Summarizing Code Entities Discussed in Stack Overflow Using Unsupervised Learning

by

Andisheh Harirpoosh

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN ELECTRICAL ENGINEERING
M.A.Sc.

MONTREAL, FEBRUARY 22, 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Professor Latifa Guerrouj, for her valuable support and guidance throughout my research study. Her positive energy and valuable advice have provided me with the motivation needed to be productive and to make of my Master, a pleasant experience. I feel very fortunate to have her as a supervisor.

I am also thankful to my other Jury members: Professor Christopher Fuhrman and Professor Diala Naboulsi, for their insightful comments and feedback.

I also would like to thank my labmate, AmirHossein Naghshzan, for our fruitful collaboration and for his contribution to the data collection phase of this research work.

This work is dedicated to my dad, Mohammad Javad Harirpoosh, and my sister, Elham Harirpoosh, as well as my partner Kambiz Nezami, for their unconditional love and support.

Synthétiser les entités de code discutées dans le Stack Overflow en utilisant l'apprentissage non-supervisé

Andisheh Harirpoosh

RÉSUMÉ

La synthèse des entités de code a été proposée pour une meilleure compréhension des entités de code et pour optimiser le temps et l'effort des développeurs tout en fournissant des descriptions exactes du code. Les synthèses du code produites sont connues d'être soit du type extractive ou abstractive en génie logiciel. De multiples méthodes et algorithmes tels que les techniques de recherche d'information ou d'apprentissage machine ont été appliquées pour générer automatiquement les synthèses des entités de code. Leur défi commun est que très peu d'entre eux ont proposé des façons complétement automatiques pour la génération des synthèses du code sans intervention humaine. En plus, la plupart ont traité les entités de code faisant partie du code et non pas de la documentation informelle et ne mettent pas l'emphase nécessairement sur l'usage des entités de code. Dans ce mémoire, je fais appel à l'apprentissage machine et en particulier à la technique TextRank pour extraire les phrases pertinentes qui doivent faire partie des synthèses produites pour les méthodes Android discutées dans le Stack Overflow. En plus, j'utilise les techniques du traitement de langage naturel en particulier les techniques Term Frequency-Inverse Document Frequency (TF-IDF) et word embedding qui sont des techniques de vectorisation, comme deux techniques de prétraitement du texte. En outre, je montre à quel degré ces deux techniques affectent la qualité des synthèses produites. Pour évaluer mon travail de recherche, j'ai mené une étude empirique faisant intervenir 3084143 discussions de Stack Overflow de la période de 2009 au Avril 2020. et 20 méthodes de la plateforme Android. Les résultats de cette étude ont démontré que nous pouvons générer des synthèses de façon complètement automatique pour les entités de code discutées en Stack Overflow en utilisant l'apprentissage machine. Ils ont indiqués aussi que la technique de vectorisation word embedding a un effet plus positif que TF-IDF lorsqu'appliquée à la technique d'apprentissage machine TextRank.

Mots-clés: Synthèse du code, entités de code, StackOverflow, apprentissage non supervisé, traitement du langage naturel, documentation informelle.

Summarizing Code Entities Discussed in Stack Overflow Using Unsupervised Learning

Andisheh Harirpoosh

ABSTRACT

Automated source code summarization is an approach that generates natural language descriptions that describe the code entities such as methods and classes to better understand the code and to optimize the time and effort of software developers. There are two ways to summarize code: extractive way as well as abstractive way. Multiple methods and algorithms including Information Retrieval techniques and supervised learning have been applied to generate automatic summaries for code entities. Their common shortcoming is that very little to none of those works has offered a completely automated way, without human intervention, to provide accurate summaries for code entities discussed in informal documentation such as emails exchanged between developers, Stack Overflow, etc. In addition, very little research has investigated informal documentation when summarizing code entities discussed in code summarization. In this thesis, I leverage unsupervised learning, *i.e.*, TextRank to extract relevant sentences that should be part of the summary produced for Android methods mentioned in Stack Overflow. In addition, I use Natural Language Processing (NLP) techniques, in particular, Term Frequency-Inverse Document Frequency (TF-IDF) and Stack Overflow's word embedding, which are used as text vectorization techniques, as two separate techniques for text vectorization, and I show how different vectorization affect the quality of the produced summaries. To evaluated my research work, I conducted an empirical study that involves 3,084,143 Stack Overflow posts from Stack Overflow tagged between 2009 and April 2020 and 20 Android methods. The results of our investigation show that we can produce automatic summaries for code entities discussed in informal documentation using TextRank and that applying a word embedding vectorization provides more stable results than the TF-IDF when applied with TextRank algorithms.

Keywords: Code Summarization, Code entities, Stack Overflow, Natural Language Processing, Informal documentation, Official Documentation, Cosine Similarity, TextRank, Similarity, Word2Vec

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABREVIATIONS

ETS	École de Technologie Supérieure
API	Application Programming Interface
AST	Abstract Syntax Tree
BR	Bug Reports
NL	Natural Language
NLP	Natural Language Processing
NLTK	Natural Language ToolKit
RQ	Research Question
SC	Source Code
SO	Stack Overflow
SWUM	Software Word Usage Mode
TF-IDF	Term Frequency-Inverse Document Frequency

INTRODUCTION

The effects of Natural language code summarization are significantly enhanced today in many areas such as software understanding, modification, and maintenance. High-quality comments in the code as well as up-to-date comments are important when the code changes. Code summaries allow developers to spend less time updating the code to meet the up-to-date software requirements. For this reason, automated source code summarization is one of the challenges that many researchers consider today. There are various sources that researchers have used as inputs for code summarization including source code, bug reports, as well as online discussion forums such as Stack Overflow. However, source code has been extensively used for code summarization. Additionally, very little research has been done to produce completely automated ways to summarize, without human intervention, code entities discussed in informal documentation. Furthermore, there has been very little work that has attempted to investigate and summarize the usage of these code entities. The purpose of this thesis is to suggest an automatic approach based on unsupervised learning to summarize the usage of code entities, *i.e.*, Android methods in our case, discussed in Stack Overflow, which is the only source of information to collect the usage of Android methods. Our technique automatically generates summaries on the usage of Android methods in informal documentation. In this thesis, we have extracted all Android posts from Stack Overflow tagged between 2009 and April 2020 and used them as an input of our approach. Then, we have identified posts according to the methods that they discuss. The output of this step is a list of posts that corresponds to each method in question, which constitutes the corpus of each studied method. As unsupervised learning technique, we have applied, TextRank, to generate an automatic natural language summary for each corpus. The main goal of the generated summaries is to provide developers with descriptions on the usage of Android methods.

Motivation

Developers need a summary of the usage of methods in different domains, such as Android, Python, Java, etc, to quickly get the overall understanding of how to use them in their source code. Also, they are generally curious to know about the gist of methods that are part of their daily routine tasks. Although various types of information sources have been introduced to help developers access methods' descriptions, there are not a lot of investigations on informal documentation. Providing developers with summaries of quality for methods is crucial, since it can guarantee that they use methods properly in their tasks. Among the information sources that have been leveraged, we find Stack Overflow that has gained more attention due to its characteristics, including an open community and platform where the developers can share their knowledge to find answers to programming questions (Mamykina, Manoim, Mittal, Hripcsak & Hartmann (2011), Guerrouj, Bourque & Rigby (2015)). And also Abdalkareem, Shihab & Rilling (2017) indicate that Stack Overflow can be a good source of information to investigate the use of code in a mobile app. We focus on Android since it is one of the top eight most discussed topics on Stack Overflow based on the type of tags assigned to questions[1], and also it is one of the most popular mobile platforms[2]. This indicates a tremendous number of developers ranging from novice to experts, seeking usage of methods belonging to the Android domain. Understanding the usage of methods in Stack Overflow, where there are different posts for each method, is not only a difficult and time-consuming task but effort-demanding since a large amount of information needs to be investigated. On the other hand, software developers need to have a complete and accurate understanding of methods and their application. To summarize the usage of methods, we extract pertinent information from all the posts in Stack Overflow that discuss the Android methods in question. We present in this thesis a novel approach that is based on unsupervised learning to extract the usage of methods from the Android

[1] https://en.wikipedia.org/wiki/Stack_Overflow

[2] http://developer.android.com/about/index.html

Stack Overflow. The summaries generated are of an extractive type. That is, sentences are selected from the whole text that express the main idea of the corpus without losing meaning.

Problem statement

In this work, we focus on summarizing the usage of methods discussed in Stack Overflow posts to help developers understand quickly the purpose and usage of code entities that are part of their task. However, descriptions of quality for the usage of Android methods are required for the process of searching and understanding code, and due to the various sources of information available, one can spend more time trying to choose between multiple sources for a certain method. In addition, there are lots of duplicate posts in Q&A forums. As Q&A forums contain lots of comments from different users, software developers need to spend more time to find the appropriate and accurate descriptions related to the usage of methods that are parts of their tasks. Moreover, researchers have shown that official documentation lacks insights, completeness, and conciseness (Robillard (2009), Uddin & Robillard (2015)), which means that informal documentation is needed to gain more insights on the usage of code entities. Exploring informal documentation to gain such insights can be time consuming since one has to navigate through all available documentation to understand the usage of a specific code entity. Therefore, automatic code summarization approaches that produce accurate and insightful code summaries are needed to overcome these challenges and help developers spend less time when understanding the usage of code entities that perplex them.

Code summarization has been studied in different articles. Based on the techniques leveraged in these research works, we can find summarization techniques based on Information Retrieval (IR) which is investigated in Haiduc, Aponte, Moreno & Marcus (2010), Natural Language Process (NLP) (Sridhara, Hill, Muppaneni, Pollock & Vijay-Shanker (2010)), Stereotype Identification (Moreno, Aponte, Sridhara, Marcus, Pollock & Vijay-Shanker (2013), and Program Analysis methods. Using these techniques brings some benefits such as finding a structured or unstructured nature in IR and generating text through artificial intelligence. However, they can also bring some challenges, such as contextual weaknesses (McBurney, Liu & McMillan (2016)). To perform optimization, several parameters must be collected from the artifacts and combined with different mining methods, which complicates such approaches. Issues such as interoperability between artifacts, different extraction methods, as well as learning from artifacts while generating automatic code summarization should be considered.

The concept of code summarization has received more attention with the use of machine learning and artificial neural network (ANN). Machine learning techniques provide the ability to create summaries using artifact training and test sets in supervised summarization. This technique is used in bug reports summarization (Rastkar, Murphy & Murray (2010)). However, if the study is focused on latent data in unlabeled dataset (McBurney *et al.* (2016)) unsupervised techniques in machine learning are used. By leveraging code summarization, technologies such as Artificial neural network (ANN) has emerged by Iyer et al. (Iyer, Konstas, Cheung & Zettlemoyer (2016)). CODENN (Iyer *et al.* (2016)) and DeepCom (Hu, Li, Xia, Lo & Jin (2018)) is an example of such ANN techniques aiming at summarizing source code automatically.

Outline

This thesis consists of five chapters. In the first chapter, we give an introduction about our research, along with the motivation and problem statement. Chapter 2 is about the literature review related to our study. Our contribution is detailed on in Chapter 3, with the explanation of each of its components. We also show in this chapter how our novel approach generates automated summaries for Android methods. Chapter 4 presents the evaluation of the proposed approach as well as the results of our empirical study. In the last chapter, we conclude our work and discuss our future research directions.

