Towards Automatic Context-Aware Summarization of Code Entities

by

Elmira MOHSENZADEH KORAYEM

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF A MASTER'S
DEGREE
M.Sc.

MONTREAL, AUGUST 2019

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

Professor Latifa Guerrouj, Thesis Supervisor
Département de génie logiciel et des TI, École de technologie supérieure

Professor Carlos Vazquez, President of the Board of Examiners
Département de génie logiciel et des TI, École de technologie supérieure

Professor Sègla Jean-Luc Kpodjedo, Member of the jury
Département de génie logiciel et des TI, École de technologie supérieure

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Latifa Guerrouj for her unbelievable help and support during my Master. It has been a great pleasure working with Prof. Latifa. She inspired me all the time by her patience, enthusiasm, motivation, and immense knowledge. I am honoured to be part of her research team.

I cannot express enough thanks to my jury members, Professor Carlos Vazquez and Professor Sègla Jean-Luc Kpodjedo who accepted to evaluate this thesis. I offer my sincere appreciation for their time and insightful feedback.

I would like also to thank my family for their unconditional love, support, and guidance. Their encouragement when the times got rough is much appreciated. They are my ultimate role models. This accomplishment would not have been possible without them. Thank you.

I express my gratitude to my friends and lab mates for sharing their knowledge, opinion, and for all their encouragements.

VERS UNE SYNTHÈSE AUTOMATIQUE CONTEXTUELLE DES ENTITÉS DE CODE

Elmira MOHSENZADEH KORAYEM

RÉSUMÉ

Les développeurs de solutions informatiques utilisent différentes classes et méthodes dans l'accomplissement de leurs tâches quotidiennes. Dans cette perspective, ils doivent synthétiser une quantité importante d'information pour comprendre celles qui font parties de leurs tâches. Produire des résumés de qualités des dites informations permettrait d'aider les développeurs à accomplir leurs tâches de maintenance et d'évolution logicielle. Dans le but de fournir de l'information synthétisée sur l'utilisation des éléments de code, la documentation informelle comme les rapports de bugs peuvent être une source d'information pertinente.

Dans ce travail, nous proposons une approche basée sur une technique de machine learning, afin de produire des résumés pour les éléments de code (méthodes ou classes) présents dans les rapports de bogues. Dans l'approche suggérée, les éléments de code sont extraits en se basant sur une appraoche d'identification de code dans la documentation informelle, ensuite le résultat avec le contenu des bogues est utilisé par une technique de machine learning afin de générer un ensemble de phrase formant le résumé. Dans notre approche, noous appliquons l'algorithme d'apprentissage machine « logistic regression » pour classifier les phrases par importance. Le but est de construire un corpus de phrases pertinentes relatives aux éléments de code. En dernier lieu, une étude a été mené afin d'évaluer la qualité des résumés proposés. Pour finir, nous discuterons de l'utilité des résumés produits à partir de rapports de bugs en utilisant un algorithme de machine learning ainsi que des limitations de notre approche. Les résultats nous indiquent que les résumés peuvent réduire le temps et l'effort nécessaires à la compréhension des éléments de code. En effet, 43,5% trouvent que les résumés sont utiles pour réduire le temps de compréhension du code. En outre, 39,1% trouvent que les résumés sont utiles pour réduire l'effort de compréhension du code.

Dans le future, les résumés peuvent être produits à partir d'autres documentation. De plus, cette approche peut être appliquée dans des configurations pratiques. Par conséquent, il peut être utilisé dans un environnement de développement intégré tel que Eclipse pour aider les développeurs lors de leurs tâches de maintenance et d'évolution des logiciels.

Mots-clés: Résumé du code, Entités de code, Rapport de bogue, Apprentissage Automatique, Documentation informelle, Documentation formelle

TOWARDS AUTOMATIC CONTEXT-AWARE SUMMARIZATION OF CODE ENTITIES

Elmira MOHSENZADEH KORAYEM

ABSTRACT

Software developers are working with different methods and classes and in order to understand those that perplex them and–or that are part of their tasks, they need to tackle with a huge amount of information. Therefore, providing developers with high-quality summaries of code entities can help them during their maintenance and evolution tasks.

To provide useful information about the purpose of code entities, informal documentation (Stack Overflow) has been shown to be an important source of information that can be leveraged.

In this study, we investigate bug reports as a type of informal documentation and we apply machine learning to produce summaries of code entities (methods and classes) in bug reports. In the proposed approach, code entities are extracted using a technique in a form of an island parser that we implemented to identify code in bug reports. Additionally, we applied machine learning to select a set of useful sentences that will be part of the code entities' summaries.

We have used logistic regression as our machine learning technique to rank sentences based on their importance. To this aim, a corpus of sentences is built based on the occurrence of code entities in the sentences belonging to bug reports containing the code entities in question.

In the last step, summaries have been evaluated using surveys to estimate the quality of produced summaries.

The results show that the automatically produced summaries can reduce time and effort to understand the usage of code entities. Specifically, the majority of participants found summaries extremely helpful to decrease the understanding time (43.5%) and the effort to understand the code entities (39.1%).

In the future, summaries can be produced by using other informal documentation such as mailing lists or stack overflow, etc. Additionally, the approach can be applied in practical settings. Consequently, it can be used within an IDE such as Eclipse to assist developers during their software maintenance and evolution tasks.

Keywords: Code summarization, code entities, Bug report, Machine learning, Informal documentation, Formal documentation

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

ETS	École de Technologie Supérieure
API	Application Programming Interface
LR	Logistic Regression
AST	Abstract Syntax Tree
BR	Bug Report
SC	Source Code
SVM	Support Vector Machine
KNN	K-Nearest Neighbors
TBCNN	Tree-Based Convolutional Neural Network
ARENA	Automatic Release Notes Generator
STSS	Short Text Semantic Similarity
RNN	Recurrent Neural Network
SVO	Subject, Verb and Object
RQ	Research Question

INTRODUCTION

A huge amount of information related to various software projects is produced at daily basis; subsequently, software developers and managers need to deal with this information in order to perform their software tasks. As part of this information, we find the source code. In effect, developers need to understand the code that is part of their tasks and to be able to achieve this goal, they have to read a substantial amount of information coming from different data sources such as source code, documentation, etc. Reading long codes and texts is tedious and time-consuming. As a result, code summarization has been suggested to facilitate and shorten this process. Generally, the most recent works like (Moreno & Marcus (2012); Moreno *et al.* (2013); Ying & Robillard (2014); Moreno *et al.* (2014); Moreno *et al.* (2015); McBurney & McMillan (2016); Armaly & McMillan (2016); Badihi & Heydarnoori (2017)) lack integration with various types of informal documentation and leverage source code for generating summaries of code entities. According to the experts in the field, including informal documentation can be important to augment formal documentation such as the official Java documentation Treude & Robillard (2016); Guerrouj *et al.* (2015). Additionally, researchers have proven that using source code (only) has the problem of lacking completeness and clarity (Treude & Robillard (2016)). In recent investigations, Guerrouj *et al.* (2015), have covered the possibilities of using Stack Overflow for summarizing code entities. Inspired by previous works on code summarization and in order to overcome shortcomings by previous alternatives, we propose an automatic approach that uses machine learning and that leverages informal documentation, in particular bug reports, with the aim of gaining insightful information about code entities from this kind of source of information, bug reports.

To reach this general objective, three different contributions have been achieved: i) Finding code entities in informal documentation using an island parser, ii) Leveraging relevant sources of information to summarize code entities by applying appropriate machine learning tech-

niques, iii) Evaluating our approach in terms of the usefulness of the produced summaries through an empirical evaluation.

CHAPTER 1

RELATED WORK

There are numerous research projects about software summarization. The majority of studies cover formal documentation and source code; while using the informal documentation for code summarization is suffering from insufficient consideration. In our work, we evaluate the effect of using informal documentation like bug reports when summarizating code entities, classes and methods.

This chapter consists of two sections: The first section concerns approaches that use of source code for code summarization, while the second section provides an overview of recent works that attempted to leverage informal documentation when summarizing code.

1.1 Previous works on code summarization

A large body of work has been done in the field of code summarization. For example, Sawant & Bacchelli (2015) focused on the history of different projects to evaluate the usage of different code entities. The focus of their approach was on classes and methods in the Java language and one specific tool: Maven. In our study, we attemot to summarize the purpose and usage of code entities in informal documentation in several open-source Java projects.

Several research studies have used machine learning algorithms for the classification of source code. For example, Phan *et al.* (2017) suggested two models of tree-based convolutional neural network (TBCNN), k-Nearest Neighbors (kNN-TED), and SVM for source code classification. In their work, the input structure of the model is presented as a tree. The Result of their tree-based approach illustrated a great enhancement in the classification performance and execution time. The model was evaluated on 52,000 C programs and the results showed that the tree-based classifiers had the great performance in comparison with the sequence-based or metrics-based classifiers. In our thesis, we also leverage machine learning to select relevant sentences that will be part of summaries.

Moreno *et al.* (2013) introduced a way to produce natural language summaries of Java classes automatically. They proposed three different factors (indicative, abstractive, and generic) for code summarization. They also considered evaluated properties in three parts of adequacy, conciseness and, expressiveness. While previous summarization approaches were only based on the class relationships, they divided class stereotypes into 13 different classes and used responsibility as a factor for summarization. Unlike this paper, that similarly to most recent works, is based on the source code, we leverage informal documentation to produce our automatic code summaries.

To apply a summary of edited parts of source code to version systems and issue trackers, ARENA (Automatic Release Notes Generator) was suggested by Moreno *et al.* (2014). For the evaluation part, they recruited 58 participants for four different empirical studies and the result showed that ARENA automatically produced summaries that were so similar to gold summaries, which were suggested by participants. The inclusion of necessary editing parts is the most fundamental part of the ARENA. This could potentially enable developers to fully understand the latest edits and fixed bugs. On the other hand, it also specifies the current bugs in the source code.

As an example of a different study, an eye-tracking study with 10 professional Java software developers has been conducted by Rodeghero *et al.* (2015) and led to a new word-based finding summarization tool. In the evaluation part, the comparison was between their words and words extracted with VSM tf/idf approach. Differently from this work, our approach is not based on the use of eye-tracking and it handles other types of information using different algorithms.

A new approach for the automatic generation of Java methods was suggested by McBurney & McMillan (2016). They summarized the context surrounding the method instead of using information which is internal of the methods. They designed a new system to produce natural language text. The output of the system is about how to use the method and why this method exists in a particular program. Results show that the suggested approach improves the

quality of summarization. Like this appraoch, we consider the context surrounding the code entities in bug reports. However, we do not leverage source code to summarize code entities.

McBurney *et al.* (2016) found four different summarization tools for Java projects. All tools are selecting sentences to make a feature list. The results of the evaluation part show that none of these tools satisfy expectations and more practices are required to improve tools. As mentioned before, due to the lack of accurate summarization approaches, we contribute to this area by suggesting a novel automatic code summarization approach.

Moreno *et al.* (2015) proposed an approach, called MUSE, to find and extract a method. The results illustrate similarities with what developers have found. Their approach consists of four steps of the client's downloader, the example extractor, the example evaluator and the example injector. The results also present that the MUSE approach helps developers to achieve more complete implementation, which is so useful for their tasks. At the final step, an experiment was conducted to understand how MUSE benefits a developer during his/her task. The suggested approach is limited to methods in the source code.

Ying & Robillard (2014) conducted a research to infer an algorithm for summarization using experienced programmers, while focusing on selection and presentation. their study was conducted by 16 participants who have at least one year of experience in Java programming. Regarding the selection part, some developers considered that method signature should be part of the summaries, while others not. Also, some highlighted that at least two statements of a method should be considered, while most developers suggested to remove exception handling blocks. What parts to include when summarizing code entities is therefore still an open question, which depends on the context, experience of developers, etc. As researchers, our aim is to provide in a short and concise way, relevant information about code entities.

A tool that can classify methods and classes based on the stereotypes and according to their intent in a software system was suggested by Moreno & Marcus (2012). In their approach, methods are divided into 15 different classes and they also defined 13 different categories for classes. Their approach is based on a set of predefined rules. One of the main advantages of

this approach is the fact that developers can evaluate their changes since it keeps track of the history of different stereotypes of classes in design time. Unlike this work, we focus on code entities discussed in informal documentation, and we do not handle the notion evolution.

McBurney & McMillan (2016) conducted an empirical study to evaluate to what extent a summary reader and author agree for what concerns the evaluation part. They used short text semantic similarity (STSS) for this purpose. The findings show that users often use summaries more than authors and that STSS can be leverage to provides an estimation on the accuracy of summarization. Once again, the focus of researchers was on the source code in this work.

