Leveraging Informal Documentation to Automatically Summarize the Usage of Code Entities

by

Amirhossein NAGHSHZAN

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, JUNE 8, 2024

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGEMENTS

I would like to start by expressing my gratitude to my supervisor, Professor Latifa Guerrouj, for her patience, invaluable guidance, and support throughout this research project. Her expertise and encouragement helped me overcome my challenges while pursuing my Ph.D.

I am immensely thankful to Professor Sylvie Ratté, who took over as my supervisor and continued to provide dedicated guidance and expertise. Her mentorship played a pivotal role in shaping the development and success of my research.

I would like to extend my heartfelt appreciation to my beloved family, who have been my unwavering support system throughout this challenging journey. Their unconditional love, encouragement, and understanding have provided me with the strength and motivation to overcome obstacles and pursue my academic goals.

Furthermore, I cannot overlook the profound admiration and sympathy I hold for the people of my country, Iran. As I write this thesis from afar, I am acutely aware of the struggles and sacrifices they endure in their fight for fundamental rights and freedoms. Their resilience, courage and unwavering commitment to justice serve as a humbling reminder of the power of determination in the face of adversity. I stand in solidarity with them, hoping for a future where their voices are heard and their aspirations are realized.

Tirer parti de la documentation informelle pour automatiquement résumer l'utilisation des entités de code

Amirhossein NAGHSHZAN

RÉSUMÉ

La génération automatique de résumé de code source génère des informations synthétisées à propos des méthodes et des classes (objectif, mise en oeuvre, utilisation) qui facilite la compréhension de ces entités de code. Ces résumés contiennent des informations précieuses qui peuvent aider et guider les développeurs de logiciels dans des tâches telles que le développement, la maintenance et le remaniement du code source.

En général, les développeurs s'appuient sur la documentation officielle pour comprendre les entités de code et les méthodes API utilisées dans un projet. Cependant, les chercheurs ont identifié plusieurs cas où la documentation s'avère insuffisante ou inadéquate, ou encore présente des défauts dans la présentation de la structure complexe des méthodes. Ces éléments constituent des obstacles à l'apprentissage d'une API. Par conséquent, pour comprendre une API, les développeurs peuvent se référer à d'autres ressources telles que Stack Overflow, GitHub, etc. D'après les résultats de recherches récentes, la documentation non officielle est une source précieuse de connaissances pour générer des résumés de code. Cependant, la collecte de données à partir de ces sources exige du temps et des efforts supplémentaires de la part des développeurs afin de trouver et d'extraire les informations nécessaires, et ce d'autant plus que les données ne sont pas exactes et peuvent contenir des erreurs.

Notre recherche propose une nouvelle approche pour résumer les entités de code en tirant parti des algorithmes d'apprentissage profond et de la documentation informelle, c'est-à-dire Stack Overflow, pour produire des résumés de haute qualité pour les méthodes API. Notre approche prend le nom de la méthode API en entrée, génère un résumé en langage naturel en utilisant les discussions Stack Overflow et fournit des informations sur les problèmes fréquents et les solutions potentielles liées à l'API.

Dans la première partie de cette thèse, nous avons utilisé l'algorithme d'apprentissage automatique non supervisé TextRank pour générer des résumés extractifs des méthodes API. Pour évaluer la qualité des résumés générés, nous avons interrogé seize développeurs professionnels afin d'évaluer les résumés générés automatiquement et de les comparer à la documentation officielle d'Android. Nos résultats indiquent que bien que les développeurs utilisent généralement la documentation officielle, nos résumés générés sont des sources d'information précieuses, en particulier lorsqu'ils fournissent des détails d'implémentation. En outre, les développeurs ont convenu que nos résumés pourraient être utilisés comme une source complémentaire à la documentation officielle, aidant les développeurs de logiciels dans leurs tâches de développement et de maintenance.

Dans la deuxième partie, nous avons amélioré la qualité des résumés en appliquant un algorithme d'apprentissage profond (BART) pour générer des résumés de type abstractif. Nous avons également créé un oracle de résumés générés par l'homme pour évaluer nos résumés générés automatiquement à l'aide des métriques ROUGE et BLEU, qui sont fréquemment utilisées dans la synthèse de code et plus particulièrement dans l'évaluation de résumés automatiques. Enfin, nous avons évalué la qualité de nos résumés de code par rapport aux résumés produits dans la première partie de notre recherche. Nos résultats indiquent que les algorithmes d'apprentissage profond améliorent la qualité des résumés de code générés. Les améliorations apportées par notre approche dans la deuxième partie de notre recherche ont considérablement augmenté la précision de notre approche, atteignant une moyenne de 57 percent pour la précision, 66 percent pour le rappel et 61 percent pour la mesure F. De plus, la nouvelle approche est plus performante en termes de temps d'exécution, s'exécutant 4,4 fois plus rapidement que la première.

Dans la troisième et dernière partie de notre étude, nous avons appliqué l'algorithme BERTopic pour déterminer les sujets clés des discussions sur les API Android sur Stack Overflow, qui sert de source de documentation informelle importante. En outre, nous avons utilisé l'algorithme BERT pour identifier les problèmes fréquents et leurs solutions possibles, en créant des résumés brefs mais informatifs pour ces sujets. Pour évaluer l'efficacité de nos résultats, nous avons demandé à trente développeurs Android d'examiner ces résumés, en se concentrant sur leurs performances, leur cohérence et leur interopérabilité. Cela nous a permis d'obtenir des informations cruciales sur l'applicabilité de notre méthode dans le monde réel.

Mots-clés: Synthèse, Apprentissage automatique, Apprentissage profond, NLP, LLMs

**Leveraging Informal Documentation
to Automatically Summarize the Usage of Code Entities**

Amirhossein NAGHSHZAN

ABSTRACT

Source code summarization generates summarized information on the purpose, usage, and implementation of methods and classes to facilitate comprehension of these code entities. These summaries contain valuable information that can assist and guide software developers in tasks such as developing, maintaining, and refactoring source code.

Most often, developers rely on official documentation to understand code entities and API methods used in a project. However, researchers have identified insufficient or inadequate documentation examples and flaws with a method's complex structure as barriers to learning an API. Therefore, to understand an API, developers may refer to other resources such as Stack Overflow, GitHub, etc. According to recent research findings, unofficial documentation is a valuable source of knowledge for generating code summaries. However, collecting data from these sources is time-consuming and requires extra effort on the developers' side to find and extract the needed information, and sometimes, the data is inaccurate and may contain some flaws.

Our research proposes a novel approach for summarizing code entities by leveraging deep learning algorithms and informal documentation, i.e., Stack Overflow, to produce high-quality summaries for API methods. Our approach takes the API method's name as input, generates a natural language summary by utilizing Stack Overflow discussions, and provides information about frequent problems and potential solutions related to the API.

In the first section of our research, we used an unsupervised machine learning algorithm, i.e., TextRank, to generate extractive summaries for API methods. To assess the quality of our generated summaries, we surveyed sixteen professional developers to evaluate the automatically generated summaries and compare them with the official Android documentation. Our findings indicate that although developers generally use official documentation, our generated summaries are valuable sources of information, especially when providing implementation details. Moreover, developers agreed that our summaries could be complementary to official documentation, assisting software developers in their software development and maintenance tasks.

In the second research section, we improved the summaries' quality by applying a state-of-the-art deep learning algorithm (BART) to generate abstractive summaries. Furthermore, we created an oracle of human-generated summaries to evaluate our automatically generated summaries using ROUGE and BLEU metrics, which are frequently used in code summarization. Finally, we assessed the quality of our code summaries compared to summaries produced during the first section of our research. Our results indicate that deep learning algorithms enhance the quality of the generated code summaries. The improvements brought by our approach in the second section of this research significantly increased the accuracy of our summarization approach, reaching an average of 57 percent for Precision, 66 percent for Recall, and 61 percent for F-measure. Additionally, the new approach performs better in terms of execution time, running 4.4 times faster than the first one.

In the final part of our study, we applied the BERTopic algorithm to determine key topics from discussions about Android APIs on Stack Overflow, which serves as an essential informal documentation source. Additionally, we employed the BERT algorithm to pinpoint frequent issues and their possible solutions, creating brief yet informative summaries for these topics. To evaluate the effectiveness of our findings, we had thirty Android developers review these summaries, focusing on their performance, coherence, and interoperability. This provided us with crucial insights into the real-world applicability of our method.

Keywords: Summarization, Machine learning, Deep learning, NLP, LLMs

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ETS	École de Technologie Supérieure
API	Application Programming Interface
PR	PageRank
SO	Stack Overflow
NLP	Natural Language Processing
ML	Machine Learning
LLM	Large Language Model
BART	Bidirectional Auto-Regressive Transformer
BERT	Bidirectional Encoder Representations from Transformers
ROUGE	Recall-Oriented Understudy for Gisting Evaluation
BLEU	Bilingual Evaluation Understudy
NLTK	Natural Language Toolkit
CRD	Completely Randomized Design
UMAP	Uniform Manifold Approximation and Projection

INTRODUCTION

Researchers in the 1950s were captivated by the field of Automatic Text Summarization, particularly after the groundbreaking study conducted by Luhn Luhn (1958). Luhn's work introduced various techniques for extracting essential information from text, sparking interest in the broader field of code summarization. Automatic code summarization, a complex aspect of software engineering, focuses on generating concise and readable summaries that capture the functionality of a piece of code (Zhu & Pan, 2019).

Automatic source code summarization is a technique for generating a short, human-readable summary of the functionality of a piece of code by extracting its essential features. This can be done by removing redundant or unnecessary code, identifying and commenting on key sections, or creating high-level documentation. This summary can include information such as the purpose of the code, the input and output data types, and any key algorithms or methods used.

With the increasing complexity of software development and the need to work with code written by others, code summarization has become an essential tool for developers to increase their productivity and improve the overall quality of their code. It can help them quickly understand the functionality of unfamiliar code, making it easier to navigate and maintain large code bases. It can also be used to generate code documentation, which can make it easier for other developers to understand and use. Additionally, it can detect code duplicates and identify code clones, which can help developers maintain the software quality and reduce development time.

According to Zhu & Pan (2019); Le, Chen & Babar (2020) using code summaries is essential to achieving a deeper understanding and comprehension of a program's source code throughout the creation and maintenance of software. In addition, developers can spend less time comprehending the code with the help of code summaries, which is beneficial while trying to satisfy or change program requirements. Frequently, software developers are not fully aware of the functionalities and purposes of code components involved in the ongoing development and maintenance tasks of software. Consequently, they are required to go through the entire codebase and any accessible documentation. Take, for instance, a developer tasked with fixing a bug introduced by someone else. The initial step involves understanding the bug's characteristics, necessitating a review of all relevant bug reports and previous discussions to figure out how to reproduce it. Similarly, when a developer is assigned to implement a new feature, they must familiarize themselves with all the associated documentation to grasp its usage. Our approach is beneficial for quickly understanding the functionality of a codebase or finding solutions to specific coding issues.

Even though developers primarily rely on official documentation and source code as their primary sources of information, previous research has demonstrated that official documentation is not always the optimal way to extract information about Application Programming Interface (API) methods. (Ponzanelli, Bavota, Di Penta, Oliveto & Lanza, 2015; Uddin & Khomh, 2017). One of the main reasons is that official documentation may not always be up-to-date with the latest version of a technology or API, leading to confusion and frustration for developers who are trying to implement a solution using outdated information. Additionally, official documentation can often be high-level and lacking in detail, making it difficult for developers to understand the nuances of a technology or API and implement it correctly. Furthermore, some official documentation can be overly complex and difficult to understand for developers who are new to technology or API. This can lead to frustration and a lack of progress. In addition, it has been demonstrated by Parnin & Treude (2011) that web pages complete software documentation, and that some forms of media, such as blog postings containing tutorials and personal experiences, include valuable information. As a consequence, programmers may consult additional sources such as Stack Overflow and bug reports as well as email exchanges between developers in addition to the official documentation to obtain information regarding a code entity, its implementation, its usage, and any additional information that the official documentation may not provide (Parnin & Treude, 2011; Kavaler *et al.*, 2013; Aggarwal, Hindle & Stroulia, 2014). In our research, as shown in Figure 0.1, unofficial documentation refers to any of these different sources the software community offers.

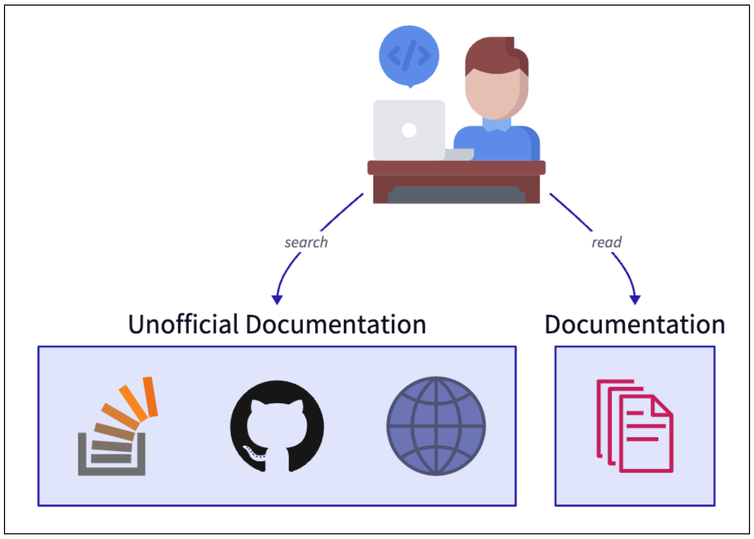


Figure 0.1 Exploring coding resources: A programmer navigates between official and unofficial documentation

To address the shortcomings of the official documentation, this research focuses on leveraging unofficial documentation sources using machine learning, deep learning, and Natural Language Processing (NLP) techniques to enhance the official documentation. One of these unofficial documentation sources is Stack Overflow, renowned for its wealth of informal documentation. It serves as a primary source of information in this investigation for two main reasons: 1) it has a great community of developers; 2) it proposes solutions to frequent problems (Barua, Thomas & Hassan, 2014). For these reasons, Stack Overflow has significantly impacted software development, providing invaluable informal documentation and practical coding insights Harirpoosh (2021).

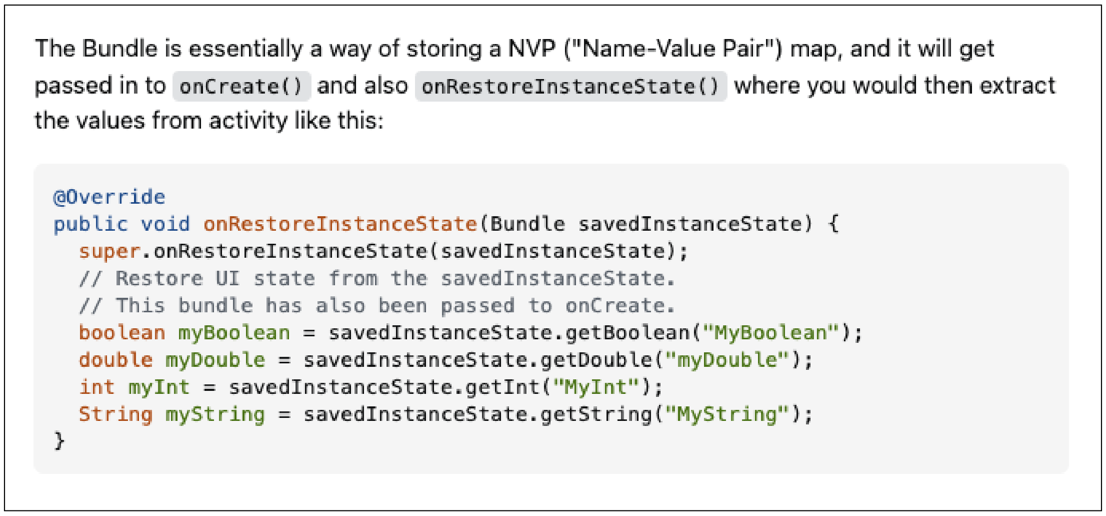
This research incorporates Stack Overflow's extensive resources and community discussions to enhance code summarization techniques, demonstrating the platform's vital role in the evolving landscape of software development learning and problem-solving. In this thesis, 'code summary' and 'code summarization' will refer to the production of a short description of API components. We aim to develop algorithms and techniques rooted in machine learning and deep learning to generate effective code summaries. These generated summaries will subsequently undergo evaluation and comparison to identify the most successful approaches.

Our research specifically focuses on generating summaries for API methods, highlighting their significance in software development. API methods serve as essential functions or operations provided by an API, enabling developers to interact with software systems or services (Subramanian, Inozemtseva & Holmes, 2014a). By summarizing these API methods, developers can gain a deeper understanding of their functionality and effectively utilize them in their applications. The generated summaries include a brief description of the API, the frequent problems that developers face while working with this API, and finally, the potential solutions to these problems.

In the context of Stack Overflow and similar platforms, code snippets are used to demonstrate specific concepts or solutions. There are generally two types of code snippets:

1. Code Entities: These are typically short, single-line references to specific functions, methods, or API calls,
2. Code Samples: These are more detailed, often multi-line blocks of code that provide a fuller example of how to use a function or implement a feature.

The goal of this kind of documentation is to assist developers in understanding how to work with certain APIs by providing concise and clear examples. An example of these two types of code snippets can be seen in Figure 0.2. `onRestoreInstanceState` and `onCreate` are the code entities. These are the names of the callback methods provided by the Android framework that the developer can override to handle the creation and restoration of an activity's state. The lower block of code in the image is an example of a code sample that shows how to use the `onRestoreInstanceState` method to retrieve values from a `Bundle`. The research referenced aims to extract such code entities from discussions and provide summaries to help developers understand and use APIs effectively.



The Bundle is essentially a way of storing a NVP ("Name-Value Pair") map, and it will get passed in to `onCreate()` and also `onRestoreInstanceState()` where you would then extract the values from activity like this:

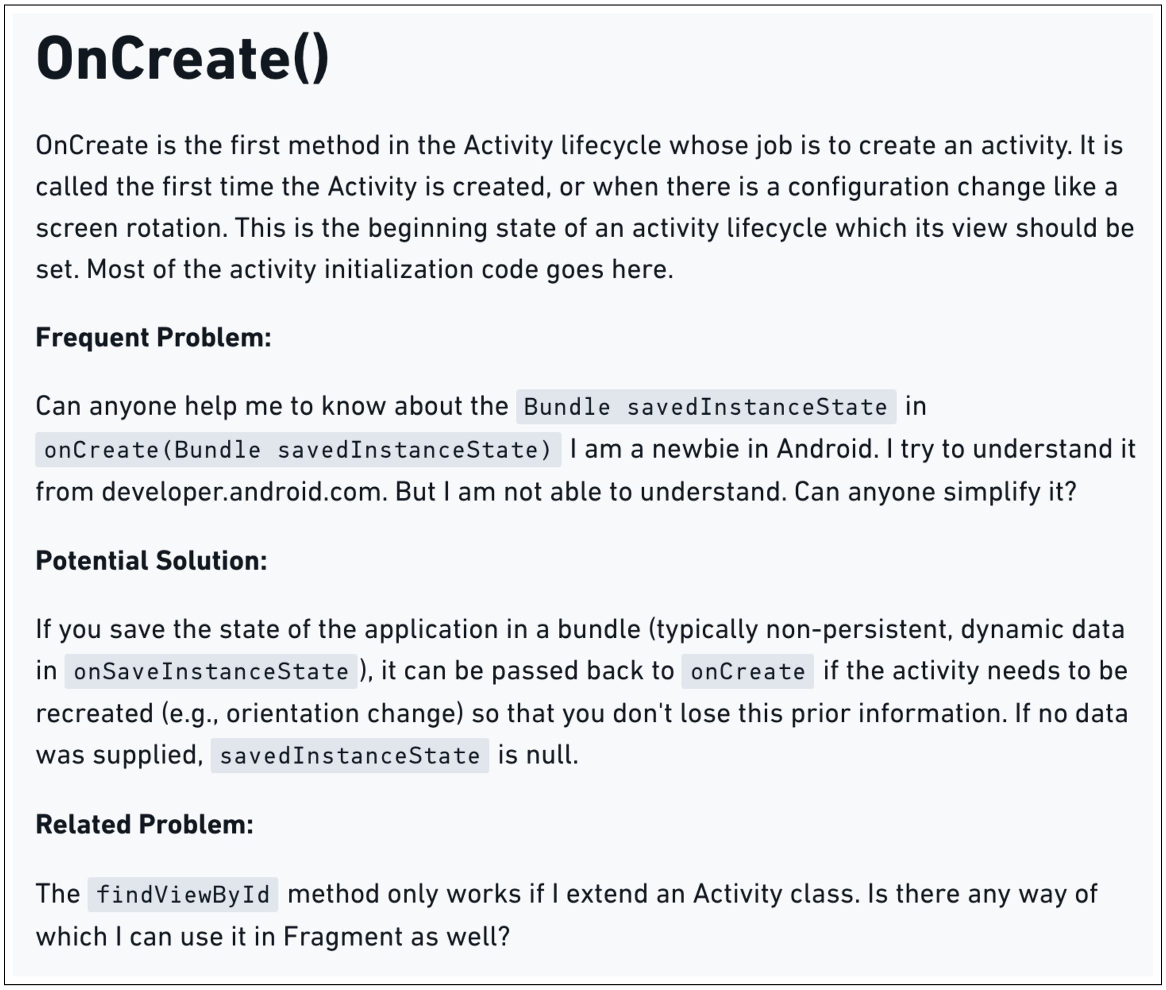
```
@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
  super.onRestoreInstanceState(savedInstanceState);
  // Restore UI state from the savedInstanceState.
  // This bundle has also been passed to onCreate.
  boolean myBoolean = savedInstanceState.getBoolean("MyBoolean");
  double myDouble = savedInstanceState.getDouble("myDouble");
  int myInt = savedInstanceState.getInt("MyInt");
  String myString = savedInstanceState.getString("MyString");
}
```

Figure 0.2 An example of code snippets extracted from Stack Overflow (Meier, 2008)

Figure 0.3 is a visual representation of the final output of our research. This figure showcases the application's interface, which assists users in understanding and resolving frequent issues with the Android API, specifically the `onCreate()` lifecycle method.

As illustrated in the figure, initially, the tool provides a succinct explanation of the `onCreate()` method. It details that this method is integral to the initiation of an Activity, called upon its creation or in response to a configuration change, such as screen rotation. This foundational explanation is essential, particularly for those new to the realm of Android development. Moving to the second aspect of assistance, the tool pinpoints a prevalent difficulty faced by novice developers: understanding the `Bundle savedInstanceState` within `onCreate()`. It perceives a need for a more intuitive explanation beyond what standard documentation offers. The text bridges this gap, offering a streamlined understanding for users who might find the official Android resources challenging. Next, the tool proposes a solution to the identified issue. It describes the importance of preserving the application's state using the `onSaveInstanceState()` method, allowing this state to be passed back to `onCreate()` if the Activity is ever recreated, thereby preventing data loss. It clarifies that in the absence of any saved state, the `savedInstanceState` parameter will be null. Finally, the tool anticipates and addresses related concerns, such as the applicability of the `findViewById()` method across different components of Android development, including both Activities and Fragments. This comprehensive approach to instruction and problem-solving underscores the value that our tool brings to Android developers, representing a significant scholarly contribution to the field, as delineated in this thesis.

Our long term goals revolve around advancing the field of summarization and providing valuable support to software developers. Firstly, we aim to develop advanced techniques and algorithms for automatic code summarization, allowing for a more efficient and accurate generation of summaries. This will help improve code comprehension and enhance productivity among developers. Secondly, we strive to enhance software developers' productivity by providing concise and informative code summaries. These summaries will enable developers to quickly understand the functionality of code, navigate large codebases, and make informed decisions during software development tasks. Lastly, we aim to improve the overall quality of software development by reducing code duplication and maintaining consistency through effective code summarization.



OnCreate()

OnCreate is the first method in the Activity lifecycle whose job is to create an activity. It is called the first time the Activity is created, or when there is a configuration change like a screen rotation. This is the beginning state of an activity lifecycle which its view should be set. Most of the activity initialization code goes here.

Frequent Problem:

Can anyone help me to know about the `Bundle savedInstanceState` in `onCreate(Bundle savedInstanceState)` I am a newbie in Android. I try to understand it from developer.android.com. But I am not able to understand. Can anyone simplify it?

Potential Solution:

If you save the state of the application in a bundle (typically non-persistent, dynamic data in `onSaveInstanceState`), it can be passed back to `onCreate` if the activity needs to be recreated (e.g., orientation change) so that you don't lose this prior information. If no data was supplied, `savedInstanceState` is null.

Related Problem:

The `findViewById` method only works if I extend an Activity class. Is there any way of which I can use it in Fragment as well?

Figure 0.3 Sample of framework output: Understanding Android's onCreate() Method

CHAPTER 1

LITERATURE REVIEW

Recently, much focus has been placed on automatic source code summarization. One reason is that as software systems become increasingly complex, it becomes more difficult for developers to understand and navigate large code bases. Automatic source code summarization can help make the code more understandable by providing a summary of its functionality and structure. Furthermore, as software development becomes more important in the industry and research, it has become an interesting topic for researchers to investigate and improve upon. Despite this, their primary focus was on providing a summary for a code block rather than an API method or a class, and very little research has leveraged unofficial documentation in the context of summarization tasks (Naghshzan, Guerrouj & Baysal, 2021; Guerrouj, Bourque & Rigby, 2015; Naghshzan, 2023b). In addition, deep learning techniques, specifically using Transformer models, have become increasingly popular in summarization and text generation (Vaswani *et al.*, 2017).

This chapter systematically explores the domain of automatic text summarization, commencing with "Summarization Techniques" where foundational methods and their evolution within software engineering are delineated. This section emphasizes the transition from extractive to abstractive summarization, highlighting their respective advantages and challenges, thereby establishing the groundwork for understanding the complexity and necessity of advanced summarization methods in software documentation. Following, "Summarization in Practice" showcases practical applications and the effectiveness of these techniques in real-world scenarios, emphasizing the significance of integrating both extractive and abstractive methods to enhance information retrieval and comprehension from informal sources like Stack Overflow. The final section, "Unofficial Documentation," delves into the pivotal role of informal documentation in software development, underscoring the value of leveraging user-generated content on platforms such as Stack Overflow for generating accurate and context-rich summaries. This progression from theoretical underpinnings to practical applications and the exploration of informal documentation sources logically scaffolds the thesis's aim to harness advanced summarization techniques for improving software documentation practices.

1.1 Summarization Techniques

Automatic summarization is essential in NLP and software engineering for creating concise summaries of large texts. This helps developers quickly understand complex codebases and documentation.Summarization techniques can broadly be categorized into extractive and abstractive methods, each with its own set of methodologies and applications.

1.1.1 Extractive vs. Abstractive Summarization

Summarization is done mainly in two forms in the software engineering domain: extractive and abstractive (Hsu *et al.*, 2018). Extractive summarization is a method of generating a summary by selecting and extracting the most important sentences or phrases from the original text. The summary is a subset of the original text, and the selected sentences or phrases are often presented in their original order (Jagan, Parthasarathi & Geetha, 2018). As Sun *et al.* (2022) mentioned, most recent code summarization techniques are based on extractive methods. On the other hand, abstractive summarization includes understanding the main concepts and relevant information of the main text and then expressing that information in a short and precise format (Jagan *et al.*, 2018). Figure 1.1 presents a side-by-side comparison of two text summarization methods: extractive and abstractive summarization, each accompanied by color-coded examples. In the extractive summarization (a), key phrases from the source text are highlighted in yellow and directly extracted to form a choppy and less cohesive summary. It shows "Peter and Elizabeth" and "attend party city" highlighted in yellow from the first sentence, and "Elizabeth rushed hospital" highlighted in the second sentence, which are then used verbatim to construct the summary.

The abstractive summarization (b) does not rely on direct extraction but instead reformulates the essence of the text into a new, succinct narrative, demonstrated by a coherent and smoothly written summary. Here, the source text is the same, but the summary, "Elizabeth was hospitalized after attending a party with Peter," is a reinterpretation of the original content into a novel form.

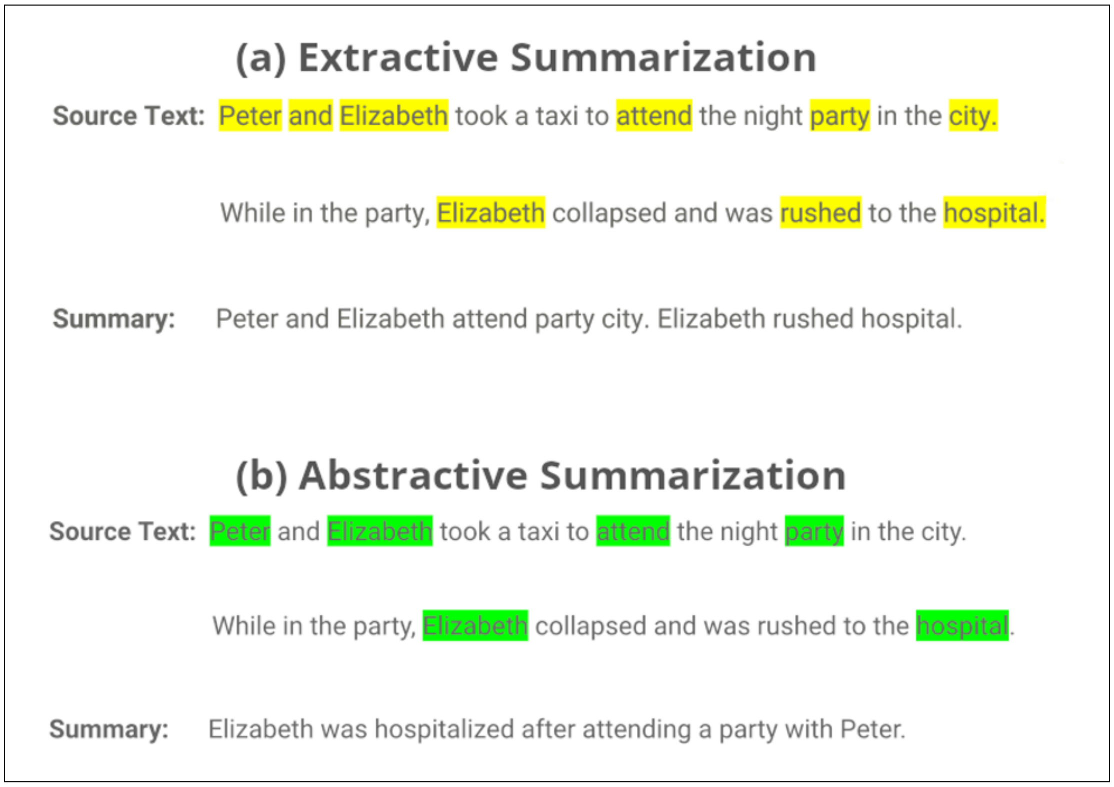

(a) Extractive Summarization
Source Text: Peter and Elizabeth took a taxi to attend the night party in the city.
While in the party, Elizabeth collapsed and was rushed to the hospital.
Summary: Peter and Elizabeth attend party city. Elizabeth rushed hospital.
(b) Abstractive Summarization
Source Text: Peter and Elizabeth took a taxi to attend the night party in the city.
While in the party, Elizabeth collapsed and was rushed to the hospital.
Summary: Elizabeth was hospitalized after attending a party with Peter.

Figure 1.1 Extractive vs Abstractive summarization
Taken from Opidi (2019, p. 01)

Each method has its advantages and disadvantages. The main advantage of extractive summarization is that it is more accurate and reliable, as it directly uses important information from the original text. (Hsu *et al.*, 2018). Also, it is computationally less complex and faster to execute. Therefore, it can summarize longer texts and documents by selecting the most important information. However, it has four main disadvantages (Hou, Hu & Bei, 2017; Hsu *et al.*, 2018; El-Kassas, Salama, Rafea & Mohamed, 2021):

1. It can generate a summary that is disconnected or difficult to understand if important connecting phrases or sentences are not included;

2. The important and relevant information is usually spread throughout the document, and the extractive techniques cannot shortly combine them. Therefore, the length of the generated summary is longer than expected;
3. Since the sentences are selected from different parts of a document, the pronouns often tend to lose their references and create confusion in tracing the meaning;
4. It may not be able to summarize the text effectively if the text is poorly written or structured.

On the other hand, abstractive summarization can capture the overall meaning and main ideas of the text, and can also paraphrase and rephrase the information in a more condensed form (El-Kassas *et al.*, 2021; Hsu *et al.*, 2018). Therefore it can generate a summary that is more coherent and readable than an extractive summary. However, the drawback is that constructing a high-quality abstractive summary can be computationally more complex and thus slower to execute compared to extractive summarization. Also, it may not be able to summarize the text effectively if the model lacks an understanding of its content. (Wang, Zhao, Li, Ge & Tang, 2017; Hou *et al.*, 2017; Hsu *et al.*, 2018).

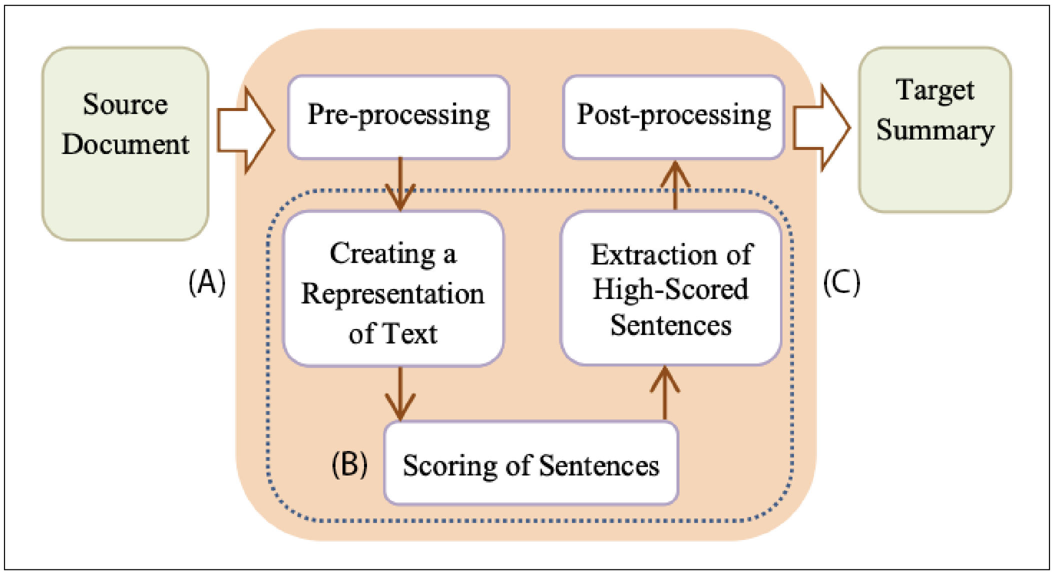


Figure 1.2 The architecture of an extractive approach
Taken from El-Kassas et al. (2021, p. 08)

Figure 1.2 shows the architecture of an extractive summarization approach. The core processes involve creating a representation of text (Box A), scoring sentences to assign a relevance score based on extracted features (Box B), and ultimately selecting sentences that have received the highest scores for extraction (Box C).

On the other hand, Figure 1.3 shows the architecture of an abstractive approach. The main steps are creating an intermediate representation (Box A) and summary generation (Box B). In the first step, the text is encoded into a numerical representation that can be used by the neural network. This can be done using a variety of techniques such as word embedding, sentence encoders, or even more advanced models like transformer models. After that, for the summary generation step, the encoded representation of the text is used as input to a decoder network, which generates the summary. The decoder network can be trained to generate new words and phrases that capture the main ideas of the original text.

