Multi-Document Summarisation from Heterogeneous Software Development Artefacts

MAHFOUTH AHMAD ALGHAMDI

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The University of Adelaide

School of Computer Science

Supervisors: A/Prof. MARKUS WAGNER and Dr. CHRISTOPH TREUDE

February 2, 2022

Contents

List of Figures

List of Tables

Abstract

Software engineers create a vast number of artefacts during project development; activities, consisting of related information exchanged between developers. Sifting a large amount of information available within a project repository can be time-consuming. In this dissertation, we proposed a method for multi-document summarisation from heterogeneous software development artefacts to help software developers by automatically generating summaries to help them target their information needs. To achieve this aim, we first had our gold-standard summaries created; we then characterised them, and used them to identify the main types of software artefacts that describe developers' activities in GitHub project repositories. This initial step was important for the present study, as we had no prior knowledge about the types of artefacts linked to developers' activities that could be used as sources of input for our proposed multi-document summarisation techniques. In addition, we used the gold-standard summaries later to evaluate the quality of our summarisation techniques. We then developed extractive-based multi- document summarisation approaches to automatically summarise software development artefacts within a given time frame by integrating techniques from natural language processing, software repository mining, and data-driven search-based software engineering. The generated summaries were then evaluated in a user study to investigate whether experts considered that the generated summaries mentioned every important project activity that appeared in the gold-standard summaries. The results of the user study showed that generating summaries from different kinds of software artefacts is possible, and the generated summaries are useful in describing a project's development activities over a given time frame. Finally, we investigated the potential of using source code comments for summarisation by assessing the documented information of Java primitive variables in comments against three types of knowledge. Results showed that the source code comments did contain additional information and could be useful for summarisation of developers' development activities.

Declaration of Authorship

Mahfouth Ahmad Alghamdi

September 2021

Acknowledgements

First and foremost, I thank Allah for enabling me to complete my studies and for guiding me through the difficult times of my PhD journey. Praise be to Allah.

I express deep gratitude to my principal supervisor, Associate Professor Markus Wagner, and co-supervisor, Dr Christoph Treude, for making my PhD experience the best it could be. I would not have been able to complete this degree without their technical and moral support. Having their support and guidance throughout my PhD has been an absolute honour. They have always helped me overcome any obstacles and to keep stepping forwards. I am appreciative of everything, including the weekly scientific discussions and the continuous encouragement to challenge myself. They allowed me to make mistakes, develop into the role of researcher, and learn deeply about my research subject.

I am grateful to Professor Takashi Kobayashi and Associate Professor Shinpei Hayashi for hosting me at the Tokyo Institute of Technology in Japan and for supporting me in so many ways. It was a pleasure to meet them, and I will always remember the good times I spent with them.

Heartfelt thanks are also due to my family members, who have all played a unique role in my own life. I especially thank my mother, whose supplications were key to my success. Certainly, I would be unable to achieve academic success without the help of my wife, who constantly encouraged me during my research and provided wonderful support and guidance. I appreciated her help and endless patience with me during this journey.

My children Adeeb and Elias have brought joy and pleasure into my life, and I know these last few years have been difficult for them. I wish them all the best in their future lives.

Finally, I would like to thank the Institute of Public Administration in the Kingdom of Saudi Arabia for funding my studies and my editor Ms Valerie Mobley for copy editing the text of this thesis in accordance with the Australian Standards for Editing Practice.

Chapter 1

Introduction

1.1 Problem Description

The proliferation of openly available software and source code and increased focus on collaborative development are facilitated by code repository services. GitHub is the leading collaborative development platform, with more than 61 million users[1] and more than 235 million repositories[2] hosted, making it the largest source code hosting service in the world. There are multiple reasons for GitHub's success over other collaborative platforms but the main reason is that GitHub offers more than a simple source code hosting service. It also provides developers and researchers with a dynamic and collaborative environment that supports peer review, commenting, and discussion (Dabbish et al., 2012).

In the course of project development, software developers create a large number of artefacts. Managing the huge amount of associated data in a software project becomes more difficult with every new artefact introduced. It is also challenging to find the relevant information exchanged between developers among the vast amount of information available within a single project repository.

The GitHub project repository provides several ways to retrieve desired information about development activities; one obvious way is to use the GitHub search engine. GitHub has introduced a built-in search engine to allow developers to search within a project repository. This can help developers get an overview of their project activity in order to progress their work. However, a search phrase will give results typically containing a large amount of heterogeneous text from many different software artefacts. The developer is still compelled to read through a large bank of returned artefacts to understand

[1]User search, `https://github.com/search?q=type:user&type=Users`, accessed on June 17, 2021.

[2]Github number of repositories, `https://github.com/search`, accessed on June 17, 2021.

it and find the answers they need. This is extremely difficult to do when the developer is working under time constraints.

An alternative is to automatically generate summaries from existing software artefacts. Such summaries can help developers quickly understand the software development activities in any given time frame. In recent years, several methods have been developed for summarising software artefacts, such as bug reports (Rastkar, Murphy, and Murray, 2014), code elements in Stack Overflow (Rigby and Robillard, 2013), code fragments in Eclipse FAQ (Nazar et al., 2016; Ying and Robillard, 2013), classes (Moreno et al., 2013), and methods (Haiduc et al., 2010). All of these techniques have mostly focused on summarising a single type of artefact, and they have not taken into consideration the production of summaries of content in a given time frame. In addition, summaries may need to include different kinds of heterogeneous software artefacts that have been created or updated during the software development process. In a GitHub repository, a developer can create many other types of software artefacts, such as pull requests, commits and issues other than bug reports, source code or code fragments. Also, it is worth noting that these artefacts are rather heterogeneous and that any one artefact can include a mixture of source code, human language, and other types of information.

1.2 Research Goal, Motivation, and Challenges

Our goal is to support developers during the software development life-cycle. In this present study we take several steps towards this goal by seeking to provide developers with multi-document summaries that can be produced from heterogeneous software artefacts and within a given time frame.

The complexity of summarising multi-document software artefacts is due to 1) the sheer variety of artefacts that are continuously created and updated by developers and 2) deciding which information needs to be selected from these artefacts for inclusion in summaries. We centred our main interest on determining which types of artefacts should be used as sources for our summariser, and on selecting information from these artefacts that best describes the developers' activities within a given time frame.

Given the nature of the software development process, which generally requires developers to understand what has been done within a particular period (e.g., a week) to progress their projects, producing such a summary may have to reference different types of heterogeneous software artefacts that describe the developer's development activities. For example, a new developer joins

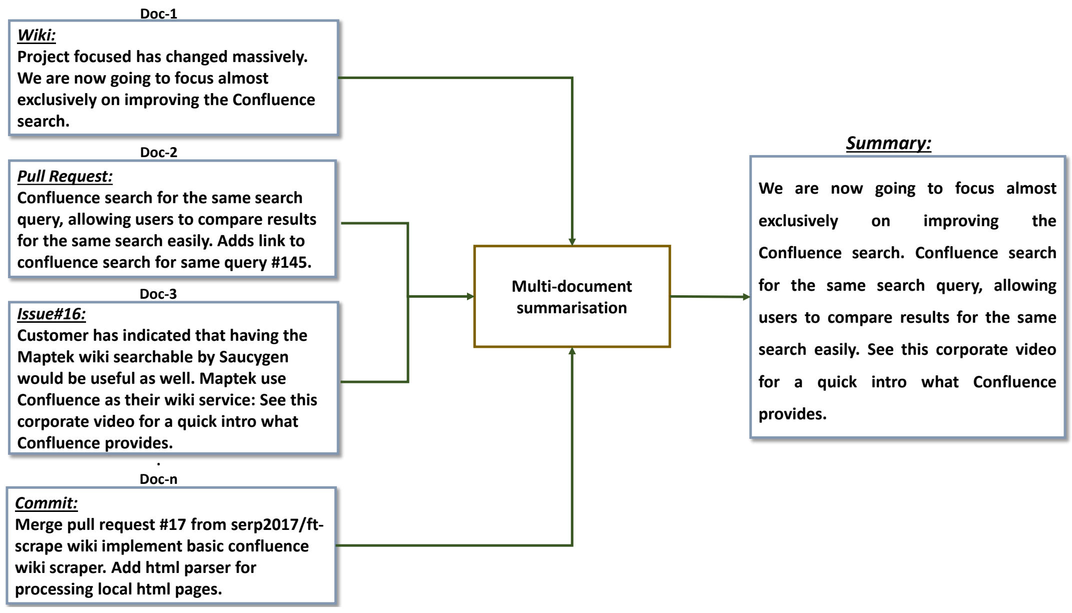

Doc-1
Wiki:
Project focused has changed massively. We are now going to focus almost exclusively on improving the Confluence search.
Doc-2
Pull Request:
Confluence search for the same search query, allowing users to compare results for the same search easily. Adds link to confluence search for same query #145.
Doc-3
Issue#16:
Customer has indicated that having the Maptek wiki searchable by Saucygen would be useful as well. Maptek use Confluence as their wiki service: See this corporate video for a quick intro what Confluence provides.
Doc-n
Commit:
Merge pull request #17 from serp2017/ft-scrape wiki implement basic confluence wiki scraper. Add html parser for processing local html pages.
Multi-document summarisation
Summary:
We are now going to focus almost exclusively on improving the Confluence search. Confluence search for the same search query, allowing users to compare results for the same search easily. See this corporate video for a quick intro what Confluence provides.

FIGURE 1.1. Summarising multi-document software artefacts containing heterogeneous data for a given time frame.

a project team and wants to get an idea about the team's activities over a particular week. The newly joining member need to collect information from different software artefacts to understand the extent of the team's activities in that week. These artefacts can contain information from source code, issues, and other artefacts, all of which can help the new developer seek and extract the desired information.

Similarly, to create an automatic summary from these artefacts, a successful summarisation approach should focus on extracting the most important sentences from a collection of related artefacts that describe the developers' activities. As an example of a summary, Figure 1.1 shows developers' development activities linked to various artefacts (on the left) and the potential automatic produced summary (on the right) that shows the important development activities in that particular week. In the example, team members want to improve the confluence search in their project. For this task, the team members discuss such changes using different artefacts, such as wiki, pull request, issues and commits. An automatic summary would help a team member who is not aware of the changes that happened in the project during that week to get an overview of the essential development activities discussed by the team members in less time. Hence, save the time and the effort to obtain the desired information from different software artefacts.

1.3 Contributions of This Thesis

- *Creation of gold-standard summaries and identify the types of software artefacts.* To better understand what summaries of time windowed software development should look like, we first created our own gold-standard summaries (human-written summaries). We then used these summaries to understand the ideal properties of summaries of software development activities related to heterogeneous software artefacts. Additionally, these summaries provided us with a better understanding of the common types of software artefacts created in the context of developers' activities. The results of this empirical study showed that there were 14 types of software artefacts in GitHub repositories linked to the gold-standard summaries, and this information could then be used as input to create multi-document summaries of developers' development activities. We then investigated whether GitHub's developers actually used the previously identified artefacts (i.e., the 14 types of software artefacts) by conducting a large-scale analysis of 1,038 software-engineered projects. A full explanation of our empirical study for the creation of the gold-standard summaries, identifying the types of software artefacts related to them, and the potential of the use of these artefacts in the GitHub repositories is included here in Chapter 3.

- *Human-Like Summaries from Heterogeneous and Time-Windowed Software Development Artefacts.* We proposed the first framework for summarising multi-document software artefacts containing heterogeneous data within a given time frame. We used the 15 software artefacts as input sources and the gold-standard summaries as target vectors (described in Chapter 3) to identify which sentences from which artefacts should be selected for summarisation approach. We utilised various optimisation heuristics algorithms to extract text from the software artefacts so that the resulting summaries are similar in style to those found in gold-standard summaries and so that they describe the developers' activities. We employed cosine similarity and 26 text-based metrics related to readability metrics, lexical features, and information-theoretic entropy to guide the optimisation approaches to generate multi-document summaries. We then evaluated the generated summaries in a user case study. We describe our approaches and their evaluation in Chapter 4.

- *Characterising the Knowledge about Primitive Variables in Java Code*

Comments. Developers produce various kinds of documentation. One form of this documentation represents source code, such as code comment, while other kinds may refer to external documentation, such as wikis. Studies have shown that source code comments can help improve the readability of source code (Tenny, 1988; Tenny, 1985), while in other studies, researchers expressed the view that code comments are an essential part of software maintenance (Hartzman and Austin, 1993; Jiang and Hassan, 2006). We became interested in assessing the code comments as part of developers' activities because we saw no clear evidence that the gold-standard summaries contained any reference to source code files. This meant that source code artefacts were not included in the artefacts we used as input sources for our summarisation approaches. Source code comments are considered to be an essential source of information about a project, as they can be used to describe the developers' development activities. We therefore wanted to assess the potential of source code comments to be considered as an input source for our multi-document summarisation approach. To achieve this aim, we proposed the first study to investigate the role of primitive variable identifiers in comments, especially how commonly these identifiers are documented in accompanying comments and what type of additional information the comments supply about these variables. We developed lexical and advanced matching techniques to capture the identifiers of primitive variables in Java source code comments, and then evaluated these approaches using a manually curated benchmark of six well-commented project repositories hosted on GitHub. We manually classified the documented information used to describe the variable identifiers in the comments into three types of knowledge: purpose, concept and directives. Finally, a large-scale analysis of 2,491 engineered Java software repositories hosted on GitHub was carried out to provide an insight into how developers document these variables in the form of source code comments. We describe our approaches and their evaluation in Chapter 5.

1.4 Thesis Outline

The rest of this thesis is organised as follows. In Chapter 2, we give an overview of natural language summarisation techniques and their applications in software engineering.

The core of this thesis starts in Chapter 3, where we create our own gold-standard summaries, characterise them and identify the types of software artefacts in GitHub projects' repositories linked to the developers development activities, and investigate how commonly developers use them in GitHub project repositories.

In Chapter 4 we introduce our first framework for summarising multi-document software artefacts containing heterogeneous data within a given time frame. This approach integrates techniques from natural language processing, software repository mining and data-driven search-based software engineering to automatically generate summaries in extractive setting from 15 types of software artefacts within a given time frame.

In Chapter 5 we present the first study that analyses a bank of documented information for primitive variables in source code comments and then characterise these findings into different types of knowledge.

Finally, in Chapter 6, we conclude the thesis by summarising its main contributions and by outlining the potential for future work in this research area.

Chapter 2

Background

2.1 Introduction

The exponential growth of the internet has given rise to information overload, a problem with which many researchers and users still struggle. Sources of information such as newspapers, opinion articles, emails, and microblogs increasingly overwhelm users who seek to use them in their personal or professional lives. Similarly, the growth of big data from bulk sources, such as scientific databases, compounds the problem of information overload. These trends have triggered the desire for a summarisation system capable of alleviating this problem. Researchers have spent considerable time and resources in developing summarisation systems that condense information from one or more documents, using natural languages.

The software engineering community has taken an huge interest in summarisation systems in recent years as software developers have created voluminous information related to system development. The variety of software artefacts that these developers create daily includes documentation, bug reports, and source codes. They also generate pull requests and make wiki entries to describe processes of system development. The automatic summarising of these software artefacts has recently become a dominant concept in software engineering research, with many studying how they can best minimise information overload in this field.

In this chapter we present a general overview of summarisation techniques in Section 2.2. We then discuss automatic summarisation in software engineering research in Section 2.3.

2.2 Automatic Text Summarisation

The origins of automatic text summarisation as a research discipline date back to the seminal work of Hans Peter Luhn (Luhn, 1958). Since then, researchers have developed numerous techniques for extracting key information from a collection of source documents to create automatic summaries (Gupta and Lehal, 2010; Nenkova, McKeown, et al., 2011; Saggion and Poibeau, 2012). The aim of automatic text summarisation is to produce a short representation of original text while preserving the overall meaning, as well as other essential information. Over the years, researchers have produced different definitions of a summary based on individual perspectives, as highlighted in the three examples below.

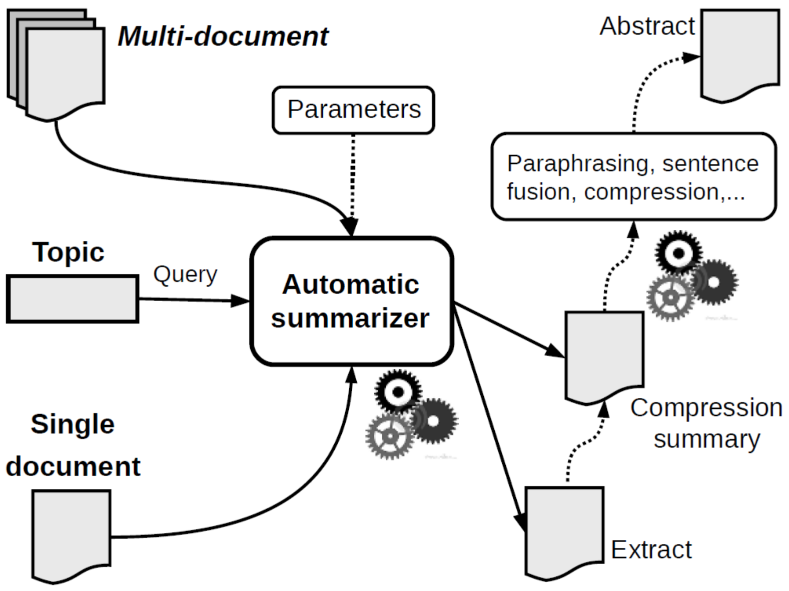


FIGURE 2.1. Typical architecture of a summarisation system (Torres-Moreno, 2014)

.

1. According to Jones (Jones et al., 1999), a summary is a "reductive transformation of source text to summary text through content reduction by selection and generalisation on what is important in the source."

2. Saggion and Lapalme described a summary as "a condensed version of a source document having a recognisable genre and a very specific purpose: to give the reader an exact and concise idea of the contents of the source." (Saggion and Lapalme, 2002)

3. Hovy and Marcu (Hovy and Marcu, 2005) defined a summary as "a text that is produced from one or more texts, that convey important information in the original text(s) and that is no longer than half of the original text (s) and usually significantly less than that."

A careful review of the definitions above reveals that they have three crucial aspects in common:

- The summaries can be generated from one or more documents.
- The summaries should preserve crucial information such as key information content and overall meaning of the original texts.
- The summaries should be concise.

Automatic text summarisation systems can be categorised into several different types (Nenkova and McKeown, 2012; Saggion and Poibeau, 2013). The dimensions of text summarisation can be generally categorised based on input type (single or multi-document), purpose (generic or domain specific) and output type (extractive or abstractive). A simplified abstracting process for a summarisation system is depicted in Figure 2.1.

Text summarisation based on input, is either single or multi-document. Text summaries based on purpose are either domain-specific or generic. Lastly, text summarisation based on form of outcome is either abstractive or extractive.

In the next subsection we give an overview of each of these categories and their overall relation to the present study.

2.2.1 Extractive summaries vs. abstractive summaries

The summarisation of tasks based on outputs can be either abstractive or extractive. Extractive summarisation refers to the selection of essential sentences or paragraphs from source documents to form a summary. The linguistic and statistical features of the sentences can determine the importance for inclusion (Gupta and Lehal, 2010). Likewise, extraction based on statistical features of sentences focuses on shallow characteristics of text such as their position within the source document (Ouyang et al., 2010), sentence length (Kupiec, Pedersen, and Chen, 1995), sentence centrality (Erkan and Radev, 2004), cue phrases (Edmundson, 1969) and word frequency (Luhn, 1958). Extractive summarisation techniques calculate the significance of sentences based on these features to generate summaries.

Unlike statistics-based extraction, summaries based on linguistic properties focus on the semantics of terms and their relationships to each other. The linguistic approach evaluates the relationships between terms based on analysis of grammar, the usage of thesaurus, and Part of Speech (POS) tagging. Several researchers, including (Tayal, Raghuwanshi, and Malik, 2017; Ferreira et al., 2014) analysed the impact of combining various shallow text features on the quality of generated summaries. These researchers tested the hypothesis that the combination of these features, based on context, would provide a rich source for summaries compared to purely linguistics-based summarisation approaches. Their findings suggest that statistics-based summaries allow more efficient computation, while linguistics-based methods can generate better summaries.

In contrast to the extractive approach, abstractive summarisation focuses on the comprehension of the main concepts within source documents before summarising them in understandable natural language (Erkan and Radev, 2004; Hahn and Romacker, 2001). The abstractive summarisation technique is either structure-based or semantic-dependent. According to some scholars, structured-based methods break down and assess texts to find new concepts that they may use to generate summaries from source documents (Kikuchi et al., 2014; Hirao et al., 2015). Others have used semantic-dependent techniques for examining and interpreting texts to find new expressions that they may use to convey key information content from original documents (Sarda and Kulkarni, 2015; Reeve Lawrence et al., 2006).

Generating abstractive summaries is a more challenging task because it necessitates the semantic representation of texts. In addition, it requires extractors to generate natural languages while keeping inference rules (Balaji, Geetha, and Parthasarathi, 2016). Abstractive summaries also face the challenge of accurately "understanding" natural language, as natural language generation techniques are still an emerging field. In fact, extractive techniques have been thought to produce better summaries than abstractive techniques (Allahyari et al., 2017; Mishra and Gayen, 2018). Although extractive summaries are relatively easier to create than sophisticated abstractive summaries, extracting salient information from the original text to form summaries still poses numerous challenges to researchers. For instance, researchers need to identify preprocessing steps for matching collected datasets and selecting appropriate features to improve summarising performance. Similarly, researchers need to apply feature engineering to learn how to exploit existing features for better summaries. Lastly, they need to find an appropriate technique for linking one approach to another to enhance the quality of the produced summaries.

In this thesis we used an extractive technique to summarise development activities from a collection of software artefacts (Chapter 4). In addition, unlike previous work, which used statistics-based shallow features to determine the most salient sentence in a text, we defined our own features to extract the salient sentence.

2.2.2 Single document vs. multi-document summarisation

Input documents to a summarisation system is classified as either single or multiple-document. A single document generates a summary from one source that includes content about a similar topic (Radev, Blair-Goldensohn, and Zhang, 2001). In contrast, a multi-document summarisation produces summaries from numerous sources or documents but still about a similar topic (Qiang et al., 2016; Widjanarko, Kusumaningrum, and Surarso, 2018). Single-document summarisation has been the subject of extensive research, particularly around sentence extraction methods. There has been less work done on multi-document summarisation, but it has recently gained more attention from researchers because of its advantages over single-document summaries in the following respects:

- It summarises a topic by offering a domain overview, which highlights information that is often mentioned across various documents.
- It identifies unique content within each document.
- It shows relationships between information contained in separate documents (Ou, Khoo, and Goh, 2006).

Multi-document summarisation presents numerous challenges, as opposed to single-document summarisation (Goldstein et al., 2000). The problems are attributed to the varied and often inconsistent data contained in multiple documents. In addition, the number of documents is typically large, and the link between documents can be quite sophisticated. Due to the large number of documents that may also be lengthy, they can overlap, conflict with, or complement each other, making it challenge for systems to extract the key content from input texts to generate non-redundant, coherent, and readable summaries (Tas and Kiyani, 2007; Peyrard, 2019). Therefore, the complexity of multi-document summarisation requires sophisticated models to assess, identify and merge dependable data. Moreover, the computation power required for multi-document summarisation is extensive because of the increasing number of parameters used in language models and the large volume of related datasets. The use of powerful statistical and optimisation techniques, however, can resolve some of these issues (Rautray and Balabantaray, 2017).

In this thesis we define our problem for summarising software development artefacts as multi-document summarisation. In Chapter 3 we identify the types of software artefacts as input documents and we then in Chapter 4 explain how we generated summaries of their content. It is important to note that the artefacts in the input collection are heterogeneous in nature, that is, they do not necessarily contain related information.

2.2.3 Generic vs. domain-specific summarisation

The majority of abstractive and extractive summarisation approaches tackle the challenge from a generic vantage point. They attempt to create a system that does not take any assumption about the characteristics of input documents such as their content or structure. Hence, the outcome is a summary that works seamlessly with any new document. Contrarily, domain-specific summarisation focus on specific characteristics that they use to identify the important content precisely. For example, the specific format or characteristics of texts are attributes of domain-specific summaries such as medical document (Afantenos, Karkaletsis, and Stamatopoulos, 2005), legal document (Grover, Hachey, and Korycinski, 2003), and scientific document (Qazvinian and Radev, 2008).

The approach proposed in this thesis (see Chapter 4) to summarise software artefacts falls into the category of domain-specific summarisation, as it makes use of the particular content of input data (i.e., the textual artefacts).

2.2.4 Evaluation methods

Evaluating the quality of summaries produced by automatic systems is necessarily subjective and is an extremely difficult task to implement. It is subjective because the perfect summary does not exist. Indeed, evaluating summaries is an open problem to which the scientific community has responded with various partial solutions (Torres-Moreno, 2014).

Despite the challenges of the task, several strategies have been proposed to evaluate and compare different summarisation techniques. The quality of produced summaries can be assessed through extrinsic or intrinsic evaluation (Steinberger and Jezek, 2009). Intrinsic evaluation encompasses internal attributes, such as the content and quality of the text, which is difficult to automate, since human interactions are involved. Human evaluators typically assess sophisticated characteristics such as cohesion, grammar, readability, or coherence, which may offer evidence of text quality. However, the automatic assessment of content summaries using gold-standard approaches can suit comparative methods of assessment. For example, the content of automatically generated summaries can be evaluated using precision, recall to the ideal gold-standard summaries.

On the other hand, extrinsic techniques assess the influence of the summaries related to a particular activity. Typically, this assessment comprises the human evaluation of tasks to determine summarisation systems that support real-life tasks significantly. According to (Mani et al., 1999), the TIPSTER (text summarisation evaluation) method was an early attempt at evaluating summaries to identify their relevance to tasks of particular interest. Another study investigated the impact of providing a group of participants with summaries and asking them to write reports on specific topic (McKeown et al., 2005). The researchers noted that the participants wrote quality reports and were greatly satisfied with their efforts.

In this thesis we used extrinsic evaluation techniques to assess the summaries automatically generated from software artefacts (Chapter 4). We conducted a user study to extrinsically evaluate the summaries by asking expert developers whether each summary mentioned all important project activities in the gold-standard (human-created) summaries.

2.3 Automatic Summarisation in Software Engineering

In the course of a software life-cycle, software developers create multiple artefacts. (Souza, Anquetil, and Oliveira, 2005) reported on 34 artefacts that are used to determine the requirements of software systems. Such artefacts can help assess, design, code, as well as test the systems. Other artefacts created by developers are also used to assist in evolving, maintaining, and understanding software tasks, including communication logs and defect reports, commits, and pull requests. Researchers have proposed automated summarisation systems to aid developers in searching for specific information from the artefacts. These summaries are beneficial at the phase of software development because retrieving necessary information from collection of artefacts is tedious and time-consuming.

Researchers have paid attention, in recent years, to summarising various types of software artefacts, including source codes, bug reports, commits and pull requests. Concerning the source codes, (Moreno et al., 2013) proposed the use of JSummarizer, a technique that identifies stereotyped classes by adapting the regulations of method distribution to the class stereotypes. Distinguishing between templates of various classes and methods helps create a range of templates for generating summaries. In another effort, (Moreno et al., 2014) attempted to generate release notes automatically by reutilising the JSummarizer to produce a description of newly created classes. Similarly, (Hu et al., 2018) proposed a tool, DeepCom, to generate comments from learned features of a massive code corpus using the natural language processing (NLP) technique. Afterwards, they utilised recurrent neural networks and long short-term memory (LSTM) neural networks to evaluate the suitability of the Java framework for methods to generate better comments.

In similar effort, Huang et al. (Huang et al., 2020) proposed a method, RL-BlockCom, to generate a block comment for code snippets. They first uses heuristic rules and machine learning algorithms to identify the scope of the block comment and then they apply the RL-BlockCom method that automatically generating block comment. Recently, Wang et al. (Wang et al., 2021) proposed a tool, CRASOLVER, that could generate summaries from given solutions of the crash traces discussed by developers on the Q&A website, such as Stack Overflow. Ruyun et al. (Wang et al., 2020), proposed a model called Fret, which used the functional reinforcer and Bert embedding to generate code comments. Adding the learning code functionalities to the model, they generated summaries for code comments that could describe the code functionalities.

In bug reports, (Jiang et al., 2017) utilised byte-level n-grams in replicating the authorship attribute properties of open-source projects. The authors sought to understand similarities in the interaction between normalised simplified profiles by generating short summaries from descriptions and comments within specific bug reports. In similar attempts (Mani et al., 2012; Lotufo, Malik, and Czarnecki, 2015) proposed unsupervised approaches based on noise reducer and heuristic rules to summarise bug reports.

As for summarising commits, (Buse and Weimer, 2010) proposed DELTADOC, a technique for summarising a commit. They first used symbolic execution and path predicate analysis to produce the behavioural difference and then applied heuristic transformations in generating descriptions of natural language. Likewise, (Cortés-Coy et al., 2014) created the ChangeScribe tool to identify stereotypes from commits in abstract syntax trees. Then, they used predefined patterns and filters in generating descriptive commit messages. In other work, (Liu et al., 2019) proposed an approach to generate pull request summaries from commit messages and added code comments in the pull requests. They used attentional encoder-decoder models with pointer generators to generate these summaries.

Most of these approaches considered the production of summaries from a single software artefact, such as only from bug reports, only from classes, and so on. In contrast to these, our approach generates summaries for a given time frame from multi-document software artefacts that contain heterogeneous data. Our focus was on summarisation approaches that consider the natural language content of software artefacts to generate text-based summaries (see Chapter 4). We then studied the potential of using code comment artefact for summarisation by assessing the domain knowledge they contain (see Chapter 5).

