build a novel tool and compared the results with human summaries.

Kevic *et al.*[64] argued that most empirical studies on change tasks were limited to small snippets or they had small granularity. Therefore, they hired 12 professional and 10 student developers to perform three change tasks and found that developers only looked at a few lines of methods — related to the data flow of variables in the method. Tracing developers' interactions may help in developing improved tools for software summarization task. Similarly, Fritz *et al.*[63] through an exploratory study with 12 developers completing change tasks in three open source systems, identified important characteristics of these context models and how they were created.

## 3.11 Task-Based Summarization

A preliminary investigation regarding usefulness and goodness of summaries, specific to a given task, was conducted by Binkley *et al.*[19] They emphasized on task specific manual summaries that help in developing

[26]www3.nd.edu/~pmcburne/summaries/, Jan. 2016.

automatic task specific summarization tools. The participants manually produced two different summaries of few classes, one aiming at reuse and the other aiming at testing. The study suggested that such summaries contained different levels of details. This study was conducted at 2013 ICSE workshop NaturaLiSE[27] and lessons learned from investigations supported the notion that the task played a significant role, and thus should be considered by researchers for building and accessing automatic software summarization tools (Subsection 2.3.1).

Similarly, McBurney and McMillan[34] conducted an empirical study, aimed at measuring the similarity between source code and source code summaries. They focused on the usefulness of summaries in a task-driven manner using a tool Sumalyze[28].

Other studies focused on autofolding[61], developer facilitation[5,52], summarizing developer activity[71] and method signature utilization from bug reports and mailing list[20]. More details are provided in the cited papers. Table 3 provides the list of studies with regard to the intended tasks (grouped).

**Table 3**. List of Studies Grouped with Task and Purpose (Applications)

| Category | Task | Author |
|---|---|---|
| Visualization | Summary visualization of bug reports | Yeasmin *et al.*[60] |
| Document generation | Template based source code summaries | Abid *et al.*[45] |
| | Natural Language based documentation | McBurney and McMillan[33-35] |
| Source to source | Code examples summaries | Ying and Robillard[55] |
| | How and why code examples can be summarized? | Ying and Robillard[65] |
| | Code example summaries with crowdsourcing based features | Nazar *et al.*[56] |
| Duplicate bug detection | Detecting duplicate bug reports using bug report summaries | Rastkar *et al.*[5] |
| Code change | Summary of a code change using multi-document summarization technique | Rastkar and Murphy[18] |
| | ChangeScribe: generates commit messages | Cortés-Coy *et al.*[46] |
| | DeltaDoc: textual description of code change | Buse and Weimer[48] |
| Tool development | Summarization tool based on eye-movements of developers | Rodeghero *et al.*[16-17] |
| | Eclipse plugin: JSummarizer | Moreno *et al.*[31] |
| | Sumslice: source code summarization tool | McBurney and McMillan[33-35] |
| | Sumalyze | McBurney and McMillan[69] |
| | Tool for candidate link analysis | Aponto and Marcus[62] |
| | Eclipse plugin: CODES-mining source code descriptions from developers discussions | Vassallo *et al.*[25] |
| | Eclipse plugin: iTrace-incorporating implicit eye tracking | Kevic *et al.*[64] |
| | Eclipse plugin: ChangeScribe | Cortés-Coy *et al.*[46] |
| | Framework for summarizing code concerns | Rastkar *et al.*[29-30] |
| Authorship characteristics | Leveraged the authorship characteristics to facilitate bug report summarization | Jiang *et al.*[54] |
| Bug report digestion | Readability, time reduction, extraction of important information from bug reports | Lotufo *et al.*[51,59] |
| Autofolding | Automatic autofolding of source code in an IDE | Fowkes *et al.*[61] |
| Eye-tracking interaction | Eye-tracking interactions to develop summarization tools | Rodeghero *et al.*[16-17] |
| | What are software developers doing during a change task? | Kevic *et al.*[64] |
| | Code context models | Fritz *et al.*[63] |
| Task-based summarization | Study on developing task-specific summaries | Binkley *et al.*[19] |
| | Similarity between source code and source code summaries | McBurney and McMillan[69] |

## 4 Tools

In this section we provide a brief overview of tools that are incorporated during summarizing software artifacts in selected studies. These tools are either produced as a result of summarization task or used for performing a summarization task. A summarization tool can facilitate a software developer in achieving the desired task quickly and effectively.

We only found one tool, BC3 annotation tool for annotating bug reports used by Rastkar *et al.*[52] This framework allows researchers to annotate emails or other conversations. It is developed at the University of British Columbia and based on Ruby on Rails and

[27]dibt.unimol.it/naturalise/, Jan. 2016.

[28]www3.nd.edu/∼pmcburne/sumalyze/, Jan. 2016.

MySQL database. Two classification tools, LibSVM[72] and LibLinear[73] are used for SVM and linear classifications in software artifact summarization. Ying and Robillard[55], Rastkar and Murphy[18], and Nazar *et al.*[56] employed LibSVM for the supervised classification of source code summaries. Whereas Rastkar *et al.*[5,52] employed LibLinear for the linear classification of bug reports.

Following Eclipse plugins, namely, Jex, JayFX, JStereotype, JSummarizer, ChangeScribe and iTrace were developed as summarization tools. Rastkar *et al.*[30] utilized JayFX, an Eclipse plug-in that extracts various relations (such as method calls) from a program. Jex is a tool for analyzing exception flow in Java programs for summarizing code concerns. Both tools are developed by the Department of Computer Science, University of British Columbia. Moreno *et al.*[32] at Wayne State University[29], developed two eclipse plugins namely JStereotype and JSummarizer, which were used for identifying code stereotypes and generating summaries respectively. ChangeScribe is an Eclipse plugin for generating commit messages. It is developed by the SEMERU group[30] at The College of William and Mary. UnitTestScribe is also developed by the same group for generating unit test case documents.

Sumslice is a document generation tool developed by McBurney and McMillan[33-35] at the University of Notre Dame. They developed another tool using Python[31] programming language, Sumalyze[69], for realizing the similarity between source code and source code summaries. iTrace[32] is an eye tracking plugin developed by Kevic *et al.*[64] at SERESL Lab Youngstown State University[33]. Stanford NLP Parser is utilized in the studies of Mani *et al.*[58], Panichella *et al.*[20] and Zhang and Hou[40]. It is developed by the Stanford University NLP Group[34] for parsing grammatical structures of sentences. Panichella *et al.*[42] developed a tool, TestDescriber, that generates user test case summaries.

Table 4 provides a list of tools regarding software artifact summarization, including their brief description, authors, and URLs for accessing them. We believe that these tools can be useful for future researchers, especially junior researchers who are interested in re-implementing existing studies or developing new or novel summarization systems. This list provides tools which are publicly available. Commercial tools are not listed in this table.

## 5 Evaluation Methods for Software Artifact Summarization

Evaluation of automatic summaries is not a straightforward process. It is difficult to judge the usefulness of summary in a given context as summarization is a subjective and non-deterministic process. There are different methods that can be taken into account for evaluating summaries[1]. In general, methods for evaluating summaries can be broadly classified into two categories: intrinsic or extrinsic[75]. Intrinsic refers to methods that evaluate the quality of the summary produced, usually through comparisons to a gold standard. This is in contrast to extrinsic evaluation where the evaluation measures the impact of the summary on task performance such as the task-based evaluations[76].