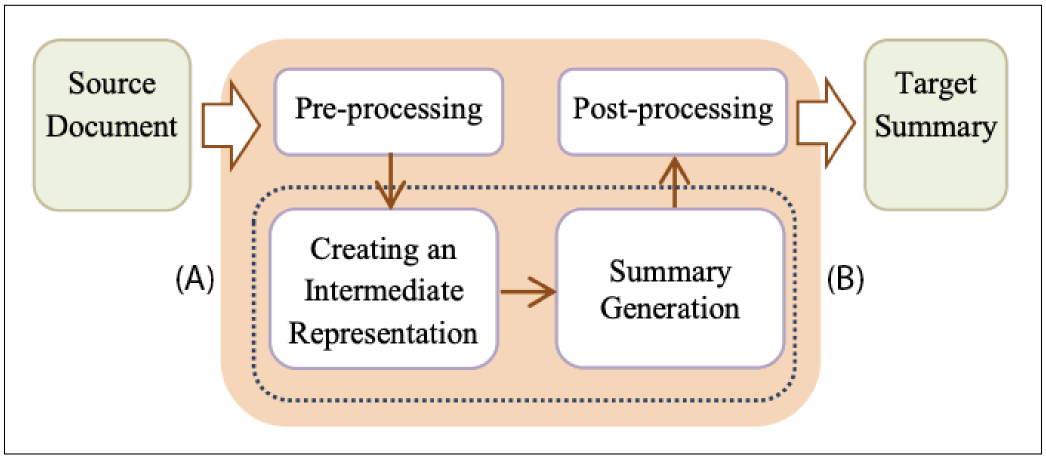


Figure 1.3 The architecture of an abstractive approach
Taken from El-Kassas et al. (2021, p. 13)

1.1.2 TextRank Algorithm

In our study, we explore both extractive and abstractive summarization methods and conduct a comparative analysis to determine their suitability within our research context. For extractive summarization, we employ the TextRank algorithm (Mihalcea & Tarau, 2004), while for abstractive summarization, we utilize the BART algorithm (Lewis *et al.*, 2020). We will now elaborate on each method and provide the rationale behind our selection.

TextRank builds on the principles of PageRank to rank sentences or phrases within a text for extractive summarization. By constructing a graph where nodes represent sentences and edges signify the similarity between them, TextRank applies the iterative ranking process used in PageRank to identify the most significant sentences. This adaptation leverages the inherent strengths of PageRank to effectively highlight key information in a given text (Mihalcea & Tarau, 2004).

PageRank, developed by Google, is an algorithm that determines the order of websites in search results based on their relevance (Page, Brin, Motwani & Winograd, 1998), and it was named after Larry Page, one of Google's founders. The relevance of a web page is estimated by calculating the quantity and quality of incoming links. The underlying assumption is that more important websites are likely to receive a higher number of links from other websites. Although initially designed to evaluate website significance, PageRank has broader potential applications (Gleich, 2015).

The PageRank of a page can be computed using the following formula:

$$PR_i = \frac{1-d}{n} + d \sum_{j \in \{1,\dots,n\}} \frac{PR_i}{c_j} \tag{1.1}$$

where PR(i) is the PageRank of page i, n is the total number of pages, d is the damping factor, cj is the number of outbound links on page j, and the summation is over all pages j link to page i.

Figure 1.4 illustrates the propagation of rank between pages, where each box with a number symbolizes a web page, with the number indicating the page's current PageRank. The red arrows represent outgoing links from a page, while the blue lines indicate incoming links to a page. The PageRank value of a page is divided among its outgoing links and passed on to the pages it points to. This process occurs iteratively until the PageRank values reach a steady state, where they change very little with further iterations. The steady-state values are used to rank pages, with higher values suggesting greater importance or authority. The image captures a single state in this iterative process, with some pages having accumulated more PageRank value than others. Here the top page has a rank of 100, which is evenly distributed across its two links. Similarly, the bottom page has a rank of 9, divided among its three links. After computing scores for all pages, they are ranked based on their scores, with higher scores indicating greater importance.

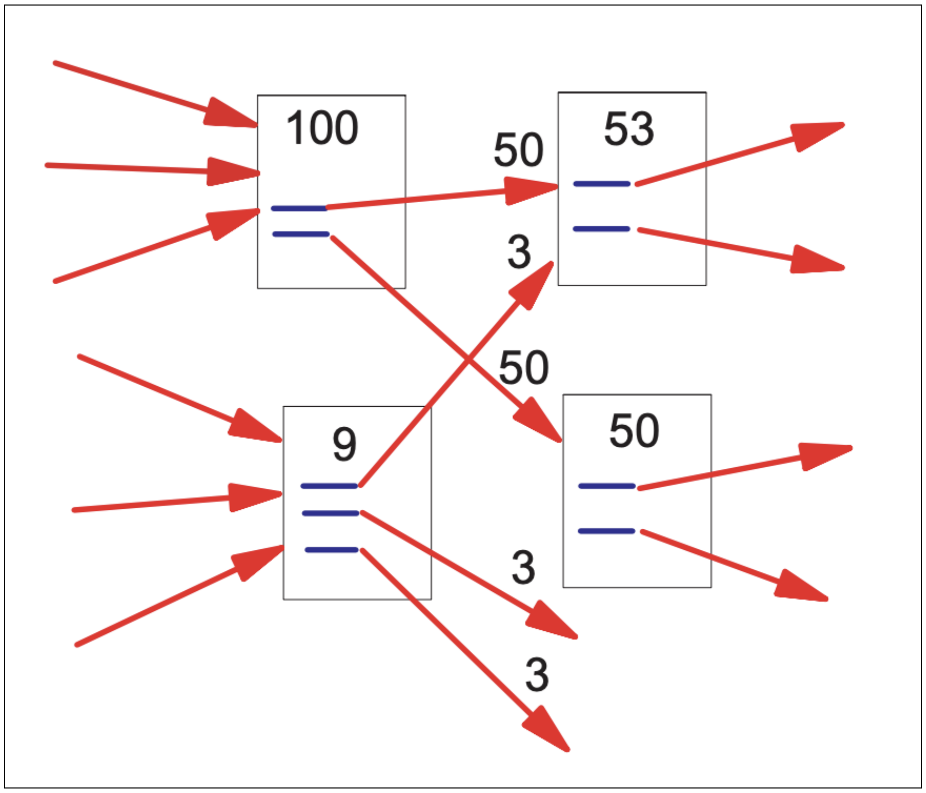


Figure 1.4 Simplied PageRank Calculation
Taken from Page et al. (1998, p. 04)

Inspired by Google's PageRank algorithm, TextRank aims to identify significant terms within a dataset (Uddin & Khomh, 2017). TextRank is an extractive summarization technique proposed by Mihalcea & Tarau (2004) that uses a graph-based approach. TextRank, as indicated by Mihalcea & Tarau (2004), has shown the ability to produce high-quality text summaries. This approach relies on two key processes: cosine similarity and the PageRank algorithm. By assessing the degree of similarity between sentences, points are assigned to them, leading to the creation of a weighted graph. Subsequently, sentences with higher weights are extracted to form the final summary (Huang *et al.*, 2020). For example, consider three sentences: S1 (“Machine learning automates model building.”), S2 (“It’s a branch of AI where systems learn from data.”), and S3 (“These systems identify patterns with minimal human intervention.”). We create a graph with these sentences as nodes and calculate the similarity between each pair of sentences. If S1 and S2 have a similarity of 0.8, S1 and S3 have 0.5, and S2 and S3 have 0.6, these values become edge weights. TextRank then iteratively updates the rank of each sentence based on the ranks of neighboring sentences, weighted by similarity. Ultimately, sentences S2 and S3, having higher ranks, would be selected to form the summary, ensuring that the most relevant information is retained. Figure 1.5 shows an overview of this algorithm.

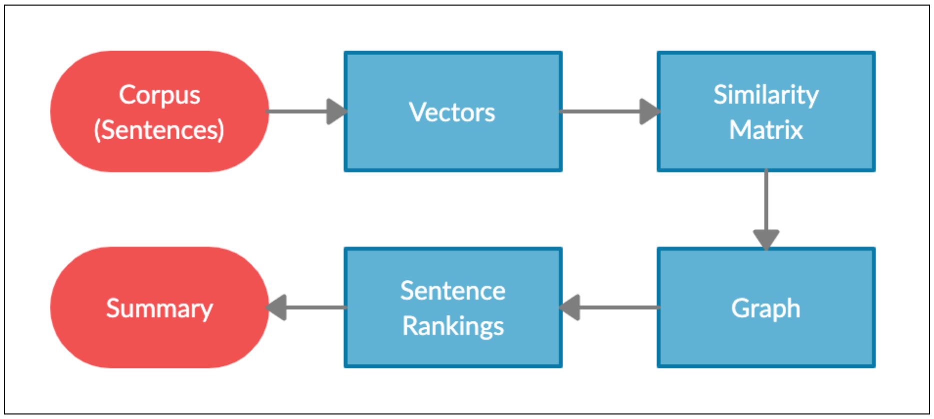


Figure 1.5 Schematic overview of TextRank
Taken from Naghshzan et al. (2021, p. 05)

The method can also be used by selecting individual words rather than whole phrases. In a manner that is analogous to the operation of the PageRank algorithm, the significance of a word is established not only by the number of votes it receives but also by the significance of the other words that vote for it. If one word is linked to another by an edge, it counts as a vote for the second word. The significance of the first word determines the importance of this vote. (Li & Zhao, 2016). TextRank is effective because, as stated in the original work (Mihalcea & Tarau, 2004), it takes into account information that has been recursively obtained from the entirety of the text (*i.e.*, graph), as opposed to merely taking into account the immediate context of a text unit (*i.e.*, vertex).

There are two main differences between Pagerank and Textrank:

- **Graph Construction:** PageRank operates on a graph constructed from the web, where nodes are web pages, and edges are hyperlinks. TextRank, however, operates on graphs constructed from text units (like sentences or words) within a single document or a collection of documents, with edges representing relationships like adjacency or semantic similarity;
- **Edge Weighting:** In PageRank, edges are typically unweighted, with the importance of a page being divided equally among its outbound links. In TextRank, edges can be weighted to reflect the strength of the connection between nodes, such as the frequency of co-occurrence of two words within a certain distance.

Both algorithms leverage the idea of "voting" or "endorsement," with the key difference lying in their application domain and the nature of the graphs they operate on. TextRank adapts the PageRank algorithm for text, making it suitable for NLP tasks by considering the relationships between text units rather than hyperlinks.

1.1.3 BART Algorithm

BART, short for *Bidirectional Auto-Regressive Transformer*, was created by the Facebook Artificial Intelligence (AI) research team in the field of natural language processing (NLP). This transformer model was designed to generate abstractive summaries (Lewis *et al.*, 2020). It was developed based on two well-known algorithms, Google's BERT (Bidirectional Encoder Representations from Transformers) and OpenAI's GPT algorithm. BERT is a pre-trained deep learning model developed by Google for NLP tasks, such as question-answering and sentiment analysis. It is trained on a massive amount of text data, using a technique called unsupervised learning. This allows the model to learn the underlying statistical structure of the language and generalize to new examples (Devlin, Chang, Lee & Toutanova, 2019). Furthermore, text generation models such as OpenAI's GPT have been pre-trained to anticipate the next token based on the sequence of tokens that came before it. This pre-training goal aims to build models suitable for text production.

BART is an example of a transformer encoder-decoder (seq2seq) model, combining a bidirectional encoder with an autoregressive. It is a language model that may be utilized for language production, and it has been employed for multi-document summary jobs (Ma, Zhang, Guo, Wang & Sheng, 2022). Like BERT, BART is also trained on a massive amount of text data, using unsupervised learning. However, while BERT is trained on a task of masked language modeling (MLM) where some words are replaced with a [MASK] token and the model must predict the original word (Devlin *et al.*, 2019), BART is trained on a task of denoising autoencoder where some words are replaced with a random token and the model must predict the original words (Lewis *et al.*, 2020).

BART is trained to generate text, which makes it a powerful model for tasks such as text summarization, text completion, and text generation. It can also be fine-tuned on specific tasks with much fewer data and computational resources than training from scratch. BART has been shown to be particularly effective for text summarization. It can extract important information from the original text and generate a condensed version that captures the main ideas. It has been also shown to be effective in other tasks like machine translation and text-to-speech synthesis.

In this algorithm, much like Google's BERT, an encoder-decoder design, the original Transformers is a sequence-to-sequence paradigm. Both the input and output of the model are sequences, such as text. The model's encoder learns a high-dimensional representation of the input, which is then mapped to the output by the model's decoder, referred to as (Vaswani *et al.*, 2017). This method is suitable for classification issues in which it is possible to generate a forecast based on information gleaned from the whole sequence.

Figure 1.6 illustrates the architecture of BART, a sequence-to-sequence model designed for natural language processing tasks. BART combines the strengths of bidirectional and autoregressive models, making it particularly effective for text generation and comprehension tasks.

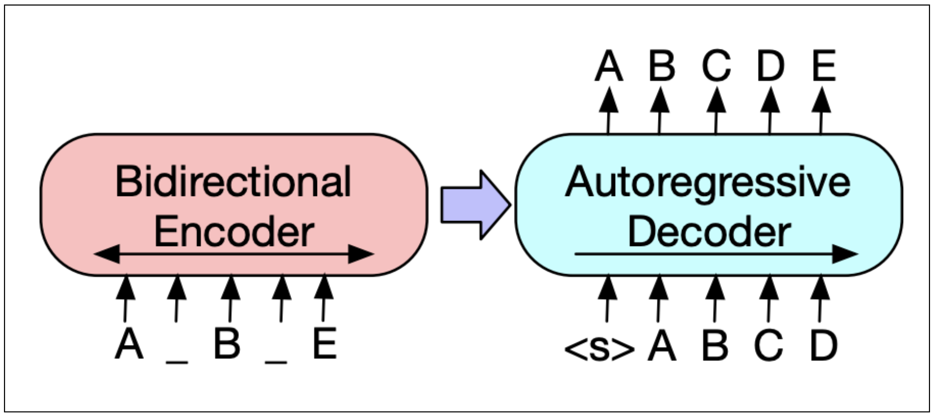


Figure 1.6 Schematic overview of BART
Taken from Lewis et al. (2020, p. 01)

- **Bidirectional Encoder:** On the left, the bidirectional encoder processes the corrupted input document. In this example, the input sequence "A _ B _ E" has missing or corrupted parts, represented by underscores. The bidirectional encoder reads the entire sequence, both forward and backward, to capture context from both directions. This allows the encoder to build a comprehensive understanding of the input, considering the surrounding words to fill in the gaps or correct errors.
- **Autoregressive Decoder:** On the right, the autoregressive decoder generates the original content sequentially. Starting with a special start token <s>, the decoder predicts each subsequent word based on the previous words it has generated. It continues this process until the entire sequence "A B C D E" is reconstructed. The decoder's autoregressive nature means it generates words one by one, ensuring each word depends on the previously generated words, which helps in maintaining coherence and context.
- **Process Flow:** The flow between the encoder and decoder involves transforming the corrupted input into a more complete and coherent output. The encoder's bidirectional approach ensures a thorough understanding of the context, while the decoder's autoregressive mechanism ensures the output sequence is generated in a logically consistent order.

1.2 Summarization in Practice: Source Code Summarization

Alambo *et al.* (2020) suggested an unsupervised multi-document summarization system. This structure is composed of a summarization of both extractive and abstractive types. They began by clustering the data, and then applied an improved multi-sentence compression technique for extractive summarization. This technique allowed them to produce summaries that were both informative and pertinent. For abstractive summarization, they used both the GPT and the BART algorithms. The ROUGE metric (check Chapter 5.3.2) was utilized to analyze the extractive summaries. At the same time, the study's authors used five human evaluation metrics to evaluate the abstractive summaries. These criteria included entailment, coherence, conciseness, readability, and grammar (Alambo *et al.*, 2020). In evaluating their methodology, the DUC-2004 benchmark dataset and scientific articles from Microsoft Academic Graph (MAG) were utilized. Their study has demonstrated that the proposed approach is comparable to state-of-the-art extractive summarization, and performs even better for abstractive summarization. Similarly to this study, we have leveraged unofficial documentation, *i.e.*, Stack Overflow, when generating abstractive summaries. However, unlike this research, we have applied BART, a Transformer model, and TextRank, an unsupervised algorithm, as components of our suggested approaches.

Pang, Lelkes, Tran & Yu (2021) made an effort to create abstract summaries for each grouping of articles provided. The researchers evaluated their strategy using various methods, including the PEGASUS model, automatic metrics such as ROUGE, and human review. The examination revealed that the generated summaries had a higher level of article summary and cluster-summary entailment than the other baselines. A limitation of this work is that it focuses primarily on clustering articles and may not handle fine-grained details necessary for summarizing specific code-related content. Our research addresses this limitation by specifically targeting API methods and using a combination of BART and TextRank algorithms to generate detailed and accurate summaries that are directly applicable to software development tasks.

The real-time quality assurance and multi-document summarization system known as CAiRE-COVID was developed by Su *et al.* (2020). To supply information pertinent to the query, which is a written question in this scenario, the system suggests abstractive and extractive summarization approaches centered on the inquiry. To accomplish the extractive summarization, the authors first built a representation at the sentence level and then scored the sentences utilizing the cosine similarity function. The BART method is used for the abstractive summarization component of the process, and the system was assessed using the CovidQA dataset (Tang *et al.*, 2020). ROUGE was chosen as the evaluation metric to make a performance comparison by the authors. Compared to the other baselines, the results showed that the ROUGE score had improved, particularly in extractive and abstractive summarization. In our research, we also used BART for abstractive summarization, and ROUGE as a metric of evaluation. This study focus on medical data, which may not generalize well to the domain of software engineering. In our research, we overcome this limitation by focusing specifically on code summarization and using a dataset composed of Android posts from Stack Overflow, ensuring that our approach is tailored to the needs of software developers.

Wu & Hu (2018) leveraged a Reinforced Neural Extractive Summarization (RNES) model to compile a coherent and informative summary from a single piece of writing. The suggested model can strike a balance between the coherence and relevance of sentences and achieve state-of-the-art performance on well-known datasets such as CNN and DailyMail. While we both aim for summarization, in this research, we have employed also an abstractive approach rather than relying only on an extractive one, and the results were measured using ROUGE and BLEU metrics.

Song, Huang & Ruan (2019) proposed a novel approach to text summarization using a combination of Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) algorithms. This innovative framework is designed to delve into the nuances of language beyond mere sentences, focusing on "semantic phrases" - specific groupings of words that encapsulate richer meanings. By honing in on these semantic phrases, the framework offers a more detailed analysis of text, enabling the generation of new sentences that are not only syntactically sound but also semantically rich. This method was tested on datasets from notable sources such as CNN and DailyMail, where it demonstrated superior performance in capturing both the semantic and syntactic essence of texts compared to other leading models at the time. A limitation of this approach is that it may not be directly applicable to the technical language and specific requirements of code summarization. Our research overcomes this limitation by applying transformer-based models like BART, which are specifically fine-tuned for technical content and code-related discussions, ensuring that the generated summaries are relevant and useful for developers.

In their research, Kim, Cheong & Lee (2019) embarked on an ambitious project to forecast a movie's success using its textual summary. They experimented with three advanced deep learning models: Embeddings from Language Models (ELMo), a one-dimensional convolutional neural network (1D CNN), and an LSTM network—a type of recurrent neural network adept at capturing long-term dependencies in sequential data like text. The comparison of these models revealed a superior ability to outperform the established baseline for classifying the majority class, with the ELMo model emerging as the most effective among the trio. However, diverging from their approach, our study leverages BART, an algorithm renowned for its proficiency in summarization tasks, suggesting a potentially more effective method for predicting movie success. (Lewis *et al.*, 2020).

Natural Language Processing has been used to generate information to summarize Java methods, as cited in Sridhara, Hill, Muppaneni, Pollock & Vijay-Shanker (2010). As the first step in their methodology, they pre-processed the source code by running it through a Software Word Usage Model (SWUM). This model takes into account both structural and linguistic clues. After that, a format-based model is used to construct the summary in a natural language format. Thirteen individuals performed a manual evaluation of the methodology's Precision, content suitability, and concision. As a result of their investigation, it was shown that the comments that are created for Java methods based on the source code are accurate and do not omit any crucial pieces of information throughout the creation process. However, a limitation of this research is its reliance on manual evaluation, which can be subjective and may not scale well with larger datasets. In contrast, we generated our summaries using the BART algorithm for abstractive summarization and the TextRank algorithm for extractive summarization. Our approach leverages quantitative evaluation metrics, i.e., ROUGE and BLEU, which provide objective measures of summary quality. Additionally, we conduct statistical tests along with surveys involving a larger number of software developers to ensure the robustness and generalizability of our findings (Naghshzan *et al.*, 2021).

McBurney, Liu & McMillan (2016) in their study aimed to create a list of features from Java source code documentation. They adopted four different approaches to achieve this goal. They described the features at the system-level granularity of software tools using Natural Language Sentences (NLS) to provide a clear feature. They divided their summarization approach into two main categories: feature list tools and textual analysis tools. In feature list tools, they used the Latent Dirichlet Allocation (LDA) approach to generate a list of topics from the prepared sentences list. In textual analysis tools, they used TextRank and TLDR tools to generate a feature list for a Java source code. They conducted two surveys to evaluate the quality and readability of the generated feature list. The evaluations showed that although LDA can find topics that assist programmers in better understanding projects, it is not always possible to characterize these subjects as natural language sentences retrieved from JavaDocs. A limitation of this study is the use of LDA, which, while effective for topic modeling, may not generate coherent and cohesive natural language summaries due to its probabilistic nature and reliance on the bag-of-words model, which disregards word order and context. In contrast, instead of using LDA, our research adopts a more recent approach, known as transformers, for summarization tasks. By leveraging the BART algorithm for abstractive summarization and TextRank for extractive summarization, we generate summaries with higher coherence and cohesion. Furthermore, our method is evaluated using quantitative metrics like ROUGE and BLEU, and involves surveys with software developers to ensure both the quality and applicability of the summaries. This approach allows us to produce more natural and contextually accurate summaries compared to traditional methods like LDA.

Alon, Zilberstein, Levy & Yahav (2019) proposed a method for code summarization using deep learning techniques. The authors aimed to generate a vector representation of a piece of code that can be used to compare and summarize code with other pieces of code. The authors proposed an approach called *Code2Vec* which is based on the Paragraph Vector algorithm. The algorithm is trained on a large corpus of code and learns to generate a dense vector representation of a code snippet. These vectors are called *code embeddings* and they capture the semantic meaning of the code snippet. The authors also proposed a method called *Code2Vec similarity search* which is used to compare two code snippets. It is based on the cosine similarity between the code embeddings of the two snippets. This method can be used to find the most similar code snippets to a given code snippet. The authors evaluated the Code2Vec approach on a dataset of Java code and compared it with previous state-of-the-art methods. The results showed that the Code2Vec approach outperforms previous methods in terms of Precision and Recall when used for similarity search and clustering. In our study, instead of using vector representation, we use advanced transformer models like BART and unsupervised learning algorithms like TextRank to generate a multi-line summary. Furthermore, instead of generating a summary for a code block we try to generate a summary for an API by analyzing the conversations and discussions related to that API.

LeClair, Haque, Wu & McMillan (2020), proposed an approach to code summarization that utilizes a Graph Neural Network (GNN) to predict summary from the graph representation of code. The authors first present a method to create a graph representation of code, where each node in the graph represents a code element such as a class, method, or variable, and edges represent relationships between these elements such as inheritance or method calls. They use this graph representation as input for a graph neural network (GNN) which is trained to generate a summary of the code. The GNN is trained to predict the summary from the code graph representation. The authors evaluate their method on a dataset of Java code, comparing it to several baselines including a standard encoder-decoder model and a graph-based method. They use several metrics to evaluate the quality of the generated summaries, such as ROUGE and METEOR, and also measure the coverage of the generated summaries. Results show that the GNN-based method outperforms the baselines in terms of both summary quality and code coverage. Unlike this research, instead of GNN, we have used transformer models since they have demonstrated their efficiency in completing summarization tasks in comparison to neural networks (Lewis *et al.*, 2020).

Rigby & Robillard (2013) have suggested an approach that uses a parser to identify code entities in documents with an average Precision and Recall at or above 90%. The author's method focuses on identifying code elements, such as variables, classes, and methods, that are mentioned in informal documentation, such as code comments, and using them to generate a summary of the code. The method uses natural language processing techniques, such as Named Entity Recognition (NER), to extract code elements from comments and relationships between them, such as usage or inheritance. The method is evaluated on a dataset of Java code, and the performance of the method is measured using standard metrics such as Precision, Recall, and F1-score. The results show that the method can extract code elements from comments with high accuracy and that the generated summaries are useful for developers trying to understand or maintain the code. Compared to our study, we focus on surrounding context and natural language sentences to generate a summary instead of analyzing code blocks.

Stapleton *et al.* (2020) discussed the importance of automatic code summarization for addressing the lack of well-documented code, which can influence comprehension for both new and experienced developers. The authors conducted a human study involving 45 participants, including undergraduate and graduate students, as well as professional developers. The study involved examining Java methods documented with either human-written or machine-generated summaries and answering comprehension questions about each. The study found that human-written summaries significantly improved developer comprehension compared to machine-generated summaries. However, there was no significant correlation between developer perception of summary quality and their comprehension. Additionally, the study found that BLEU and ROUGE scores were both significantly uncorrelated with comprehension. Based on these findings, the authors suggest the need for revised metrics to assess and guide automatic summarization techniques. The article concludes that while automatic code summarization can be beneficial for improving developer comprehension, it is crucial to consider the effectiveness of the generated summaries on human comprehension when evaluating the quality of such techniques.

Fowkes *et al.* (2017) explored the possibility of using summarization methods to assist developers in reading and navigating source code. The authors introduce a new problem called the "autofolding problem", which aims to automatically create a code summary by folding less informative code regions. They note that while modern code editors do provide code folding options, these options are often impractical to use as they require manual folding decisions or simple rules. To address this issue, the authors propose a novel solution by formulating the problem as a sequence of abstract syntax tree (AST) folding decisions, which allows them to leverage a scoped topic model for code tokens. This approach involves identifying which tokens are most relevant to their enclosing files and projects. They evaluate their approach on a set of popular open-source projects and demonstrate that their summarizer outperforms simpler baselines, resulting in a 28% error reduction. They also conduct a case study that shows experienced developers prefer their summarizer over other methods used in modern integrated development environments (IDEs). The authors conclude that while their approach is not without limitations, it has the potential to be a valuable tool for aiding program comprehension and turning code folding into a more usable feature.

Liang & Zhu (2018) proposed a new framework for automatically generating descriptive comments for source code blocks that outperform previous methods. The framework consists of two novel neural network models, Code-RNN and Code-GRU, that extract features from the source code and generate text descriptions. The Code-RNN model constructs a parse tree of the source code and extracts the topic or function of the code, while the Code-GRU model generates descriptive comments for the code using the vector representation generated by Code-RNN. The authors evaluated the framework on ten different source code repositories and reported that it achieved significantly higher accuracy than other learning-based approaches, including a sequence-to-sequence model. The accuracy was measured using the Rouge-2 value, which evaluates the similarity between the generated comments and human-written comments. The results show that the proposed framework can generate accurate and informative comments for source code blocks without relying on fixed templates. This could have significant benefits for software developers and researchers, as it can assist with code understanding, maintenance, and documentation.

Liu *et al.* (2019) introduced RoBERTa, an optimized version of BERT for pretraining natural language processing models. By modifying the pretraining process, including extending training time, increasing batch sizes, and removing the next sentence prediction objective, RoBERTa significantly outperforms BERT and other state-of-the-art models on a wide range of benchmark tasks such as GLUE (General Language Understanding Evaluation), RACE (Reading Comprehension Dataset), and SQuADn (Stanford Question Answering Dataset). The study highlights the importance of hyperparameter tuning, training duration, and data size in achieving superior model performance. Additionally, RoBERTa's success suggests that the masked language modeling objective remains a powerful tool for pretraining models in natural language processing tasks.

1.3 Summarization in Practice: Unofficial Documentation

S. Rastkar & Murray (2014) presented a method for automatically summarizing bug reports using natural language processing techniques. The authors use a three-step approach to summarize the bug reports, first, they extract the problem description, then they extract the affected component, and finally, they extract the suggested solution. They use natural language processing techniques to extract the information from the bug report and generate a summary that includes the problem description, the affected component, and the suggested solution. The authors evaluate the performance of the proposed method on a dataset of bug reports from the Eclipse and Mozilla projects. They compared the summaries generated by the method to the summaries provided by human evaluators and found that the method was able to accurately summarize the bug reports. The authors also found that the summaries generated by the method were more consistent than the summaries provided by human evaluators.

A. Bacchelli & Lanza (2010) proposed a method for extracting source code from e-mails based on natural language processing and machine learning techniques. The method uses a natural language parser to analyze the text of an e-mail and identify portions of the text that are likely to contain source code. These portions are then passed through a machine learning classifier that is trained to distinguish source code from non-source code text. The authors evaluate the performance of the proposed method on a dataset of e-mails from the AspectJ and JBoss projects. They found that the method was able to accurately identify source code within e-mails, with a Precision of 96% and a Recall of 85%. They also found that the method was able to extract source code from e-mails that were in a variety of programming languages, including Java, C, and C++. The main difference between this work with our study is that we focus not only on extracting codes but also on generating summaries for them based on the surrounding context.

Uddin & Khomh (2017) attempted to summarize the opinions and reviews of Stack Overflow users regarding an API. Users can benefit from learning about each API's limitations and other users' opinions regarding it. We are interested in summarizing API methods in terms of their purpose, implementation, and usage, as opposed to people's opinions.

In a more recent study, Uddin, Khomh & Roy (2020) have expanded their previous findings to include the effectiveness of API methods. They proposed a method for mining API from Stack Overflow that is based on natural language processing and machine learning techniques. The method involves extracting problematic API from Stack Overflow questions and answers and then clustering them to identify frequent patterns of usage and occurred problems. The authors evaluate the proposed method on a dataset of Stack Overflow questions and answers that contain references to the Java and Python programming languages. They found that the method was able to accurately extract API from the text of the questions and answers and that the resulting scenarios were useful for understanding how developers use the API methods. The method was able to identify frequent patterns of usage and frequent problems that developers encounter when using the API methods. Compared to our work, in our research, we do not examine problematic API methods. We are interested in all API methods and trying to generate a summary of their usage and implementation instead of solving frequent problems.

Saddler *et al.* (2020) aimed to understand how developers read and interact with Stack Overflow when working on API summarization tasks. API summarization tasks involve finding code snippets that demonstrate how to use a specific API. The authors used eye-tracking and mouse-tracking technology to record the reading behavior of 23 participants while they completed API summarization tasks. The study found that developers spend more time reading when the task is more difficult and that the use of API documentation is not a significant factor in completing the task. This suggests that developers rely more on community-generated information and less on official documentation when working on more challenging tasks. The paper concludes that this information can be used to improve the design of API documentation and tools for developers. For example, by highlighting the most relevant information, such as the top-voted answer, or by providing more context, such as the task difficulty, in the documentation. In contrast to this work, we use machine learning and deep learning algorithms to produce summaries for techniques covered in unofficial documentation.

Another study by Jiang, Armaly & McMillan (2017) suggested a method for automatically generating commit messages from code changes, using neural machine translation. The authors use a dataset of code changes and corresponding commit messages from GitHub and train a neural machine translation model to generate commit messages from the code changes. The authors found that the neural machine translation model was able to generate commit messages that were both grammatically correct and semantically meaningful and that were similar to the original commit messages. The authors also evaluated the generated commit messages by having human evaluators rate their quality and found that the generated messages were comparable to messages written by humans. Oppositely to this research work, we have proposed approaches that summarize API methods discussed in unofficial documentation using a set of algorithms ranging from an unsupervised machine learning algorithm (TextRank) to an advanced deep learning transformer model (BART).

Neural networks have been also exploited in the context of the summarization task. In effect, Iyer, Konstas, Cheung & Zettlemoyer (2016) has proposed a method for generating summaries of source code using a neural network-based approach with an attention mechanism. The authors use a dataset of Java code and corresponding natural language summaries and train a neural network to generate summaries. The model uses an attention mechanism to weigh the importance of different parts of the code and to identify the most relevant parts for the summary. The generated summaries were found to be grammatically correct and semantically meaningful and were comparable to the original human-written summaries. The authors conclude that the proposed method can be useful for software development and maintenance tasks and that the attention mechanism can be applied to other natural language generation tasks.

Korayem (2019) has attempted to investigate the use of machine learning to produce summaries of code entities, such as methods and classes, in bug reports. The proposed approach involved extracting code entities from bug reports using an island parser, a technique to identify code in text and apply machine learning to select a set of useful sentences to be included in the summaries. The machine learning technique used is logistic regression, which ranks sentences based on their importance by building a corpus of sentences that contain the code entities of interest. The summaries were evaluated using surveys to assess their quality, and the results have shown that bug reports are not a good source of information for generating summaries. While this study used basic machine learning algorithms such as logistic regression for generating summaries, we applied advanced deep learning transformer models which are the state-of-the-art techniques in summarization tasks (Lewis *et al.*, 2020).

More recently, Harirpoosh (2021) investigated the use of automated source code summarization, specifically unsupervised learning, TextRank, to extract relevant sentences that should be part of the summary produced for Android methods mentioned in Stack Overflow. The study uses Natural Language Processing (NLP) techniques, specifically Term Frequency-Inverse Document Frequency (TF-IDF) and Stack Overflow's word embedding as text vectorization techniques, and examines how different vectorization techniques affect the quality of the produced summaries. The research is evaluated using an empirical study that involves Stack Overflow posts. The results show that automatic summaries for code entities discussed in informal documentation can be produced using TextRank and that using word-embedding vectorization provides more stable results than using TF-IDF with TextRank algorithms. Likewise, we used TextRank as one of our algorithms however, we also applied deep learning to generate summaries and conducted empirical evaluations and surveys to evaluate our summaries.

Parnin & Treude (2011) focused on the impact of social media on software documentation, specifically in relation to the jQuery API. The researchers examined a total of 1,730 links gathered from web searches related to API methods. The analysis of these links revealed that software development blogs play a crucial role in providing comprehensive coverage of the jQuery API, accounting for 87.9% of the resources. These blogs offer tutorials, share personal experiences, and serve as a valuable resource for developers seeking information and guidance. The study suggests that social media platforms have the potential to complement or even replace traditional documentation approaches. By leveraging social media, developers can access a wealth of information and insights from the developer community. This shift towards crowd documentation is seen as a promising approach for software documentation, as it harnesses the collective knowledge and experiences of a diverse range of developers.

Subramanian *et al.* (2014a) discusses the challenges of using Application Programming Interfaces (APIs) and the limitations of traditional API documentation. It highlights the importance of example-based resources such as Stack Overflow and Github Gists in complementing API documentation.

The authors propose an iterative and deductive method called "Baker" to link source code examples to API documentation. Baker is a precise tool that supports both Java and JavaScript languages. It aims to enhance traditional API documentation by providing up-to-date source code examples and incorporating links to the API documentation within code snippets.

The paper identifies two main issues in using APIs: the difficulty in understanding how to use them correctly and the lack of up-to-date documentation. It emphasizes the significance of online resources like Stack Overflow, which cover a wide range of APIs but lack integration with the official API documentation.

CHAPTER 2

OBJECTIVES AND RESEARCH QUESTIONS

This section focuses on the objectives, and research questions of our research study on automatic code summarization. Our research study aims to guide and assist developers in comprehending the API methods that are part of their engineering tasks. Our approaches have the potential to be developed into practical tools that can benefit not only the software engineering research community but also practitioners interested in adopting such approaches in real-world contexts.