A large portion of Section 3.2 in the next chapter has been previously published (Alghamdi, Treude, and Wagner, 2019) and presented in the Genetic and Evolutionary Computation Conference Companion (GECCO'19). I contributed to the design, implementation, and evaluation of the proposed approach.

Chapter 3

Creating Resources for Summarising Multi-document Software Artefacts

3.1 Introduction

Researchers working on automatic summarisation require resources to compare their work to other researchers when evaluating the quality and performance of their summarising approaches. These resources consist of 1) a collection of documents used as an input to the summariser tool and 2) a collection of human written (gold-standard) summaries. The gold-standard summaries are those created by human summarisers to compare against the automatically generated summaries. In the context of this thesis, the gold-standard summaries are those summaries created by students developers on a weekly basis to describe their development activities in their projects.

When our work in this thesis started, there were no gold-standard summaries or publicly available summaries of developmental activities that occurred within a given time frame. Therefore, it was necessary to create our gold-standard summaries for three reasons:

1. To identify the types of software artefacts for the generation of multi-document summaries that describe developers' development activities over a given time frame.

2. To better understand the general properties of human-written summaries of developers' development activities in order to produce automatic summaries close in style to human summaries.

3. To finally quality-compare our multi-document generated summaries against the gold-standard summaries (human-written summaries).

This chapter describes the creation of resources for the multi-document summarisation. In Section 3.2 we describe the process of creating the gold- standard summaries and understanding the characteristics of the texts. In Section 3.3 we describe the process of identifying the collection of artefacts for our multi-document summarisation. Section 3.4 describes how we conducted a large analysis to determine whether GitHub's developers utilised these artefacts during their software developments. Section 3.5 describes our investigation of other types of artefacts that could be used for summarisation purposes. Finally, we conclude the chapter in Section 3.8.

3.2 Creating and Characterising Gold-standard Summaries

3.2.1 Gold-standard summaries

To the best of our knowledge, there is no existing approach, in the context of software engineering, to create multi-document summaries from heterogeneous software artefacts within a given time frame. However, past work has shown that developers desire such an approach (Treude, Figueira Filho, and Kulesza, 2015). Therefore, as our first step toward the goal of generating human-like summaries from heterogeneous software artefacts in a given time frame, we created our gold-standard summaries. The summaries were generated by 53 student developers over 14 weeks and related to 15 (university-internal) GitHub capstone projects. These projects were set during three courses taught to the students in 2017 and 2018. The students worked in teams of three or four on their undergraduate (Bachelor) projects with clients from local industry (43 students) or with clients from academia on projects toward their Master's degrees (10 students). While the students were working on their projects, they were asked to write summaries that described their development activities. The students wrote their summaries in response to the question: *If a team member had been away, what would they need to know about what happened this week in your project?*. We used a Slack bot to automatically ask this question on a weekly basis and to record the responses. We collected a total of 545 responses where each response constituted a summary. The median number of the collected summaries over the 14 weeks and from all the students' projects was 11 summaries, indicating that each student wrote almost one summary per week.

3.2.2 Characteristics of gold-standard summaries

Using quantitative indicators of the text complexity features, such as text length, the number of sentences, text readability and the amount of information in these summaries can help us to understand the general properties of summaries containing software development activities. In addition, knowing such characteristics of the student summaries can guide us to automatically generate summaries with text features close in style to the text features of the gold-standard summaries. Toward this aim, we identified 27 text-based features to capture different aspects of the summaries based on lexical features, readability metrics, and information-theoretic entropy. The 27 features are listed in Table 3.1.

Each of the gold-standard summaries first underwent text cleansing before we calculated its features. We split each summary into paragraph. We then removed all non-ASCII characters, special characters (e.g. exclamation marks, brackets, single/double quotes), URLs, file paths and file extensions and replaced these with single white spaces. We then split each paragraph into sentences and then tokens using the Apache OpenNLP toolkit.

Figure 3.1 illustrates the text characteristics of our gold-standard summaries. The distribution of the summaries based on word count shows that most of the summaries contained between 30 and 90 words. In addition, calculating the unique words used (i.e., words appearing only once in each summary) shows that most of the students used around 30 to 60 unique words in their summaries. We also calculated the incidence of difficult words in the summaries. Difficult words are those words that do not belong to the Dale list of 3,000 familiar words (Dale and Chall, 1948). The median result of difficult words was 19 words per summary. Calculating the unique and difficult words reflected that the students performed a wide range of development activities, and thus, they needed to use specific words to describe their activities. In addition, the median of sentences for the summaries shows that most of the students tended to use 3 sentences per summary.

Next, we calculated the Flesch reading ease score (Flesch, 1948) for each summary to understand how readable these summaries were. The Flesch reading ease test is widely used. For example, the US military uses Flesch to check the readability of technical documents. The basic concept behind Flesch is to measure two distinct ratio elements: 1) the word to sentence ratio and 2) the syllable to word ratio. It then converts the ratios into a score that can be used to describe a text's readability. The resulting readability score can give negative

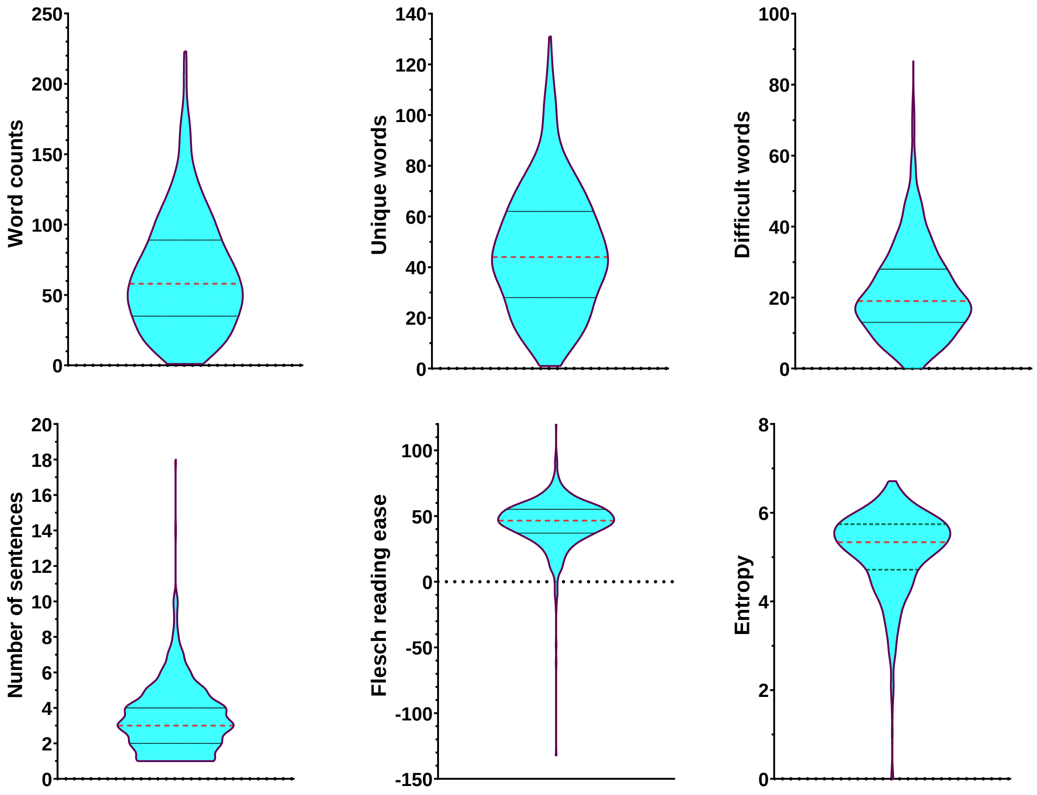


FIGURE 3.1. Characteristics of the student summaries using statistical analysis of text features.

Table 3.1. Features used to analyse the text properties.

No.	Feature
F1.	Word count
F2.	Chars count including spaces
F3.	Chars without spaces
F4.	No. of syllables in a word
F5.	Sentence length
F6.	Paragraph length
F7.	Unique words
F8.	Avg. word length (chars)
F9.	Avg. sentence Length (words)
F10.	No. of monosyllabic words
F11.	No. of polysyllabic words
F12.	Syllables per word
F13.	Difficult words
F14.	No. of short words ($\leq$ 3 chars)
F15.	No. of long words (>= 7 chars)
F16.	Longest sentence (chars)
F17.	Longest words (chars)
F18.	Longest words by number of syllables
F19.	Estimated reading time
F20.	Estimated speaking time
F21.	Dale-Chall readability index
F22.	Automated readability index
F23.	Coleman-Liau index
F24.	Flesch reading ease score
F25.	Flesch-Kincaid grade level
F26.	Gunning fog index
F27.	Shannon entropy

or positive values. Lower values indicate that the text is very difficult to read, and the higher values indicate the text is easy to read. The formula to obtain the readability score is illustrated in Equation 3.1. Figure 3.1 shows the distribution of the summaries based on the Flesch readability score. The distribution shows that the student summaries scored 46.5 based on median value, which indicates that the summaries are *difficult to read* and can only be understood by college students or above[1]. This result is not unexpected as these summaries were created by undergraduate and graduate students.

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right) \tag{3.1}$$

Lastly, we measured the amount of information in each summary using Shannon's entropy (Shannon, 1948). The Shannon entropy was calculated for each summary using Equation 3.2, where p is the probability of word x_i appeared in the summary. The distribution of the summaries based on the entropy values shows that most of the summaries included a large amount of information (median value = 5.33).

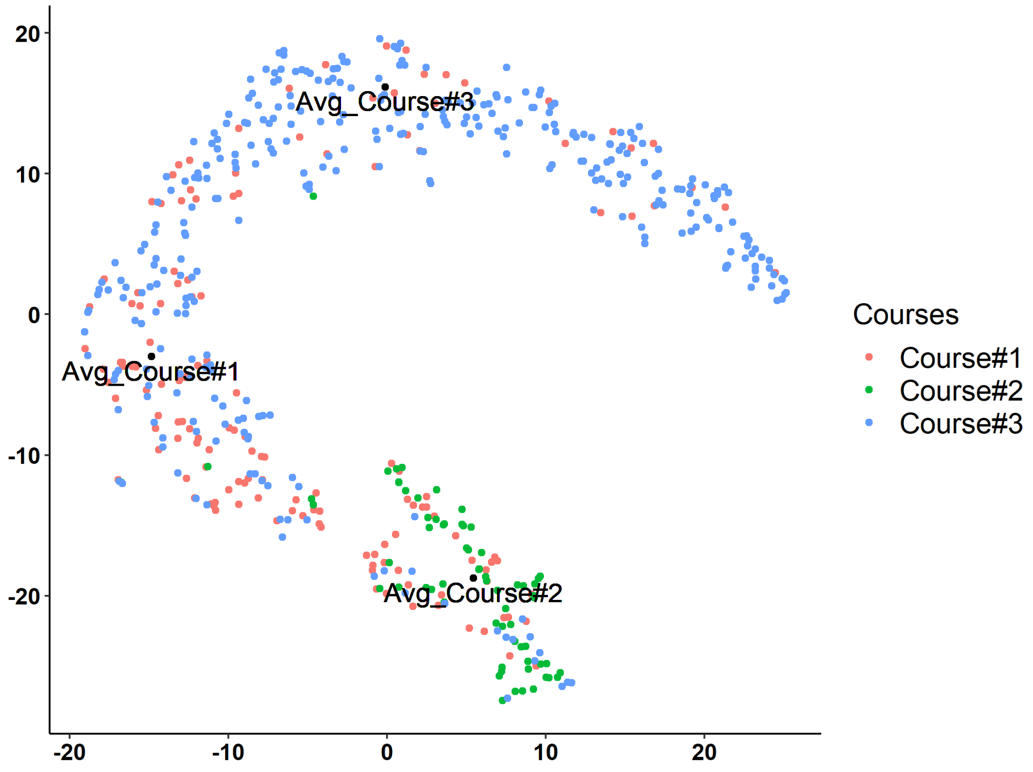
$$E = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{3.2}$$

A brief summary: We created our gold-standards summaries to help us to identify software artefacts that describe developers' development activities as cited in the summaries. We analysed these summaries using quantitative indicators of the text complexity features to reveal the general properties of human-written summaries, which can guide us to then generate summaries with text features close in style to the textual features of the gold-standard summaries. For example, we found that students tended to use three sentences per summary. Based on the median value of readability, the summaries were considered to be *difficult to read*, and most summaries included a lot of information (Entropy's median value is 5.33).

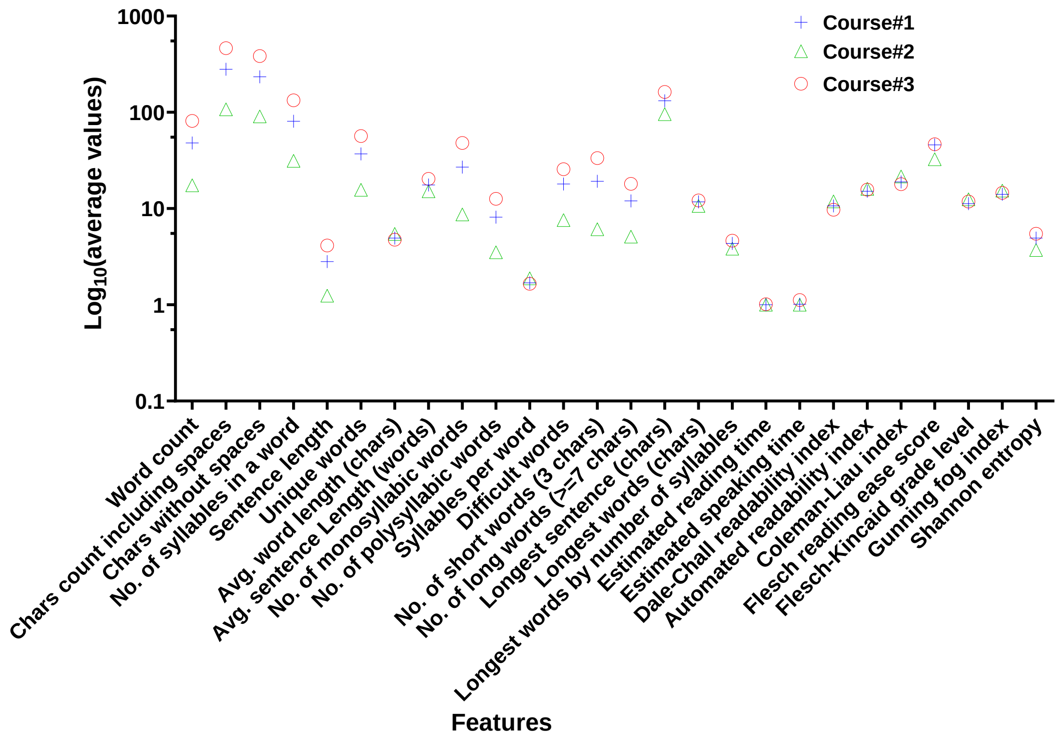
Insights per grouping:

As the student summaries are intended to guide us in our creation of automated human-like summaries (Chapter 4), we investigated our dataset for possible hidden biases and also for changes over time; note that both are purely observational. First, we calculated for each of the 545 summaries generated

[1]Flesch-Kincaid readability tests: `https://en.wikipedia.org/wiki/Flesch%E2%80%93Kincaid_readability_tests`, accessed on July 25, 2021.



(A)



(B)

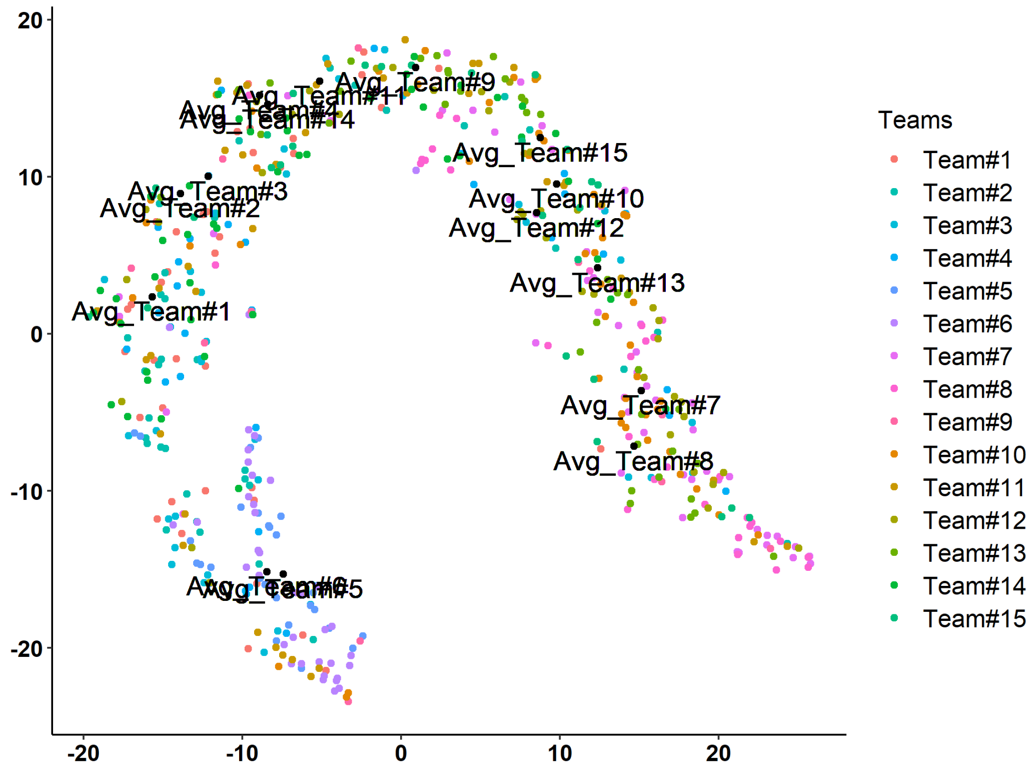
FIGURE 3.2. (a): 27 text-based features of the students summaries grouped by courses and projected into 2D using t-SNE. The axes do not have any particular meaning in projections like these. (b): The average feature values for each course.

by 53 students in 14 weeks 27 features (see again Table 3.1) related to readability metrics, lexical features, and information theoretic entropy to analyse all summaries. We then visually inspected the resulting 27-dimensional characterisation. To enable this, we used t-distributed stochastic neighbour embedding (t-SNE) (Maaten and Hinton, 2008) to project the data-points into 2D. The t-SNE reduction process attempts to preserve the distances in the high-dimensional space as much as possible.

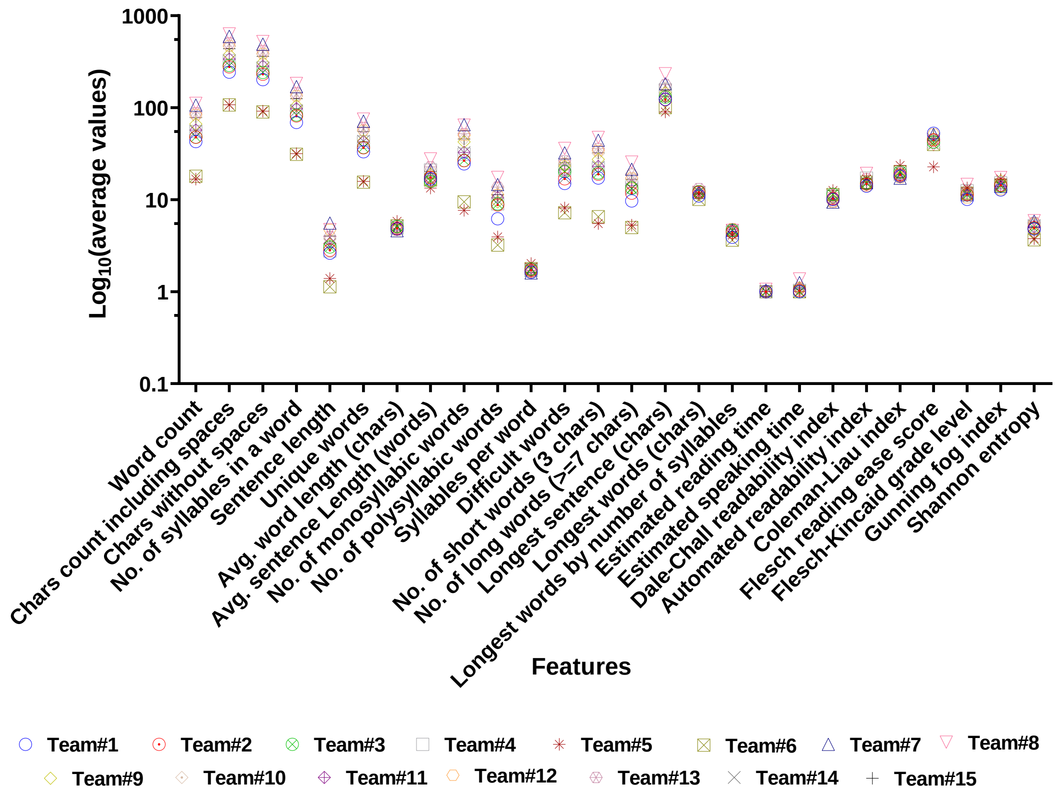
To facilitate the interpretation, we added (before employing t-SNE) to each grouping the respective Euclidean average as each group's centre. Consequently, the projections unavoidably vary slightly. We highlight a few interesting observations as follow:

Grouping by students' courses: The student summaries grouped by courses are shown in Figure 3.2a. These courses are taught to graduate students (Course #2) and undergraduate students (Course #1 and Course #3). Also, the student projects involved in these courses are categorised as either industrial projects (Course #1 and Course #3) or non-industrial projects (Course #2). It is apparent from the distribution of the summaries that Course #2 sits apart at the bottom and far from other groups while the two other courses are close to each other at the top. The distribution of these courses reveals that the summaries generated by the graduate students (with projects categorised as non-industrial) have different text properties while the two other courses have similar text properties as depicted in Figure 3.2b. For example, the majority of the average textual features of the students summaries from Course#2 had fewer features values, such as word count, sentence length, and summary readability, compared to average features values calculated from the students summaries from Course#1 and Course#3. This variation in the summary properties can be attributed to many factors, including type of project (industrial/non- industrial projects), education level (undergraduate/graduate students), and writing style (students in Course #2 are less likely to be native English speakers).

Grouping by project teams: As shown in Figure 3.3a, the summaries are grouped by teams from each course. Summaries produced by Teams #5 and #6 have similar text properties. In the same manner, Teams #4, #11, and #14 have similar text properties, but they belong to two different courses (Course #1 and Course #3, respectively—although these courses are different instances of the same course offered in different years). Teams #8 and #5 have the highest and lowest average values, respectively, across all teams in terms of some of the features calculated as depicted in Figure 3.3b . A review of the members

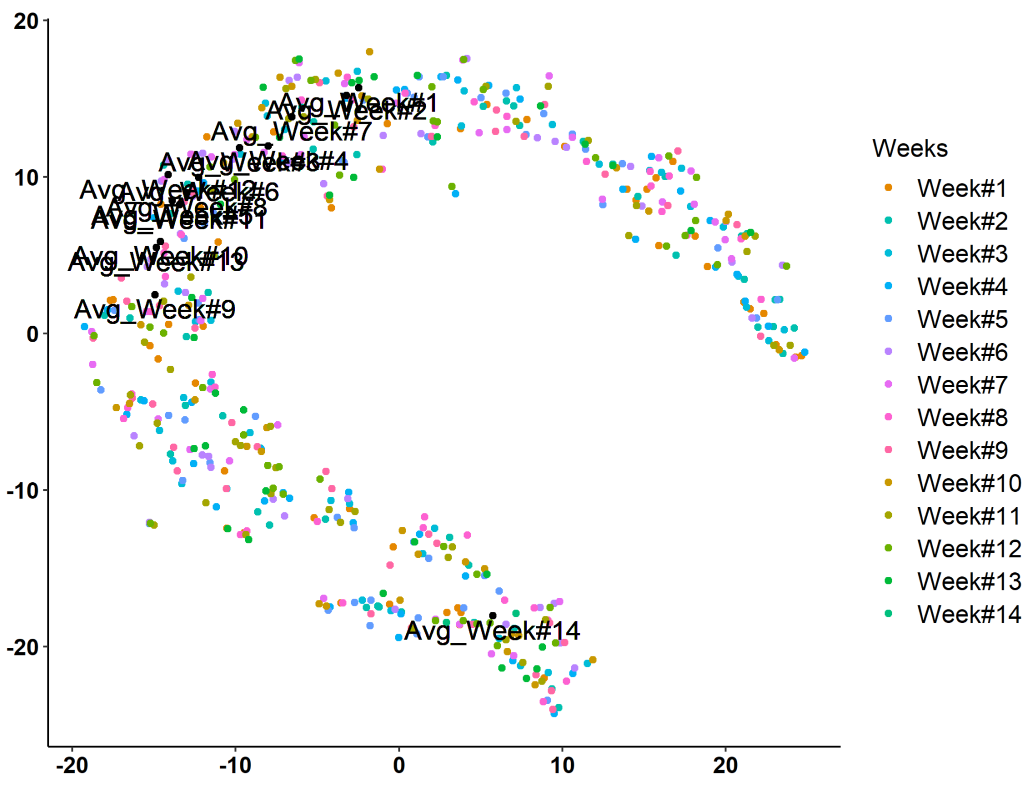


(a)

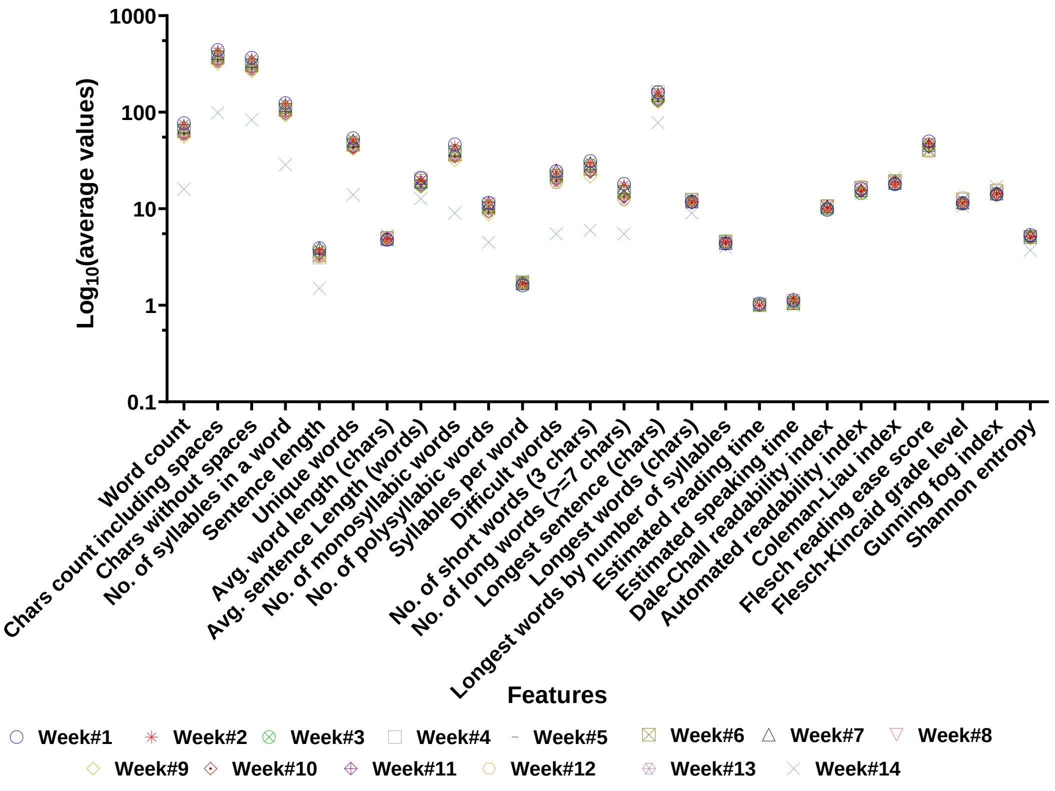


(b)

Figure 3.3. (a): 27 text-based features of the students summaries grouped by teams and projected into 2D using t-SNE. The axes do not have any particular meaning in projections like these. (b): The average feature values for each team.



(A)



(B)

FIGURE 3.4. (a): 27 text-based features of the students summaries grouped by weeks and projected into 2D using t-SNE. The axes do not have any particular meaning in projections like these. (b): The average feature values for each week.

of Team #5 found that most of the team are non-native English speakers, unlike those in Team #8, and thus certain features calculated, such as word count, average sentence length, and unique words, are less compared to the same features calculated for members of Team #8.

Grouping by weeks: Figure 3.4a shows that summaries written in later parts of the semester appear to have different textual features compared to the initial weeks. In the initial weeks, students have to do considerable work to develop their projects compared to the advanced weeks, where the students are close from finalising their projects, resulting in less development activities performed in their projects. This observation can be seen in the features calculated (Figure 3.4b) from their summaries which shows that the majority of the features values have less average values compared to the initial weeks. Besides, summaries created by the students who belonged to different teams seem to exhibit similar text properties in different weeks. For example, weeks #1 and #2 have text properties that are very close to each other. We noted the same behaviour in weeks #10 and #13.

A brief summary: Utilising t-SNE to interpret the student summary data at different grouping levels, using the 27 features, shows that features of the text depend on many factors, such as the type of project, education level, and the weeks in which the summaries were written. Each of these factors therefore influences text features, such as average sentence length, unique words used and the amount of information the summaries may contain.