CHAPTER 1

LITERATURE REVIEW

Recently, providing a useful and beneficial summary for the usage of code in the software engineering literature has been investigated. One of the main challenges faced in code summarization is the type of artifacts and techniques used when summarizing code. Modifying, maintaining, and understanding source code is needed in the industrial, scientific, and medical spectrum. Therefore summarizing code is crucial, and generated summaries should be accurate in order not to mislead developers. In this study, we focus on summarizing code entities discussed in Stack Overflow. Any other artifacts are out of the scope of this research. Choosing the right artifacts and techniques is essential when it comes to producing summaries of quality. Several artifacts have been leveraged for code summarization such as source code investigated in Sridhara *et al.* (2010); McBurney & McMillan (2015); McBurney *et al.* (2016), bug reports (BR) analyzed in Jiang, Nazar, Zhang, Zhang & Ren (2017a); Rastkar *et al.* (2010); Rastkar, Murphy & Murray (2014); Korayem (2019), code changes leveraged in Ying, Murphy, Ng & Chu-Carroll (2004)), as well as Online discussion forums as in Saddler, Peterson, Sama, Nagaraj, Baysal, Guerrouj & Sharif (2020); Guerrouj *et al.* (2015). Nazar, Hu & Jiang (2016) demonstrated that among all these artifacts, source code has been extensively used due to the its characteristics. Nazar *et al.* (2016) consider that in this type of artifact, not only structured data, for example, semantics, syntax exist, but also non-structured data such as comments, identifiers exist.

Also, the quality of code summaries may be affected by several shortcomings in source code such as abbreviations and acronyms. Also, the descriptions of code entities that exist in code source are most often related to implementation rather than usage. Therefore, considering other sources when summarizing usage may be beneficial. Also, high-quality summaries with high accuracy, content adequacy, and conciseness are required to meet the needs of newcomers and software developers (McBurney *et al.* (2016)). For this reason, it is necessary to use the appropriate artifacts to accomplish this goal. Due to the nature of the results of the code summarization which are presented in the next section, Stack Overflow seems to be an interesting source of information when it comes to describing the usage of code entities. In the following, I will present the most relevant approaches on summarization related to our work.

Moreno *et al.* (2013) proposed an approach which is a combination of stereotype Identifiers (Moreno & Marcus (2012)) and Lexicalization (Sridhara *et al.* (2010)) to understand the content and responsibility of classes. Although in this approach the focus of summarization is not on the relationships between the classes, it is based on heuristics and creates Natural language summaries for classes. The purpose of this approach is to produce text from the content selection with respect to stereotype identification. A manual evaluation was done to validate the content adequacy, conciseness, and readability of the produced summaries. The results of this evaluation have shown that the proposed approach can be used as structured natural language summary for Java classes. Contrary to their work of summarizing Java classes based on a Java project, we use code summarization based on informal documentation, and we also do this code summarization at the level of Android methods, not classes.

Rodeghero, Liu, McBurney & McMillan (2015) suggested an approach to summarize methods without using source code comments. They conducted a study on eye-tracking and eye-movement based on a Vector Space Model (VSM) (Haiduc *et al.* (2010)) and Information Retrieval (IR) to find important keywords and list them according to a gaze time of the expert developers. They focused solely on method signature, method call, and control flow. They found that the method signature was more important than methods calls in source code. As mentioned in Abid *et al.* (Abid, Sharif, Dragan, Alrasheed & Maletic (2019)), Rodeghero *et al.* did not consider the method name, data declaration or another relevant method, outside of a method in question. Another limitation of this approach is the fact that methods with LOC greater than 22 are removed. For evaluating their approach, Mann-Whitney and Wilcoxon Signed Rank tests were used. Unlike Rodeghero *et al.*, our focus is on informal documentation. Also, we have applied unsupervised learning to generate the summary on the usage of code entities.

In a more recent article, Phan, Chau, Le Nguyen & Bui (2018) have proposed a source code summarization approach based on k-Nearest Neighbors (kNN-TED) and SVM neural network.

The results have shown that using this approach not only improves the classification performance but also the execution time. In this approach, the input of their system is an Abstract Syntax Tree (AST) to obtain a classification based on the functionality of the programs, as well as finding specific patterns for code snippets. To extract both structural and basic information from the ASTs, a combination of TBCNN + kNN-TED and TBCNN + SVM models was used to increase the classification accuracy. To reduce execution time and increase classification accuracy, heuristic techniques for pruning waste and remodeling under ASTs were proposed. Another reason for using these techniques is that high dimensional data not only losses time and memory but it also has an impact on the performance of the algorithms. The empirical evaluation of the results in Phan *et al.* (2018) has shown that the performance of this approach has improved compared to sequence-based or metrics-based classifiers. Unlike this work, we do not leverage source code but rather unofficial documentation and unsupervised learning to extract important sentences that can summarize the propose and usage of a code entities.

Sridhara *et al.* (2010) have used the Natural Language Process to extract information to summarize Java methods. In their approach, after pre-processing the source code by Software Word Usage Model (SWUM), which considers both structural and linguistic clues, the Natural Language summary is generated with a format-based model. The accuracy, content adequacy, and conciseness of the approach were manually evaluated by 13 participants. The result of their work demonstrated that the comments generated for Java methods from source code are accurate and do not miss significant information in the process of creating comments. However, the focus of our work is on summarizing Android methods from unofficial documentation, and our most important goal in this research is to generate the usage of methods, not to describe methods.

In another work that aims at generating documentation for Java methods, McBurney & McMillan (2015) have suggested a new approach for producing natural language text and leveraged the context around the method for the summarization purpose. To classify parts of speech, the authors have used SWUM for different terms. Additionally, they have used the PageRank algorithm and investigated method calls to produce summaries for Java methods (Cf. Figure 1.1). The output of this approach can be used to understand how to use this method and why it exists in a specific program. Adequacy, accuracy of contents have been evaluated. To validate their work, they have compared a summary of their approach with human summaries as well as with a summary of Sridhara *et al.* (Sridhara *et al.* (2010)) approach. The results have shown that although the summary produced in the human method was better in terms of accuracy and precision, their approach was improved in terms of quality than the Sridhara *et al.* (Sridhara *et al.* (2010)) approach. Contrary to their work, our focus is on producing a summary of Android methods extracted from Stack Overflow.

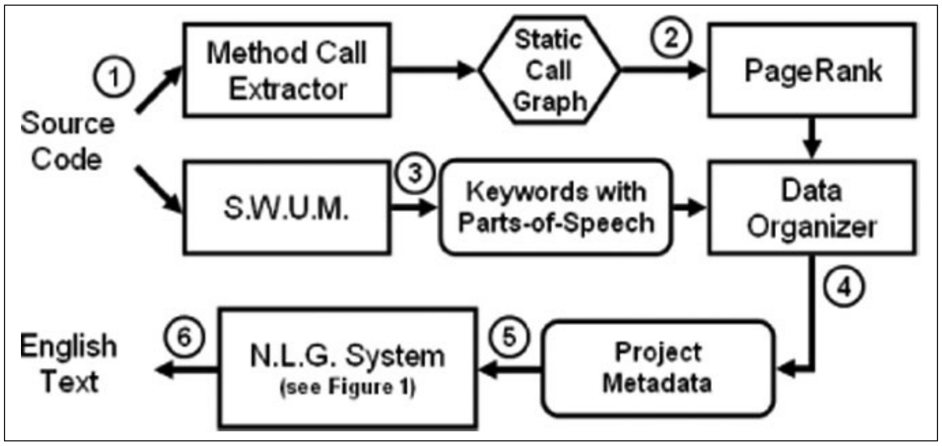


Figure 1.1 Overview Approach of Automatic source code summarization
Taken from McBurney & McMillan (2015)

McBurney *et al.* (2016) used the impact of four different approaches to create a list of features *i.e.*, a list of sentences that describe the main functionality from Java source code documentation. The summarization approach used in this paper is divided into two tools: the feature list tools and textual analysis tools. In the feature list tools, the Latent Dirichlet Allocation (LDA) approach has been used to generate a list of topics from the preparation sentences list. The outline of this approach is showed in figure 1.2. From the list of topics and the list of sentences, the feature list is generated by Lightweight Semantic Similarity (LSS) or Overlap. Extraction of sentence lists in textual analysis tools and features list tools is the same. However, in textual analysis tools, TextRank and TLDR tools have generated the feature list for a Java source code. To evaluate the quality and readability of the generated feature list, the authors have conducted two surveys and have used the Mann-Whitney test. The results of their evaluation have shown that none of the approaches was appropriate (McBurney *et al.* (2016)). In this study, they applied four different approaches to summarize the code. One of the approaches they chose was TextRank. The difference between our work and theirs is in the choosing of the artifact. They did the summary from the Java source code, but we got the summary from Stack Overflow posts that have the Android tag using the TextRank algorithm.

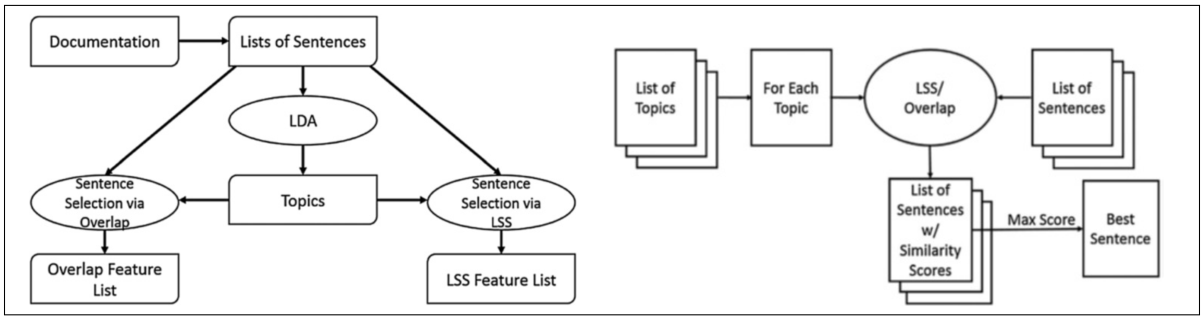


Figure 1.2 Overall Approach of Automated feature discovery
Taken from McBurney *et al.* (2016)

Hu *et al.* (2018) suggested an approach to generate code comments from large code corpus called DeepCom. They applied Natural Language Processing (NLP) techniques and solved some limitations like extracting accurate keywords from methods which are not named correctly. As it can be noticed from figure 1.3, when generating natural language descriptions from source code, both code structure and semantics information were considered. After a corpus of Java methods and their corresponding comments are obtained from GitHub, they are converted into an AST sequence and then fed to learn structural information. Then, based on a sequence-to-sequence model, descriptive comments for Java methods are generated from learned features. The language model that is used in this paper is based on a Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber (1997)) which is the state-of-the-art Recurrent Neural Networks (RNNs). The accuracy of the generated summaries was measured using the BLEU (Bilingual Evaluation Understudy) score which is an appropriate method to evaluate in machine translation (Papineni, Roukos, Ward & Zhu (2002)). Unlike their work, we focus on the usage of code entities discussed in Android, and our data were collected from Stack Overflow as a type of informal documentation.

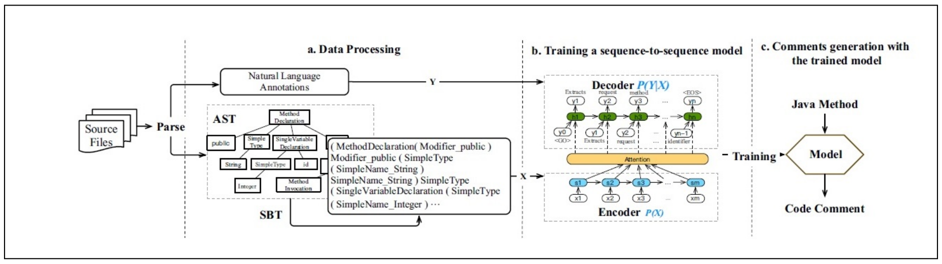

a. Data Processing
b. Training a sequence-to-sequence model
c. Comments generation with the trained model
Source Files
Parse
Natural Language Annotations
AST
SBT
Decoder P(Y|X)
Attention
Encoder P(X)
Training
Java Method
Model
Code Comment

Figure 1.3 Overall framework of DeepCom
Taken from Hu *et al.* (2018)

Jiang *et al.* (2017a) have applied PageRank-based Summarization Technique (PRST) to generate summaries for bug reports (BR), which is based on a PageRank and logistics regression algorithms. To find the similarity of sentences in this algorithm, they applied three distinct models namely Vector Space Model (VSM), WordNet, and Jaccard. The results showed that WordNet returns a better output than the other two models. Contrary to what they did, we used Stack Overflow post to summarize the usage of code entities. We used Cosine similarity to find sentence similarity, then PageRank, and finally we extracted three important sentences to generate our summaries.