We have derived specific objectives to guide our research in the field of automatic code summarization. These objectives are designed to address the challenges and requirements identified in our goals and drive our efforts toward achieving meaningful outcomes. By pursuing these objectives, we aim to advance the state of the art in code summarization and provide valuable support to software developers. In summary, the objectives of our research can be summarized as follows:

1. **Exploring the use of informal documentation:** We aim to investigate the potential of leveraging informal documentation, such as Stack Overflow, to generate code summaries for API methods. By analyzing and extracting insights from real-world code discussions, we can capture valuable information that may not be available in official documentation. This exploration will enable us to understand the usefulness and effectiveness of informal documentation in generating code summaries;
2. **Comparing and analyzing different algorithms:** We intend to explore and compare various algorithms for generating code summaries. By examining different approaches, such as machine learning and natural language processing techniques, we can identify the most effective algorithms for generating relevant and informative code summaries. This analysis will contribute to advancing the field of automatic code summarization by identifying the most promising techniques;
3. **Identification of API topics within Stack Overflow:** In the final part of our study, we employed the topic modeling algorithm to identify key topics from discussions about Android APIs on Stack Overflow, an important source of informal documentation. This

approach helped in pinpointing the central themes in these discussions, which are crucial for understanding frequent issues and solutions related to API usage;

4. **Generating informative summaries for identified topics:** Alongside topic identification, we used the BERT algorithm to create brief yet comprehensive summaries of these topics. These summaries encapsulate the frequent problems and their potential solutions, providing a concise but thorough understanding of the issues at hand;
5. **Evaluating the quality, effectiveness, and performance of automatically generated summaries:** Our objective is to comprehensively evaluate the generated summaries by comparing them both against the official documentation and human-generated summaries. These comparisons aim to assess the quality, effectiveness, reliability, completeness, and conciseness of the automatically generated summaries. Comparing these summaries with official documentation provides a framework for developers to assess the trustworthiness and utility of the generated summaries in real-world scenarios. Simultaneously, by analyzing their level of similarity, accuracy, and comprehensibility in relation to human-crafted summaries, we can validate the effectiveness and reliability of our automated summarization techniques. This holistic evaluation approach ensures a thorough understanding of the strengths and limitations of automatic summarization methods.

By deriving these objectives from our goals, we ensured that our research efforts aligned with the overarching vision of advancing code summarization, improving developers' productivity, and enhancing the overall quality of software development.

To effectively pursue these objectives, it is essential to establish clear research questions that guide our investigation. The primary research questions addressed in this study are as follows:

- ***RQ1:*** *Can we leverage informal documentation to automatically generate code summaries for API methods discussed in informal documentation?*
- ***RQ2:*** *Are the automatically generated summaries comparable to descriptions from official documentation?*

- ***RQ3:*** *Which of the investigated algorithms are generating the most relevant code summaries for API methods discussed in informal documentation?*
- ***RQ4:*** *What are the prevalent Android API topics on Stack Overflow, and can summarization methods identify and solve frequent issues within these topics?*

These research questions serve as the foundation for our study, guiding our investigations into improving code summarization techniques and providing valuable insights for developers and researchers.

CHAPTER 3

RESEARCH APPROACH

We divide this chapter into three parts to answer the mentioned research questions. We investigate RQ1 and RQ2 in the first part, while we focus on RQ3 in the second part of this research. Finally, we seek the answers to RQ4 and RQ5 in the third part of our research. Figure 3.1 shows the research flow of this thesis.

In part one, we leveraged Stack Overflow as an unofficial documentation and applied the TextRank algorithm to generate extractive summaries. Furthermore, we have evaluated our results by conducting surveys with 16 professional Android developers who have years of experience in the industry. Our findings have shown that unofficial documentation is a valuable source of information for developers, and it has a good potential to be used as a source of generating summaries. Moreover, our investigation has shown that our generated summaries can compete with official documentation in terms of their usefulness to developers. The results of this section are published in the IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) (Naghshzan *et al.*, 2021).

In the second part, based on the promising results of the first part, we have focused on improving the quality of our generated summaries using deep learning. Abstractive summarization creates summaries by paraphrasing the main ideas of a text, rather than merely selecting specific sentences or phrases. Extractive summarization, on the other hand, compiles a summary by identifying and extracting key sentences or phrases directly from the text. Since researchers have shown that abstractive summaries are more coherent and concise than extractive summaries (Hsu *et al.*, 2018), we have leveraged BART, a transformer model for generating abstractive summaries instead of extractive summaries generated by TextRank. Additionally, we have created an oracle of human-generated summaries for the investigated API methods and compared them against our automatically generated summaries by applying widely-used NLP metrics: ROUGE (Recall-Oriented Understudy for Gisting Evaluation) and BLEU (Bilingual Evaluation

Understudy). The results of the second section were submitted to the Journal of Software: Evolution and Process (Naghshzan, Ratté, Guerrouj & Baysal, 2023b).

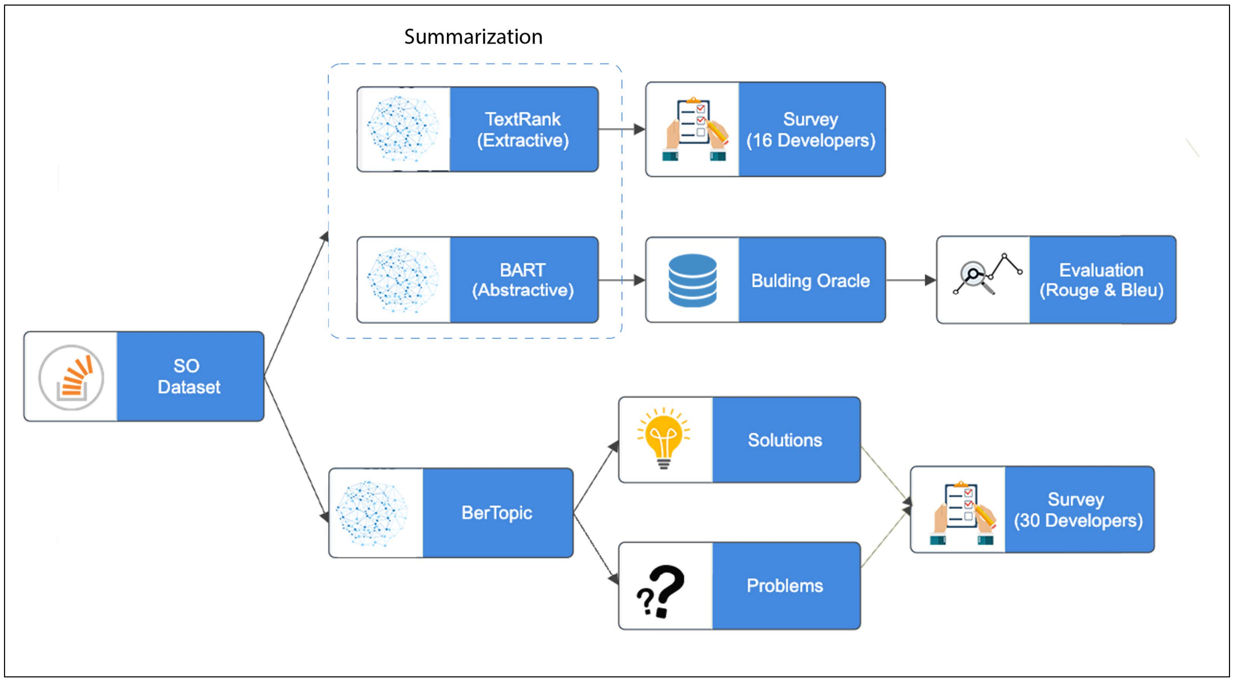


Figure 3.1 Research flow of thesis.

In the final part of our research, we utilized the BERTopic algorithm to identify key topics from Stack Overflow discussions about Android APIs, which are significant sources of informal documentation. This approach was enhanced with BERT's capabilities in identifying frequent issues and their potential solutions, thus generating concise yet informative summaries for these topics. We evaluated the effectiveness of our method by having thirty Android developers review these summaries, focusing on their performance, coherence, and interoperability. This evaluation provided valuable insights into the real-world applicability of our method, highlighting its potential to aid developers with practical and relevant information derived from informal documentation sources. The results of the second part were presented at the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (Naghshzan & Ratte, 2023b).

The results of our research work have resulted in the following research contributions:

- **Leveraging Unsupervised Learning to Summarize APIs Discussed in Stack Overflow**
  AmirHossein Naghshzan, Latifa Guerrouj, Olga Baysal.
  Published in the Proceedings of the IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), 142-152.

- **Revolutionizing API Documentation through Summarization**
  AmirHossein Naghshzan, Sylvie Ratté.
  Presented at 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).

- **Enhancing API Documentation: A Combined Summarization and Topic Modeling Approach**
  AmirHossein Naghshzan, Sylvie Ratté, Latifa Guerrouj, Olga Baysal.
  Submitted to the International Journal of Software Engineering and Knowledge Engineering.

- **Leveraging Advanced Data Mining Algorithms to Recommend Source Code Changes**
  AmirHossein Naghshzan, Saeed Khalilazar, Pierre Poilane, Latifa Guerrouj, Olga Baysal, Foutse Khomh.
  Accepted in Journal of Software Engineering Research and Development (JSERD).

CHAPTER 4

LEVERAGING UNSUPERVISED LEARNING TO SUMMARIZE API METHODS DISCUSSED IN STACK OVERFLOW

This chapter focuses on finding answers for *RQ1: Can we leverage informal documentation to automatically generate code summaries for API methods discussed in informal documentation?* and R*Q2: Are the automatically generated summaries comparable to descriptions from official documentation?* mentioned earlier in Chapter 2. In this chapter, we focus on the extractive summarization of API methods discussed in Stack Overflow. The results are evaluated by surveying professional developers. This chapter is based on an article entitled *Leveraging Unsupervised Learning to Summarize API methods Discussed in Stack Overflow* published in the IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM) (Naghshzan *et al.*, 2021).

4.1 Introduction

By using automatic code summarization, software developers can easily understand source code through natural language summaries. This eliminates the need for manually reading and interpreting the whole code, saving time and effort. Moreover, automatic code summarization can also be useful to detect bugs or discrepancies in the code, making it a valuable tool for software development and maintenance. According to Zhu & Pan (2019), the use of code summaries is essential to achieving a deeper level of understanding and comprehension of a program's source code throughout the creation and maintenance of software. In addition, developers can spend less time comprehending the code with the help of code summaries. They can be used to quickly identify key sections of the code, such as functions and classes, and provide an overview of their functionality, which is beneficial while trying to satisfy or change program requirements. Automatic source code summarization is one of the engineering problems that has received high attention and it is still a topic of interest for researchers and practitioners in recent days.

There are many instances in which developers are not aware of the purpose and–or usage of a code entity. This can happen for a variety of reasons, such as poor documentation, a lack of clear naming conventions, or a lack of communication between team members. This can lead to confusion and errors, as well as make it more difficult to maintain and update the code. They must read through a substantial quantity of code or documentation to comprehend the idea associated with a code entity. Therefore, it would be beneficial to have an automatic method that might offer them summaries on the purpose, implementation, and/or usage of code entities that are a part of their engineering tasks. Take, for example, a developer who inherits a codebase from another team or developer. Without proper documentation or clear naming conventions, the developer may have difficulty understanding the function of certain parts of the code and how they fit into the overall system. For example, the developer may come across a function called but without any further information, it's unclear what kind of data it processes, what's the input format, and what's the output format. This can lead to confusion and errors as the developer may make changes to the code without fully understanding the implications of those changes.

Despite developers primarily relying on official documentation as a source of information about code entities (Ponzanelli, Bavota, Di Penta, Oliveto & Lanza, 2014), research has shown that these resources can be incomplete, lacking in insight and succinctness (Robillard, 2009; Uddin & Khomh, 2017). As a result, developers may turn to other sources, such as Stack Overflow or GitHub, to gain a deeper understanding of a code entity, its implementation, usage, and additional information that may not be provided in the official documentation. These alternative sources, provided by the software community, are referred to as unofficial documentation.

This research work suggests an automatic method for summarizing API methods using unofficial documentation and unsupervised learning. Within this investigation's scope, we use Stack Overflow as informal documentation for our inquiry. In addition, we emphasize extractive code summarization, a process that takes documents, *i.e.*, Stack Overflow posts in our case, and select the most significant sentences as a final summary.

The goal is to guide and assist developers while they work on the activities associated with software maintenance and evolution by providing accurate summaries to make it easier to comprehend the API methods that are a part of those chores. The findings of this study have the potential to be developed into practical tools that could improve the way developers access and understand information about code entities. Our preliminary findings indicate that developers are interested in the obtained results that can be provided as an assistance tool.

In this chapter, we address the following research questions. By understanding the impact of using unofficial documentation sources on the software development process, and suggesting best practices for documentation, we can help make code more accessible and understandable for future developers. This will make the development process more efficient, consistent, and maintainable.

- **Q1**: *Can we leverage unofficial documentation and unsupervised learning to summarize API methods?*
- **Q2**: *Are the automatically generated summaries perceived valuable by developers?*
- **Q3**: *Can the automatically generated summaries compete with the descriptions presented by official documentation?*

4.2 Methodology

In this section, we outline the key steps of the methodology we have followed. Figure 4.1 shows the main steps are: data collection, extraction of API methods mentioned in Stack overflow, building a corpus for each API method, pre-processing the corpus, summarization (vectorization and applying TextRank), empirical evaluation, data analysis, and finally generation of the results. In this section, we go over each step in detail.

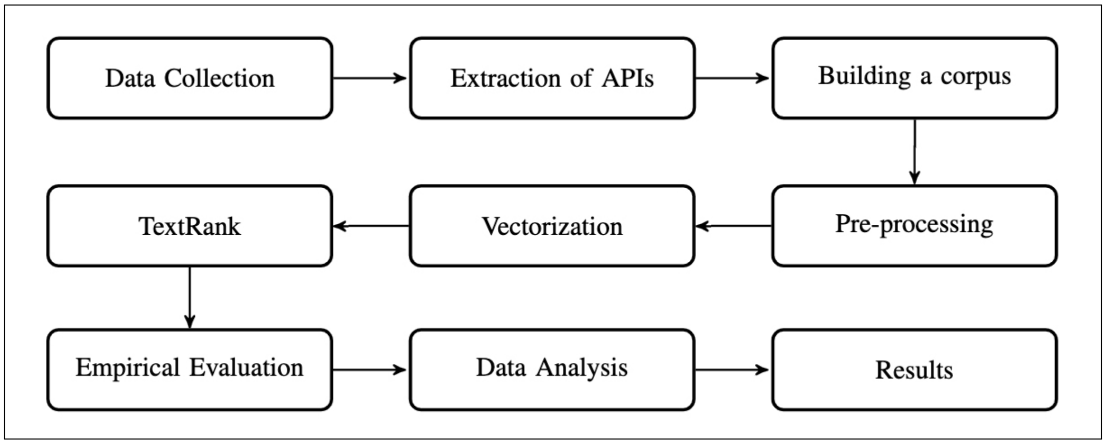


Figure 4.1 Main steps of the methodology
Taken from Naghshzan et al. (2021, p. 04)

4.2.1 Data Collection

As an alternative to official documentation, we investigated Stack Overflow questions and answers with *Android*. Our selection of Android was guided by its status as one of the top eight subjects of discussion on Stack Overflow, as demonstrated by the high volume of questions tagged with Android-related tags. The first author's extensive knowledge of Android development also made it easier to comprehend and evaluate the results obtained from the various methods.

We have used the Stack Exchange API to collect all *Android* tagged questions from January 2009 through April 2020 on Stack Overflow. We have extracted 1,266,269 distinct Android questions. As shown in Table 4.1, we extracted 1,817,874 answers to the collected questions. In total, we have leveraged Stack Overflow to collect 3,084,143 unique Android posts.

Table 4.1 Analyzed Data Extracted from Stack Overflow

Posts	**Count**
All Questions	21,165,633
Android Questions	1,266,269
Answers to Android Questions	1,817,874
Total of Android Posts	3,084,143

4.2.2 Extraction of API Methods

The next step is the extraction of the API methods discussed in Stack Overflow posts. We extracted this data using Stack Overflow's *Code Snippet* function, which has been available since August 25, 2014. This feature highlights code blocks in a post to make it simpler for users to recognize and use the code. For Stack Overflow to recognize a user's post as a code snippet, the user must write the code between quotation marks. If a user forgets to place the code inside quotation marks, other users can edit the post and put the code between quotation marks. Stack Overflow converts the quotation to an HTML element in the post's pure HTML format (<`code`>). We have detected code snippets within posts by automatically parsing the body of posts and extracting the text enclosed by the <`code`> tag.

We have processed the posts and retrieved all the API methods discussed in Stack Overflow. However, in some cases, the code entities do not represent API methods. For instance, *android:configChanges="orientation|screenSize"*, is not an API name. To address the problem and ensure the accuracy of API names, we employed a semi-automated process for code entity verification. Initially, we utilized regular expression patterns to identify non-pertinent code entities. In the context of Android, method names are referred to as "identifiers," which consist of a series of letters and numbers without any special characters, except for underscores ("`_`"). As a result, only code entities consisting of letters, digits, underscores, dots, and parentheses were considered in our analysis, using the pattern "`^[^a-zA-Z0-9_.()]`". The inclusion of dots and parentheses was necessary as they may represent fully qualified API names with package names or methods written with parentheses and arguments, as demonstrated in Figure 0.2. However, in some instances, code entities took the form of file names such as `studio.sh`. To distinguish between code entities and file names, we applied the regex pattern `^(/+\w{0,}){0,}.\w{1,}$`, which helped us identify and remove the latter. Finally, the first author conducted a manual review of the remaining code entities to eliminate any that were irrelevant or invalid Android methods. During this process, 37 code entities were found to be non-essential and removed, mostly consisting of Android class names such as `Activity` and `TextView`.

We have discovered that the most frequent method in Android posts is the *app.Activity.onCreate()* method. This method has been mentioned 7,933 times in questions and 21,103 times in answers. In total with 29,036 mentions, `app.Activity.onCreate()` is the most frequently referenced Android method in Stack Overflow. Figure 4.2 shows the most frequent Android API methods discussed in Stack Overflow.

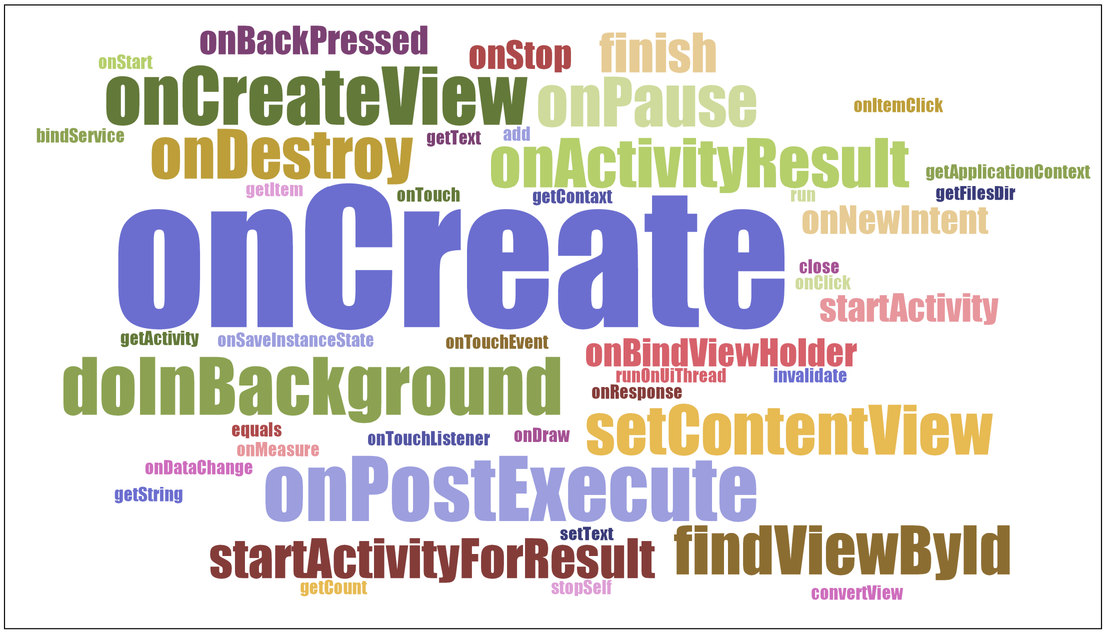


Figure 4.2 Most repeated Android methods in Stack Overflow
Taken from Naghshzan et al. (2021, p. 04)

After sorting the discussed API methods based on their frequency, the 15 most frequently used API methods were chosen for the summarization task, and the deprecated API methods were eliminated. Our observations have led us to conclude that the ability of algorithms to produce high-quality summaries diminishes in proportion to the reduction in corpus size. This finding underscores the importance of having an adequate amount of data to facilitate optimal algorithmic performance. Consequently, we have taken the liberty of selecting the top 15 frequently used summarization methods since the availability of data for other methods drops significantly beyond this threshold. Table 4.2 shows how frequent these methods are in Stack Overflow posts.

Table 4.2 Top Android API methods appeared in Stack Overflow posts.

Method	**Question**	**Answer**	**Total of Posts**
activity.onCreate	7,933	21,103	29,036
asyncTask.onPostExecute	1,413	5,452	6,865
app.Fragment.onCreateView	1,769	4,291	6,060
apctivity.onActivityResult	2,196	3,853	6,049
os.AsyncTask.doInBackground	1,135	4,569	5,704
activity.onPause	1,556	4,031	5,587
activity.findViewById	804	4,182	4,986
activity.onDestroy	1,588	3,333	4,921
activity.finish	1,105	3,224	4,329
activity.setContentView	627	3,562	4,189
activity.onStop	949	2,188	3,137
activity.startActivityForResult	779	2,325	3,104
recyclerview.onBindViewHolder	845	1,973	2,818
activity.startActivity	558	2,217	2,775
activity.onBackPressed	652	1,906	2,558
Total	23,909	68,209	92,118

4.2.3 Building a Corpus of API methods

In the field of natural language processing, building a corpus is a crucial step in preparing for summarization tasks. A corpus refers to a large and diverse collection of text data that serves as a representative sample of the language and text structures that a summarization model will encounter in real-world scenarios. Having a large and diverse corpus allows the model to learn the language patterns, structures, and relationships between words, sentences, and paragraphs that are relevant to summarization. This helps the model generate accurate and relevant summaries by taking into account the context and content of the text being summarized. In summary, building a corpus is crucial for training effective summarization models that can handle a wide range of text types and topics. In the following, we explain in detail the corpus creation process for our summarization process.

Creating a corpus for each API method requires selecting posts with relevant information about each API method. However, the difficulty is in determining which posts have this relevant information. For each API method, we have extracted all the Stack Overflow posts that the API methods appeared in. When an API method's corpus includes Stack Overflow questions, the resulting summaries may feature more question-style sentences than usual declarative ones. As a result, we have relied solely on Stack Overflow answers and question titles to construct our corpus. We have considered titles to be concise descriptions of queries that reveal key information about the question.

Furthermore, we have observed that not all answers on Stack Overflow are helpful, and some may contain false information that confuses developers rather than helping them. Incorporating such answers into the corpus may cause erroneous data to be included in our summaries. Therefore, we have reduced this possibility and enhanced the quality of our corpus by including only responses that have a higher than the average score for all Android answers in our dataset, which is 2.87. As a result, since scores are integer numbers, only answers with a score of 3 or higher were chosen.

Once the Stack Overflow posts have been selected, we have considered their bodies for incorporation into our corpus. However, evaluating all the sentences in the body of a Stack Overflow post may not be particularly relevant, as Stack Overflow posts may contain irrelevant sentences. Hence, we have considered the following criteria: relevant sentences (criterion 1 and 2) in the context of the summarization task as well as proximity (criteria 3 and 4) that have been already applied in previous works (Naghshzan *et al.*, 2021; Guerrouj *et al.*, 2015; Dagenais & Robillard, 2012):

1. The opening sentence of each post;
2. The primary sentence that contains the method;
3. A sentence that precedes the primary sentence;
4. The sentence that follows the primary sentence.

We have used the standard Natural Language Toolkit (NLTK) sentences Tokenizer to split extracted posts' body text into sentences. Following the above methodology, a corpus has been constructed for each API method in our dataset.

4.2.4 Pre-Processing

After constructing a corpus for each API method, the corpus has been prepared for the summarization task based on the following steps.

We have first removed punctuation, numbers, HTML tags, and other special characters in sentences. Using the Natural Language Toolkit (NLTK), we have deleted stop words from sentences in our corpus. Furthermore, lemmatization has been utilized to accurately calculate the weight of words by extracting their roots. Moreover, duplicate sentences in the corpus have been removed.

Text can be converted into vectors using various methods such as one-hot encoding, TF-IDF, word embeddings, bag of N-grams, and Doc2Vec. One-hot encoding represents a text as binary values, TF-IDF assigns weights based on frequency and rarity, word embeddings use continuous values to capture semantic and syntactic properties, bag of N-grams uses the frequency of

N-grams and Doc2Vec uses the meaning and context of the entire text. The choice of method depends on the task and desired representation level. We have used Word Embeddings in the data representation and summary creation process. Word embedding provides a compact and dense representation of the meaning and context of words in a text. Unlike other methods such as one-hot encoding, which are based on the presence or absence of individual words, word embeddings capture the relationships between words and the way they are used in context. Another advantage of word embeddings is that they are learned from large amounts of text data, which enables them to capture the rich and complex relationships between words (Efstathiou, Chatzilenas & Spinellis, 2018). We have used Stack Overflow's pre-trained word embeddings, which were suggested by Efstathiou *et al.* (2018). They trained their word2vec model using over 15 GB of textual data extracted from Stack Overflow.

4.2.5 Summarization

There are different algorithms for extractive summarization like TextRank, KL-Sum, etc. To summarize API methods discussed in Stack Overflow, we have applied the TextRank algorithm, which is an unsupervised learning algorithm for the extractive summarization task (Mihalcea & Tarau, 2004). The rationale behind this choice is that TextRank produces high-quality text summaries, as shown in Mihalcea & Tarau (2004). Furthermore, it is efficient, and robust, and considers semantic relationships between sentences, leading to more meaningful and coherent summaries. TextRank is also unsupervised and customizable, making it a useful tool for summarizing large text collections. This algorithm relies on two primary techniques to achieve its goals, cosine-similarity and the PageRank algorithm described in Section 1.1.2.

The cosine similarity method has been employed to estimate the similarity between sentences. As per the explanation given by Baoli et al. Li & Han (2013), the cosine similarity method is a technique used to examine the similarity between two vectors. In addition to this, the method can also be utilized in text classification and text summarization.

The TextRank algorithm output is a ranked list of sentences, each ranked based on its relevance to the content being summarized. The final step in the summarization process is to combine these sentences to create a concise and informative summary. However, the length of the summary is a crucial factor to consider. If the summary is too long, it may be too time-consuming and difficult for readers to digest. On the other hand, if the summary is too short, important information may be omitted.

To address this challenge, we have used a threshold to select the top three sentences for each method to generate the final summary. This threshold was established based on our experience and aims to strike a balance between the length of the summary and its comprehensiveness. The final summary generated in this way should be reasonable in length, easy to read, and contain all the essential information. Examples of summaries generated for five randomly selected methods from our data can be found in Table 4.3. This table showcases examples of summaries for Android API methods, generated using the TextRank algorithm based on discussions from Stack Overflow. It pairs API methods such as `app.activity.onCreate`, `app.activity.startActivity`, and `app.activity.onActivityResult` with concise summaries that highlight key functionalities and practical usage scenarios. For instance, it explains how `onCreate` serves as the initial setup phase for an activity, `startActivity` is used to navigate between activities with considerations for context, and `onActivityResult` manages data returned from subsequent activities. These summaries distill complex technical discussions into accessible insights, demonstrating the algorithm's ability to capture essential aspects of API methods and their applications, as informed by real-world developer experiences and challenges.

Overall, the TextRank algorithm, combined with careful consideration of the summary length, provides a reliable and efficient approach to summarizing text. The full list of automatically generated summaries for the set of examined API methods is available in the Appendix (cf. Table I-1).

Table 4.3 Examples of automatically generated summaries for API methods discussed in Stack Overflow using TextRank

API Method	Automatically-generated Summary
app.activity onCreate	When the app is launched, the first thing that's going to run is onCreate() in this case, onCreate() has a method that inflates the view of your activity, that method is called setContentView(). When you pass data from one activity to another using a Bundle, the data is received inside the onCreate() method of the second activity, not insideonActivityResult() unless you've specifically implemented that. Inside your Activity instance's onCreate() method you need to first find your Button by its id using findViewById() and then set an OnClickListenerfor your button and implement the onClick() method so that it starts your new Activity.
app.activity startActivity	if this method is being called from outside of an Activity Context, then the Intent must include the FLAG_ACTIVITY_NEW_TASK launch flag. What you can do is to simply start the activity using startActivity() and have the called activity call startActivity() to return to your activity, sending back data as necessary as extras in the Intent it uses. You still only need to call startActivity() to launch a different Activity.
app.activity onActivityResult	To get the returning data from your second activity in your first activity, just override the onActivityResult() method and use the intent to get the data. In the second activity perform whatever validation you need to do (this could also be done in the first activity by passing the data via an Intent) and call setResult(int resultCode, Intent data) (documentation) and then finish(); from the second activity. From your activity1 start activity for result, and from activity2 use the setResult(int resultCode, Intent data) method with the data you want your activity1 to get back, and call finish() (it will get back to onActivityResult() in the same state activity1 was before).

4.3 Empirical Evaluation

Our empirical study was designed to evaluate the accuracy of the automatically generated summaries. This study was conducted with the goal of providing developers with valuable tools to help them quickly understand API methods that are essential to their engineering tasks. We believe that by giving developers accurate and concise summaries of API methods, they will be able to make informed decisions and complete their engineering tasks more efficiently.

The results of our study will provide important insights into the effectiveness of our approach and help us make any necessary improvements. Additionally, by understanding how developers rate the accuracy of our summaries, we can better serve their needs and provide them with the tools they need to succeed in their engineering tasks. Ultimately, our goal is to empower developers by providing them with tools that allow them to save time and effort, enabling them to focus on other important tasks.

4.3.1 Participants

We have contacted 28 Android developers through email and invited them to participate in our survey after receiving approval from our university's ethical committee. We received 16 developers' responses who consented to take part in the survey. The participants consisted of professional Android developers and students with professional Android programming expertise. Our participants included both males and females with different levels of Android development skills. To ensure our conclusions are accurate from an industrial perspective and can be used in the long-term by professional developers in practical contexts, we have intentionally recruited individuals with relevant work experience.

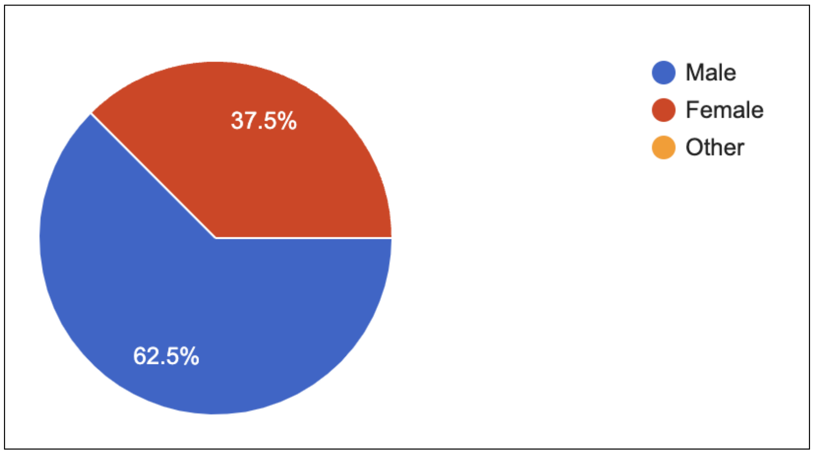


Figure 4.3 Participants gender

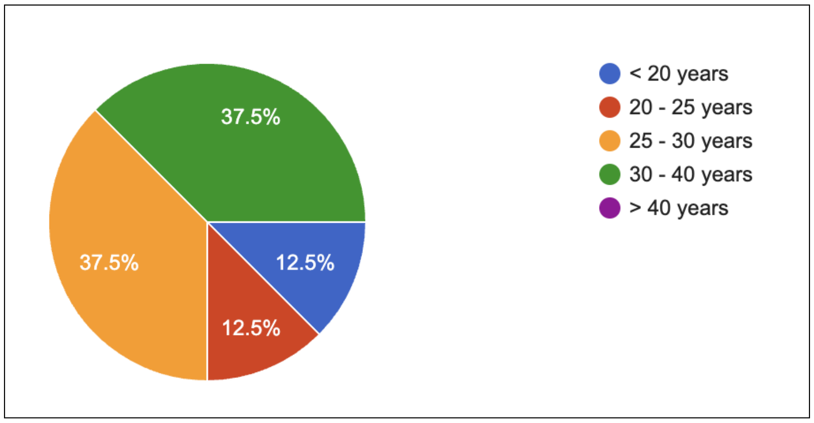


Figure 4.4 Participants age range

As shown in Figure 4.3, a majority of our participants, 62.5%, identified as male, while 37.5% identified as female. Additionally, the data shown in Figure 4.4 indicates that the majority of our participants were between 25 to 40 years old. This information is valuable as it helps us understand the gender distribution of our participant pool and ensure that we are reaching a diverse group of individuals.

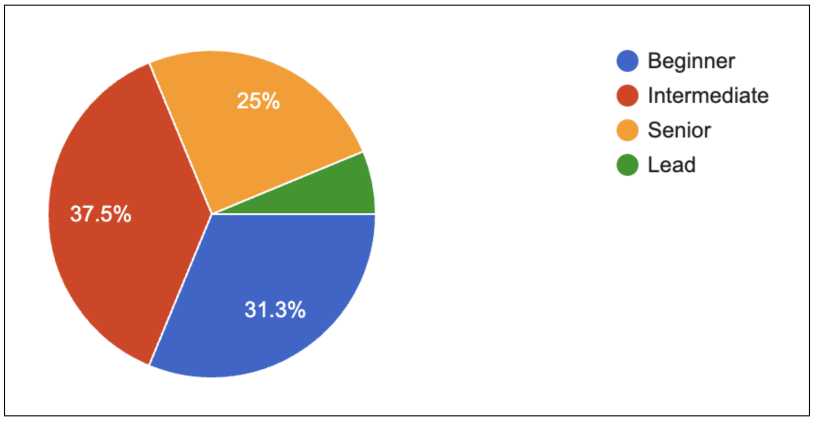


Figure 4.5 Participants' Android development skills

As we can see in Figure 4.5, the participants mostly have either intermediate (37.5%) or junior (31.3%) skills in Android Development. Moreover, 25% of them are senior Android developers and only 6.2% of participants are lead Android developers. This variety of skill levels is valuable for the study because each developer has a different behavior based on the skill level. For instance, Gao *et al.* (2018) found that developers with more experience were less likely to make errors in API usage and more likely to identify and correct errors when they did occur, therefore the results are not biased based on a specific group.