Armaly & McMillan (2016) proposed a technique to reuse functions in C and C++ programs. They used execution record and replay technology and empirically evaluated their techniques. Participants agreed on the fact that using the suggested approach is much easier than manually reusing code, and also code is simpler and smaller (by up to a factor of six). Unlike this approach, our work do no investigate reuse.

Badihi & Heydarnoori (2017) proposed a crowd summarizer using crowdsourcing, gamification, and language processing. They implemented a web-based code summarization game in a form of Eclipse plug-in. The results showed that the proposed crowd summarizer is able to extract most critical keywords in comparison with the eye-following methodology in question.

Rodeghero *et al.* (2017) suggested an approach to extract important information about the meetings taken place between developers and clients. They used an artificial dataset, called AM, to this aim. Their approach showed a precision of 70.8%. Differently from this work, we do not summarize conversations but rather code entities discussed in informal documentation.

1.2   Code summarization using informal documentation

Few recent research works have attempted to investigate informal documentation for code summarization:

Trivedi & Dey (2016) evaluated different classifiers and a combination of various classifiers for emails. They examined three datasets: Enron, SpamAssassin and, LingSpam, and multiple algorithms to find the best boosting algorithm on Naive Bayes and Bayesian. The result of their work show that Adaboost is the best algorithm. Additionally, their evaluation of several classifiers illustrates that using a combination of classifiers improves accuracy and decreases false-positive rate. In our work, we use logistic regression for the classification since it has been proven to be among the best for sumamrization tasks (Rastkar *et al.* (2010)).

Rastkar *et al.* (2010) suggested an approach to summarize bug reports. In their study, they compared the effectiveness of different classifiers in terms of precision, recall, and F-measure. They examined and compared three classifiers: Bug Report Classifier (BRC), Email Classifier (EC) and Email and Meetings Classifier (EMC). The results show that BRC is the best classifier among the three different classifiers. For the classification task, they used liblinear and applied logistic regression. Unlike this work, we do not summarize bug reports but rather code entities discussed in bug reports. We take advantage from the findings of this research for what concerns the best classifiers and we therefore apply the logistic regression for our summarization task as well.

Guerrouj *et al.* (2015) investigated the utilization of the context which is surrounded by the code entities in the Stack Overflow. Results illustrated that the approach has an R-Precision of 54%. Similarly to this work, we consider the context around the code entities as an important component in our proposed approach. Although they investigated the stack overflow as a source of information, their results inspired us since informal documentation, in general, can contain important information about code entities.

Nguyen & Nguyen (2017) suggested a new summarization framework, called SoRTESum, which in spite of other methods, uses social information of a web document like tweets from twitter. The summarization consists of two parts: scoring and ranking. According to the results, utilizing social information enhances the quality of summaries. In our suggested work, machine learning is used to give a priority to different sentences to be included in the summary.

Tayal *et al.* (2017) suggested an approach for document summarization that is based on training and SVO (object, verb, and subject) rules and a data processing that involves steps such as sentence combination, NLP parser, sentence reduction, semantic representation, ambiguity removal, and POS tagger. This approach has revealed an F-score that ranges from 0.112561 to 0.4036. The approach has been evaluated with five language specialists and 20 random participants. Unlike this work, we do not investigate text documentation but we aim to summarize code entities trapped in documentation by applying machine learning techniques.

Jiang & McMillan (2017) worked on the automatic generation of short summaries of commits since most developers need concise ideas on commit messages that may be so long sometimes. They compared commits messages from users and automatically generated commits. Results have shown that most produced comments by users (82%) are short (just a line) while the latest automated approaches produced multi-lines messages. The authors studied several ways to shorten commits messages. They used (verb + object) format to generate messages and considered different batches for verbs, which are more important in commits like add, create, and make. Additionally, they applied the Naive Bayes classifier to classify them. Similarly to this work, we leverage machine learning for our summarization task. However, we deal with a totally different problem, that is the summarization of code entities discussed in bug reports.

Treude & Robillard (2016) suggested SISE, a machine learning based-approach for computing the similarity between features of sentences in Stack Overflow and APIs. An evaluation was done with eight developers and achieved an accuracy of 0.64. Results show that Stack Overflow meta-data along with machine learning could be useful for extracting features. Unlike this work, we focus on summarizing code trapped in bug reports.

Panichella *et al.* (2016) suggested an approach that automatically produces summaries for test cases. The goal is to improve the understandability of test cases. The approach consists of a summary aggregation, test coverage analysis, summary generation, and test case generation. Their empirical evaluation's results have shown that developers can find twice as many bugs and that this approach highly increases the understandability of test cases. Unlike this work,

our aim is not to summarize test cases but rather the purpose and possibly of code entities discussed in informal documentation.

CHAPTER 2

TECHNICAL BACKGROUND AND PROPOSED APPROACH

2.1 Technical Background

2.1.1 Overview of machine learning approaches

The automated text classification has seen a huge enthusiasm in recent years. Researches show that using the machine learning techniques is the predominant way to deal with the automated text classification. When machine learning techniques have been used, a classifier is built by learning from a set of pre-classified documents (Sebastiani (2002)). Figure 2.6 shows various machine learning algorithms for different purposes.
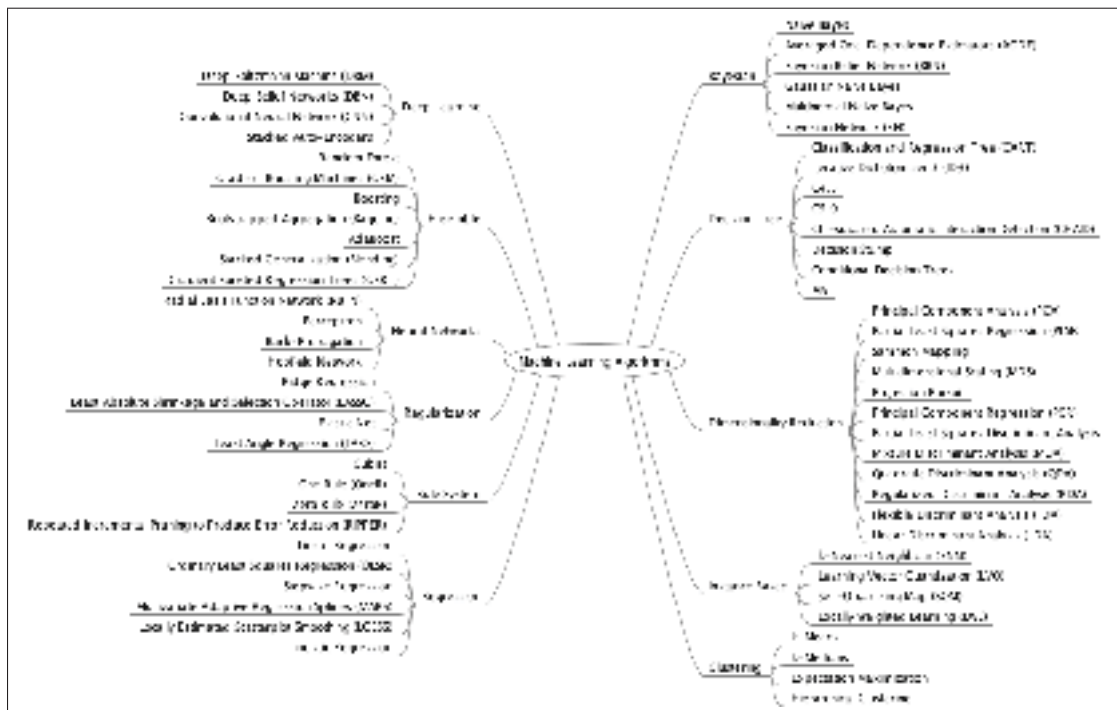
Figure 2.1   Different machine learning approaches (Chen *et al.* (2009)).

To classify sentences into important and non-important classes, various algorithms for the text

classification can be applied. As an example, Lewis (1998) described the Bayesian classifi-
cation as the probability of the document $P(c_j|s_i)$ for each class of c_j by a vector of words
$d_j = w_j^1, w_j^2, ..., w_j^n$ as follows:

$$P(c_j|s_i) = \frac{P(s_i|c_j) \cdot P(c_j)}{P(s_i)}$$

In this context, a text is represented with a |N| dimensional vector of words (Chen *et al.* (2009)).
To tackle the problem of the high dimension of the data vector, the Naive Bayes assumption
was suggested. Naive Bayes classifier also uses Bayes rules for the text classification but it
considers that features are independent of each other. The probability of a sentence s_i for a
class c_j is calculated as:

$$P(s_i|c_j) = \prod_{1=1}^{n} P(w_j^l|c_j)$$

Researches also show that the logistic regression was beneficial for the text classification (Al-
Tahrawi (2015); Rastkar *et al.* (2010)). For example, Al-Tahrawi (2015) examined three algo-
rithms of Support Vector Machine, Naïve Bayes, and Logistic Regression for the Arabic text
categorization. The results show that logistic regression has the highest performance compar-
ing to the other two algorithms.

2.1.2 Logistic regression

Logistic regression (LR) is a statistical algorithm and it provides the probability model for
various machine learning applications. LR estimates the probability that each input x can be
categorized as a class label y.

$$P(y \mid x) = \frac{1}{1 + \exp(\text{-}y\alpha^T x)} \tag{2.1}$$

In the formula above, the alpha is known as the model parameter (Al-Tahrawi (2015)).
For example, in the context of our problem, when detecting informative sentences and words

that are part of the summaries of code entities, when its value is high, the term is considered to be important. The model can be used as a classifier if a threshold is chosen (Antoniol *et al.* (2008)).

In our case, we use the logistic regression because it has been proven from past research on summarization Rastkar *et al.* (2010), that it is one the most suitable algorithms for the summarization task. The output of the logistic regression provided different probabilities for sentences that can be part of summaries of code entities.

Rastkar *et al.* (2010) summarized the whole bug reports, while in this study, we apply logistic regression to produce summaries for the code entities mentioned in bug reports. As an output of the logistic regression algorithm, the sentences will be ranked based on the probability value.

2.2 Proposed Approach

2.2.1 Overview of the approach

For different software projects, a huge amount of information is archiving in various sources of information. To help software developers find code entities easily and to facilitate their work especially when dealing with complex and–or large software systems, we suggest a novel summarization approach that unlike past research, leverage bug reports to summarize the purpose of code entities trapped in bug reports. This approach can be applied using other sources of informal documentation such as Stack Overflow, emails, etc. (Figure 2.2).

In existing studies, summarization is effective in many applications such as code change, duplicate bug detection, bug report digestion, traceability link recovery, document generation, summary visualization, source-to-source summaries, etc. (Nazar *et al.* (2016)).

In this chapter, we describe our proposed methodology for building an automatic context-aware summarization approach that summarize code entities discussed in informal documentation. Our methodology consists of three main steps:

- Step 1: an island parser to extract code entities in informal documentation is utilized.
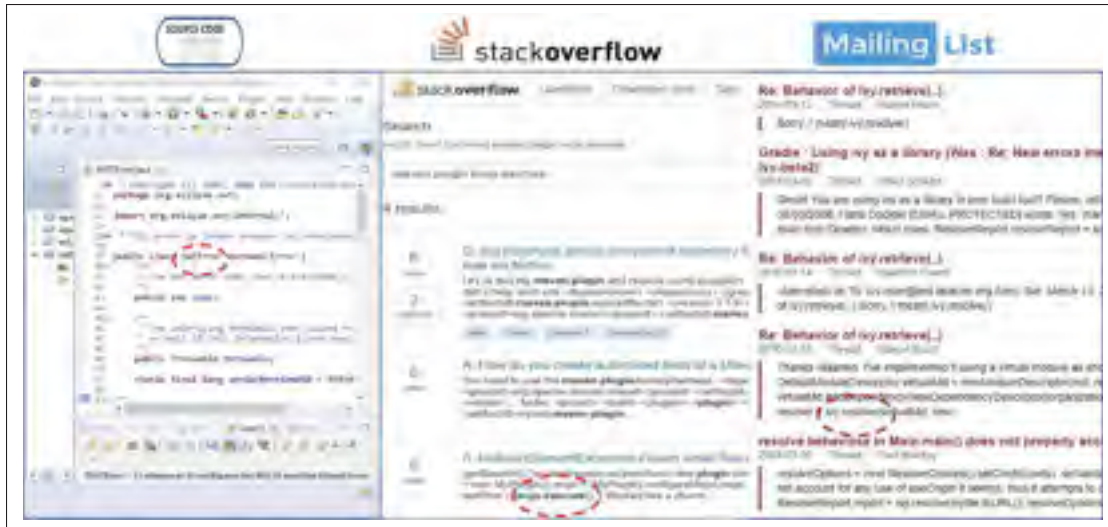
14

Figure 2.2   Using different sources of information for the code entities summarization.