3.3 Software Artefacts Describing Developers' Activities

We defined our gold-standard summaries and characterised them in the previous section (Section 3.2) to understand their general textual properties. However, generating automatic summarisation also requires a collection of documents as input. Therefore, we first needed to identify suitable artefacts for generating multi-document summaries from heterogeneous software artefacts in a given time frame. In particular, we searched for the types of software artefacts that best described developers' development activities to use as input sources for the summarisation approach. Hence we inspected the student summaries for content related to the GitHub artefacts (Section 3.3.1). Then we investigated the relationship between different artefacts to determine how commonly they related to the developers' activities in the summaries (Section 3.3.2).

3.3.1 Identifying software artefacts

As mentioned in Section 3.2.1, our gold-standard summaries were collected from three different instances of the three courses taught to students in two consecutive years (2017 and 2018). Because some courses were still running or going to be taught when we started identifying the software artefacts (in Semester 2, 2017), we used the available 209 summaries collected from the students in Semester 1 of 2017. The 209 student summaries were produced by 22 students working on six capstone projects. The students were working in teams of three or four towards their Bachelor degree (15 students in total) or towards their Masters degree (7 students in total).

The 209 summaries were manually annotated to identify the types of software artefacts they referred to. To do this, we matched the text in the summaries provided by the students week by week with the text found in the software artefacts of the corresponding student projects. The matching process considered the time relation between each summary's creation and the corresponding creation time of the software artefacts. For example, summaries generated in week one were matched with corresponding text in the software artefacts for that week. However, as the students often used different terms in their summaries to those found in the artefacts' text, it was sometimes impossible to exactly match the text found in the summaries with the text in the artefacts. To solve this issue, we determined the overall meaning of each sentence in the student summaries and matched it with text found in the corresponding software artefacts. After inspecting all of the summaries provided, six types of software artefacts were found to be related to the development activities described in the student summaries. These were issues, pull requests, milestones, README files, wikis, and commits.

The annotation process undertaken to determine the types of artefacts also took into account texts found in the sub-artefacts of each of the aforementioned six types of artefacts. For example, each issue (GitHub's bug tracker) created by a developer may result in two different kinds of artefacts (along with the initial creation of the issue): the issue body that describes what the issue is about, and feedback from other developers on this issue in the form of comments. Similarly, pull requests, which allow developers to submit their contributions to an open project's development, may result in four different kinds of artefacts in response. These are the pull requests bodies, comments on each pull request, reviews of

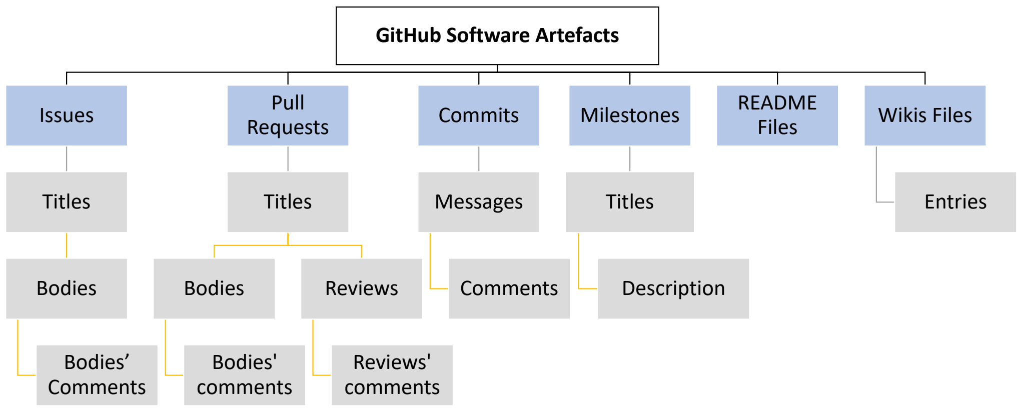


FIGURE 3.5. Artefacts and sub-artefacts related to students' development activities.

the pull requests changes, and subsequent comments on the reviews. Figure 3.5 shows a complete list of the six artefacts and their related sub-artefacts. In total there are 14 types of artefact that reflect the students' development activities included in 209 summaries. These types, as shown in the figure, are issues (titles, bodies, and comments), pull requests (titles, bodies, comments, reviews, and review comments), commits (messages and comments), milestones (titles and descriptions), README files, and wiki entries.

Table 3.2 shows the total number of artefacts created by students in their projects and the number of times the content of each artefact was referenced in the student summaries. For example, the first row in the table shows that four students created 101 issues artefacts, including issue titles, issue bodies, and issue comments, while developing their project over 14 weeks. During that time, the students produced a total of 40 summaries. Inspecting the text in the 40 summaries for matches yielded 103 issues that referenced the students' development activities. Here we should note that a single artefact can be referenced more than once in the student summaries. For example, sentences in summaries created in a particular week can reference a text in a particular artefact more than once. Therefore, some cases in the table (e.g., issue: # of matches = 103) show that the number of the linked artefacts to the summaries is greater than the total number of artefacts (e.g., total issues = 101) created in a project.

Table 3.2 illustrates that contents found in issues were more referenced in the student summaries, followed by commits, pull requests, wikis, milestones,

TABLE 3.2. Number of software artefacts created per project and the number of times referenced in the student summaries.

Project	**# of Students**	**# of Summaries (14 Weeks)**	**Issues**		**Pull Requests**		**Milestones**		**README**		**Wikis**		**Commits**	
			Total	**# Matches**	**Total**	**# Matches**	**Total**	**# Matches**	**Total**	**# Matches**	**Total**	**# Matches**	**Total**	**# Matches**
1	4	40	101	103	74	92	8	15	1	5	17	31	344	119
2	4	37	122	84	34	30	11	37	1	4	30	33	145	39
3	3	36	79	56	53	35	7	35	1	0	41	26	309	57
4	4	30	133	69	96	29	8	13	1	2	20	31	316	53
5	3	28	195	161	197	80	6	24	1	1	31	27	617	98
6	4	38	96	82	61	26	6	41	1	0	36	43	223	51
Total	**22**	**209**	**726**	**555**	**515**	**292**	**46**	**165**	**6**	**12**	**175**	**191**	**1,954**	**417**

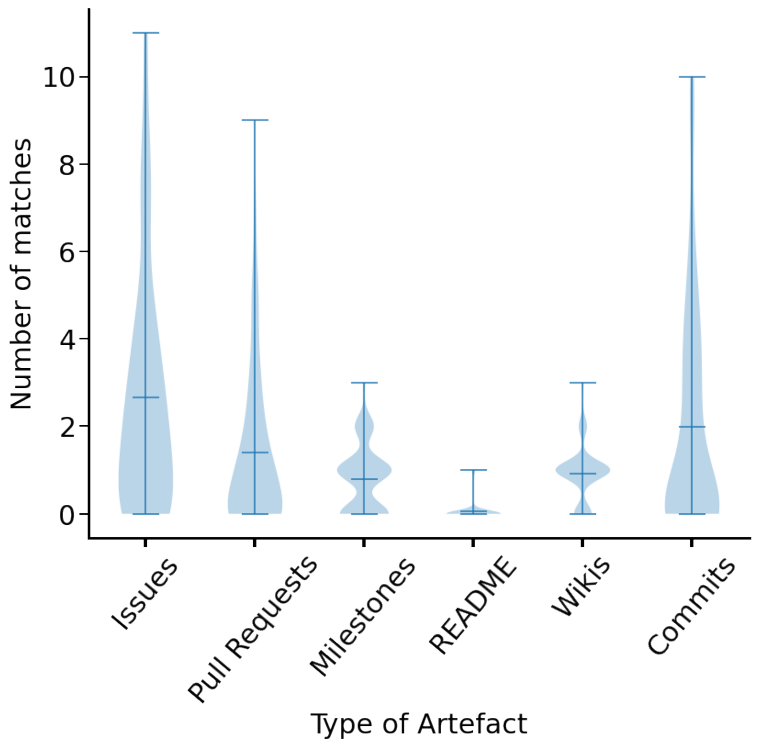


FIGURE 3.6. Distribution of number of matches between text in each of software artefacts and student summaries.

and README files. However, to validate this conclusion, we tested whether the differences between these artefacts are statistically significant using Wilcoxon signed-rank test. The Wilcoxon test assesses whether the mean rank between two matched samples is statistically different. Based on the Wilcoxon test, we found that all differences between pairs of artefacts are statistically significant (p-values < 0.05). Figure 3.6 illustrates the distribution of the number of text matches between each of the artefacts and the student summaries over 14 weeks and for six projects. The figure shows that most of the students' development activities related to issues (mean $= 2.65$) followed by commits (mean $= 1.99$) and pull requests (mean $= 1.39$).

Issues artefacts are typically used to keep track of a project's tasks, enhancements, and to fix bugs. Thus, it is not surprising that the content of issues-type artefacts was extensively used in student summaries at their projects' development stage. Pull requests are commonly used in a project to allow for collaboration between team members. For example, developers can use pull requests to tell their team members about changes made to a project repository. The pull requests are used to authorise other team members to discuss potential changes before these changes are implemented. However, the content of the student summaries was less related to pull requests than to issues artefacts.

This may indicate that changes made to their projects were discussed less by the students and thus explain the lower amount of content matched for pull requests compared with issues in the student summaries.

In addition, milestone artefacts appeared to be less attractive to students than either issues, pull requests or commits in the content of their summaries. Developers use milestones to track progress on groups of issues or pull requests and to provide short information, such as the milestone's description and its due dates; this may explain the fewer instances of matched content within the summaries.

Finally, a GitHub repository project includes a wiki section where developers can create various documents to share information about the project, including how the project was designed or to explain its core principles. On the other hand, a README file can be created automatically during a project setup; it allows the developer to briefly gives an overview of the project and to explain how other developers can use and contribute to the project. Rather than using README files, the students often used the wikis entries as a meeting-area where they could discuss and regularly document their development activities, making it a useful source when creating their summaries.

A brief summary: 209 of the summaries in gold-standard, which were manually annotated for the type of artefacts, revealed 14 types of artefacts describing developers' development activities, which could be used as input sources for our multi-document summarisation. These types are issues (titles, bodies, and comments), pull requests (titles, bodies, comments, reviews, and reviews' comments), commits (messages and comments), milestones (titles and descriptions), README files, and wiki entries. Among these types of artefacts, we found that most of the student summaries linked to issues, followed by commits and then pull requests.

3.3.2 Relationships between software artefacts

Investigating the relationship between the contents of pairs of software artefacts can give insight into common development activities between artefacts, as referenced in the 209 student summaries. This will help us to then develop techniques to produce multi-document summaries from heterogeneous artefacts. For example, developers may use one artefact to describe briefly some development activity and then give additional information about the same activity in another artefact. On the other hand, there might be cases where no common development activities are shared between different artefacts. Therefore, we assumed that the artefacts that share common activities would provide summaries containing more details about the same development activity, while adding content from another artefact that has no such common activity may introduce new content and add diversity to the automatic summaries.

The correlation between pairs of these artefacts was calculated using Pearson product-moment correlation and results are presented in Figure 3.7. In each cell we show the Pearson p-value and the correlation coefficient, respectively. The relationship between some pairs of artefacts is statistically significant ($p < 0.05$). For example, there is some degree of positive correlation between the content of issues and the content of each of other artefacts referenced in the student summaries, with the exception of the README files where no relationship was shown to exist. Similarly, a strong positive correlation between pull requests and commits indicates that the students commonly discussed the same activities that were performed in a particular week in these artefacts and usually referred to contents of these artefacts when writing their summaries.

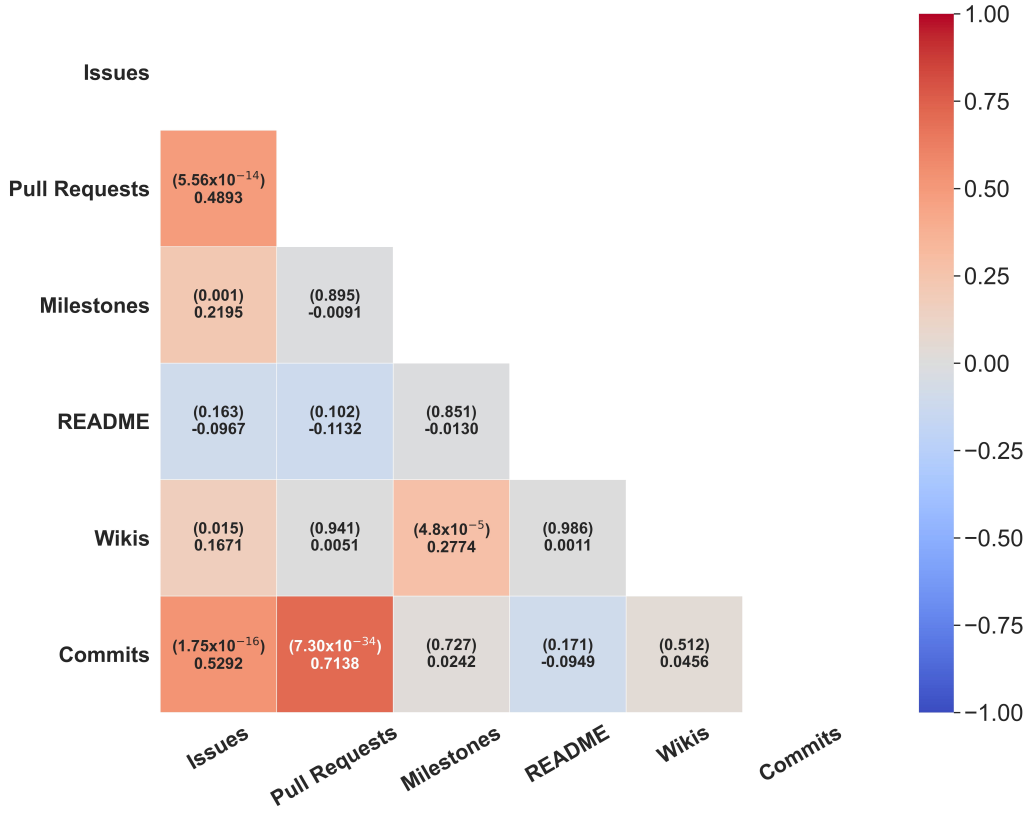


FIGURE 3.7. Pearson product-moment correlation between pair of software artefacts.

We further analysed the association between the software artefacts to extract useful information from the 209 summaries by using association rule mining. We used the Apriori algorithm and the *arules* (Revelle and Revelle, 2015) library to find the frequent item set that could describe the rule between the set of software artefacts and the related student summaries. The association rule takes the form of $A => B$, where $A \in I$, $B \in I$, $A \cap B = \emptyset$, and I is the set of items. The support of the rule $A => B$ determines the probability that A and B occur in all of the observations, while the confidence determines how often A and B appear in an observation that contains A. As the rules generated by the Apriori algorithm are dependent on the input parameters of support and confidence, choosing these inputs is a matter of trade-off.

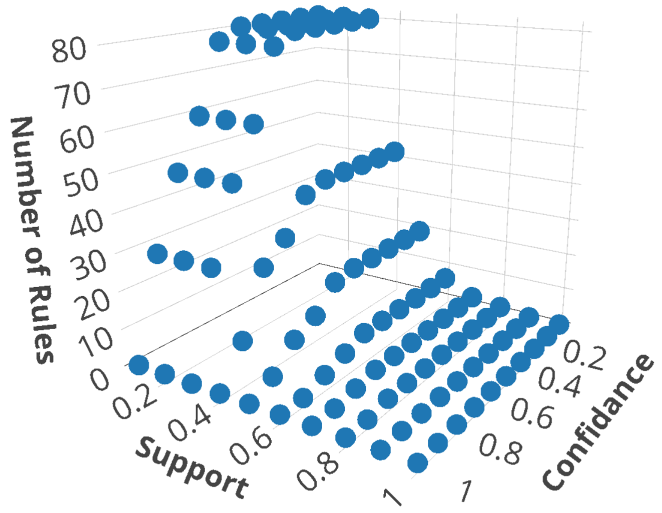


FIGURE 3.8. Total number of rules that can be generated from the summaries, based on different values of support and confidence.

Therefore, sliding confidence and support values were used, starting at 0.1, as shown in Figure 3.8. As the data used in this work were sparse, a minimum support value was taken as 0.1, and the confidence value was taken as 0.6 in the analysis. This provided us with a satisfactory number of rules (77 rules).

The partial result of the relationship between the software artefacts and the student summaries is shown in Table 3.3. The rules were sorted in decreasing order based on the confidence value. Association rule mining reveals information about the relationship between the content of different software artefacts in student summaries. For example, the first rule in Table 3.3 identifies a high

Table 3.3. Top 10 association rules generated using a support value of 0.1. and a confidence value of 0.6.

Rules	**Support**	**Confidence**	**Count**
{1} {Commits,Milestone,Wiki} => {Issues}	0.335	0.985	66
{2} {Commits,Milestone,PullRequests,Wiki} => {Issues}	0.324	0.984	64
{3} {Milestone,PullRequests,Wiki} => {Issues}	0.345	0.971	68
{4} {Commits,Milestone,Wiki} => {PullRequests}	0.329	0.970	65
{5} {Commits,Issues,Milestone,Wiki} => {PullRequests}	0.324	0.969	64
{6} {Commits,PullRequests,Wiki} => {Issues}	0.416	0.964	82
{7} {Commits,Milestone} => {PullRequests}	0.370	0.960	73
{8} {Commits,Issues,Milestone} => {PullRequests}	0.350	0.958	69
{9} {Commits,Wiki} => {Issues}	0.441	0.956	87
{10} {PullRequests,Wiki} => {Issues}	0.477	0.949	94

probability of selecting content from issue artefacts to form a summary if such content was also selected from commits, milestones, and wiki entries. Conversely, with a low probability, rule #5 shows that content from pull requests could take part in summary formation if the content from commits, milestones, and wiki entries was also selected.

A brief summary: Investigating the relationship between pairs of 14 artefacts referenced in the gold-standard summaries can reveal the common development activities. We found strong positive correlation between pull requests and commits indicates that students frequently discussed the common activities in these artefacts during a given week and frequently refer to the contents of these artefacts when writing their summaries. In addition, analysing the association between the software artefacts using association rule mining, we found that selecting content from issue artefacts to form a summary is highly associated with content being selected from commits, milestones, and wikis entries.

3.4 Validation and Characterisation through Large-Scale Analysis

We discussed in Section 3.3.1 the types of software artefacts that reflect the development activities in the student summaries, which could then be used as input sources to automatically generate multi-document summaries in a given time frame. Then, using large-scale analysis of GitHub repositories, we first tested whether GitHub developers commonly use these types of artefacts during their projects' developmental life-cycles (Section 3.4.1). We then investigated the textual characteristics of the artefacts (Section 3.4.2).

3.4.1 Validating the existence of software artefacts

Our previously identified software artefacts contain README files and wikis entries. README files are often automatically set by the GitHub developers, while wiki files represent external sources of information. As a consequence, README and wiki files may not reflect any of the project development activities. Therefore, we needed to validate that the GitHub developers utilise the six artefacts and their sub-artefacts, including README files and wikis entries, by performing a large-scale analysis of how commonly developers use these artefacts.

Table 3.4. Number of randomly sampled projects from both data sets.

Type of projects	Population	Sample (Conf.95% , Conf. inter. 1%)
Engineered software projects	446,863 (24%)	9,402
Non-engineered software projects	564,467 (30%)	9,443

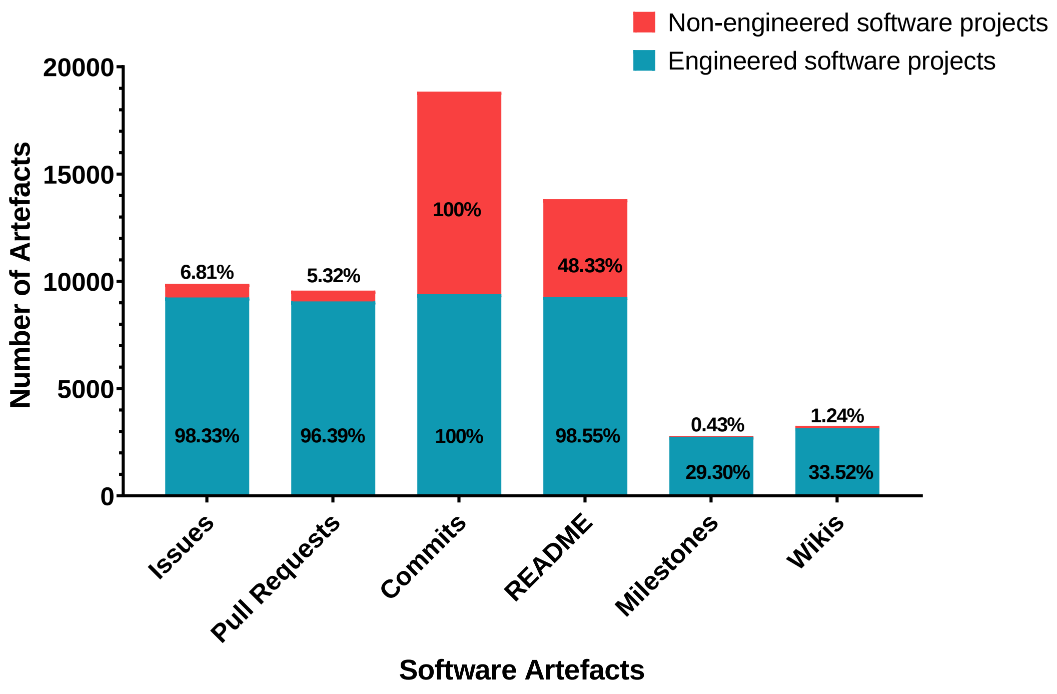


Figure 3.9. Number of software artefacts found in engineered and non-engineered software projects.

To select repositories for our study, we utilised the RepoReaper tool (Munaiah et al., 2017). The purpose of building the tool was to predict whether a project is an engineered software project or a noise project (i.e., non-engineered software projects) using two classifiers: Random Forest classifier and Score-based classifier. The two classifiers were trained with organisation and utility data sets. As we aimed at knowing whether GitHub developers commonly use our defined types of artefacts during their project development, we formed two data sets, and for each data set, we randomly selected projects from the RepoReaper dataset, which contains 1853,207 projects. The first data set we formed included engineered software projects predicted by the Random Forest classifier, and the second data set contained noise projects predicted by both the classifiers. We selected the engineered software projects classified by the Random Forest classifier because its precision scores, when trained with utility and organisation data sets, scored higher values compared to the score-based classifier trained with the same data sets (Kalliamvakou et al., 2014). Table 3.4 shows the population of both data sets and the sample size.

To determine whether a project in each data set (engineered or non-engineered projects) contains a particular type of artefact, we built our web crawler tool. The tool counts the number of issues, pull requests, commits, README files, milestones and wikis found in each project in each dataset. The variance in the number of artefacts between the two data sets can confirm whether developers interact with these artefacts to develop software applications. For example, software developers usually use the issue artefacts to track bugs in their applications. On the other hand, non-engineered projects are noise projects (i.e. assignment projects), and developers usually do not interact with such artefacts enough to provide information about their development activities. Therefore, we needed to track such variance as we particularly wanted to generate multi-document summaries from software artefacts that described the developers' development activities.

Our result, depicted in Figure 3.9, reveals that the types of software artefacts that reflected developers' development activities in the summaries were found more in engineered software projects than in non-engineered software projects. We thus conclude that these artefacts and their sub-artefacts can be utilised as input sources to generate our automatically multi-document summaries describing the developers' development activities. Note: both types of projects have the same amount of commit artefacts because every project hosted in GitHub has an initial commit, an activity always introduced by the developer to start a new project.

A brief summary: Our result shows that developers of engineered software projects do interact with the six artefacts and their sub-artefacts. This confirms that these artefacts can be used as input sources to generate summaries including information about developers' development activities.

3.4.2 Characteristics of GitHub artefacts

To study the characteristics of artefacts text, we first randomly selected 1,038 projects (confidence level 99% and confidence interval 4%) out of the engineered software projects (see Table 3.4) to form the sample data set. Then, we collected the content found in six artefacts and their sub-artefacts using GitHub REST API[2] and our web crawler, which was used to collect the wikis' contents. The types of artefacts we collected were issues (titles, bodies, and comments), pull requests (titles, bodies, comments, reviews, and reviews' comments), commits (messages and comments), milestones (titles and descriptions), wiki entries, and README files.

We first explored our collected 1,038 engineered software projects (see Table 3.5) to reveal the number of artefacts that the developers used during their projects' development life-cycle. Table 3.5 illustrates that developers used the commit artefact in all the projects, followed by README files (100% and 85.55% respectively). In addition, issues and pull requests artefacts were used almost equally (36% for each artefact) by developers across all the projects. Wikis and milestones were the least used among the artefacts (7.04% and 6.75%, respectively). The low percentage of wiki use could show that developers usually do not share information about the project's design or explain its main principles. In addition, the low percentage of milestones could indicate that developers rarely track the progress of issues or pull requests in their projects.

We noted that developers usually provide descriptions for the issues they encounter in their projects and commonly discuss them by providing comments. For example, we found that 10,902 (92.36%) of the issues have descriptions (i.e., issue bodies) and, on average, 2.68 of comments were provided per issue. Similarly, 6,041 (92.24%) of collected pull requests had descriptions (pull requests bodies), and each pull request had one comment on average. Hence, developers use pull requests bodies and comments to discuss their proposed features and receive feedback from collaborators. Before a pull requests is merged into a project, pull requests reviews can also be used to allow collaborators to discuss and comment on the changes proposed in a pull requests, approve these changes, or even request further changes. Across the 1,038 collected projects, 5.49% of

[2]GitHub API REST: `https://docs.github.com/en/rest`, accessed on September 3, 2018

TABLE 3.5. Number of artefacts found in engineered software projects data set.

Software artefact	**Projects have artefact**	**# of artefact**	**Projects do not have artefact**
Issue Titles	36.81%	11803	63.19%
Issue Bodies	-	10902	-
Issue Comments	-	31746	-
Pull Request Titles	36.04%	6549	63.96%
Pull Requests Bodies	-	6041	-
Pull Requests Comments	-	8771	-
Pull Request Reviews	-	360	-
Pull Request Reviews' Comments	-	1252	-
Milestone Titles	6.75%	283	93.25%
Milestone Description	-	109	-
Commit Messages	100%	272320	0%
Commit Comments	-	1254	-
Wiki Files	7.04%	291	92.96%
README Files	85.55%	888	14.45%

the pull requests had reviews and each review artefact received on average 3.47 comments.

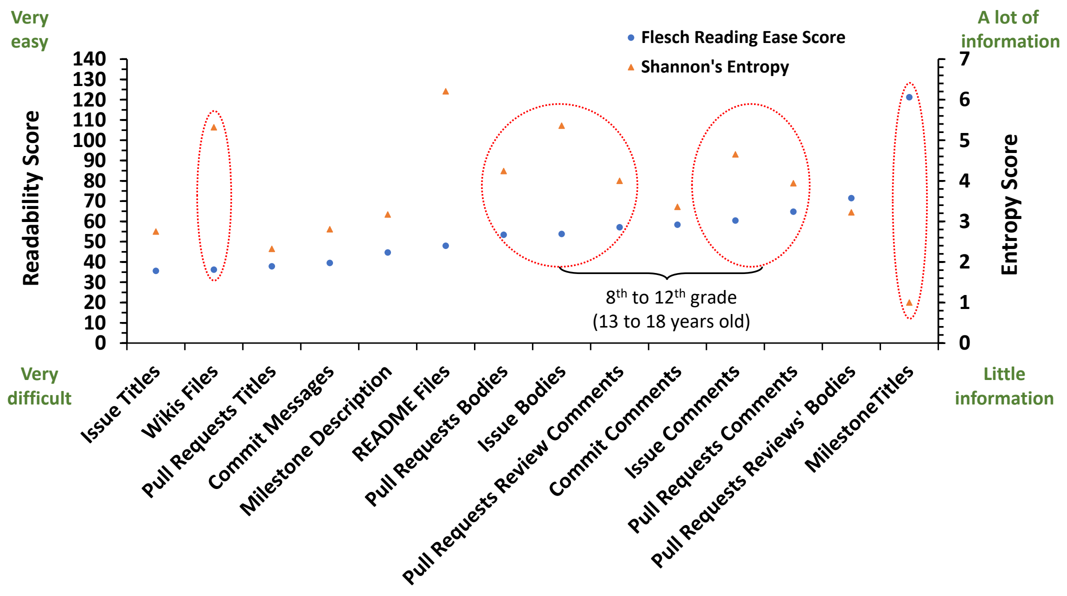


FIGURE 3.10. Shannon's entropy and Flesch reading ease scores for artefacts texts.

Next, we analysed the text in our collected artefacts (as shown in Table 3.5), looking for two features: entropy and readability. To calculate readability of the artefacts' texts we used the Flesch reading ease metric. We applied Shannon's entropy to gauge the amount of information in each artefact's text. For the purpose of achieving rich multi-document summarisation, we are interested in identifying and including artefact texts that are easy to read and that contain lots of information.

Figure 3.10 shows the median values of the Flesch reading ease scores and Shannon's entropy values for each of the software artefacts. For example, the Flesch metric test found that texts within issue and pull requests comments, issue and pull requests bodies, pull requests reviews' comments, and commit comments ranged from *easy to read to fairly difficult to read* and could thus be understood by 13-year-old to 18-year-old students (readability median values ranged from 53.4 to 64.8). Comparing the median of readability scores of the artefact texts with the median value of the student summaries (median value = 46.5, see Figure 3.1), we could conclude that the text in the artefacts would be easier to read and understand.