In existing studies, several methods have been applied for evaluating summaries. These methods focus on the amount of information, quality, context, and content factors reflected in the summaries. Evaluation also depends on the intention or purpose for which it is tested. For instance, Panichella *et al.*[20] evaluated summaries using humans with the intention of assessing the quality of summaries — qualitative evaluation. In the subsections below, we attempt to distinguish and provide the details of the types of summary evaluations employed by the authors of the selected studies.

### 5.1 Statistical Evaluation Methods

Statistical evaluation involves different information retrieval metrics such as precision, recall, $F$-measure, pyramid precision, ROC-AUROC curve, TPR and FPR. In some statistical evaluations, these measures are evaluated against baseline metrics such as random classifier[35][55] or an existing classifier[52]. Generally, a summary content is compared with a human-model, a reference summary[1], called a gold standard summary.

[29]engineering.wayne.edu/cs/, Jan. 2016.

[30]www.cs.wm.edu/semeru/, Jan. 2016.

[31]www.python.org, Jan. 2016.

[32]www.csis.ysu.edu/∼bsharif/itraceMylyn/, Jan. 2016.

[33]www.csis.ysu.edu/, Jan. 2016.

[34]http://nlp.stanford.edu/, Jan. 2016.

[35]In which a coin toss is used to decide which sentences to include in a summary.

**Table 4**. List of Tools Regarding Software Artifact Summarization

| Tool Name | Description | Authors | URL |
|---|---|---|---|
| LibLinear | A library for linear classification | Rastkar *et al.*[5,52] | www.csie.ntu.edu.tw/∼cjlin/liblinear/ |
| BC3 Annotation Framework | A tool for bug reports annotation | Rastkar *et al.*[5,52], Jiang *et al.*[54] | www.cs.ubc.ca/nest/lci/bc3/framework.html |
| JGibLDA | Java implementation of LDA | Yeasmin *et al.*[60] | jgibblda.sourceforge.net/ |
| Sentiment140 | Twitter Sentiment API | Yeasmin *et al.*[60], Zhang and Hou[40] | help.sentiment140.com/api |
| Stanford NLP Parser | A statistical parser by Stanford University | Mani *et al.*[58], Panichella *et al.*[20], Zhang and Hou[40] | nlp.stanford.edu/software/lex-parser.shtml |
| JayFX | Eclipse Plugin | Rastkar *et al.*[30] | cs.mcgill.ca/∼swevo/jayfx/ |
| Jex | Eclipse Plugin | Rastkar *et al.*[30] | cs.mcgill.ca/∼swevo/jex/ |
| Jena | Framework for building semantic web apps | Rastkar *et al.*[30] | jena.sourceforge.net |
| srcML | XML representation of source code | Panichella *et al.*[20,45] | www.srcml.org/ |
| CODES | Source code descriptions from developers discussions | Panichella *et al.*[20], Vasallo *et al.*[25] | www.ing.unisannio.it/spanichella/pages/tools/CODES/ sourceforge.net/p/codesplugin/code/ci/master/tree/ |
| JStereoCode | Eclipse Plugin for stereotype identification | Moreno *et al.*[31] | www.cs.wayne.edu/∼severe/jstereocode/ |
| JSummarizer | Eclipse Plugin for automatic summarization | Moreno *et al.*[32] | www.cs.wayne.edu/∼severe/jsummarizer |
| LibSVM | A library for SVM classification | Ying and Robillard[55], Rastkar and Murphy[18], Nazar *et al.*[56] | www.csie.ntu.edu.tw/∼cjlin/libsvm/ |
| CFS | Code fragment summarizer implementation for SVM and naive Bayes classifiers | Nazar *et al.*[56] | oscar-lab.org/CFS/ |
| Sumslice | Similarity between source code and source code summaries | McBurney and McMillan[33-35] | www3.nd.edu/∼pmcburne/summaries/ |
| CallGraph | Static and dynamic call graphs for Java | McBurney and McMillan[33] | github.com/gousiosg/java-callgraph |
| EyeSum | Implementation of eye-tracking study | Rodeghero *et al.*[16-17] | www3.nd.edu/∼prodeghe/projects/eyesum/ |
| Ogama | Tool for capturing gaze movements | Rodeghero *et al.*[17] | www.ogama.net |
| Sumaylze | Tool for text similarity between source code and source code summaries | McBurney and McMillan[69] | www3.nd.edu/∼pmcburne/sumalyze/ |
| TASSAL | Tool for autofolding | Fowkes *et al.*[61] | github.com/mast-group/tassal |
| ARENA | Complete package for release set | Moreno *et al.*[37] | www.cs.wayne.edu/∼severe/fse2014/ |
| ChangeScribe | Eclipse plugin for generating commit messages | Cortés-Coy *et al.*[46] | github.com/SEMERU-WM/ChangeScribe |
| DeltaDoc | Implementation for source code change affect | Buse and Weimer[48] | code.google.com/archive/p/deltadoc/ |
| iTrace | Eye-tracking study implementation | Kevic *et al.*[64] | github.com/YsuSERESL/itrace-gaze-analysis github.com/YsuSERESL/iTrace |
| MUCCA | Email unified content classification approach | Bacchelli *et al.*[7] | mucca.inf.usi.ch/ |
| DECA | Development emails content analyzer | Sorbo *et al.*[8] | www.ifi.uzh.ch/seal/people/panichella/tools/DECA.ht |
| AutoComment | Automatic comment generation tool | Wong *et al.*[39] | asset.uwaterloo.ca/AutoComment/ |
| CloCom | Automatic comment generation tool | Wong *et al.*[74] | asset.uwaterloo.ca/clocom/ |
| CoreNLP POS | Stanford log-linear part-of-speech tagger | Rahman *et al.*[26] | nlp.stanford.edu/software/tagger.shtml |
| NLP Sentiment Analyzer | Sentiment analysis | Rahman *et al.*[26] | nlp.stanford.edu/sentiment/ |
| CodeInsight | CodeInsight: recommending insightful comments for source code using crowdsourced knowledge | Rahman *et al.*[26] | github.com/masud-technope/CodeInsight homepage.usask.ca/∼masud.rahman/codeinsight |
| Tregex | Tregex is a utility for matching patterns in trees | Zhang and Hou[40] | nlp.stanford.edu/software/tregex.shtml |
| Haystack | Documentation of online forums | Zhang and Hou[40] | www.clarkson.edu/∼dhou/projects/haystack2013.zip |
| Language Tools | Style and grammar checker | Panichella *et al.*[42] | github.com/languagetool-org/languagetool |
| Randoop | Automatic test generator of Java | Panichella *et al.*[42] | github.com/randoop/randoop |
| Javaparser | Java 1.8 parser and Abstract Syntax Tree for Java | Panichella *et al.*[42] | github.com/javaparser/javaparser |
| TestDescriber | Test case summary generator and evaluator | Panichella *et al.*[42] | ifi.uzh.ch/seal/people/panichella/tools/TestDescriber.html |
| TraceLab | LDA-GA implementation | Panichella *et al.*[22] | dibt.unimol.it/reports/LDA-GA github.com/CoEST/TraceLab coest.org/index.php/tracelab |
| UnitTestScribe | Unit test case document generator | Li *et al.*[43] | cs.wm.edu/semeru/data/ICST16-UnitTestScribe/ |

Gold standard summary (GSS) or simply a gold standard is generated through a human annotation procedure. In an annotation procedure, human annotators select a subset of an artifact that is used as a reference set for evaluating automatic summaries. It is generally performed prior to the development of summarization system. For instance, Fowkes *et al.*[61] hired two expert Java developers as annotators for performing manual folding of source code. Rastkar *et al.*[5,52] hired human annotators from the same institution for manually creating a corpus of reference summaries and they trained their logistic regression classifier on this corpus. Similarly, Ying and Robillard[55] and Nazar *et al.*[56] evaluated source-to-source summaries on a set of manually created source code gold standard summaries.