In another work, Jiang, Armaly & McMillan (2017b) conducted a study to generate committed messages based on the difference between two versions of the source code with the Neural Machine Translation (NLT) algorithm. A corpus which is used for training their algorithm is prepared from a GitHub project. They use a quality assurance filter to qualify their output. This filter works by detecting inappropriate commit messages and replacing them with a warning message. We used the TextRank algorithm to generate a summary for the usage of Android methods from informal documentation. Our research is done automatically and without any human intervention.

One of the new methods for automated code summarization is using Artificial Neural Network (ANN). This method helps identify the latent relationship between structural source code and natural language descriptions. LeClair, Jiang & McMillan (2019) suggested a new approach that generates summaries for code even if there is no internal documentation such as the programmer's comments for methods, especially when there is no good naming. This approach is considered an improvement of the approach by Hu *et al.* (2018) that processes the words from source code to represent an AST. Unlike this research work, we used the TextRank algorithm to generate summaries on the usage of Android methods from informal documentation, in particular the Stack Overflow.

Saddler *et al.* (2020) conducted a study using eye-tracking methods to summarize API elements in Stack Overflow posts as input sources. To evaluate this method, the authors invited 30 participants, consisting of students and professionals, to evaluate the produced summaries. Although there was not much difference in the accuracy between the two groups, there was a difference in how each page's content was viewed. Unlike this work, we do not rely on eye-tracking methods, however we leverage unsupervised learning to generate summaries for the usage of Android methods from informal documentation. There are other limitations when it comes to summarizing code entities from source code. For example, one of these limitations is the need for fully documented projects that can lead to accurate summaries. Another limitation is that the output, in some cases, is not in a natural language format but in the form of keywords. Additionally, the applied algorithms have not been generalized to other programs or languages.

Overall, while a large body of research works have extensively focused on the use of source code to summarize code entities, very little to no work has investigated the use of informal documentation when summarizing code entities. We agree with these previous works that automatic code summarization can help reduce the time and efforts required to understand the purpose and usage of code entities. However, to overcome the shortcomings of previous works, we attempt to generate summaries that contain more insights and information about code entities. For such a purpose, we leverage informal documentation, *i.e.*, Stack Overflow and produce in a fully automatic way summaries in the form of full Natural Language sentences instead of a set of keywords.

CHAPTER 2

COMPONENTS OF THE PROPOSED APPROACH

Our automatic code summarization targets developers that need to understand the usage of code entities that are part of their software maintenance and–or evolution tasks. Automatic code summarization automatically generates a summary for code entities, *i.e.*, methods in our case. The summary includes the top three most relevant sentences extracted from online discussion forums. The relevance is based on the scores of sentences. The reason why we choose three sentences was that when we select more than three sentences, the output looks more like a paragraph instead of a brief description. Learning the meaning, semantics, and different types of contexts are assigned based on a pre-trained word embedding technique in large datasets of Stack Overflow posts which seems to yield the best result in the context of this research problem.

The main objective of this research is to generate a meaningful summary on the usage of code entities discussed in informal documentation, *i.e.*, methods in our case. Our conjecture is that developers discuss and exchange most often about code entities that perplex them. Several answers can be found for example on online discussion forums regarding the use of a specific method. Not all answers are relevant and not all information in a specific answer is pertinent. Our challenge is to find an appropriate way to extract relevant information about the usage of code entities and provide them in an automatic fashion to developers that can be overwhelmed by pursuing a large amount of online posts or other documentation about a particular code entity.

2.1 Term Frequency-Inverse Document Frequency (TF-IDF)

To run our machine learning algorithm, we have first performed our analysis on the input of our approach, *i.e.*, collection of Stack Overflow posts for each code entity, to convert it to numbers. There are many techniques for text vectorization that can be implemented through NLP such as Bag of Words and TF-IDF (Das & Chakraborty (2018)). In the Bag of Words technique, after the pre-processing is done on the sentences, we have to tokenize all the sentences. Then, we have to make a histogram for all the sentences. In the histogram, the history of each word is found, *i.e.*, how many times each word is repeated in all the sentences. And a dictionary is created in which each word is considered as a key in this dictionary. And the value for each key in this dictionary is the number of repetitions of the word in the whole corpus. Some of the words in this dictionary must be filtered. Because the total number of words in a document is so large, we must reduce the number of words and in this filter, it is determined to select some of the most repetitive words. The next step is to create a vector for the mentioned dictionary. In this vector, a column is assigned for all filtered words, and each row of this vector is equivalent to a document in our corpus. If the word is in the document, we assign it one, otherwise, we assign zero. With the creation of this vector, we have converted the text, *i.e.*, collection of Stack Overflow posts for each code entity into a vector through the Bag of Words model. This output has been fed to our machine learning technique, *i.e.*, TextRank. The problem with this method is that when our input document is too large, the number of rows, which is the same as the number of documents, is very high. On the other hand, the number of words with a lot of repetition increases. So the vector that is made is very large. Therefore, the analysis of this vector becomes very time consuming and difficult. The main problems in Bag of Words can be summarized as follows:

- The importance of all words in this model is the same and there are no criteria for recognizing words. Therefore, if this model is fed into the machine learning algorithm, the algorithm cannot detect what the effect of each word in a document is (Khan, Salim, Farman, Khan, Jan, Ahmad, Ahmed & Paul (2018)).
- There is no semantic information in this model (Khan *et al.* (2018)).

To solve the problem mentioned in Bag of Words and improve it, researchers have suggested TF-IDF, considered as more effective than Bag of Words (Das & Chakraborty (2018)). By using the TF-IDF technique, the importance of specific words can be determined. This is because some words play a more important role, and by changing those words, the whole meaning of the sentence may change. TF stands for Term Frequency, for a particular word, in a document. IDF stands for Inverse Document Frequency, and in contrast of TF, it is calculated for the entire corpus. In this technique, like the Bag of Words technique, we have to get the histogram of all the words and then select the words that have the most repetitions. We then calculate the TF-IDF for each document. This technique is mostly used in text classification (Amin, Uddin, Hassan, Khan, Nasser, Alharbi & Alyami (2020)).

With TF-IDF we can get the importance of each word in a text , *i.e.*, in our case is multiple documents of Android posts for a code entity. In TF-IDF, it is better to use consecutive words in the document, such as Bi-gram, Tri-gram, etc., instead of just one word to find the words that are important (Piskorski & Jacquet (2020)). For example, Bi-gram is two consecutive words and Tri-gram is three consecutive words in a sentence. With the *n*-gram approach, we can have a sequence of *n* words which helps us achieve better results in terms of the concept of words that are used together (Subramanian (2019)). Therefore, we can capture the context of the words (Guerrouj *et al.* (2015)).

2.2 Word Embedding

In this section, the aim is to show how to represent a text, *i.e.*, in our case multiple Android posts for a code entity, as a vector, that is understandable to our machine learning technique. Converting text to a vector can be very simple, by encoding, for example, each word in a sentence with a number. Another way is to consider a vector for each word.

Word embedding is one of the most widely used techniques in NLP. One way to use Word embedding is to use a pre-trained word embedding that gives us the ability to find similarity between sentences. By combining this technique with machine learning algorithms, we can see the applications of Word embedding in translation machines (Zou, Socher, Cer & Manning (2013)) or question and answer machines (Shen (2015)). Word embedding is also applied to extract information like a summary for a document. There is a lot of research in this field. Kobayashi et al. (Kobayashi, Noguchi & Yatsuka (2015)) used word embedding as a feature extraction in text summarization. Another use of Word embedding is in text classification (Reimers & Gurevych (2019); Pham & Le (2018)).

One of the most obvious ways to interpret each word as a number is to assign a single number to each word in our text. One of the problems with this method is that there is no logic to assign a number to any word. And it is only based on the letters of the alphabet and their order, which is a number for each word. Instead of considering a number corresponding to each word, we can consider one column vector for each word. In this case, a single vector can be considered for each word, which is such that in all rows of this single-column vector, zero is applied, except for the row where there is a label of the same word, for which the label in that row, one is applied. This model of vectors is called a one-hot vector. One-hot vector is a simple way to encode words with a binary variable, and you can easily get a vector equivalent to each word, and you can also get to any word with a simple mapping of the one-hot vector. For example, let us suppose we have a list of words like: [Teddy Bear, book, soft] to display on a one-hot vector. For each word we have a vector of 1 x 3:

$$TeddyBear = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, book = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, soft = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{2.1}$$

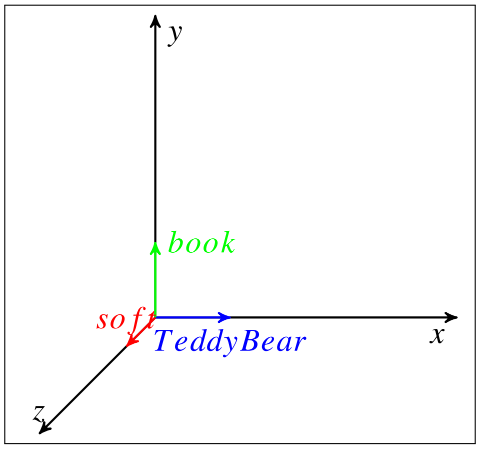


Figure 2.1 An example of one-hot vector representation.

One-hot vector has some limitations in NLP, which we mention here: one-hot vector is very large sparse vectors containing many zeros and leads to high dimensional space, and the representation of words are completely independently each other, so working with it is very time consuming and requires a lot of space for analysis. Another limitation of this technique is that the vector has not carried any meaning for the words. This technique is not valid for word similarity (Wang, Xu, Xu, Tian, Liu & Hao (2016)). Due to the limitations of One-Hot Vector, the Word embedding technique is used in NLP, because in this technique, the meaning of words are not lost (Pham & Le (2018)). Pham et al. (Pham & Le (2018)) proposed a Multichannel framework using Convolutional Neural Networks (MCNN) with a combination of word embedding (word2Vec and GloVe) and hot vectorization and showed that this model is suitable for solving the semantic problem. In their study, they showed that using just one-hot vectorization as a technique in their model is not as effective as word embedding. However, they claim that combining word embedding with one-hot techniques provides a good accuracy for their model.

In the Word embedding technique, if we want to explain, in general, how these vectors carry the meaning of words, it means that when words are semantically similar, their vectors are also closer to each other. Therefore, the word embedding of words with similar context are close together in the space. For example, in Figure 2.2, you can notice that "Teddy Bear" and "soft" vector representations are closer than "soft" and "book". In this technique, we do not have only zeros and ones, but a series of decimal numbers can be seen.