Similarly, the entropy test applied to these aforementioned artefacts gave median values between 3.35 and 5.36. In comparison, the median score of the student summaries was 5.33, which is a bit higher than some of the artefacts' scores, indicating that the student summaries contained a lot of information. Further analysis showed an opposite trend between entropy and readability in wikis and milestone titles across all the artefacts. The wikis texts scored a median value of 36.2 (i.e., difficult to read) on the readability metric, while they scored 5.31 (contained much information) on the entropy metric. On the other hand, milestone titles scored the lowest median value (1.00) on the entropy metric, while their readability scored the highest median value (i.e., very easy to read). The wiki contents contained lengthy text with median values of 78 words, 1.9 syllables per word, 13 sentences and 46 unique words; this was favourable for their entropy but not for readability. On testing milestone titles, these were found to be quite short (median values: 3 words, 1 syllables per word, 1 sentence, and 2 unique words). Therefore, there was negative impacts on the amount of information they contained but not for readability.

A brief summary: Analysing the artefacts produced within 1,038 engineered software projects, we found that commits and README files were the most used artefacts; 100% and 85.55% of the projects contained these artefacts, respectively. In addition, 36% of the projects included issues and pull requests, while wikis and milestones were used less by developers: Wikis and milestones were included in only 7.04% and 6.75% of the projects, respectively. In analysing text quality of our collected artefacts, using Shannon's entropy and Flesch reading ease metric, we found that issue and pull requests comments, issue and pull requests bodies, pull requests reviews' comments, and commit comments were easier to read than those of the student summaries and contained a reasonable amount of information. Therefore, for our summarisation approaches, selecting contents from these artefacts shows the potential to generate readable summaries contain a reasonable amount of information.

3.5 Characteristics of Source Code Comments

Section 3.3.1 discussed the types of software artefacts contributed in the formation of the student summaries. However, there was no clear evidence that the student summaries referenced information from comments in the source code. Code comments are an essential part of the maintenance and the development of software because they assist developers in comprehending programs (Takang, Grubb, and Macredie, 1996; Roehm et al., 2012). This has led researchers to endeavour to automatically generate comments for source code, that is, to convert the code into a natural language descriptor that describes a program's functionality (Haiduc, Aponte, and Marcus, 2010; Sridhara et al., 2010). We conjecture that functionality implemented in source code would play an important role in a summary of developer activity, and we therefore investigate whether the characteristics of code comments are suitable for them to be considered as a source for the summaries as well.

As we are not aware of any related work that analyses comments manually written by developers in a detailed way, we studied the potential of code comments to be used in summarisation by assessing the comments in Java source code against three types of knowledge about primitive variables (see chapter 5). This section provides a descriptive analysis of code comments extracted from 980 software engineered projects, written in seven common programming languages: C/C++, C#, Java, JavaScript, Python, PHP and Ruby. The analysis will provide an understanding of the comment text properties using 26 text-based features.

To select repositories for our study, we used 1,038 that were classified as engineered software projects (discussed in section 3.4.2). We then filtered out 58 projects that had no source code files written using these programming languages, or else had been made private. We then extracted the source code comments from the remaining 980 projects using the "Comment Lister" tool[3], which was built to automatically extract code comments that support the aforementioned seven programming languages (Hata et al., 2019). The total number of comments extracted was 1,868,073 from 115,405 files.

TABLE 3.6. Number of comments extracted from different types of files found in 980 engineered software projects written in seven programming languages.

No.	File type	Total number of files	Total number of comments	Total number of projects
1	C/C++[4]	21,407	643,031	190
2	Java	37,838	423,989	230
3	PHP	23,084	294,154	195
4	JavaScript	10,612	221,206	236
5	Python	13,982	170,190	262
6	C#	3,649	76,860	49
7	Ruby	4,833	38,643	181
Total	**7**	**115,405**	**1,868,073**	**1,343**

[3]Comment extractor tool: `https://github.com/takashi-ishio/CommentLister`, accessed on 6 July 2021

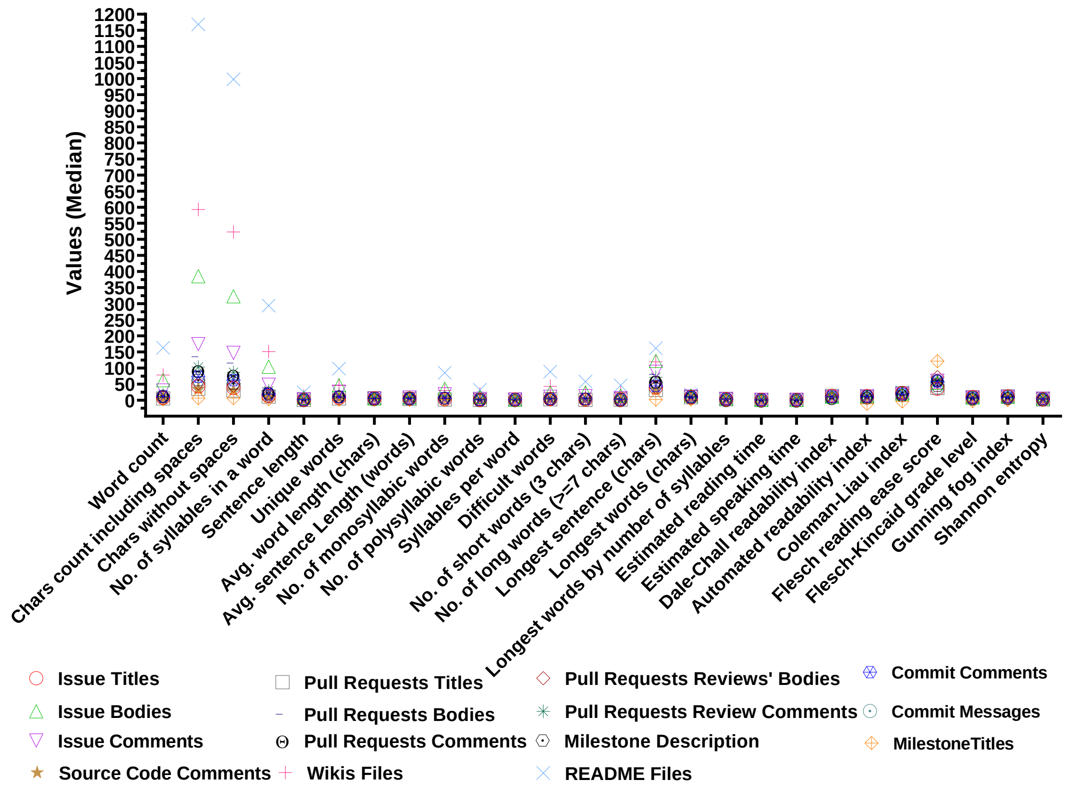
[4]File types, including .c, .cc, .cp, .cx, .cxx, .c+, .c++, .h, .hh, .hxx, .h+, .h++, .hp, and .hpp

Table 3.6 shows the number of extracted comments from the source code file found in 980 projects. The first column of the table shows the file type from which the comments were extracted. The second column, shows the total number of files for that file type. The third column shows the total number of comments extracted from all the files for that file type. The last column shows the number of projects containing the file type. Note that a single project can have files written in one or more languages, which can explain why the total numbers of projects (1,343) is not equal to the total projects collected (980 projects). From the table, we can see that across all the projects, developers most often provided comments for code written in C/C++ followed by Java programming languages.

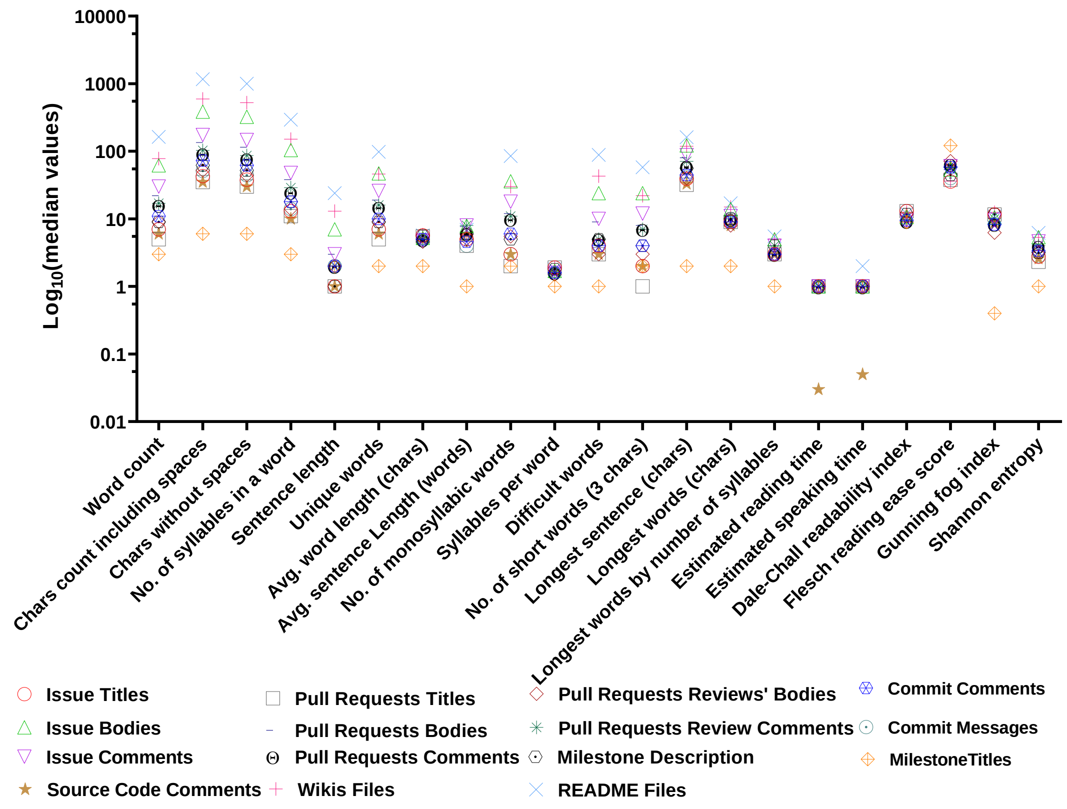
We next analysed the texts of the comments to explore their textual features compared to other software artefacts, as depicted in figures 3.11a and 3.11b. Figure 3.11b includes fewer features than Figure 3.11a because not all feature values can be plotted on a log scale. In addition, we plotted the median values in both figures since the median values are less sensitive to outliers.

From the Figure 3.11, it is clear that most artefacts' texts share common textual properties. For example, developers tend to write comments consisting of a single sentence. Comparing the median number of sentences in comments to the number of sentences found in other artefacts, we found that issue titles, pull requests titles, and milestone descriptions were generally one sentence in length. This was expected as these artefacts usually contain only brief texts. In contrast, all the other artefacts were found to consist of 2 to 3 sentences, except issue bodies, wikis, and README files, which were found to usually consist of 7, 13, and 24 sentences, respectively. This is because the nature of these artefacts would require developers to provide more details about their development activities. Concerning the average sentence length by the number of words, we found code comments, pull requests comments and milestone descriptions had median value of 6 words per sentence. The highest number of words per sentence was found in pull requests reviews' comments, issue comments (median values of 8 words and 7.9 words, respectively). In contrast, milestone titles had the lowest average sentence length—the comments tended to be single words only.

We next explored the number of difficult words (i.e., words not belonging to the Dale list of 3,000 familiar words) found in the text of the artefacts. Thirty percent of words in pull requests comments, issue comments, milestone titles, pull requests reviews' bodies, commit comments, and issue bodies were found to contain difficult words, compared to 50% to 60% of words found in code



(A)



(B)

FIGURE 3.11. Text characteristics of source code comments and compared to other 14 types of software artefacts. (a): Median values of 26 text features. (b): Log_{10} of the median values of 21 text features—features with median values less than or equal to zero were excluded.

comments, README files, wikis entries, milestone descriptions, issue titles, commit messages, and pull requests titles. This variance in the percentage of difficult words may be because developers used specific words related to their activities to describe their projects' development activities.

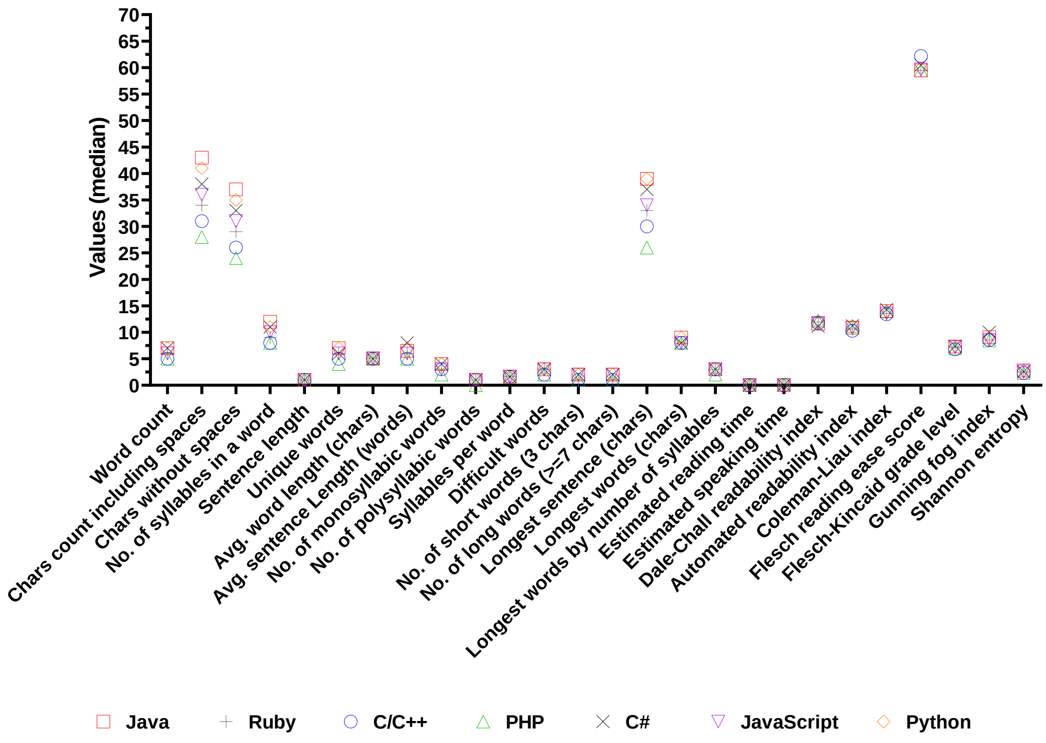


FIGURE 3.12. Text characteristics of source code comments grouped by file types.

In addition, the median of the entropy values and the Flesch reading ease scores indicates that the comments contained little information (entropy: 2.58) and that the readability of text was *easy to read*. Comparing the entropy value of the comments to those of the other artefacts, we found that pull requests titles (median: 2.32), issue titles (median: 2.75), and commit messages (median: 2.80) were the artefacts that shared the closest entropy values with the source code comments. This could be because these artefacts share a common number of words (issue titles and comment messages had a median score of 7 words and pull requests titles had a median score of 5 words), the number of unique words (issue titles and comment messages had a median of 7 unique words and pull requests titles had a median of 5 unique words), and the number of sentences (issue and pull requests titles had a median score of one sentence, and comment messages had a median score of 2 sentences). On the other hand, README Files scored the highest entropy value (median: 6.20) where the milestone titles (median: 1) scored the lowest value among all the artefacts. Furthermore, concerning the readability of comments, we found that the comments scored 60.82 on the Flesch reading ease metric and comparing this value with the readability of other artefacts, we found that issue comments and pull requests comments were also classed as *easy to read.*

Grouping the comments by file types to investigate the characteristics of the comments based on their text features is depicted in Figure 3.12. From the figure, we can see that the majority of comments share the same text features, such as average word length by the number of characters, average sentence length by the number of words, indicating that the developers' style in writing the comments is similar among the seven programming languages.

A brief summary: Analysing 1,868,073 comments extracted from source codes written by seven programming languages reveals that the comments share text features, such as the number of sentences, the average number of words per sentence, difficult words, entropy and readability, to those features of other artefacts. Code comments had a median of one sentence, containing an average of 6 words per sentence; 30% of words in the comments were considered difficult; they contained less information (entropy: 2.58) but were easy to read and understand. In addition, developers' writing styles did not differ much with any of the programming languages when documenting the source code.

3.6 Threats to Validity

Our gold-standard summaries were created by students (undergraduate and graduate students) who may not have experience in software development. Therefore, involving professional developers to create gold-standard summaries may reveal different types of artefacts or different characteristics of the summaries compared to our results discussed in Section 3.3.

Considering the duration of each course in which the students are involved, which is usually a semester, the students can gain good acknowledge of interacting with the GitHub platform and gain a good experience about software development. Therefore, creating gold-standard summaries using students developers should have no effect on our results for identifying the software artefacts referenced in these summaries.

In addition, our results shown Figure 3.2 revealed some variance of the summaries characteristics that were written by 43 undergraduate students (Course#1 and Course#3), and those summaries characteristics written by 10 graduate students (Course#3). Due to the low number of students involved in Course#2, our results drawn from Figure 3.1 should be valid.

3.7 Implications

Our gold-standard summaries were created by undergraduate and graduate students involved in different courses towards their capstone projects. A careful look at Figure 3.2, the summaries have different text characteristics, suggesting that determining the ideal characteristics of the gold-standard summaries are subject to various factors, including the participants' educational background and the types of the projects. Therefore, developers need to be aware of these factors to generate automatic summaries whose properties close in style to the human-written summaries, such as the length and the readability of summaries.

3.8 Conclusion

Gold-standard creation plays a vital role in multi-document summarisation. To the best of our knowledge, there are no resources available that can generate multi-document summarisation from heterogeneous software development artefacts. In this chapter, we created our gold-standard summaries, studied their textual features, and then used them to manually identify a collection of software artefacts recording developers' development activities. In addition, we validated the existence of the identified software artefacts through undertaking a large-scale analysis of engineered software projects. Following that, as the gold-standard summaries did not contain evidence that developers referenced source code, we explored the features of those textual artefacts (i.e., the source code comments) written in seven programming languages. These findings provided us with the necessary resources for our study that is reported in the following chapters of this thesis.

The next chapter was published (Alghamdi, Treude, and Wagner, 2020) and presented in the 16th International Conference on Parallel Problem Solving from Nature (PPSN 2020).The conference ranked (A) based on the Australian CORE ranking system. I contributed to the design, implementation, and evaluation of the proposed approach.

Chapter 4

Multi-document Summarisation of Heterogeneous Software Artefacts

4.1 Introduction

In Chapter 3 we identified the types of software artefacts and we also showed that the these are common types used by developers during the development of their projects using the GitHub platform. In this chapter, we present a first framework to create multi-document summaries from heterogeneous software artefacts within a given time frame. In particular, we aimed at an extractive approach, which generates a new summary from documents without creating new sentences. Figure 4.1 provides an overview of the proposed approach.

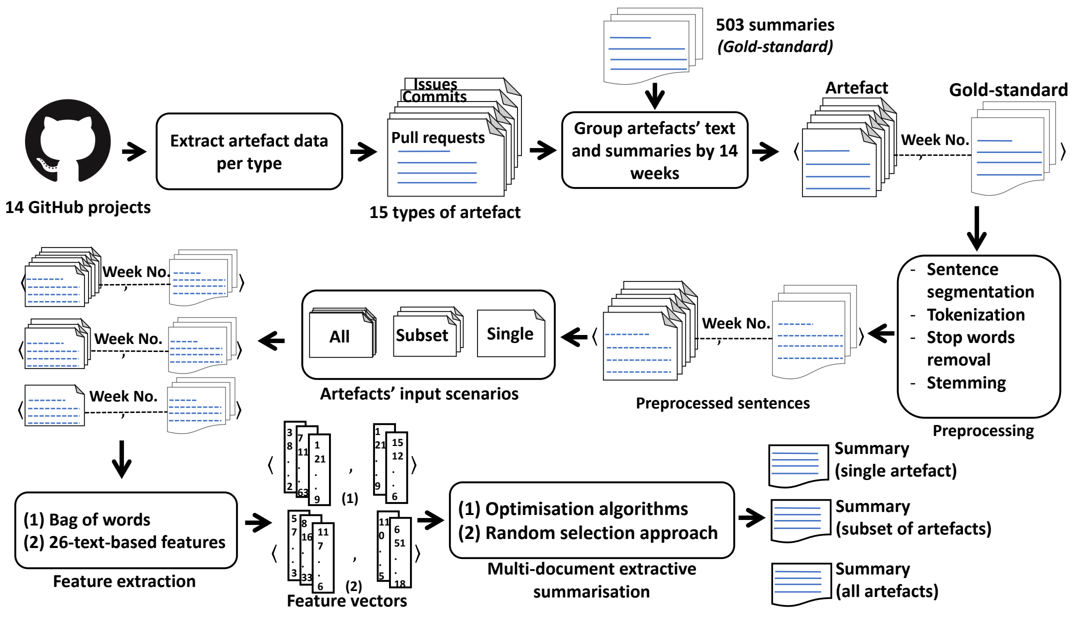
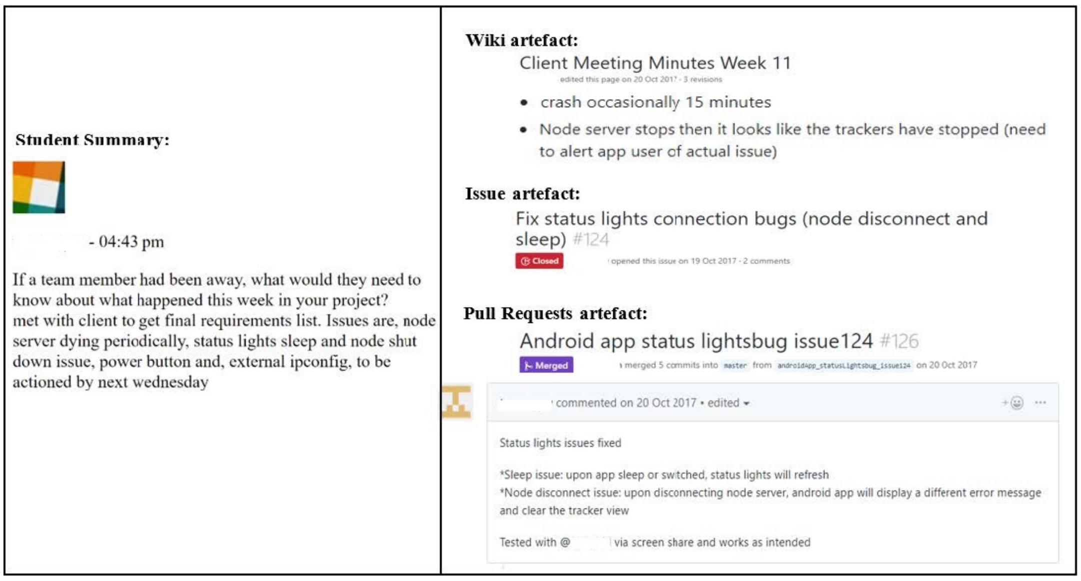


FIGURE 4.1. Overview of the proposed multi-document summaries of heterogeneous software artefacts.

We can illustrate our problem in summarising developers' artefacts by looking at an example of a summary written by a student software developer, alongside the various artefacts that contain parts of the information conveyed in this manually written summary (Figure 4.2).

Let us consider two possible scenarios using the aforementioned example: (1) a developer has been on holiday during this period and would like to be updated, and (2) a new developer joins the team and would like to know what has happened recently in the current software project. In both cases, going through many related artefacts and collecting the most useful information from them can be tedious and time-consuming. It is scenarios like these that we are targeting in our study, as solutions to these problems can ultimately increase the productivity of software developers and reduce information overload (Treude, Figueira Filho, and Kulesza, 2015).


Student Summary:

- 04:43 pm

If a team member had been away, what would they need to know about what happened this week in your project?
met with client to get final requirements list. Issues are, node server dying periodically, status lights sleep and node shut down issue, power button and, external ipconfig, to be actioned by next wednesday

Wiki artefact:

Client Meeting Minutes Week 11
edited this page on 20 Oct 2017 · 3 revisions

- crash occasionally 15 minutes
- Node server stops then it looks like the trackers have stopped (need to alert app user of actual issue)

Issue artefact:

Fix status lights connection bugs (node disconnect and sleep) #124
Closed opened this issue on 19 Oct 2017 · 2 comments

Pull Requests artefact:

Android app status lightsbug issue124 #126
Merged merged 5 commits into `master` from `androidApp_statusLightsbug_issue124` on 20 Oct 2017

commented on 20 Oct 2017 • edited

Status lights issues fixed

*Sleep issue: upon app sleep or switched, status lights will refresh
*Node disconnect issue: upon disconnecting node server, android app will display a different error message and clear the tracker view

Tested with @ via screen share and works as intended

FIGURE 4.2. An example of an anonymised student summary (left) linked to the content of related software artefacts (right).

To devise solutions in such cases, we used a combination of methods from data-driven search-based software engineering (DSE) (Nair et al., 2018). DSE combines insights from mining software repositories (MSR) and search-based software engineering (SBSE). While MSR formulates software engineering problems as data mining problems, SBSE reformulates such problems as optimisation problems and uses meta-heuristic algorithms to solve them. Both MSR and SBSE share the common goal of constantly advancing software engineering. In this study, we suggest to improve software engineering – in particular the creation of software development activities – by mining the created artefacts for summaries.

As discussed in Chapter 2.3, several approaches have been studied for summarising heterogeneous software artefacts. These approaches have mostly focused on summarising a single type of artefact, and they have not taken into consideration the production of summaries in a given time frame. We therefore decided to focus on those two elements: the production of automatic multi-document summaries from heterogeneous software artefacts within a given time frame.

The remainder of this chapter is structured as follows. First, Section 4.2 describes the data sources and preprocessing steps for this study. Section 4.3 defines the problem of summary-generation as an optimisation problem based on cosine similarity and 26 text-based metrics. Sections 4.4 and 4.5 include the results of our computational study and expert annotation of the results. Threats to validity are discussed in Section 4.6 and our conclusions are presented in Section 4.7.

4.2 Data Preparation

To better understand what human-written summaries of time-windowed software development artefacts look like, it was necessary to create our own gold-standard (described in Section 3.2). The basis of this gold-standard was a total of 503 summaries that were produced (mostly) on a weekly basis by 50 students over 14 weeks and for 14 (university-internal) GitHub projects[1]. The students were working in teams of three or four on their capstone projects, toward a Bachelor degree, with clients from local industry (43 students in total) or toward their Masters degree, with clients from academia (7 students in total). To ensure the usefulness of the student summaries, each student summary was assessed as part of their course assessments during the particular semester. The summaries were anonymised before conducting this work to ensure confidentiality and anonymity of the students.

We examined these summaries, as discussed in Section 3.2.2, to understand the general properties of human-written summaries, such as a summary's typical length and the amount of information we could expect each summary to contain. Additionally, the student summaries could provide us with an understanding of the common types of artefacts related to development activities and would help

[1]One of the student projects, which involved three students, was unavailable for data collection when our summarisation approach was built. Therefore, the number of gold-standard summaries was reduced by 42 summaries compared to the projected number of gold-standard summaries mentioned in Chapter 3.2.1.

Algorithm 1: Cleansing artefacts text(T) and the text of student summaries (S)

```
Input: $T$ - Artefacts text
Output: $C_t$ - Cleaned $T$
  1. Remove code blocks, tables, and images.
  2. Split text into paragraphs.
  3. Remove embedded URLs, file paths, file extensions, and non-ASCII
     characters if they formed a paragraph.
  4. Replace special characters (e.g., exclamation marks, brackets, single or
     double quotes) with single white spaces.
  5. If a paragraph contains only one sentence ending with a comma or
     semicolon, connect this paragraph to the next one.
  6. Split the paragraph into sentences using Apache OpenNLP sentence
     detector.
  7. Discard any sentences that contain only URLs, file paths, and
     punctuation. This is needed as the textual artefacts are not well-formed
     and inconsistent in style.
  8. Tokonise each sentences using OpenNLP tokenizer tool.
  9. Apply word stemming using Snowball steaming tool and stop word
     removal.
 10. return $C_t$

Input: $S$ - Text of student summaries
Output: $C_s$ - Cleaned $S$

  1. Split text into paragraphs.
  2. Remove embedded URLs, file paths, file extensions, and non-ASCII
     characters if they formed a paragraph.
  3. Replace special characters (e.g., exclamation marks, brackets, single or
     double quotes) with single white spaces.
  4. Split the paragraph into sentences using Apache OpenNLP sentence
     detector.
  5. Tokonise each sentence using the OpenNLP tokeniser tool.
  6. Apply word stemming using Snowball steaming tool and stop word
     removal.
  7. return $C_s$
```

TABLE 4.1. The extracted sentences from the student summaries per project over 14 weeks.

Project No.	**Number of Students**	**Number of Summaries in 14 Weeks**	**Number of Sentences**
1	3	35	100
2	3	3	38
3	4	38	42
4	4	40	101
5	4	37	104
6	4	30	80
7	3	32	141
8	3	11	36
9	4	46	165
10	4	44	141
11	3	38	150
12	4	46	142
13	3	27	105
14	4	51	264
Total	50	503	1,609

TABLE 4.2. Total number of artefacts per type and number of extracted sentences for each type.

Type	**Number of Artefacts**	**Number of Sentences**
Issue titles (IT)	1,885	1,885
Issue bodies (IB)	1,885	5,650
Issue body comments (IBC)	3,280	8,754
Pull requests titles (PRT)	1,103	1,103
Pull requests bodies (PRB)	1,103	5,176
Pull requests body comments (PRBC)	897	1,811
Pull requests reviews (PRRv)	2,019	2,762
Pull requests reviews' comments (PRRvC)	1,286	1,737
Commit messages (CM)	4,562	7,856
Commit comments (CMC)	30	55
Milestone titles (MT)	103	103
Milestone description (MD)	103	142
README files (RMe)	14	2,678
Wiki files (Wiki)	492	16,436
Releases (Rel)	1	4
Total	18,763	56,152

us identifying which sentences from which artefacts should be selected for our extractive summarisation approach.