Since creating golden summaries requires significant manual effort and should be done by experts, building a reasonably-sized training set of golden summaries could be considered as an impediment for such technique[51]. Another problem with annotation is that annotators often do not agree on the same summary, as summarization is a subjective process, and there is no single best summary for a given set of data. To mitigate this problem researchers applied Kappa test also known as Cohen's Kappa[77] to measure the level of agreement between the annotators. All statistical measures employed in the surveyed papers are briefly discussed below.

### 5.1.1 Precision and Recall

To evaluate the usefulness of generated summaries, information retrieval metrics of precision and recall are used[76]. In general settings, a person is asked to select the sentences, conveying the real meanings of the text to be summarized. Next, the sentences selected automatically by the system are evaluated against the human selections[76]. However, it varies slightly as per the settings and requirements of the proposed system.

In information retrieval with a binary classification problem, precision is the fraction of retrieved instances that are relevant, while recall is the fraction of relevant instances that are retrieved. Both precision and recall are therefore based on an understanding and measure of relevance. In simple terms, high precision means that an algorithm returns substantially more relevant results than irrelevant ones. While high recall means that an algorithm returns most of the relevant results. In summarization settings, precision and recall are commonly measured as described in (1) and (2). However, these notions could be extended to rank different retrieval situations[11].

$$precision = \frac{TP}{TP+FP}\,. \tag{1}$$

Recall is measured as follows:

$$recall = \frac{TP}{TP+FN}\,. \tag{2}$$

where,

- $TP$ is the true positives,
- $FP$ stands for false positives, and
- $FN$ denotes the false negatives.

### 5.1.2 F-Score

As there is always a quality compromise between precision and recall, being desirable but different features, the $F$-score is used to counter this problem. In binary classification settings, $F$-score is generally employed for testing accuracy. The main problems with precision and recall for summarization are their incapability of distinguishing between many possible summaries, and content-wise different summaries may get very similar scores. $F$-score is a harmonic mean (one of the several kinds of average) of precision and recall. (3) shows the formula for measuring $F$-score

$$F\text{-}score = 2 \times \frac{precision \times recall}{precision + recall}\,. \tag{3}$$

### 5.1.3 Pyramid Precision

Pyramid precision provides normalized evaluation measure taking into account the multiple annotations available for each artifact[52]. It was first developed by Nenkova and Passonneau[78] for analyzing the content variation[76] and quality of summaries when multiple annotations were available[52].

Lotufo *et al.*[51,59] performed statistical evaluation based on precision, recall, and pyramid precision measure to evaluate the efficacy of unsupervised summarizer. Furthermore, they compared their results with the BRC summarizer[52]. Similarly, Rastkar *et al.*[5,52], Jiang *et al.*[54] and Yeasmin *et al.*[60] performed statistical evaluation using precision, recall, $F$-score, ROC-AUROC curve, and FPR-TPR values. Besides, Haiduc *et al.*[9,14], Moreno and Aponte[15] and Binkley *et al.*[19] additionally employed cosine similarity for statistical evaluation of summaries. Rahman *et al.*[26] also utilized recall and precision along with mean reciprocal rank (MMR) and gold summaries to evaluate the summarization system that produced insightful comments for source code. McBurney and McMillan[33-35] and

Rodeghero *et al.*[16-17] employed Mann-Whitney[36] and Wilcoxon Signed Rank tests for evaluating summaries distributions.

## 5.2 Criteria-Based Evaluation

Another kind of evaluation performed in selected studies is criteria-based evaluation. Criteria evaluation is based on certain measures which are related to the main goal, intent or purpose of the summarization task. These standards target aspects of accuracy, context, content adequacy, and conciseness of summaries. Researchers hired developers or participants from the same institution or companies to evaluate summaries on different criteria. For instance, Sridhara *et al.*[3,27] hired 13 participants to evaluate summaries based on accuracy, conciseness, and adequacy criteria. Eight participants performed the criteria-based evaluation in [30]. They assessed if summaries are helpful for developers. Wong *et al.*[39] also tested summaries based on factors of accuracy, usefulness, and adequacy.

## 5.3 Qualitative Evaluation

Evaluation that intends to measure the quality of generated summaries is called qualitative evaluation. Panichella *et al.*[20] conducted qualitative analysis of generated summaries by calculating false positives and false negatives. In another study[42], Panichella *et al.* measured the quality of summaries on content adequacy, conciseness, and expressiveness standards. McBurney *et al.*[36] employed human participants to suggest what keywords need to be or need not to be in automatically generated summaries. Similarly, they performed a qualitative analysis based on verbosity, accuracy and informativeness of summaries document in [33].

## 5.4 Content-Based Evaluation

Some researchers evaluated summaries' content on different standards. Moreno *et al.*[31-32] asked 22 programmers to evaluate three aspects of summaries related to the content of the summaries. These aspects are if the summaries' contents are expressive or concise (limiting unnecessary information), or adequate (whether there was a missing information) in most cases. Buse and Weimer[48] performed a content and quality-based evaluation focusing primarily on informative and conciseness of the summaries.

Table 5 provides a list of summary evaluation methods or metrics employed in the surveyed papers.

## 6 Collection and Distribution of Studies

This research is undertaken as a systematic literature review based on certain guidelines we defined. These guidelines are inspired by the principles proposed in studies [79-80]. Our guidelines regarding the selection of studies are mainly composed of two major steps: 1) information sources and 2) the methodology for selecting these sources. In the following subsections, we discuss these steps.

## 6.1 Information Sources

It is necessary to define the sources to perform the selection of the sources where searches for primary studies will be executed. With this regard, we defined the following criteria.

- publications in journals and conferences with the most impact;
- availability of search mechanism using keywords;
- availability on the Web.

Concerning language studies, the obtained primary studies must be written in English. The sources have been identified on the basis of the judgment of the authors of this paper. The list of sources includes relevant conferences and journals in which summarizing software artifacts research area is widely dealt with. We considered papers published in the proceedings of different conferences such as: International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Automated Software Engineering (ASE), The Working Conference on Reverse Engineering (WCRE) and IEEE International Working Conference on Source Code Analysis and Manipulation. Moreover, we also considered papers published in high quality peer reviewed journals, e.g., IEEE Transactions on Software Engineering, Empirical Software Engineering. Besides, we considered other online resources, e.g., arXiv, CiteSeerX, IEEE, ACM, Google Scholar, as potential sources for information gathering.