- Step 2: a novel machine learning based approach is applied.

- Step 3: the usefulness of the novel approach is empirically evaluated through an empirical study. The evaluation step assesses whether the summaries provided by the new approach are pertinent to software developers or not (Figure 2.3).
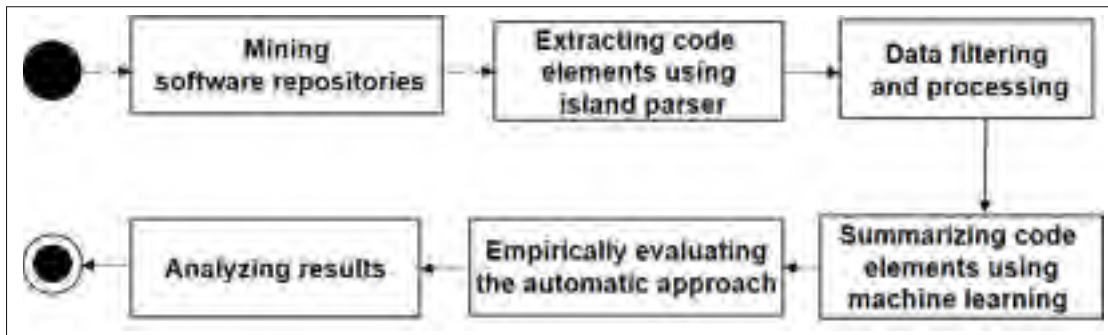

Figure 2.3   Overview of our proposed methodology.

2.2.2 Mining and filtering the data

The first step in the methodology is related to mining and filtering of the data. Figure 2.4 shows an example of report from the bug reports of Eclipse. In the suggested approach, the required data is extracted from the page source of the various bug reports. To this aim, the page source of reports is searched and data is extracted by using tags on each page. Figure 2.5 shows the details of different tags in the page source of one example from Eclipse's bug reports. As it can be noticed, information such as bug id, the title of the report, date of report, etc. can be extracted from the page source.

Figure 2.4   An example of software artifact : Eclipse bug report.

Due to the structure of the aforementioned documentation, different extraction strategies have been considered for mining bug reports of the different software systems that we dealt with. Figure 2.6 shows the details of the report in the bug report. As we can see, the summary of the report which mostly includes useful information is specified. Figure 2.6 shows the details of the report in the bug report. In this step, all needed information is extracted by using the specific tags of each value.

Figure 2.5   An example of page source in bug reports.

2.2.3   Extracting code entities discussed in bug reports

In this study, code entities are extracted from the bug reports using an island parser that iden-
tifies terms like code in informal documentation. Indexes of each extracted class and method
are kept to identify code entities for which we generate summaries. Figure 2.7 shows an exam-
ple of code entities: *org.netbeans.swing.tabcontrol.TabbedContainer.paint* is a fully qualified
name utilized in different positions of the bug report. To produce a proper summary for the
aforementioned code entity all the related posts in the bug report will be considered.   We
should also consider that one big challenge in documentation processing is that most docu-
mentation is not well organized. For example, many sentences are not complete or some users
put URL addresses in their sentences. To tackle this problem, we considered a pre-processing
step and all unusable words and parts like URLs have been removed.

After that, we extract code entities in informal documentation using an island parser to pro-
vide proper input for our classifiers. By using island parser, we parse structured information in
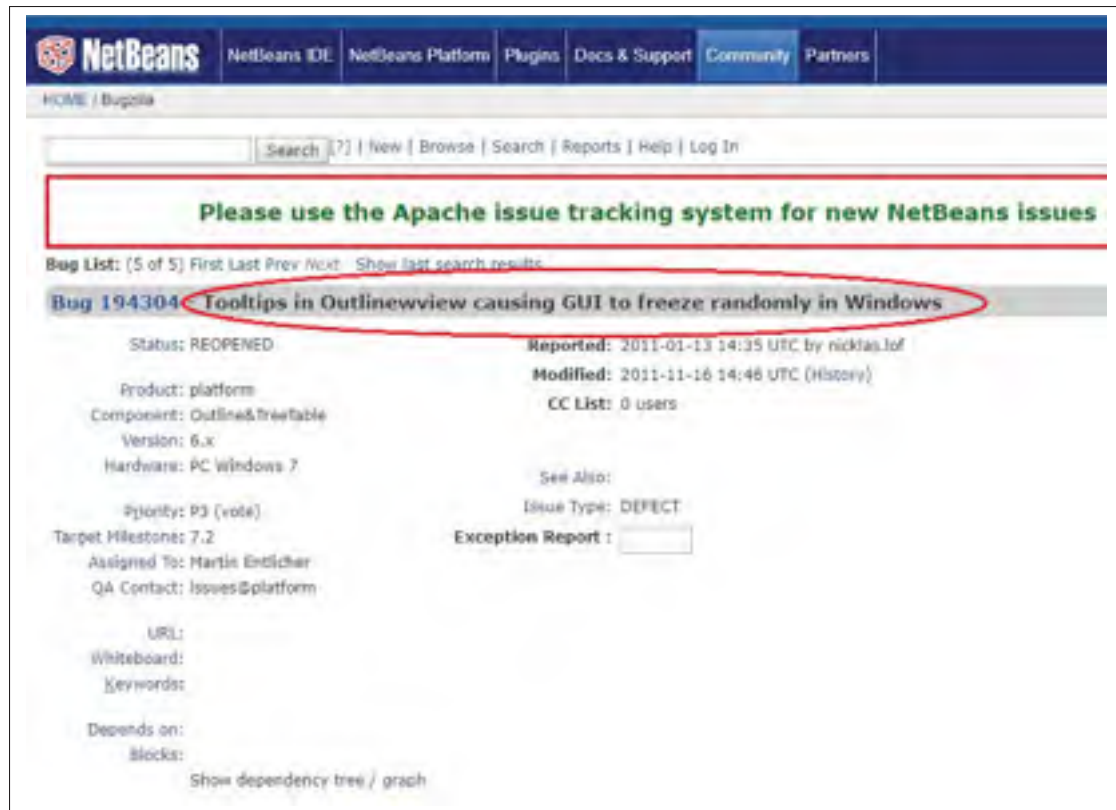
Figure 2.6   An example of summarized title in the bug report.

natural language. Our island parser is an implementation of Bacchelli *et al.* (2010b) approach that we have customized for bug reports. In this approach, the authors identified code in emails by applying naming conventions and using regular expressions to recognize methods, which seems to be an effective technique (Bacchelli *et al.* (2010a)).

Figure 2.8 shows an example of a bug report extracted from Bugzilla[1] for the Eclipse system. As it can be seen, the bodies of the extracted reports include HTML tags. We have removed all tags and cleaned versions of comments are stored in the comments table (Figure 2.9). The pk field in the figure is the key of each comment and the report-pk is the key of each related report.
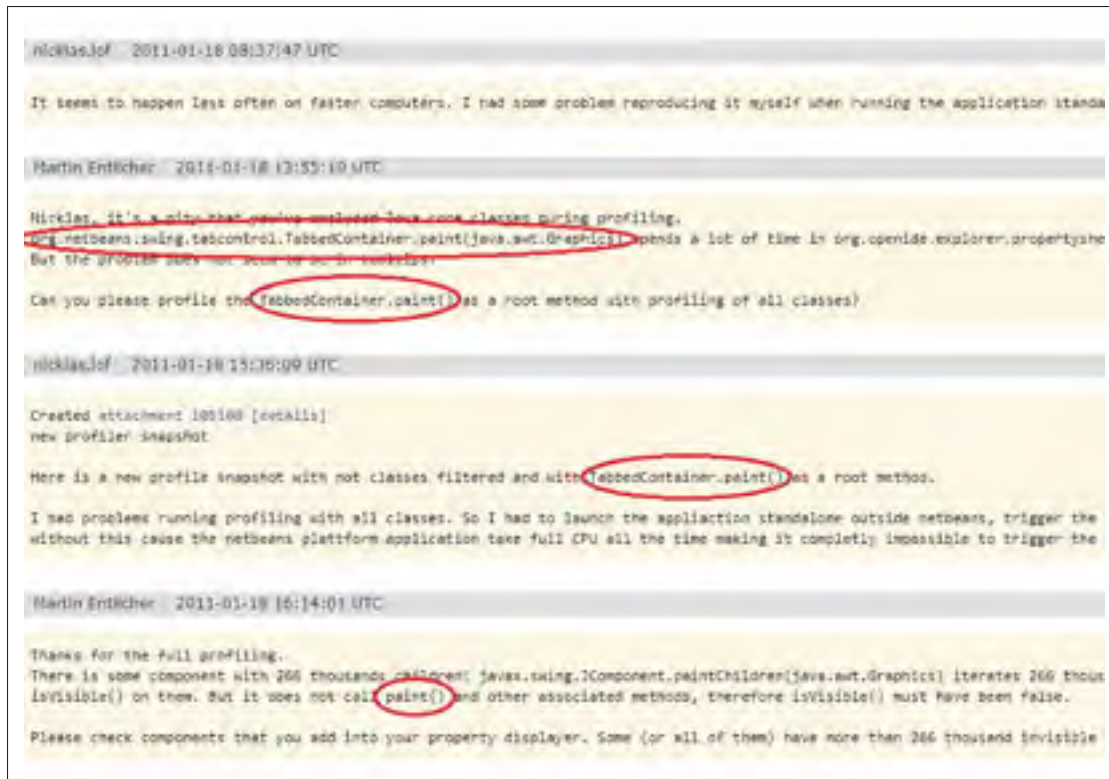
---

[1]   https://www.bugzilla.org/

Figure 2.7   An example of code entities in a report.

Figure 2.8   Sample of extracted bug reports.

Figure 2.9   Sample of extracted comments in bug reports.

Since the main focus of this research is related to code entities, all the code entities in comments are found (Figure 2.10). The 'start' and 'end' fields in the figure are the starting and ending positions of the code entities. The comment_pk is the key of the related comment and the matched_text column shows the code entities. Table 2.1 shows a sample of extracted code entities by the developed island parser. Each report itself has a sentence as a title in bug reports. The titles of bug reports have also been stored in separated fields because they include significant information about code entities and thus they will be considered as important sentences.

pk	comment_pk	start	end	matched_text
511401	142419	141	151	pageChange
511402	142419	207	217	getAdapter
511403	142491	468	475	startup
511404	142491	1314	1321	startup
511405	142515	27	43	getExpandedState
511406	142517	215	237	collectVisibleElements
511407	142517	391	401	result.add
511408	142517	419	438	tv.getExpandedState
511409	142517	457	479	collectVisibleElements
511410	142517	488	502	cp.getChildren
511411	142522	47	69	collectVisibleElements
511412	142522	231	241	result.add
511413	142522	261	280	tv.getExpandedState
511414	142522	301	323	collectVisibleElements
511415	142522	332	346	cp.getChildren
511416	142543	11	23	showViewMenu
511417	142584	44	88	org.eclipse.core.resources.IFile.setContents
511418	142584	1122	1171	org.eclipse.core.internal.filesystem.Policy.error
511419	142584	1192	1241	org.eclipse.core.internal.filesystem.Policy.error
511420	142584	1262	1339	org.eclipse.core.internal.filesystem.local.LocalFile.checkTargetIsNotWritable
511421	142584	1363	1421	org.eclipse.core.internal.filesystem.local.LocalFile.mkdir
511422	142584	1446	1500	org.eclipse.core.internal.localstore.BlobStore.addBlob
511423	142584	1524	1583	org.eclipse.core.internal.localstore.HistoryStore2.addState
511424	142584	1611	1679	org.eclipse.core.internal.localstore.FileSystemResourceManager.write
511425	142584	1720	1780	org.eclipse.core.internal.resources.File.internalSetContents
511426	142584	1800	1852	org.eclipse.core.internal.resources.File.setContents

Figure 2.10   Sample of extracted code entities.

Table 2.1   A sample of the extracted data using our developed island parser.