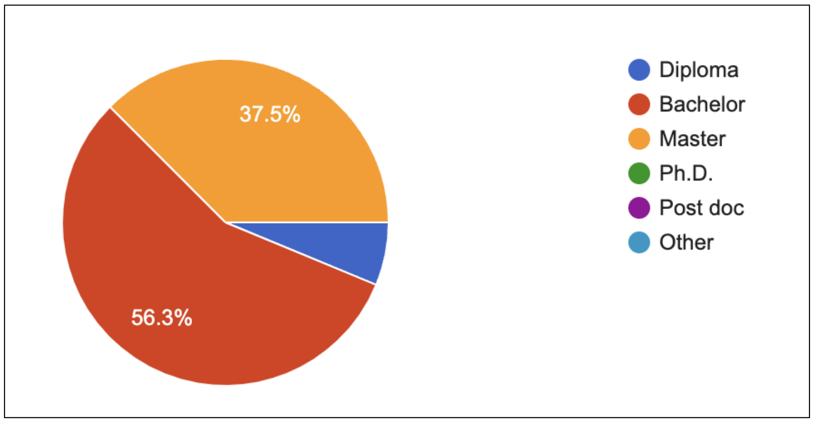


Figure 4.6 Participants level of study

In terms of the study level of participants, as shown in Figure 4.6, the highest level of study among the participants is a master's degree. Most of the participants have a bachelor's degree while 37.5% of them have a master's degree and the rest (6.2%) either have a high school diploma. Furthermore, the majority of the participants (50%) are professional developers, currently employed in industry, while 43.8% are students with job experience (Cf. Figure 4.7) and the rest are people working in academia.

Figure 4.8 shows the participants' programming experience in general. As you can notice, the majority of participants, *i.e.*, 56.3% have less than 5 years of professional experience in programming, and 25% of them have 5 to 10 years of professional programming experience. The rest of them have more than 10 years of professional programming experience.

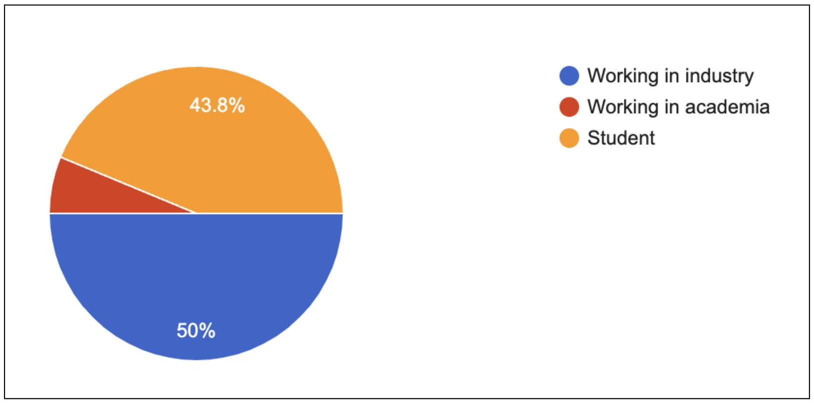


Figure 4.7 Participants current status

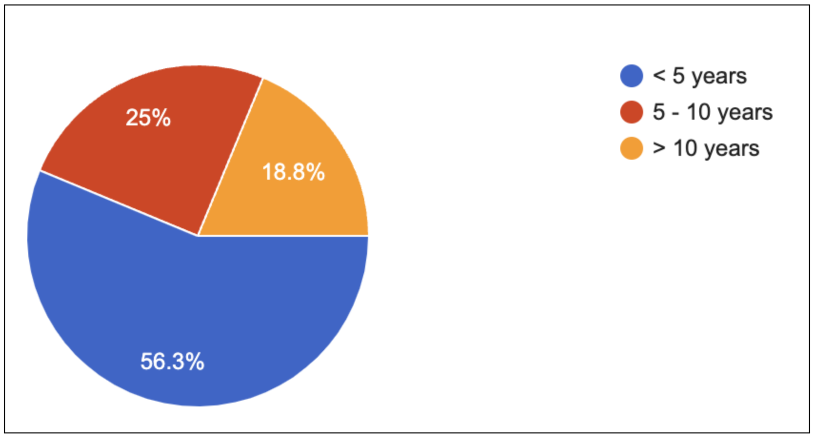


Figure 4.8 Participants programming experience

The rest of the participants either worked less than 2 years (12.5%) or 10 to 15 years (12.5%) as professional developers. Only 6.2% of participants have more than 15 years of professional experience in programming.

As we can notice from Figure 4.9, all the participants in our study use Stack Overflow for development purposes and 43.8% of them use it regularly for their daily tasks. These numbers show how popular is Stack Overflow among developers.

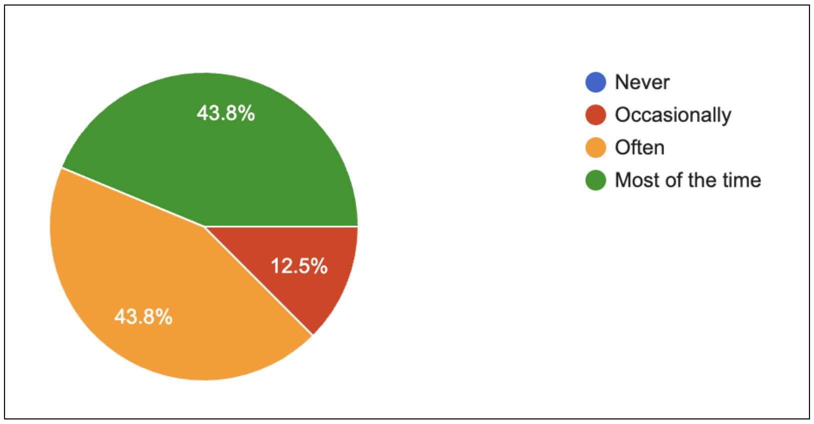


Figure 4.9 How often participants use Stack Overflow

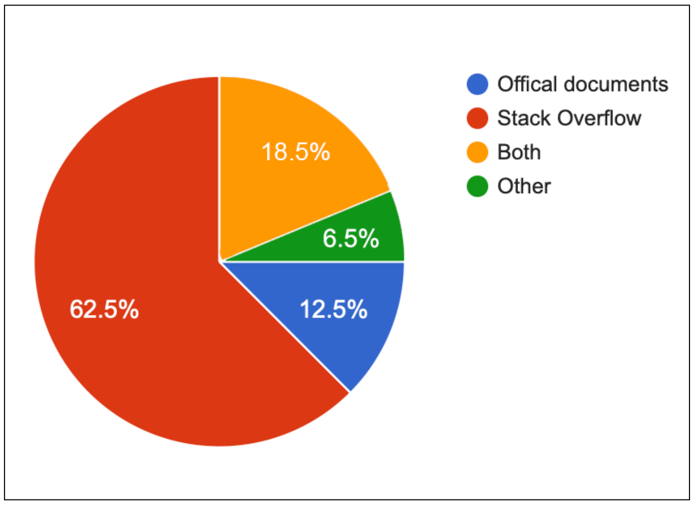


Figure 4.10 Use of Stack Overflow vs. official documentation
Taken from Naghshzan et al. (2021, p. 07)

In addition, we asked participants about the specific resources they use during the development process. As indicated in Figure 4.10, 62.5% rely on Stack Overflow to learn about API methods. Only 12.5% of participants make use of official documentation as their primary source of information, while 18.5% use both Stack Overflow and official documentation. These results show that Stack Overflow is the most popular option among our users.

4.3.2 Study Design

Since most of the participants in our study were at a similar Android development skill level, we applied Completely Randomized Design (CRD) (Wohlin *et al.*, 2012). CRD is a statistical design used in experimental research. In a CRD, the units being studied are randomly assigned to different treatment groups. The objective of using a CRD is to ensure that each treatment group is similar concerning all relevant factors so that any differences observed between the groups can be attributed to the treatment applied (Wohlin *et al.*, 2012). Consequently, we did not have blocks based on developers' knowledge and–or experience with Android development. Participants in a completely randomized design receive their treatments at random.

The questionnaire consists of three parts: pre-questionnaire, main questionnaire, and post-questionnaire. In the pre-questionnaire, we asked participants some standard demographic questions like their gender, age, years of experience, etc. to be sure that we could reach a diverse group of developers.

After answering the pre-questionnaire questions, participants were given three tasks for each API method as the main questionnaire and we asked them to repeat these tasks for three API methods. We have limited each participant to only three API methods to avoid confusion and burnout. The following items are the list of tasks we asked participants to do for each API:

1. In each task, we asked participants to write a summary for a given Android API method. We provided participants with Stack Overflow links for each API method being studied. These links offered participants unfamiliar with a particular API method the opportunity to gain insights and understanding before summarizing it. The purpose of this was to improve the accuracy and comprehensiveness of the summaries;
2. Then, we provided the participants with the summary that has been automatically generated for each Android API method and asked for feedback on how well it meets a set of criteria, including the accuracy, coherence of a summary, its length, and its relevance. Accuracy refers to the degree to which the information in the summary accurately reflects the information in the original text. Coherence, on the other hand, refers to the degree

to which the summary is a coherent and cohesive text that flows logically and is easy to understand;

We have been based on these criteria since they are the most frequently applied standards for evaluating summaries when conducting studies that involve developers (Moreno *et al.*, 2013; Jiang *et al.*, 2017).

3. The final task consists of asking participants to evaluate our automatically generated summaries in terms of usefulness and completeness compared to the official Android documentation. These descriptions are extracted from the official Android Developer website.[1] and they depend on the intricacy of API methods, these descriptions could be either brief single-line summaries or full-page ones.

After completing their tasks, we provide participants with a post-questionnaire about the survey to understand whether or not our summaries could be used as a complement to the official Android documentation the questions of our pre-questionnaires and post-questionnaires are available in I.

4.3.3 Questionnaire

As described in Table 4.2, we have selected the top 15 frequent Android API methods to generate our summaries. Participants were given the automatically generated summaries to evaluate and compare with the official Android descriptions. We did not reveal which summary was taken directly from the official documentation and which was generated automatically by our method to reduce the *courtesy bias* of our study. For each task, the summaries were selected in a completely randomized order. The survey asks participants to rate our automatically generated summaries based on their accuracy, coherence, length, and usefulness. We also asked them about their thoughts and comments on how we can improve our summaries. Those who agreed to participate in our study were given instructions regarding the questionnaire so they would understand the stages, how to complete the questions, and general notions such as coherence, accuracy, and usefulness.

[1] https://developer.android.com

Table 4.4 presents the set of questions asked to participants. Q1 to Q3 focus on aspects related to summary quality and its length. Q4 - Q8 focus on the information that summaries covered and Q9 to Q16 focus on the usefulness of summaries to understand whether the summaries could help developers or not. The complete survey is available in Appendix I.

4.4 Study's Results

The top 15 Android API methods have been selected after we analyzed 3,084,143 distinct Stack Overflow topics. In the following, we present the most significant findings of our survey:

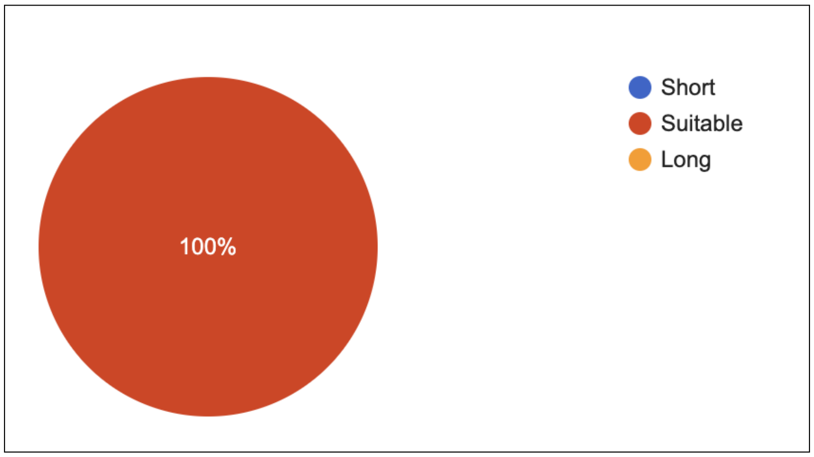


Figure 4.11 Developers' perception on the length of summaries
Taken from Naghshzan et al. (2021, p. 08)

As demonstrated in Figure 4.11, the results of our study show that all participants, 100%, believe that the length of the generated summaries is suitable. To create the summaries, we employed a fixed number of sentences approach, using 3 sentences for each method. However, it's important to note that the length of the sentences can vary depending on the complexity of the methods being summarized. Typically, for simpler methods, the sentences are shorter, while for more complex methods, the sentences tend to be longer.

Table 4.4 Questions asked to participants during the conducted survey

Question ID	**Question**
Question 1	Is the summary coherent?
Question 2	Is the summary accurate?
Question 3	How did you find the length of summaries?
Question 4	Does the summary contain all information about the method?
Question 5	Does the summary contain only important information about the method?
Question 6	Does the summary contain information that helps understand how to implement the method?
Question 7	Does the summary contain information that helps understand how to use the method?
Question 8	Does the summary contain information that helps understand how to implement the method?
Question 9	Can the summaries help developers reduce their effort of searching for relevant information?
Question 10	Can the summaries help developers to reduce the time of development?
Question 11	Can the summaries be helpful for developers in any steps of development?
Question 12	Which summary do you find more useful?
Question 13	Which summary can help you better understand the usage of the method?
Question 14	Which summary can help you better understand the implementation of the method?
Question 15	Do you think the summaries can be used as a complementary source for each other?
Question 16	Is it helpful to have a summarizer within your IDE to get information about methods?

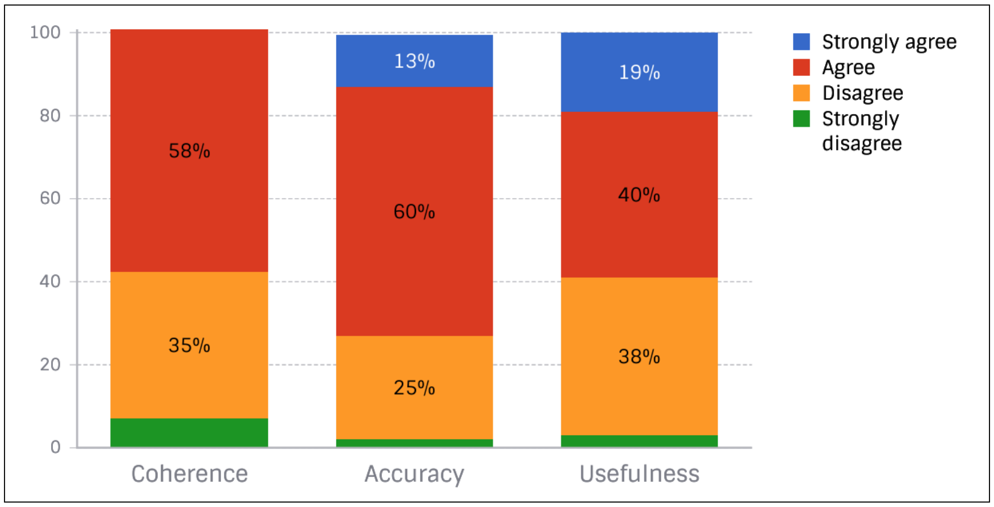


Figure 4.12 Developers' satisfaction with the quality of the automatically generated summaries
Taken from Naghshzan et al. (2021, p. 08)

Participants have evaluated the generated summaries based on three metrics, *i.e.*, coherence, accuracy, and usefulness. Both accuracy and coherence are important for determining the quality of a text summary. A highly accurate but poorly coherent summary may miss important information or provide a confusing or disjointed representation of the original text. On the other hand, a highly coherent but poorly accurate summary may contain incorrect information or distort the original meaning of the text. Therefore, both accuracy and coherence are important considerations when evaluating the quality of text summarization. As it can be noticed from Figure 6.8, approximately half of the participants, *i.e.*, 58% believe that the automatically generated summaries are coherent. The unexpected outcome may be probable because, in this method, summarization is performed by extracting relevant sentences (using unsupervised learning) from a collection of different Stack Overflow posts. The selected sentences are then combined to create the summaries, which may have impacted the coherence. We acknowledge this issue and plan to address it in our future endeavors.

Figure 6.8 also indicates that 73% of participants found that our automatically generated summaries accurately represent Android API methods and methods. The fact that some Stack Overflow questions include incorrect or misleading answers may explain the remaining proportion. When dealing with such posts, erroneous data may be produced and included in the summaries. To address this potential problem, we implemented a mitigation strategy by only including answers that have a minimum score of 3, which indicates a high degree of accuracy and reliability.

Additionally, Figure 6.8 shows that 9% of the participants agreed that the automatically generated summaries have vital and relevant information on the examined Android API methods. Still, developers can benefit from these summaries, even though they are not comprehensive and extracted from a single source (Stack Overflow posts). We are aware of this problem and plan to consider different forms of informal documentation in our future work.

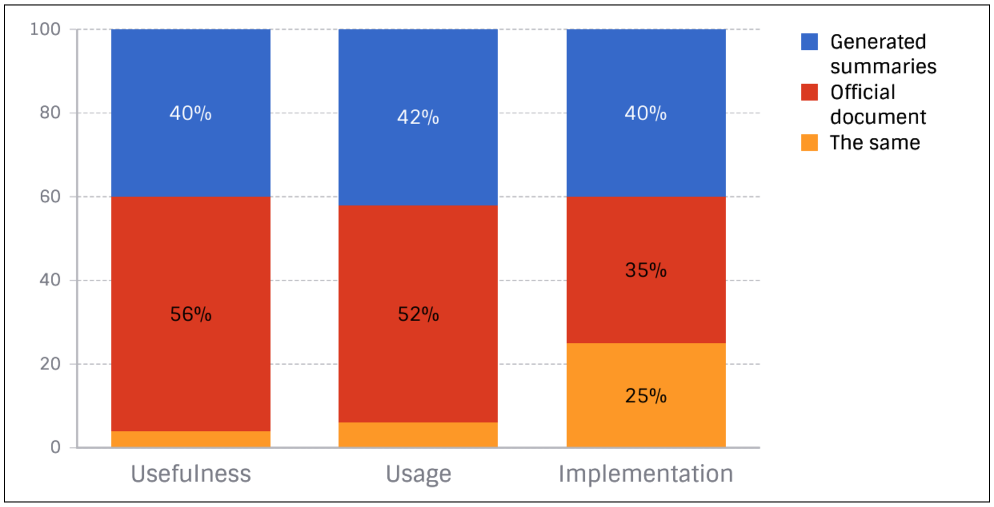


Figure 4.13 Genertaed summaries vs official documentation
Taken from Naghshzan et al. (2021, p. 09)

In case comparing our results with official documentation, Figure 4.13 shows that there are not many differences when comparing the generated summaries to the official Android documentation except a little difference on whether the summary is for implementation or usage. Participants prefer to utilize the automatically generated summaries when implementing Android methods while preferring to read the official documentation when searching for the usage of an API method. This is probably because the official documentation explained clearly the purpose and usage of API methods however when it comes to implementation, it lacks completeness.

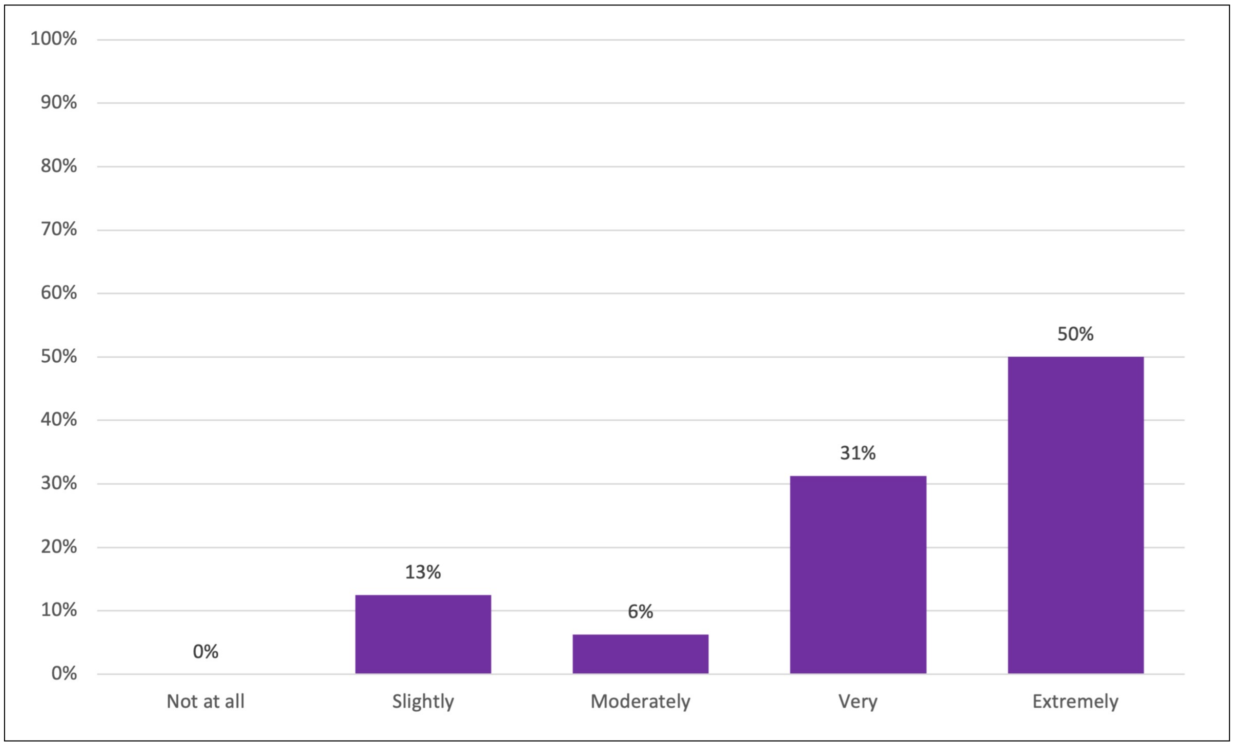


Figure 4.14 Would it be helpful to have a summarizer plugin within IDE

Figure 4.14 shows that the participants agreed on the usefulness of an integrated plugin to display our automatically generated summaries during development steps. The feedback indicates that our approach has the potential to support software developers in their work by providing them with useful and relevant information. With the proper enhancements, the summaries could serve as an effective tool that software developers can utilize to save time and improve their productivity.

Figure 4.15 reveals that the participants in the study have come to an agreement that enhancing the automatically generated summaries with official documentation is a beneficial idea. As has been noted before, research has indicated that official documentation may be inadequate in

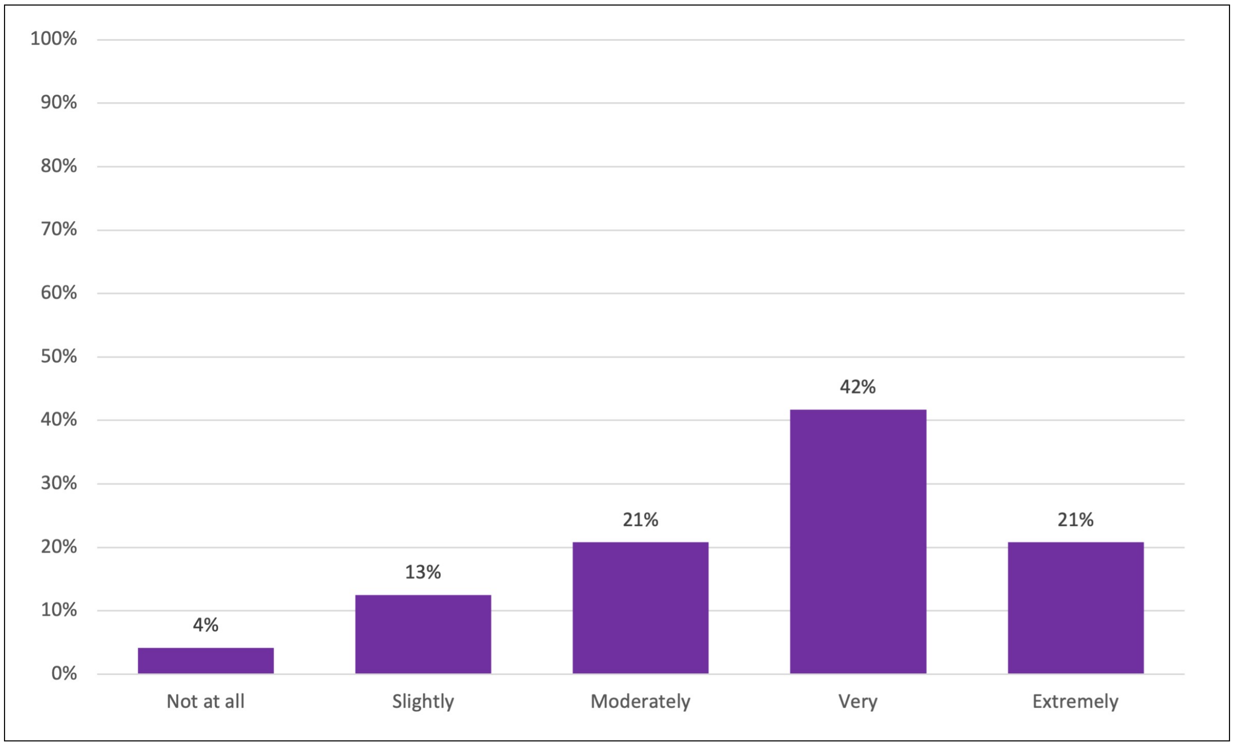


Figure 4.15 Can summaries be used as a complementary source for official documentation

certain aspects, such as being insufficiently detailed, unclear, or lacking in valuable insights. These findings should be taken into consideration in future work related to this field.

4.5 Discussion

In this research, we introduce a novel summarization approach that summarizes API methods discussed in informal documentation using an unsupervised learning algorithm *i.e.*,TextRank.As our dataset, we utilized Android posts from Stack Overflow, and we evaluated the generated summaries by surveying 16 expert software engineers.

Our research revealed that the automatically generated summaries could prove to be a valuable tool for software developers during the development process. The quality of these summaries is comparable to the official documentation in terms of providing insight into the usage and implementation of Android methods. Furthermore, participants in our study concurred that these generated summaries could serve as an additional resource alongside official documentation. This finding has significant implications for the software development industry, as it means that developers no longer have to rely solely on official documentation for understanding complex Android methods. The generated summaries provide a more concise and to-the-point representation of the information, making it easier and faster for developers to grasp the key concepts. Additionally, by supplementing official documentation, developers can now have a better overall understanding of the methods they are working with. This can lead to improved productivity, as developers can more efficiently utilize the information available to them. Overall, our research suggests that the use of automatically generated summaries has the potential to greatly enhance the software development process. In the following, we summarize our main contributions by answering the mentioned research questions in 4.1.

Q1 is related to whether we can use unofficial documentation and unsupervised learning for summarization purposes. Our findings show that we could generate automatic summaries of Stack Overflow posts and Android engineers considered the summaries accurate and pertinent. Our initial dataset contained a number of methods, including some that were less popular. Unfortunately, our attempts to generate summaries for these methods resulted in only a few sentences. To ensure accuracy in our machine learning findings, this study is focused only on the more popular methods. However, we are aware of the limitations posed by this approach and one potential solution is to include additional sources of information, such as source code and official and unofficial documentation, for API methods. Overall, we believe that our findings convey an interesting message to the software engineering research community and practitioners interested in code summarization: *It is possible to generate automatic code summaries for API methods using informal documentation, in particular, Stack Overflow*.

Q2 focuses on whether our automatically generated code summarises are useful for developers or not during the software development process. We have performed a survey that involved 16 Android developers who graciously volunteered to participate in our study. We have requested them to evaluate the automatically generated summaries. Our findings have shown that developers found the automatically generated summaries helpful. Overall, we can conclude that our automatically generated summaries have the potential to help developers during their daily tasks.

Q3 focuses on the comparison of our automatically generated summaries with official documentation. We have requested participants to read the official descriptions and compare them with the automatically generated summaries to address them. The findings have proven that although developers prefer official documentation over generated summaries, there are few differences between the two. Even in the case of implementation, our summaries could provide more helpful information and be more beneficial for developers than official documentation. In addition, participants agreed that the automatically generated summaries could be used as a complementary source for official documentation. These findings motivate us to keep digging further into the problem and enhancing the quality of our automatically generated summaries to utilize them as a practical tool to enhance official descriptions.

Our next step is to evaluate our proposed approach against the current gold standard in the field. In this chapter, we have only tested the viability of our approach with experienced programmers so far. Our preliminary findings are encouraging, but further work is needed to improve our proposed summarization approach. In the next part of this research, we focus on improving the quality of our summaries by applying a state-of-the-art deep learning algorithm in summarization and expanding our data set beyond Android API methods.

4.6 Threats to Validity

Despite our findings, the study has some limitations that pose threats to the validity of our results. Despite the thorough methodology employed, these limitations cannot be completely eliminated. In the following section, we will highlight the most significant limitations and discuss how they may affect the validity of our findings. It is important to consider these limitations when interpreting our results and using the study's conclusions. Understanding these limitations will provide a more nuanced perspective on the implications of our research and ensure a balanced evaluation of its contributions and potential shortcomings.

4.6.1 Internal Validity

Internal validity in research evaluates the degree to which a study's results can be attributed to the independent variable being studied, free from systematic error or bias. It is the foundation for making causal inferences and drawing conclusions about the relationship between variables in a study (Godwin *et al.*, 2004).

To ensure clarity and avoid boredom, we kept our survey straightforward and of moderate length. Despite this, external factors like environmental conditions may impact the results. Our research included analyzing Stack Overflow posts from 2009 to 2020, during which various API versions were released. To account for this, we monitored the changes to the API methods we examined over the years and eliminated any deprecated methods from our data to reduce any potential conflicts with previous Stack Overflow posts.

4.6.2 External Validity

External validity in research assesses the generalizability of a study's results to populations, settings, or situations beyond the sample and conditions of the study. It determines the real-world applicability of the findings (Godwin *et al.*, 2004).

Our research focuses exclusively on the Android mobile platform and APIs discussed in Android Stack Overflow posts. It would be advantageous to broaden our scope to include other programming languages and areas of study. Another limitation is the limited number of APIs we have analyzed - we only studied the top 15 most frequently mentioned APIs from the packages *android.app, android.widget, android.os*, and *androidx.recyclerview*. To produce more comprehensive results, it would be necessary to consider a greater number of methods. Finally, the quality of the data used to generate API summaries is a concern. To maintain the accuracy and usefulness of our summaries, we selected only the most widely discussed methods and avoided summarizing those with limited Stack Overflow coverage.

4.6.3 Reliability Validity

Reliability validity in research assesses the quality and accuracy of a study's findings. They measure the consistency and stability of the results over time, ensuring that if the study were repeated using the same methods and conditions, the results would be similar. (Godwin *et al.*, 2004).

In our endeavor to ensure complete transparency and facilitate the replication of our study, we have meticulously documented the details of our survey and the corresponding responses in the online appendix.[2]

In the next chapter, we try to overcome these problems by applying an abstractive summarization approach. In the abstractive approach, the algorithm reads the whole corpus and generates sentences based on its interpretation, therefore, the exact sentences are not selected from different irrelevant posts. It would help us to generate high-quality summaries in terms of coherence and be closer to human-generated summaries. We have also extended the number of API methods to 25 by using deep learning.

[2] https://github.com/scam2021-so/SCAM2021

CHAPTER 5

LEVERAGING DEEP LEARNING FOR ABSTRACTIVE CODE SUMMARIZATION OF UNOFFICIAL DOCUMENTATION

In this chapter, we present the second part of our research, which addresses our *RQ3: Which of the investigated algorithms are generating the most relevant code summaries for API methods discussed in informal documentation?* mentioned earlier in Chapter 2. This chapter is based on our research paper entitled *Enhancing API Documentation: A Combined Summarization and Topic Modeling Approach* that we have submitted to the International Journal of Software Engineering and Knowledge Engineering (Naghshzan *et al.*, 2023b). This chapter presents a method for generating API summaries using the BART algorithm, a cutting-edge transformer model for APIs discussed on Stack Overflow. To evaluate the accuracy of our approach, we created an oracle of human-generated summaries and compared our results to it using ROUGE and BLEU, the most widely used evaluation metrics for text summarization. Additionally, we conducted an empirical evaluation of our summaries in comparison to our previous extractive approach.

5.1 Introduction

This chapter builds on the first part of our research work presented in Chapter 4 (Naghshzan *et al.*, 2021). During our previous investigation, we have used the TextRank algorithm (Mihalcea & Tarau, 2004) to produce extractive summaries for API methods trapped in informal documentation, *i.e.*, Stack Overflow. This work has been empirically validated using a survey involving professional developers. Our findings have shown that the automatically generated summaries can be helpful during the software development process and those context-aware summarizers are needed and can be implemented in the form of plugins that can be integrated within the integrated development environments.

In this part of our research, the focus is made on enhancing the quality of the automatically generated summaries by our summarization approach published in (Naghshzan *et al.*, 2021). Unlike our previous work (Naghshzan *et al.*, 2021), we leverage a deep learning transformer as a state-of-the-art algorithm to generate abstractive summaries. The extractive summarization method involves selecting important phrases, paragraphs, and sentences from the primary source and combining them to create a short description of a document that is considered a summary (Moratanch & Chitrakala, 2017). However, abstractive summarization reproduces the key content of the main source in a new way after understanding and analyzing the text. It uses advanced natural language techniques to generate a new text that offers essential information from the original one (Hsu *et al.*, 2018).

Researchers such as Hsu *et al.* (2018) have discovered that abstractive summaries can be more coherent and succinct than extractive summaries. We have been therefore inspired to improve our summarization approach by considering the abstractive fashion and generating summaries for API methods using our innovative deep-learning techniques.

To compare the findings of our work's extension with our summaries produced during the first part of this research, we have leveraged the same dataset and corpus, *i.e.*, Stack Overflow has been the type of informal documentation investigated in our case. Additionally, we have used the BART algorithm, a state-of-the-art transformer model for text generation, to generate automatic summaries. Recent comparisons on CNN and DailyMail datasets have demonstrated that BART beats all current works. In addition, it is considered one of the most advanced algorithms in the summarization field (Lewis *et al.*, 2020).

Our comparison has been made with respect to an oracle of human-generated summaries that we have built to assess our approaches and compare them in terms of the accuracy and usefulness of the automatically generated summaries. The accuracy of our automatically generated summaries has been measured in terms of Precision, Recall, and F-measure using ROUGE and BLEU, two widely used metrics in the NLP domain. Compared to other metrics they are easy to compute, provide a clear evaluation score, are based on n-gram overlap, and have been widely used and validated in NLP tasks.

We have considered as our baseline the approach that we have suggested in the first part of this research work (Naghshzan *et al.*, 2021) and we have compared our extended work against this baseline to investigate whether the quality of our automatically generated summaries has been improved or not.

The following is a list of the primary research questions that we address in this research study:

- **Q1:** *Can our deep-learning-based approach be able to automatically generate summaries for API methods discussed in informal documentation?*
- **Q2:** *How does our deep learning-based approach compare to a baseline when summarizing API methods discussed in informal documentation?*

The objective of RQ1 is to explore the feasibility of using deep learning algorithms to generate summaries for API methods discussed in unofficial documentation. The aim is to determine if this approach can be used to automatically generate accurate and concise summaries for this type of information.

The objective of RQ2 is to improve the quality of the previously generated summaries. By applying an abstractive approach using deep learning algorithms, the goal is to see if the generated summaries can become more accurate and comprehensive, while still being concise. The objective is to enhance the usefulness of the generated summaries as a complementary source to the official documentation.

5.2 Methodology

Figure 5.1 outlines the main steps of the research conducted in the second part of this thesis. To compare the summaries generated in this research chapter with those developed previously in Chapter 4, we used the same dataset and corpus for generating summaries. Therefore, our dataset comprises all 3,084,143 unique Android posts previously extracted from Stack Overflow.