Taking into account the defined sources selection criteria, we got the initial primary list of sources as shown in Table 6. It should be noted that the list is huge and only primary sources are mentioned here. The authors of this paper have evaluated the list of

[36]The Mann-Whitney test is non-parametric, and it does not assume that the data are normally distributed.

**Table 5**. List of Evaluation Methods Employed w.r.t. Surveyed Studies

| Author | Evaluation |
|---|---|
| Haiduc *et al.*[9,14] | Qualitative evaluation based on developers perception + content evaluation |
| Rastkar *et al.*[29-30] | Criteria-based evaluation based on developers survey |
| Sridhara *et al.*[3,27], Cortés-Coy *et al.*[46] | Criteria → accuracy, conciseness, adequacy |
| Eddy *et al.*[4] | Criteria |
| Moreno and Aponte[15] | Qualitative → usefulness |
| Moreno *et al.*[31-32] | Criteria → adequacy, conciseness, expressiveness |
| Buse and Weimer[48] | Qualitative |
| Rastkar *et al.*[5,18,52], Kulkarni *et al.*[38,58], Ying and Robillard[55], Jiang *et al.*[54,61], Nazar *et al.*[56], Panichella *et al.*[22], Apnote and Marcus[62], Zhang and Hou[40] | Statistical + gold standard summary |
| Binkley *et al.*[19] | Content → readable, responsiveness + statistical → cosine similarity between human and automated summaries |
| McBurney and McMillan[33-35] | Statistical → Mann-Whitney test + qualitative → informativeness, verbosity, accuracy |
| McBurney *et al.*[36] | Qualitative → accuracy |
| Panichella *et al.*[20] | Qualitative |
| Kamimura and Murphy[41] | Content → relevance, usefulness |
| Moreno *et al.*[37] | Qualitative → completeness, correctness, importance |
| Rodeghero *et al.*[16-17] | Criteria based + statistical → Wilcoxon signed rank test, Mann-Whitney |
| Treude *et al.*[71] | Statistical → Wilcoxon signed rank test, Mann-Whitney |
| Abid *et al.*[45] | No evaluation |
| Lotufo *et al.*[51,59] | Statistical + qualitative |
| Yeasmin *et al.*[60] | Statistic + users comparison with non-visualized and visualized summaries |
| Wong *et al.*[39] | Criteria → accuracy, adequacy, and usefulness |
| Rahman *et al.*[26] | Statistical → recall, mean reciprocal rank (MMR) + gold standard summary |
| Sridhara *et al.*[28] | Human evaluation |

**Table 6**. Primary List of Sources

| Source | Name | Website |
|---|---|---|
| 1 | IEEE Explore Digital Library | http://ieeexplore.ieee.org/xpl/conferences.jsp |
| 2 | ACM Digital Library | http://dl.acm.org |
| 3 | Springer Link | http://link.springer.com/ |
| 4 | Science Direct | http://sciencedirect.com/ |
| 5 | Google Scholar | http://scholar.google.com |
| 6 | Web of Knowledge | http://webofknowledge.com/ |
| 7 | ICSE and related conference proceedings (Jan. 2010∼Apr. 2016) | http://icseconferences.org/proceedings.html |
| 8 | IEEE TSE | http://computer.org/tse |
| 9 | Empirical Software Engineering | http://www.springer.com/computer/swe/journal/10664 |
| 10 | arXiv | http://arxiv.org |

sources obtained and approved all the elements in this list. Once the sources have been defined, it is necessary to describe the processes and the criteria for selecting studies[80]. We considered the proposals in [79-80] to define the criteria for the inclusion and exclusion of selected papers in the context of the systematic review.

## 6.2 Methodology

Our methodology is mainly based on two major steps namely: keyword search and citation search. Both steps are explained as following.

### 6.2.1 *Keyword Selection*

As defined in Table 7, we passed the selected keywords into the sources defined in Table 6 to find selected papers. The keyword selection process is manual in nature. The selected list is defined manually by reading the title and abstract of the papers. To further refine the list, we applied the combination of operators AND and OR with keywords as search strings, with adaptions to the desired search engines. We sorted the initial list of selected papers through this process. Table 7 lists keywords that were passed in the sources defined in Table 6 to find the initial list of related studies. In the end, we evaluated the list of papers obtained and approved all elements of the list.

**Table 7.** List of Search Strings

| No. | Search String |
|---|---|
| 1 | Bug reports AND (noise OR bias) |
| 2 | Duplicate AND (bug OR code) |
| 3 | Duplicate detection AND summarization |
| 4 | Code change AND summarization |
| 5 | Analysis AND summarization |
| 6 | Code fragment AND summarization |
| 7 | Traceability AND (bug OR code) |

### 6.2.2 *Citation Search*

To further support the manual keyword search, we performed a citation analysis on selected papers using the same set of keywords and reading them one by one. After removing duplicates, we evaluated each paper for inclusion in the set of candidate papers. We again adopted the same strategy of reading title and abstract of these papers. The main emphasis was to include papers unless they were clearly irrelevant. Papers which all authors agreed to include were included and any papers which all authors agreed to exclude were excluded. Any papers for which the inclusion/exclusion assessment differed among authors were discussed until either agreement was reached or the paper was provisionally included[79]. Fig.5 illustrates the flowchart depicting our defined methodology. It mentions two activities. The goal of the first activity is selecting primary studies through keyword search while the second activity focuses on selecting studies through manual citation search.

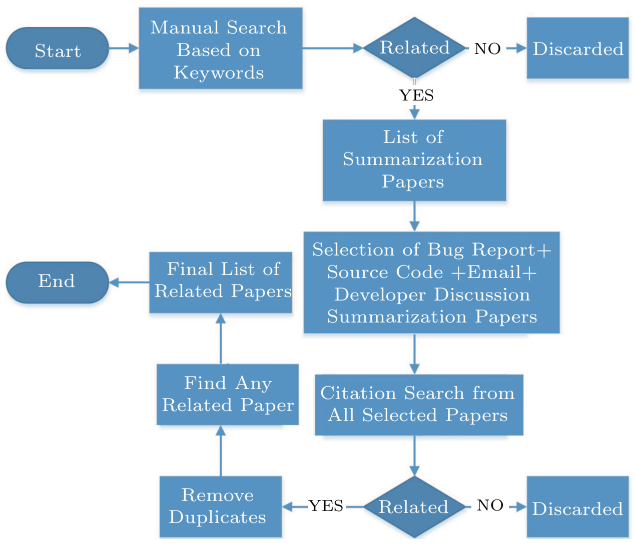

Start
Manual Search Based on Keywords
Related
NO
Discarded
YES
List of Summarization Papers
Selection of Bug Report+ Source Code +Email+ Developer Discussion Summarization Papers
Citation Search from All Selected Papers
Related
NO
Discarded
YES
Remove Duplicates
Find Any Related Paper
Final List of Related Papers
End

Fig.5. Basic procedure for executing the selection of systematic review studies.

## 6.3 Distribution and Trend of Studies over Publication Channels and Years

Fig.6 shows the distribution of studies over the years that are considered for review. Most of the studies were carried out from 2010 onward, and there is a notable increment in 2013, 2014 and 2015 with maximum studies published in 2014. This recent increase may be a reflection of growing interest in the field of summarizing bug reports, source code, emails and developer discussions.