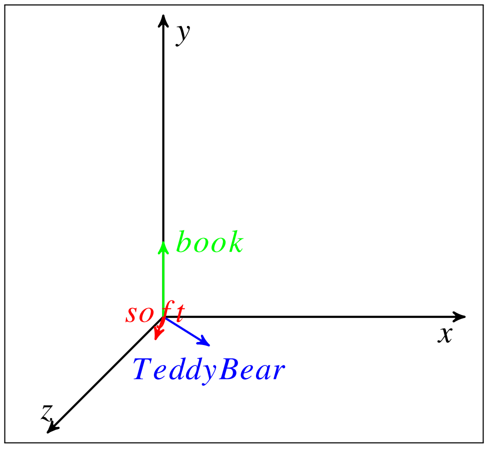


Figure 2.2 Word Embedding Representation

Additionally, the depth of the vectors can be increased. This depth represent how many features exists for each words. By increasing the depth of the vectors to n for words, you describe them with an *n* dimensional vector. And by increasing the dimension in vectors as well as the size of the document, the accuracy of similarity between word increases (Mikolov, Chen, Corrado & Dean (2013a)). As Mikolov et al. (Mikolov *et al.* (2013a)) state that this technique is used for a large corpus with billions of words.

The process of applying Word embedding consists of two steps. The first step is the creation of the *corpus*. This corpus must contain the words that we want to embed and we must manage them in the way that we want them to exist in our context. For our purpose, the corpus is all Stack Overflow posts in which a specific code entity has been discussed. The next step is the *Word embedding method* that builds Word embedding. There are many methods in Word embedding, such as machine learning models.Word embedding can be set by hyper parameters like the depth of Word embedding vectors.

Several methods are followed to build word embedding, some of which are listed below:

- Word2vec (Mikolov *et al.* (2013a)): Mikolov *et al.* (2013a) proposed this model for large datasets in two models, CBoW and Skipgram. The result of their study demonstrated that these models have better efficiency in terms of syntactic and semantic word similaritiy.
- Global Vectors (GloVe): Pennington, Socher & Manning (2014) proposed GloVe technique as text vectorization technique that is trained over Wikipedia articles. We will use the 200-dimensional GloVe embedding of 6 billion words computed on a Wikipedia.
- FastText.

In these models, each word always contains a specific vector and the vector does not change as the context changes.

Typically, we have a fixed vocabulary of words and we cannot generalize our model to the words that does not exist in our dataset. For example, if the word *onCreate* does not exist in our dataset, the learning algorithm cannot generate word embedding for this word. To address this problem, there are popular pre-trained word embedding developed by Google and Standford named as Word2Vec and GloVe (Pennington *et al.* (2014)) respectively. They are trained on a large text corpus with a a wide range of vocabularies such as Google News dataset. These pre-trained word embedding are publicly available and can be used in real-world NLP problems.

In the domain of software engineering, word embedding has been presented by Efstathiou, Chatzilenas & Diomidis (2018). In this research work, we have used a pre-trained Word2Vec embedding that has been trained by Efstathiou *et al.* (2018) with more than 15GB of data from Stack Overflow posts between August 2008 and December 2017. It supports a wide range of topics in the field of software engineering. This pre-trained model is based on skip-gram (Cf. Figure 2.3) neural embedding (Mikolov *et al.* (2013a)) which uses the current words to predict the surrounding window of context words. The resulting model has a dimension of 200 features and a vocabulary of 1,787,145 words. Because this model is trained over a domain-specific corpus, the model captures the meaning of technical terms in a software engineering context, which is suitable in the context of our problem.

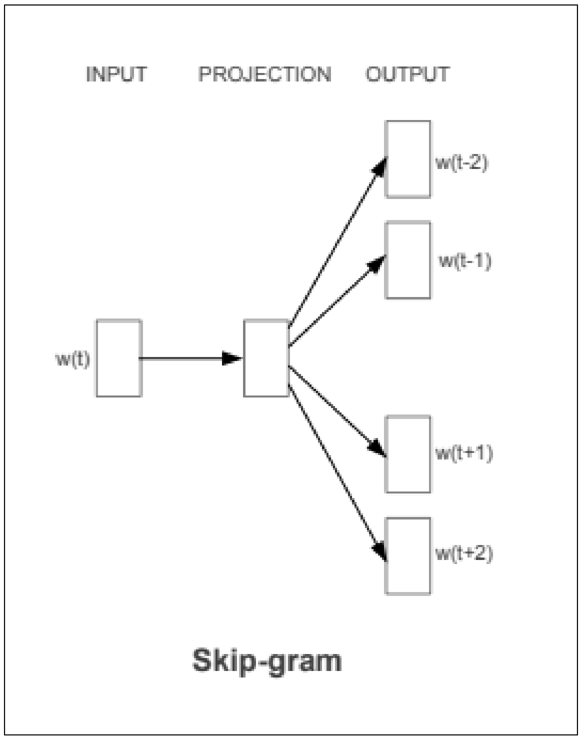


Figure 2.3 Skip-gram model
Taken from Mikolov *et al.* (2013a)

2.3 PageRank

One of the old ways of search engines was text-based ranking systems (Prajapati & Kumar (2016)). In this method, a search engine stored the index related to all web pages. When a query searched, the search engine searches the query in the entire indexes and finally obtains the number of times the query exists in each index. This method had many problems, including the fact that this search is only based on the number of times the query is repeated, and its content may not be related to the query. For example, consider a web page that consists of just one particular query that has been repeated a billion times.

To solve this problem, today's search engines based their work on being able to find the best outputs for the searched query. Therefore, the algorithm they considered is based on the content of the query in the web page. One of the most popular of these search engines is the PageRank (Brin & Page (1998)) algorithm provided by Google. The basis of this algorithm is based on the fact that the importance of each web page is based on how many other web pages are linked to it. Research has been done to summarize the code according to the PageRank in the code summary like Jiang's et al. (Jiang *et al.* (2017a)) work. In this study, we use the PageRank algorithm and cosine similarity to determine the importance of each sentence in the corpus.

PageRank is an iterative algorithm in which each node is a web page and the edge between each two nodes represents the links between web pages (Figure 2.4). For example, node a is pointing to node b, it means that the web page a contains a link pointing to b. Therefore, the graph in Figure 2.4 is an unweighted directed graph.

The formula (2.2) is provided by Brin & Page (1998). In this formula, v_i is the target node and $S(v_i)$ is the score for that node. d is a damping value, which is set between zero and one. The damping value is used to overcome the problem of disconnected web pages and the dangling web pages (nodes without outgoing edges) as well.

$$S(v_i) = (1 - d) + d * \sum_{j \in I_n(v_j)} \frac{S(v_j)}{|Out(v_j)|} \tag{2.2}$$

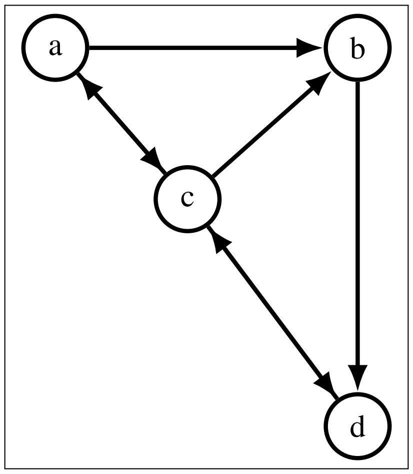


Figure 2.4 PageRank example

2.4 TextRank

One of the challenges that programmers face when understanding the right use of methods is that the amount of data and information that is available to them about a specific method is very large. This data includes emails, Bug-Reports, question and answer forums, source code, etc. This can be confusing and time consuming for programmers. Due to the existence of so much information in various sources, it becomes necessary to help programmers by providing them with summaries of the usage of each method that is part of their task, to identify the general concept and the purpose of each method. Hence, automatic code summarization is important. We have used an unsupervised learning, TextRank, to generate summaries for Android methods.

TextRank is an unsupervised machine learning algorithm for both keywords and sentences extraction presented by Mihalcea & Tarau (2004). It is based on a PageRank algorithm model by Brin & Page (1998). In the TextRank algorithm, the sentences are ranked by how similar they are to a given sentence. It is an algorithm that does not require human intervention, and it is suitable for automated summarization in an extractive way. In the extractive summarization, the whole corpus is examined, and then the sentences that express the meaning of the corpus are extracted. In this way, the concept of the corpus should not be changed. Moreover, the sentences are ranked according to their importance with respect to the whole corpus. When applying TextRank, the corpus is displayed as an interconnected graph. In this graph, each sentence is a vertex, and the edge between two vertices shows the similarity between sentences. After converting sentences to vectors by applying Word2Vec and TF-IDF techniques, cosine similarity is used to calculate the similarity between each two nodes. TextRank is a weighted undirected graph (Cf. Figure 2.5).

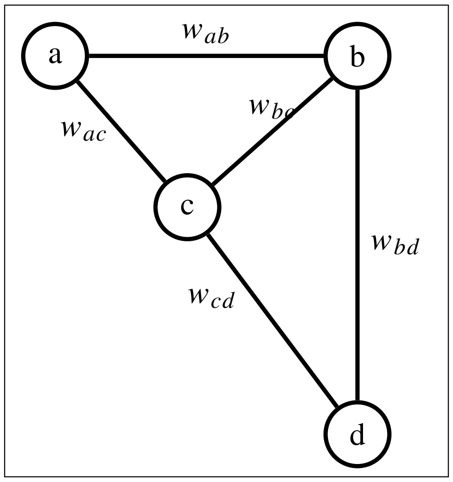


Figure 2.5 TextRank example

We have used the TextRank machine learning algorithm to extract relevant sentences from Stack Overflow posts of each examined code entity, *i.e.*, Android methods in our case.

Formula (2.3) is provided by Mihalcea & Tarau (2004), which shows TextRank score.

- $WS(v_i)$ shows the weighted score for node v_i;
- $I_n(v_i)$ is the set of vertices that points to v_i in the graph;
- v_j is one of the nodes in $I_n(v_i)$;
- $\sum_{v_k \in Out(v_j)} w_{jk}$ is used to reduce the influence of sentences that are similar with other sentences but are not good enough;
- d is the damping value. As I described in Section 2.3, this parameter is used to overcome the problem of disconnected web pages and the dangling web pages; Mihalcea & Tarau (2004)

have set the damping value (d) to 0.85 (Brin & Page (1998)). It means that there is 85% chance that the surfer will follow links on the actual pages (Brin & Page (1998)).

$$WS(v_i) = (1 - d) + d * \sum_{v_j \in I_n(v_i)} \frac{w_{ji}}{\sum_{v_k \in Out(v_j)} w_{jk}} * WS(v_j) \tag{2.3}$$

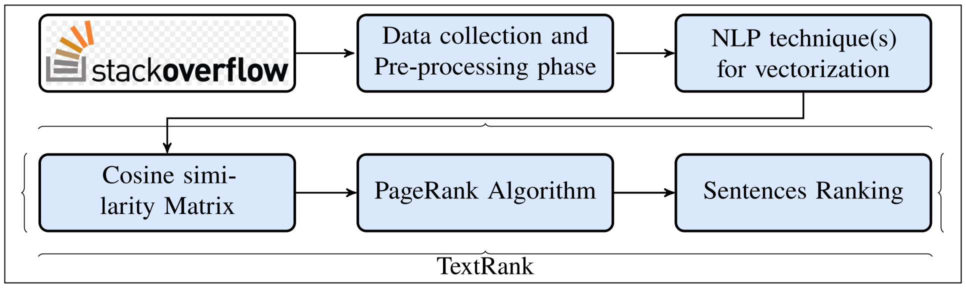


Figure 2.6 Main Steps of the TextRank Algorithm

We have tried empirically several parameters of the TextRank algorithm such as the damping value, the maximum number of iterations. However, by changing these parameters empirically, we did not see any improvements in the generated summaries. Hence, we have decided to set the maximum number of iterations to the default value which is 100, and set an amount of the damping value, in the TextRank, equal to the traditional amount provided by Brin & Page (1998). Also, Mihalcea & Tarau (2004) has set this value to 0.85 in Brin & Page (1998).