To automatically collect the summaries, we used a Slack bot that asked the students to write summaries on a weekly basis to record their project development activity. The written summaries were automatically recorded and collected in response to the question: "If a team member had been away, what would they need to know about what happened this week in your project?". Figure 4.2 shows an example: the question and the student's summary are in the left column, and the relevant artefacts are in the right column. Note that the students only provided the summary; that is, they did not provide a list of the relevant artefacts.

We then used 15 types of textual artefacts in the GitHub repositories as input sources for our multi-document summarisation techniques. These artefacts are issues (titles, bodies, and comments), pull requests (titles, bodies, comments, reviews, and reviews' comments), commits (messages and comments), milestones (titles and descriptions), releases[2], wiki entries, and README files. As discussed in Section 3.3, the selection of these artefacts was based on how much information they provided about developers' development activities, as also found in our gold-standard summaries.

GitHub is generally characterised as an unstructured and informal means of communication, even though best practice guides might be in place at times. Consequently, the documents to be summarised as well as the summaries themselves needed to first undergo various preprocessing steps to eliminate noisy data, as these would otherwise negatively affect the eventual evaluation of our summarisation approaches. These preprocessing steps included sentence splitting, stop word removal, and stemming. Also, we removed source code blocks from the software artefacts due to lack of evidence that the student summaries cited code from actual files. Algorithm 1 describes our data cleaning process for texts of both the artefacts and the summaries. The number of sentences remaining in the artefact texts and the student summaries, after the cleansing process, are shown in Table 4.2 and Table 4.1, respectively.

4.3 Methodology

In our approach, we sought to extract text from a set of heterogeneous software artefacts so that the resulting summaries would be similar in style to those

[2]Release artefacts were added in this study because we found that these were used by students in their projects, and formed part of their own project development activities.

found in the gold-standard summaries.

In the next sections, we introduce two ways of measuring similarity (Section 4.3.1), we revisit the definition of cosine similarity in Section 4.3.2, and we define the iterative search heuristics in Section 4.3.3.

4.3.1 Summaries based on word similarity and feature vector similarity

Historically, the selection of important sentences for inclusion in a summary is based on various features represented in the sentences such as sentence position (Ouyang et al., 2010), sentence length (Kupiec, Pedersen, and Chen, 1995), sentence centrality (Erkan and Radev, 2004), and word frequency (Luhn, 1958). Determining these features in the selection of important sentences is not simple and depends largely on the type of documents to be summarised (Torres-Moreno, 2014).

We considered two ways of characterising sentences: (1) based on the similarity of words, and (2) based on the similarity of feature vectors. In both cases, the goal was to select sentences from the collection of software artefacts so that the characteristics of the resulting summary would be close to the characteristics of a target.

First, word similarity between two texts was defined by the number of times a term occurred in both texts, after text cleansing was performed (See algorithm 1). To achieve this, we used a vector-based representation, where each element denoted the number of times a particular word occurred in the sentence.

Second, as an alternative to word similarity and for situations where a reference text is unavailable, we considered 26 text-based features of sentences that can capture different aspects of readability metrics, information-theoretic entropy and other lexical features (see Table 4.3). Each sentence is represented as a 26-dimensional vector of the feature values. For an initial characterisation of this high-dimensional dataset, we refer the interested reader to (Alghamdi, Treude, and Wagner, 2019).

Table 4.3. Features used to represent each sentence.

No.	**Feature**
F1.	Word count
F2.	Chars count including spaces
F3.	Chars without spaces
F4.	No. of syllables in a word
F5.	Sentence length
F6.	Unique words
F7.	Avg. word length (chars)
F8.	Avg. sentence Length (words)
F9.	No. of monosyllabic words
F10.	No. of polysyllabic words
F11.	Syllables per word
F12.	Difficult words
F13.	No. of short words ($\leq$ 3 chars)
F14.	No. of long words (>= 7 chars)
F15.	Longest sentence (chars)
F16.	Longest words (chars)
F17.	Longest words by number of syllables
F18.	Estimated reading time
F19.	Estimated speaking time
F20.	Dale-Chall readability index
F21.	Automated readability index
F22.	Coleman-Liau index
F23.	Flesch reading ease score
F24.	Flesch-Kincaid grade level
F25.	Gunning fog index
F26.	Shannon entropy

4.3.2 Cosine similarity

The most popular similarity measure used in the field of text summarisation is cosine similarity (Manning, Raghavan, and Schütze, 2008) as it has advantageous properties for high-dimensional data (Sohangir and Wang, 2017).

To measure the cosine similarity between two sentences x and y – respectively their representation as a vector of word counts or the 26-dimensional representation – we first normalised the respective feature values (each independently) based on the observed minimum and maximum values, and then calculated the cosine similarity:

$$\cos(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x}\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|} = \frac{\sum_{i=1}^{n} \mathbf{x}_i \mathbf{y}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{x}_i)^2}\sqrt{\sum_{i=1}^{n} (\mathbf{y}_i)^2}} \tag{4.1}$$

We used cosine similarity in our optimisation algorithms as the fitness function to guide the search toward summaries that are close to the target vector.

4.3.3 Algorithmic approaches

Extractive multi-document summarisation can be seen as an optimisation problem, where the source documents form a collection of sentences, and the task is to select an optimal subset of the sentences under a length constraint (Peyrard and Eckle-Kohler, 2016). In this study, we aimed to generate summaries with up to five sentences, as this was approximately the length of the gold-standard summaries that the students had written.

We now present our optimisation algorithms to automatically produce summaries from heterogeneous artefacts within a given time frame. We utilised five algorithms, and we also created summaries at random to estimate a lower performance bound. We used cosine similarity as the scoring function in order to compute either the word similarity or the feature similarity with respect to a given target. In our case, the targets are the gold-standard summaries. By doing so, we aim at capturing the developers' activities described in the software artefacts that were created or updated within the given time frame and that were then cited in the gold-standard summaries in order to generate new human-like summaries.

Our first approach was to use a brute force algorithm (Algorithm 2). The algorithm keeps track of the best combination in a given set of artefacts' sentences to serve as an automatic summary. The summaries in each of the generated combinations were limited a maximum of five sentences. We used this as a performance reference, because we did not know a-priori what good cosine similarity values might be. The algorithm starts with an empty set (GS) that makes up the potential automatic summary generated from a set of artefacts' sentences (AS). The algorithm then creates all the possible combinations (CB) of five sentences from (AS). In steps 4 to 7, the algorithm iterates over all the combinations to select the best combination that maximises the cosine similarity between each combination (CB_i) and the student summary (SS). Finally, the algorithm returns this combination (GS) as an automatically produced summary.

Algorithm 2: Brute force algorithm

```
Input: $AS$ - artefacts' sentences, $SS$ - student summary, and $TLGS$ -
       targeted length of the generated summary.
Output: $GS$ – generated summary
1: $GS \leftarrow \emptyset$
2: $CB \leftarrow createAllCombinations(AS, TLGS)$
3: for all ($CB_i \in CB$) do
4:    if $cosSimilarity(CB_i, SS) \geq cosSimilarity(GS, SS)$ then
5:       $GS \leftarrow CB_i$
6:    end if
7: end for
8: return $GS$
```

The second algorithm is a Greedy approach (see Algorithm 3). The algorithm iteratively builds up a summary sentence-by-sentence so that the cosine similarity between the potential automatic generated summary and the student summary is optimal. In step 1, the algorithm creates up to five sentences to form the summary. In each iteration, the algorithm compares each of the artefacts' sentences (AS) to the student summary (SS) to determine the best single sentence whose cosine similarity to the (SS) is better than other sentences in (AS). Then it adds this sentence to the potential produced summary (GS) as shown in steps 5-9. The algorithm in steps 10-12 checks whether adding the best single sentence in each iteration (K_{best}) to the (GS) would increase the cosine similarity value compared to the sentence added to (GS) in the previous iteration. If so, the (K_{best}) would be removed from the (AS), and a new iteration will start with an addition of a new sentence to (GS). Finally, the algorithm stops if the (GS) reaches five sentences or the new additional sentence in an iteration would result in a worsening of the cosine similarity between (GS) and (SS).

Algorithm 3: Greedy algorithm

```
Input: $AS$ - artefacts' sentences, $SS$ - student summary, and $TLGS$ -
       targeted length of the generated summary.
Output: $GS$ – generated summary
1:  $GS \leftarrow \emptyset$
2:  while ($len(GS) \leq TLGS$) do
3:    $K \leftarrow \emptyset$ {$K$: unused sentences in $AS$}
4:    $K_{best} \leftarrow \emptyset$ {best single sentence to add in this iteration}
5:    for all ($K_i \in K$) do
6:      if $cosSimilarity(GS + K_i, SS) \geq cosSimilarity(GS + K_{best}, SS)$ then
7:        $K_{best} \leftarrow K_i$
8:      end if
9:    end for
10:   if $cosSimilarity(GS + K_{best}, SS) < cosSimilarity(GS, SS)$ then
11:     return $GS$ {do not add $K_{best}$ if it worsens the similarity}
12:   end if
13: end while
14: return $GS$
```

Algorithm 4: Random Local Search with unrestricted summary length (RLS-unrestricted)

```
Input: $AS$ - artefacts' sentences and $SS$ - student summary
Output: $GS$ – generated summary
1:  $GS \leftarrow \emptyset$
2:  while (running time < 10 seconds) do
3:    select a sentence $AS_r$ from $AS$ uniformly at random
4:    $GS_{temp} \leftarrow GS$
5:    if $AS_{pos} \notin GS$ then
6:      $GS_{temp} \leftarrow GS_{temp} + AS_r$
7:    else
8:      $GS_{temp} \leftarrow GS_{temp} - AS_r$
9:    end if
10:   if $cosSimilarity(GS_{temp}, SS) \geq cosSimilarity(GS, SS)$ then
11:     $GS \leftarrow GS_{temp}$
12:   end if
13: end while
14: return $GS$
```

In addition to brute force and Greedy algorithms, we used three variations of random local search (RLS) algorithms. First, RLS-unrestricted (see Algorithm 4) can create summaries without being restricted by a target length. Within 10 seconds, the algorithm starts by randomly selecting a new sentence from the collection of artefacts' sentences (AS) to form the potential summary (GS). Then, in steps 4-9, the algorithm checks the inclusion status of the randomly selected sentence. That is, if the randomly selected sentence is already included in the summary (GS), this sentence will be removed, otherwise added to (GS). Steps 10-11 ensure that the included/removed sentence would increase the cosine similarity value between the (GS) and the student summary (SS). Finally, the algorithm returns the best collection of sentences created within the specified time from artefacts to serve as a summary.

Algorithm 5: Random Local Search with restricted summary length (RLS-restricted)

```
Input: $AS$ - artefacts' sentences, $SS$ - student summary, and $TLGS$ -
       targeted length of the generated summary.
Output: $GS$ – generated summary
 1: $GS \leftarrow \emptyset$
 2: while (running time < 10 seconds) do
 3:   select a sentence $AS_r$ from $AS$ uniformly at random
 4:   $GS_{temp} \leftarrow GS$
 5:   if $AS_{pos} \notin GS$ and $GS_{temp} < TLGS$ then
 6:     $GS_{temp} \leftarrow GS_{temp} + AS_r$
 7:   else
 8:     $GS_{temp} \leftarrow GS_{temp} - AS_r$
 9:   end if
10:   if $cosSimilarity(GS_{temp}, SS) \geq cosSimilarity(GS, SS)$ then
11:     $GS \leftarrow GS_{temp}$
12:   end if
13: end while
14: return $GS$
```

Second, RLS-restricted algorithm (5) is like RLS-unrestricted, but it can only generate summaries of at most a given target length.

Third, RLS-unrestricted-subset algorithm (6) runs RLS-unrestricted first, but it then runs the brute force approach to find the best summary of at most a given target length.

These algorithms share common characteristics, such as the execution time limit and the ability to explore the search space by including or excluding sentences. One notable characteristic of RLS-unrestricted is that it can produce

Algorithm 6: RLS-unrestricted with subset selection (RLS-unrestricted subset)

```
Input: $AS$ - artefacts' sentences, $SS$ - student summary, and $TLGS$ -
       targeted length of the generated summary.
Output: $GS$ – generated summary
 1: $GS \leftarrow \emptyset$
 2: $GSU \leftarrow RLS-unrestricted(AS, SS, TLGS)$
 3: $GS \leftarrow BruteForce(GSU, SS, len(GSU))$
 4: return $GS$
```

summaries that exceed the target length. We investigated this to show whether five sentences were enough to create close summaries.

As the sixth approach, we used a random search (Algorithm 7) as a naive approach to provide a lower performance bound. This approach iteratively creates, within 10 seconds, a set of potential summaries (GS_{temp}) of five unique sentences (K) selected from the collection of artefacts' sentences (AS), as shown in steps 2-5. The algorithm then iterates over each of the potential summaries in (GS_{temp}) to calculate the cosine similarity. Finally, the algorithm returns the best randomly created five-sentence summary (GS).

Note that the student summary (SS), which is used as an input in all approaches could either be an actual summary (i.e., in words) in which case the co-occurrence was calculated, or it could be a summary in the form of a feature vector in the high-dimensional feature space.

Algorithm 7: Baseline summary based on Random selection algorithm

```
Input: $AS$ - artefacts' sentences and $SS$ - student summary.
Output: $GS$ – generated summary
 1: $GS \leftarrow \emptyset$
 2: while (running time < 10 seconds) do
 3:    randomly select subset of five unique sentences $K$ from $AS$ uniformly at
       random
 4:    $GS_{temp} \leftarrow K$
 5: end while
 6: $GS_{best} \leftarrow \emptyset$
 7: for ($\forall GS_i \in GS_{temp}$) do
 8:    $GSC \leftarrow$ cosSimilarity($GS_i$,$SS$)
 9: end for
10: sort GSC in descending order and return the best randomly created
    summary
11: $GS_{best} \leftarrow GSC_0$
12: $GS \leftarrow GSC_{best}$
13: return $GS$
```

Lastly, to investigate the impact of the individual artefacts on the summaries, we considered three scenarios as input sources to generate summaries, with each of the algorithms, in a given time window. These scenarios were as follows:

1. Single artefacts – In this approach, each of the 15 artefacts listed in Table 4.2 is considered as individual source, and the goal of our summariser algorithms is to generate summaries based on each of these sources separately, given the time window and the project.
2. All artefacts – Here, all sentences from a project during the relevant time window are considered, and there is no limitation on the type of artefact.
3. Most relevant artefacts – Assuming we know a developer's preferences for particular types of artefacts, we only considered sentences from those types.

Implementation note. We removed a-posteriori all instances where we encountered at least one empty summary for two reasons: (1) word similarity between a generated summary and the student's summary can be zero, and (2) we encountered co-linear vectors even in the 26-dimensional space. Generating summaries from all artefacts as an input source, we detected 670 and 1065 empty summaries generated from all algorithms using word similarity and feature similarity, respectively. On the other hand, we found 845 and 980 empty summaries generated by all algorithms using word similarity and feature similarity, respectively, when the most relevant artefacts were considered as an input source.

4.4 Experimental Results and Discussion

In our experiments, we considered the 503 summaries written by students, 6 algorithms, and three scenarios (i.e., the sentences' sources).

For both similarity measures, we used the gold-standard as the target, i.e., the students' original summaries, either as bag of words or as high-dimensional feature vectors. An alternative to feature similarity was to use the average vector across all students, to aim at an "average style"; however, then it would no longer be clear if it could be approximated. As this is the first such study, and in order to study the problem and the behaviour of the algorithms in this extractive setting under laboratory conditions, we aimed for the solutions defined in the gold-standard.

A comparison with brute force.

To better understand what quality we could expect from our five randomised approaches, we compared these approaches with the brute force approach to extractive summarisation. The artefact type for this first investigation is “issue title“. The maximum number of sentences here per project and summary combination was 35. For our brute force approach, this resulted in a manageable number of $324,632 + 52,360 + 6,545 + 595 + 35 = 384,167$ subsets of up to five sentences for that particular week. The computational budget that we give each RLS variant was 10 seconds.

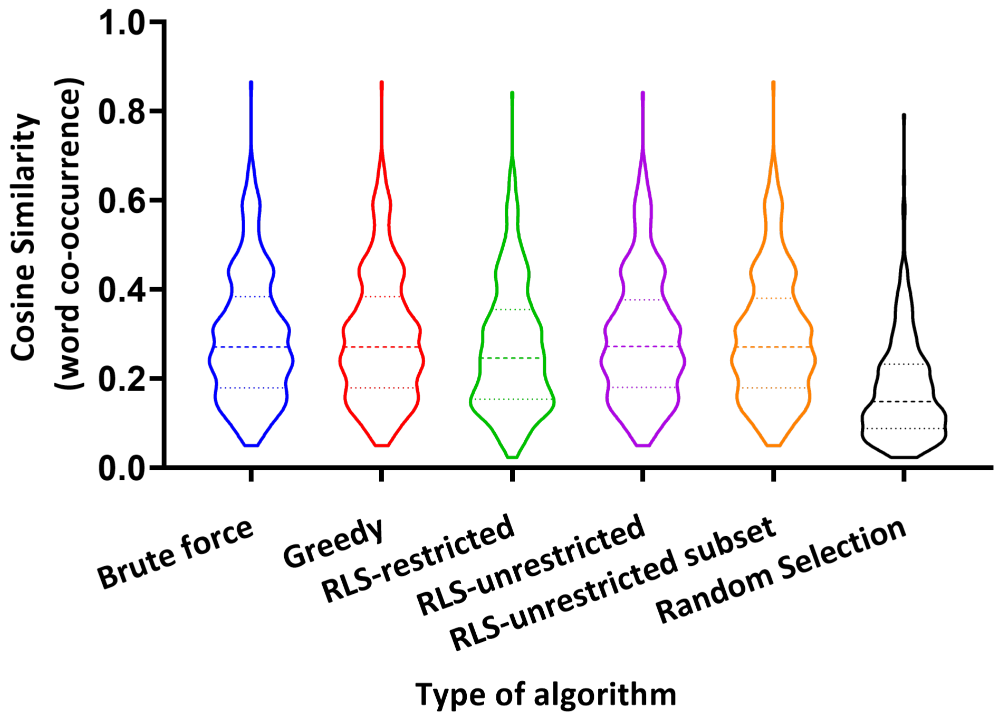


FIGURE 4.3. Cosine similarity based on word co-occurrence of the generated summaries.

Comparing the results obtained by these algorithms (see Figure 4.3), we observed that the Greedy algorithm was able to generate summaries whose overall distribution was close to the distribution of summaries generated by brute force[3]. Similarly, the two RLS-unrestricted approaches also produced comparable summaries. The RLS-restricted performed worse, but still better than the Random selection approach.[4]. From this first comparison, we concluded that

[3]Based on a two-sided Mann-Whitney U test, there is no statistically significant difference at p=0.05 between Greedy and brute force.

[4]Note that random selection is not limited to producing only one summary, but returns many until the time limit is reached, and it then returns the best.

Greedy is a very good approach, as it achieved a performance comparable to that of brute force (which was our upper performance bound), and it required only 0.49 seconds on average to form a summary compared to other algorithms (see Figure 4.4). We can moreover conclude that a maximum summary length of five sentences is acceptable, as the *RLS-unrestricted subset* did not perform differently from the others, which were restricted.

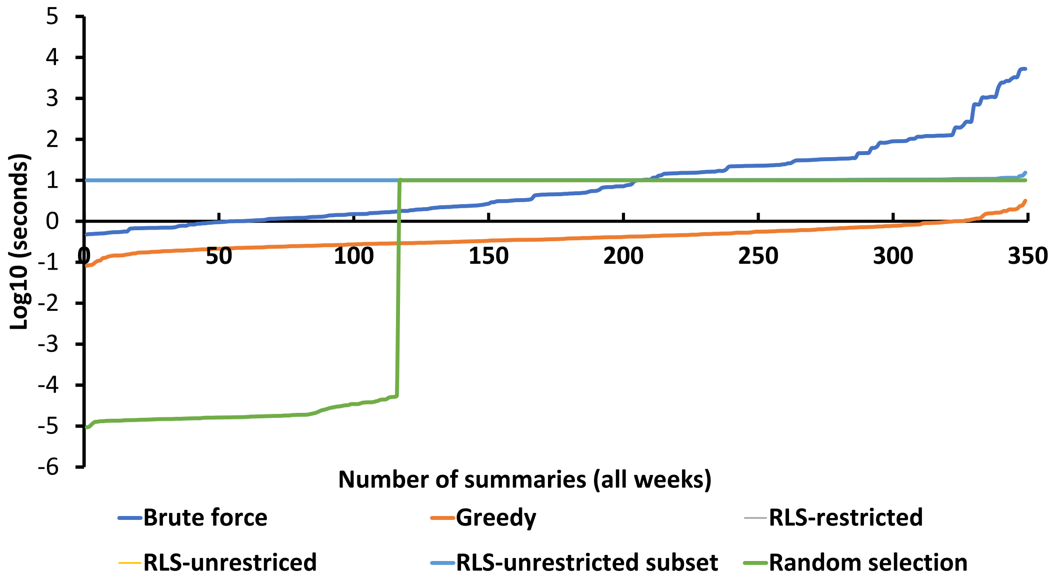


FIGURE 4.4. Running time of each of the algorithms required to generate summaries – log_{10} was used for better visualising the results. The average running time per algorithm (in seconds) to generate a summary is, brute force: 151.92, Greedy: 0.49, RLS_restricted: 10.0, RLS_unrestricted: 10.0, RLS_unrestricted subset: 10.20, Random selection: 6.67 seconds.

To explain Greedy's performance, and that the performance of Greedy and of some of the RLS variants was very comparable, we conjectured that the problem of maximising the cosine similarity w.r.t. a target vector given a set of vectors is largely equivalent to a submodular pseudo-Boolean function without many local optima. A formal proof of this, however, remains future work.

Ability to generate non-empty summaries.

As mentioned above, the heuristics can produce empty summaries. Figure 4.5 shows the success rate of generating non-empty summaries using the various approaches (except brute force). As one might expect, the algorithms are more often successful when sentences can be taken from all artefacts. Also, when the high-dimensional feature similarity is used, more runs are successful, as feature similarity does not encounter the issue of having no word overlap between sentences and the target summary. Both similarity measures have, however, the challenge of being unable to deal with co-linear vectors, which we did encounter even in the 26-dimensional space, and which explains why the success rate is not 100%.

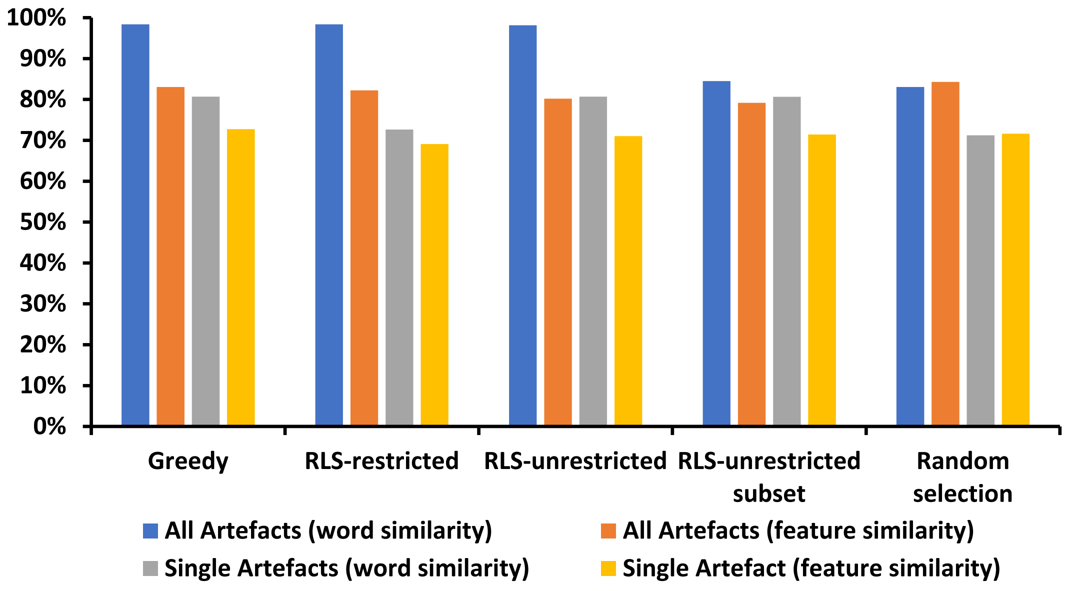


FIGURE 4.5. Success rate of generating non-empty summaries.

Used types of artefacts.

Next, we used each algorithm in turn to create a weekly summary for all cases where we had student summaries. In particular, we investigated from which artefact types the sentences were taken in these generated summaries. In total, there were 22,313 (39.73% of the total) sentences found in the source input linked to the student summaries. Note that while this number appears to be very large, it includes the very large summaries produced by RLS-unrestricted (average length 29.6), and that we were aiming at hundreds of different target summaries for one-week time-windows, which thus appear to require very different sentences from the artefacts.

Figure 4.6 shows that the summaries generated by each of the algorithms are composed of sentences from almost all of the artefact types. In particular, we can note that sentences from wiki pages are most prevalent. Possible reasons for this include that (1) wiki pages make up the largest fraction of the source sentences, and (2) developers might have described their activities best on the wiki pages.

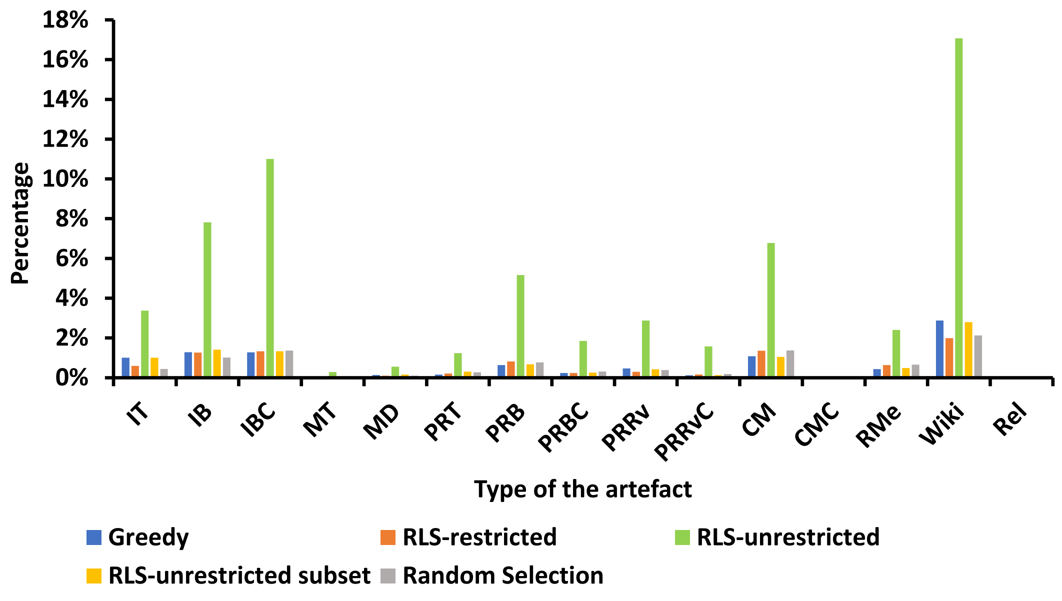


FIGURE 4.6. Average contribution of artefacts to summaries, aggregated across the two similarity measures.

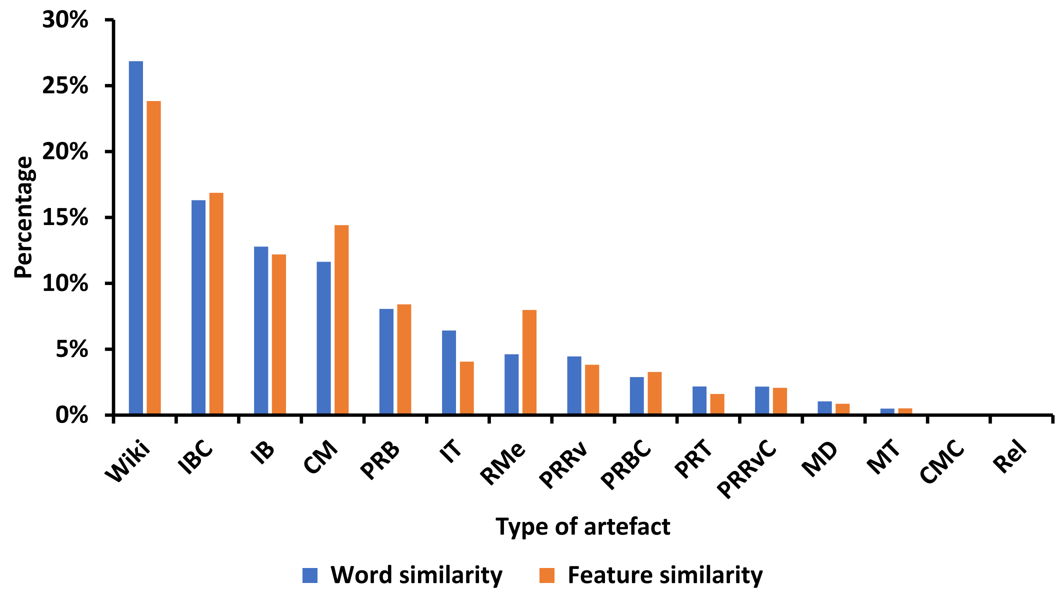


FIGURE 4.7. Average contribution of artefacts to summaries, aggregated across all algorithms.