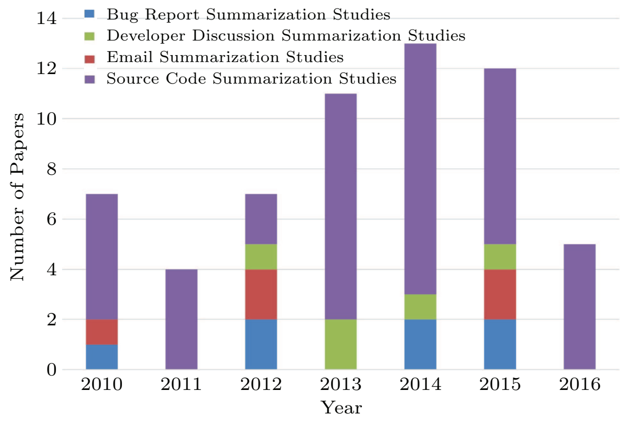


Fig.6. Stack column chart depicting the trend of studies over years.

With respect to total studies, i.e., 59 selected for this review, we see that 42 (71.18%) studies in this review are regarding source code summarization, whereas, only seven studies (11.86%) are directly related to bug report summarization. Ten studies (16.94%) discuss emails and developer discussions and their utilization for software artifact summarization. Interestingly, there are only nine papers (15.25%) published in high impact journals and all remaining papers are conference papers (84.74%). Table 8 provides the distribution of studies in conferences and journals.

## 7 Discussion

Here we discuss modern communication channels, commonalities and differences of software artifacts discussed above, and their limitations, challenges and future directions.

**Table 8.** Distribution of Studies, Publication Channels and Occurrences

| Publication Channel | Type | Number of Papers |
|---|---|---|
| ASE | Conference | 4 |
| CRE | Conference | 1 |
| FSE | Conference | 7 |
| ICPC | Conference | 11 |
| ICSE | Conference | 12 |
| ICSM | Conference | 4 |
| ICSME | Conference | 2 |
| ICST | Conference | 1 |
| MSR | Conference | 1 |
| RAISE | Conference | 1 |
| SCAM | Conference | 2 |
| TEFSE | Conference | 1 |
| SANER | Conference | 1 |
| TAinSM | Conference | 1 |
| Emp. Soft Engg | Journal | 3 |
| Front. Comp. Sci. | Journal | 1 |
| CLEI | Journal | 1 |
| IEEE-Trans. | Journal | 3 |
| Sci. China. Inf. Sci. | Journal | 1 |
| arXiv | Database | 1 |

### 7.1 Modern Communication Channels

Traditional communication channels, i.e., bug reports and mailing lists are discussed in Subsection 2.1. Here we briefly discuss modern communication channels, which are IRC chat logs/channels, online forums and question & answers sites. Online forums and question & answer sites are interchangeable terminologies in an SE paradigm and both are discussed in Subsection 2.1.4.

On an IRC channel, participants talk about the project and implementation details in general. The chat logs contain the record of the instant messaging conversations between project stakeholders, and typically contain a series of time-stamped, author-stamped text messages[81]. QA sites are online forums, where engineers, developers, and users post different questions and discuss them. The developer mailing list is the primary communication channel for an open source software (OSS) project. Other communication channels include vulnerability databases, revision control databases, version control, requirement and design documents, and execution logs.

### 7.2 Complementarities and Commonalities

As discussed in Subsection 2.1, bug reports and mailing lists are conversational data, which constitute a precious source of information. Both artifacts contain structured as well as unstructured data. Therefore, the structured content, i.e., source code, stack trace and patches can be useful for developers in different software engineering tasks, such as program comprehension and source code re-documentation. Rastkar *et al.*[52] first recognized the similarity between email threads and bug reports, and utilized an existing technique[53] created for emails and conversations summarization to produce concise summaries of bug reports. Other efforts[51,54,58-60] in bug report summarization complemented Rastkar *et al.*'s work and used it as a benchmark for developing or improving summarization systems.

However, these artifacts differ in structure, purpose and collaboration, as bug reports concern the creation and resolution of software bugs and other maintenance tasks. Whereas, mailing lists constitute a set of time-stamped messages among a group of people across the Internet. Collaboration in bug reports develops as a conversation, similar to email threads: participants post messages — commonly referred to as comments — as their contributions. A bug report is, therefore, the result of the communication that took place in order to address a bug. Unlike a forum, it is not collaboratively constructed with the intention of being easy to read and comprehend. Since comments have a context set by their previous comments and useful information is spread out throughout the thread, to comprehend a bug report, it is often necessary to read almost the entire conversation.

Guzzi *et al.*[82] first defined the taxonomy of mailing lists that include following categories: implementation, technical infrastructure, project status, social interactions, usage and discarded. Sorbo *et al.*[8] further refined this taxonomy on the basis of the conceptual framework of discussions of different natures inferred by the categories across different channels. These categories are intent, feature request, opinion asking, problem discovery, solution proposal, information seeking, and information giving.

## 7.3 Challenges and Future Directions

All studies related to bug reports and source code summarization share a common limitation: they treat every artifact as a purely textual artifact, or they limit their summarization techniques to a single type of artifact. We discuss limitations and challenges concerning the summarization software artifact paradigm from the surveyed studies in subsections below.

### *7.3.1 Bug Reports and Mailing Lists*

As discussed in Subsection 2.1, bug reports and mailing lists are conversational data, which constitute a precious source of information. Both artifacts contain structured and unstructured data. Therefore, the structured content, i.e., source code, stack trace and patches can be useful for developers in different software engineering tasks, such as program comprehension and source code re-documentation. Rastkar *et al.*[52] first recognized the similarity between email threads and bug reports, and utilized an existing technique[53] created for emails and conversations summarization to produce concise summaries of bug reports. Their results showed that the quality of summaries is dependent on the training and actual corpus.

Mani *et al.*[58] evaluated the quality of summaries generated through four well-known unsupervised algorithms. They selected unsupervised classifiers in order to eliminate the overhead of creating supervised predictor in [52]. They found, however, that these approaches are only effective after removing noise from bug reports. Therefore, they proposed a heuristic-based noise reduction approach that tries to automatically classify sentences as either a question, an investigative sentence, or a code snippet. When using this noise reduction heuristic, they found that each of the four well-known textual summarizers produced a summary of at least equal quality compared with the supervised approach. These efforts along with [51, 54, 59-60] did not consider the structural content of bug reports and emails for summarization purpose. Therefore, these portions along with new techniques for noise reduction in text data can be considered for future research in conversational software artifacts summarization, in particular, bug report summarization.

Sridhara *et al.*[3] generated textual summaries to describe Java methods. They used a number of heuristics on a method's signature and a method's code to find the main intent of the code. Haiduc *et al.*[9], with the same objectives, used well-known information retrieval methods to generate term-based summaries of methods and classes. These term-based summaries do not compose a grammatically valid phrase but are intended to be the set of most relevant terms that together should describe an entity. Other efforts by Eddy *et al.*[4] and Moreno and Aponte[15] complemented this study and extended it by employing hPAM and more human evaluators. To further develop better and more accurate summaries McBurney and McMillan[33-35] employed NLP methods based on SWUM and Verb-DO techniques. However, they argued that previous studied focused primarily on the goodness of summaries rather than the usefulness. Therefore, we believe another limitation regarding generated summaries is whether software artifact summary can be useful for a developer or not.