For the pre-processing section, we followed the same steps as described in Chapter 4, which are as follows:

1. Removing any irrelevant information, such as punctuation marks, special characters, and stop words, which are unlikely to contribute to the meaning of the text;

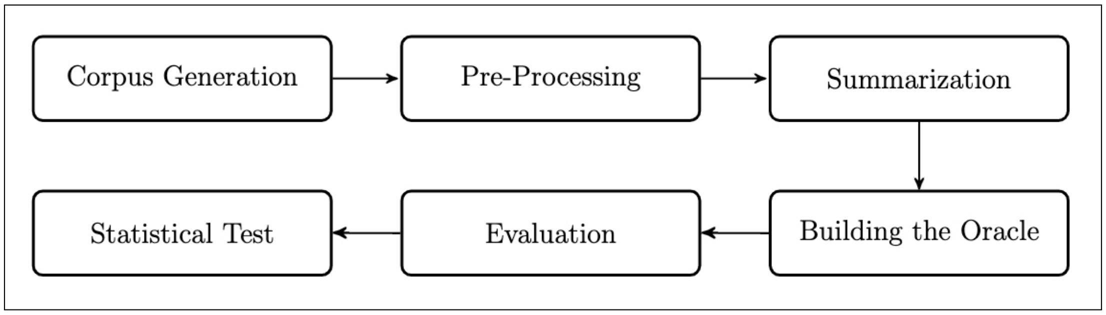


Figure 5.1 Main steps of the methodology

2. Using the Natural Language Toolkit (NLTK) to delete stop words. This helps the summarization algorithm focus on the most important words and create a more concise and meaningful summary;
3. Apply tokenization which is breaking the text into smaller units, such as sentences, to allow for easier manipulation and analysis;
4. Removing duplicate sentences from the main source. The objective of this step is to reduce the redundancy in the text and ensure that the summary only contains unique and non-repeated information.

Once the pre-processing is performed, the corpus is ready for the summarization task of API methods. Previous research studies have proven that abstractive summaries are more coherent and closer to natural languages compared to extractive summaries Hsu *et al.* (2018). We have been, therefore, inspired to generate abstractive summaries using a state-of-the-art deep learning algorithm to increase the accuracy and quality of our automatically generated summaries.

Recent context encoders such as GPT, BERT, and BART (advanced deep learning language models) have been used in a wide range of NLP studies. These algorithms can deliver novel data techniques with superior token embeddings because they have been pre-trained on a big unlabeled corpus. As a result, developing approaches based on them can result in an improved performance (Zhang, Xu & Wang, 2019). Therefore, we have decided to select one of these pre-trained algorithms to increase the chance of generating high-quality abstractive summaries.

In the second part of our research, we have selected BART as our primary code summarization algorithm since the software engineering literature has shown that it outperforms all available summarization algorithms (Lewis *et al.*, 2020). BART is a sequence-to-sequence model trained by using a random noise function to mess up text and then learning how to put the original text back together (Lewis *et al.*, 2020).

The BART language model can be customized for a specific natural language processing task because it is an unsupervised, pre-trained model. Pre-training effectively summarizes and even increases the content selection power (Huang *et al.*, 2020). The model can be fine-tuned without massive labeled datasets because it has already been pre-trained. We have selected the Wolf *et al.* (2020) pre-trained BART model since it is optimized for summarization purposes. Like previous research (Hartman & Campion, 2022), transfer learning was done on our dataset for three epochs and four beams with a maximum input token length of 1,024 and a maximum output token length of 50.

The summarization approach consists of two sections. In the first section, we utilized the *Bart-Tokenizer* to tokenize the text sequences into smaller units. Similar to the ROBERTa tokenizer (Liu *et al.*, 2019), the BART tokenizer uses byte-level Encoding and has been trained to consider spaces as tokens. As a result, the encoding of a word will vary based on its position within a phrase. In the second section, we utilized the *BartForConditionalGeneration* to generate the summaries. The BART Model, which includes a language modeling head, can be utilized for summarization and was used in this study. This model is built on the pre-trained Model Wolf *et al.* (2020).

Table 5.1 presents a collection of examples showcasing the capabilities of an automatically generated summary system developed using a BART-based deep learning approach. It illustrates the system's proficiency in creating concise and informative summaries for various Android API methods. Each row in the table corresponds to a different API method, specifically focusing on key lifecycle methods and tasks within Android development, such as `app.activity.onCreate`, `os.asyncTask.onPostExecute`, and `app.activity.onCreateView`. For each API method listed, the table provides a detailed summary that encapsulates the method's primary function, its role within the Android lifecycle, or its operational context. The full list of generated summaries has been provided in Table I-2 of the Appendix. The next section explains the steps undertaken to evaluate the automatically generated summaries and the process followed to build the oracle that we have used for comparison purposes.

Table 5.1 Examples of our automatically generated summaries using the BART-based approach

API Methods	**Automatically-generated Summaries**
app.activity. onCreate	OnCreate is the first method in the Activity lifecycle whose job is to create an activity. It is called first time the Activity is created, or when there is a configuration change like a screen rotation. This is the beginning state of an activity lifecycle which its view should be set. Most of the activity initialization code goes here.
os.asyncTask. onPostExecute	This method is called on the thread that created the task. It is called after doInBackground has returned and the result parameter is the value that doIn background has returned. This method is the first method of android activity lifecycle. It helps deliver the calculations that has been done in doInbackground to the main thread.
app.activity. onCreateView	The Fragment.onCreateView() method is responsible for creating and returning a View. This View is displayed in the UI Fragment, so the desired View must be created in this method. You can return null if the fragment does not provide a UI. The FragmentFragment() method creates a Fragment Fragment and returns a View component that is the root of your fragment's layout.

5.3 Empirical Evaluation

In this section, we outline the methodology for evaluating the automatically generated summaries. We created a benchmark by developing an oracle of human-written summaries to assess the performance of the deep learning-based summarization algorithm. The objective was to determine whether the state-of-the-art deep learning summarization technique could effectively generate summaries for API methods discussed in unofficial documentation. To answer the second research question about the impact of deep learning on the quality of generated summaries for API methods, we conducted a statistical comparison between the abstractive summarization approach and the previous extractive approach mentioned in Chapter 4.

5.3.1 Building the Oracle

For evaluation purposes, we needed reference summaries to assess the summaries that our approaches have automatically generated. Consequently, we have built an oracle of human-generated summaries for the investigated API methods. We have used this oracle as gold standard summaries to compare them against our abstractive summaries by applying ROUGE and BLEU metrics. The construction of the oracle and the methods utilized for the analysis are broken down into the sections below.

For building the oracle, two annotators have contributed to this process, *i.e.*, the author of this research and another student unrelated to this research. As shown in Figure 5.2, the process of building the oracle consists of three primary steps: API investigation, summary generation, and summary validation.

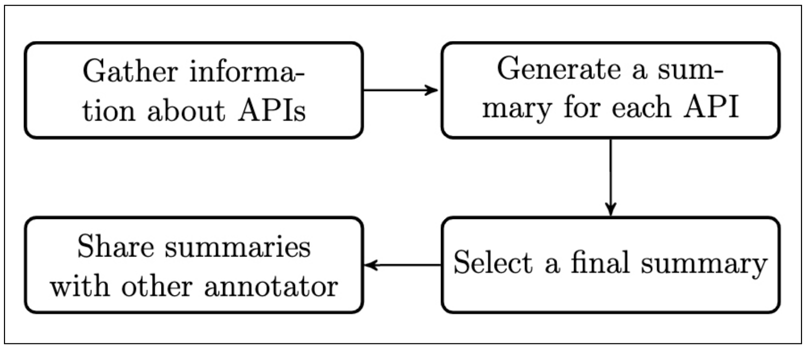

Gather information about APIs
Generate a summary for each API
Select a final summary
Share summaries with other annotator

Figure 5.2 Main steps of the process of building our oracle

At first, each annotator began gathering knowledge about the selected API methods by examining official and unofficial documentation such as Stack Overflow, GitHub, Bug Reports, etc., in order to comprehend the purpose and usage of each API method.

When annotators have sufficient knowledge about an API method, the second step is to compose a summary of that API method based on the obtained information. Based on their knowledge extracted from consulted official and unofficial documentation, each annotator produces a summary for each API method. After two annotators independently write summaries for each API method, the final step consists of discussing the summaries by comparing them and establishing a consensus on the final summary whenever there are disagreements.

The task of summarizing APIs involved a systematic and thorough process that was divided into three distinct steps. The first step required the annotators to gather information about the selected APIs by researching and studying the official documentation and other sources of information such as Stack Overflow, GitHub, Bug Reports, etc. The goal was to gain a comprehensive understanding of the purpose and usage of each API method.

Once the annotators had understood an API method, the second step was to write a summary based on the data that was gathered. Each annotator has generated a summary for each API method, based on their understanding of the information that was investigated.

The final step consists of validating the generated summaries that have been suggested separately by annotators. This was done by comparing the summaries created, for each API method, by both annotators and working together to reach a consensus on the summary that will be part of our benchmark.

The outcome of this process is a comprehensive list of summaries for each API method among the examined ones.

5.3.2 Metrics

The oracle provides us with benchmark summaries to assess the quality of our output, *i.e.*, the automatically generated summaries. Hence, the next step is to select some metrics to evaluate our summaries. In terms of text summarization evaluation, various evaluation approaches have been used to assess the metrics of summarization (Graham, 2015). ROUGE and BLEU

(Yang, Liu, Liu, Lyu & Li, 2018; LeClair *et al.*, 2020) are among the most widely-used metrics. Additionally, they are the standard evaluation metrics leveraged in source code summarization research works as well (Hu *et al.*, 2018b; LeClair, Jiang & McMillan, 2019).

ROUGE or Recall-Oriented Understudy for Gisting Evaluation (Lin, 2004) is the most cited evaluation metric in text summarization (Barbella & Tortora). ROUGE is a set of metrics for evaluating the automatic summarization of texts and machine translations. It compares an automatically generated summary or translation to a list of reference human-generated summaries (M'rabet & Demner-Fushman, 2020). ROUGE focuses on the overlapping of n-grams between the system and human summaries, regardless of semantic or syntactic accuracy (Barbella & Tortora).

ROUGE has various unique versions, including eight n-grams that count method options (ROUGE-1; 2; 3; 4; S4; SU4; W; L), binary settings such as word-stemming of summaries in addition to another option to eliminate or maintain stop-words (Graham, 2015). Hong *et al.* (2014) have proved that ROUGE-2 achieves the strongest correlation with human assessment for the summarization evaluation. Because of the aforementioned reasons, we have been motivated to use this variant of ROUGE in our evaluation.

The Bilingual Evaluation Understudy, also known as BLEU, is a metric used to compare the candidate's translation of a text to one or more reference translations. Even though it was developed for translation purposes, it is also possible to use it to evaluate text output for natural language processing tasks such as paraphrasing and text summarization. The BLEU score strongly correlated with human evaluations of the translated texts produced by natural language processing techniques. Even though BLEU has some drawbacks (Qin & Specia, 2015), it is nevertheless widely used since it is an automated and low-cost metric that can be used to evaluate the quality of translation models (Tran, Tran, Nguyen, Nguyen & Nguyen, 2019). Assuming that human translations accurately convey the original text's meaning, the BLEU score uses a straightforward method for calculating the number of n-grams shared by the system translation and the human translations (Qin & Specia, 2015).

While BLEU makes focuses on the n-gram Precision, *i.e.*, how much of the generated text appears in the reference text, ROUGE emphasizes Recall, *i.e.*, how much of the reference appears in the generated text (LeClair *et al.*, 2020).[1] Therefore, like previous studies (Yang *et al.*, 2018; LeClair *et al.*, 2020), we have used a combination of these two metrics to assess our automatically generated summaries. Specifically, we have used BLEU for calculating the Precision (P_t), while we have leveraged ROUGE to calculate the Recall (R_t). Furthermore, we have also reported the F-measure (F_t) score, which is the harmonic mean of Precision and Recall (van Rijsbergen, 1979):

$$F_t = \frac{2.P_t.R_t}{P_t + R_t} \tag{5.1}$$

We have examined 15 popular API methods in our first investigation based on TextRank 5.2. In this second part of our research, we have extended our dataset to more API methods and investigated new techniques, *i.e.*, the BART algorithm to automatically generate summaries. The metrics' scores for the latter summaries are reported in Table 5.3.

As it can be noticed from Table 5.3, the average Precision is equal to 0.57, while the average Recall is equal to 0.66 and the average F-measure is equal to 0.61. The range of scores varies between 0.39 to 0.93 for Precision, while it varies between 0.38 and 0.87 for Recall, and between 0.38 to 0.86 for F-measure. Based on our observation, API methods with a high number of occurrences in Stack Overflow such as *activity.onCreate* yield higher scores in terms of Precision, Recall, and F-measure compared to other methods because popular API methods have a high number of Stack Overflow discussions associated with them, which is required for a deep learning algorithm like BART in order to generate high-quality summaries. In contrast, API methods with fewer data like *activity.onStop* yield lower scores in terms of precision, recall, and F-measure. This fact shows how a lack of data could affect the results in the context of the code summarization task. In the next section, we use our first suggested approach as a

[1] BLEU and ROUGE scores are between 0 and 1. A score of 0.6 or 0.7 is considered the best you can achieve. typically, values around 0.3 or 0.4 can be considered very good.

Table 5.2 Metrics' scores of the summarization approach based on TextRank suggested in the first section of our research

Method	**Precision**	**Recall**	**F-measure**
asyncTask.onPostExecute	0.3818	0.5908	0.4638
fragment.onCreateView	0.1940	0.3939	0.2599
activity.onCreate	0.1690	0.4	0.2376
asyncTask.doInBackground	0.2413	0.3159	0.2736
activity.onPause	0.1666	0.2391	0.1964
activity.findViewById	0.1791	0.3636	0.2399
activity.onDestroy	0.1408	0.3333	0.1980
activity.finish	0.2222	0.3636	0.2758
activity.setContentView	0.3030	0.5454	0.3896
activity.startActivityForResult	0.2222	0.5128	0.3100
recyclerview.onBindViewHolder	0.1941	0.4878	0.2777
activity.startActivity	0.2941	0.4878	0.3669
activity.onBackPressed	0.4745	0.5284	0.5578
activity.onActivityResult	0.274	0.3285	0.2987
activity.onStop	0.1281	0.1934	0.1541
Average	**0.2504**	**0.4056**	**0.3096**

baseline against which we compare the summarization approach we have proposed in the next part of our research. Our goal is to investigate whether we can improve or not the quality of the automatically produced summaries.

5.3.3 Statistical Test

To address our research questions, we have followed the Basili framework (Basili, Selby & Hutchens, 1986), which consists of four parts: definition, planning, operation, and analysis. This widely used framework has been applied here to structure and organize our empirical study.

Table 5.3 Metrics' scores of the summarization approach using BART suggested in the second section of our research

Method	**Precision**	**Recall**	**F-measure**
activity.onCreate	0.9393	0.7948	0.8611
fragment.onCreateView	0.6418	0.7277	0.6820
asyncTask.onPostExecute	0.6608	0.7448	0.7002
asyncTask.doInBackground	0.6563	0.6237	0.6396
activity.onPause	0.4545	0.4381	0.4461
activity.findViewById	0.875	0.6764	0.7630
activity.onDestroy	0.9324	0.6666	0.7774
activity.finish	0.62	0.5864	0.6027
activity.setContentView	0.5555	0.7963	0.6545
activity.startActivityForResult	0.5	0.475	0.4871
recyclerview.onBindViewHolder	0.75	0.6153	0.6760
activity.startActivity	0.5530	0.6825	0.6109
activity.onBackPressed	0.9318	0.7735	0.8453
activity.onActivityResult	0.4042	0.6551	0.4999
activity.onStop	0.3902	0.3809	0.3855
activity.onStart	0.7741	0.8482	0.8095
adapter.notifyDataSetChanged	0.4258	0.3905	0.4073
adapter.getView	0.6728	0.7231	0.6970
view.onDraw	0.7032	0.8451	0.7676
activity.onSaveInstanceState	0.8027	0.6459	0.7158
activity.onNewIntent	0.5725	0.4286	0.4902
activity.onActivityCreated	0.7394	0.8725	0.8005
activity.onCreateOptionsMenu	0.856	0.6551	0.7421
activity.runOnUiThread	0.53	0.6034	0.5643
asyncTask.onProgressUpdate	0.856	0.7348	0.7907
Average	**0.5718**	**0.6634**	**0.6142**

5.3.3.1 Definition and Planning

The *objective* of our study is to evaluate if the quality of our automatically generated API summaries can be improved compared to our previous unsupervised approach in Chapter 4.

The *quality* focus of our study is the performance of two suggested approaches for API methods summarization, *i.e.*, the TextRank-based approach and the BART-based approach. The performance of each approach will be measured in terms of the precision, recall, and F-measure metrics. By comparing the performance of the two approaches using these metrics, we aim to determine which approach is most accurate when generating summaries for API methods trapped in informal documentation.

The *perspective* is of researchers and developers who need to search for additional information about API methods while carrying out their software engineering tasks.

The *independent variable* of our study is the type of algorithm being applied, it has two factors: the BART-based approach and the TextRank-based approach.

The *dependent variables* of our study are the measures leveraged for assessing the performance of the proposed approaches, *i.e.*, the Precision, Recall, and F-measure. These metrics are frequently used in the field of text summarization to evaluate the relevance of automatically generated summaries.

5.3.3.2 Operation

Our main research questions aim to examine the impact of deep learning algorithms on the relevance of the produced summaries for API methods. To provide a comprehensive answer, we have divided our questions into three sub-research questions that we will address in the following in order to have a more complete understanding of the topic. For each research question, we present its corresponding null and alternative hypotheses :

RQ1: *How do the BART-based approach and TextRank-based approach compare in terms of Precision when summarizing API methods discussed in Stack Overflow?*

- H_{0_1}: There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of Precision when summarizing API methods discussed in Stack Overflow.
- H_{a_1}: There is a statistically significant difference between the BART-based approach and TextRank-based approach in terms of Precision when summarizing API methods discussed in Stack Overflow.

RQ2: *How do the BART-based approach and TextRank-based approach compare in terms of Recall when summarizing API methods discussed in Stack Overflow?*

- H_{0_1}: There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of Recall when summarizing API methods discussed in Stack Overflow.
- H_{a_1}: There is a statistically significant difference between the BART-based approach and TextRank-based approach in terms of Recall when summarizing API methods discussed in Stack Overflow.

RQ3: *How do the BART-based approach and TextRank-based approach compare in terms of F-measure when summarizing API methods discussed in Stack Overflow?*

- H_{0_1}: There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of F-measure when summarizing API methods discussed in Stack Overflow.
- H_{a_1}: There is a statistically significant difference between the BART-based approach and TextRank-based approach in terms of F-measure when summarizing API methods discussed in Stack Overflow.

5.3.3.3 Analysis Method

To answer our mentioned research questions, we have conducted proper statistical tests. We have been interested in investigating whether our second proposed summarization approach could enhance the quality of the automatically generated summaries.

Our method of analysis relies on both descriptive statistics and statistical analyses. We used box plots to present the results for each descriptive statistic (McGill, Tukey & Larsen, 1978), while we leveraged the Shapiro-Wilk normality test (Shapiro & Wilk, 1965) to determine whether or not our data follows a normal distribution. When the p-value exceeds 0.05, the data distribution is not significantly different from the normal distribution.

Table 5.4 Results of the Shapiro-Wilk test for BART-based approach

Precision		**Recall**		**F-measure**	
W	**p-value**	**W**	**p-value**	**W**	**p-value**
0.9536	0.3031	0.9367	0.1242	0.9418	0.1636

Table 5.5 Results of the Shapiro-Wilk test for TextRank-based approach

Precision		**Recall**		**F-measure**	
W	**p-value**	**W**	**p-value**	**W**	**p-value**
0.8951	0.08014	0.9615	0.7188	0.9139	0.1559

The results of the Shapiro-Wilk normality test are reported in Tables 5.4 and 5.5. As you can notice, the p-value for all performance measures is more than 0.05. Therefore, both algorithms seem to follow a normal distribution. Consequently, we had to use a parametric statistical test on our data. For such a purpose, we have applied the paired T-test (Student, 1908) statistical test and we have proceeded as follows. In Chapter 4, we have used a qualitative metric, *i.e.*, the survey, for evaluating our automatically generated summaries. However, to be able to compare the previous approach with the new one, we need first evaluate it with quantitative metrics. We have computed the Precision, Recall, and F-measure of the approach that we have previously proposed *i.e.*, the one based on TextRank.

Then, we have compared the two approaches in terms of their performance by applying the T-test. In the following, we present the null and alternative hypotheses associated with our main research question: *Does deep learning algorithm improve the quality of previously generated summaries?*.

The paired T-test (Student, 1908) is a parametric test for pair-wise median comparison. The T-test determines whether or not the median difference between the two compared approaches is zero. The results are interpreted as statistically significant at $\alpha = 5\%$. Then parametric t-score and p-value have been calculated to determine the magnitude of the difference between the two compared approaches, *i.e.*, the BART-based and TextRank-based approaches. A t-score of 0 indicates that the sample results equal the null hypothesis. As the difference between the sample data and the null hypothesis increases, the absolute value of the t-score increases. Larger t-scores mean more difference between groups, while smaller t-scores mean more similarity between groups. The p-value is the variable that allows us to reject the null hypothesis (H_0) or, in other words, to establish that the two groups are different. A p-value greater than α shows failure to reject the statistical test's null hypothesis. Otherwise, we can reject the null hypothesis of the statistical test.

5.4 Results of our Empirical Investigation

In this section, we present the findings of our comparison of the BART-based and TextRank-based approaches.

Figure 5.3 shows the performance of the BART and TextRank-based approaches in terms of precision. As we can notice, the BART-based approach performs better than the TextRank-based approach in terms of Precision. The average precision for the TextRank-based approach is 0.22 while for the BART-based approach is 0.64.

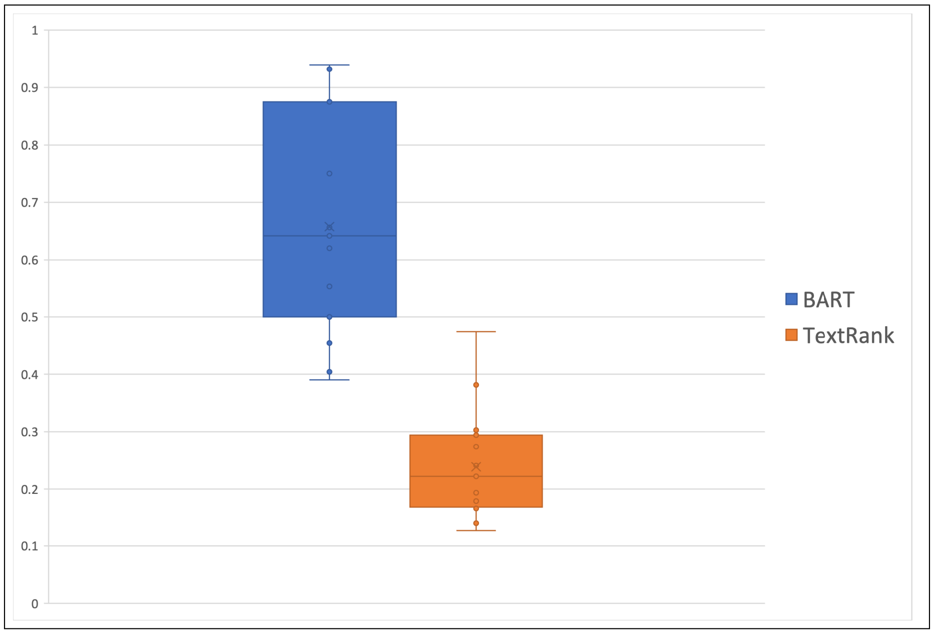


Figure 5.3 Boxplots of precision for the BART-based and TextRank-based approaches

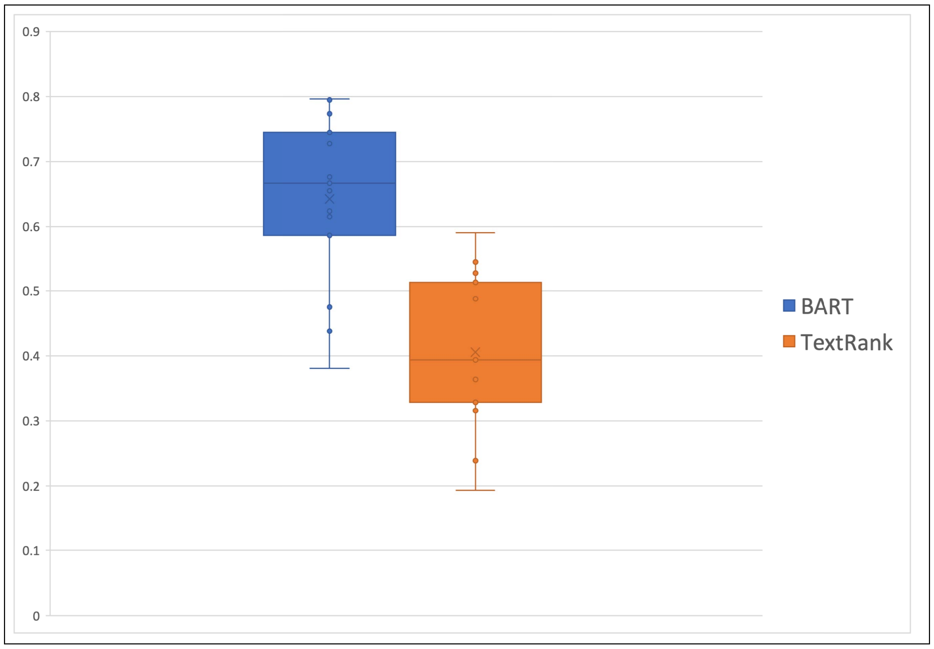


Figure 5.4 Boxplots of recall for the BART-based and TextRank-based approaches

Figure 5.4 shows the performance of the BART and TextRank-based approaches in terms of Recall. Likewise, the BART-based approach seems to perform better than the TextRank-based approach in terms of Recall. The average recall for the TextRank-based approach is 0.396 while for the BART-based approach is 0.667.

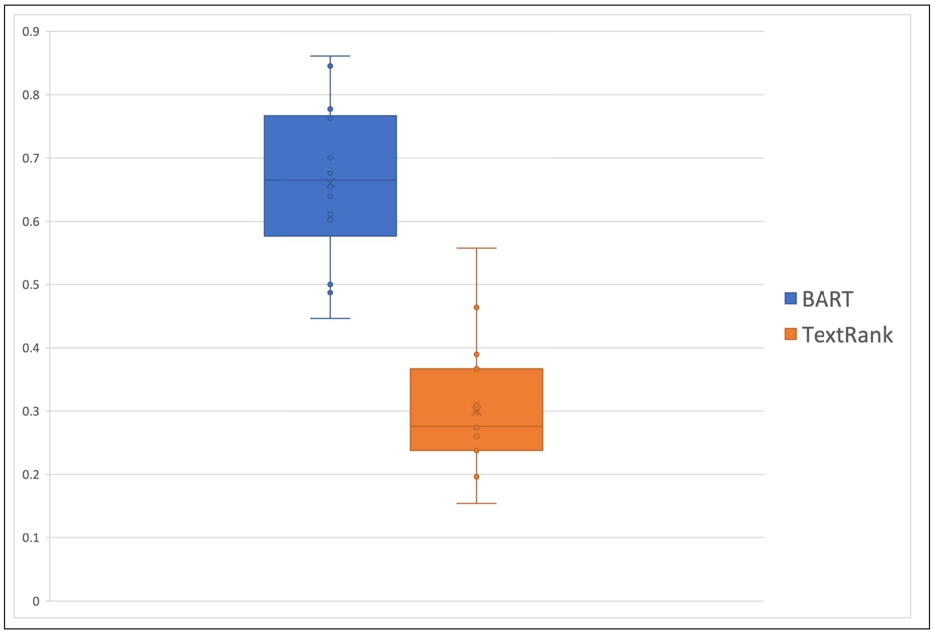


Figure 5.5 Boxplots of F-measure for BART-based and TextRank-based approaches

Additionally, Figure 5.5 demonstrates their performance in terms of F-measure. As we can notice from Figure 5.5, the BART-based approach seems to perform better than the TextRank-based approach in terms of F-measure as well. The average F-measure for the TextRank-based approach is 0.28 while for the BART-based approach is 0.672.

Overall, the box plots show that the BART-based approach performs, generally, better than the TexRank-based approach in terms of Precision, Recall, and F-measure.

While Boxplots remains a simple way to illustrate the results visually, we need rigorous statistical tests to investigate if there is significant differences between our approaches.

Table 5.6 Results of the comparison performed between the BART and TextRank-based approaches

Precision		**Recall**		**F-measure**	
t-score	**p-value**	**t-score**	**p-value**	**t-score**	**p-value**
8.9454	3.64E-07	9.0516	3.16E-07	10.9914	2.86E-08

Table 5.6 presents the results of the statistical test applied for comparing the BART-based and TextRank-based approaches, *i.e.*, the T-test statistic. As indicated in the table, for the Precision, the t-score is 8.9454, which is greater than the t-critical value, which is 1.761 for this statistical test. This finding reveals that the difference, in this case, is significant. Moreover, the p-value for Precision is less than α = 0.05. Therefore, we can reject the null hypothesis of **RQ1:** *There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of precision when summarizing API methods discussed in Stack Overflow.* Overall, we conclude that the BART-based and TextRank-based approaches have significant differences in terms of Precision.

Regarding Recall, Table 5.6 indicates that the t-score is 9.05, which is greater than our t-critical (1.76), and also the p-value for Recall is less than α = 0.05. Similarly to the finding obtained with the Precision, we could reject the null hypothesis of **RQ2:** *There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of Recall when summarizing API methods discussed in Stack Overflow.* and conclude that BART and TextRank-based approaches have significant differences in terms of Recall.

Considering the F-measure, we have obtained a similar finding. In effect, as shown in Table 5.6, the t-score is 10.99, which is greater than the t-critical value (*i.e.*, 1.761) and the p-value For F-measure is less than α = 0.05. Therefore, we could reject the null hypothesis of **RQ3:** *There are no statistically significant differences between the BART-based approach and TextRank-based approach in terms of F-measure when summarizing API methods discussed in Stack Overflow* and conclude that BART and TextRank-based approaches have significant differences in terms of F-measure.

Our T-test statistical results show that there are significant differences between the BART-based and TextRank-based approaches in terms of precision, recall, and F-measure. Besides these performance scores, we have also compared the BART and TextRank approaches in terms of their execution time. Table 5.7 shows the execution time of both approaches for the frequently examined API methods summarized using both approaches. As you can notice from the findings reported in Table 5.7, the average execution time for the BART-based approach is 145.64 seconds, while the average execution time for the TextRank-based approach is 641.29 seconds. This means that the BART-based approach could perform 4.4x faster compared to the TextRank-based approach considering the same system configuration (Processor: Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8, Memory:16 GB). It is worth mentioning that usually, deep learning algorithms take a considerable time to run because of their training time (Blumenthal, Luo, Schilling, Holme & Uecker, 2023). However, since we have used a pre-trained model of BART, the training time is not included in Table 5.7. This is why BART, a deep learning algorithm that requires training time, runs faster than TextRank, which is an unsupervised summarization machine learning algorithm.

5.5 Discussion

In this research, we have proposed a novel approach to generate summaries for API methods discussed on informal documentation based on their surrounding context. Like our previous approach, we have leveraged Stack Overflow as an unofficial source of documentation and used Android API methods discussed in Stack Overflow posts as input for our approach. Differently from the TextRanked-based approach in Chapter 4, we have applied BART, a pre-trained deep learning algorithm, to summarize code entities.

Our empirical investigation has involved generating summaries for a set of twenty-five popular Android API methods discussed in Stack Overflow posts. Additionally, we have used the

Table 5.7 Performance of the BART and TextRank-based approaches in terms of execution time.

Method	**BART(sec)**	**TextRank(sec)**
asyncTask.onPostExecute	184.81	739.63
fragment.onCreateView	126.88	583.75
activity.onCreate	269.92	1435.85
asyncTask.doInBackground	113.81	437.92
activity.onPause	123.91	585.05
activity.findViewById	114.37	446.65
activity.onDestroy	123.42	547.29
activity.finish	125.90	586.11
activity.setContentView	123.79	485.70
activity.startActivityForResult	131.08	490.42
recyclerview.onBindViewHolder	136.85	459.36
activity.startActivity	153.03	556.74
activity.onBackPressed	142.70	655.53
activity.onActivityResult	188.41	716.73
activity.onStop	125.33	892.73
Average	**145.64**	**641.29**

ROUGE and BLEU metrics to evaluate the quality of our automatically generated summaries. Then, we compared the produced summaries to an oracle of human-generated summaries, which have been manually generated by two annotators. The oracle was used as the gold standard for comparison against the automatically generated summaries.

Our findings have shown that the BART-based approach could generate summaries with an average of 0.57 Precision, 0.66 Recall, and 0.61 F-measure. This finding could answer our first research question: *Can our deep-learning-based approach be able to automatically generate summaries for API methods discussed in informal documentation?* These findings demonstrate that a deep learning-based approach is able to generate summaries for API methods mentioned in informal documentation.

Considering our second research question: *How does our deep learning-based approach compare to a baseline when summarizing API methods discussed in informal documentation?*. The results of our empirical study have indicated that the BART-based approach performs better than the baseline in terms of precision, recall, and F-measure. In effect, it yields an average of 0.57 precision, 0.66 recall, and 0.61 of F-measure, while the baseline shows an average of 0.25 precision, 0.40 recall, and 0.30 of F-measure. Additionally, in terms of execution time, the BART-based approach takes an average of 145.64 seconds to generate a summary while the baseline, takes an average of 641.20 seconds.

These findings could be justified by the difference in the nature of the applied algorithms in both approaches, *i.e.*, BART and TextRank. In fact, BART is a state-of-the-art deep learning algorithm that provides abstractive summaries (Lewis *et al.*, 2020), which, as shown by Hsu *et al.* (2018), are more consistent and come closer to human-generated summaries. TextRank is an unsupervised algorithm that generates extractive summaries. This is more likely why the BART-based approach yields overall better results than the TextRank-based approach because it provides a more concise and to-the-point summary, capturing the essence of the original text.

During our investigation, we have found that some API methods such as *activity.onStop* have lower scores in terms of precision, recall, and F-measure than other API methods in both BART-based and TextRank-based approaches. Based on the results of our investigation, we have concluded that the algorithms we used are not suitable for generating high-quality summaries for APIs with limited data in our corpus. Our findings indicate that the quality of the summaries produced by the algorithms decreases when the number of data in the corpus is low. This highlights the importance of having a sufficient amount of data for these algorithms to work effectively. For instance, we have collected 34,562 posts from Stack Overflow for the API method *asyncTask.onPostExecute*, but we could only collect 3,389 posts for the API method *activity.onStop*. Therefore, a lack of sufficient data may result in code summaries that are not close enough to human-generated summaries. This is particularly true in the case of the TextRank-based approach, which generates extractive summaries, *i.e.*, summaries that are a set of sentences extracted from different posts, which may or may not be related to one another.

Clearly, the software engineering research community and practitioners working in the area of code summarization would be interested in automatic approaches that perform well in terms of relevance and execution time as the case for the BART-based approach. In addition, the deep learning-based approach can be 4.4 times faster than the TextRank-based approach, while improving the relevance of produced summaries by an average of 57 percent in terms of Precision, 66 percent in terms of Recall, and 61 percent in terms of F-measure.

5.6 Threats to Validity

Despite the extensive efforts to ensure the validity of our research study, it is still susceptible to various threats to validity, which can potentially impact the results and inferences made from the study. By acknowledging these potential threats and taking appropriate measures to control them, we can increase confidence in the validity of our findings and the generalizability of our conclusions. The various types of validity, such as external, reliability, and conclusion validity, are considered in our study to ensure that our results are valid and trustworthy.