Figure 4.7 illustrates that content from wiki artefacts contributed around 27% to the summaries generated by all algorithms. Also, sentences found in issue bodies (IB), issue body comments (IBC), and commit messages (CM) contributed between 13% and 17%. On the other hand, artefacts such as pull request reviews (PRRv), pull request titles (PRT) and milestone titles (MT) had the lowest contributions, which indicates that the students did not commonly use these artefacts, or at least did not mention their content, during their project's development life-cycle.

Generating summaries based on the most relevant artefacts.

By generating summaries based on the most relevant artefacts found in the students' original summaries, we aimed to generate more human-like summaries that better reflect the developers' preferences for certain artefact types.

To achieve this, we considered the generated summaries as a starting point, as each of them was generated to be similar to a particular student summary, and hence it could indirectly reflect a student's preferences. Then, we identified the most relevant ones by using the median as the cut-off (i.e., based on Figure 4.7). This selection process revealed that the eight most commonly referred to artefacts were (from most common to least common): wiki, issue title, issue bodies, issue body comments, commit messages, pull request bodies, README files, and pull requests reviews. In total, this reduced the number of candidate sentences by 10.5%, to 50,246.

We next investigated the performance of the subset of artefacts in terms of its ability to generate good summaries. Figure 4.8 shows the cosine word co-occurrence similarity and feature similarity achieved by each of the algorithms. Blue violin plots show the distributions of similarities achieved when all 15 artefacts were considered, and the red violin plots show the same for the eight most relevant artefacts. As we can see, focusing on only eight artefact types appeared to have little to no negative impact.

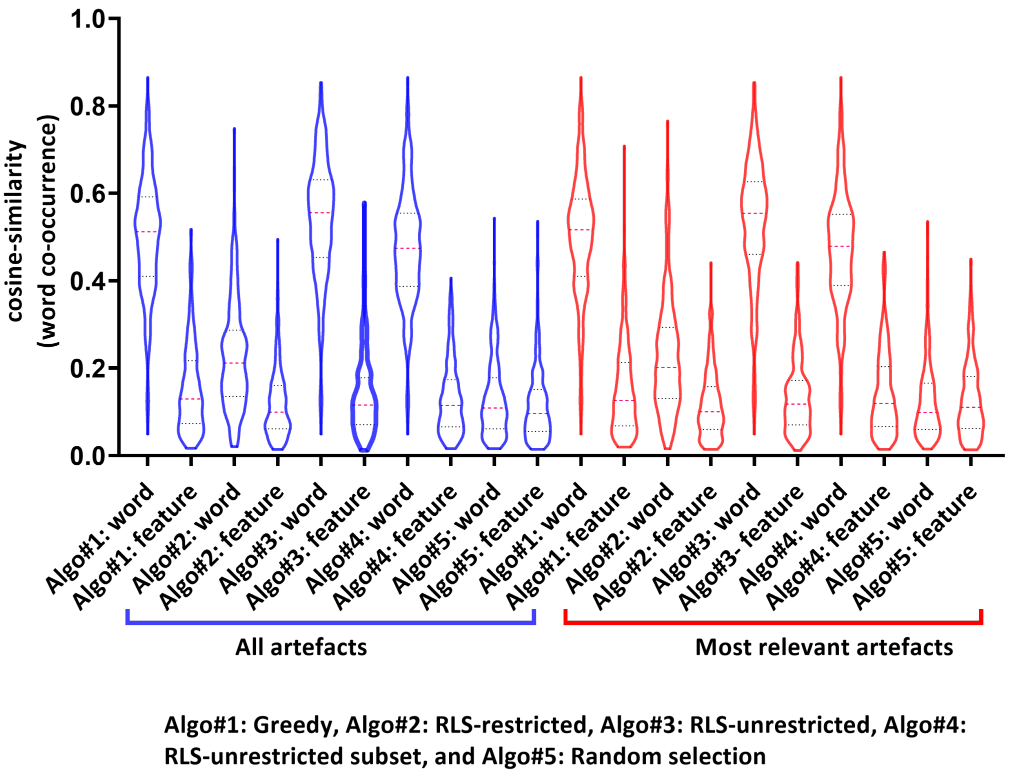


Figure 4.8. Similarities: when all artefacts are used (blue, overall average 0.266) and when only the relevant eight are used (red, overall average 0.258).

4.5 Expert Annotation

To evaluate the extent to which the summaries that the different approaches generate matched the summaries written by the students *in the perception of software developers*, we asked two expert annotators to evaluate the results. Both were in their first year of study toward a Computer Science PhD, and neither was affiliated with this study. Both annotators indicated that developing software is part of their job, and they claimed four to six years of software development experience. Annotator 1 stated that they had one to two years of experience using GitHub for project development; Annotator 2 claimed two to four years of experience with GitHub.

The selection of algorithms to be used for expert annotation was based on the highest median value of the cosine similarities between the gold-standard summaries and the summaries generated from each of the algorithms. Therefore, summaries generated by the Greedy algorithm were chosen for annotation.

For the study, we randomly selected ten out of the total of fourteen weeks, and for each week, we randomly selected one project. For each of these ten, we then produced six different summaries in relation to the gold standard (i.e., the summaries written by the students):

1. The best summary based on *word similarity* between sentences contained in all artefacts in the input data (issues, pull requests, etc.) and the gold standard student summary,
2. Same as (1), but only using the eight most relevant artefacts as input data,
3. The best summary based on *feature similarity* between sentences contained in all artefacts in the input data and the gold standard student summary,
4. Same as (3), but only using the eight most relevant artefacts as input data,
5. A random baseline by randomly selecting five sentences from all artefacts,
6. Same as (5), but only using the eight most relevant artefacts as input data.

We created a questionnaire that asked the annotators first to produce a summary for the ten selected weeks after inspecting the corresponding GitHub repositories (to ensure that annotators were familiar with the projects), and then to rate each generated summary on a Likert-scale from 1 (strongly disagree) to 5 (strongly agree) in response to the question “Please indicate your agreement with the following statement: The summary mentions all important project activities present in the gold-standard summary“.

TABLE 4.4. Average rating from each annotator for output produced by the different approaches.

Approach	**Annotator 1**	**Annotator 2**
Word (all)	3.7	3.3
Word (subset)	**3.8**	**3.5**
Feature (all)	3.7	2.0
Feature (subset)	3.7	2.0
Random (all)	3.5	1.8
Random (subset)	3.0	1.8

Table 4.4 shows the results, separately per annotator. While it is apparent from the data that Annotator 1 generally gave out higher scores than Annotator 2, both annotators perfectly agreed on the (partial) order of the different approaches: Word (subset) $\geq$ Word (all) $\geq$ Feature (subset) $\geq$ Feature (all) $\geq$ Random (all) $\geq$ Random (subset). In Table 4.5, we show an example of a student summary and the summaries generated by the Greedy approach using different scenarios as input sources and for a particular week. We can observe that 1) using all artefacts or a subset produced the same summary, 2) less information related to developers’ activities in the student’s summary was captured in the summary that was randomly produced compared to summaries generated by the Greedy algorithm.

Figure 4.9 shows box plots for each approach, also indicating that the approaches based on text similarity achieve the best result in terms of human perception, followed by approaches based on feature similarity, and then random baselines.

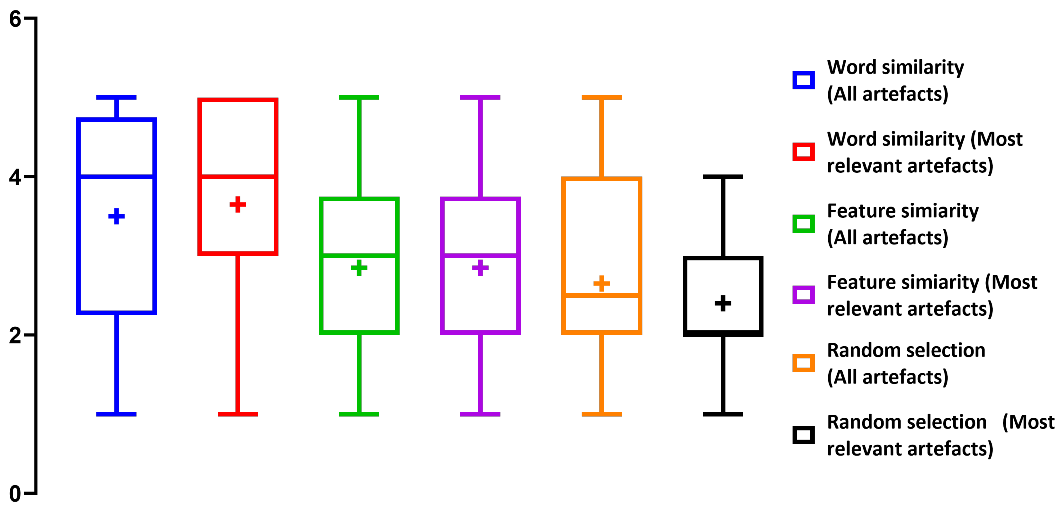


FIGURE 4.9. Ratings by our annotators of different approaches.

TABLE 4.5. Comparing summary written by human with generated summaries.

Student Summary
Change of direction for the project as a result of Monday's client meeting discussion and survey feedback, wiki integration is now priority so we've needed to look into the Confluence REST API to fetch wiki pages and spaCy for Natural Language Processing. We met with <redacted> again and discussed the change of project direction and details on the wiki pages with <redacted>. Much work has been done this week in writing the wiki parser and the search-by-category feature has been implemented

Approach	**Generated summaries**
Word (All)	Progress: Have extracted paragraphs and headers from a <redacted> wiki page, and this is the result. TLDR; Project focused has changed massively, we are now going to focus almost exclusively on improving the Confluence search. When running the parser it extracted this content (from the Hello World wiki page available on your drives). The results of the survey indicate that most developers are unsatisfied with the wiki searching, but are mostly satisfied with their own code search tools (IDE, grep). Week-7:-Team-Meeting-Weekly-Log.

Feature (All)	Our very first transaction is going to just display the text "Hello World" in a dialog window of the Eureka application, but lets assume we are wanting to calculate the centroid of a point selection and outputting the result to the report window. As the call for a page and a pages content are seperate, the wiki structure can be traversed and only new content requested if it has been modified (last modified available in the response header). Request home page from some wiki.atlassian.com address (confluence).
Word (subset)	Progress: Have extracted paragraphs and headers from a <redacted> wiki page, and this is the result. TLDR; Project focused has changed massively, we are now going to focus almost exclusively on improving the Confluence search. When running the parser it extracted this content (from the Hello World wiki page available on your drives). The results of the survey indicate that most developers are unsatisfied with the wiki searching, but are mostly satisfied with their own code search tools (IDE, grep). Week-7:-Team-Meeting-Weekly-Log.
Feature (subset)	Our very first transaction is going to just display the text "Hello World" in a dialog window of the Eureka application, but lets assume we are wanting to calculate the centroid of a point selection and outputting the result to the report window. As the call for a page and a pages content are seperate, the wiki structure can be traversed and only new content requested if it has been modified (last modified available in the response header). Request home page from some wiki.atlassian.com address (confluence).
Random (All)	The prefix for the sdp library is SDP. To run the Eureka application in the workbench: The workbench will be opened. We use a text object called serC_Text to put any user facing strings. Fix search overlap and result count problem. Thursday.

Random (subset)	To create a dialog window we will need to add a transaction to the Eureka application. This patch adds selectable toggles to the search template page that will enable the user to do just that. So for this exercise we will use the sdpServer. The second argument is the struct in which the data from the user is stored, and the third is the class which is run in the place of the struct. <redacted> has made significant progress on API scraper.

4.6 Threats to Validity

Our study, like many other studies, has a number of threats that may affect the validity of our results.

First, our research subjects involved summaries written by graduate and undergraduate students with good knowledge about interacting with the GitHub platform. Although, using professional developers to create the gold-standard summaries, the results may have been different. However, the students summaries' quality should not be considered a limitation because each student summary was evaluated as part of their course assessments during the particular semester. The evaluation of the students summaries took into account the students' development activities that happened for a particular week.

Our results, illustrated in Table 4.4, show that the eight most relevant artefacts were found to be sufficient to generate summaries containing developers' activities. These types – such as issues, pull requests, and commits – are all essential elements of a GitHub repository. However, as these are essential elements of probably any software repository, we expect this finding to be transferable to other repositories.

Also, evaluating the automatically generated summaries relies on human judgement. Subjectivity and bias are likely to be issues when the number of human experts involved is small, as in the present study. Hence, we plan, for future work, to include more experts to mitigate these issues.

4.7 Conclusion

Software engineering projects produce many artefacts over time, ranging from wiki pages, to pull request and issue comments. Summarising these can be helpful to a developer, for example, when they return from a holiday, or when they need to gain an overview of a project's background in order to move forward with their team.

In this chapter, we have presented the first framework to automatically summarise heterogeneous artefacts produced during a given time window. We have defined our own gold standard and devised ways of measuring similarity at a text-based level. Then we compared various optimisation heuristics using different input scenarios, and found that a Greedy algorithm can generate summaries that are close to the human-written summaries in less running time than other algorithms. Our study then found that experts preferred the combination that used word similarity to generate summaries based on the eight most relevant artefacts. Interestingly, the generated summaries have been found useful, even though the optimisation approaches have not yet considered temporal connections between the sentences and also not yet the actual meaning; this will involve future work to improve the quality of the generated summaries.

As mentioned previously in this chapter, the student summaries lack evidence of citing the source codes from actual files. Consequently, none of the generated summaries contained information about the source code. In the next chapter, we report our investigation into the potential for including textual artefacts (i.e., code comments) in the source codes for later use in summarisation.

The next chapter was published (Alghamdi et al., 2021a) and presented in the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR21). The conference ranked (A) based on the Australian CORE ranking system. This publication resulted from collaboration with Professor Takashi Kobayashi and Associate Professor Shinpei Hayashi, Tokyo Institute of Technology, Japan, during my research trip in Japan. I contributed to the design, implementation, and evaluation of the proposed approach. Prof. Takashi and A/Prof Shinpei contributed to the approach evaluation.

Chapter 5

The Potential of Code Comments for use in Summarisation

5.1 Introduction and Motivation

In Chapter 4 we explored different techniques and scenarios to generate automatic summaries of developers' development activities relating to different software artefacts. In addition, we saw that neither the generated summaries nor the gold-standard summaries referenced any information from the source code files. However, source code files can also contain lots of natural language text, such as code comments, which could be included in summaries.

Using source code comments has several advantages. For example, code comments can provide additional information to help developers perform a wide range of software engineering tasks. For instance, code comments can be used for bug detection (Rubio-González and Liblit, 2010; Silva and Ribeiro, 2003; Subramanian, Inozemtseva, and Holmes, 2014), specification inference (Blasi et al., 2018; Pandita et al., 2012) and testing (Goffi et al., 2016; Wong et al., 2015).

This chapter reports on our investigation into the role of primitive variable identifiers in comments, in particular to see how commonly these identifiers are documented in accompanying comments and what type of additional information the comments might contain about these variables.

To lay a foundation for understanding the role of primitive variables and their documentation in Java source code, our preliminary findings showed that the occurrence of primitive data types (in 2,491 Java repositories from GitHub) was dominant, with 60.10% (8,646,435), compared to non-primitive data types at 39.90% (5,741,380).

We also noted that while non-primitive types tended to have their intention encoded in the name of the type, primitive types did not allow for the encoding of information such as the variable's purpose, concepts and directives.

The introduction of Generics in Java has made many more type names self-explanatory (e.g., List<File> instead of List), while similar advantages are not available for primitive variables; hence a developer often has to guess what a String or float might contain. This motivated us to study the available knowledge about primitive variables in Java code comments. For instance, Listing 5.1 shows a comment that is not informative with respect to the local variable “ry“. It would therefore take developers quite some effort to understand the purpose, concepts or directives related to this identifier in the source code. There were two reasons for this: 1) the developer did not explicitly mention the identifier in the comment nor followed the naming conventions to encode such types of knowledge in the variable’s identifier, and 2) the lack of additional information about it in the comment. As a consequence, this can make program comprehension more difficult and ultimately impact the developer’s productivity. Cases like the example shown in Listing 5.1 are what we are targeting in our work, in an effort to identify knowledge types related to the variables’ identifiers in their accompanying comments.

```java
// CURVE-INSIDE
final float ry = t * (t * Ay + By);
```

LISTING 5.1. Comment with respect to local variable “ry“ (JogAmp Community, 2021).

Here we introduce the first empirical study to detect primitive variables’ identifiers in comments using two levels of matching techniques, to characterise the knowledge they contain. We contributed with the following:

1. We developed lexical and advanced matching techniques to capture the identifiers of primitive variables in Java source code comments, and then evaluated these approaches using a manually curated benchmark of six well-commented project repositories (Alghamdi et al., 2021b) hosted on GitHub.

2. We manually classified the documented information, used to describe the variable identifiers in comments, into three types of knowledge: purpose, concept and directives.

3. A large-scale analysis of 2,491 engineered Java software repositories (Alghamdi et al., 2021b) hosted on GitHub was carried out to provide an insight into how developers document these variables in the form of source code comments.

The remainder of this chapter is structured as follows. We introduce our research questions in Section 5.2. Our detection techniques are described in detail in Section 5.3. In Section 5.4, we describe our study design as well as the methods used for data collection and analysis. Our findings are reported separately for each of our research questions in Section 5.5. We then discuss threats to validity in Section 5.6 and related work in Section 5.7. Finally, we conclude the chapter and outline future work in Section 5.8.

5.2 Research Questions

Our ultimate goal in this study was to reveal the nature of the documented information for variables in source code comments. We define documented variables as those variables that have comments above their declaration, and where the developers mention the identifiers in their accompanying comments. To analyse the documented variables' information, we had to capture code comments that include the variables' information, using lexical matching. However, lexical matching (i.e., exact matching) may not be considered the best choice of method to detect an identifier in a comment (see Listing 5.2) because the variable's identifier may not explicitly appear in the comments: for example, developers might use abbreviations for the identifier or explain it with different terms. To address this problem, we developed a detection method considering lexical and advanced matching, which has the ability to capture the exact identifier and its meaning in a comment.

```java
/**
Writes a message into the database table.
Throws an exception, if an database-error occurs !
*/
public void append(String _msg) throws Exception {
```

Listing 5.2. Example to show the inability of lexical matching to detect variables' identifiers in a comment (The Apache Software Foundation, 2021a).

To the best of our knowledge, there is no guideline to detect a variable's identifier in a comment, although researchers have tried different techniques in the past (Chen et al., 2019). Therefore, we proposed and evaluated two detection techniques for detecting the variables' identifiers in comments, with a manually annotated benchmark of six well-commented project repositories hosted on GitHub, and we then investigated the nature of the documentation of the variables. The creation of the manual evaluation data set was necessary to evaluate our approaches, as a reference data set does not currently exist in the literature. To assess the quality of our detection approaches, we asked our first research question.

- **RQ1**: To what extent can different techniques detect variables in comments?
  - **RQ1.1**: To what extent is the lexical matching technique able to detect the variable names in the comments?
  - **RQ1.2**: To what extent are different advanced matching techniques able to detect the meaning of variable names in the comments?

Answers to RQ1 would confirm the quality of our detection approaches to be used for large-scale analyses, in order to then answer RQs 3 to 5.

As the identifiers of primitive variables require developers to provide additional information to explain the functionality of the variables in the comments, we need to then ask the following question:

- **RQ2**: What types of knowledge do comments provide about the variables?

The method used to answer RQ2 was inspired by previous work (Maalej and Robillard, 2013). As far as we know, the types of knowledge relevant to variables have not yet been reported in the literature. To remedy this lack, we identify three knowledge types that are closely related to the variable's domain significance and that, based on previous work, can meet our purpose to answer this question.

Since there is little known about the prevalence of primitive variables and their documentation, as used by developers, we sought to investigate how these variables are used and documented in source code comments by asking the following questions.

- **RQ3**: How frequently are primitive variables documented in comments?
- **RQ4**: What are the distributions of documented variables by their scope of declarations?
- **RQ5**: What are the types of comments associated with the scopes of the documented variables?

5.3 Detecting Documented Variables

In this section, we present our techniques to detect variables' identifiers in comments. To this end, several detection approaches were developed, using lexical matching, advanced matching and the union of lexical and advanced matching.

5.3.1 Preprocessing

Preprocessing variables' identifiers and comments is an essential step for our detection approaches. For this purpose, we partly followed the steps proposed by Ratol and Robillard (Ratol and Robillard, 2017) for preprocessing both variables' identifiers and the text of comments, while we also created our own preprocessing steps to suit our particular matching techniques. We present these steps as following:

Identifiers. We split an identifier based on its typographical conventions as either a single term or multiple terms (i.e., compound terms), using the camelCase rules and other rules we designed based on common conventions (for example, splitting the identifier using an underscore). We then converted each term to its root using the Lemmatiser of the Stanford Core NLP library (Manning et al., 2014) while retaining the original term for later use. As a result, each term then had a dictionary entry $\langle \text{term}, \text{lemma} \rangle$.

Comments. We processed the comments by first splitting them into sentences, using the sentence detector model in the OpenNLP library. Each sentence was then further split into tokens using the same library. Tokens that are detected to be compound terms were further split into one or more terms in the same way that the identifiers were split. After tokenisation, we removed stop words such as "the" and "or" from the list of tokens obtained, unless these stop words were part of the variables' identifier. Finally, we generated a dictionary entry $\langle \text{token}, \text{lemma} \rangle$ for each token in the list.

Implementation note. Lemmatisation is only applied to the variables' identifiers and comments' tokens, to comply with lemma and metaphonic advanced matching techniques.

5.3.2 Lexical matching

Our approach to lexical matching was to search for exact (i.e., literal) matching of variables' identifiers in the head comments. For example, in Listing 5.3, lexical matching can detect the variable identifier *ignore* in the accompanying comment. Lexical matching may present some issues, such as spelling or typographical errors, that may degrade its detecting performance, but it can be useful to show how often developers have documented the variable's identifier in comments with its exact form.

```
//if ignore is true, this column will be ignored by building
    sql-statements.
boolean ignore = false;
```

LISTING 5.3. Example of matching using lexical technique (The Apache Software Foundation, 2021b).

5.3.3 Advanced matching

To tackle the lexical matching problem indicated in the previous section (Section 5.3.2), we introduced a new method of detection; i.e., capturing the meaning of the variable's identifier through the wording in the comment. This advanced matching technique involves lemmas matching, metaphonic matching and SEthesaurus matching.

A) Lemmas matching

Lemmatisation is the process of grouping together the different inflected forms of a word into a single term. Our lemmas matching approach works by lemmatising the variable's identifier and the words in the text of the comment. It then looks for the presence of each identical lemma term between the identifier and the token in the text of a comment. If all lemmas of the identifier are found, it can be concluded that the variable was documented in the comments. For instance, in Listing 5.4, the comment contains two lemma terms, matching the two lemma terms of the identifier.

```
// Make sure at least one connection for this protocol succeeds (if
    expected to)
boolean connectionSucceeded = false;
```

LISTING 5.4. Example of advanced matching using lemma technique (Huß, 2021).

B) Metaphonic matching

Our second advanced detection approach is based on the Double Metaphone encoding algorithm (Philips, 2000). This algorithm is a search technique that can overcome matching problems encountered due to misspelling of given keywords. It works by taking two input strings and returns true if they phonetically match; i.e., the algorithm takes into consideration similar sounds produced by different characters. The encoding of a misspelled word will often match the encoding of the word that was intended. For example, in Listing 5.5, the Double Metaphone algorithm encodes the identifier *configured* and the term *configuration* which appears in the comment with code "KNFK" because they phonetically match. In contrast, using the same example for the detection of the variable name in the comment using lemma and SEthesaurus matching techniques would yield no matches.

```
//A flag to indicate configuration status
private boolean configured = false;
```

Listing 5.5. Example of advanced matching using Metaphonic technique (The Apache Software Foundation, 2021c).

C) SEthesaurus matching

The final matching technique used for the detection of the meaning of variable's identifier in the head comment is based on the SEthesaurus (Chen, Xing, and Ximing, 2017). SEthesaurus covers a large set of software-specific terms (52,645 terms), counting 4,773 abbreviations and 14,006 synonym groups, with high accuracy. This can be useful, in our case, to detect such abbreviations and the synonyms between the variables' identifiers and terms in the text of the comments.

```
/**
 * TRACE level integer value.
 *
 * @since 1.2.12
 */
public static final int TRACE_INT = 5000;
```

Listing 5.6. Example of advanced matching using SEthesaurus technique (The Apache Software Foundation, 2021d).

For instance, in Listing 5.6, the variable's identifier *TRACE_INT* consists of two terms: 1) TRACE and 2) INT, where the second part of the identifier *INT* is an abbreviation of its full form *integer*, which appears in the comment. It is worth noting that lemmas and metaphonic matching can fail to detect variables' identifiers in the text of the comments in similar situations.

5.3.4 Union of matching approaches

As each detection approach has its own advantages, we considered using a union of lexical and advanced matching approaches to overcome performance limitations and provide high detection coverage of the identifiers in the comments. We considered using two ways to unify these approaches: 1) a union of the three advanced approaches (i.e., lemmas, metaphone and SEthesaurus) and 2) a union of all the approaches (i.e., lexical and advanced approaches).

5.4 Study Design

5.4.1 Data collection

This section outlines the data set used to evaluate our approaches, the data set used for large-scale analysis, and the types of variables, comments, and scopes of the variables' declarations.

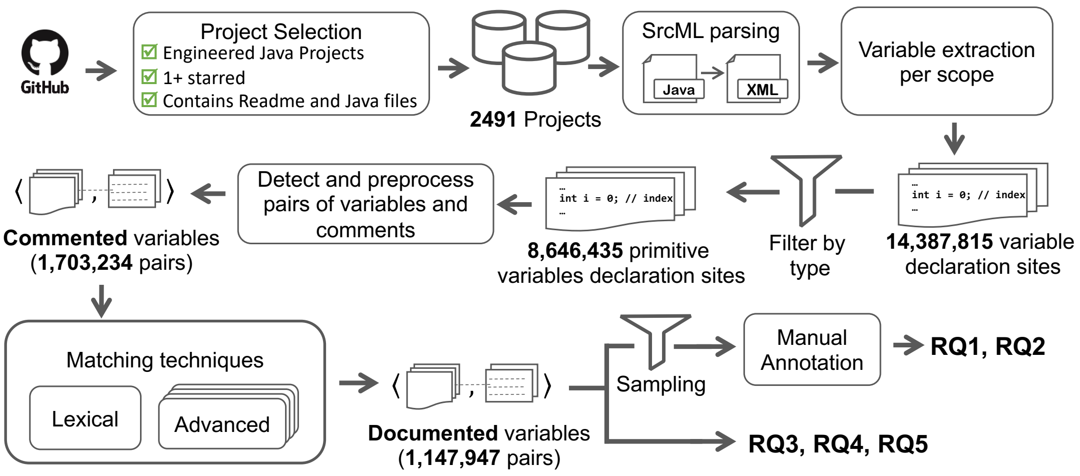

GitHub
Project Selection
Engineered Java Projects
1+ starred
Contains Readme and Java files
2491 Projects
SrcML parsing
Java
XML
Variable extraction per scope
int i = 0; // index
14,387,815 variable declaration sites
Filter by type
8,646,435 primitive variables declaration sites
Detect and preprocess pairs of variables and comments
Commented variables (1,703,234 pairs)
Matching techniques
Lexical
Advanced
Documented variables (1,147,947 pairs)
Sampling
Manual Annotation
RQ1, RQ2
RQ3, RQ4, RQ5

FIGURE 5.1. Overview of our study.

A) Repositories

Our repository preparation considers the data set used to evaluate our detection approaches and to study common characteristics of the documentation of primitive variables using large-scale quantitative analysis.

Table 5.1. Used variables for our detection approaches.

Data	**Evaluation data set**		**Large-scale data set**	
types	**# vars**	**# commented**	**# vars**	**# commented**
boolean	1,161	261 (4.91%)	733,294	125,598 (17.13%)
byte	2,755	181 (11.65%)	187,432	32,762 (17.48%)
char	158	29 (0.67%)	54,116	9,833 (18.17%)
int	7,592	1,952 (32.12%)	2,505,266	424,265 (16.93%)
short	58	12 (0.25%)	30,703	6,792 (22.12%)
long	4,715	976 (19.95%)	640,504	83,088 (12.97%)
float	76	26 (0.32%)	134,263	26,868 (20.01%)
double	669	160 (2.83%)	291,384	56,707 (19.46%)
String	6,456	1,501 (27.31%)	4,069,473	937,321 (23.03%)
Total	23,640	5,098 (21.57%)	8,646,435	1,703,234 (19.70%)

Table 5.2. Number of primitive and other variables.

	Primitive variables	**Other variables**
Fields	1,697,033	537,067
Local variables	3,020,165	3,693,085
Parameters	3,929,237	1,511,228
Total	8,646,435 (60.1%)	5,741,380 (39.9%)

Dataset used for large-scale analysis. Motivated by previous work on the promises and perils of mining GitHub (Kalliamvakou et al., 2014), which concluded that many repositories on GitHub do not contain engineered software projects, we used the RepoReaper framework (Munaiah et al., 2017) to obtain repositories for this work. RepoReaper was developed to differentiate between repositories with engineered software projects and those with noise (e.g., assignment projects). This is an important step, as noise projects could lead to incorrect conclusions in our study.