### *7.3.2 Heterogeneous Artifacts*

Heterogeneous artifacts are the combination of both structured and unstructured information. Commonly a software artifact is considered or treated homogeneously while most of the current approaches in summarization do not take into account the multidimensional nature of software artifacts. For example, Rastkar *et al.*[52] treated bug reports as conversational textual data and ignored other important elements such as code snippets while generating summaries. However, software artifacts cannot be considered solely as containers of homogeneous information. The information provided by software artifacts is rather heterogeneous and includes complete source code, code snippets, text, and many other types of information[83]. Stack Overflow is an example of such artifacts, where discussion content includes natural language text, source code or code snippets, XML configurations, images and many other things.

According to Bacchelli *et al.*[7]: "Most of the general purpose summarization approaches are tested on well-formed, or sanitized, natural language documents. When summarizing development emails, however, we have to deal with natural language text which is often not well formed and is interleaved with languages with different syntaxes, such as code fragments, stack traces, patches, etc. [...] Currently no summarization technique takes this aspect into account [...]".

Though we have found that studies of [25-26, 39, 74] have utilized information from Stack Overflow for summarization tasks, more efforts need to be invested in order to make more precise and accurate systems for both homogeneous and heterogeneous artifacts.

### *7.3.3 Academic and Industry Cooperation*

While the actual challenge in our opinion is that academic researchers sometimes have limited insight into the development processes and they depend fully on assumptions that cannot be verified (and usually do not hold) in industrial settings. Academic researchers employ open source projects for experimentation. Whereas, in the industry, there are predefined and refined processes. Therefore, software developers in the industry have different requirements or perspectives for a given task from what researchers hypothesize in academia. For instance, in email artifacts, it is not yet clear whether emails written in OSS communities are equivalent to those written in the industry[7]. Can we generalize findings of summarization learnt from OSS emails to other settings[7]? Similarly, for source code summarization, experiments are done on OOS whereas industrial software may require different settings, features, and requirements. Can we generalize the findings regarding source code summarization in OSS to the source code in industrial settings?

Another issue is the role of participant and corpus size in academic research. Finding proper participants for software engineering research is a daunting task, especially in an academic setting[7]. Generally, researchers use smaller samples or corpus sizes and fewer annotators for annotating corpus. Could this correspond well in industrial settings? Here corpus size is often quite large when compared with academic settings. Annotation is mostly done by graduate students or the participants from the same institute. How can a non-expert in a specific software project capture the proper meanings of an artifact? Should we have the best golden set[7]? Therefore, we believe this area needs immediate attention and these issues shall be addressed immediately. Moreover, there is a demand for the development of new methods, tools, and approaches, in cooperation with industry as the industry requires more effective and accurate summarization tools and systems.

### *7.3.4 Summaries Perspective*

McBurney and McMillan[69] argued that the source code summarization approaches have such a key similarity that they influence the reader's perspective and ignore the writer's perspective. Authors are the writers of the source code and readers seek to understand it. Therefore, there is a conflict between the authors' and readers' perspective. Authors translate the high-level concepts into low-level implementation, while readers must deduce the concepts and behaviours from low-level details. Hence, readers struggle in understanding the real meanings behind the source code and inevitably make mistakes. At the same time, authors who write the documentation of their source code must choose which key concepts and details to communicate with readers via documentation. The concepts and details described in documentation should be the ones that authors believe readers would need to know. This challenge needs to be addressed in future summarization studies.

### *7.3.5 Crowdsourcing*

Crowdsourcing is one of the emerging Web 2.0 based phenomena and in recent years has attracted great attention from both practitioners and researchers[84]. It is used as a platform for connecting people, organizations and societies in order to increase mutual cooperation between each other. In 2006, Howe[85] first coined the term crowdsourcing and based it on the notion that virtually everyone can contribute a valuable information, or participate in an activity online through an open call. Academic scholars from different disciplines have examined various issues and challenges, and applied crowdsourcing to resolve such issues and challenges[86]. A typical crowdsourcing process works in the following way. An organization identifies tasks and releases them online to a crowd of outsiders who are interested in performing these tasks. A wide variety of individuals then offer their services to undertake the tasks individually or collaboratively. Upon completion, the individuals submit their work to an organization which later evaluates it[85,87].

Crowdsourcing can be employed as a problem solving model or a mechanism for summarizing software artifacts, both source code and bug reports. For example, crowdsourcing may help in extracting syntactic or query based features for source code. It can be further extended for corpus creation and annotation as well as for the summary comparison. Furthermore, crowdsourcing can be applied for creating a shared corpus by employing geographically separated researchers. Recently, Nazar *et al.*[56] applied crowdsourcing on a limited scale to extract source code features from code fragments for a summarization task. However, crowdsourcing could be extended on a wider scale for software artifact summarization and other mining software repositories activities.

## 8 Conclusions

This study conducted a literature review of the state of the art in summarizing software artifacts. We focused on what kind of and how summaries are generated for bug reports, source code, mailing lists and developer discussion artifacts and discussed data mining and machine learning methods that were employed to accomplish summarization task from Jan. 2010 to Apr. 2016. Furthermore, we provided the real-life applications in the context of what tasks have been achieved through the summarization of software artifacts. Tools that are developed for generating summaries, or used during summarization process were also listed in order to facilitate junior researchers. The evaluation problem is unavoidable during review, and consequently, a special attention was given to the evaluation methods and measures used in the existing studies. In the end, we discussed some major challenges in software artifact summarization that require immediate attention.

## References

[1] Lloret E, Palomar M. Text summarisation in progress: A literature review. *Artificial Intelligence Review*, 2012, 37(1): 1-41.

[2] Murphy G C. Lightweight structural summarization as an aid to software evolution [Ph.D. Thesis]. University of Washington, 1996.

[3] Sridhara G, Hill E, Muppaneni D, Pollock L L, Vijay-Shanker K. Towards automatically generating summary comments for java methods. In *Proc. the 25th IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2010, pp.43-52.

[4] Eddy B P, Robinson J A, Kraft N A, Carver J C. Evaluating source code summarization techniques: Replication and expansion. In *Proc. the 21st International Conference on Program Comprehension*, May 2013, pp.13-22.

[5] Rastkar S, Murphy G C, Murray G. Automatic summarization of bug reports. *IEEE Transactions on Software Engineering*, 2014, 40(4): 366-380.

[6] Bettenburg N, Premraj R, Zimmermann T, Kim S. Extracting structural information from bug reports. In *Proc. the International Working Conference on Mining Software Repositories*, May 2008, pp.27-30.

[7] Bacchelli A, Lanza M, Mastrodicasa E S. On the road to hades-helpful automatic development email summarization. In *Proc. the 1st International Workshop on the Next Five Years of Text Analysis in Software Maintenance*, Sept. 2012.

[8] Di Sorbo A, Panichella S, Visaggio C A, Di Penta M, Canfora G, Gall H C. Development emails content analyzer: Intention mining in developer discussions (T). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2015, pp.12-23.

[9] Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text summarization techniques for summarizing source code. In *Proc. the 17th Working Conference on Reverse Engineering*, Oct. 2010, pp.35-44.

[10] Nenkova A, McKeown K. A survey of text summarization techniques. In *Mining Text Data*, Aggarwal C C, Zhai C (eds.), Springer US, 2012, pp.43-76.