5.6.1 External Validity

External validity in research is the ability to generalize the results of a study to other populations, settings, or situations beyond the specific sample and conditions in which the study was conducted. It refers to the real-world relevance and applicability of a study's findings (Godwin *et al.*, 2004).

In our investigation, we have used Stack Overflow as an unofficial type of documentation and examined Android API methods discussed in Stack Overflow posts. We have made these specific selections because they are widely used and well-known, but our approaches are not limited to a specific kind of data. It has the flexibility to be adapted to other programming languages and alternative forms of documentation that may not be officially recognized. The goal is to provide a versatile and comprehensive solution that can cater to a variety of needs, regardless of the specifics of the project. This adaptability makes our suggested approaches a valuable tool for a range of use cases and research endeavors.

5.6.2 Reliability Validity

Reliability validity in research measures the consistency and stability of a study's results over time. It assesses the extent to which a study produces the same results when repeated using the same methods and conditions (Godwin *et al.*, 2004).

To evaluate the accuracy of the automatically generated summaries for the examined API methods, we have used the Python package *rouge* to calculate the ROUGE score, which is a metric for evaluating text summarization algorithms by comparing the similarity between a machine-generated summary and a reference summary written by a human. Furthermore, we have used the *nltk* package to calculate the BLEU score that is widely used to evaluate the quality of machine translation models and to compare their performance.

We make available the information and data that may be required for replicating our study under the online appendix[2]. This appendix serves as a comprehensive resource for researchers and–or practitioners interested in replicating our research studies.

[2] https://github.com/DataMining2022/summarization

CHAPTER 6

ENHANCING API DOCUMENTATION: A COMBINED SUMMARIZATION AND TOPIC MODELING APPROACH

In this chapter, we present the second part of our research, which addresses *RQ4: What are the prevalent topics discussed on Stack Overflow related to Android APIs?* and *RQ5: Can summarization methods first identify frequent issues within topics and subsequently derive solutions for these issues?* mentioned earlier in Chapter 2. This chapter is based on our research paper entitled *Revolutionizing API Documentation through Summarization* that has been accepted in The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (Naghshzan & Ratte, 2023b). In this chapter, we utilize BERTopic to extract and identify frequent problems and solutions discussed within Android APIs, drawing from Stack Overflow discussions as a valuable source of informal documentation. Furthermore, we employ extractive summarization techniques to generate concise and informative summaries for these identified topics. To assess the quality of our results, we had Android developers evaluate the summaries based on performance, coherence, and interoperability, providing valuable feedback on the practicality of our approach.

6.1 Introduction

The growing availability of textual data, especially in software development, offers both challenges and opportunities for efficient information extraction. This is particularly relevant in the context of complex API documentation, which is essential for programmers but can be lengthy, intricate, and occasionally incomplete. Official documentation, while valuable, can also be time-consuming to navigate. Ponzanelli *et al.* (2014) recognized challenges in utilizing official documentation due to its often static and comprehensive nature, which can make it difficult for developers to quickly find solutions to specific problems they encounter during software development. Official documentation may not always address the practical, context-specific issues developers face, leading to a gap between the available documentation and the immediate needs of developers for actionable insights and solutions. This recognition underpins the motivation for tools, which aim to bridge this gap by integrating community-driven discussions and solutions directly into the development environment, providing more accessible and relevant support for developers.

Developers often turn to unofficial sources like Stack Overflow and GitHub for quicker access to information, highlighting the need for a more user-friendly approach to extracting insights from API documentation. To address this, our research introduces a novel method that utilizes BERTopic (Grootendorst, 2022), a topic modeling technique that combines BERT embeddings (Devlin *et al.*, 2019) and c-TF-IDF, for topic extraction and identifies frequent problems and solutions on Android APIs as a case study. Then, we use extractive summarization to generate summaries for these topics and provide insightful documentation.

Our approach involves data gathering, preprocessing, employing BERTopic modeling for topic extraction, and utilizing extractive summarization. We assess the resulting summaries and topics for their performance, coherence, and interpretability, all intending to make API and method comprehension easier for developers (Naghshzan & Ratte, 2023a).

To guide our research, we aim to answer the following research questions:

Q1: What are the prevalent topics discussed on Stack Overflow related to Android APIs?

Q2: Can summarization methods effectively identify frequent issues within these topics?

Q3: Is it feasible to derive solutions for these frequent issues by leveraging information from unofficial documentation sources?

The goal is to assist developers in understanding relevant API methods for their daily tasks. This research could lead to valuable tools like IDE plugins or recommender systems for practical use by researchers and developers.

6.2 Methodology

Our research methodology consists of three main steps: data collection, topic modeling, and summarization. In the following section, we delve into a comprehensive examination of these key processes.

6.2.1 Data Collection and Pre-processing

To ensure consistency across our research and to enable comparison of our results, we used the dataset generated in previous chapters for this part of our research as well. Here, we briefly explain the key points to recapitulate the steps. We utilized the Stack Exchange API to retrieve all questions on Stack Overflow tagged with *Android* from January 2009 to April 2020. We chose Android because it ranks among the top eight most frequently discussed topics on Stack Overflow (Naghshzan *et al.*, 2021). Additionally, our first author has expertise in Android development, which made it a more convenient choice for us to understand various APIs and compare outcomes.

We obtained all the corresponding answers to those questions, resulting in a dataset of 3,698,168. unique Android posts. Since we were interested in natural languages for topic modeling and summarization, we needed to remove code blocks from posts. To do this, we used the *Code Snippet* feature of Stack Overflow. This feature highlights code segments by wrapping them with an HTML code tag (<code>), making them more accessible to users. By searching for this tag in the posts, we were able to identify code blocks and remove them.

To prepare our data for further analysis, we performed the following pre-processing steps:

- Elimination of stop words and punctuation.
- Tokenization of text into individual words.
- Lemmatization and stemming.
- Removal of special characters and numerical values.

6.2.2 Topic Modeling

To identify the topics discussed in Stack Overflow, we utilized the Bidirectional Encoder Representations from Transformers Topic Modeling (BERTopic)(Grootendorst, 2022). BERTopic is a powerful topic modeling technique that leverages the capabilities of pre-trained transformer models, particularly BERT (Devlin *et al.*, 2019). Traditional topic modeling techniques, such as Latent Dirichlet Allocation (LDA) (Jelodar *et al.*, 2019), have been widely used to discover latent topics within a corpus of documents. However, these techniques often struggle to capture the semantic meaning and context of words, leading to less accurate and less interpretable topic representations (Egger & Yu, 2022). BERTopic overcomes these challenges by integrating the contextual information encoded within BERT's transformer architecture.

To implement BERTopic, we utilized a pre-trained model available on Hugging Face[1]. Notably, this model was trained on a vast corpus exceeding one million Wikipedia pages. For the computations, we relied on Google Colab Pro, which comes equipped with a T4 GPU, boasting 25GB of VRAM and 26GB of system RAM. We further leveraged cuML, thereby enabling GPU-accelerated machine learning.

Figure 6.1 shows word scores for the top 16 topics, each characterized by a set of keywords that define the semantic field of the topic. The scores indicate the importance or relevance of specific words within each identified topic. Higher word scores suggest that the word is more representative or characteristic of a particular topic. The keywords for each topic are displayed alongside horizontal bars, the lengths of which presumably represent the relative importance or frequency of the keyword within the topic. For example, Topic 0 is dominated by terms such as "studio," "gradle," and "emulator," suggesting a focus on software development environments. Similarly, Topic 4 features keywords like "camera" and "video," hinting at a topic related to multimedia technologies. The colorful representation offers a quick visual summary of the topics, providing insights into the thematic structure of the dataset the algorithm was applied to, which appears to be related to technological and programming domains.

[1] https://huggingface.co

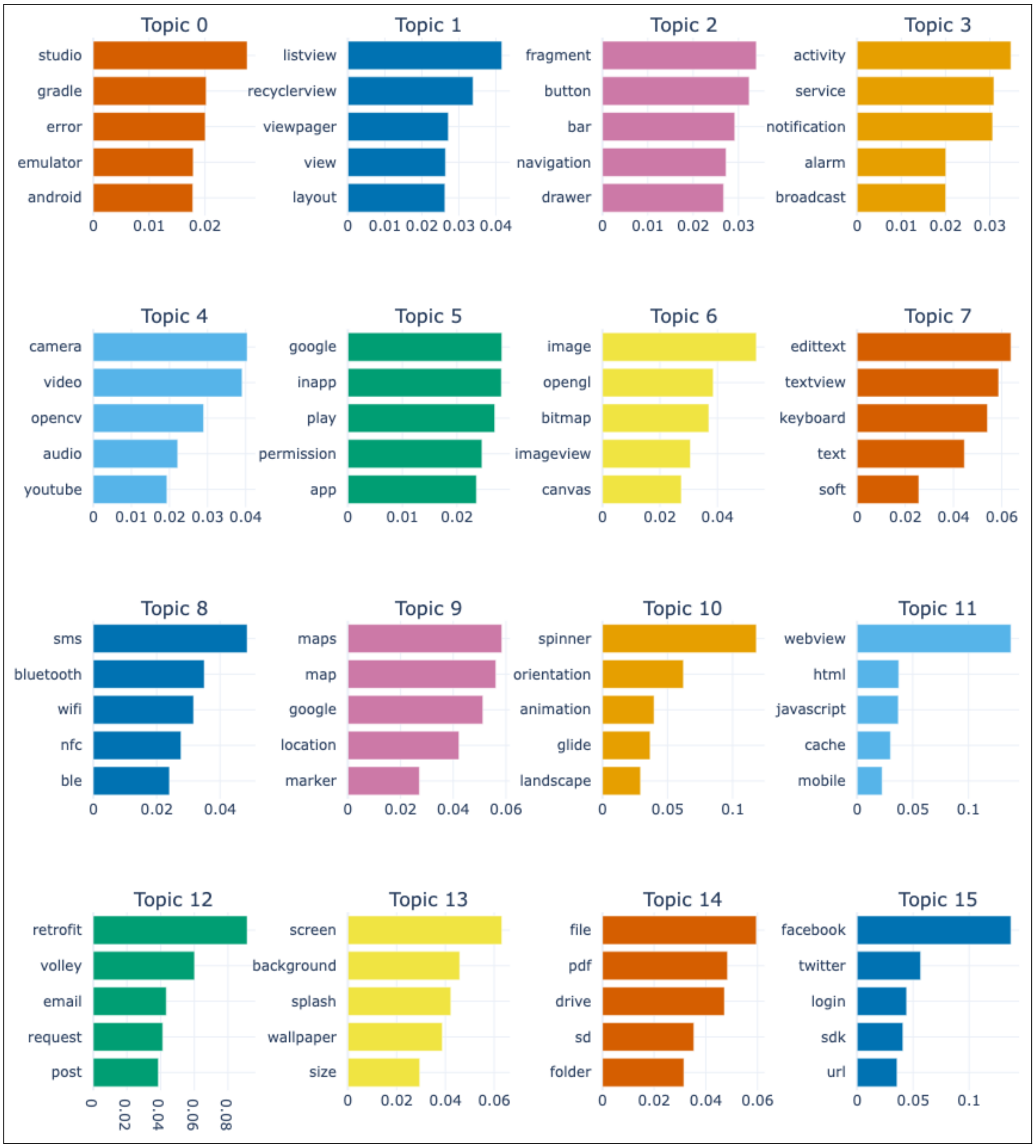


Figure 6.1 Word scores of Stack Overflow's Android topics

BERTopic uses clustering to group documents with similar content into topics. It first transforms texts into numerical embeddings to capture their meanings. Clustering these embeddings allows BERTopic to identify and organize topics based on semantic similarities, making it efficient in handling and categorizing large datasets. Figure 6.4 presents a dendrogram, a tree-like diagram that illustrates the arrangement of the clustered topics extracted from Stack Overflow discussions about Android APIs. The x-axis possibly measures the distance or dissimilarity between clusters, whereas the y-axis represents the various topics, which are labeled with keywords summarizing their content.

Figure 6.2 presents a bi-dimensional scatterplot visualizing the clustering of topics derived from Stack Overflow discussions related to Android APIs. Each point on the graph represents a distinct topic, with its size correlating to the frequency of discussions within that topic. The axes, labeled D1 and D2 denote two principal components obtained through dimensionality reduction using the Uniform Manifold Approximation and Projection (UMAP) technique allowing for the visualization of topic distribution in a reduced space(McInnes, Healy & Melville, 2020). The positioning of the circles, which symbolize individual topics, indicates their relationship to one another; topics that are closer together are more similar, suggesting they share a significant number of keywords or documents.

The varying sizes of the circles could imply the number of documents associated with each topic, with larger circles representing more prevalent topics. The clustering of circles in certain areas highlights the concentration of topics within the corpus, whereas isolated circles point to unique, less related topics. This visualization aids in comprehending the overall structure and distribution of topics within a large dataset, providing intuitive insights into the thematic composition and the relationships between different topics uncovered from the text data. By analyzing these data points, our study harnesses advanced natural language processing and machine learning techniques, notably BERTopic, to synthesize concise and insightful summaries that encapsulate prevalent issues, typical use cases, and potential solutions—thereby revolutionizing the way developers engage with API documentation.

The Figure 6.3 depict a series of zoomed-in views of topic clusters, showcasing how topics that seem to overlap at a higher level (Figure 6.2) are in fact distinct upon closer examination. The initial image (a) suggests overlapping topics, but progressing to the fourth image (d)

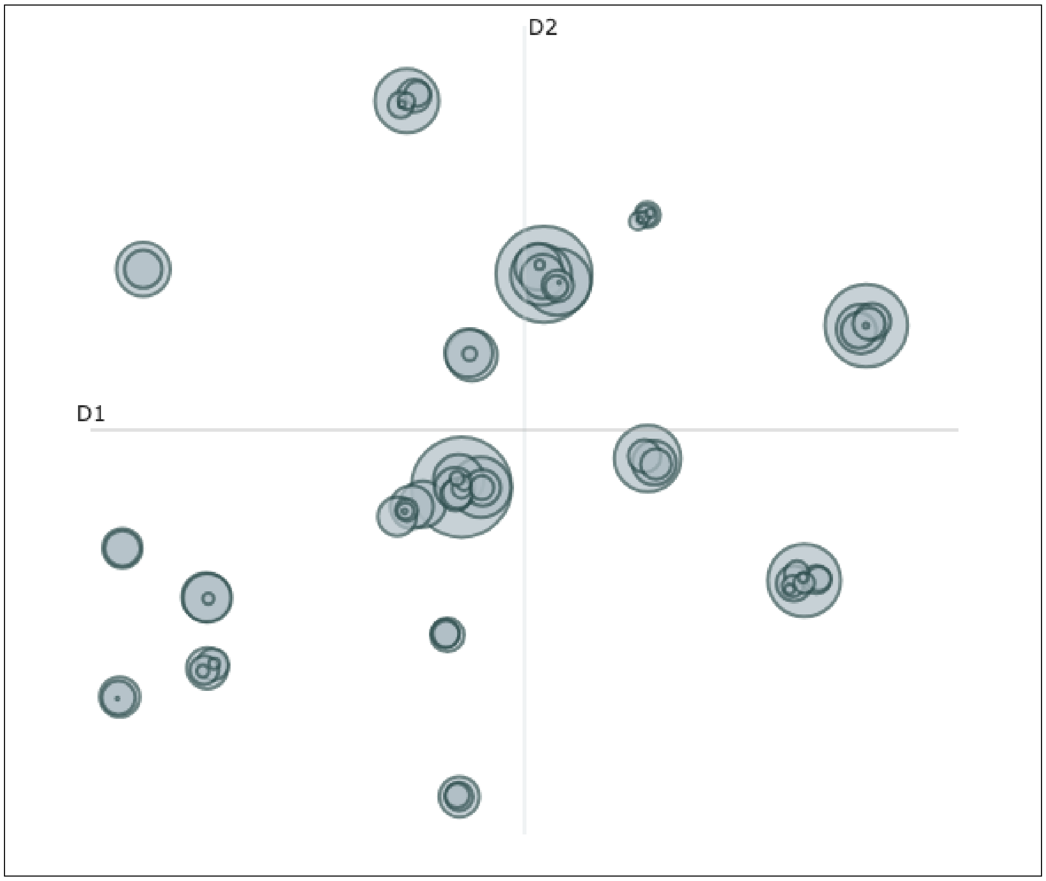
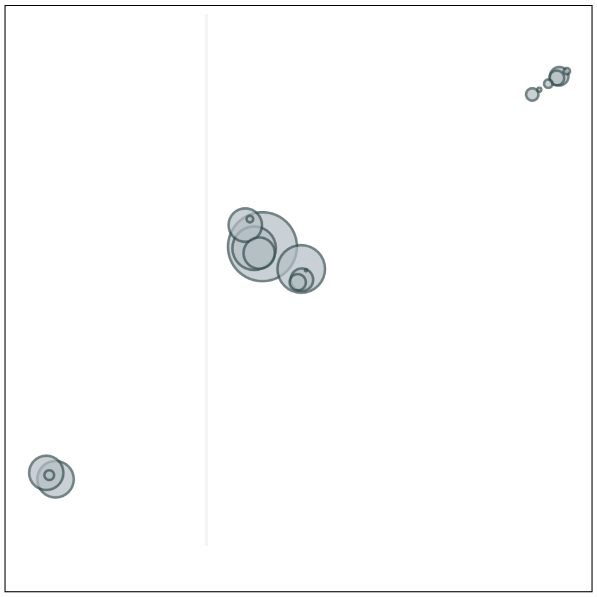
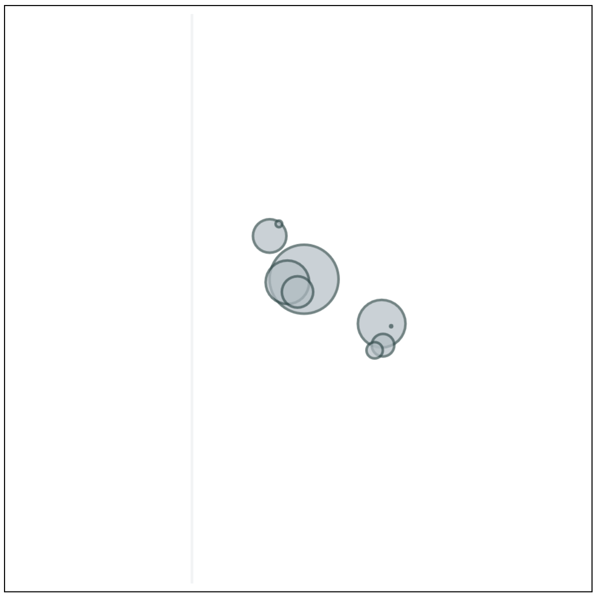


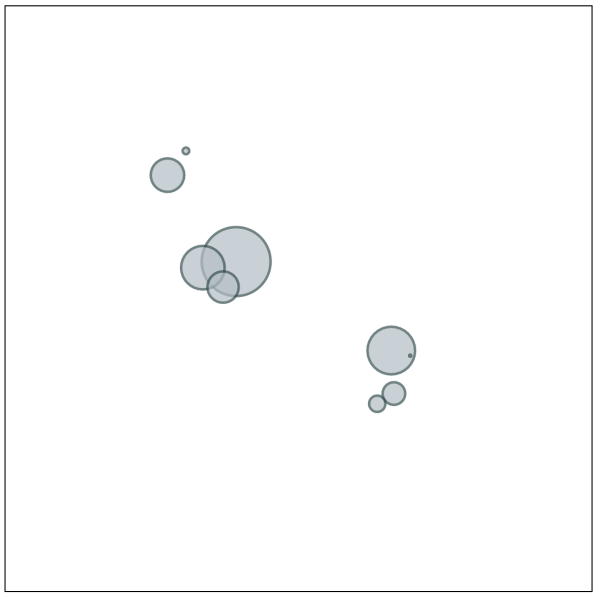
Figure 6.2 Intertopic Distance Map of Stack Overflow's Android topics

zooming reveals that while the topics are closely related, they do not overlap, maintaining their individuality. This nuanced visualization highlights the subtle semantic relationships between different topics without conflating them, demonstrating the effectiveness of BERTopic in differentiating between closely related thematic content.

Figure 6.4 is the result of hierarchical clustering, a method that seeks to build a hierarchy of clusters. The branching style indicates how topics are related to one another based on the similarity of their content, with the height of the branches representing the distance between clusters. Topics that are more closely related, such as *studio_gradle_error* and *file_gdrive_native*, are grouped under lower branches, suggesting a closer relationship or more significant overlap in content.


(a) 2x zoom


(b) 4x zoom


(c) 8x zoom

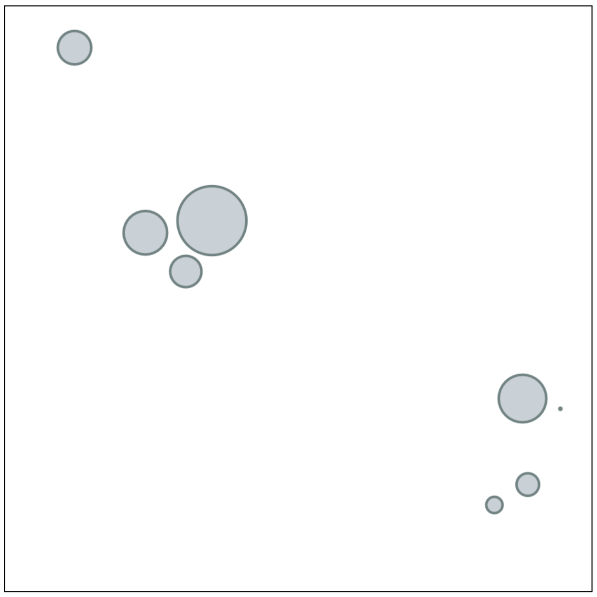

(d) 16x zoom

Figure 6.3 Progressive zoom reveals distinct topic clusters in BERTopic visualization

In our research, we use such visualizations to identify frequent themes and challenges faced by developers, which are then subject to further analysis through BERTopic modeling and extractive summarization techniques. These methods enable us to distill complex discussions into essential insights and, consequently, to generate comprehensive and actionable summaries of API documentation. The dendrogram not only aids in the visual exploration of topic relations but also serves as a guide for the subsequent summarization process, ensuring that the most significant and related topics are addressed in our effort to streamline the comprehension of API documentation for software development.

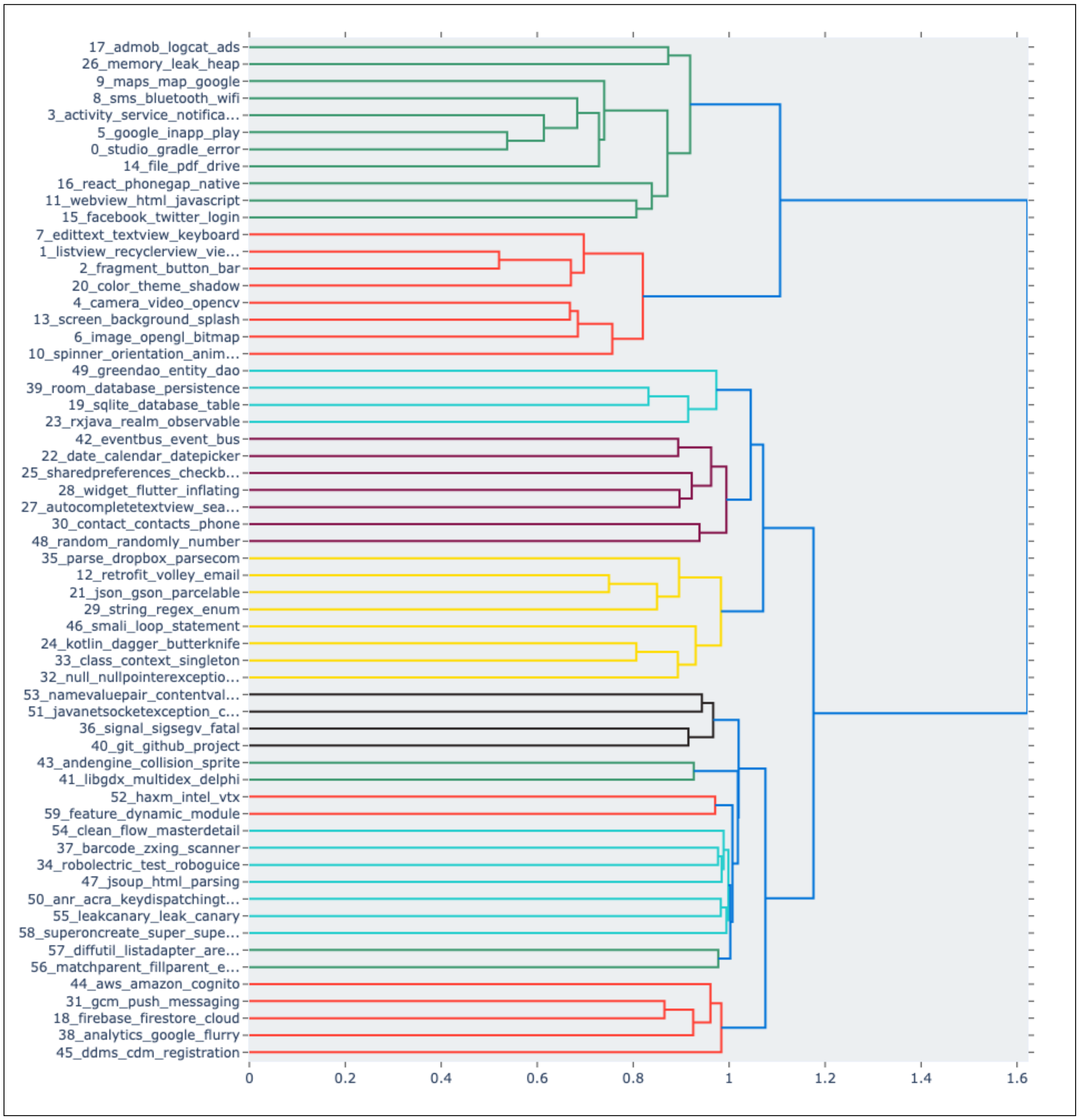


Figure 6.4 Hierarchical Clustering of Stack Overflow’s Android topics

6.2.3 Summarization

In Chapter 4 we demonstrated that extractive summarization can be employed to extract information from Stack Overflow. Therefore, we utilize the extractive version of BERT (Miller, 2019). To implement this, we tokenize the input document with the BERT tokenizer and feed the tokens into the BERT model, which encodes contextual information and captures semantic meaning. The summarization algorithm, based on BERT, calculates importance scores for each sentence or phrase, selecting the highest-scoring ones for the summary.

Our methodology is meticulously designed to tackle the multifaceted challenges developers face when working with APIs. By delving into the intricacies of these challenges, our aim is to highlight prevalent issues and articulate effective solutions. This endeavor unfolds through a systematic, two-phased approach: the identification of frequent problems followed by the proposition of potential solutions.

Phase 1: Identifying Frequent Problems

In this phase, our strategy is to distill the essence of frequent issues encountered in API usage through a novel summarization technique that leverages the power of questions. We believe that by synthesizing the core topics into question-based summaries, we can unearth the most pressing concerns and challenges within each domain. Specifically, we target Stack Overflow, focusing on the most discussed topics within the Android development community. To ensure the relevance and quality of our analysis, we impose a stringent criterion for inclusion: only questions that have garnered an upvote score of 3 or higher are considered. This benchmark is derived from our analysis of the dataset, where we found the average upvote score for all Android-related questions to be 2.87.

Leveraging the advanced capabilities of the BERTopic algorithm, we identify pertinent posts related to each topic. Subsequently, these selected posts undergo a summarization process powered by BERT. This dual application of BERT and BERTopic algorithms allows us to capitalize on their respective strengths, significantly enhancing our ability to document and understand the challenges in Android development.

Phase 2: Outlining Potential Solutions

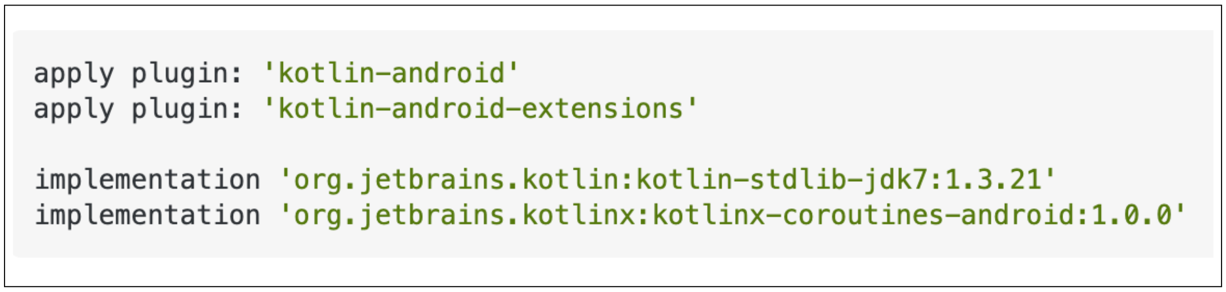
The subsequent phase is dedicated to formulating summaries of potential solutions to the previously identified issues. This is achieved by harnessing the collective expertise and insights of the Stack Overflow community. Our process begins with the identification of questions that relate to the challenges uncovered in Phase 1. We meticulously record the IDs of these questions, which serves as a gateway to access the corresponding answers. In our quest for the most informative and reliable solutions, we prioritize answers that have been marked as accepted by the community or those with a score surpassing the average, typically a threshold of 2 or more.

These selected answers are then processed through the same BERT-based summarization technique employed in the first phase. This methodological consistency ensures that our summaries of potential solutions are not only comprehensive but also directly aligned with the problems they aim to address. Through this rigorous process, we are able to distill actionable insights and recommendations that can significantly alleviate the challenges faced by developers in the realm of API utilization.

In addition to the generated summaries, we tried to provide available code blocks to help users implement the potential solution. To do this, in the candidate answer that BERT used for generating the summary, we looked for code blocks. If there were any, we included that code block as part of the solution. For instance, Figure 6.5 is a code block for the first solution mentioned in Table 6.1.

Table 6.1 Generated summaries for questions and answers of topic project_error_build_gradle

Problem	Solution
I am trying to add Kotlin sources of an AAR in Android Studio 3.3.2. It doesn't work when I select "Choose Sources" and choose the corresponding source.jar. Android Studio only displays "Attaching" but nothing happens.	Configure kotlin in your project following these steps in build.gradle. Also add classpath org.jetbrains.kotlin gradle plugin clean and build your project it worked for me.
Jenkins tries to launch tools instead of emulator I'm trying to set up jenkins ui tests and it fails on running emulator command	I'm answering my own question its an issue with android emulator plugin 3.0 not working with "Command line tools only" sdk package.
I have an sdk works with google api and implementing a lot of dependencies. then I implement the lib into my new app which is also implements dependencies everything goes fine until I try to run the app. When i run (on device) my project I get the following error message: Program type already present	the problem is that you have a duplicated dependency with different version if you can find the implementation that needs com.google.android.gms.location you can put under it exclude com.google.android.gms or simply in build.gradle module


```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.3.21'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.0'
```

Figure 6.5 Code block suggestion for a potential solution.

6.3 Results

We identified 81 topics for Android discussions on Stack Overflow. As shown previously in Figure 6.1, the first topic with the highest number of occurrences is *project_error_build_gradle* with 14,663 posts on Stack Overflow. Furthermore, Table 6.1 presents sample summaries for the mentioned topic, highlighting three frequent problems for each topic and offering potential solutions for the problems. A comprehensive list topics and of our findings can be accessed in the online Appendix (Naghshzan, 2023a).

6.4 Empirical Evaluation

As we aim to ultimately equip developers with tools that assist them in swiftly comprehending APIs relevant to their work, we've undertaken an empirical study to gather insights regarding developers' opinions on the quality of automatically generated summaries and the effectiveness of our approach.

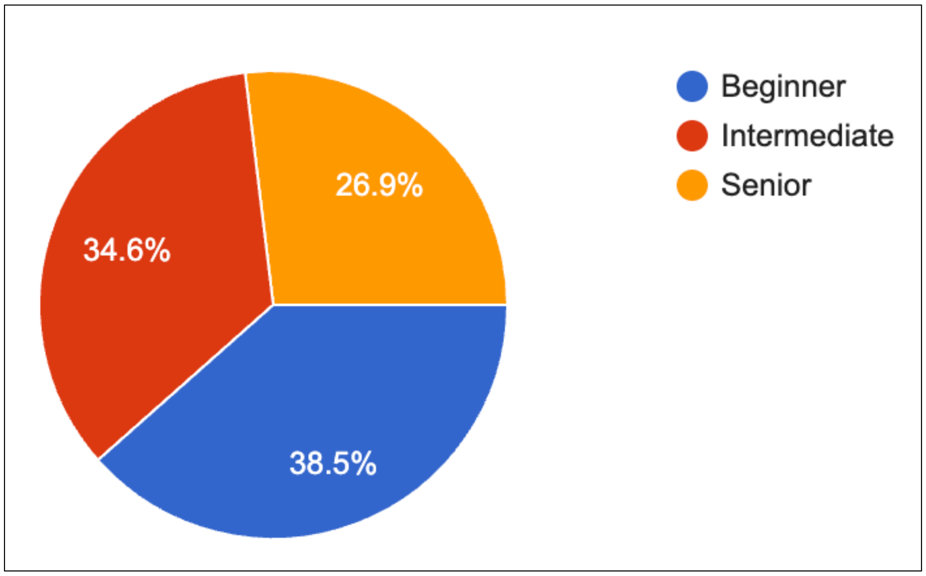


Figure 6.6 Participants' Android programming skill levels

We conducted a user study with 30 Android developers to evaluate our API documentation summarization approach. Sixteen participants were working professionally in the industry, while the rest were in academia, either as students or faculty. The participants were categorized into three levels of Android development skills, as shown in Figure 6.6. Based on this factor, we employed a completely randomized design (CRD) for our study. Each participant evaluated three Android problems and their corresponding solutions. The survey's objectives were to assess:

- Whether the generated problems were related to the identified topics.
- The quality of the generated problems and to evaluate their understandability.
- Whether the potential solutions could effectively address the problem.

The study found that the BERTopic algorithm effectively identified topics and related posts. Participants, on average, rated the summaries as 3.8 for coherence, 3.7 for informativeness, and 3.9 for user satisfaction. Additionally, with a rating of 4.2 out of 5, participants expressed that the tool could be useful in their development routines. These findings suggest that the approach has the potential to aid developers in comprehending complex API documentation. The full survey form is accessible in the online Appendix (Naghshzan, 2023a).