To select repositories for our study, we first obtained 17,243 Java projects that had at least one star and were classified as containing engineered software projects by the Random Forest classification of RepoReaper. We then eliminated 680 projects that no longer existed on GitHub (e.g., deleted or made private). For the remaining projects (16,563), we filtered out 1,441 projects that did not have README files and those projects (12,578) that did not have the word *documentation* in their README files. This step was necessary as we wanted to obtain only well-documented projects. From the remaining projects (2,544) we removed 53 projects that did not contain any Java files. Finally, in total, 1,040,026 Java files were collected from 2,491 projects to be used in this study.

Dataset used for evaluation. Six project repositories were used to evaluate our proposed approaches, motivated by previous work of Ratol and Robillard (Ratol and Robillard, 2017) to detect fragile comments. These projects (see Table 5.3) were selected because they were previously used in Ratol and Robillard's work (Ratol and Robillard, 2017) and had several advantages, such as their availability as open-source, a diversity of application domains, and being well-commented. To allow further investigation, the six projects were cloned locally to ease the process to analysis.

The same study (Ratol and Robillard, 2017) was followed to determine our sample size (i.e., we utilised stratified random sampling strategy to achieve diversity of variables in our sample). Stratified random sampling can prevent bias while ensuring that all classes of interest are covered in a sampled population. Following the procedure in this strategy, we randomly selected 100 variables from each of the six projects, in proportion to the number of variables declared in each scope (i.e., fields, local variables and methods' parameters) from the target population. We defined the target population as the primitive variables that have at least one comment above their declaration, and ones where the identifier appeared in the comment. For each project (see Table 5.3), we provided the total number of variables in a given scope, followed by the number of variables (in column *Pop*) of this scope for which our approach can detect the variable name in the comment above those variables' declarations. In column *Smp* we indicated the number of variables in this scope that were selected for the sample. For example, for the project *Chronicle Map*, the total number of variables detected in the field scope was 452, and there were ten variables for which our approaches could find a match in the comments above their declaration. Finally, the total number of variable identifiers that our advanced and lexical approaches detected with associated comments in the Chronicle Map project was 137.

B) Types of variables, comments and scopes

The source code in any programming language is divided into small pieces of code elements, for example, classes, methods, fields, etc. According to the syntax of the Java programming language, there are four types of comments:

Table 5.3. Evaluation sample.

Projects		**Fields**			**Local variables**			**Parameters**			**Total**	
Name	**Version**	**Total**	**Pop.**	**Smp.**	**Total**	**Pop.**	**Smp.**	**Total**	**Pop.**	**Smp.**	**Pop.**	**Smp.**
Chronicle Map	3.19.42	452	10	7	1,359	16	12	1,494	111	81	137	100
Joda time	2.10.6	847	82	5	2,230	26	2	3,559	1,392	93	1,500	100
JUnit	4.13	292	1	0	320	0	0	721	201	100	202	100
Log4j	1.9.0	813	126	28	1,053	12	2	1,330	313	70	451	100
Spring Data Redis	1.2.18	539	5	1	1,076	2	0	4,910	924	99	931	100
JFlex	2.3.1	755	109	27	989	31	7	901	269	66	409	100
Total	—	3,698	333	68	7,027	87	23	12,195	3,210	509	3,630	600

1) inline comment (i.e., `//` shown in the same line where the variable is declared), 2) line comment (i.e., `//` shown above variable declarations), 3) block comment (i.e., `/* ··· */`), and 4) Javadoc comment (i.e., `/** ··· */`).

In the present study, we focused on detecting the identifiers of primitive variables: boolean, byte, char, int, long, float, double and String, that are documented in these three types of comments (i.e., line and inline, block and Javadoc), declared in three categories of program elements (i.e., fields, local variables found in methods' bodies, and parameters of the methods). We considered the String as a primitive data type in Java source code, motivated by the fact that it is a primitive type in other languages, such as Python. As the first step, we collected all variables (1,703,234) that had at least one comment above their declarations (Table 5.1), and we then used our detection approaches to identify these identifiers within their associated comments.

C) Detecting documented variables in source code

We developed two approaches and then combined these two approaches to detect the variables' identifiers in the source code comments. These approaches are based on lexical or advanced matching (see Section 5.3) to capture variables' identifiers in their accompanying comments. In lexical matching, our matching technique captures exact matches (i.e., case sensitive) of these identifiers in the comments, while advanced matching captures the meaning of the identifier in the comments by applying lemmas, metaphonic and SEthesaurus matching techniques.

Our approach to detect the documented variables in their associated comments is shown in Figure 5.1. It is worth noting that projects used in our input source are classified as engineered software projects based on the RepoReaper tool (Munaiah et al., 2017), which was developed to distinguish between engineered projects and other projects. We make use of RepoReaper's Random Forest classification, trained with organisation and utility data sets.

After cloning the targeted projects from GitHub and removing all non-Java files, we employed of srcML (Collard, Decker, and Maletic, 2011) to convert the source code of each .java file into its XML representation. The advantage of using srcML is its ability to perfectly preserve the format of the original source at different levels, such as lexical, documentary (e.g., comments, white space), structural (e.g., classes, methods) and syntactic (e.g., statement).

Each of the XML versions was then used to facilitate the detection and the extraction of the variables with their head comments (i.e., comments that appeared immediately above the variables' declaration) and their scopes in which they appeared. Note: the white-spacing between the variables' declarations and their head comments were ignored. Finally, out of 14,387,815 variables obtained from 2,491 projects, we extracted 60.10% (8,646,435) as primitive variables (see Table 5.2). Of these primitive variables, 19.70% (1,703,234) had comments above their declarations, as shown in Table 5.1, to be used as input sources for our detection approaches.

5.4.2 Data analysis

In this section, we outline the data analysis methods used to answer the research questions.

A) Accuracy of detection approaches

To evaluate the accuracy of our lexical and advanced matching approaches, four authors of this study manually annotated 600 variables to investigate the presence of the variables' identifiers in the text of the comments. We considered the union of these techniques (lexical and advanced matching) to capture the variables in comments, and we evaluated the accuracy of each of these approaches individually. To calculate inter-rater agreement between the annotators, the annotators were split into six pairs (i.e., each of the four authors was paired with each of the other authors), where each pair annotated 100 randomly selected variables. Each pair of annotators proceeded to annotate each of the identifiers with their related comments to evaluate the accuracy of the matching techniques, and the presence of the additional information (see Table 5.4). We then measured the agreement reliability between pairs of annotators using Cohen's Kappa metric. We noted that the kappa reported is the average kappa value across all annotator pairs for each annotation question, which shows substantial to almost perfect agreement (Landis and Koch, 1977). We further studied the disagreements between pairs of annotators and resolved all conflicts to enhance the kappa agreement values for each question, that is with almost perfect agreement.

Table 5.4 shows our annotation questions, the answer options for each one, a description that shows how to answer each of these questions, and the average kappa value across all the annotators' pairs. Each pair of annotators used AQ1 and AQ3 to evaluate the lexical and advanced matching techniques performance, respectively. AQ1 investigates whether a variable identifier is literally documented in the comment, whereas AQ3 queries whether the meaning of the

Table 5.4. Annotation questions.

	Question	Answer	Description	avg. κ
AQ1	Was the variable name mentioned in the comment?	Yes / No	N/A	0.93
AQ2	Did the comment add additional information other than just mentioned the variable name?	Yes / No	N/A	0.82
AQ2.1	Did the comment provide any additional knowledge type about the purpose or patterns of the variable?	Yes / No	Explanation about the lifecycle of the variable in the comment; how to be referred/defined. This might be a part of the post-conditions of a method. In addition, the description of the initialisation process of the field variable can be treated as this category.	0.72
AQ2.2	Did the comment provide any additional knowledge types about the concept of the variable?	Yes / No	Explanation about the content of the variable in the comment. Sometimes, the variable's identifier is enough to be explained.	0.72
AQ2.3	Did the comment provide any additional knowledge types about the directive of the variable?	Yes / No	Explanation about the variable's domain, type or nullability as a part of the prerequisite for a method.	0.85
AQ3	Was the variable meaning mentioned in the comment?	Yes / No	Explanation about the variable's identifier in term of the meaning, which used to describe the variable in the comment.	0.75

identifier is documented in the comment. Finally, AQ2 investigates the existence of any type of knowledge regarding purpose, concept and directives found in the comment, associated with the identifiers and based on each answer to the sub-questions.

To annotate a variable's identifier based on AQ1, which is used to evaluate the performance of the lexical matching, we differentiated between variable identifiers by their typographical conventions; for example, an identifier consists of a single term or multiple terms, as described in Section 5.3.1. In case of an identifier composed of multiple terms, each annotator searches for all terms in the identifier and the corresponding words with the exact wording of each term in the text of the comment. The manual annotation process involved using the same sequence for the comment in which the terms appeared in the identifier while white-spacing between these terms was ignored. In the case of an identifier containing only a single term, the annotator looked for a literal match between the identifier and its corresponding word in the comment.

The performance of each of our matching techniques was then separately measured using the commonly used evaluation measures: recall, precision, and the F-score. Recall measures the degree of absence of false negatives, while precision measures the degree of absence of false positives. For example, perfect recall reported for an approach can indicate that the technique was able to detect all the variables' identifiers in their comments, which were also detected by manual inspection. It is worth mentioning that "recall", as used in our study, is an approximation, as a theoretical value of recall cannot be computed precisely because it would take too much effort to manually annotate the data.

B) Knowledge types present in comments

Since there is no existing taxonomy of information in the comments which would provide additional information around the variable identifiers, we established three types of knowledge, inspired by previous work of Maalej and Robillard (Maalej and Robillard, 2013). The description of these types was only applicable to API documentation. However, we identified three types of knowledge from their work, which was applicable in our context with some adjustments, namely purpose, concept and directives, and we then provided a description for each one of those types to fit our purpose, as illustrated in AQ2.1, 2.2 and 2.3 in Table 5.4.

Each pair of annotators then inspected each identifier and its accompanying comment to determine the existence of identifiers' purpose, concept and directives. We then quantitatively analysed how many of the variables' identifiers were documented in their accompanying comments with one or more of these types of knowledge.

C) Number of times variables' identifiers commented in source code

To investigate how frequently primitive variables are commented in the source code, we first obtained all the variables that have at least one comment (1,703,234 variables). Then, we quantitatively analysed the variables documented using the best matching approach, resulting from RQ1, to determine how many of the variables' identifiers are mentioned in their comments in different scopes of declaration.

D) Distribution of documented variables by their scopes of declaration

Our investigation in this study takes into account both the type of documented identifier and the scope in which they are declared. We hypothesise that some data types of variables tend to be documented in the comments in particular scopes more than other data types. Thus, to explore the distribution of these identifiers, we quantitatively analysed these variables by their scopes of declaration.

E) Types of comments associated with the documented variables by scope

We further investigated the types of comments associated with the documented data types of variables. For example, we investigated the scopes of the variables in which they are declared, the data types of these variables and the types of comments associated with these variables. We then analysed the descriptive statistics of our data set.

5.5 Findings

In this section we present our findings individually for each of the research questions.

5.5.1 RQ1: To what extent can different techniques detect variables in comments?

To understand the performance of lexical and advanced techniques used to detect variables' identifiers in the comments, we present our results of the evaluation of the lexical and the advanced matching in Table 5.5. The evaluation of these techniques was based on 600 variables manually annotated, as shown in Table 5.3, and using the annotation questions (i.e., AQ3), shown in Table 5.4, to assess the performance of all techniques. As a result of the evaluation process, we found that 584 variables' identifiers were correctly detected by our approaches along with their accompanying comments, whereas only 16 variables were not detected correctly.

Table 5.5. Accuracy of different matching techniques.

Matching techniques	Recall	Precision	F-score
Lexical Matching	0.896	0.994	0.942
Advanced Matching (Lemmas)	0.985	0.985	0.985
Advanced Matching (Metaphone)	0.995	0.973	0.984
Advanced Matching (SEthesaurus)	0.462	0.985	0.629
Union of all advanced matchings	1.000	0.973	0.986
Union of lexical and advanced matching	1.000	0.973	0.986

The overall performance based on the F-score for each of the techniques used to detect variables' identifiers, across all the scopes, showed that the ability of lemmas' and metaphonic matching techniques to capture the variables' identifiers in the comments were superior to the performances of the lexical and SEthesaurus approaches; that is, the F-score measures for lexical, lemmas, metaphonic and SEthesaurus were 0.942, 0.985, 0.984, and 0.629, respectively.

In terms of recall, SEthesaurus scored very low as compared to other approaches, and this can be attributed to the limited vocabulary in its dictionary, which was used to capture the abbreviations and synonyms of the identifiers in the text of comments. On the other hand, the recall of metaphonic matching, which can catch the spelling and pronunciation of the identifiers in the comments, scored the highest.

Furthermore, combining lexical and advanced matching resulted in a slight improvement (4.3%) of the F-score value, due to few false positives instances being detected when the metaphonic matching technique was used.

The main observation we can draw from these results is that the F-score value of advanced matching techniques to detect the identifiers in the comments is higher than for lexical matching. This means that advanced matching can detect the identifiers with 5% more true positive instances than lexical matching (555 true positive instances), which has ultimately impacted the value of the F-score of the union approach.

The ability of the advanced matching to detect the variables' identifier in comments based on F-score scored a higher value (0.986), compared to lexical matching, which scored 0.942.

5.5.2 RQ2: What types of knowledge do comments provide about the variables?

The answer to RQ2 reveals the types of additional information in comments about documented variables found in the 600 variables manually annotated. Table 5.6 shows three types of knowledge—purpose, concept and directive—that the developer might use to describe the identifiers in the comment. As shown in the table, the documented variables were grouped into scopes, and it shows the number of documented variables with any of these types of knowledge and percentages of the additional information of these knowledge types captured in the comment to describe the identifier.

For instance, our result revealed that 60.00% (6/10) of the boolean variables, declared in the field scope, have information in their accompanying comments that could explain the variable's concept. In contrast, 88.46% (23/26) of the String variables, declared in the field scope, were documented with their concept. We also observed that all variables declared in the parameters of the methods were documented with additional information covering the three knowledge types. This indicates that parameter variables are crucial elements in source code, to allow developers to provide additional information about their concept, purpose or directives in their accompanying comments.

In addition, combining the numbers of each data type of variables, from all the scopes can show how often variables with a particular data type were documented in comments with their concept, purpose or directives. In general, we can see only 20.67% (124) of these variables are documented in comments with

TABLE 5.6. Types of knowledge of purpose (P), concept (C), and directives (D) documented with each data type of variables in comments.

Data type	**Fields**				**Local variables**				**Parameters**				**Combined**			
	Count	**P** (%)	**C** (%)	**D** (%)	**Count**	**P** (%)	**C** (%)	**D** (%)	**Count**	**P** (%)	**C** (%)	**D** (%)	**Count**	**P** (%)	**C** (%)	**D** (%)
boolean	10	50.00	60.00	0.00	1	100.00	100.00	0.00	26	65.38	46.15	0.00	37	62.16	51.35	0.00
byte	0	N/A	N/A	N/A	0	N/A	N/A	N/A	17	64.71	58.82	47.06	17	64.71	58.82	47.06
char	1	0.00	0.00	100.00	0	N/A	N/A	N/A	1	100.00	100.00	0.00	2	50.00	50.00	50.00
int	25	28.00	72.00	4.00	14	42.86	42.86	0.00	134	74.63	74.63	19.40	173	65.32	71.68	15.61
short	0	N/A	N/A	N/A	0	N/A	N/A	N/A	1	100.00	100.00	0.00	1	100.00	100.00	0.00
long	6	33.33	83.33	0.00	3	33.33	33.33	0.00	88	80.68	72.73	17.05	97	76.29	72.16	15.46
float	0	N/A	N/A	N/A	0	N/A	N/A	N/A	10	100.00	80.00	10	10	100.00	80.00	10.00
double	1	0.00	100.00	0.00	3	0.00	100.00	0.00	32	68.75	81.25	9.38	36	61.11	83.33	8.33
String	26	34.62	88.46	11.54	2	50.00	50.00	0.00	199	64.82	51.26	33.17	227	61.23	55.51	30.40
Numeric	32	28.13	75.00	3.13	20	35.00	50.00	0.00	282	76.24	74.11	18.79	334	69.16	72.75	16.17
String and char	27	33.33	85.19	14.81	2	50.00	50.00	0.00	200	65.00	51.50	33.00	229	61.14	55.46	30.57
Total	69	33.33	76.81	7.25	23	39.13	52.17	0.00	508	71.26	63.78	23.43	600	65.67	64.83	20.67

directive information, compared to variables documented with their purpose 65.67% (394) or concept 64.83% (389)—developers rarely documented primitive variables with any directive information.

TABLE 5.7. Relationship between groups of data types of variables with respect to knowledge types.

	Data type of variables	Types of knowledge P	C	D
1	Numeric String and char	0.0484	0.0001	0.0001
2	Numeric boolean	0.3846	0.0067	0.0081
3	String and char boolean	0.9053	0.6413	0.0001

We further grouped the data types of the primitive variables into numeric (i.e., byte, int, long, short, float and double), String and char, and booleans to investigate whether data types of variable identifiers would affect the amount of additional information regarding purpose, concept and directives associated with variables that was documented in comments. We found that only 55.46% (127/229) of the identifiers of Strings and chars were documented in the comments with their concept, compared to 72.75% (243/334) of identifiers of numeric types.

Table 5.7 shows the associations between the groups of identifiers of type String and char vs. numeric to test the association between these groups. We found that the difference in documenting the concept of an identifier is statistically significant between numeric types and String/char using the Chi-square test with $p = 0.0001$. Numeric types were well documented in terms of their concept as compared to String and char; that is to say, numeric was more often conceptualised with meaning.

Additionally, we found that the information of type directives is strongly related to the data types of numeric, String and char, and boolean variables, with $p < 0.05$. For instance, we found that boolean identifiers are not documented in terms of directives (0/37), as expected, because their range is only two elements. In contrast, developers tend to provide additional information that describes the directives of identifiers of strings and chars more than numeric variables, 30.57% and 16.17%, respectively. This is likely because String variables are well directive-documented due to extra information, such as "nullability" that is usually documented with identifiers in the comment.

Developers documented the variables' identifiers of numeric data type with their purpose (69.16%) and concept (72.75%), more often than those of type String, which were documented for purpose (61.14%) and concept (55.46%).

5.5.3 RQ3: How frequently are primitive variables documented in comments?

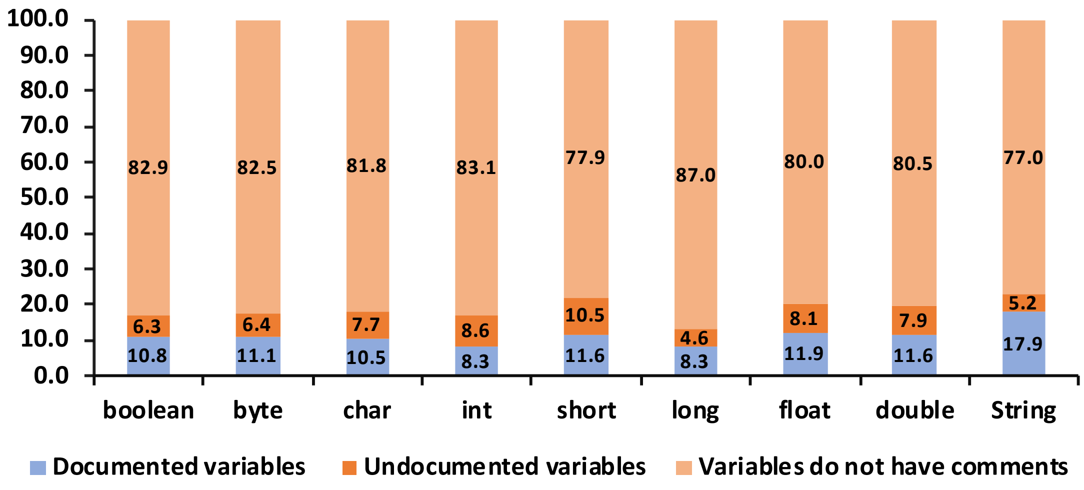


Figure 5.2. Percentage of variables documented, or r undocumented in the comment, and the percentage of variables that did not have comments.

Next, we used our large-scale data set which contains 1,703,234 (19.70%) commented variables (i.e., developers wrote a comment above the declaration of the variable) out of a total of 8,646,435 primitive variables (see Table 5.1), to investigate how often developers documented the variables' identifiers in comments, that is, developers mentioned the variables' identifiers in the associated comments. Out of 1,703,234 commented variables, our matching approaches were able to detect 67.40% (1,147,947) of the identifiers that were documented in their related comments, using the union approach, as its F-score scored the best value (0.986) among all detecting approaches (see RQ1). We used 1,147,947 documented variables to answer RQ3 and subsequent research questions, RQ4 and RQ5. We now report on findings in answer to RQ3.

Figure 5.2 shows for each of the data types 1) the percentage of the variables that have comments and are documented in their accompanying comments, 2) the percentage of the variables that have comments but are undocumented (i.e., not mentioned in their accompanying comments), and 3) the percentage of variables that do not have comments.

We can see from the figure that the percentage of documented variables in the comments compared to undocumented variables is high except for the int data type where the opposite was observed. Moreover, 17.9% of String variables were documented in their comments compared to undocumented (5.2%), which showed the highest ratio among all the documented variables, followed by byte, boolean and float. This can be explained, as developers tend to document the identifiers of String more than other data types, due to its capability to store any data ranging from textual information to symbols and numbers, which might be required by the developers to provide additional information of type directive to describe the content of the String variables. In addition, 80.30% (6,943,201) of the primitive variables were found to be uncommented, where the long data type showed the highest ratio of uncommented variables among other primitive types. The String data type, on the other hand, showed the lowest ratio of uncommented variables compared to other primitive variables.

80.30% of the variables did not have comments, and among those commented variables, 13.28% of the variables were documented in their comment, while 6.42% of the variables were undocumented.

5.5.4 RQ4: What are the distributions of documented variables by their scope of declarations?

As reported in Section 5.5.3, 13.29% (1,147,947) of the variables were documented in their related comments. For these 1,147,947 variables, Figure 5.3 illustrates the distribution of variable data types in their accompanying comments in each scope. Identifiers of the String data type, as declared in methods' parameters, were usually documented (67.02%) in their associated comments. In contrast, identifiers of the data types of int and boolean were most often documented in local and field scopes (33.70% and 10.95%, respectively).

Comparing data types of each variable with their declaration scopes, we found that String 63.39% (727,631), int 19.12% (208,058), and boolean 6.91% (79,283) data types were frequently used and documented across all the scopes.

On the other hand, the lowest number of variables documented in the comments were short (0.31%), char (0.49%), float (1.39%) and double (2.93%) data types across all the scopes of declaration.

Variables declared in methods' parameters were more documented (85.32%) in their comments than those declared in the scopes of field (10.98%) and methods' bodies (3.70%). The high number of documented variables in parameters

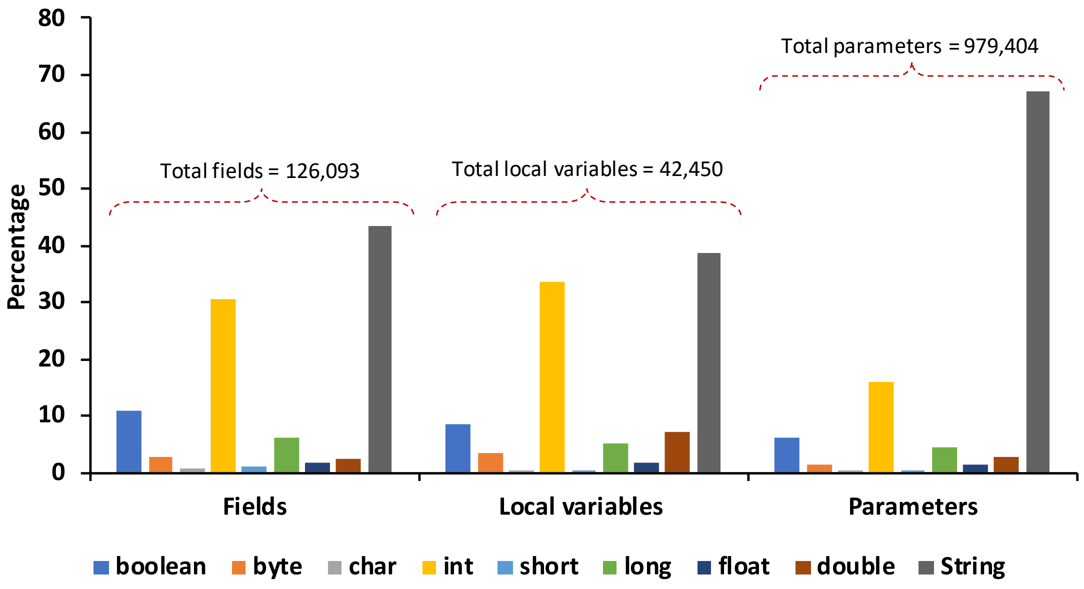


FIGURE 5.3. Percentage of variable data types declared in each scope.

might be because comments were automatically generated by Javadoc and usually mentioned the variables' identifiers in the comment. In contrast, the lower numbers of the variables documented in the methods' bodies were due to their smaller scope, where developers tend not to document these identifiers in the comments.

Across all scopes, developers often use and document variable identifiers of type String (63.39%), int (19.12%) and boolean (6.91%) in their accompanying comments more than other primitive data types. Variables declared in methods' parameters were more documented in their comments than those declared in scopes of field and methods' bodies.

5.5.5 RQ5: What are the types of comments associated with the scopes of the documented variables?

Finally, we report our result that reveal the types of comments associated with the documented variables in each scope, as shown in Table 5.8. The results can be interpreted in the same way as those in Figure 5.3, except that in this case, we show the types of comments (e.g., inline, line, block and Javadoc) associated with these data types and variables per scope. Our result reveals that a vast majority of the variables declared in the methods' parameters were documented in a comment of type Javadoc 99.04% (969,976), followed by block 0.5% (4,878).

TABLE 5.8. Percentage of the variables of different data types documented with different types of comments in each scope.

Data Type	**Fields**				**Local variables**				**Parameter variables**			
	Inline	**Line**	**Block**	**Javadoc**	**Inline**	**Line**	**Block**	**Javadoc**	**Inline**	**Line**	**Block**	**Javadoc**
boolean	0.57	1.43	0.24	8.71	0.66	7.51	0.42	0.10	0.00	0.03	0.03	6.26
byte	1.04	0.48	0.04	1.40	0.17	3.28	0.16	0.02	0.00	0.01	0.02	1.55
char	0.02	0.05	0.02	0.58	0.04	0.45	0.04	0.00	0.00	0.00	0.01	0.46
int	1.53	2.66	0.47	25.75	2.95	28.82	1.58	0.33	0.00	0.14	0.18	15.54
short	0.02	0.03	0.02	0.92	0.04	0.27	0.00	0.00	0.00	0.00	0.00	0.22
long	0.26	0.75	0.15	4.98	0.41	4.48	0.23	0.07	0.00	0.02	0.02	4.40
float	0.07	0.12	0.02	1.58	0.21	1.60	0.05	0.03	0.00	0.01	0.01	1.29
double	0.38	0.26	0.06	1.92	0.82	5.84	0.63	0.05	0.00	0.04	0.02	2.72
String	0.96	2.96	0.63	38.93	1.76	35.30	1.45	0.19	0.00	0.20	0.22	66.60

Similarly, out of 126,093 variables declared in field scope, we found that 84.77% of these variables were documented using Javadoc, followed by line comment (8.76%). Finally, 87.55% of local variables were documented using line comment, followed by inline comment (7.08%).

97.41% of the variables declared in field and parameter scopes were documented in Javadoc comments, while 87.55% of local variables were documented using line comments.

5.6 Threats to Validity

As with any empirical study, for our study, there are a number of threats that may affect our results' validity.

First, the taxonomy used to categorise the documentation of variables was based on three types of knowledge inspired by Maalej and Robillard (Maalej and Robillard, 2013), and adapted to fit our purpose. Future work should explore other dimensions of these types of knowledge.

Second, the number of primitive variables used to evaluate our detection approaches was small, which could introduce bias, but this limitation was necessary due to the laborious task of manual annotation. As well, the recall value referred to in this work is an approximation, as a theoretical value of the recall was not computed precisely, because again, a great deal of effort would be required to manually annotate the data.

Finally, our work may not generalise to other programming languages, especially those that are loosely typed. Therefore, we cannot claim that our findings are generalisable beyond the evaluation data set used for this work.

5.7 Related Work

In this section, we discuss literature related to approaches for the detection of code comment inconsistencies and taxonomies of knowledge types.

Automatic assessment of comment quality: Researchers have developed tools and metrics to measure the quality of code comments. For example, Khamis et al. (Khamis, Witte, and Rilling, 2010) developed a tool called JavadocMiner based on natural language processing (NLP) to assess the quality of Javadoc comments by evaluating the "quality" of the language used in the comment as well as its consistency with the source code. The quality of the language is assessed using readability metrics such as the Gunning Fog Index and combined with the several heuristics, e.g., checking whether the comment uses well-formed sentences, including nouns and verbs. Furthermore, the consistency between code and comments is checked with a heuristic-based approach. For example, a method having a return type and parameters is expected to have these elements documented in the Javadoc with @return and @param.