As can be seen in the figure 2.6, the main input for the TextRank algorithm is feature extraction, which means vectorization. In this study, we used two techniques, TF-IDF and word2Vec, to see the effect of this input on the performance of the TextRank algorithm.

CHAPTER 3

PROPOSED APPROACH

Text summarization refers to the process of receiving a corpus as input and outputting the useful and important information, without losing the overall meaning. With the advent of Natural Language processing in text summarization and the need to generate automatic summaries in different domains such as software engineering, approaches such as TextRank and deep learning gained more interest.

To obtain an automatic summary in our case, the TextRank model is implemented with two various vectorizations, the word embedding, and TF-IDF. Therefore, two types of summaries are generated for each method, and both of the results are then compared to official documentation. To compare and analyze these results, cosine similarity has been utilized as a measurement. The results of these analyses show that when word embedding is applied, which uses Stack Overflow as a pre-trained model to generate the word, our model has better results in comparison with TF-IDF. However, in some cases, the summary generated with the TF-IDF vector is more similar to the official documentation. Additionally, in this research, the relationship between the two summaries shows that these two summaries are very similar to each other.

We have performed an automatic process to compute the similarity between official documentation and the two types of created summaries using cosine similarity. Regarding vectorization, three different techniques are used before applying cosine similarity (TF-IDF, word embedding, which Stack Overflow is used as a pre-trained model for vector, and word embedding which uses Pre-trained GloVe Embedding models as a pre-trained model).

Overall, with all three different vectorizations, the results indicate that if we use the word embedding technique for vectorization in a summary generation, the result is closer to the official documentation.

3.1 Overview of the Approach

The main goal of our automatic summarization approach is to generate accurate and concise descriptions on the usage of code entities. The summaries are supposed to help developers not only understand how to use a code entity, but to also optimize their time when browsing online forums and navigating through informal documentation.

As mentioned earlier, the API in official documentation can have the problem of being incomplete and non-insightful (Robillard (2009), Uddin & Robillard (2015)). In addition, there is a lot of information as well as duplicated one on the usage of code entities in online resources. Therefore, developers have to spend a lot of time finding the reliable and suitable information to understand how to use a method. We conjecture that software developers solve programming problems in less time if they have access to accurate automatic code summaries. There are different information sources that contain code entities ranging from official documentation to source code to online forums. Software developers may be overwhelmed when dealing with all this information. We believe our approach may help them overcome the above challenges by providing them with short and concise descriptions of methods.

The purpose of our code summarization approach is to summarize the usage of Android methods. We used Stack Overflow as a source of information to collect the usage of Android methods (Guerrouj *et al.* (2015)).

The method we have followed to find the application of popular methods in Android is based on four steps, shown in Figure 3.1. In this section, each of these steps is fully explained.

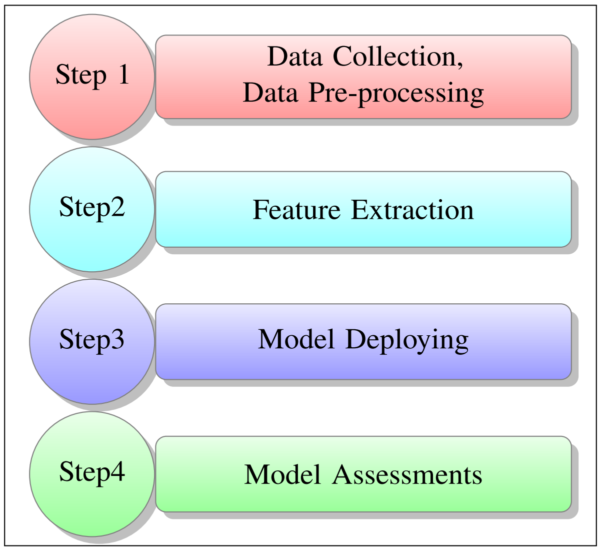


Figure 3.1 An overview of the main steps of our approach

3.1.1 Data Collection and Data Pre-processing

The initial step of our approach is to collect proper data. To collect the needed and proper data, we have crawled the Stack Overflow site to extract the data we need from this online Q&A forums. We have identified the set of popular methods to be examined and then, for each method, we have built a corpus tailored to our requirements. Once the data has been collected, we have cleaned it up to have a corpus suitable for our unsupervised technique. For this reason, we have followed a set of various pre-processing steps. Pre-processing plays a significant role in the final results since it filters noisy and unwanted data *e.g.*, "a" , "and" words that can impact the quality of the produced summaries.

The Data pre-processing includes the following two phases:

3.1.1.1 Data pre-processing

Data pre-processing involves the conversion of raw data extracted from Stack Overflow into useful and practical data for employment in machine learning algorithms. In our case, we eliminate answers and questions with a score lower than average (2.7). Then, for each selected post, we extract relevant and useful sentences, and consequently generate a corpus of sentences for each code entity. In the following section, we describe each step in more details:

Input sources for our code summary are all questions Android tags, collected from 2009 to April 2020 at StackOverflow's questions. We used Stack Exchange API to extract these posts. Finally, 1,266,269 posts were extracted with the Android tags. In addition, there is the proper information in answering each question that can be used as a complement to Android APIs. Therefore, we also extracted these answers: The total number of answers we collected for all the questions is 1,817,874. The total number of posts (questions and answers post) we collected is 3,084,143. These are posts that are unique.

As mentioned, the purpose of extracting these posts is to generate a summary for each method in Android. Therefore, from the posts we have extracted, we have to collect and categorize all the posts that are related to a method in one document so that we can prepare a summary. To achieve this goal, we need to analyze the body of the posts so that we can identify the methods in each post. We used Code Snippet to do this. Code Snippet was introduced in 2014 by Stack Overflow to identify the code inside a text quickly (Treude, Barzilay & Storey (2011b)). Before we can find the methods in the posts, we need to exclude the posts which are semantically problematic, incomplete, or do not support to detect the usage of the methods, from the archive of the collected posts. This is the case when the questions in the posts do not provide any relevant information regarding our goal. Accordingly, we have eliminated all these posts.

In the second stage, we have categorized the answers based on their score in the Stack Overflow, and we have excluded the posts with a score, below a threshold value. We have determined the threshold value by calculating the average rating of all scores for all posts. The reason is that some posts not only do not provide us with useful information and data, but may also deviate from the main topic and include information that is incorrect. Therefore, for the sake of reducing the impact of useless sentences, we have considered only Stack Overflow answers that have a score above the threshold value, which is 2.7 in our study.

After finding the related posts, we have to select the most important sentences from the body of these posts. We were inspired by the notion of proximity by previous works of Guerrouj *et al.* (2015) and Dagenais & Robillard (2012). Like other researchers, we believe that context that surrounds code entities is the most relevant. For this purpose, we have selected four sentences from each post with the following properties. The first sentence in each post that is considered as the main sentence in the posts. The second sentence that is selected is the sentence in which the name of the method is mentioned. And the third and fourth sentences are the sentence before and after this sentence, respectively. It is important to mention that we have empirically decided about $N = 4$ for the choice of our sentences.

To find a sentence that contains the name of a method, we have parsed the whole selected posts using Code Snippet. Code Snippet gives us the ability to identify where there is a code. This capability was introduced in 2014 by Stack Overflow so that users could easily access the code between sentences.

There are regularly two kinds of Code Snippets in each post: one type of code surrounded by natural language, which we call code entities, and the other, a code sample that exists in a post. Our goal is to find code entities that are surrounded by natural language. Because with code entities, we can access the names of methods or functions that are the purpose of this section, while code examples are used to show how to use or solve a problem using that code snippet.

At the end of this step, we extracted the list that contains the names of all the methods. As mentioned earlier, since our goal is to meet the needs of software developers to find applications for useful methods, we have selected the 15 methods that are most frequently repeated in these posts as well as five methods that are unpopular in terms of posts in Stack Overflow.

We created a separate corpus for each code entity. As mentioned before, we selected four sentences in each post and save them in the document in a format of CVS file. Finally, for each code entity, we have a corpus that contains all Stack Overflow posts that contain the code entity, with the condition that the score of that post is above the required average. So in this step, we have multiple corpuses which contain all useful sentences from Stack Overflow for each method

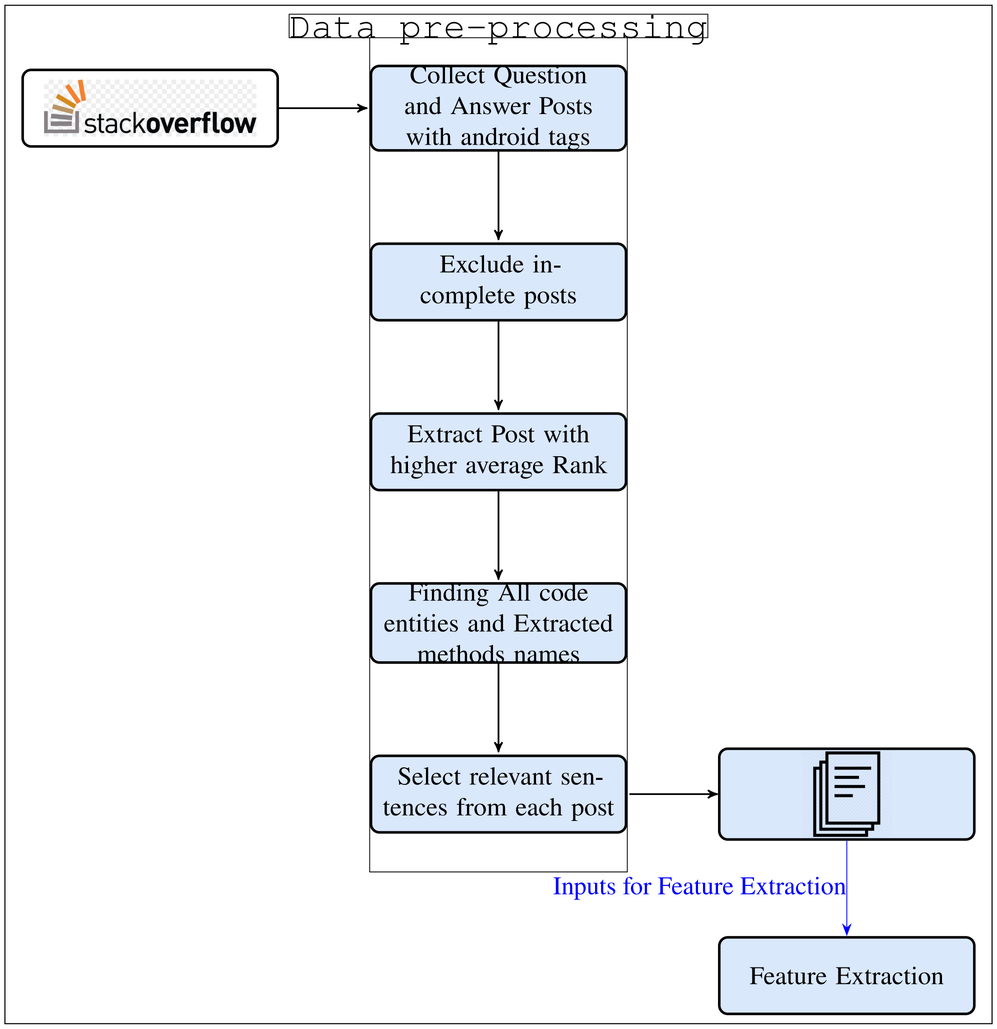

Data pre-processing
stackoverflow
Collect Question and Answer Posts with android tags
Exclude in-complete posts
Extract Post with higher average Rank
Finding All code entities and Extracted methods names
Select relevant sen-tences from each post
Inputs for Feature Extraction
Feature Extraction

Figure 3.2 Data collection and pre-processing task

found in these posts. Our dataset for the experiment is selected from four libraries of Android: *android.app*, *android.os*, *android.os.AsyncTask*, and *androidx.recyclerview*.