[11] Manning C D, Raghavan P, Schütze H. Introduction to Information Retrieval (1 edition). Cambridge University Press, 2008.

[12] Kagdi H, Collard M L, Maletic J I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 2007, 19(2): 77-131.

[13] Bacchelli A, Lanza M, Robbes R. Linking e-mails and source code artifacts. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering* - Volume 1, May 2010, pp.375-384.

[14] Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp.223-226.

[15] Moreno L, Aponte J. On the analysis of human and automatic summaries of source code. *CLEI Electronic Journal*, 2012, 15(2).

[16] Rodeghero P, McMillan C, McBurney P W, Bosch N, D'Mello S. Improving automated source code summarization via an eyetracking study of programmers. In *Proc. the 36th International Conference on Software Engineering*, May 2014, pp.390-401.

[17] Rodeghero P, Liu C, McBurney P, McMillan C. An eyetracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering*, 2015, 41(11): 1038-1054.

[18] Rastkar S, Murphy G C. Why did this code change? In *Proc. the 2013 International Conference on Software Engineering*, May 2013, pp.1193-1196.

[19] Binkley D, Lawrie D, Hill E, Burge J, Harris I, Hebig R, Keszocze O, Reed K, Slankas J. Task-driven software summarization. In *Proc. the 29th IEEE International Conference on Software Maintenance*, Sept. 2013, pp.432-435.

[20] Panichella A, Aponte J, Di Penta M, Marcus A, Canfora G. Mining source code descriptions from developer communications. In *Proc. the 20th International Conference on Program Comprehension (ICPC)*, Jun. 2012, pp.63-72.

[21] Blei D M, Ng A Y, Jordan M I. Latent Dirichlet allocation. *The Journal of Machine Learning Research*, 2003, 3: 993-1022.

[22] Panichella A, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, De Lucia A. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.522-531.

[23] De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Using IR methods for labeling source code artifacts: Is it worthwhile? In *Proc. the 20th International Conference on Program Comprehension*, Jun. 2012, pp.193-202.

[24] De Lucia A, Di Penta M, Oliveto R, Panichella A, Panichella S. Labeling source code with information retrieval methods: An empirical study. *Empirical Software Engineering*, 2014, 19(5): 1383-1420.

[25] Vassallo C, Panichella S, Di Penta M, Canfora G. Codes: Mining source code descriptions from developers discussions. In *Proc. the 22nd International Conference on Program Comprehension*, May 2014, pp.106-109.

[26] Rahman M M, Roy C K, Keivanloo I. Recommending insightful comments for source code using crowdsourced knowledge. In *Proc. the 15th International Working Conference on Source Code Analysis and Manipulation* (*SCAM*), Sept. 2015, pp.81-90.

[27] Sridhara G, Pollock L L, Vijay-Shanker K. Generating parameter comments and integrating with method summaries. In *Proc. the 19th IEEE International Conference on Program Comprehension*, Jun. 2011, pp.71-80.

[28] Sridhara G, Pollock L, Vijay-Shanker K. Automatically detecting and describing high level actions within methods. In *Proc. the 33rd International Conference on Software Engineering* (*ICSE*), May 2011, pp.101-110.

[29] Rastkar S. Summarizing software concerns. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering* - Volume 2, May 2010, pp.527-528.

[30] Rastkar S, Murphy G C, Bradley A W J. Generating natural language summaries for crosscutting source code concerns. In *Proc. the 27th International Conference on Software Maintenance*, Sept. 2011, pp.103-112.

[31] Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L L, Vijay-Shanker K. Automatic generation of natural language summaries for java classes. In *Proc. the 21st International Conference on Program Comprehension*, May 2013, pp.23-32.

[32] Moreno L, Marcus A, Pollock L L, Vijay Shanker K. Jsummarizer: An automatic generator of natural language summaries for java classes. In *Proc. the 21st International Conference on Program Comprehension* (*ICPC*), May 2013, pp.230-232.

[33] McBurney P W, McMillan C. Automatic documentation generation via source code summarization of method context. In *Proc. the 22nd International Conference on Program Comprehension*, Jun. 2014, pp.279-290.

[34] McBurney P W, McMillan C. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 2016, 42(2): 103-119.

[35] McBurney P W. Automatic documentation generation via source code summarization. In *Proc. the 37th International Conference on Software Engineering* - Volume 2, May 2015, pp.903-906.

[36] McBurney P W, Liu C, McMillan C, Weninger T. Improving topic model source code summarization. In *Proc. the 22nd International Conference on Program Comprehension*, June 2014, pp.291-294.

[37] Moreno L, Bavota G, Di Penta M, Oliveto R, Marcus A, Canfora G. Automatic generation of release notes. In *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014, pp.484-495.

[38] Kulkarni N, Varma V. Supporting comprehension of unfamiliar programs by modeling an expert's perception. In *Proc. the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, Jun. 2014, pp.19-24.

[39] Wong E, Yang J, Tan L. Autocomment: Mining question and answer sites for automatic comment generation. In *Proc. the IEEE/ACM 28th International Conference on Automated Software Engineering* (*ASE*), Nov. 2013, pp.562-567.

[40] Zhang Y, Hou D. Extracting problematic API features from forum discussions. In *Proc. the 21st International Conference on Program Comprehension* (*ICPC*), May 2013, pp.142-151.

[41] Kamimura M, Murphy G C. Towards generating human-oriented summaries of unit test cases. In *Proc. the 21st International Conference on Program Comprehension* (*ICPC*), May 2013, pp.215-218.

[42] Panichella S, Panichella A, Beller M, Zaidman A, Gall H C. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proc. the 38th International Conference on Software Engineering*, May 2016, pp.547-558.

[43] Li B, Vendome C, Linares-Vásquez M, Poshyvanyk D, Kraft N A. Automatically documenting unit test cases. In *Proc. the IEEE Int. Conf. Software Testing, Verification and Valication*, Apr. 2016, pp.341-352.

[44] Dragan N, Collard M, Maletic J. Automatic identification of class stereotypes. In *Proc. the IEEE International Conference on Software Maintenance* (*ICSM*), Sept. 2010, pp.1-10.

[45] Abid N, Dragan N, Collard M, Maletic J. Using stereotypes in the automatic generation of natural language summaries for C++ methods. In *Proc. the International Conference on Software Maintenance and Evolution*, Sept.29-Oct.1, 2015, pp.561-565.

[46] Cortés-Coy L F, Linares-Vásquez M, Aponte J, Poshyvanyk D. On automatically generating commit messages via summarization of source code changes. In *Proc. the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, Sept. 2014, pp.275-284.

[47] Moreno L, Marcus A. Jstereocode: Automatically identifying method and class stereotypes in java code. In *Proc. the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2012, pp.358-361.

[48] Buse R P, Weimer W R. Automatically documenting program changes. In *Proc. the IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2010, pp.33-42.

[49] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis. Springer, 2015.

[50] Kupiec J, Pedersen J O, Chen F. A trainable document summarizer. In *Proc the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Jul. 1995, pp.68-73.

[51] Lotufo R, Malik Z, Czarnecki K. Modelling the 'hurried' bug report reading process to summarize bug reports. In *Proc. the 28th IEEE International Conference on Software Maintenance*, Sept. 2012, pp.430-439.