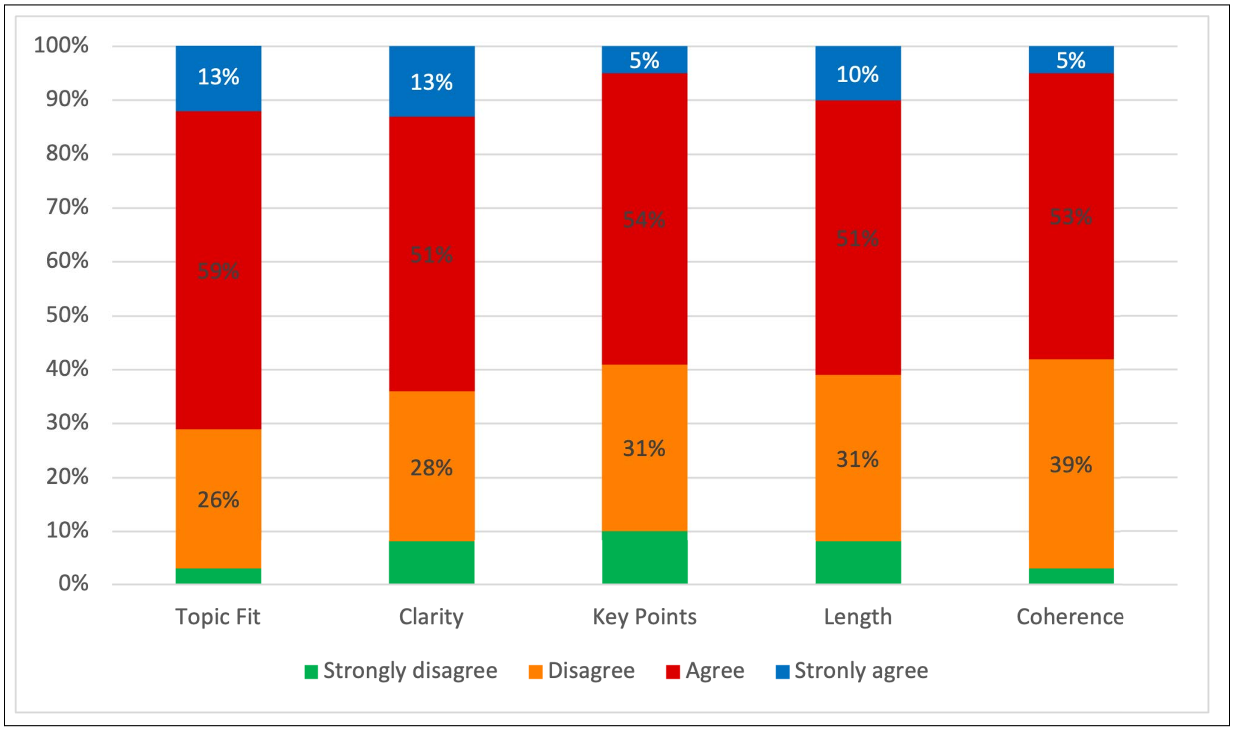


Figure 6.7 Developers' satisfaction with the quality of the automatically generated summaries

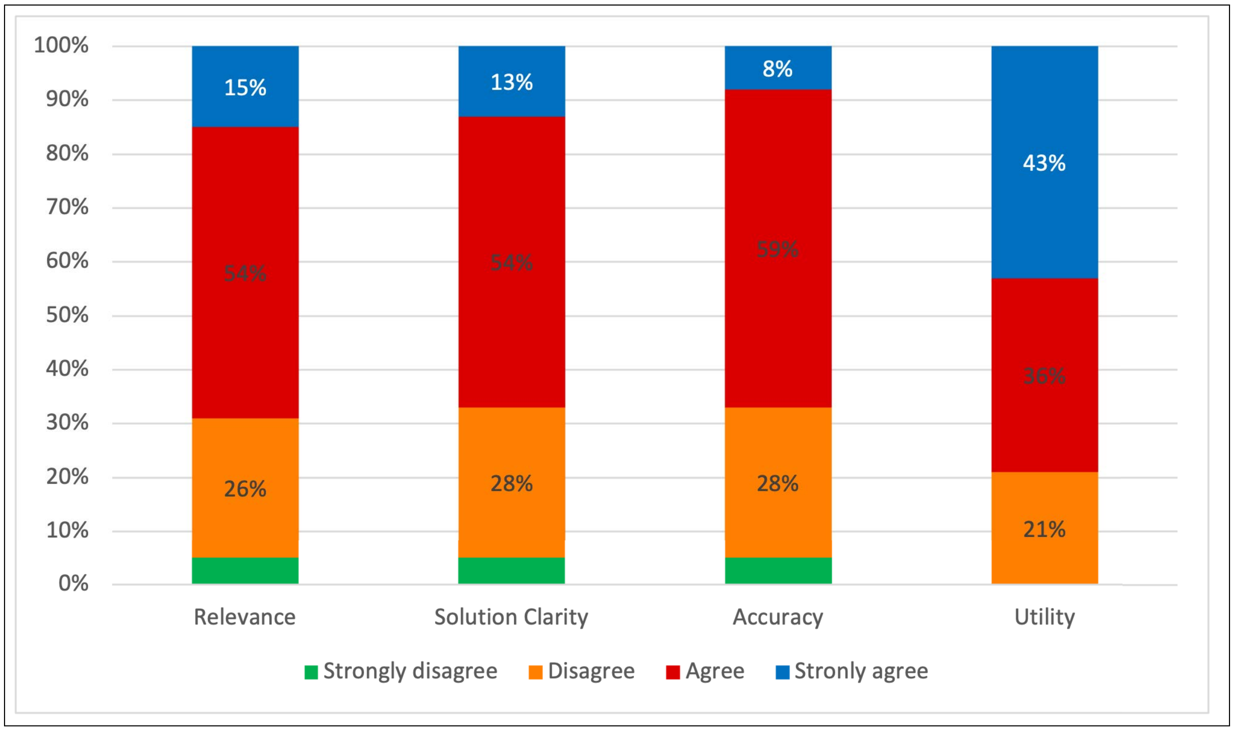


Figure 6.8 Developers' satisfaction with the quality of the automatically generated summaries

6.5 Discussion

In our research, we tackled the development challenges by introducing an innovative approach for generating summaries from informal sources like Stack Overflow. Our methodology involved BERTopic for topic modeling and BERT text summarization algorithm.

We found the results of this research interesting and inspiring. Using the identified information, we could find the most discussed Android topics on Stack Overflow. We believe this information can be very useful for extracting data. For instance, the topics can reveal that the most important problem developers face is building the project and exporting the application. Additionally, word scores can reveal important information, such as which libraries are mostly used for push notifications or analytics. We believe this data can serve as inspiration for future research directions.

To address our first research question **(Q1)**, we identified recurring topics discussed on Stack Overflow concerning Android APIs. We applied BERTopic to a dataset of 3,698,168 unique Android posts, extracting 80 prominent topics that shed light on frequently discussed issues in Android development.

Moving on to our second research question **(Q2)**, we employed text summarization techniques to identify frequent issues within these topics. By summarizing questions related to each topic, we pinpointed prevalent problems gleaned from the Stack Overflow community's collective knowledge.

Lastly, addressing our third research question **(Q3)**, we harnessed information from unofficial documentation to generate solutions to these frequent issues. Summarizing answers to related questions, we provided potential solutions sourced from the Stack Overflow community's insights, aiming to address identified challenges. Furthermore, Furthermore, we conducted a survey with Android developers to evaluate our approach.

To wrap up, our research demonstrated the feasibility of automatically generating summaries from informal sources, streamlining the understanding of APIs and methods for developers. While this study offers valuable insights and serves as a foundational exploration, further research and validation are essential to refine and expand upon these findings.

6.6 Future Plan

As we continue to refine and expand our approach, there are several avenues for future work that can enhance and expand upon our findings. The following are potential areas for future research:
Extension to other programming languages and domains: Although our current study focused on the Android programming language and its APIs, the methodology we presented can be extended to other programming languages and domains. By adapting the techniques to different contexts, we can broaden the applicability and impact of our work.
Exploration of alternative information sources: While our research primarily relied on Stack Overflow as the source of informal documentation, there are other platforms and sources of

information that developers frequently consult, such as GitHub, bug reports, blogs, and forums. Exploring these alternative sources and incorporating them into our methodology can provide a more comprehensive and diverse set of data.

Investigation of alternative summarization approaches: While our study utilized BERT-based summarization techniques, there are several other approaches that can be explored to further improve the quality and effectiveness of the generated summaries. Extractive and abstractive summarization methods, as well as hybrid approaches, could be investigated to determine their suitability for API summarization tasks.

Integration of API summaries into developer tools and IDEs: An important future direction is the integration of API summaries into developer tools and Integrated Development Environments (IDEs). By seamlessly incorporating the generated summaries within the development workflow, developers can access relevant information more easily, reducing the time spent searching for documentation and improving productivity.

By continuing to explore these research directions and addressing the aforementioned areas for future work, we can further enhance the capabilities of our approach and contribute to the advancement of developer productivity and software engineering practices.

6.7 Conclusion

In summary, our research offers a promising solution to the challenges developers encounter with complex API documentation. We've introduced an innovative approach using BERTopic and BERT-based text summarization to extract key insights from informal sources like Stack Overflow. Our method successfully identifies frequent topics in Android discussions, summarizes prevalent issues, and provides potential solutions from the Stack Overflow community.

Results from our user study indicate that our approach is well-received by developers, with positive ratings for coherence, informativeness, relevance, and user satisfaction. This research has the potential to lead to practical tools like IDE plugins or recommender systems that aid developers in understanding API documentation more effectively.

CONCLUSION AND RECOMMENDATIONS

This thesis makes a substantial contribution to the field of automated code summarization by leveraging informal documentation as a primary data source, innovatively combining deep learning with traditional machine learning methods for generating nuanced, context-rich summaries. This hybrid approach efficiently bridges the gaps in official documentation by utilizing platforms like Stack Overflow, thereby offering developers more practical and application-centric insights. The introduction of topic modeling techniques further enhances the semantic understanding and thematic coherence of the summaries, enabling the categorization of complex discussions into more structured, theme-based insights. This research, underscored by empirical evaluations involving professional developers, not only advances the technical methodology of code summarization but also provides practical tools that significantly boost software developers' productivity and comprehension, offering technically accurate and topically organized summaries that are highly applicable in real-world scenarios.

We divided our research into three parts. In the first part of our work, we have proposed an approach that leverages unsupervised learning to produce automatically generated summaries for unofficial documentation. In effect, we have applied the TextRank algorithm to generate extractive summaries for Android API methods discussed in Stack Overflow. Our approach has been empirically evaluated using a study that has involved 16 professional developers, who have evaluated our generated summaries to assess their relevance and quality and compared them against official documentation.

The findings of this first part of our research have shown that informal documentation, in particular, the Stack Overflow can be an interesting source of information for the code summarization task and that the automatically generated summaries using the TextRank-based approach can compete with official documentation (Naghshzan *et al.*, 2021).

While the results obtained during the first part of our research were promising, we aimed to improve the quality and maturity of our automatically-produced summaries especially since our first study has revealed that developers and practitioners are interested in having code summarizer tools to use within their IDEs. Therefore, we have leveraged abstractive summarization in the second part of our research to improve the quality of the generated summaries by increasing not only the accuracy of information but also the coherence and cohesion. The use of deep learning has been motivated by the fact that deep learning models have the ability to process and understand large amounts of data, making them well-suited for the task of summarization. Additionally, deep learning models can be trained on large datasets, allowing them to learn complex patterns and relationships within the data. This ability to capture complex patterns is what makes deep learning models so effective in summarization tasks, where the goal is to extract the most important information from a large input and condense it into a brief, concise summary.

In the second part of our research, we have applied the BART algorithm, which is a state-of-the-art deep learning algorithm that uses bidirectional encoding to produce better representations of input text. It is used in a variety of tasks, including text summarization, where it is used to generate concise summaries from longer texts. To validate our deep learning-based approach, we have conducted an empirical investigation to evaluate the accuracy of our generated summaries and also compare them against the baseline. Additionally, two annotators have created an oracle of human-generated summaries to use for comparison purposes. The automatically generated summaries have been evaluated using the widely used metrics ROUGE and BLEU, which are both evaluation metrics used in the field of natural language processing (NLP) and are frequently used to assess the quality of machine-generated text such as text summaries and machine translations.

The results of this second part of our research have indicated that abstractive summarization could improve the quality of the automatically generated summaries in terms of precision by 32 percent, recall by 25 percent, and F-measure by 21 percent. In addition, the average execution time of generating a summary has been improved by 4.4 times (Naghshzan *et al.*, 2023b).

In the third part of our study, we developed a novel methodology combining summarization and topic modeling techniques to enhance the understanding of API documentation. This approach leverages both the content and contextual information from informal sources, particularly Stack Overflow discussions, to generate more comprehensive and informative summaries of API methods. We employed advanced topic modeling algorithms to identify key themes and issues discussed in these forums. These insights were then integrated with our summarization process, resulting in summaries that not only describe API functionalities but also provide contextual information such as frequent challenges and solutions. Our empirical evaluation, involving software developers, demonstrated the effectiveness of this combined approach in providing valuable insights and assisting developers in understanding complex API documentation.

The results of the research section provided valuable insights into its real-world applicability, highlighting the potential to aid developers with practical and relevant information derived from informal documentation sources. These results were presented at the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) (Naghshzan & Ratte, 2023b).

The main contributions related to our research are as follows:

- **RQ1**: *Can we leverage unofficial documentation to generate code summaries automatically?* The findings of our research have shown that informal documentation, in particular, Stack Overflow can be an interesting source of information that provides insights about the purpose and usage of API methods trapped in Stack Overflow posts. Other sources of information need to be investigated such as bug reports, email exchanges between developers, chats, etc.

A combination of several sources of information may also enrich the produced summaries and make them more comprehensive. Furthermore, our approach can also be extended to include other sources of information such as API reference documentation, code snippets, and developer forums to further improve the quality and accuracy of the generated summaries.

- **RQ2**: *Are the automatically generated summaries comparable with official documentation?* Our results have shown that while developers mostly prefer to use official documentation to seek information related to their engineering tasks, our summaries can compete with the official descriptions provided by official documentation since they can contain more insights on the API methods that are not included in the official documentation. Moreover, the developers have agreed that our automatically generated summaries could be used as a complementary source of information to the official documentation. Our automatic API methods summaries can hence be used to enrich descriptions from the official documentation.
- **RQ3**: *Which of the investigated algorithms are generating the best-quality code summaries?* We have proposed two approaches for the summarization of API methods discussed in informal documentation. The first approach is based on unsupervised learning while the second approach leverages the deep learning Transformer model. Additionally, we have compared our approaches using ROUGE and BLEU metrics. These metrics are useful for comparing the performance of different summarization models, as they provide a quantitative evaluation of the generated summaries and determine which approach is most effective for a given task.

  We have applied rigorous statistical tests to investigate if there are any significant differences between approaches. The results of statistical tests show that not only the difference between approaches is significant but also the deep learning approach could improve the quality of the generated summaries by 32 percent for precision, 25 percent for recall, and 21 percent for F-measure. Moreover, the execution time for the deep learning-based approach was 4.4 times faster than the unsupervised learning-based approach.

- **RQ4**: *What are the prevalent Android API topics on Stack Overflow, and can summarization methods identify and solve frequent issues within these topics?*

  Our results revealed 81 prominent topics, highlighting frequent issues such as project build errors and library usage challenges. We could effectively summarize frequent problems and derive potential solutions from community insights by applying summarization techniques to these identified topics. This method was validated through a user study involving Android developers, who rated the generated summaries positively for coherence, informativeness, and practical utility, thus confirming the feasibility and effectiveness of our approach in enhancing API documentation.

In our future work, we intend to leverage other types of informal documentation such as bug reports, GitHub, etc. In addition, a combination of source code, official documentation as well informal documentation is an idea that we plan to explore to examine the effect of the combination of several types of sources of information on the quality of the automatically produced summaries.

While our automatically generated summaries contain information on the purpose and usage of API methods, we intend to enrich our automatically produced summaries with information about API methods, in particular API methods that are related to a specific API method. In such a way, our summaries would contain not only the purpose and usage of API methods but also the API methods that are related to them. This information can help developers understand the interconnections between the different parts of the code and how changes to one method may impact the rest of the codebase. Additionally, information about related methods can also help developers understand the relationships between different methods and how they are used together. For example, if a method is called by several other methods, it can provide insights into how it is being used and what its intended purpose is.

We have set the foundation for this research by first suggesting a recommender system that leverages advanced data mining techniques to recommend related files. Although this research has made the focus on the method level, the approach can be applied to another granularity level, *i.e.*, API methods. We aimed to investigate, at the first stage, the use of advanced data mining techniques at the class level, and then extend this work to the API method level. The recommender system can predict which summaries would be most relevant and useful to the developer, given their current task and context. This can help the developer save time by quickly finding the most relevant information, without having to search through the entire source code.

The finding of this research work, whose aim was to leverage advanced Data Mining algorithms to recommend source code changes at the class level, has been submitted to Elsevier Journal of Information and Software Technology (Naghshzan *et al.*, 2023a).

This research work can set the foundation for a follow-up study that involves API methods. In effect, the data mining techniques leveraged in this study can be applied in the context of the code summarization task. In effect, we have investigated four data mining algorithms, namely Apriori, FP-Growth, Eclat, and Relim for the recommendation task. The same algorithms can be applied when recommending related API methods to summarize an API method. The only difference that will need to be addressed is the extraction of API methods and their change history.

Looking forward, the introduction of ChatGPT, a variant of the Generative Pre-trained Transformer models tailored for conversational contexts, presents an exciting avenue for further research into automated code summarization. ChatGPT's ability to engage in dialogue and mimic human conversational patterns offers a unique opportunity to enhance the interaction between developers and automated summarization systems. By integrating ChatGPT into the automated code summarization framework, we aim to significantly improve the usability and accessibility of these tools. Enabling developers to query summaries through natural language conversations could revolutionize how they access, understand, and interact with code summaries. This approach promises to make these interactions more intuitive and efficient, providing a more user-friendly interface that allows for the dynamic retrieval of information based on the developer's current context and needs. Such advancements could lead to a more engaging and effective way to navigate complex code documentation, aligning with our goal to provide practical, application-centric insights for software development.

By pursuing this direction, we hope to open new pathways for research in software development tools, further bridging the gap between human developers and the vast amounts of code documentation available to them. This could significantly contribute to the field by making software development more accessible and efficient, thereby supporting the ongoing evolution of software engineering practices.

APPENDIX I

GENERATED SUMMARIES

Table I-1 presents the result of extractive summaries generated by the TextRank algorithm for Android API methods discussed on Stack Overflow.

Table-A I-1 Examples of our automatically generated summaries using TextRank

Method	**Summary**
app.activity. onCreate	when the app is launched, the first thing that's going to run is your onCreate() in this case, onCreate() has a method that inflates the view of your activity, that method is called setContentView(). When you pass data from one activity to another using a Bundle, the data is received inside onCreate() method of the second activity not insideonActivityResult() unless you've specifically implemented that. Inside your Activity instance's onCreate() method you need to first find your Button by it's id using findViewById() and then set an OnClickListenerfor your button and implement the onClick() method so that it starts your new Activity.
os.asyncTask. onPostExecute	You will need to return some data (probably contactList) from your doInBackground() method, and then move the offending code to the onPostExecute() method, which is run on the UI thread. You can also do whatever post processing you need to do with the JSON object in the onPostExecute() method itself (for ex: parse the object and display it to the user) since this method is running on the UI thread after the async task completes its operations in the background thread. You just put your work code in one function (doInBackground()) and your UI code in another (onPostExecute()), and AsyncTask makes sure they get called on the right threads and in order.

Method	**Summary**
os.asyncTask. doInBackground	To get result back in Main Thread you will need to use AsyncTask.get() method which make UI thread wait until execution of doInBackground is not completed but get() method call freeze the Main UI thread until doInBackground computation is not complete . You will need to return some data (probably contactList) from your doInBackground() method, and then move the offending code to the onPostExecute() method, which is run on the UI thread. You most likely don't want to directly instantiate a Handler at all... whatever data your doInBackground() implementation returns will be passed to onPostExecute() which runs on the UI thread. You just put your work code in one function (doInBackground()) and your UI code in another (onPostExecute()), and AsyncTask makes sure they get called on the right threads and in order. Basically, you call the AsyncTask from the UI such as in your onClick() then you do network operations in doInBackground() which is called when the task first starts then you can update the UI in any of its other methods.
app.activity. onPause	Called as part of the activity lifecycle when the user no longer actively interacts with the activity, but it is still visible on screen. The counterpart to onResume(). When activity B is launched in front of activity A, this callback will be invoked on A. B will not be created until A's onPause() returns, so be sure to not do anything lengthy here. This callback is mostly used for saving any persistent state the activity is editing, to present a "edit in place" model to the user and making sure nothing is lost if there are not enough resources to start the new activity without first killing this one. This is also a good place to stop things that consume a noticeable amount of CPU in order to make the switch to the next activity as fast as possible.

Method	Summary
app.activity. startActivityForResult	In the second activity perform whatever validation you need to do (this could also be done in the first activity by passing the data via an Intent) and call setResult(int resultCode, Intent data) (documentation) and then finish(); from the second activity. I'm not sure you can directly programmatically remove activities from the history, but if you use startActivityForResult() instead of startActivity(), then depending on the return value from your activity, you can then immediately finish() the predecessor activity to simulate the behaviour you want. You need to also start any new activity with startActivityForResult method then pass result back with a chain of setResult calls, getting it inside onActivityResult and setting again with setResult.
app.Fragment. onCreateView	Anyway in your proposed single-pane Activity with 2 fragments, your setShownIndex method could set a private field in DetailsFragment which is loaded in onCreateView or onActivityCreated. You dont call setContentView in fragments, in fact you need to return a View from onCreateView. Fragments can be added inside other fragments but then you will need to remove it from parent Fragment each time when onDestroyView() method of parent fragment is called. The Views in your Fragment's layout xml are being inflated into the Fragment's View hierarchy, which won't be added to the Activity's hierarchy until the onAttach() callback, so findViewById() in the context of the Activity will return null for those Views at the time that onCreate() is called. If the view with that id is present in your fragment itself then you should use the onCreateView method within your fragment.

Method	Summary
app.activity. finish	Note: make sure you are calling finish() in onBackPressed() in Activity B; which indicates that you no longer need this Activity(Activity B) and can resume last activity which was paused/stopped and is in background stack Basically, use setResult before finishing an activity, to set an exit code of your choice, and if your parent activity receives that exit code, you set it in that activity, and finish that one, etc... finish() is only called by the system when the user presses the BACK button from your Activity, although it is often called directly by applications to leave an Activity and return to the previous one. You could try and explicitly call for finish() in activity then onDestroy will be called and see how the situation will change.But then the activity will be removed from stack.
app.activity. setContentView	You don't have setContentView() in your Activity, hence, there's actually no View referenced to your activity and no views to find using findViewById() method, make sure the setContentView() was actually called... You are actually trying to use View.findViewById() which is wrong as you need to find the view from the layout attached to the activity, that you have set using setContentView() method. In an activity you need to call setContentView() first to set a layout. Typically, you call setContentView() only once in Activity.onCreate(), specifying a layout that contains all of the Views you wish to work with (you can of course add more Views to any ViewGroups in your layout, at a later time). When you call setContentView, Android "adds", the widgets that you declared in your layout as part of the Activity's view hierarchy, making them available for you in the context of the current activity.

Method	Summary
app.activity. onStop	if onStop() in one of your activities is called, and onStart() in another of your activities is not called within X period of time, presumably some other app's activity is in the foreground. There can be exceptions of course, like if the called activity causes an app crash in either of its onCreate(), onStart() or onResume() then the onStop() of the calling activity will not be called, obviously, I'm just talking about the general case here. Most developers care about any case where their activity moves to the background, in which case you can either use a lifecycle method (e.g., onPause(), onStop()) or try onUserLeaveHint(). I'm not sure which emulator you are testing with, but onPause is the one method that is always guaranteed to be called when your Activity loses focus (and I say always because on some devices, specifically those running Android 3.2+, onStop is not always guaranteed to be called before the Activity is destroyed).
app.activity. startActivity	In the second activity perform whatever validation you need to do (this could also be done in the first activity by passing the data via an Intent) and call setResult(int resultCode, Intent data) (documentation) and then finish(); from the second activity. I'm not sure you can directly programmatically remove activities from the history, but if you use startActivityForResult() instead of startActivity(), then depending on the return value from your activity, you can then immediately finish() the predecessor activity to simulate the behaviour you want. You need to also start any new activity with startActivityForResult method then pass result back with a chain of setResult calls, getting it inside onActivityResult and setting again with setResult.

Method	**Summary**
adapter. onBindViewHolder	Called by RecyclerView to display the data at the specified position. This method should update the contents of the RecyclerView.ViewHolder.itemView to reflect the item at the given position. Note that unlike ListView, RecyclerView will not call this method again if the position of the item changes in the data set unless the item itself is invalidated or the new position cannot be determined. For this reason, you should only use the position parameter while acquiring the related data item inside this method and should not keep a copy of it.
app.Activity. onActivityResult	To get the returning data from your second activity in your first activity, just override the onActivityResult() method and use the intent to get the data. In the second activity perform whatever validation you need to do (this could also be done in the first activity by passing the data via an Intent) and call setResult(int resultCode, Intent data) (documentation) and then finish(); from the second activity. so what you should do is: from your activity1 start activity for result, and from activity2 use the setResult(int resultCode, Intent data) method with the data you want your activity1 to get back, and call finish() (it will get back to onActivityResult() in the same state activity1 was before..). You need to also start any new activity with startActivityForResult method then pass result back with a chain of setResult calls, getting it inside onActivityResult and setting again with setResult. When you pass data from one activity to another using a Bundle, the data is received inside onCreate() method of the second activity not inside onActivityResult() unless you've specifically implemented that.

Method	Summary
app.activity. findViewById	The layout with your ProgressBar in it has to have been inflated first, and you may need to use View.findViewById() instead of Activity.findViewById(). you can call findViewById() directly for Activity, however as you are using a Fragment, youo will need a view object to call findViewById(). In other words, use findViewById to get the activity's view, then call findViewWithTag() on that view. It won't work because your are calling the findViewById() method before the setContentView() one, so it will search for a view that it hasn't created yet.
adapter. getView	In the method getView of the Adapter when you get a recycled item, you must check if it's not already the one you want to display. In case this is not enough (say you'd like to always show the last selection differently beside the current selection), you should store your last selected item (a data which populates the ListAdapter) as lastSelectedItem, and in your adapter's getView method assign a different background resource to the renderer if it equals this lastSelectedItem. If we need to show different type of view in list-view then its good to use getViewTypeCount() and getItemViewType() in adapter instead of toggling a view View.GONE and View.VISIBLE can be very expensive task inside getView() which will affect the list scroll. You need to make sure you completely reset the state of your item view in each call to getView() so that changes like this don't leak from one use of the view item to the next.

Method	**Summary**
app.activity. onStart	if onStop() in one of your activities is called, and onStart() in another of your activities is not called within X period of time, presumably some other app's activity is in the foreground. In other way you can set it in onStart or onResume, but you should understand, that its bad practice, because onStart and onResume calls every time, when user see your activity. There can be exceptions of course, like if the called activity causes an app crash in either of its onCreate(), onStart() or onResume() then the onStop() of the calling activity will not be called, obviously, I'm just talking about the general case here. The other callbacks you will receive are onStart() and onResume(), which are also called any time that Activity comes to the foreground (include the first time). Note that this method will ensure that absolutely no UI state information will be stored when the Activity is killed by the system - this has the effect of also not restoring any Views when onRestoreInstanceState() gets called with the same Bundle after onStart().
adapter. notifyDataSetChanged	Preferred way to change the appearance/whatever of row views once the ListView is drawn is to change something in the data ListView draws from (the array of objects that is passed into your Adapter) and make sure to account for that in your getView() function, then redraw the ListView by calling You only need to change the adapter data of your second RecyclerView and call notifyDataSetChanged. Filtering is actually repopulation the ListView, whereas you create/get a new collection and tell the Adapter it's content has changed by calling notifyDataSetChanged. off course adapter.add(item); and after that call notifyDataSetChanged() is the better way rather than initialize new adapter and set it again.

Method	Summary
app.activity. onResume	You can set it in onStart or onResume, but you should understand, that its bad practice, because onStart and onResume calls every time, when user see your activity. When results from network calls return, either post to a bus such as Otto which the activity (register for the event on onResume() and deregister during onPause()) has registered to, or make sure the callback interface implemented by the activity has been updated to the last activity in the presenter. You may need to refresh the data, if MainActivity requires info from Options 1-5, as CLEAR_TOP will, in most cases, bring the old instance of MainActivity into focus rather than completely recreating it (onCreate will not be called, but onStart and onResume will).
app.Activity. onBackPressed	Make sure you do call the Activity onBackPressed() method if whatever you're doing with the back button doesn't require handling, otherwise, the back button will be disabled... and you don't want that! It's not necessary to override onBackPressed(), since if the activity is destroyed the necessary lifecycle methods will be called (plus, it could be finished without pressing the back button, for example if calling finish() on it). What you can do is override the onBackPressed() method in the parent activity of your Foo fragment, then using an interface communicate to the fragment that back button was pressed by the user. Override the method onBackPressed() in whatever Activity you want to create a different behaviour to the back button. Wherever I'm starting an intent new activity (or pressing "back" from an activity that isn't the entry point, overriding the onBackPressed method), I'd set that value to true, and have onPause not stop the music if that flag is set.

Method	Summary
app.activity. onPuase	Most developers care about any case where their activity moves to the background, in which case you can either use a lifecycle method (e.g., onPause(), onStop()) or try onUserLeaveHint(). Your Activity may still be alive and in memory after onPause() has been called, but there is no scenario which I can think of in which your Activity is alive and in the background without onPause() being called. Note: When your activity receives a call to onPause(), it may be an indication that the activity will be paused for a moment and the user may return focus to your activity. onSaveInstanceState() won't be called even if you call finish() outside of onCreate() because the user will never be able to return to the activity so it will never need to restore its state.
app.activity. onDestroy	Activity.onDestroy() only gets called in the case of a controlled shutdown of the activity - something that happens extremely rarely, as the Android OS can kill your process in a variety of states without ever calling your onDestroy() method. You could try and explicitly call for finish() in activity then onDestroy will be called and see how the situation will change.But then the activity will be removed from stack. Activity doesn't have an equivalent method, but you could use onDestroy() as long as you know it may never be called if the system decides to terminate your app unexpectedly. There is no guarantee that onDestroy will ever be called; once an activity has been moved to the background, it is fair game to be killed if system resources are required, even without calling onDestroy. By default when you call finish in your activity it will automatically call onDestroy method of all the fragments that is attach to the activity thus destroying it, so no need to worry about destroying the fragment in onDestroy method of your activity.

Table I-2 presents the result of abstractive summaries generated by BART algorithm for Android API methods discussed on Stack Overflow.

Table-A I-2 Examples of our automatically generated summaries using BART

Method	**Summary**
app.activity. onCreate	OnCreate is the first method in the Activity lifecycle whose job is to create an activity. It is called first time the Activity is created, or when there is a configuration change like a screen rotation. This is the beginning state of an activity lifecycle which its view should be set. Most of the activity initialization code goes here.
os.asyncTask. onPostExecute	This method is called on the thread that created the task. It is called after doInBackground has returned and the result parameter is the value that doIn background has returned. This method is the first method of android activity lifecycle. It helps deliver the calculations that has been done in doInbackground to the main thread.
app.fragment. onCreateView	The Fragment.onCreateView() method is responsible for creating and returning a View. This View is displayed in the UI Fragment, so the desired View must be created in this method. You can return null if the fragment does not provide a UI. The FragmentFragment() method creates a Fragment Fragment and returns a View component that is the root of your fragment's layout.
os.asyncTask. doInBackground	AsyncTask provides a simple way to run tasks in background thread and publish progress and result in the main thread. When creating the task you provide the type of the input parameters, progress, and result with its generic type mechanism. In the doInBackground method you do the long running job and publish the progress.

Method	Summary
app.activity. onPause	In activity's lifecycle when the activity goes from running state to pause state (when we can see the activity but it is not interactable) this method will called. This method implemented, but based on our functions and scenario, we can override it. such as deallocating memory or stop something from processing. Note that when you are on onPause() your Activity is still alive.
app.activity. findViewById	android.app.Activity.findViewById is a method that returns the View corresponding to the id passed. getView() can be use to get the id which can then be passed to this method to find the View. The method returns the first View with the given id or returns null.
app.activity. onDestroy	It's one of activity lifecycle methods that's called when the activity is finishing. It's one way to show that an activity is finished. It can also be used to show the end of an activity or a new one. It is called the activity completion method.
app.activity. finish	this method invoke onDestroy() method on activity. onDestroy(); method is the final call that the activity receives. Its like death of an application. Once this is done all the progress in the app is lost unless app data is saved locally or in a database online.
app.activity. setContentView	We can set an activity's view with this method. The given view directly attached to the activity. Use this to set the content View of your Activity. Either by passing the layout resource id or an instance of View subclasses. android.app.Activity.setContentView this method is used to set. the content of the Activity to a View.

Method	Summary
app.activity. onStop	The main mechanism in and Android app to show interface is through Activity class. The platform itself handles its life-cycle and in each step calls methods which can be overrided in your derived class. onStop is one such method that gets called once the activity is no longer visible. This is a good place to stop refreshing UI, running animations and other visual things.
app.activity. startActivityForResult	It is called on the Activity that started another Activity with startActivity-ForResult right after the second Activity has finished. It's functionality is very similar to startActivity() except for the fact that when calling an activity using start activityForResult we expect the second activity to return some data to the first activity. This method launches another activity that can provide us with a result.
adapter. onBindViewHolder	RecyclerView.Adapter.onBindViewHolder is a method that binds ViewHolder to an Adapter, which is called by Recycler view to display data at the specified position. In this method you bind the data for the item in the given position to the provided viewholder.
app.activity. startActivity	The startActivity() method is used to run and display a new Activity. It must be run on a thread that can change the UI. The started activity can also set a result code which the caller can use to determine if the activity was canceled or not. It starts a new activity which is placed at the top of activity stack, often used with intent that defines the targeted components.
app.activity. onBackPressed	this method called when you press back button and The Android operating system handles calling this method for you, so you don't have to do it yourself every time the back button is pressed. When the user activity is pressing the back key, this event is called. This method called is called when the user Activity is pressing back key.

Method	**Summary**
arrayadapter. getView	Array.ArrayAdapter class has getView() method that is responsible for creating the views. A listview calls this method to get a view for a particular position. The getView() method can inflate the view from a layout resource or create it programmatically.
app.activity. onStart	onstart() method is called when activity enters the started state and become visible to user. On start() method can also be used to start a new activity. The on start() function is called as soon as activity enters started state. It is called by the user when the activity has been started. The activity is then visible to the user.
adapter. notifyDataSetChanged	An adapter is a bridge between UI components and Data items. It fills the UI components with the underlying Data. android.widget.BaseAdapter.notifyDataSetChanged is used to trigger a UI refresh. Notifies the attached observers that the underlying data has been changed.
app.activity. onResume	OnResume() is one of the methods called throughout the activity lifecycle. It is the counterpart to onPause() which is called anytime an activity is hidden from view, e.g. if you start a new activity that hides it. OnResume is called when the activity that was hidden comes back to view on the screen.
app.activity. onResume	OnResume() is one of the methods called throughout the activity lifecycle. It is the counterpart to onPause() which is called anytime an activity is hidden from view, e.g. if you start a new activity that hides it. OnResume is called when the activity that was hidden comes back to view on the screen.

Method	Summary
app.activity. onDraw	onDraw is one of the methods in the View class and it uses canvas to draw views. The method is used especially for creating custom views. The input for this method is a canvas object. The Canvas object is used for drawing graphical shapes, lines, texts, bitmaps, etc. This method is used to create a customized user interface.
app.activity. onSaveInstanceState	the method is called before placing the activity in such a background state, allowing you to save away any dynamic instance state in your activity into the given Bundle, to be later received in onCreate if the activity needs to be re-created.
app.activity. onNewIntent	when the activity is re-launched while at the top of the activity stack instead of a new instance of the activity being started, onNewIntent() will be called on the existing instance with the Intent that was used to re-launch it. A new intent cannot be given to an activity if it is on resumed state. If the activity is on resumed state, it should be paused and the new intent is given and then it will be resumed.
app.activity. onActivityCreated	This method notifies that the calling method Activity.onCreate() of a fragment's activity is completed. This method is called usually between onViewCreated() and onViewStateRestored(). If the fragment is detached or re-attached, it could be called multiple times.
app.activity. onCreateOptionsMenu	It Initializes the contents of the Activity's options menu. This is only called once, the first time the options menu is displayed. To update the menu every time, we should call onPrepareOptionsMenu(Menu).
app.activity. runOnUiThread	Executes the action specified on the UI thread. If the current thread is the user interface thread, the action is instantly executed. The action is posted to the event queue of the UI thread if the current thread is not the UI thread.