We decided to add a comparison with official documentation to see the extent to which our automatic summaries compare to official documentation. Hence, we extracted all the API references for Android from Android API[1]. Each API reference contains different packages, and individual packages include classes. Each class includes several public and protected methods.

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

In this extraction, there exists all public and protected methods, because some descriptions of public methods are linked to protected methods.

Finally, we generated a separate JSON file for each API. All packages, classes, as well as all related methods and descriptions, are attached to the JSON file. The number of classes in each package, as well as the number of methods in each class, is calculated and added to these JSON files.

3.1.1.2 Data cleaning

Our corpus is composed of unstructured data from Stack Overflow. And this corpus is generated from the previous step. In this phase, we need to clean the corpus to make it ready for feature extraction. Because there are lots of unknown symbols, lines, etc. in unstructured data, it forces us to clean it first before feeding our learning model with unstructured data.

After creating, for each code entity, a corpus from the previous step, in the form of a set of natural language sentences, we have applied natural language processing to clean our corpus and prepare it for the machine learning algorithm. In this section, we present the steps followed in the data cleaning phase.

- **Tokenization**: We have used the standard Natural Language Toolkit (NLTK) sentences Tokenizer, to split our text to sentences[2].

- **Stopwords Removal**: Some words in sentences are used only to construct and support sentences (such as "the", "a", "an", "in") but do not affect the meaning of the sentences. Hence, we have removed them from our corpus to focus on important words. For such a purpose, we have used the list of English Stopwords by the NLTK[2].

- **Text Normalization**: To normalize our corpus, we have used the *replace* functions in Python. Stemming and lemmatization are two ways to normalize text. In the following, we describe these two techniques:

[2] https://www.nltk.org/

- **Stemming:**

  The intention of stemming is to transform words into their original forms. The reason why stemming can be useful in pre-processing is that without this method, similar words will behave differently[3]. The morphological of ending English words is removed to transform them into the original forms of words. There are examples for Stemming, such as converting "sses" to "ss" at the end of words or "ies" to "i". In this process, there are several problems, such as using only a few patterns to achieve the base of words, and these patterns cause some limitations, which in some words show this lack so the result is not appropriate. For example, when using Stemming for "battling" it converts to "battl", which has no meaning. Because of this limitation, we don't use this normalization in our study . In general, it can be said that the problem with the stemming process in some words is that their meaning disappears after the stemming process is applied. It means, they cannot carry any meaning at all, and the generated word from this process is not present in a English dictionary.

- **Lemmatization:**

  The lemmatization process was used to overcome the stemming problem. This is the process by which another check is performed through the word dictionary to extract the base word. This process is slower but does not have the problem of stemming. After checking whether the word exists in the English dictionary, the lemmatization returns the words. If this word is not find in the dictionary, the original word itself as output. Hence, all the words have meaning and don't lose their meaning.

- **Spelling correction:**

  Without checking the spelling of words, we may have lost some useful and valuable information. Therefore, it is important to use Python auto Correct Library to correct spelling.

[3] https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html

3.1.2 Feature Extraction

The feature represents the characteristics of a text that may or may not be unique. For instance, the general features of the text, such as the text size, that do not directly affect the language of the text are not unique. However, some specific characteristics can be used to recognize that text distinctly and represent the text like Bag of Words (BoW) and TF-IDF. Because our data is unstructured, *i.e.*, it does not have a structure that is meaningful for machine learning, it is essential to convert it to numerical features. In this step, we demonstrate how to use TF-IDF and word embedding as feature extraction.

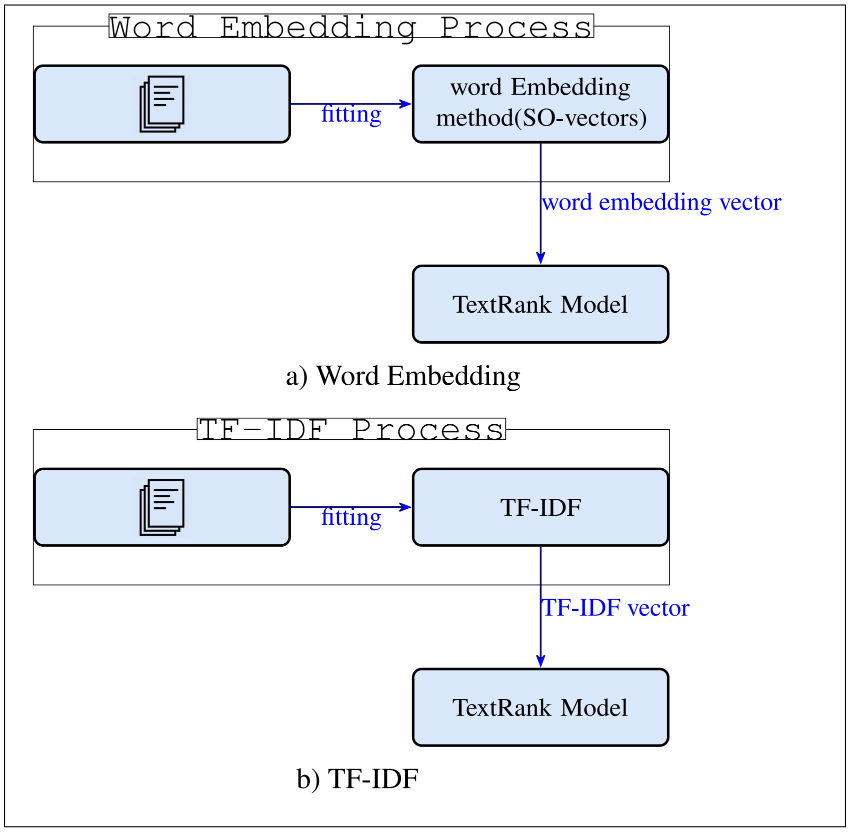


Figure 3.3 Feature Extraction

3.1.3 Model Deploying

Once the feature set is ready, we need a model which is suitable for extracting summaries for a generated corpus. These models could be for example machine learning-based algorithms, deep learning based, or reinforcement learning. In our case, we use TextRank unsupervised learning (Mihalcea & Tarau (2004)), which we explained in details here.

The output of code summarization can be in two ways: Extractive or Abstractive.

In the extraction method, which we have used in this research, the whole corpus is first examined, and then the sentences that express the meaning of the corpus are extracted. In this way, the concept of the corpus should not be changed. The general rule used in this extraction method is that sentences are ranked based on their importance in the corpus. After scoring the sentences, the sentences with the highest rank are selected. The ranking of sentences in this extraction method can be based on various approaches, some of which we will introduce and also point out their advantages and disadvantages.

- **Word frequency:**

  In this approach, sentences are ranked based on word frequency. In this approach, sentences with common and usual words are picked as a representative that shows the general meaning of the body. The disadvantage of this approach is that words that may not be representative of the corpus are more repetitive, so sentences that contain these words may rank high. Additionally, words that refer more to the meaning of the corpus may have less repetition and be ignored in this approach.

- **Sentence Similarity:**

  This approach uses a criterion that determines how similar the sentences selected as a summary are to other sentences in the corpus. Based on this, it can be concluded that each sentence selected as a summary represents several sentences with similar meanings that are not included in the summary. As a result, in a nutshell, each sentence in a summary represents a large part of the body. This approach is employed in the TextRank algorithm. And we have applied this algorithm in our code summarization technique.

The abstractive summarization approach is a method in which the sentences we see in a summary do not exist in the main corpus, they are however semantically similar. That is, there is a need for an algorithm that learns the meaning of the original text and then generates the summary. We did not use this method in this study. More advanced techniques such as neural network are an example of this approach. The input to this approach should be a set of words and letters that are placed sequentially (Nallapati, Zhou, Gulcehre, Xiang et al. (2016)). And also, the output is a sequence of words. This method, commonly called sequence-to-sequence, is used in translation models (Zou *et al.* (2013); Bahdanau, Cho & Bengio (2014)) from one language to another or from question to answers (Shen (2015)).

The model we have implemented in this study is based on an extractive approach from the original text using the TextRank algorithm. TextRank is a well-known algorithm in the approach of extracting text in the format of a summary from the primary corpus. This algorithm works based on the PageRank algorithm which is a search algorithm. The basis of the PageRank algorithm is that each page is assigned a rank based on the total number of other pages that refer to this page (Brin & Page (1998)). This means that more links to a page will lead to higher rankings. Similarly, the TextRank algorithm works by calculating the similarity matrix between whole sentences. These similarities are based on how similar each sentence is. In the next step, we need to create a graph of the similarity matrix created in the previous step. Finally, we rank each sentence according to the generated graph and then select the sentences that have the highest rank as summary sentences (Mihalcea & Tarau (2004)).

We are aware that questions and answers posted on Stack Overflow might discuss more than one Android method. For such a purpose, we have considered, for this preliminary study, *proximity* when selecting sentences that describe a particular code entity, *i.e.*, by limiting the corpus of each code entity to sentences that surrounds it. However, we are aware that the position of two

Android methods in a post might be not that much far, which may result in similarities in the produced summaries. We plan to tackle this challenge as part of our future work.

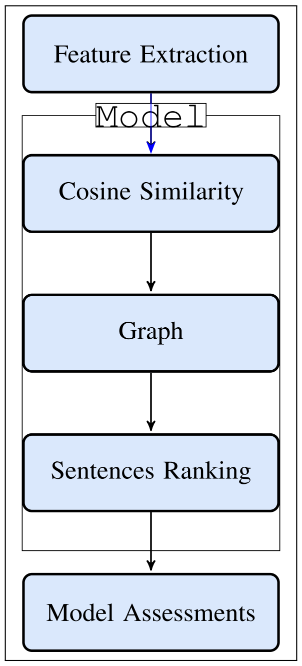


Figure 3.4 Model Deploying

3.1.4 Model Assessment

The last step is to create a benchmark to evaluate our output. We first explored manually the automatically generated summaries to make sure we do not have empty summaries, and that we have some summaries that make sense in the context of the examined methods. We have not performed any further evaluations for the quality of the summaries. However, we have compared our automatically generated summaries when using different configurations of vectorization, as well as with official documentation.

To the best of our knowledge, there is no source with which we could compare the results of our work. There is also no specific standard for this summary evaluation. Therefore, how to evaluate the generated summaries remains a challenge. We are planning to conduct a controlled experiment to evaluate the quality of the produced summaries and their usefulness for software developers in the context of their software engineering tasks as part of our future work.

Due to the COVID-19 pandemic, we were unable to assess the generated summaries through controlled experiments with software developers so the quality of generated summary remains unexplored empirically at this stage. For this reason, we have decided to perform a preliminary evaluation in an automatic way.

The first assessment we have conducted was to compare the generated summaries of both the TF-IDF and word2Vec techniques from the TextRank, thereby illustrating the impact of using different techniques in TextRank on the resulting summary. To achieve this goal, we had to make this comparison for all the methods we have summarized.

There exist many metrics to compare two texts like cosine similarity (Li & Han (2013)) and BLEU score. In this study, we used the cosine similarity metric. This metric is measured based on the similarity between sentences, thus showing how similar the two summaries produced are. Another assessment we have performed was to compare, using cosine similarity, each of the generated summaries with a description of the method in the official documentation.