Bug Report ID	Title of the bug report	code entity
1917	Multi-page editor support for separate property and outline pages	pageChange
11233	Add error icon to InputDialog class	IInputValidator.isValid
22782	Need API to draw disabled text in native platform way	GC.drawText
2713	Feature request: API to traverse tree viewer	CP.getChildren
20054	Unable to register ruler context menu for MultiPageEditorPart	AbstractRulerActionDelegate. setActiveEditor
28306	Data loss when disk is full	org.eclipse.core.internal. filesystem.Policy.error
28317	Support for blit operations within Image Data	SWT.error

Finally, the number of comments and the number of existing code entities in each report are stored as comments_num and findings_num fields in the database. By using this table, all reports without a code entity will be not considered as part of the input of the machine learning technique (Figure 2.11). Since each bug report consists of a huge amount of comments and data, by sorting comments of bug reports based on the number of found code entities, only reports with at least one code entity will be used.

pk	report_pk	comments_num	findings_num
167710	28293	16	2
167711	28294	10	0
167712	28295	22	0
167713	28296	8	0
167714	28297	7	0
167715	28298	8	0
167716	28299	9	0
167717	28300	9	2
167718	28301	12	0
167719	28302	13	11
167720	28303	10	0
167721	28304	9	1
167722	28305	13	0
167723	28306	10	0
167724	28307	23	97
167725	28308	19	0
167726	28309	3	0
167727	28310	10	1
167728	28311	5	2
167729	28312	7	9
167730	28313	6	0
167731	28314	3	0
167732	28315	8	1
167733	28316	8	19
167734	28317	25	0

Figure 2.11   An example showing the number of code entities
found for Eclipse.

2.2.4   Code Summarization using machine learning

Machine learning techniques and its applications have been widely considered by researchers in various fields both in academia and industry. A vast application of machine learning algorithms related to pattern recognition, image processing, text mining, etc. has made these algorithms crucial in science and industry-related projects. In our work, a new approach that summarizes code entities discussed in bug reports is implemented. The approach uses the code entities trapped in documentation as the input for the machine learning techniques.

Using binary classification is one way to produce summaries by distinguishing between significant and insignificant sentences. To implement binary classification, supervised or unsupervised techniques can be used and it should be considered that each mechanism has its advantages and disadvantages. To illustrate, while in unsupervised learning, classification is based on the input document and there is no obligation to provide a predefined annotated corpus, the

supervised technique needs an annotated corpus (Rastkar (2013)), but the advantage of choosing supervised techniques is to specify various features which are not possible in unsupervised mechanisms (Trivedi & Dey (2016)).

In recent years, various summarization techniques have been suggested. Some of the popular summarization techniques are graph-based approaches, machine learning based-approaches, cohesion based-approaches, etc.

Summary production techniques can be abstractive and extractive. Produced abstractive summaries need natural language generation techniques and require to rewrite the sentences. In extractive summaries, sentences would be selected and sentence rewriting is not needed (Khatri *et al.* (2018)). In our work, extractive summaries are produced.

To rank sentences in the produced summaries, we applied the logistic regression classifier. The logistic regression algorithm belongs to the supervised learning algorithms. For each code entity, all related sentences are extracted and prioritized based on the probability of being a part of a summary. The logistic regression classifier gives a probability instead of zero or one value (Rastkar (2013)). The suggested approach provides an extractive summary, keywords in the sentences are selected based on their priority.

To train the statistical classifier, a corpus of important and non-important sentences sampled from bug reports is made, and each sentence is labeled based on its importance. One problem in this step is related to the existence of code snippets. Researches show that users of various sources of information like Stack Overflow are using <pre><code>...</pre></code> tag for code snippet in posts. Therefore to tackle this problem, we removed the code snippets parts (Ahasanuzzaman *et al.* (2018)) since our goal is not to summarize code snippets but rather classes and methods trapped in the natural language parts of bug reports. Once we filtered and cleaned our data and kept only the code entities and natural language text that surround them, we applied the following steps:

2.2.4.1 Creation of the corpus for code summarization

To be able to train our applied classifier to learn the model to classify important and non-important sentences to be included in our final descriptions of code entities, a corpus of labeled sentences is needed.

Figure 2.12   Sample input as sentences for the machine learning technique.

Since the island parser provides a text for each code entity that includes both important and non-important sentences, extracting sentences and specifying proper classes becomes crucial. The title field of reports already saved in our database is added as the first sentence of the body. Thereafter, one program has been developed by C# to assign the proper class to each sentence. The first sentence of each text which is the title of each bug report is considered as an important sentence and we give it label 1. The second important class of sentences are those sentences surrounding classes and–or methods in bug reports. Our definition of important sentences was inspired by previous works (Dagenais & Robillard (2012); Guerrouj *et al.* (2015)) that

considered term proximity and local contexts when dealing with summarizing code trapped in informal documentation.

Figure 2.12 shows a sample of the initial corpus. The fields api_pk, sentence, matched_text, and calassname are respectively the keys of the table, the extracted sentences, the code entities, and the label of each sentence. The initial corpus incorporates conversational data which does not bring useful information, and which can take time to read. Additionally, sentences are presented in the form of a set of keywords. Since our goal in this thesis is to produce summaries in the form of keywords, we have removed all useless words including stop words and informal words such as the names of developers, greetings-related words, etc. Figure 2.13 shows an example of a corpus. The matched text field shows the code entities name and the class field has a binary value of zero for non-important sentences and one for important sentences. Unlike past research Rastkar *et al.* (2010) that summarizes entire bugs, we created a corpus for each code entity since each class or method is discussed in a specific set of bug reports, which constitutes its context. Table 2.2 shows the number of sentences with labels for each studied project.

Table 2.2   Characteristics of the corpus of each studied project.

Project	#Sentences	#Important sentences	#Not-Important sentences
Eclipse	71,957	16,275	55,682
NetBeans	9,400	2,275	7,125
KDE	5,799	1,716	4,083
Apache	17,614	3,318	14,296

Since some sentences consist of many conversational forms which are not helpful, to provide one abstract explanation about the code entities, a list of useful keywords has been produced instead of sentences. An example of a list of keywords is presented in figure 2.13.

Figure 2.13 An example of input of our machine learning-based approach.

2.2.4.2 Training the logistic regression model for the classification task

To train a model to recognize a sentence that will be part of the produced summaries, we apply machine learning. Firstly, to be able to work with WEKA (Holmes *et al.* (1994)) to train the model, the data from the corpus is converted to the ARFF (Attribute-Relation File Format), which is actually the input format of WEKA. The document text is presented as a "string" attribute and the document class is considered as a "nominal attribute" (le Cessie & van Houwelingen (1992)). In our work, we also deal with two attributes, that are the text and class attributes. The first attribute is the text attribute which, in our case, includes the sentences from the corpus of the code entity in question. The second attribute consists of the label of classes, which has two values, zero or one.

In the training step, sentences are considered as the input for the used classifier. The "String-ToWordVector" in filter of WEKA is applied to transform the string attributes to a number representation.
We have used "StringToWordVector" (weka.filters.unsupervised.attribute.StringToWordVector) filter; it is an unsupervised filter that has options for the binary occurrence of words to create a bag of words. The dictionary of words is formed using the data of the training set.

Thereafter, we performed a five folds cross validation to split our data into test and training data sets. After training the model using the logistic regression, the evaluation class (weka.classifiers.Evaluation) is utilized for the evaluation of the machine learning model.

Based on the probability value of each sentence, all sentences will be prioritized. There are different approaches for choosing the number of sentences that will be included in the final summary. To give an example, McBurney & McMillan (2016) selected six sentences to generate code summaries. The first six sentences with higher probabilities have been chosen as important sentences. Rastkar *et al.* (2010) selected sentences for the summary up to reach 25% of the bug report word count because the percentage is so close to the value of their gold standard summaries sentences. In our case, we selected all sentences because most of our produced summaries were short. We therefore believe that considering all generated ranked sentences, ordered based on their importance will be not overwhelming and help gain a better understanding of code entities.

2.2.4.3 Feature selection

In various sources of information, we can define different groups of features. For example in Stack Overflow, features can be considered as title features, body features, and answer features. For instance, for the title features, the title of the informal documentation can be considered as an important class if it includes the words of code entities or "issue", "bug", "error" and "exception" word (Ahasanuzzaman *et al.* (2018)).

In text classification, each text can be used as a bag of words and words in sentences would be the features. Therefore, the sentences will be converted to the vector of words (Boulis & Ostendorf (2005)).

In the suggested approach, the presence of code entities in sentences is considered as features. Therefore, the lexical features which are about the occurrence of code entities are important in our project. In our study, we will leverage the sentences that include the code entities and those that surrounded it, as in recent works like (Dagenais & Robillard (2012); Guerrouj *et al.*

(2015)), these sentences represent useful information for summaries. To simplify, below is the list of features in the suggested approach.

- The occurrence of code entities (they are being recognized by the methods and classes patterns using regular expression and naming conventions).

- Is the sentence a title of a bug report or not?

We remove stop words since we want to provide summaries with important keywords and words like "and," or "a," are nor useful neither informative.

2.2.4.4 Sentence classification

In our work, we first apply the logistic regression algorithm to the set of sentences related to each code entity to be summarized, the output provides different probability values for each sentence. Then, sentences will be ranked based on the probability values in a form of a list. For instance, the first sentence in the composed list would be the most important sentence for the summary. In Table 2.3, we show a sample of produced summaries for various code entities.

Some machine learning techniques such as decision tree algorithms prefer to work with discrete attributes. Discretization is the way toward changing over a real-valued attribute into an ordinal attribute. Discrete attributes depict nominal attributes category and nominal attributes have an ordering of the variables (Eibe *et al.* (2016)).

For the implementation of logistic regression, an instance filter can be used to discretize a range of attributes in the dataset into nominal attributes. Discretize is an unsupervised filter in Weka (Eibe *et al.* (2016)).

One problem related to data is informal discussions in bug reports. Many sentences include conversational words that are not informative at all. Also, some contains a lots of informal words exchanged between developers. Therefore, to obtain summaries with better quality, many uninformative words, such as stop words and informal ones have been removed.

Table 2.3 A Sample of produced summaries by the
proposed approach.

Project	code entity	Summary
Eclipse	BrowserInformationControl.isAvailable	"hovering default information control wrapping error tabs"
Eclipse	Dispaly.asyncExec	"application window active runnable runs immediately problems display async exec partly fixed suppose problem inactive application window"
NetBeans	ALD.atomicLock	"extract document manipulation editorlib"
NetBeans	EntityManager.joinTransaction	"solved moding eclipse link target database property file correctly references database"
KDE	linkProviders.add	"bluetooth device link"
KDE	ViewportParamsTest.cpp	"optimized returns line string holds detail level information optimized returns line string holds"
Apache	directory.mkdirs	"task apache ant discards parts files"
Apache	parser.parse	"entered comp ld xml document created passed tree walker sparse filter works exponential times"

2.3   Limitations of the developed approach

Although the produced summaries can provide valuable information about the code entities,
the process of creating summaries faced some challenges.

We observed similar summaries for some different code entities. The source of this problem
is originated from the fact that developers mention several code entities when discussing one
that perplex them. Since our approach considers as a context, the sentences that surrounds a
code entity, the context is the same for code entities discussed together, which results in similar
or almost similar summaries for different code entities. Figure 2.14 shows an example of this
phenomenon.

Figure 2.14　An example with different code entities discussed in the same report.

Another challenge is related to the amount of handled data for machine learning. In effect, to provide significant results, machine learning algorithm needs a significant amount of data, which is not always the case in our context since some code entities are discussed in very few bug reports. Additional studies are needed to investigate whether other sources of information such as emails and Stack Overflow posts, etc. contain a high number of discussions of code entities in comparison with bug reports. If this is the case, the same approach can be applied to another type of information that may lead to better performances in terms of the quality of the produced summaries for each code entity, or a combination of different types of information may be also considered. Overall, we could gain some valuable information about code entities, their purpose from bug reports. However, this information can be enriched by the use of other sources of information to yield to better results.

CHAPTER 3

EMPIRICAL EVALUATION

During the final step of the methodology, the summary provided by the novel approach has been examined to assess its usefulness for software developers. The main goal of the novel approach is to generate an accurate code entity summary for developers to assist them during their maintenance and evolution tasks.

To investigate the accuracy of our machine learning based summarization approach, we evaluated our approach based on the usefulness via an online survey with 23 human participants.

The evaluation of effectiveness of the proposed approach is the main goal of this section. This section consists of three different parts: (i) definition, (ii) design and (iii) analysis method for the experiment. Firstly, we describe the definition of the experiment based on Basili *et al.* (1994). The second part is about the experiment design and procedure, participants, etc. Finally, in the third part, we present the analysis method.

3.1 Definition and planning

In this part, the objective, context, and object of the experiment are presented. Data sources for the experiment are four open-source projects: Eclipse[1], NetBeans[2], KDE[3], and Apache[4].

The *objective* of the experiment is to evaluate the effectiveness of the summarization approach for summarizing methods and classes in bug reports.