In another work, Steidl et al. (Steidl, Hummel, and Juergens, 2013) proposed a machine learning model for comment quality analysis and assessment based on various comment categories including header comments, member comments, in-line comments, section comments, code comments and task comments. The model described the quality attributes in terms of coherence, consistency, completeness, and usefulness, and assessed two quality metrics. The validity of the model was then evaluated with a survey among 16 experienced developers.

Similarly, Sun et al. (Sun et al., 2016) extended the work of Steidl et al. by evaluating code comments in jdk8.0 and jEdit to provide comprehensive comment assessment and recommendation. The study consists of header comment analysis and method comment analysis that highlights the correlation between method's name and comment.

Hata et al. (Hata et al., 2019) investigate the role of links contained in source code comments. They find that licenses, software homepages, and specifications are among the most prevalent types of link targets, and that links are often used to provide metadata or attribution.

Hao (He, 2019) carried out an approach to understand comments in source code, investigating whether projects practice commenting differently and what may cause the differences. They chose five programming languages including JavaScript, Java, C++, Python and Go. They found that the most commented project was a Java design patterns project and that comments in Java and Python were more prevalent than in C++.

Other studies have mainly focused on the automatic detection of code-comment inconsistencies such as the work presented by Tan and colleagues (Tan et al., 2007; Tan et al., 2012). In their work, they presented iComment (Tan et al., 2007) which combined program analysis, a technique using NLP and machine learning to detect code-comment inconsistencies. iComment can detect inconsistencies related to the usage of locking mechanisms in code and their description in comments. They also presented @TCOMMENT in their follow-up work (Tan et al., 2012). @TCOMMENT is an approach which is able to test the consistency between Javadoc comments related to null values and exceptions with the behaviour of the related method's body.

A rule-based approach named Fraco was proposed by Ratol et al. (Ratol and

Robillard, 2017) to detect code-comment inconsistencies resulting from rename refactoring operations performed on identifiers. Their evaluation shows the superior performance achieved by Fraco as compared to the rename refactoring support implemented in Eclipse. Our work, while not related to the automatic assessment of comments quality, provides an empirical study on how developers tend to document identifiers of primitive variables in source code comments.

Taxonomy of knowledge types: Padioleau et al. (Padioleau, Tan, and Zhou, 2009) proposed a taxonomy based on meanings of comments and manually classifiers 1,050 comments. They found that 52.6% of these comments can be leveraged to improve software reliability and increase developer productivity. Monperrus et al. (Monperrus et al., 2012) empirically studied API directives which are constraints about usages of APIs and they built a corresponding taxonomy. Maalej and Robillard (Maalej and Robillard, 2013) leveraged grounded methods and analytical approaches to build a taxonomy of knowledge types in API reference documentation and manually classified 5,574 randomly sampled documentation units to assess the knowledge they contain. In contrast to those studies, we developed a taxonomy of knowledge types for identifiers detected in the code comments, as existing taxonomies did not fit the purpose of our particular study.

5.8 Implications

A) Documenting primitive variables in Java source code

Primitive variable data types are fundamental elements in any programming language. Because of the nature of these variables, which requires developers to inject meaning in the accompanying comments, their role might be difficult to understand if not sufficiently explained in code comments, which could ultimately reduce the productivity of developers.

Our results indicate which variables in what context tend to be documented and which do not. Knowing which primitive variables (i.e., most of the variables in our data set) tend to be documented and what knowledge this documentation contains are first steps towards raising awareness among developers about what should be documented and what is often missing. This can provide a pathway towards integrated automated tooling to assess the presence of knowledge types and to potentially issue recommend changes in cases of lack of knowledge types in comments related to the variables' identifiers. We plan similar work on non-primitive variables and are considering expanding the taxonomy of knowledge types regarding variables' identifiers documented in comments to cover more knowledge types, which could help reveal additional information linked to these identifiers.

B) The potential of code comments for summarisation:

Our results, reported in Section 5.5.2, showed the amount of additional information (i.e., knowledge types: purpose, concept, and directive) that the comments could contain to explain the use of primitive variables in source code. To enrich the automatic summaries with information involved in the developers' development activities, we needed, as the first step, to assess the knowledge types of primitive variables that comments can contain.

Assessing the information contained in the comments is an essential step towards knowing whether inclusion of these comments in summaries would benefit the developers' understanding of their development activities—given that the source code is regarded as one of the most important artefacts for creation of software applications and the code comment is a primary way of documenting the source code. However, comments can be fragile items that may not provide additional information about the variables in the source code. Therefore, assessing the knowledge types contained in comments could reveal additional information about the developers' development activities in their projects, thus enriching the automatically generated summaries.

Our results (Table 5.6) reveal the types of knowledge found in the comments in our evaluation data set. The table shows that 65.67% of the variables were documented with information related to purpose; 64.83% were documented with information relating to concepts; and 20.67% of the variables were documented with directives. These results highlight that the developers provided additional information about how the variables could be used, what they contained, and their domains. Thus, including source code comments as an additional artefact beside the 15 types of artefacts (see Chapter 4.2) to the process of summarisation would add to our knowledge about the developers' development activities.

Chapter 6

Conclusion and Future Work

The automatic generation of multi-document summaries from heterogeneous software artefacts is a promising research direction in the software engineering field, as demonstrated by this present contribution. Prior to the work described in this thesis, there was little known about summarising heterogeneous data from different kinds of software artefacts in a given time frame. Previous studies looked at automated summary production from only one single type of artefact, and they did not consider the process in terms of a time dimension. In this thesis, we proposed to address this gap in the research in order to help software developers better understand their projects' development activities and to meet their growing information needs. In addition, we evaluated comments contained in Java source code files for the types of knowledge they contained about primitive variables. These findings showed that the source code comments did contain additional information about developers' activities that could be used for summarisation purposes.

This dissertation makes the following specific contributions:

- We created our gold-standard summaries, investigated their text properties and then used these summaries to identify which types of software artefacts reflected the developers' development activities. We then investigated whether GitHub developers actually used these software artefacts in their projects by conducting a large-scale analysis of 1,038 software engineered projects. Our results indicated that our identified software artefacts were commonly used by GitHub developers working on engineered software projects: hence, they could be useful inputs for our summarisation techniques.

- We proposed, implemented and evaluated our first framework for summarising multi-document software artefacts containing heterogeneous data within a given time frame. We used a variety of optimisation heuristics algorithms to extract text from the software artefacts in such a way that the resulting summaries were similar in style to those found in the gold-standard summaries. The generated summaries were then extrinsically evaluated by means of a user study. This indicated that experts found the generated summaries useful in identifying all of the key project activities listed in the gold-standard summaries.

- We noted that the gold-standard summaries showed no clear evidence that the student developers cited information related to the source code files. However, source codes are considered one of the most important artefacts in the development of software applications, and comments are a primary source of information for documenting the source code. We decided to assess source code comments for their potential for summarisation, dividing them into three types of knowledge: purpose, concept, and directive. We then designed and conducted a study to ascertain how often developers document variable identifiers in their related comments, and what additional information the comments provide with respect to these identifiers. Our findings showed that there is potential for including source code comments for summarisation as an additional source of information about developers' activities.

There are several directions that future studies could take. For example, we found 15 types of software artefacts mentioned in the student summaries (gold-standard summaries) in describing their development activities. However, due to the low number of summaries produced by the students, it is unclear whether our findings can be generalised to other GitHub repository projects. For example, we found no clear evidence that the source code artefact was leveraged in the student summaries. In future work we could include source code comments and other types of artefacts that describe developmental activities. In addition, because we started with a base of only a few hundred summaries, creating reliable machine learning-based summaries was difficult. So the construction of standard large-scale summarisation datasets will be an important future direction to advance this field of research.

In addition, we employed cosine similarity in our optimisation algorithms as the fitness function to compute the similarity between the artefacts' sentences and the sentences in our target vectors (i.e., the gold-standard summaries) using either a bag of words or high-dimensional feature vectors. However, when comparing two sentences closely connected, but with no words in common, a bag of words representation does not allow for dealing with the semantic similarities between concepts. We plan to improve our similarity measure by applying word embeddings (e.g., Word2Vec (Mikolov et al., 2013) or Glove (Pennington, Socher, and Manning, 2014)) to capture the semantic word relationships, to attempt to improve the quality of the automatic generated summaries.

Another dimension for future work could be to generate automatic summaries without relying on gold-standard summaries. The present summarisation approach has demonstrated the effectiveness of automatically generated summaries in assisting developers to understand their development activities within a given time frame. We plan to build on this by developing machine-learning based extractive and abstractive summarisation systems in practical settings to provide developers with instant summaries.

Finally, the summarisation systems could be enhanced by involving user interactions in the summarisation process to provide the developers with customised summaries. For example, a user could modify the input parameters, such as type of software artefact and the time duration the developers choose for summary generation (e.g., number of days or weeks), and thus summaries could be automatically constructed and updated based on a series of adjusted parameters.

Bibliography

Afantenos, Stergos, Vangelis Karkaletsis, and Panagiotis Stamatopoulos (2005). "Summarization from medical documents: a survey". In: *Artificial intelligence in medicine* 33.2, pp. 157–177.

Alghamdi, Mahfouth, Shinpei Hayashi, Takashi Kobayashi, and Christoph Treude (2021a). "Characterising the Knowledge about Primitive Variables in Java Code Comments". In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 460–470. DOI: `10.1109/MSR52588.2021.00058`.

Alghamdi, Mahfouth, Shinpei Hayashi, Takashi Kobayashi, and Christoph Treude (2021b). *Online Appendix of "Characterising the Knowledge about Primitive Variables in Java Code Comments"*. Zenodo. `https://doi.org/10.5281/zenodo.4626387`.

Alghamdi, Mahfouth, Christoph Treude, and Markus Wagner (2019). "Toward Human-like Summaries Generated from Heterogeneous Software Artefacts". In: *Genetic and Evolutionary Computation Conference Companion*. Prague, Czech Republic: ACM, 1701–1702. ISBN: 9781450367486.

Alghamdi, Mahfouth, Christoph Treude, and Markus Wagner (2020). "Human-Like Summaries from Heterogeneous and Time-Windowed Software Development Artefacts". In: *International Conference on Parallel Problem Solving from Nature*. Springer, pp. 329–342.

Allahyari, Mehdi, Seyedamin Pouriyeh, Mehdi Assefi, Saeid Safaei, Elizabeth D Trippe, Juan B Gutierrez, and Krys Kochut (2017). "Text Summarization Techniques: A Brief Survey". In: *International Journal of Advanced Computer Science and Applications (ijacsa)* 8.10.

Balaji, J, TV Geetha, and Ranjani Parthasarathi (2016). "Abstractive summarization: A hybrid approach for the compression of semantic graphs". In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 12.2, pp. 76–99.

Blasi, Arianna, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos (2018). "Translating code comments to procedure specifications". In: *Proceedings of the 27th*

ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 242–253.

Buse, Raymond PL and Westley R Weimer (2010). “Automatically documenting program changes”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 33–42.

Chen, Chunyang, Zhenchang Xing, and Wang Ximing (2017). “Unsupervised Software-Specific Morphological Forms Inference from Informal Discussions”. In: *Proceedings of the 39th International Conference on Software Engineering*, pp. 450–461.

Chen, Huanchao, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo (2019). “Automatically detecting the scopes of source code comments”. In: *Journal of Systems and Software* 153, pp. 45–63.

Collard, Michael L, Michael J Decker, and Jonathan I Maletic (2011). “Lightweight transformation and fact extraction with the srcML toolkit”. In: *Proceedings of the 11th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 173–184.

Cortés-Coy, Luis Fernando, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk (2014). “On automatically generating commit messages via summarization of source code changes”. In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, pp. 275–284.

Dabbish, Laura, Colleen Stuart, Jason Tsay, and Jim Herbsleb (2012). “Social coding in GitHub: transparency and collaboration in an open software repository”. In: *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pp. 1277–1286.

Dale, Edgar and Jeanne S Chall (1948). “A formula for predicting readability: Instructions”. In: *Educational research bulletin*, pp. 37–54.

Edmundson, Harold P (1969). “New methods in automatic extracting”. In: *Journal of the ACM (JACM)* 16.2, pp. 264–285.

Erkan, Günes and Dragomir R Radev (2004). “Lexrank: Graph-based lexical centrality as salience in text summarization”. In: *Journal of artificial intelligence research* 22, pp. 457–479.

Ferreira, Rafael, Frederico Freitas, Luciano de Souza Cabral, Rafael Dueire Lins, Rinaldo Lima, Gabriel França, Steven J Simske, and Luciano Favaro (2014). “A context based text summarization system”. In: *2014 11th IAPR international workshop on document analysis systems*. IEEE, pp. 66–70.

Flesch, Rudolph (1948). “A new readability yardstick.” In: *Journal of applied psychology* 32.3, p. 221.

Goffi, Alberto, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè (2016). "Automatic generation of oracles for exceptional behaviors". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pp. 213–224.

Goldstein, Jade, Vibhu O Mittal, Jaime G Carbonell, and Mark Kantrowitz (2000). "Multi-document summarization by sentence extraction". In: *NAACL-ANLP 2000 Workshop: Automatic Summarization.*

Grover, Claire, Ben Hachey, and Chris Korycinski (2003). "Summarising legal texts: Sentential tense and argumentative roles". In: *Proceedings of the HLT-NAACL 03 Text Summarization Workshop*, pp. 33–40.

Gupta, Vishal and Gurpreet Singh Lehal (2010). "A survey of text summarization extractive techniques". In: *Journal of emerging technologies in web intelligence* 2.3, pp. 258–268.

Hahn, Udo and Martin Romacker (2001). "The SynDiKATe Text Knowledge Base Generator". In: *Proceedings of the First International Conference on Human Language Technology Research.*

Haiduc, Sonia, Jairo Aponte, and Andrian Marcus (2010). "Supporting program comprehension with source code summarization". In: *2010 acm/ieee 32nd international conference on software engineering.* Vol. 2. IEEE, pp. 223–226.

Haiduc, Sonia, Jairo Aponte, Laura Moreno, and Andrian Marcus (2010). "On the use of automated text summarization techniques for summarizing source code". In: *2010 17th Working Conference on Reverse Engineering.* IEEE, pp. 35–44.

Hartzman, Carl S and Charles F Austin (1993). "Maintenance productivity: Observations based on an experience in a large system environment". In: *Proceedings of the 3rd Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering.* Vol. 1, pp. 138–170.

Hata, Hideaki, Christoph Treude, Raula Gaikovina Kula, and Takashi Ishio (2019). "9.6 million links in source code comments: Purpose, evolution, and decay". In: *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, pp. 1211–1221.

He, Hao (2019). "Understanding source code comments at large-scale". In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1217–1219.

Hirao, Tsutomu, Masaaki Nishino, Yasuhisa Yoshida, Jun Suzuki, Norihito Yasuda, and Masaaki Nagata (2015). "Summarizing a document by trimming the discourse tree". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23.11, pp. 2081–2092.

Hovy, Eduard and Daniel Marcu (2005). "Automated text summarization". In: *The Oxford Handbook of computational linguistics* 583598.

Hu, Xing, Ge Li, Xin Xia, David Lo, and Zhi Jin (2018). "Deep code comment generation". In: *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, pp. 200–20010.

Huang, Yuan, Shaohao Huang, Huanchao Chen, Xiangping Chen, Zibin Zheng, Xiapu Luo, Nan Jia, Xinyu Hu, and Xiaocong Zhou (2020). "Towards automatically generating block comments for code snippets". In: *Information and Software Technology* 127, p. 106373.

Huß, Roland (2021). *jolokia*. URL: `https://github.com/rhuss/jolokia/blob/cd3a93fad780b5a0b073bef005c4427fae540402/agent/jvm/src/test/java/org/jolokia/jvmagent/JolokiaServerTest.java#L301` (visited on 03/21/2021).

Jiang, He, Jingxuan Zhang, Hongjing Ma, Najam Nazar, and Zhilei Ren (2017). "Mining authorship characteristics in bug repositories". In: *Science China Information Sciences* 60.1, pp. 1–16.

Jiang, Zhen Ming and Ahmed E Hassan (2006). "Examining the evolution of code comments in PostgreSQL". In: *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pp. 179–180.

JogAmp Community (2021). *JOGL*. URL: `https://github.com/sgothel/jogl/blob/ecf6e499d3b582d651a28693c871ca14d6e8c991/src/jogl/classes/jogamp/graph/geom/plane/Crossing.java#L200` (visited on 03/21/2021).

Jones, K Sparck et al. (1999). "Automatic summarizing: factors and directions". In: *Advances in automatic text summarization*, pp. 1–12.

Kalliamvakou, Eirini, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian (2014). "The promises and perils of mining GitHub". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 92–101.

Khamis, Ninus, René Witte, and Juergen Rilling (2010). "Automatic quality assessment of source code comments: The JavadocMiner". In: *Proceedings of the 15th International Conference on Application of Natural Language to Information Systems*, pp. 68–79.

Kikuchi, Yuta, Tsutomu Hirao, Hiroya Takamura, Manabu Okumura, and Masaaki Nagata (2014). "Single document summarization based on nested tree structure". In: *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 315–320.

Kupiec, Julian, Jan Pedersen, and Francine Chen (1995). "A trainable document summarizer". In: *Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 68–73.

Landis, J Richard and Gary G Koch (1977). "The measurement of observer agreement for categorical data". In: *Biometrics* 33.1, pp. 159–174.

Liu, Zhongxin, Xin Xia, Christoph Treude, David Lo, and Shanping Li (2019). "Automatic generation of pull request descriptions". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 176–188.

Lotufo, Rafael, Zeeshan Malik, and Krzysztof Czarnecki (2015). "Modelling the 'hurried'bug report reading process to summarize bug reports". In: *Empirical Software Engineering* 20.2, pp. 516–548.

Luhn, Hans Peter (1958). "The automatic creation of literature abstracts". In: *IBM Journal of research and development* 2.2, pp. 159–165.

Maalej, Walid and Martin P Robillard (2013). "Patterns of knowledge in API reference documentation". In: *IEEE Transactions on Software Engineering* 39.9, pp. 1264–1282.

Maaten, L. v. d. and G. Hinton (2008). "Visualizing data using t-SNE". In: *Journal of Machine Learning Research* 9, pp. 2579–2605.

Mani, Inderjeet, David House, Gary Klein, Lynette Hirschman, Therese Firmin, and Beth M Sundheim (1999). "The TIPSTER SUMMAC text summarization evaluation". In: *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pp. 77–85.

Mani, Senthil, Rose Catherine, Vibha Singhal Sinha, and Avinava Dubey (2012). "Ausum: approach for unsupervised bug report summarization". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11.

Manning, Christopher D, Prabhakar Raghavan, and Hinrich Schütze (2008). *Introduction to information retrieval.* Cambridge University Press.

Manning, Christopher D, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky (2014). "The Stanford CoreNLP natural language processing toolkit". In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60.

McKeown, Kathleen, Rebecca J Passonneau, David K Elson, Ani Nenkova, and Julia Hirschberg (2005). “Do summaries help?” In: *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 210–217.

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean (2013). “Distributed representations of words and phrases and their compositionality”. In: *Advances in neural information processing systems*, pp. 3111–3119.

Mishra, Ritwik and Tirthankar Gayen (2018). “Automatic Lossless-Summarization of News Articles with Abstract Meaning Representation”. In: *Procedia Computer Science* 135, pp. 178–185.

Monperrus, Martin, Michael Eichberg, Elif Tekes, and Mira Mezini (2012). “What should developers be aware of? An empirical study on the directives of API documentation”. In: *Empirical Software Engineering* 17.6, pp. 703–737.

Moreno, Laura, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker (2013). “Automatic generation of natural language summaries for java classes”. In: *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, pp. 23–32.

Moreno, Laura, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora (2014). “Automatic generation of release notes”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 484–495.

Munaiah, Nuthan, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan (2017). “Curating GitHub for engineered software projects”. In: *Empirical Software Engineering* 22.6, pp. 3219–3253.

Nair, Vivek, Amritanshu Agrawal, Jianfeng Chen, Wei Fu, George Mathew, Tim Menzies, Leandro Minku, Markus Wagner, and Zhe Yu (2018). “Data-Driven Search-Based Software Engineering”. In: *15th International Conference on Mining Software Repositories (MSR)*. Gothenburg, Sweden: ACM, 341–352. ISBN: 9781450357166.

Nazar, Najam, He Jiang, Guojun Gao, Tao Zhang, Xiaochen Li, and Zhilei Ren (2016). “Source code fragment summarization with small-scale crowdsourcing based features”. In: *Frontiers of Computer Science* 10.3, pp. 504–517.

Nenkova, Ani and Kathleen McKeown (2012). “A SURVEY OF TEXT SUMMARIZATION TECHNIQUES”. In: *Mining Text Data*, p. 43.

Nenkova, Ani, Kathleen McKeown, et al. (2011). “Automatic Summarization”. In: *Foundations and Trends® in Information Retrieval* 5.2–3, pp. 103–233.

Ou, Shiyan, Christopher SG Khoo, and Dion H Goh (2006). "Automatic multi-document summarization for digital libraries". In:

Ouyang, You, Wenjie Li, Qin Lu, and Renxian Zhang (2010). "A study on position information in document summarization". In: *Coling 2010: Posters*, pp. 919–927.

Padioleau, Yoann, Lin Tan, and Yuanyuan Zhou (2009). "Listening to programmers—Taxonomies and characteristics of comments in operating system code". In: *Proceedings of the 31st International Conference on Software Engineering*, pp. 331–341.

Pandita, Rahul, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar (2012). "Inferring method specifications from natural language API descriptions". In: *Proceedings of the 34th International Conference on Software Engineering*, pp. 815–825.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Peyrard, Maxime (2019). "A Simple Theoretical Model of Importance for Summarization". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 1059–1073.

Peyrard, Maxime and Judith Eckle-Kohler (2016). "A general optimization framework for multi-document summarization using genetic algorithms and swarm intelligence". In: *26th International Conference on Computational Linguistics: Technical Papers (COLIN)*, pp. 247–257.

Philips, Lawrence (2000). "The double metaphone search algorithm". In: *C/C++ Users Journal* 18.6, pp. 38–43.

Qazvinian, Vahed and Dragomir R Radev (2008). "Scientific paper summarization using citation summary networks". In: *arXiv preprint arXiv:0807.1560.*

Qiang, Ji-Peng, Ping Chen, Wei Ding, Fei Xie, and Xindong Wu (2016). "Multi-document summarization using closed patterns". In: *Knowledge-Based Systems* 99.C, pp. 28–38.

Radev, Dragomir R, Sasha Blair-Goldensohn, and Zhu Zhang (2001). "Experiments in single and multidocument summarization using MEAD". In: *First document understanding conference.* Citeseer, 1À8.

Rastkar, Sarah, Gail C Murphy, and Gabriel Murray (2014). "Automatic summarization of bug reports". In: *IEEE Transactions on Software Engineering* 40.4, pp. 366–380.

Ratol, Inderjot Kaur and Martin P Robillard (2017). “Detecting fragile comments”. In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 112–122.

Rautray, Rasmita and Rakesh Chandra Balabantaray (2017). “Cat swarm optimization based evolutionary framework for multi document summarization”. In: *Physica A: Statistical Mechanics and its Applications* 477, pp. 174–186.

Reeve Lawrence, H, Han Hyoil, V Nagori Saya, C Yang Jonathan, A Schwimmer Tamara, and D Brooks Ari (2006). “Concept frequency distribution in biomedical text summarization”. In: *ACM 15th Conference on Information and Knowledge Management (CIKM), Arlington, VA, USA*.

Revelle, William and Maintainer William Revelle (2015). “Package ‘psych’”. In: *The comprehensive R archive network* 337, p. 338.

Rigby, Peter C and Martin P Robillard (2013). “Discovering essential code elements in informal documentation”. In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, pp. 832–841.

Roehm, Tobias, Rebecca Tiarks, Rainer Koschke, and Walid Maalej (2012). “How do professional developers comprehend software?” In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 255–265.

Rubio-González, Cindy and Ben Liblit (2010). “Expect the unexpected: Error code mismatches between documentation and the real world”. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 73–80.

Saggion, Horacio and Guy Lapalme (2002). “Generating indicative-informative summaries with sumum”. In: *Computational linguistics* 28.4, pp. 497–526.

Saggion, Horacio and Thierry Poibeau (2012). “Automatic Text Summarization: Past, Present and Future”. In: *Multi-source, Multilingual Information Extraction and Summarization*, p. 1.

Saggion, Horacio and Thierry Poibeau (2013). “Automatic Text Summarization: Past, Present and Future”. In: *Multi-source, Multilingual Information Extraction and Summarization*, pp. 3–21.

Sarda, AT and AR Kulkarni (2015). “Text summarization using neural networks and rhetorical structure theory”. In: *International Journal of Advanced Research in Computer and Communication Engineering* 4.6, pp. 49–52.

Shannon, Claude Elwood (1948). “A mathematical theory of communication”. In: *The Bell system technical journal* 27.3, pp. 379–423.

Silva, Catarina and Bernardete Ribeiro (2003). "The importance of stop word removal on recall values in text categorization". In: *Proceedings of the International Joint Conference on Neural Networks.* Vol. 3, pp. 1661–1666.

Sohangir, Sahar and Dingding Wang (2017). "Improved sqrt-cosine similarity measurement". In: *Journal of Big Data* 4.1, p. 25.

Souza, Sergio Cozzetti B de, Nicolas Anquetil, and Káthia M de Oliveira (2005). "A study of the documentation essential to software maintenance". In: *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75.

Sridhara, Giriprasad, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker (2010). "Towards automatically generating summary comments for java methods". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52.

Steidl, Daniela, Benjamin Hummel, and Elmar Juergens (2013). "Quality analysis of source code comments". In: *Proceedings of the 21st International Conference on Program Comprehension*, pp. 83–92.

Steinberger, Josef and Karel Jezek (2009). "Evaluation measures for text summarization". In: *Computing and Informatics* 28.2, p. 251.

Subramanian, Siddharth, Laura Inozemtseva, and Reid Holmes (2014). "Live API documentation". In: *Proceedings of the 36th International Conference on Software Engineering*, pp. 643–652.

Sun, Xiaobing, Qiang Geng, David Lo, Yucong Duan, Xiangyue Liu, and Bin Li (2016). "Code comment quality analysis and improvement recommendation: An automated approach". In: *International Journal of Software Engineering and Knowledge Engineering* 26.6, pp. 981–1000.

Takang, Armstrong A, Penny A Grubb, and Robert D Macredie (1996). "The effects of comments and identifier names on program comprehensibility: an experimental investigation". In: *J. Prog. Lang.* 4.3, pp. 143–167.

Tan, Lin, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou (2007). "/*iComment: Bugs or bad comments?*/". In: *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, pp. 145–158.

Tan, Shin Hwei, Darko Marinov, Lin Tan, and Gary T Leavens (2012). "@tComment: Testing Javadoc comments to detect comment-code inconsistencies". In: *Proceedings of the 5th IEEE International Conference on Software Testing, Verification and Validation*, pp. 260–269.

Tas, Oguzhan and Farzad Kiyani (2007). "A survey automatic text summarization". In: *PressAcademia Procedia* 5.1, pp. 205–213.

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

Tenny, Ted (1985). "Procedures and comments vs. the banker's algorithm". In: *ACM SIGCSE Bulletin* 17.3, pp. 44–53.

Tenny, Ted (1988). "Program readability: Procedures versus comments". In: *IEEE Transactions on Software Engineering* 14.9, pp. 1271–1279.

The Apache Software Foundation (2021a). *log4j*. URL: `https://github.com/apache/log4j/blob/da17f661144500538274925a8f87c27fd5a4717b/contribs/ThomasFenner/JDBCLogger.java#L67` (visited on 03/21/2021).

The Apache Software Foundation (2021b). *log4j*. URL: `https://github.com/apache/log4j/blob/da17f661144500538274925a8f87c27fd5a4717b/contribs/ThomasFenner/JDBCLogger.java#L415` (visited on 03/21/2021).

The Apache Software Foundation (2021c). *log4j*. URL: `https://github.com/apache/log4j/blob/da17f661144500538274925a8f87c27fd5a4717b/contribs/ThomasFenner/JDBCAppender.java#L190` (visited on 03/21/2021).

The Apache Software Foundation (2021d). *log4j*. URL: `https://github.com/apache/log4j/blob/af689d1e2517b8b4f83159e7e96bb8ad59b00bec/src/main/java/org/apache/log4j/Level.java#L61` (visited on 03/21/2021).

Torres-Moreno, Juan-Manuel (2014). *Automatic text summarization.* John Wiley & Sons.

Treude, Christoph, Fernando Figueira Filho, and Uirá Kulesza (2015). "Summarizing and measuring development activity". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 625–636.

Wang, Haoye, Xin Xia, David Lo, John Grundy, and Xinyu Wang (2021). "Automatic Solution Summarization for Crash Bugs". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE, pp. 1286–1297.

Wang, Ruyun, Hanwen Zhang, Guoliang Lu, Lei Lyu, and Chen Lyu (2020). "Fret: Functional reinforced transformer with BERT for code summarization". In: *IEEE Access* 8, pp. 135591–135604.

Widjanarko, Agus, Retno Kusumaningrum, and Bayu Surarso (2018). "Multi document summarization for the Indonesian language based on latent dirichlet allocation and significance sentence". In: *2018 International Conference on Information and Communications Technology (ICOIACT).* IEEE, pp. 520–524.

Wong, Edmund, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan (2015). "Dase: Document-assisted symbolic execution for improving automated software testing”. In: *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*. Vol. 1, pp. 620–631.

Ying, Annie TT and Martin P Robillard (2013). “Code fragment summarization”. In: *9th Joint Meeting on Foundations of Software Engineering (FSE)*, pp. 655–658.