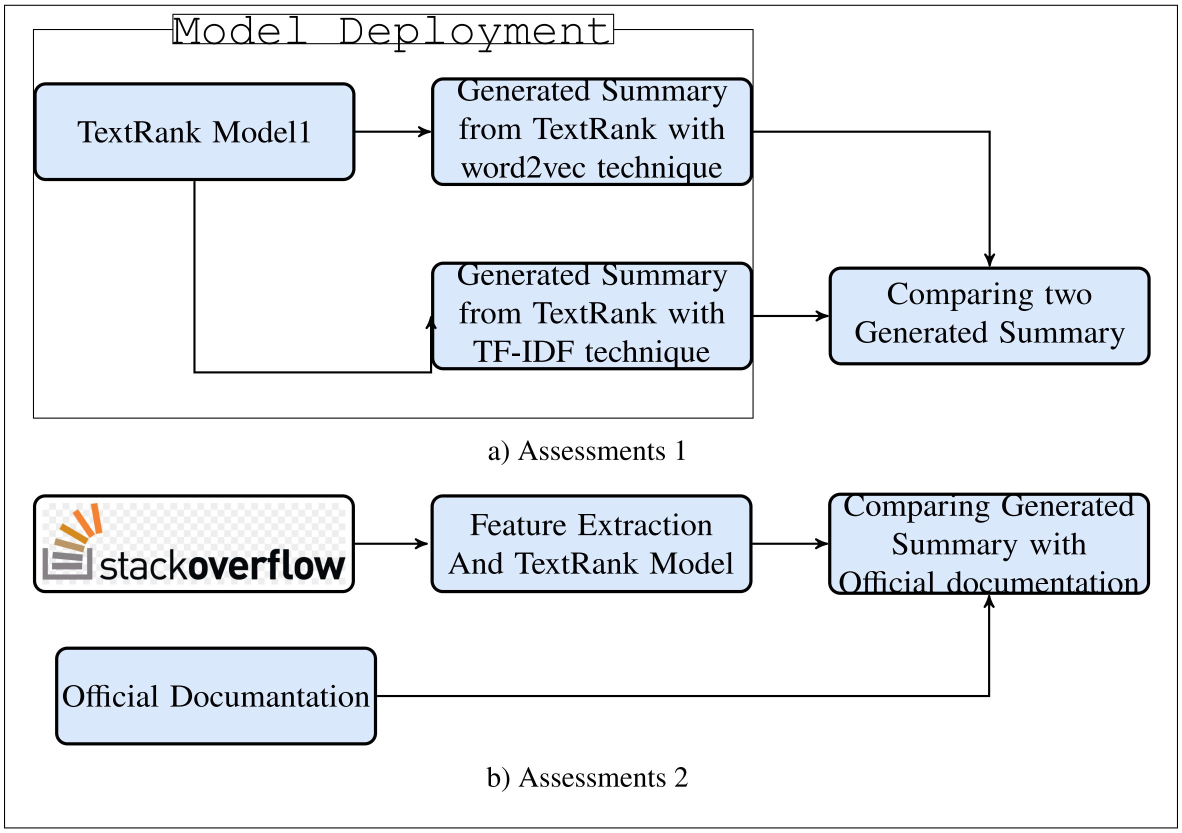

Model Deployment
TextRank Model1
Generated Summary from TextRank with word2vec technique
Generated Summary from TextRank with TF-IDF technique
Comparing two Generated Summary
a) Assessments 1
stackoverflow
Feature Extraction And TextRank Model
Comparing Generated Summary with Official documentation
Official Documantation
b) Assessments 2

Figure 3.5 Model Assessment

3.2 Limitations of our Approach

One of the limitations of our research is that our granularity level is limed to the method level. Our approach could be extended to other levels such as the class level. However, we know that in informal documentation, it is seldom that developers mention fully qualified names. Additionally, finding partially or fully qualified names in informal documentation is a hard problem and still an open question on which many researchers are working on right now (Moreno & Marcus (2012)). To solve this problem, researchers need techniques that can accurately extract fully and–or partially qualified names.

Another limitation is that our approach is dealing with popular Android methods only. The approach should be extend to deal with all kinds of methods. Our approach was run on unpopular methods too. But, the focus of the study is limited to popular Android methods.

We limited our approach to four sentences when extracting information from the bodies of Stack Overflow. The four sentences include the title of the post, the first sentence of the post and the sentence before and after the code entity. We were inspired by previous research that have highlighted the fact that the title and first sentence contain, most often, relevant information Rosen & Shihab (2016). We have limited the context that surrounds a code entity to one sentence before and after a code entity by mirroring the work by Guerrouj *et al.* (2015). In effect, considering more sentences may result in similarities in summaries for different code entities that have been discussed in the same post. One could consider other information in Stack Overflow posts such as comments in code examples, information from Stack Traces, etc. In the future, we plan to look at the effect of this kind of information on the quality of the produced summaries.

Due to the COVID-19 pandemic, we have not had chance to conduct controlled experiments with professional developers. It could have been beneficial for us if we could run controlled experiments in our research so as to evaluate the quality and usefulness of the produced summaries. We will intent to investigate in the future the extent to which our summaries could be beneficial to novice developers as well.

We selected Android methods that have many posts in Stack Overflow, *i.e.*, that have been mentioned several times in Stack Overflow. While this makes sense since we need information to be able to summarize methods, it is still a limitation and one could use other sources in the data collection phase such as bug reports, emails, etc. to complement with.

As we know, a better and more practical understanding of any method would be to provide an example of its usage. This is still a limitation in our case since we only provide a summary in the form of natural language descriptions.

CHAPTER 4

EMPIRICAL EVALUATION

In this section, we explain the design of the experiments performed, which includes how the methods in Stack Overflow are collected and categorized. Moreover, the steps of how to generate a useful and meaningful summary for software developers based on the documentation built for each method, are discussed in this section. To examine the effect of TextRank on the quality of the produced summaries, we examined two different vectorization methods that we leveraged in our approach. Then performed another investigation, which consists of determining the extent to which our automatically generated summaries are similar or not to the official documentation of the examined methods.

Overall, the objective of this chapter is to design a novel machine learning approach to produce summaries for the usage of Android methods based on informal documentation, *i.e.*, StackOverflow. The reason is that past and recent research have shown that official documentation lacks insights, completeness and conciseness (Robillard (2009) and Uddin & Robillard (2015)). Our focus is on vectorization techniques and unsupervised machine learning techniques to find high-ranked sentences and extract them as an output. In other words, we would like to leverage unsupervised machine learning to extract relevant sentence on the usage of methods by scoring sentences extracted from informal documentation.

Programmers always require accessing the usage of methods when they perform software maintenance and evolution tasks that involve these methods. As we know there exists official documentation for Android. However, researchers have shown in previous works that the official documentation lacks insights, completeness, and context Robillard (2009). Hence, we attempt to summarize code entities using informal documentation that may provide developers with more insights on code entities.

Our TextRank approach with Stack Overflow pre-trained dataset vectorization can solve this problem by automatically providing the summary of the usage of an Android method using informal documentation. Once a vector is assigned to any sentence, the TextRank approach is applied to find the highly-ranked sentences based on an importance of sentence without losing the idea of whole document, which forms an automatic summary for an Android method in question. To produce a summary for an Android method, we have applied two separate vectorizations. The first vectorization is TF-IDF, which creates a sequence of vectors for each sentence in which the importance of specific words can be specified. The obtained set of vectors is then used for the TextRank approach. The second vectorization is named word embedding. Its goal is to represent the semantics of the word. In practice, our purpose is to produce an accurate summary for the usage of methods based on informal documentation.

4.1 Definition and Planning of the Study

The definition of the experiment is explained based on Basili framework (Basili, Caldiera & Rombach (1994)).

The *goal* of this study is to evaluate the automated summaries generated for Android methods discussed in Stack Overflow, as well as to evaluate whether the generated summaries could be used as a resource to find insights about the usage of methods. The *purpose* of this research is to help developers spend less time when updating their code to meet or update the software requirements, as well as use the source code quickly and accurately.

The *quality focus* of our study is the cosine similarity metric. Cosine similarity compares the generated summary with the official documentation to find the extent to which they are similar to each other and to have an idea about whether our summaries make sense in the context of a specific method. Cosine similarity is a proper metric to use in our study since it is not influenced by the size of sentences chosen for comparison. Besides, the output range of this metric is between zero and one, so we can easily compare the generated summary in front of official documentation. So this metric gives us an overall measure of the similarity between the automatically generated summaries and those by official documentation. To evaluate the quality of the generated summaries, we planned a controlled experiments that we did not start given the

COVID-19 pandemic. We intend however to evaluate the accuracy, quality and impact of the produced summaries as part of our future work.

The research *perspective* is of researchers and Android developers interested in having a better understanding of the methods that are parts of their tasks as well as their usage.

The *context* consists of a dataset that is gathered and categorized from Stack Overflow posts, which is a Q&A forum. All Android Q&A posts tagged between 2009 and April 2020 have been extracted, and we collected a total of 3,084,143 unique posts. A parser was implemented by a Ph.D. student in our laboratory. In this parser, all Stack Overflow post bodies are analyzed and cleaned, then most repeated Android methods were selected. We have reported the results of this parser in Table 4.1. Table 4.1 contains the exact names of the Android methods along with the classes and packages since our method requires the exact names of these methods.

There have been 15 most popular Android methods involved in our study. We have considered as a measure of popularity, the number of posts in which a code entity has appeared. We have selected only the top 15 code entities for summarization because after that, the number of posts have decreased dramatically, which was not reasonable as an input for our machine learning algorithm. Regarding the other five methods, we have selected five unpopular Android methods as a sample.

Qualified names of Android methods are extracted based on data from Android Official documentation[1]. Popularity is based on a high occurrence of code entities in Stack Overflow as well a score higher than the average score set in our study. Checking the official documentation, revealed that the 20 code entities that we have collected, belong to the following four libraries: *android.app*, *android.os*, *android.os.AsyncTask*, and *androidx.recyclerview*.

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

[2] The Qualified names of Android methods are extracted based on data from: https://developer.android.com

Table 4.1 Most popular Android methods based on the number of Stack Overflow posts

Method with Fully Qualified Name[2]	**Add in API level**	**Deprecated in API level**	**Questions**	**Answers**	**Total**
android.app.Activity.**onCreate()**	1 (October 2008)	x	7,933	21,103	29,036
android.os.AsyncTask.**onPostExecute()**	3 (May 2009)	x	1,413	5,452	6,865
android.app.Fragment.**onCreateView()**	11 (February 2011)	28 (December 2018)	1,769	4,291	6,060
android.app.Activity.**onActivityResult()**	1 (October 2008)	x	2,196	3,853	6,049
android.os.AsyncTask.**doInBackground()**	3 (May 2009)	x	1,135	4,569	5,704
android.app.Activity.**onPause()**	1 (October 2008)	x	1,556	4,031	5,587
android.app.Activity.**findViewById()**	1 (October 2008)	x	804	4,182	4,986
android.app.Activity.**onDestroy()**	1 (October 2008)	x	1,588	3,333	4,921
android.app.Activity.**finish()**	1 (October 2008)	x	1,105	3,224	4,329
android.app.Activity.**setContentView()**	1 (October 2008)	x	627	3,562	4,189
android.app.Activity.**onStop()**	1 (October 2008)	x	949	2,188	3,137
android.app.Activity.**startActivityForResult()**	1 (October 2008), 16 (June 2012)	x	779	2,325	3,104
androidx.recyclerview.**onBindViewHolder()**			845	1,973	2,818
android.app.Activity.**startActivity()**	1 (October 2008), 16 (June 2012)	x	558	2,217	2,775
android.app.Activity.**onBackPressed()**	5 (November 2009)	x	652	1,906	2,558
Total			23,909	68,209	92,118

4.2 Research Questions

To address our research problem, we have tackled the following research questions that we have attempted to answer during our empirical evaluation:

- **RQ1: Is our machine-learning based approach able to produce summaries on the usage of code entities discussed in Stack Overflow?**

  Lots of research on summarization has exploited source code. Recently, some research works (Rastkar *et al.* (2014); Korayem (2019)) have tried to leverage bug reports in the context of summarization. The first work has summarized bug reports (Rastkar *et al.* (2014)), while the second (Korayem (2019)) has leveraged bug reports to summarize code elements using supervised learning. We were inspired by these works in dealing with informal documentation. However, unlike these works, we leverage Stack Overflow as a source of information since it has been proven to contain rich information about code entities (Guerrouj *et al.* (2015)). Also, we apply unsupervised learning and focus on the usage of code elements. Our goal is to show if our suggested approach can produce summaries on the usage of code entities that can be useful for developers. In addition, finding techniques that can extract sentences (instead of keywords) without losing their meaning leads to better summarization of code elements. In this scenario, the objective is not just to discuss the automatically generated summaries, but to also explore and compare the generated summaries, while using different configurations ad vectorization techniques. Based on this comparison, we can show the effect of different configurations in this research. We have examined these configurations by employing similarity metrics such as cosine similarity.