The *context* consists of 23 participants, students from the Software Engineering and IT Department of École de technologie supérieure, as well as software developers from industry. Prior to running our survey, we applied for ethics committee's approval to conduct such type of research. Once, we had their approval, we invited participants by email, while attaching the consent to the invitation. Participants who accepted to participate, filled in the consent

---

[1] https://bugs.eclipse.org/bugs/

[2] https://netbeans.org/bugzilla/

[3] https://bugs.kde.org

[4] https://bz.apache.org/bugzilla

and signed it. Then, they received a follow-up email from us containing three distinct questionnaires. The pre-questionnaire is to collect demographic data, participants' characteristics, background, level of education and programming experience, etc. The questionnaire which contains all the tasks. And the post-questionnaire to gain insights and feedback about the study. There was no restrictions on the time allocated to the experiment. It was compulsory however that participants have a basic programming knowledge and skills to be able to deal with code and informal documentation.

The *object* of the experiment consists of 25 different classes and methods chosen from four different open-source Java projects (Eclipse, NetBeans, KDE, and Apache).
The *quality focus* is the accuracy and usefulness of the automatically produced summaries.

3.2 Research questions

By analyzing the data collected from participants in the survey, we asked two main questions:

RQ: Can our context-aware automatic summarization approach be useful for software developers?

3.3 Context of the experiment

The context of our experiment consists of code entities and bug reports from four open-source projects:

- Eclipse

- NetBeans

- KDE

- Apache

The table 3.1 represents the characteristics of the studied projects. We present, for each analyzed project, the number of commits, the number of sub-projects it contains, the number of lines of code, as well as the total of bug reports.

Table 3.1   The characteristics of the studied projects.

Projects	#Commits	#Projects	#Lines of code	#Bugs
Eclipse	2,000,000	75	68,100,000	88,950
NetBeans	535,000	-	6,000,000	65,187
KDE	2,000,000	-	10,000,000	138,715
Apache	3,022,836	350 +	1,058,321,099	18,890

We extracted, using our island parser code entities from Eclipse[5], NetBeans[6], KDE[7], and Apache[8] bug reports. The prototype of the island parser has been developed by one of the lab members as part of the lab's infrastructure and we customized it to parse data of the aforementioned projects.

The characteristics of the four examined open-source Java projects in terms of the number of code entities are shown in table 3.2. We also extracted the name of the authors, the report date and the position of code entities from in the bug reports since we needed this information for the machine learning part. As an example, 88,950 comments of the Eclipse bug reports have been considered, and 152,738 code entities have been extracted from the comments of the bug reports (Cf. Table 3.2).

---

[5] https://eclipse.org.

[6] https://netbeans.org.

[7] https://kde.org/.

[8] https://www.apache.org/.

Table 3.2   Characteristics of the studied projects in terms of the
number of bugs and code entities.

Bug Report Name	#Reports	#code entities
Eclipse	88,950	152,738
NetBeans	65,187	59,466
KDE	138,715	93,110
Apache	18,890	26,703

We have examined 25 different code entities from the four different Java projects (Eclipse, NetBeans, KDE, and Apache). The selection of code entities was as follows: We first formed a pool of code entities that excludes code entities with empty summaries since we cannot evaluate the empty summaries. Then, we excluded code entities for which similar summaries have been produced. As explained earlier in the thesis, these code entities are those co-occurred frequently with a set of other code entities. We consider that our approach is still not equiped and mature enough to deal with such complicated cases. This is a challenge that we are still investigating and will be addressed as part of our future work. From these pool that excludes particular cases, we randomly selected 25 code entities for examination.

Based on the number of participants we had, that is 23 participants who confirmed their participation, we constituted five different groups of participants. Each group acting on five different code entities.

Tables 3.3, 3.4, 3.5, 3.6, and 3.7 provide the lists of code entities on which each group has worked. For each group, one of the four projects has two code entities. Table 3.3 presents the five code entities on which the first group has worked.

The five code entities provided to the second group are presented in table 3.4. code entities have been selected from different packages and two code entities are considered form the Eclipse project.

Table 3.3   Code entities examined by Group 1 of participants.

Project	code entities
Eclipse	BrowserInformationControl.isAvailable
NetBeans	ALD.atomicLock
NetBeans	org.netbeans.swing.tabcontrol.TabbedContainer.paint
KDE	linkProviders.add
Apache	directory.mkdirs

Table 3.4   Code entities examined by Group 2 of participants.

Project	code entities
Eclipse	Dispaly.asyncExec
NetBeans	EntityManager.joinTransaction
Eclipse	Collections.synchronizedMap
KDE	ViewportParamsTest.cpp
Apache	parser.parse

Table 3.5 represents the five code entities on which group number 3 has worked. As it can be noticed, the selected code entities can be fully or partially qualified names.

Table 3.6 shows the selected code entities for group 4 where the Apache project has two code entities in this group.

The code entities for the last group are presented in table 3.7. The last three participants worked on the five code entities because we had 23 participants.

Table 3.5   Code entities examined by Group 3 of participants.

Project	code entities
Eclipse	column.pack
NetBeans	mockery.mock
Apache	PDFRenderer.renderText
KDE	window.navigator.userAgent.indexOf
Apache	evaluator.evaluateFormulaCell

Table 3.6   Code entities examined by Group 4 of participants.

Project	code entities
Eclipse	resource.createMarker
NetBeans	DatabaseRuntime.managesRuntimeStatus
Apache	ExceptionUtils.handleThrowable
KDE	node.nextSibling
Apache	XSSFFormulaEvaluator.evaluateAllFormulaCells

3.3.1   Design of the experiment

We produced five various tasks of code entities for five groups and each participant should perform each task based on the dedicated group.

We followed a Block Randomized Design when designing our study. In effect, we have blocked participants based on their knowledge of Java and experience, which resulted in groups such

Table 3.7   Code entities examined by Group 5 of
participants.

Project	code entities
Eclipse	resource.createMarker
NetBeans	DatabaseRuntime.managesRuntimeStatus
Apache	ExceptionUtils.handleThrowable
KDE	node.nextSibling
Apache	XSSFFormulaEvaluator.evaluateAllFormulaCells

Table 3.8   Experimental design.

	code entity1	code entity2	code entity3	code entity4	code entity5
Task1	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$
Task2	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$
...	...	...	...	...	...
Task5	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$	$\text{rnd}(M_i/C_j)$ $\text{rnd}(P_k)$

M_i Represents the method 1<i<5

C_j Represents the class 1<j<5

P_k Represents the project 1<k<4

as a group of participants with a high level of experience in Java, a group of participants with
a basic level of experience in Java, etc. Then, we have randomly selected from each block
equal (whenever possible) proportions of participants and generated new groups where we

made sure there is no bias due to a dominant characteristic of participants. To perform a block randomized design study, we asked participants to fill in a pre-questionnaire (along with the consent) when we sent them the invitation email the first time. This enabled us to create our blocks and generate our final groups of participants. Table 3.8 shows the experimental design that we have followed along with the given tasks. In this table, the parameters M_i, C_j, and P_k represent respectively the method, class, and project. The function rnd in $rnd(M_i/C_j)rnd(P_k)$ means that, for each task, we randomly selected methods and classes in the examined projects. As an example, the cell for the code entity 1 and Task 1 has the $rnd(M_i/C_j)rnd(P_k)$ value. It means that the first selected code entity for the first task can be method i or class j from project k.

3.3.2 Pre-questionnaire

Participants start the experiment with the pre-questionnaire form, which includes information about their background, expertise, and experiences in programming. Table 3.9 shows the questions asked and information gathered from the pre-questionnaire:

3.3.3 Questionnaire

Each participant evaluated summaries of five unique code entities in the experimental questionnaire. Therefore, we have 25 distinct summaries for 25 code entities. The set of experimental tasks of each participant has been presented in the form of a questionnaire, which we refer to as the survey questionnaire. This questionnaire consists of five questions on the quality of produced summaries. The quality of summaries has been measured in terms of criteria including conciseness, relevance, and understand ability. We were inspired by previous work like Moreno *et al.* (2013) who used these criteria to evaluate their summaries. The main questions related to these criteria and asked in the questionnaire are as follows:

— Is this description (produced summary) accurate?

— Does this description contain all the information about the class/method?

Table 3.9   Sample of pre-questionnaire questions.

ID	Question
1	Gender
2	The age range
3	How many years of active programming experience do you have?
4	What is your level of expertise in the Java programming language?
5	Please select all the degrees you have and are currently enrolled in.
6	Current positions (student, working in industry, etc.)
7	How many years of work experience do you have in the industry?
8	Are you familiar with using bug reports?
9	Have you contributed (code and/or documentation) to an open source project?

– Does this description contain only the necessary information about the class/method?

– Does this description contain information that helps understand how to use the class/method?

– Does this description contain information that helps understand the implementation of the class/method?

– Is this description easy to read and understand?

We provide in the appendix, an example of the questionnaire provided to participants.

3.3.4   Post-questionnaire

After completing the pre-questionnaire and the questionnaire survey, participants have been invited to answer the post-questionnaire to obtain additional information about the collected data during the experiment (Cf. Table 3.10).

Table 3.10   Sample of post-questionnaire questions.

ID	Question
1	Are you familiar with the following projects?
2	Can the given descriptions be used in the context of any software maintenance and evolution tasks to help reduce the time for developers?
3	Can the given descriptions be used in the context of any software maintenance and evolution tasks to help reduce the efforts of software developers?
4	Do you find bug reports useful to understand classes/methods?
5	Which part(s) of bug reports did you find the most useful?
6	Overall, how difficult did you find the study?
7	What is your idea about the results of survey?
8	Comments

3.3.5   Participants

We conducted a study involving software developers and graduate students. Students are from École de technologie supérieure, while developers belong to various companies such as Informatics Services Corporation, TID Development Co, F. Ménard, etc. Table 3.11 shows the main characteristics of the participants.

3.4   Analysis method

Participants judged the quality of the produced summaries, based on the above-mentioned criteria, using on a 4-point Likert scale (Joshi *et al.* (2015)). The collected data were then analyzed by producing charts related to each question. We believe that at these level, we do not need advanced statistical tests since we seek a general overview of how useful would be

Table 3.11   Characteristics of the participants.

Characteristic	Level	Number of Participants
Program of Studies	Bachelor	14
	Master	16
	Ph.D.	4
	Post-doc	1
Number years of programming experience	Less than 1 year	1
	Between 1 and 2 years	4
	Between 3 and 6 years	10
	Between 6 and 10 years	3
	More than 10 years	5
Number years of industry experience	None	4
	Less than 1 year	3
	Between 1 and 2 years	4
	Between 3 and 6 years	6
	Between 6 and 10 years	3
	More than 10 years	3
Expertise in Java	Poor	4
	Basic	10
	Good	5
	Very Good	4
Using bug reports	Yes	15
	No	8
Open source projects contribution	Yes	9
	No	14

the produced summaries instead of how accurate are. Once our approach would be mature enough, we will compute other measures such as the precision, recall, and F-measure and perform advanced statistical tests for comparison purposes.

CHAPTER 4

RESULTS AND DISCUSSION

In this chapter, we first present the results obtained from the analysis of the data collected. Then, we identify the threats to the validity of our study. Thereafter, we discuss future works, and finally we conclude our work.

4.1 Findings of our study

In the following we present the results obtained from the collected data in terms of the criteria that we have used as measures to quantify the usefulness of our approach.

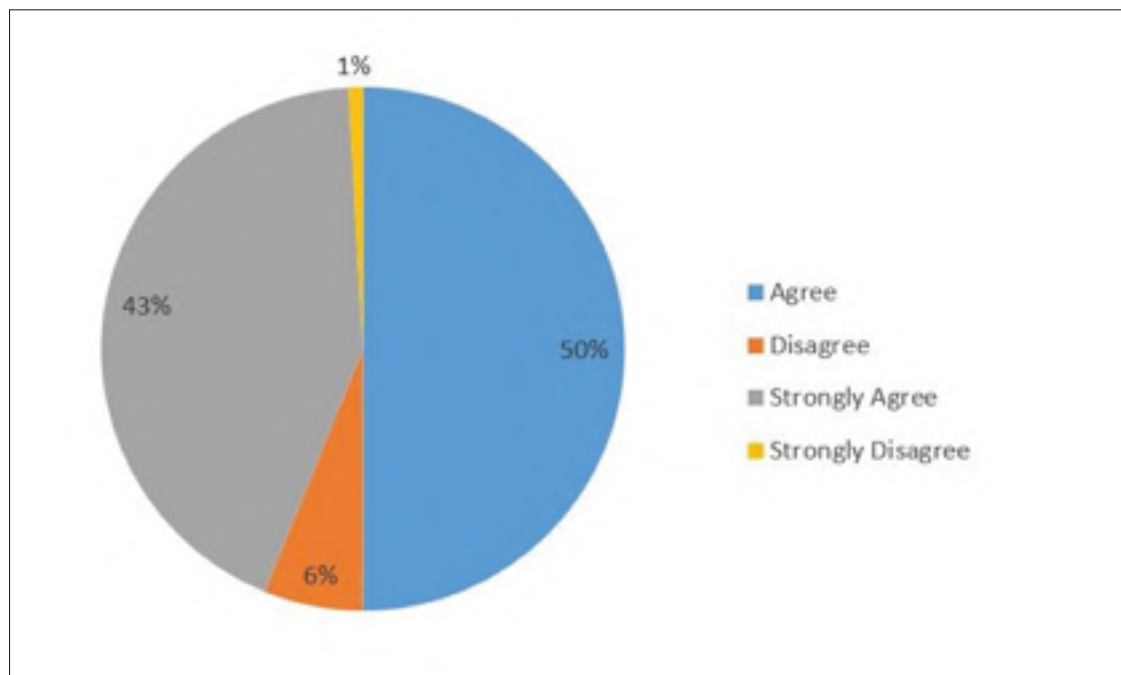### 4.1.1 Accuracy of the produced summaries

Figure 4.1   Participants answers to the question "Is this description accurate?"