Method	Summary
app.activity. onProgressUpdate	method is executing, it calls this method. This method updates the UI's elements such as text and progress bar to show that there are some progress in the executing method while the background process is running
app.activity. onTouchEvent	This method handles the motion events in touch screen. This method detects the click actions and the actions could be managed by perform-click() function. This facility handles the behaviors such as click sound preferences, OnClickListener calls and ACTION$_{C}LICKaction.$
app.activity. onDataChange	This method reads the static snapshot of a path and is triggered when the listener is attached and there are some changes in content. By calling this method, a snapshot is returned that contains all data such as child data. If there is no data, the snapshot value will be null. This method is called when there are some changes on a database reference such as changes to children.
app.activity. onMeasure	This method allows you to determine the height and width of a view. It specifies if you want your view's size would be dependant on the size of the parent layout or not. You have the option to set the height and width value to a specific value or match it based on the parent's view.
app.fragment. onViewCreated	This method is called after OnViewCreate() method and assures that the fragment's vroot view is not null. It gives the opportunity to subclasses to initialize themselves after their hierarchy has been created completely.
app.activity. onRestoreInstanceState	When an activity is re-initialized from a previously state by onStart() method, then this method is called. This method restores the state of a view that has been accomplished by onSaveInstanceState(Bundle). This method is called when recreating an activity and between onStart() and onPostCreate(Bundle) methods.

Survey

Title: Leveraging informal documentation to automatically summarize the usage of code entities
Lead Researcher: Latifa Guerrouj, Software Engineering, and Information Technology
Student: Amirhossein Naghshzan, Ph.D. student, Software Engineering and Information Technology

This research project aim is to contribute to the enhancement of the state-of-the-art and practice on code summarization. By contributing to this survey, you will provide us with insights that can be leveraged to build actionable summarization techniques and tools to assist and guide developers during their software maintenance and evolution tasks.

* Indicates required question

Participation

You are invited to participate in this research project. After you read this document and agree to participate in the project, you will be asked to complete two questionnaires about code summarization. Completing the questionnaires will take approximately 40 to 60 minutes.
Your participation is voluntary, meaning you may decline if you wish. You can cease participating at any time by leaving any remaining questions blank. You will not have to provide a reason to justify your decision. Once you have finished completing the questionnaire and submitted it, you will no longer be able to withdraw from the experiment. Since the questionnaires are anonymous, we will not be able to determine which answers are yours.

Privacy

The data collected for this project will be confidential to the extent permitted by law. This information will be kept by the lead researcher on the project at ETS for one year.
In order to preserve your identity and the confidentiality of your information, the information gathered will be coded. The data may be published in reports, articles, be the subject of scientific discussion or used for teaching purposes. However, it will not be possible to identify you in any of these cases.

Ethical Considerations

The ÉTS Research Ethics Committee has authorized this research project. If you have questions about your rights as a participant in the research project, please contact the coordinator of the Research Ethics Committee at 514- 396-8800 extension 7807.

Contact Person

If you have questions about the project or your role in it, please contact (Naghshzan, Amirhossein, project lead, amirhossein.naghshzan.1@ens.etsmtl.ca)

1. Consent *

Check all that apply.

☐ I have read and understood the project description
☐ I AGREE to participate by completing the questionnaires

A short background survey

2. What is your gender? *

Mark only one oval.

- ⬭ Male
- ⬭ Female
- ⬭ Other

3. How old are you? *

Mark only one oval.

- ⬭ < 20 years
- ⬭ 20 - 25 years
- ⬭ 25 - 30 years
- ⬭ 30 - 40 years
- ⬭ > 40 years

4. What is your latest degree obtained or currently enrolled? *

Mark only one oval.

- ⬭ Diploma
- ⬭ Bachelor
- ⬭ Master
- ⬭ Ph.D.
- ⬭ Post doc
- ⬭ Other

5. How many years of programming experience do you have? *

Mark only one oval.

- ⬭ < 2 years
- ⬭ 2 - 5 years
- ⬭ 5 - 10 years
- ⬭ 10 - 15 years
- ⬭ > 15 years

6. How many years of professional experience do you have in the industry? *

Mark only one oval.

- ◯ < 2 years
- ◯ 2 - 5 years
- ◯ 5 - 10 years
- ◯ 10 - 15 years
- ◯ > 15 years

7. What is your skill level in Android development? *

Mark only one oval.

- ◯ Beginner
- ◯ Intermediate
- ◯ Senior
- ◯ Lead

8. What is your current status? *

Mark only one oval.

- ◯ Working in industry
- ◯ Working in academia
- ◯ Student
- ◯ Post doc
- ◯ Faculty member
- ◯ Other

9. How do you find answers to your programming questions? *

Mark only one oval.

- ◯ Offical documents
- ◯ Stack Overflow
- ◯ Quora
- ◯ Reddit
- ◯ Code Project
- ◯ Other: ____________________

10. How often do you use Stack Overflow to find solutions? *

Mark only one oval.

- ⬭ Never
- ⬭ Occasionally
- ⬭ Often
- ⬭ Most of the time

In this experiment, we invite you to write and evaluate the summaries of 3 Android methods. To perform this task, you will need to follow these steps for the 3 methods:

1. Write in your own words a summary of a given method.
2. Read the first summary given to you and evaluate it.
3. Read the second summary given to you and evaluate it.
4. Answer the post-questionnaire.

To summarize methods, we provide you with the link to the Stack Overflow pages that discuss the methods in question.

PS: All answers should be in English for this survey.

android.os.AsyncTask.onPostExecute

Please consider the method above and answer the following questions.

You can find more information about the method on the following link:

https://stackoverflow.com/search?q=%5Bandroid%5D+oncreate

11. Provide a summary of the method above. There is no restriction on the length of the summary. *

____

____

____

____

____

Summary 1

Please read the summary below for the method "android.os.AsyncTask.onPostExecute" and answer the following questions:

You will need to return some data (probably contactList) from your doInBackground() method, and then move the offending code to the onPostExecute() method, which is run on the UI thread.
You can also do whatever post processing you need to do with the JSON object in the onPostExecute() method itself (for ex: parse the object and display it to the user) since this method is running on the UI thread after the async task completes its operations in the background thread.
You just put your work code in one function (doInBackground()) and your UI code in another (onPostExecute()), and AsyncTask makes sure they get called on the right threads and in order.
Though you are using AsyncTask, its onPostExecute() method is executed on UI thread (to let you update your views etc).
It will then be passed to the onPostExecute method of the AsyncTask which will be executed on the UI thread (so you can use the data to manipulate the UI if you want).

12. The summary is accurate: *

Mark only one oval.

- Stronly agree
- Agree
- Disagree
- Strongly disagree

13. The summary contains all information about the method: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly disagree

14. The summary contains only important information about the method: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly Disagree

15. The summary contains information that helps understand how to use the method: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly Disagree

16. The summary contains information that helps understand how to implement the method: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly disagree

17. The summary is coherent *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly disagree

Summary 2

Please read the description below for the method "android.os.AsyncTask.onPostExecute" and answer the following questions:

Runs on the UI thread after doInBackground(Params...). The specified result is the value returned by doInBackground(Params...). To better support testing frameworks, it is recommended that this be written to tolerate direct execution as part of the execute() call. The default version does nothing.

This method won't be invoked if the task was cancelled.

This method must be called from the Looper#getMainLooper() of your app.

18. Which information do you find more useful? (You can access summary 1 at any time by using the button 'Back' at the bottom of the page) *

Mark only one oval.

- Summary 1
- Summary 2
- The same

19. Which information can help you better understand the usage of the method? (You can access summary 1 at any time by using the button 'Back' at the bottom of the page) *

Mark only one oval.

- Summary 1
- Summary 2
- The same

20. Which information can help you better understand the implementation of the method? (You can access summary 1 at any time by using the button 'Back' at the bottom of the page) *

Mark only one oval.

- Summary 1
- Summary 2
- The same

21. Do you think the summaries can be used as a complementary source for each other? *

Mark only one oval.

	1	2	3	4	5	
Not						Exteremly

android.app.Activity.onCreate

Please consider the method above and answer the following questions.

You can find more information about the method on the following link:

https://stackoverflow.com/search?q=%5Bandroid%5D+oncreate

22. Provide a summary of the method above. There is no restriction on the length of the summary. *

Summary 1

Please read the summary below for the method "android.app.Activity.onCreate" and answer the following questions:

when the app is launched, the first thing that’s going to run is your onCreate() in this case, onCreate() has a method that inflates the view of your activity, that method is called setContentView(). When you pass data from one activity to another using a Bundle, the data is received inside onCreate() method of the second activity not insideonActivityResult() unless you’ve specifically implemented that. Inside your Activity instance’s onCreate() method you need to first find your Button by it’s id using findViewById() and then set an OnClickListenerfor your button and implement the onClick() method so that it starts your new Activity.

23. The summary is accurate: *

Mark only one oval.

- ◯ Stronly agree
- ◯ Agree
- ◯ Disagree
- ◯ Strongly disagree

24. The summary contains all information about the method: *

Mark only one oval.

- ◯ Strongly agree
- ◯ Agree
- ◯ Disagree
- ◯ Strongly disagree

25. The summary contains only important information about the method: *

Mark only one oval.

- ◯ Strongly agree
- ◯ Agree
- ◯ Disagree
- ◯ Strongly Disagree

Survey

Title: Enhancing API Documentation through BERTopic Modeling and Summarization
Lead Researcher: Sylvie Ratté, Software Engineering, and Information Technology
Student: Amirhossein Naghshzan, Ph.D. student, Software Engineering and Information Technology

This research project aim is to contribute to the enhancement of the state-of-the-art and practice on code summarization and topic modeling. By contributing to this survey, you will provide us with insights that can be leveraged to build actionable summarization and topic modeling techniques and tools to assist and guide developers during their software maintenance and evolution tasks.

~~* Indicates required question~~

Participation

You are invited to participate in this research project. After you read this document and agree to participate in the project, you will be asked to complete a questionnary.
Your participation is voluntary, meaning you may decline if you wish. You can cease participating at any time by leaving any remaining questions blank. You will not have to provide a reason to justify your decision. Once you have finished completing the questionnaire and submitted it, you will no longer be able to withdraw from the experiment. Since the questionnaires are anonymous, we will not be able to determine which answers are yours.

Contact Person

If you have questions about the project or your role in it, please contact (Naghshzan, Amirhossein, project lead, amirhossein.naghshzan.1@ens.etsmtl.ca)

A short background survey

1. How many years of programming experience do you have? *

Mark only one oval.

- ◯ < 2 years
- ◯ 2 - 5 years
- ◯ 5 - 10 years
- ◯ 10 - 15 years
- ◯ > 15 years

2. How many years of professional experience do you have in the industry? *

Mark only one oval.

- ◯ < 2 years
- ◯ 2 - 5 years
- ◯ 5 - 10 years
- ◯ 10 - 15 years
- ◯ > 15 years

3. What is your skill level in Android development? *

Mark only one oval.

- ◯ Beginner
- ◯ Intermediate
- ◯ Senior

4. What is your current status? *

Mark only one oval.

- ◯ Working in industry
- ◯ Working in academia
- ◯ Student
- ◯ Faculty member
- ◯ Other

While analyzing documentation related to Android problems and their solutions, we discovered topics. In our project, topics are a bundle of words that try to capture the essence of the topic.
For example, the following bundle <**project_error_build_gradle**> is considered a topic. Reading the bundle, we get a sense of what kind of problems are covered in that topic. Here is an example of such a problem.

"Jenkins tries to launch tools instead of emulator I'm trying to set up jenkins ui tests and it fails on running emulator command"

The current questionnaire aims to evaluate the following:
1) if the topic (bundle of words) corresponds to the text describing a problem;
2) the quality of the text describing the problem according to different particularities (clarity, completeness, coherence, etc.)
3) the quality of the text describing the solution to the problem and the code associated with it.

project_error_build_gradle

Please consider the topic above and answer the following questions.

Problem

I am trying to add Kotlin sources of an AAR in Android Studio 3.3.2. It doesn't work when I select "Choose Sources" and choose the corresponding source.jar. Android Studio only displays "Attaching" but nothing happens.

5. The problem is related to the mentioned topic (project_error_build_gradle): *

Mark only one oval.

- Stronly agree
- Agree
- Disagree
- Strongly disagree

6. The problem is clear and easy to understand: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly disagree

7. The problem provides enough information to understand the key elements of the issue: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly Disagree

8. The length of the text desribing the problem is appropriate to grasp the main issue being described: *

Mark only one oval.

- Strongly agree
- Agree
- Disagree
- Strongly Disagree

9. The problem text is coherent: *

Mark only one oval.

- ⬭ Strongly agree
- ⬭ Agree
- ⬭ Disagree
- ⬭ Strongly disagree

Solution

Configure kotlin in your project following these steps in build.gradle. Also add classpath org.jetbrains.kotlin gradle plugin clean and build your project it worked for me

```
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk7:1.3.21'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.0'
```

10. The recommended solution is directly related to the problem described in the initial question: *

Mark only one oval.

- ⬭ Strongly agree
- ⬭ Agree
- ⬭ Disagree
- ⬭ Strongly disagree

11. The solution is clear and straightforward in conveying the recommended solution? *

Mark only one oval.

- ⬭ Strongly agree
- ⬭ Agree
- ⬭ Disagree
- ⬭ Strongly disagree

12. The summary provides an accurate solution to the problem: *

Mark only one oval.

- ◯ Strongly agree
- ◯ Agree
- ◯ Disagree
- ◯ Strongly disagree

13. The code block includes the necessary code snippets or instructions to implement the recommended solution: *

Mark only one oval.

- ◯ Strongly agree
- ◯ Agree
- ◯ Disagree
- ◯ Strongly disagree

14. Did the inclusion of a code block enhance the overall effectiveness of the solution? *

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Extremely

Post-Questionnaire

15. How would you rate the quality of the summary and the provided solution? *

Mark only one oval.

	1	2	3	4	5	
Very	◯	◯	◯	◯	◯	Very good

16. Could you see yourself using the provided solution in a real-world scenario? *

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Extremely

17. Do you see the provided problem and solution as a useful tool for addressing similar issues or problems in the future? *

Mark only one oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Exteremly

BIBLIOGRAPHY

A. Bacchelli, M. D. & Lanza, M. (2010). Extracting Source Code from E-Mails. *IEEE 18th International Conference on Program Comprehension*, 24–33.

Aggarwal, K., Hindle, A. & Stroulia, E. (2014). Co-evolution of project documentation and popularity within github. *Proceedings of the 11th working conference on mining software repositories*, pp. 360–363.

Alambo, A., Lohstroh, C., Madaus, E., Padhee, S., Foster, B., Banerjee, T., Thirunarayan, K. & Raymer, M. (2020, dec). Topic-Centric Unsupervised Multi-Document Summarization of Scientific and News Articles. *2020 IEEE International Conference on Big Data (Big Data)*, pp. 591-596. doi: 10.1109/BigData50022.2020.9378403.

Alhaj, F., Al-Haj, A., Sharieh, A. & Jabri, R. (2022). Improving Arabic Cognitive Distortion Classification in Twitter using BERTopic. *International Journal of Advanced Computer Science and Applications*, 13(1). doi: 10.14569/IJACSA.2022.0130199.

Allamanis, M., Barr, E. T., Devanbu, P. & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4), 1–37.

Alon, U., Zilberstein, M., Levy, O. & Yahav, E. (2019). Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.*, 3(POPL). doi: 10.1145/3290353.

Barbella, M. & Tortora, G. Rouge Metric Evaluation for Text Summarization Techniques. *Available at SSRN 4120317*.

Barua, A., Thomas, S. W. & Hassan, A. E. (2014). What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical software engineering*, 19, 619–654.

Basili, V. R., Selby, R. W. & Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7), 733-743. doi: 10.1109/TSE.1986.6312975.

Bianchini, M., Gori, M. & Scarselli, F. (2005). Inside PageRank. *ACM Trans. Internet Technol.*, 5(1), 92–128. doi: 10.1145/1052934.1052938.

Blumenthal, M., Luo, G., Schilling, M., Holme, H. C. M. & Uecker, M. (2023). Deep, deep learning with BART. *Magnetic Resonance in Medicine*, 89(2), 678–693.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1), 107-117. doi: https://doi.org/10.1016/S0169-7552(98)00110-X. Proceedings of the Seventh International World Wide Web Conference.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. & Amodei, D. (2020). Language Models Are Few-Shot Learners. *Proceedings of the 34th International Conference on Neural Information Processing Systems*, (NIPS'20).

Dagenais, B. & Robillard, M. P. (2012). Recovering traceability links between an API and its learning resources. *2012 34th International Conference on Software Engineering (ICSE)*, pp. 47-57. doi: 10.1109/ICSE.2012.6227207.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186. doi: 10.18653/v1/N19-1423.

Efstathiou, V., Chatzilenas, C. & Spinellis, D. (2018). Word Embeddings for the Software Engineering Domain. *Proceedings of the 15th International Conference on Mining Software Repositories*, (MSR '18), 38–41. doi: 10.1145/3196398.3196448.

Egger, R. & Yu, J. (2022). A Topic Modeling Comparison Between LDA, NMF, Top2Vec, and BERTopic to Demystify Twitter Posts. *Frontiers in Sociology*, 7. doi: 10.3389/fsoc.2022.886498.

El-Kassas, W. S., Salama, C. R., Rafea, A. A. & Mohamed, H. K. (2021). Automatic text summarization: A comprehensive survey. *Expert Systems with Applications*, 165, 113679.

Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M. & Sutton, C. (2017). Autofolding for source code summarization. *IEEE Transactions on Software Engineering*, 43(12), 1095–1109.

Ganesan, K. (2021, Sep). What is Rouge and how it works for evaluation of summaries? Retrieved from: https://kavita-ganesan.com/what-is-rouge-and-how-it-works-for-evaluation-of-summaries/#.Y21LvuzML9s.

Gao, J., Yang, X., Jiang, Y., Liu, H., Ying, W. & Zhang, X. (2018). Jbench: A Dataset of Data Races for Concurrency Testing. *Proceedings of the 15th International Conference on Mining Software Repositories*, (MSR '18), 6–9. doi: 10.1145/3196398.3196451.

García-Pérez, M. (2012). Statistical Conclusion Validity: Some Common Threats and Simple Remedies. *Frontiers in Psychology*, 3. doi: 10.3389/fpsyg.2012.00325.

Gleich, D. F. (2015). PageRank beyond the Web. *siam REVIEW*, 57(3), 321–363.

Godwin, M., Ruhland, L., Casson, I., MacDonald, S., Delva, D., Birtwhistle, R., Lam, M. & Seguin, R. (2004). Pragmatic controlled clinical trials in primary care: the struggle between external and internal validity. *BMC medical research methodology*, 3, 28. doi: 10.1186/1471-2288-3-28.

Graham, Y. (2015). Re-evaluating Automatic Summarization with BLEU and 192 Shades of ROUGE. *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 128–137. doi: 10.18653/v1/D15-1013.

Grootendorst, M. R. (2022). BERTopic: Neural topic modeling with a class-based TF-IDF procedure. *ArXiv*, abs/2203.05794. Retrieved from: https://api.semanticscholar.org/CorpusID:247411231.

Guerrouj, L., Bourque, D. & Rigby, P. C. (2015). Leveraging Informal Documentation to Summarize Classes and Methods in Context. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2, 639-642. doi: 10.1109/ICSE.2015.212.

Harirpoosh, A. (2021). *Summarizing Code Entities Discussed in Stack Overflow Using Unsupervised Learning*. (Master's thesis, École de technologie supérieure ÉTS).

Hartman, V. & Campion, T. R. (2022). A Day-to-Day Approach for Automating the Hospital Course Section of the Discharge Summary. *AMIA Annual Symposium Proceedings*, 2022, 216.

Hassan, M. & Hill, E. (2018). Toward Automatic Summarization of Arbitrary Java Statements for Novice Programmers. *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 539-543. doi: 10.1109/ICSME.2018.00063.

Hong, K., Conroy, J., Favre, B., Kulesza, A., Lin, H. & Nenkova, A. (2014). A Repository of State of the Art and Competitive Baseline Summaries for Generic News Summarization. *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pp. 1608–1616. Retrieved from: http://www.lrec-conf.org/proceedings/lrec2014/pdf/1093_Paper.pdf.

Hou, L., Hu, P. & Bei, C. (2017). Abstractive document summarization via neural model with joint attention. *National CCF conference on natural language processing and Chinese computing*, pp. 329–338.

Hsu, W.-T., Lin, C.-K., Lee, M.-Y., Min, K., Tang, J. & Sun, M. (2018). A unified model for extractive and abstractive summarization using inconsistency loss. *arXiv preprint arXiv:1805.06266*.

Hu, X., Li, G., Xia, X., Lo, D. & Jin, Z. (2018a). Deep Code Comment Generation. (ICPC '18), 200–210. doi: 10.1145/3196321.3196334.

Hu, X., Li, G., Xia, X., Lo, D., Lu, S. & Jin, Z. (2018b, 7). Summarizing Source Code with Transferred API Knowledge. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 2269–2275. doi: 10.24963/ijcai.2018/314.

Huang, D., Cui, L., Yang, S., Bao, G., Wang, K., Xie, J. & Zhang, Y. (2020). What have we achieved on text summarization? *arXiv preprint arXiv:2010.04529*.

Iyer, S., Konstas, I., Cheung, A. & Zettlemoyer, L. (2016). Summarizing source code using a neural attention model. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083.

Jagan, B., Parthasarathi, R. & Geetha, T. (2018). Graph-Based Abstractive Summarization: Compression of Semantic Graphs. In *Innovations, Developments, and Applications of Semantic Web and Information Systems* (pp. 236–261). IGI Global.

Jelodar, H., Wang, Y., Yuan, C., Feng, X., Jiang, X., Li, Y. & Zhao, L. (2019). Latent Dirichlet allocation (LDA) and topic modeling: models, applications, a survey. *Multimedia Tools and Applications*, 78, 15169–15211.

Jiang, S., Armaly, A. & McMillan, C. (2017). Automatically Generating Commit Messages from Diffs Using Neural Machine Translation. (ASE 2017), 135–146.

Kavaler, D., Posnett, D., Gibler, C., Chen, H., Devanbu, P. & Filkov, V. (2013). Using and asking: Apis used in the android market and asked about in stackoverflow. *International Conference on Social Informatics*, pp. 405–418.

Kim, Y. J., Cheong, Y. G. & Lee, J. H. (2019). Prediction of a Movie's Success From Plot Summaries Using Deep Learning Models. *Proceedings of the Second Workshop on Storytelling*, pp. 127–135. doi: 10.18653/v1/W19-3414.

Korayem, E. M. (2019). *Towards Automatic Context-Aware Summarization of Code Entities*. (Master's thesis, École de technologie supérieure ÉTS).

Le, T. H., Chen, H. & Babar, M. A. (2020). Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys (CSUR)*, 53(3), 1–38.

LeClair, A., Jiang, S. & McMillan, C. (2019). A Neural Model for Generating Natural Language Summaries of Program Subroutines. *Proceedings of the 41st International Conference on Software Engineering*, (ICSE '19), 795–806. doi: 10.1109/ICSE.2019.00087.

LeClair, A., Haque, S., Wu, L. & McMillan, C. (2020). Improved Code Summarization via a Graph Neural Network. *Proceedings of the 28th International Conference on Program Comprehension*, (ICPC '20), 184–195. doi: 10.1145/3387904.3389268.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V. & Zettlemoyer, L. (2020). BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880. doi: 10.18653/v1/2020.acl-main.703.

Li, B. & Han, L. (2013). Distance Weighted Cosine Similarity Measure for Text Classification. *Intelligent Data Engineering and Automated Learning*.

Li, W. & Zhao, J. (2016). TextRank algorithm by exploiting Wikipedia for short text keywords extraction. *2016 3rd International Conference on Information Science and Control Engineering (ICISCE)*, pp. 683–686.

Liang, Y. & Zhu, K. Q. (2018). Automatic Generation of Text Descriptive Comments for Code Blocks. (AAAI'18/IAAI'18/EAAI'18).

Lin, C.-Y. (2004). ROUGE: A Package for Automatic Evaluation of Summaries. *Text Summarization Branches Out*, pp. 74–81. Retrieved from: https://aclanthology.org/W04-1013.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L. & Stoyanov, V. (2019). RoBERTa: A Robustly Optimized BERT Pretraining Approach. *ArXiv*, abs/1907.11692.

Luhn, H. P. (1958). The automatic creation of literature abstracts. *IBM Journal of Research and Development*, 2(8), 159–165.

Ma, C., Zhang, W. E., Guo, M., Wang, H. & Sheng, Q. Z. (2022). Multi-Document Summarization via Deep Learning Techniques: A Survey. *ACM Comput. Surv.* doi: 10.1145/3529754. Just Accepted.

McBurney, P. W. & McMillan, C. (2016). Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering*.

McBurney, P. W., Liu, C. & McMillan, C. (2016). Automated Feature Discovery via Sentence Selection and Source Code Summarization. *J. Softw. Evol. Process*, 28(2), 120–145. doi: 10.1002/smr.1768.

McGill, R., Tukey, J. W. & Larsen, W. A. (1978). Variations of Box Plots. *The American Statistician*, 32(1), 12–16. Retrieved from: http://www.jstor.org/stable/2683468.

McInnes, L., Healy, J. & Melville, J. (2020). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction.

Meier, R. (2008). How can I save an activity state using the Save Instance State? Retrieved from: https://stackoverflow.com/questions/151777/how-can-i-save-an-activity-state-using-the-save-instance-state.

Mihalcea, R. & Tarau, P. (2004). TextRank: Bringing Order into Text. *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pp. 404–411. Retrieved from: https://aclanthology.org/W04-3252.

Miller, D. (2019). Leveraging BERT for Extractive Text Summarization on Lectures.

Moore, J., Gelman, B. & Slater, D. (2019). A Convolutional Neural Network for Language-Agnostic Source Code Summarization. *ENASE*.

Moratanch, N. & Chitrakala, S. (2016). A survey on abstractive text summarization. *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pp. 1-7. doi: 10.1109/ICCPCT.2016.7530193.

Moratanch, N. & Chitrakala, S. (2017). A survey on extractive text summarization. *2017 International Conference on Computer, Communication and Signal Processing (ICCCSP)*, pp. 1-6. doi: 10.1109/ICCCSP.2017.7944061.

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L. & Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for Java classes. *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 23-32. doi: 10.1109/ICPC.2013.6613830.

Moreno, L. & Marcus, A. (2012). JStereoCode: automatically identifying method and class stereotypes in Java code. *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 358-361. doi: 10.1145/2351676.2351747.

M'rabet, Y. & Demner-Fushman, D. (2020). HOLMS: Alternative summary evaluation with large language models. *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 5679–5688.

Naghshzan, A. (2023a). BERTopic. Retrieved from: https://github.com/amirarcane/BERTopic.

Naghshzan, A. (2023b). Towards Code Summarization of APIs Based on Unofficial Documentation Using NLP Techniques.

Naghshzan, A. & Ratte, S. (2023a). Enhancing API Documentation through BERTopic Modeling and Summarization.

Naghshzan, A. & Ratte, S. (2023b). Revolutionizing API Documentation through Summarization.

Naghshzan, A., Guerrouj, L. & Baysal, O. (2021). Leveraging Unsupervised Learning to Summarize APIs Discussed in Stack Overflow. *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 142-152. doi: 10.1109/SCAM52516.2021.00026.

Naghshzan, A., Khalilazar, S., Poilane, P., Baysal, O., Guerrouj, L. & Khomh, F. (2023a). Leveraging Data Mining Algorithms to Recommend Source Code Changes.

Naghshzan, A., Ratté, S., Guerrouj, L. & Baysal, O. (2023b). Enhancing API Documentation: A Combined Summarization and Topic Modeling Approach. *International Journal of Software Engineering and Knowledge Engineering*.

Opidi, A. (2019). FloydHub Blog. Retrieved from: https://blog.floydhub.com/gentle-introduction-to-text-summarization-in-machine-learning/.

Page, L., Brin, S., Motwani, R. & Winograd, T. (1998). The PageRank citation ranking: Bringing order to the Web. *Proceedings of the 7th International World Wide Web Conference*, pp. 161–172. Retrieved from: citeseer.nj.nec.com/page98pagerank.html.

Pang, R. Y., Lelkes, A., Tran, V. & Yu, C. (2021). AgreeSum: Agreement-Oriented Multi-Document Summarization. *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 3377–3391. doi: 10.18653/v1/2021.findings-acl.299.

Papineni, K., Roukos, S., Ward, T. & Zhu, W.-J. (2002). BLEU: A Method for Automatic Evaluation of Machine Translation. (ACL '02), 311–318. doi: 10.3115/1073083.1073135.

Parnin, C. & Treude, C. (2011). Measuring API Documentation on the Web. *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, (Web2SE '11), 25–30. doi: 10.1145/1984701.1984706.

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R. & Lanza, M. (2014). Mining StackOverflow to Turn the IDE into a Self-Confident Programming Prompter. *Proceedings of the 11th Working Conference on Mining Software Repositories*, (MSR 2014), 102–111. doi: 10.1145/2597073.2597077.

Ponzanelli, L., Bavota, G., Di Penta, M., Oliveto, R. & Lanza, M. (2015). Turning the IDE into a self-confident programming assistant.

Qin, Y. & Specia, L. (2015, 11–13). Truly Exploring Multiple References for Machine Translation Evaluation. *Proceedings of the 18th Annual Conference of the European Association for Machine Translation*. Retrieved from: https://aclanthology.org/2015.eamt-1.16.

Rigby, P. C. & Robillard, M. P. (2013). Discovering essential code elements in informal documentation. *32nd ACM/IEEE International Conference on Software Engineering*, 832–841.

Robillard, M. P. (2009). What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6), 27-34. doi: 10.1109/MS.2009.193.

Rodeghero, P., Liu, C., McBurney, P. W. & McMillan, C. (2015). An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization. *IEEE Transactions on Software Engineering*, 41(11), 1038-1054.

S. Rastkar, G. C. M. & Murray, G. (2014). Automatic summarization of bug reports. *IEEE Trans. Software Eng*, 40(4), 366–380.

Saddler, J. A., Peterson, C. S., Sama, S., Nagaraj, S., Baysal, O., Guerrouj, L. & Sharif, B. (2020). Studying Developer Reading Behavior on Stack Overflow during API Summarization Tasks. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 195-205. doi: 10.1109/SANER48275.2020.9054848.

Shapiro, S. S. & Wilk, M. B. (1965). An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4), 591–611. Retrieved from: http://www.jstor.org/stable/2333709.

Song, S., Huang, H. & Ruan, T. (2019). Abstractive Text Summarization Using LSTM-CNN Based Deep Learning. *Multimedia Tools Appl.*, 78(1), 857–875. doi: 10.1007/s11042-018-5749-3.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. & Vijay-Shanker, K. (2010). Towards Automatically Generating Summary Comments for Java Methods. (ASE '10), 43–52. doi: 10.1145/1858996.1859006.

Stapleton, S., Gambhir, Y., LeClair, A., Eberhart, Z., Weimer, W., Leach, K. & Huang, Y. (2020). A Human Study of Comprehension and Code Summarization. *Proceedings of the 28th International Conference on Program Comprehension*, (ICPC '20), 2–13. doi: 10.1145/3387904.3389258.

Student. (1908). The probable error of a mean. *Biometrika*, 1–25.

Su, D., Xu, Y., Yu, T., Siddique, F. B., Barezi, E. & Fung, P. (2020). CAiRE-COVID: A Question Answering and Query-focused Multi-Document Summarization System for COVID-19 Scholarly Information Management. *Proceedings of the 1st Workshop on NLP for COVID-19 (Part 2) at EMNLP 2020*. doi: 10.18653/v1/2020.nlpcovid19-2.14.

Subramanian, S., Inozemtseva, L. & Holmes, R. (2014a). Live API Documentation. (ICSE 2014), 643–652. doi: 10.1145/2568225.2568313.

Subramanian, S., Inozemtseva, L. & Holmes, R. (2014b). Live API Documentation. *Proceedings of the 36th International Conference on Software Engineering*, (ICSE 2014), 643–652. doi: 10.1145/2568225.2568313.

Sun, W., Fang, C., Chen, Y., Zhang, Q., Tao, G., Han, T., Ge, Y., You, Y. & Luo, B. (2022). An Extractive-and-Abstractive Framework for Source Code Summarization. *arXiv preprint arXiv:2206.07245*.

Tang, R., Nogueira, R., Zhang, E., Gupta, N., Cam, P., Cho, K. & Lin, J. (2020). Rapidly Bootstrapping a Question Answering Dataset for COVID-19. arXiv. Retrieved from: https://arxiv.org/abs/2004.11339.

Tas, O. & Kiyani, F. (2007). A survey automatic text summarization. *PressAcademia Procedia*, 5(1), 205–213.

T.B. Brown,B. Mann, N. R. M. S. J. K. P. D. A. N. P. S. G. S. A. A. e. a. (2005). Language models are few-shot learners. *arXiv*.

Tran, N., Tran, H., Nguyen, S., Nguyen, H. & Nguyen, T. (2019). Does BLEU Score Work for Code Migration? *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pp. 165-176. doi: 10.1109/ICPC.2019.00034.

Treude, C. & Robillard, M. P. (2016). Augmenting API Documentation with Insights from Stack Overflow. *Proceedings of the 38th International Conference on Software Engineering*, (ICSE '16), 392–403. doi: 10.1145/2884781.2884800.

Uddin, G. & Khomh, F. (2017). Automatic summarization of API reviews. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 159-170. doi: 10.1109/ASE.2017.8115629.

Uddin, G., Khomh, F. & Roy, C. K. (2020). Mining API usage scenarios from stack overflow. *Information and Software Technology*, 122, 106277. doi: https://doi.org/10.1016/j.infsof.2020.106277.

van Rijsbergen, C. (1979). *Information Retrieval*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017). Attention is All You Need. Retrieved from: https://arxiv.org/pdf/1706.03762.pdf.

Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J. & Yu, P. S. (2018). Improving automatic source code summarization via deep reinforcement learning. *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 397–407.

Wang, S., Zhao, X., Li, B., Ge, B. & Tang, D. (2017). Integrating extractive and abstractive models for long text summarization. *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 305–312.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B. & Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q. & Rush, A. (2020). Transformers: State-of-the-Art Natural Language Processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45. doi: 10.18653/v1/2020.emnlp-demos.6.

Wu, Y. & Hu, B. (2018). Learning to Extract Coherent Summary via Deep Reinforcement Learning. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, (AAAI'18/IAAI'18/EAAI'18).

Yang, A., Liu, K., Liu, J., Lyu, Y. & Li, S. (2018). Adaptations of ROUGE and BLEU to Better Evaluate Machine Reading Comprehension Task. *Proceedings of the Workshop on Machine Reading for Question Answering*, pp. 98–104. doi: 10.18653/v1/W18-2611.

Zhang, H., Xu, J. & Wang, J. (2019). Pretraining-based natural language generation for text summarization. *arXiv preprint arXiv:1902.09243*.

Zhu, Y. & Pan, M. (2019). Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*.