[52] Rastkar S, Murphy G C, Murray G. Summarizing software artifacts: A case study of bug reports. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering*, Volume 1, May 2010, pp.505-514.

[53] Murray G, Carenini G. Summarizing spoken and written conversations. In *Proc. the Conference on Empirical Methods in Natural Language Processing*, Oct. 2008, pp.773-782.

[54] Jiang H, Zhang J, Ma H, Nazar N, Ren Z. Mining authorship characteristics in bug repositories. *Science China Information Sciences*, 2015. (Accepted)

[55] Ying A T T, Robillard M P. Code fragment summarization. In *Proc. the 9th Joint Meeting on Foundations of Software Engineering*, Aug. 2013, pp.655-658.

[56] Nazar N, Jiang H, Gao G, Zhang T, Li X, Ren Z. Source code fragment summarization with small-scale crowdsourcing based features. *Frontiers of Computer Science*, 2016, 10(3): 504-517.

[57] Petrosyan G, Robillard M P, Mori R D. Discovering information explaining API types using text classification. In *Proc. the 37th International Conference on Software Engineering*-Volume 1, May 2015, pp.869-879.

[58] Mani S, Catherine R, Sinha V S, Dubey A. AUSUM: Approach for unsupervised bug report summarization. In *Proc. the 20th International Symposium on the Foundations of Software Engineering*, Nov. 2012, Article No. 11.

[59] Lotufo R, Malik Z, Czarnecki K. Modelling the 'hurried' bug report reading process to summarize bug reports. *Empirical Software Engineering*, 2015, 20(2): 516-548.

[60] Yeasmin S, Roy C, Schneider K. Interactive visualization of bug reports using topic evolution and extractive summaries. In *Proc. the IEEE International Conference on Software Maintenance and Evolution*, Sept. 2014, pp.421-425.

[61] Fowkes J, Chanthirasegaran P, Allamanis M, Lapata M, Sutton C A. TASSAL: Autofolding for source code summarization. In *Proc. the 38th International Conference on Software Engineering Companion*, May 2016, pp.649-652.

[62] Aponte J, Marcus A. Improving traceability link recovery methods through software artifact summarization. In *Proc. the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, May 2011, pp.46-49.

[63] Fritz T, Shepherd D C, Kevic K, Snipes W, Bräunlich C. Developers' code context models for change tasks. In *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014, pp.7-18.

[64] Kevic K, Walters B M, Shaffer T R, Sharif B, Shepherd D C, Fritz T. Tracing software developers' eyes and interactions for change tasks. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, Aug.31-Sept.4, 2015, pp.202-213.

[65] Ying A T T, Robillard M P. Selection and presentation practices for code example summarization. In *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2014, pp.460-471.

[66] Sun C, Lo D, Khoo S C, Jiang J. Towards more accurate retrieval of duplicate bug reports. In *Proc. the 26th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Nov. 2011, pp.253-262.

[67] Wang X, Zhang L, Xie T, Anvik J, Sun J. An approach to detecting duplicate bug reports using natural language and execution information. In *Proc. the 30th ACM/IEEE International Conference on Software Engineering*, May 2008, pp.461-470.

[68] Runeson P, Alexandersson M, Nyholm O. Detection of duplicate defect reports using natural language processing. In *Proc. the 29th International Conference on Software Engineering*, May 2007, pp.499-510.

[69] McBurney P W, McMillan C. An empirical study of the textual similarity between source code and source code summaries. *Empirical Software Engineering*, 2014: 21(1): 17-42.

[70] Hill E, Pollock L, Vijay-Shanker K. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *Proc. the 31st International Conference on Software Engineering*, May 2009, pp.232-242.

[71] Treude C, Filho F F, Kulesza U. Summarizing and measuring development activity. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, Sept. 2015, pp.625-636.

[72] Chang C C, Lin C J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2011, 2(3): Article No. 27.

[73] Fan R E, Chang K W, Hsieh C J, Wang X R, Lin C J. Liblinear: A library for large linear classification. *Journal of Machine Learning Research*, 2008, 9: 1871-1874.

[74] Wong E, Liu T, Tan L. Clocom: Mining existing source code for automatic comment generation. In *Proc. the 22nd International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), Mar. 2015, pp.380-389.

[75] Jones K S, Galliers J R. Evaluating Natural Language Processing Systems: An Analysis and Review. Springer-Verlag Berlin Heidelberg, 1995.

[76] Nenkova A, McKeown K. Automatic summarization. *Foundations and Trends in Information Retrieval*, 2011, 5(2/3): 103-233.

[77] Cohen J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 1960, 20(1): 37-46.

[78] Nenkova A, Passonneau R J. Evaluating content selection in summarization: The pyramid method. In *Proc. the Human Language Technology/North American Chapter of the Association for Computational Linguistics*, May 2004, pp.145-152.

[79] Kitchenham B, Brereton P. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 2013, 55(12): 2049-2075.

[80] Mesquida A L, Mas A, Amengual E, Calvo-Manzano J A. It service management process improvement based on ISO/IEC 15504: A systematic review. *Information and Software Technology*, 2012, 54(3): 239-247.

[81] Shihab E, Jiang Z M, Hassan A E. Studying the use of developer IRC meetings in open source projects. In *Proc. the IEEE International Conference on Software Maintenance*, Nov. 2009, pp.147-156.

[82] Guzzi A, Begel A, Miller J K, Nareddy K. Facilitating enterprise software developer communication with cares. In *Proc. the 28th IEEE International Conference on Software Maintenance* (*ICSM*), Sept. 2012, pp.527-536.

[83] Ponzanelli L, Mocci A, Lanza M. Summarizing complex development artifacts by mining heterogeneous data. In *Proc. the 12th IEEE/ACM Working Conference on Mining Software Repositories*, May 2015, pp.401-405.

[84] Zhao Y, Zhu Q. Evaluation on crowdsourcing research: Current status and future direction. *Information Systems Frontiers*, 2014, 16(3): 417-434.

[85] Howe J. The rise of crowdsourcing. http: //www.wired.com/2006/06/crowds/, July 2006.

[86] Greengard S. Following the crowd. *Communications of the ACM*, 2011, 54(2): 20-22.

[87] Whitla P. Crowdsourcing and its application in marketing activities. *Contemporary Management Research*, 2009, 5(1): 15-28.

**Najam Nazar** received his B.Sc. (Hons.) degree in computer science from University of the Punjab, Lahore, Pakistan, in 2005, and M.S. degree in software engineering from Chalmers University of Technology, Sweden, in 2010. He is currently working towards his Ph.D. degree in software engineering at Dalian University of Technology, Dalian. His current research interest includes mining software repositories, data mining, natural language processing, and machine learning.

**Yan Hu** received his B.S. and Ph.D. degrees in computer science from University of Science and Technology of China (USTC), Hefei, in 2002 and 2007, respectively. He is currently an assistant professor at Dalian University of Technology, Dalian. His research interests include model checking, program analysis, and software engineering. He is a member of CCF and ACM.

**He Jiang** received his Ph.D. degree in computer science from the University of Science and Technology of China, Hefei, in 2004. He is currently a professor with the Dalian University of Technology, Dalian. His current research interests include search-based software engineering and mining software repositories. Dr. Jiang is also a member of CCF and ACM.