Figure 4.1 presents the feedback of participants about the accuracy of the produced summaries. As it can be noticed, the results show that participants mostly agree that the automatically produced summaries for the code entities they evaluated are accurate.

4.1.2 Conciseness of the produced summaries

Figure 4.2 shows the answers of participants to the question of whether the provided summaries contain all the information about the classes or methods.

The chart indicates that only 32% of participants strongly agree that the produced summaries contain all the information. Therefore, our automatically produced summaries still needs to be improved to include all pertinent information about classes and methods. It suffers from this problem because some bug reports are really almost empty and do not contain information about the purpose of a code entity even thought they mention it.
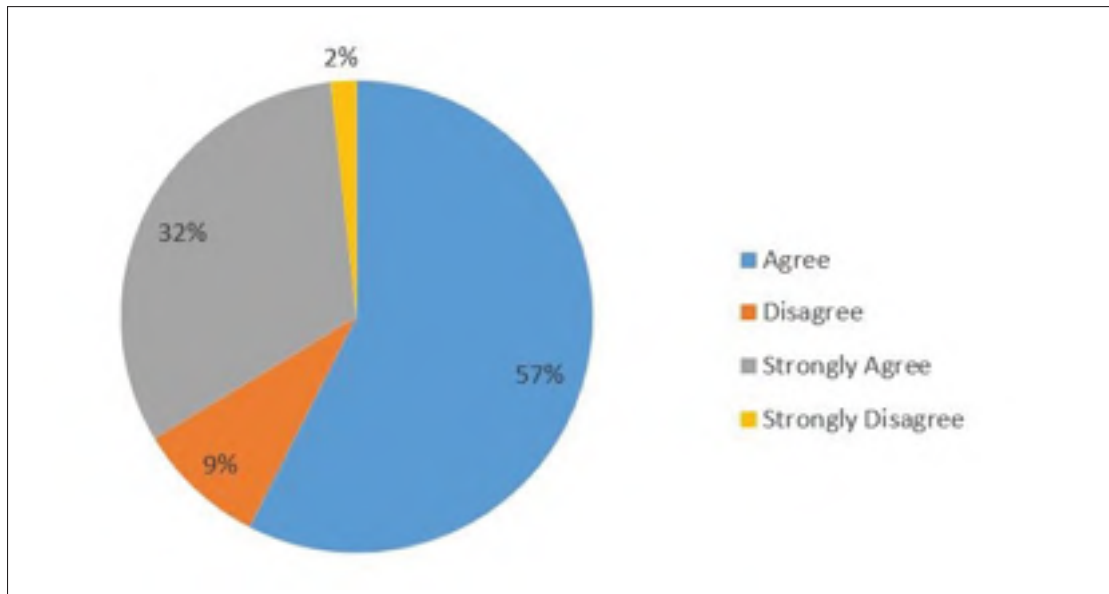

Figure 4.2   Participants answers to the question of "Does this description contain all the information about the class/method?"

4.1.3 Relevance of the produced summaries

To investigate the relevance of the information contained in the produced summaries, we asked the following question "Do the summaries contain only the necessary information?"
The results reported in 4.3 show that 12% of participants believe that summaries did not contain only the necessary information. As a result, there are some information that are not useful in the provided descriptions of methods and classes. We are aware of that and consider to tackle this problem as part of our future work.
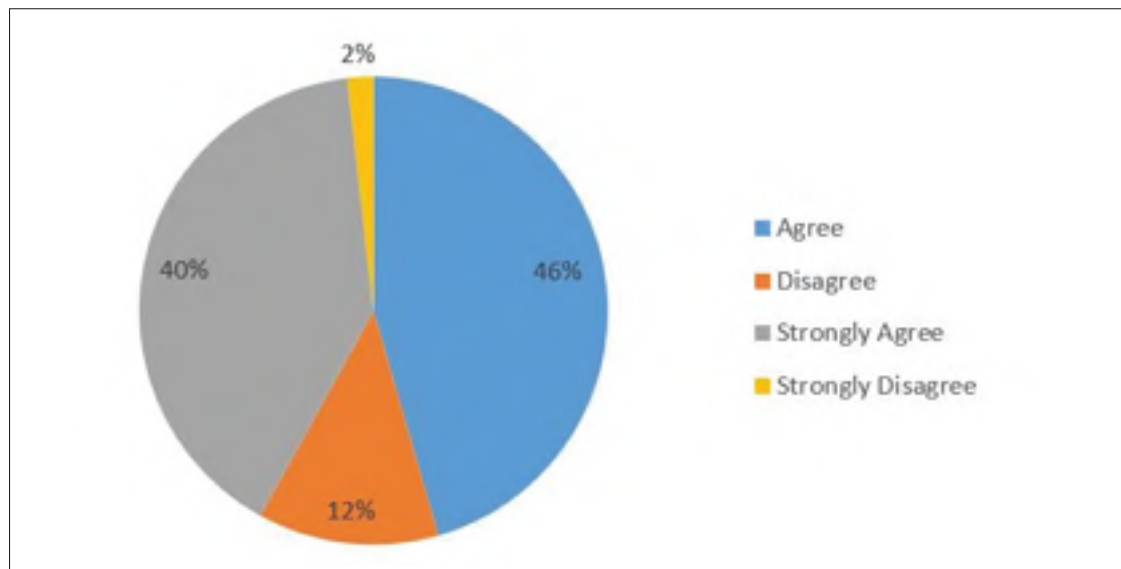

Figure 4.3   Participants answers to the question of "Does this description contain only the necessary information?"

4.1.4 Expressiveness of the produced summaries

Another criteria for the evaluation is the expressiveness of the produced summaries. We asked participants about the existence of information that helps understand how to use classes or methods.
Figure 4.4 shows that 44% of participants strongly agree that the summaries contain information about code entities.
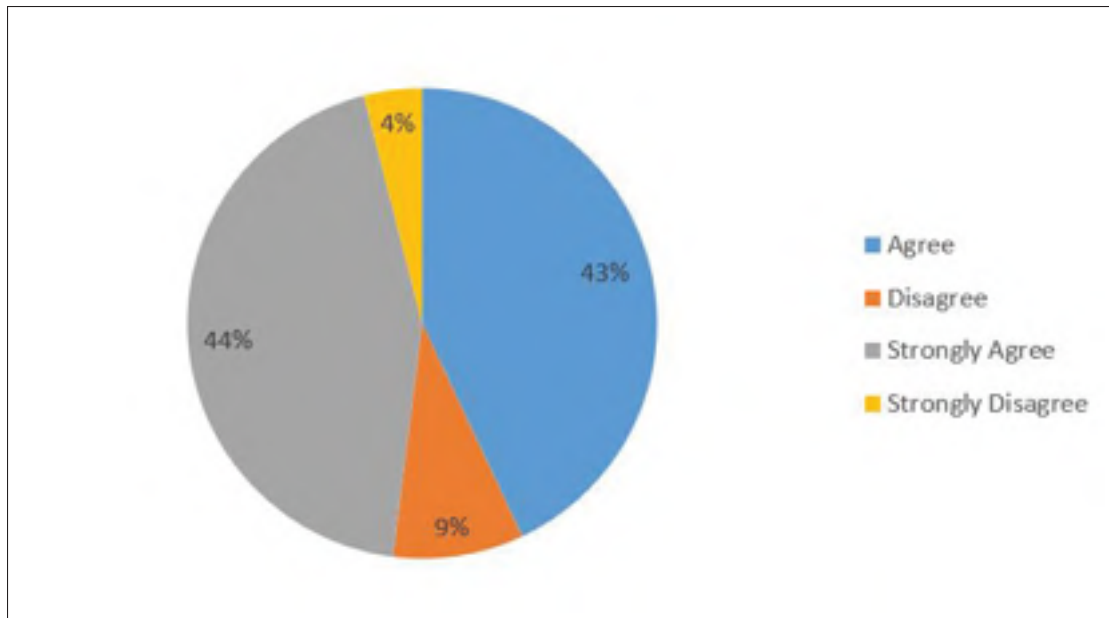
Figure 4.4   Participants answer to the question "Does this description contain
information that helps understand how to use the class/method?"

4.1.5   Adequacy of the produced summaries

For the sake of evaluating the produced summaries based on their adequacy, we asked a question related to the information about the implementation of methods and classes.

Figure 4.5 shows that the majority of participants stated that the summaries include information about the implementation of methods and classes that they were part of their tasks.
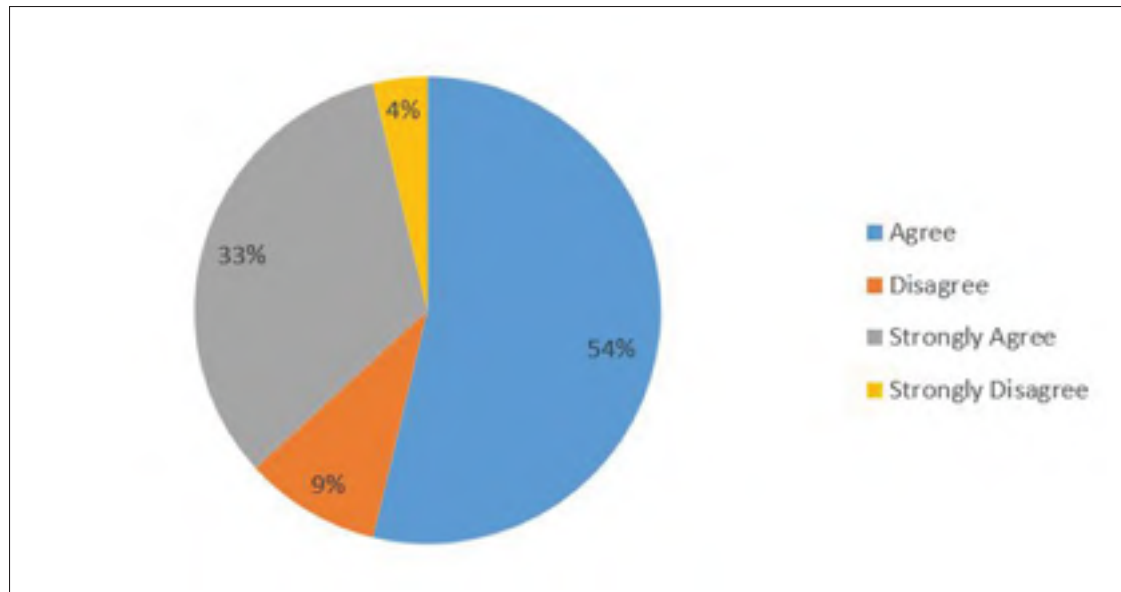
Figure 4.5   Participants answers to the question "Does this description contain information that helps understand the implementation of the class/method?"

4.1.6   Readability of the produced summaries

The last criteria for the assessment of our automatically produced summaries is related to the readability factor. We asked the participants whether summaries are readable.

The feedback of participants in figure 4.6 reveals that the provided descriptions of code entities are, in general, easy to read and understand.

We also analyzed data gained from the post-questionnaire. In the following we present results related to the usefulness of the approach in terms of reducing time and efforts to developers.

4.1.7   Usefulness of the produced summaries

It is important to mention that we cannot rigorously assess the usefulness of our proposed automatic approach unless we perform a controlled experiment where we provide it to an experimental group that do maintenance and–or evolution tasks using our approach. However, in the following, we present insights on this aspect from the feedback we collected from the participants.
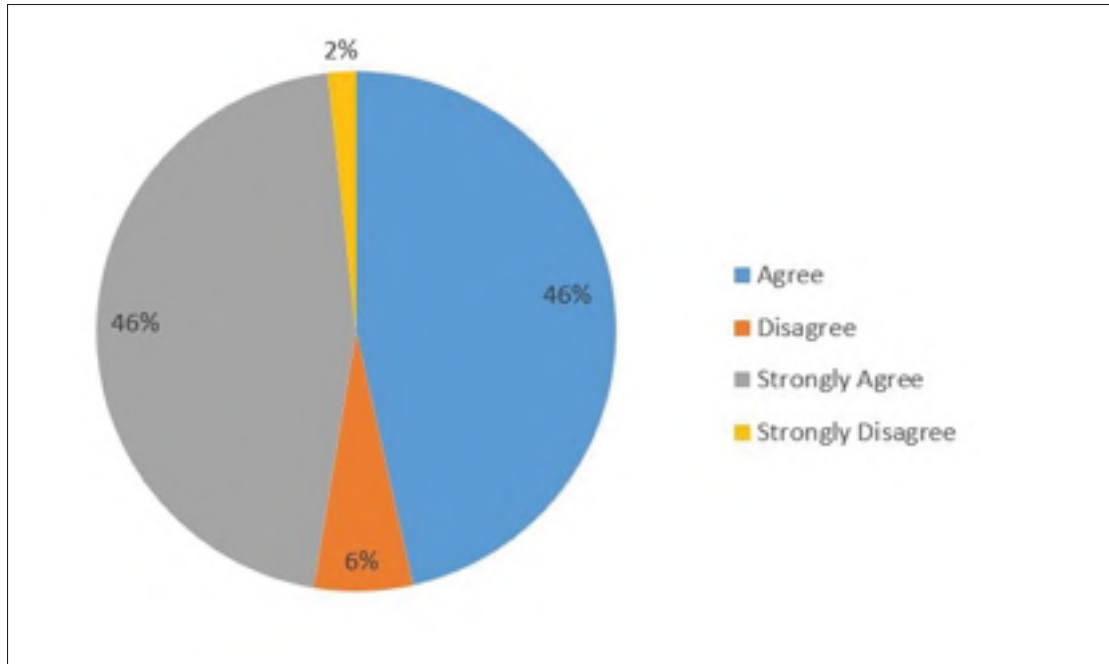
Figure 4.6   Participants answers to the question "Is this description easy to read and understand?"

Figure 4.7 presents the participants' feedback about the reduction of time spent to understand the purpose of code entities. As you can notice, 43.5% of participants found summaries extremely helpful to decrease the understanding time of code entities.
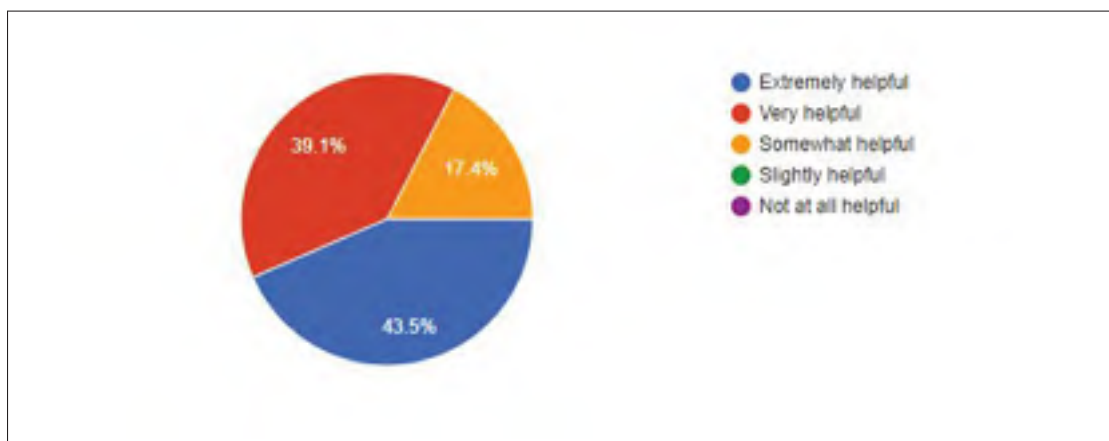

Figure 4.7   Participants feedback about the reducing the time to understand the code entity purpose by produced summaries.

4.1.8 Usefulness of the approach in terms of efforts to understand code entities

Figure 4.8 implies the participants' feedback about the reduction of time spent to understand the purpose of code entities by the suggested approach. As shown in Figure 4.8, 39.1% of participants found that the summaries are extremely helpful to decrease the efforts to understand the code entities.
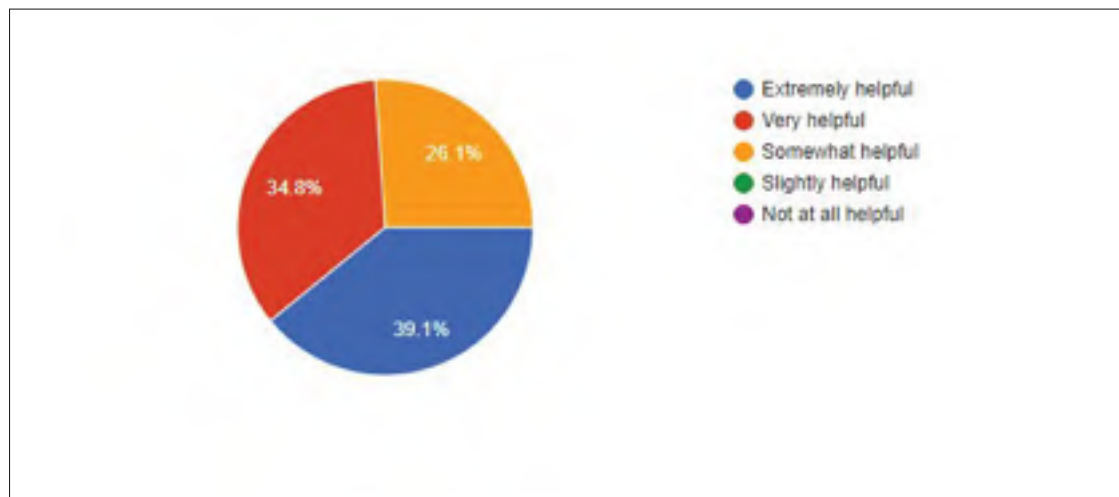

Figure 4.8   Participants feedback about efforts to understand code entities.

By examining bug reports as a type of informal documentation to produce our automatic summaries, we were also interested to find out whether using bug reports is useful to understand the methods and classes or not. The results in the figure 4.9 show that approximately half of the participants (47.8%) rated the bug reports as useful.

In the survey, we also asked participants which parts of bug reports they find the most useful. The findings in Figure 4.10 indicate that the code examples are the best part to find the solution in the bug reports. We believe such an information is important for the research community interested in knowing the type of information to leverage when building summarization tools using bugs as a source of information.
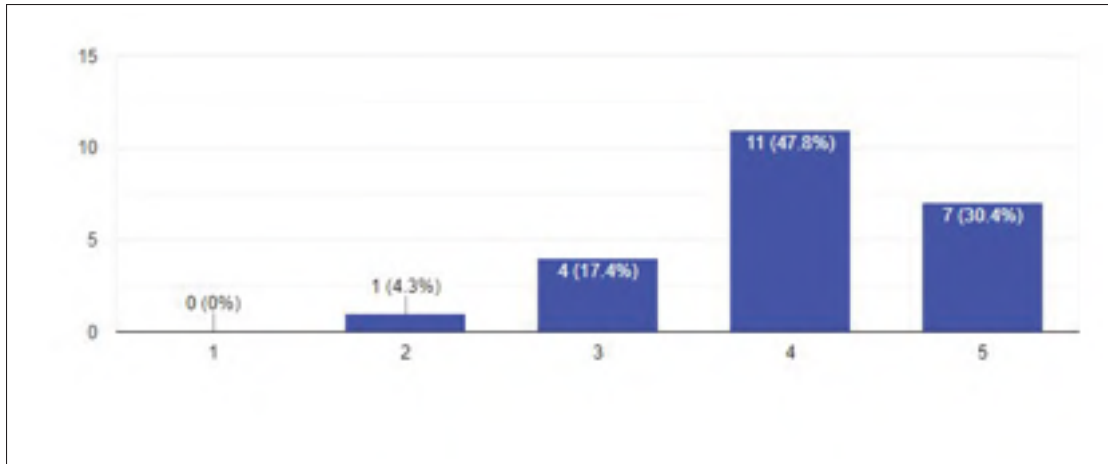
Figure 4.9   Participants answer to the question "Do you find bug reports useful
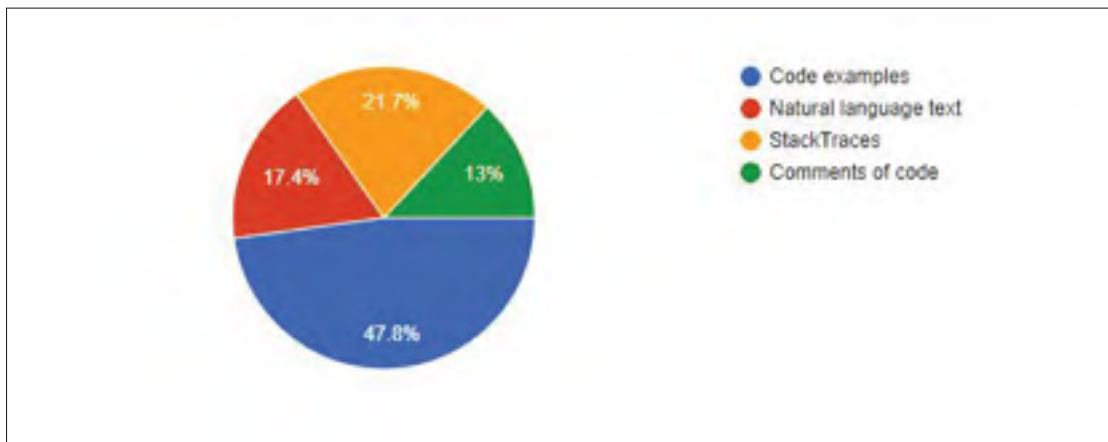to understand classes/methods?"

Figure 4.10   Participants answer to the question of which part(s) of bug reports
they find the most useful.

Overall, even though our study is still preliminary and our proposed approach is still not mature, we believe that the results attained so far are promising and can be a foundation for further research works that aim to build summarizers using informal documentation.

4.2   Threats to the validity

In this section, we present the threats to the validity of our approach. We mainly focus on those that are significant such as the external and internal threats to validity.

- *External validity* : One of the primary threats to external validity is the generalization of our work. To reduce this threat, we have randomly chosen from a large pool of code entities 25 different code entities belonging to various open-source projects: Eclipse, NetBeans, KDE, and Apache. We also examined four different open-source projects from different application domains. We are also aware that the number of participants is important to be able to generalize our results. However, we believe that 23 participants for this preliminary empirical investigation is reasonable.

- *Internal validity* : The first threat to internal validity is about the variation of participants' performance in the experimental tasks, we divided participants into blocks of participants having the same level of experience and–or skills, and then sampled in a stratified manner participants to ensure that their experience is uniformly distributed across the five groups of participants in the experimental design.

  Another threat related to the internal validity is associated with the learning and fatigue effects. We mitigated such a threat by giving a reasonable number of tasks to participants. Specifically, we gave only five code entities and their summaries to participants to evaluate.

4.3 Future Work

In this section, we discuss possible research directions that can lead to the enhancement of the quality of our summaries.

- As part of our future work, we first aim to resolve and overcome the limitations of our approach. As discussed in the thesis, we had several code entities for which we could not generate a summary. This is due to the fact that the code entities in question were most likely discussed in very few bug reports and the machine learning algorithm failed to function properly in this case. Also, when several code entities frequently co-occur together, our approach generate similar summaries for these code entities. Consequently, we need to reflect on how to work on the data improvement part so that our approach can provide summaries and better summaries.

- We also plan to conduct a large-scale study to evaluate our approach by investigating more code entities and more projects. We also plan to examine other artifacts such Stack Overflow, mailing lists and–or the combination of different sources of information.

- Comparing different machine learning algorithms for code entity summarization: While we apply logistic regression as a classifier, we can also examine other algorithms for the classification task. For example, based on recent studies, Bayesian and Naive Bayes algorithms can provide proper results for conversational documentation such as emails, which we can apply for comparison purposes (Trivedi & Dey (2016)). It is important to note that we did not perform any comparisons in this work. However, we plan to create a baseline against which we can compare our summaries. For example, we can generate summaries in a random way by selecting, using a proper algorithm, random words from bug reports to be parts of the summaries and then compare our automatic summaries with this kind of summaries, to see if we at least outperform the random.

4.4 Conclusion

Source code summarization has always been a topic of attention in software engineering. Researchers have greatly contributed to this area with a focus on source code. Unlike past research, we propose to summarize code entities, that are classes and methods, using other types of information. As an example, we examined bug reports.

Extracting useful data from different information sources can be considered as a big challenge in the area of code summarization. The utilization of machine learning for the classification of information is becoming increasingly popular and is quite common especially with the use of large data sets. In this work, we also leverage machine learning to classify relevant words/sentences that should be part of the summaries of code entities.

To answer our research question, we have set up a process for extracting, filtering and collecting data from four well-known open-source projects of Eclipse, NetBeans, KDE, and Apache

that have already been widely-investigated in the state of the art.

The results of our study show that 43% of participants found that summaries are accurate. Regarding the conciseness of our produced summaries, 57% of participants agreed that the summaries contain all information about the examined classes and methods. Additionally, 46% of participants strongly agreed that the automatically produced summaries are readable.

The feedback of participants in the post-questionnaire revealed that 43.5% of participants agreed that the produced summaries can be useful to reduce the time of understanding the purpose of a code entity. Additionally, 39.1% of participants agreed that the generated summaries can reduce the effort of understanding code entities.

The results of this work can be exploited by researchers and practitioners who are interested in building automatic code summarization tools in natural language processing. They can leverage such tools within their Integrated Development Tools to assist developers during their software maintenance and evolution tasks.

APPENDIX I

LINKS TO THE 5 QUESTIONNAIRES FOR 5 GROUPS
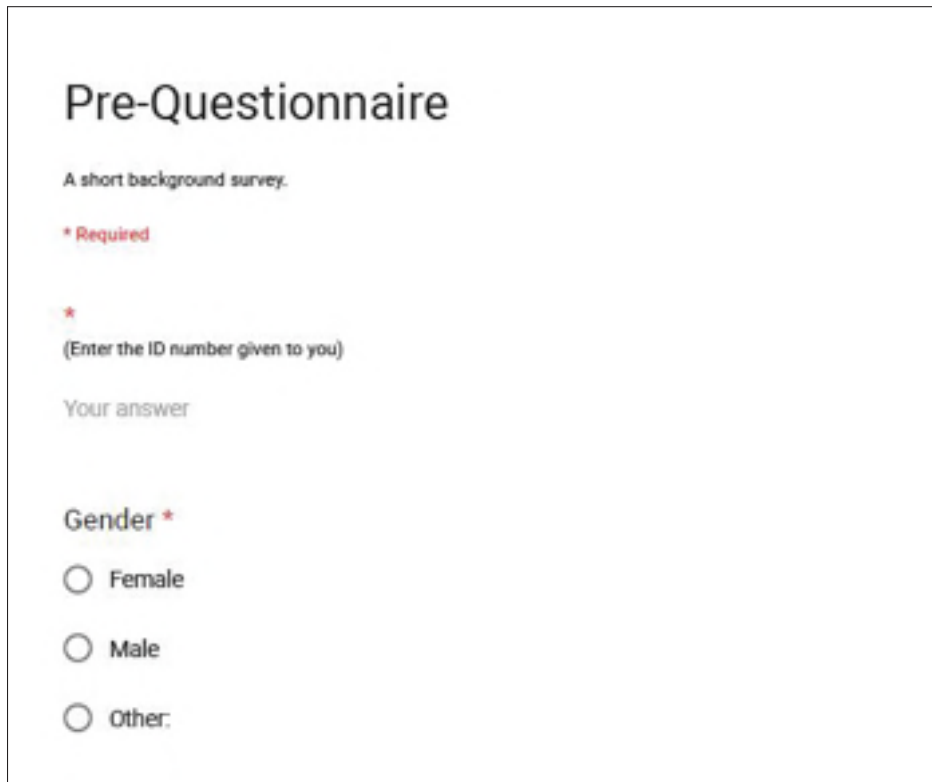
Table I-1 shows the links to each survey.

Table-A I-1   Links to the main questionnaires for each group.

#Group	Links to the main questionnaires
1	https://docs.google.com/forms/d/e/1FAIpQLSfIGC52Fup8iZI55Uut-_TdSQ_PiZnqPueIDwpv9ojjawS2yA/viewform
2	https://docs.google.com/forms/d/e/1FAIpQLSdt25ZRQrfBsTAkYRaszzFJCuiOskdxki09DAeOV-NuwT2TmA/viewform
3	https://docs.google.com/forms/d/e/1FAIpQLSexCpRRqfAyyOTjFwk9ub21YITh3POOe6H4t3KttWa4RsDjRQ/viewform
4	https://docs.google.com/forms/d/e/1FAIpQLSdleTkjfNhJ01VyFJ2_C8IJ9f_hvo_PLcU2zhTQQBelCthyqQ/viewform
5	https://docs.google.com/forms/d/e/1FAIpQLSdoQrMjVx1AjtJQ2DYrW3LuQPGR__5otEbwr6Q_fJrtvm6EcQ/viewform

QUESTIONNAIRES PRESENTED TO PARTICIPANTS

1.   Pre-questionnaire

Figure-A II-1   Sample of pre-questionnaire.

Figure-A II-2   Sample of pre-questionnaire.

What is your level of expertise in the Java programming language? *

○ Poor

○ Basic

○ Good

○ Very Good

○ Excellent

Please select all the degrees you have and are currently enrolled in. *
Check all that apply

☐ Bachelor

☐ Master

☐ Ph.D.

☐ Other:

Current Positions - Select all that apply *
Check all that apply

☐ I currently work in industry

☐ I currently work in academia

☐ I am currently a student

☐ I am currently a faculty member

☐ I am currently a post doc

Figure-A II-3   Sample of pre-questionnaire.

Figure-A II-4   Sample of pre-questionnaire.

2. Questionnaire

Figure-A II-5   Sample of questionnaire.

Does this description contain all the information about the class/method?

○ Strongly Agree

○ Agree

○ Disagree

○ Strongly Disagree

Does this description contain only the necessary information about the class/method?

○ Strongly Agree

○ Agree

○ Disagree

○ Strongly Disagree

Does this description contain information that helps understand how to use the class/method?

○ Strongly Agree

○ Agree

○ Disagree

○ Strongly Disagree

Figure-A II-6   Sample of questionnaire.

Figure-A II-7   Sample of questionnaire.

3. Post-questionnaire

Post Questionnaire

Complete this post- questionnaire after you are done with survey.

* Required

ID number: *
(Enter the ID number given to you)

Your answer

Are you familiar with the following projects? (please check those
that you are familiar with)

☐ Eclipse

☐ KDE

☐ Netbeans

☐ Apache

Figure-A II-8   Sample of post-questionnaire.

Can the given descriptions be used in the context of any software maintenance and evolution tasks to help reduce the time for developers?

○ Extremely helpful

○ Very helpful

○ Somewhat helpful

○ Slightly helpful

○ Not at all helpful

Can the given descriptions be used in the context of any software maintenance and evolution tasks to help reduce the efforts of software developers?

○ Extremely helpful

○ Very helpful

○ Somewhat helpful

○ Slightly helpful

○ Not at all helpful

Do you find bug reports useful to understand classes/methods?
*

	1	2	3	4	5	
Not useful	○	○	○	○	○	Extremely useful

Figure-A II-9   Sample of post-questionnaire.

Figure-A II-10   Sample of post-questionnaire.

BIBLIOGRAPHY

Ahasanuzzaman, M., Asaduzzaman, M., Roy, C. K. & Schneider, K. A. (2018). Classifying stack overflow posts on API issues. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 244–254.

Al-Tahrawi, M. M. (2015). Arabic text categorization using logistic regression. *International Journal of Intelligent Systems and Applications*, 7(6), 71.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F. & Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement?: a text-based approach to classify change requests. *CASCON*, 8, 304–318.

Armaly, A. & McMillan, C. (2016). Pragmatic source code reuse via execution record and replay. *Journal of Software: Evolution and Process*, 28(8), 642–664.

Bacchelli, A., D'Ambros, M. & Lanza, M. (2010a). Extracting source code from e-mails. *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pp. 24–33.

Bacchelli, A., Lanza, M. & Robbes, R. (2010b). Linking e-mails and source code artifacts. *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 375–384.

Badihi, S. & Heydarnoori, A. (2017). CrowdSummarizer: Automated Generation of Code Summaries for Java Programs through Crowdsourcing. *IEEE Software*, 34(2), 71–80.

Basili, V. R., Caldiera, G. & Rombach, H. D. (1994). Experience factory. *Encyclopedia of software engineering*.

Boulis, C. & Ostendorf, M. (2005). Text classification by augmenting the bag-of-words representation with redundancy-compensated bigrams. *Proc. of the International Workshop in Feature Selection in Data Mining*, pp. 9–16.

Chen, J., Huang, H., Tian, S. & Qu, Y. (2009). Feature selection for text classification with Naïve Bayes. *Expert Systems with Applications*, 36(3), 5432–5435.

Dagenais, B. & Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. *Proceedings of the 34th International Conference on Software Engineering*, pp. 47–57.

Eibe, F., Hall, M. & Witten, I. (2016). The WEKA Workbench. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques. *Morgan Kaufmann*.

Guerrouj, L., Bourque, D. & Rigby, P. C. (2015). Leveraging informal documentation to summarize classes and methods in context. *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, 2, 639–642.

Holmes, G., Donkin, A. & Witten, I. H. (1994). Weka: A machine learning workbench.

Jiang, S. & McMillan, C. (2017). Towards automatic generation of short summaries of commits. *Proceedings of the 25th International Conference on Program Comprehension*, pp. 320–323.

Joshi, A., Kale, S., Chandel, S. & Pal, D. (2015). Likert scale: Explored and explained. *British Journal of Applied Science & Technology*, 7(4), 396.

Khatri, C., Singh, G. & Parikh, N. (2018). Abstractive and Extractive Text Summarization using Document Context Vector and Recurrent Neural Networks. *arXiv preprint arXiv:1807.08000*.

le Cessie, S. & van Houwelingen, J. (1992). Ridge Estimators in Logistic Regression. *Applied Statistics*, 41(1), 191-201.

Lewis, D. D. (1998). Naive (Bayes) at forty: The independence assumption in information retrieval. *European conference on machine learning*, pp. 4–15.

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

McBurney, P. W., Liu, C. & McMillan, C. (2016). Automated feature discovery via sentence selection and source code summarization. *Journal of Software: Evolution and Process*, 28(2), 120–145.

Moreno, L. & Marcus, A. (2012). Jstereocode: automatically identifying method and class stereotypes in java code. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 358–361.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L. & Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for java classes. *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*, pp. 23–32.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R., Marcus, A. & Canfora, G. (2014). Automatic generation of release notes. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 484–495.

Moreno, L., Bavota, G., Di Penta, M., Oliveto, R. & Marcus, A. (2015). How can I use this method? *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, 1, 880–890.

Nazar, N., Hu, Y. & Jiang, H. (2016). Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology*, 31(5), 883–909.

Nguyen, M.-T. & Nguyen, M.-L. (2017). Intra-relation or inter-relation?: exploiting social information for web document summarization. *Expert Systems with Applications*, 76, 71–84.

Panichella, S., Panichella, A., Beller, M., Zaidman, A. & Gall, H. C. (2016). The impact of test case summaries on bug fixing performance: An empirical investigation. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 547–558.

Phan, A. V., Chau, P. N., Le Nguyen, M. & Bui, L. T. (2017). Automatically classifying source code using tree-based approaches. *Data & Knowledge Engineering*.

Rastkar, S. (2013). *Summarizing software artifacts*. (Ph.D. thesis, University of British Columbia).

Rastkar, S., Murphy, G. C. & Murray, G. (2010). Summarizing software artifacts: a case study of bug reports. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pp. 505–514.

Rodeghero, P., Liu, C., McBurney, P. W. & McMillan, C. (2015). An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering*, 41(11), 1038–1054.

Rodeghero, P., Jiang, S., Armaly, A. & McMillan, C. (2017). Detecting user story information in developer-client conversations to generate extractive summaries. *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pp. 49–59.

Sawant, A. A. & Bacchelli, A. (2015). A dataset for api usage. *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 506–509.

Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1), 1–47.

Tayal, M. A., Raghuwanshi, M. M. & Malik, L. G. (2017). ATSSC: Development of an approach based on soft computing for text summarization. *Computer Speech & Language*, 41, 214–235.

Treude, C. & Robillard, M. P. (2016). Augmenting api documentation with insights from stack overflow. *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 392–403.

Trivedi, S. K. & Dey, S. (2016). A novel committee selection mechanism for combining classifiers to detect unsolicited emails. *VINE Journal of Information and Knowledge Management Systems*, 46(4), 524–548.

Ying, A. T. & Robillard, M. P. (2014). Selection and presentation practices for code example summarization. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 460–471.