To answer RQ1, we have accomplished an investigation on the types of summaries generated using the TextRank approach. We have applied two types of vectorization techniques prior to applying TextRank. We have used cosine similarity, which is a useful metric in data mining to compare two texts with each other. It provides a measure on how two automatically generated summaries are similar to each other. Then, the differences between the generated summaries are studied.

- **RQ 2: How do our automatically produced summaries compare to official documentation?**

  To have some insights regarding whether our summaries have something in common with official documentation or not, we have extracted all Android methods examined in our empirical evaluation from official documentation[3] and then used them as a baseline to compare with our generated summaries. We are aware that while our focus is on the usage of Android methods, the descriptions provided by official documentation lacks insights, completeness, and context (Robillard (2009), Uddin & Robillard (2015)). However, we wanted to investigate if at least our automatic approach provided summaries that make sense and are related in terms of their context to official documentation. More evaluations of the quality of the summaries and their usefulness will be part of our future work.

  To answer our RQ2, We have used 20 Android methods from four Android libraries. For each Android method, when TF-IDF is used as the vector technique, one summary is generated by TextRank, and the other summary is generated using word2Vec technique which used Stack Overflow post for training their model as the vector technique in the TextRank model. We then compare the produced summaries by each technique with the corresponding summaries provided by the official documentation. This comparison is measured in an automated evaluation. Due to COVID-19, we were unable to perform a manual evaluation.

[3] https://developer.android.com/reference

We intend however to do that in a follow-up study as part of our future work. Automated evaluation involves comparing automatically generated summaries with the descriptions that are available for each method in the reference, *i.e.*, Android official documentation. One of the advantages of this method is that it is automatic and objective.

- **RQ3: Do the obtained results vary when using different vectorization techniques?**

  To evaluate which vectorization configuration is appropriate, when summarizing our code entities using our unsupervised learning, *i.e.*, TextRank, we investigated three different configurations: TF-IDF, Word2Vec with Stack Overflow pre-training and as well as GloVe.

4.3 Design of the Empirical Study

The empirical evaluation that we have designed to evaluate the automatic code summaries involves Android methods that we have collected from Stack Overflow. We have chosen Stack Overflow to reach the use of methods in Android because it is not only a resource that has been considered in previous researches as good informal documentation for code summarization (Saddler *et al.* (2020)), but it is also a popular question and answer online forum with wide range of topics in programming that developers frequently refer to search answers to their questions. Their questions are usually related to the concept of how, when, where, and why they use a particular code entity in their source code. Furthermore, in this forum, developers share their knowledge with other developers. Professional developers share their experiments to help others improve their jobs and get good results. Another characteristic that motivated us to choose this online forum in our empirical study was that Stack Overflow not only supports programming topics, but each topic in each post is focused on a specific problem, which helps us to be confident in the content collected. The statistics reported in Stack Overflow show that it has more than 100 million monthly visitors, and a new question loaded in this forum every 14 seconds[4].

In this research, our goal is to extract the correct usage of Android methods from Stack Overflow. Android is one of the most popular and dominating mobile platforms. Other characteristics

[4] https://stackoverflow.com/company

of Android is an open source platform whose operating system is Linux. There are many apps on the market whose platform is Android [5]. Another reason for selecting Android is the fact that it represents one of the top eight most discussed topics on Stack Overflow [6]. In this context, as I mentioned in section 4.1, we have selected four different libraries from android: *android.app*, *android.os*, *android.os.AsyncTask*, and *androidx.recyclerview*. Table 4.1 provides some descriptive statistics about the Android methods selected from these libraries for the sake of this study.

4.4 Analysis Method

One of the most popular metrics to find the similarity between two entities is Cosine Similarity (Li & Han (2013)). Basically, to find the similarities between two vectors cosine similarity is used. If the angle between two vectors is small, they are more similar to each other. So the angle between two vectors is used to find the similarity of vector representations. This metric is not biased in terms of the size of vectorization like the Euclidean distance.

The Cosine similarity formula (Li & Han (2013)) is:

$$similarity(v, w) = \cos(\beta) = \frac{\vec{v}.\vec{w}}{\|\vec{v}\|\|\vec{w}\|} = \frac{\sum_{i=1}^{n}(v_i.w_i)}{\sqrt{\sum_{i=1}^{n}(v_i)^2}\sqrt{\sum_{i=1}^{n}(w_i)^2}} \tag{4.1}$$

with the above formula, the similarity between any pair of vectors can be computed. This cosine is proportional to the similarity between the vectors. Moreover, this metric gives value between 0 and 1.

Cosine Similarity has applications in data mining processes, data retrieval, and text matching (Li & Han (2013)). For example, when the corresponding vector for each sentence is created in

[5] https://www.tutorialspoint.com/android/android_overview.htm

[6] https://en.wikipedia.org/wiki/Stack_Overflow

a large corpus, cosine similarities can be applied to find the similarities and differences between every two pairs of vectors.

In Cosine similarity if two vectors have the same direction, they are almost similar (Cf. Figure 4.1). As presented in Figure 4.1, if two vectors have the same direction, their similarity is to 1. When the angle between two vectors are 90°, their similarity is 0. As mentioned, the cosine similarity occurs between the two pairs of vectors, and this is a symmetric algorithm, determining that the result of calculating the vector "v" and "w" is the same. We can calculate the similarity between two pairs only once. So the number of calculations is divided by 2. Because the similarity between the vector "v" and "w" is the same as the vector "w" and "v".

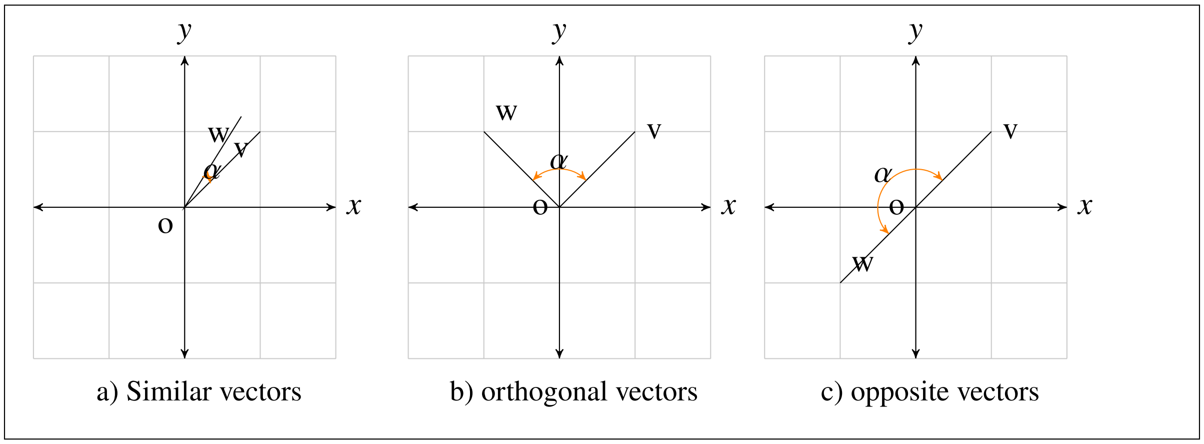


Figure 4.1 The cosine similarity between two vectors

CHAPTER 5

RESULTS OF THE STUDY

This section presents the results of the novel approach suggested to generate automatic summaries for Android methods from Stack Overflow, along with the data analysis performed to compare the automatically produced summaries for each method with their summaries in official Android documentation. This evaluation involves a sample of Android methods: popular methods and unpopular ones. As stated earlier, two different techniques (word embedding in our case pre-trained word2Vec model in stack Overflow post and TF-IDF vectorization) have been applied to generate a summary of the code entity using the TextRank approach. We have applied Cosine Similarity on each examined method and its counterpart method in formal documentation. Additionally, this evaluation has been done based on three vectorizations techniques as explained in the following sections.

5.1 Results of RQ1 - Is our machine-learning based approach able to produce summaries on the usage of code entities discussed in Stack Overflow?

In this section, we present the results of our first RQ, *i.e.*, whether our novel approach is able to produce summaries for code entities discussed in informal documentation. Our approach is based on the TextRank model that we implemented by using two separate vectorization techniques in natural language processing (NLP). As mentioned in previous chapters, to apply the TextRank algorithm to summarize the qualified corpus, we must first represent the text with its corresponding vector. Therefore, we have utilized two different techniques for this representation. These techniques are TF-IDF and Pre-trained Word2Vec in Stack Overflow. We need suitable data to feed to our machine learning algorithm. This data should be in a numerical format. Hence, we need to convert the dataset which is in a text format, to numerical format. For such a reason, we have applied two techniques and compared their results. Following this step, using the TextRank algorithm, we rank the sentences bases on an association graph vertices with the text for sentences and consider the three sentences with the highest score as the summary of the method.

In the following, we provide examples of summaries automatically produced by our approach using the above-mentioned configurations for vectorization, for both our sample of popular and unpopular methods.

Table 5.1 shows the summaries produced for a set of studied Android methods. For space reasons, we only show examples for two Android methods. The produced summaries for the remaining methods can be found in Appendix I, Table-A I-1. In these two tables (5.1, A- I-1) As you can notice, the summaries produced by our approach are created from sentences extracted (as they are, *i.e.*, including mistakes, typos, etc.) from Stack Overflow.

Table 5.1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques

Method Name	**Vectorization**	**Usage summary**[1]
onCreate	Word2Vec	["As you can see, 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 inside onActivityResult() 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 OnClickListener for your button and implement the onClick() method so that it starts your new Activity."]
	TF-IDF	["In your MainActivity, call the Injector in the onCreate() method." "The problem is that you are 'planting' trees in the Activities onCreate method." "As you can see, 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()."]

Continued on next page

Table 5.1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onPostExecute	Word2Vec	["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."]
	TF-IDF	["When you use a Thread, you have to update the result on the main thread using the runOnUiThread() method, while an AsyncTask has the onPostExecute() method which automatically executes on the main thread after doInBackground() returns." "Note that the AsyncTask run the doInBackground() method in a different thread than the UI thread and onProgressUpdate() and onPostExecute() on the UI thread because they should update UI." "When you use AsyncTask you can use the onPostExecute method to return the value to the main thread."]

Another purpose of this section is to find out how much the automated summaries created by the TextRank model, with two different configuration for vectorization, are semantically similar and close to each other. To determine whether the generated summaries for the methods discussed in Stack Overflow are similar to each other, we have applied cosine similarity with three different configurations with regards to vectorization and compared the obtained results. This metric is

[1] The summaries below are generated based on data from the following sources: https://stackoverflow.com, https://developer.android.com

applied to calculate the differences between two generated summaries. As mentioned in the article McBurney *et al.* (2016) cosine similarity is a useful metric in mining text.

For this purpose, for every examined method, we have measured the cosine similarity using three techniques of vectorization. Summaries in this cases have been generated summary using the SO word embedding and TF-IDF (Cf. Table 5.2).

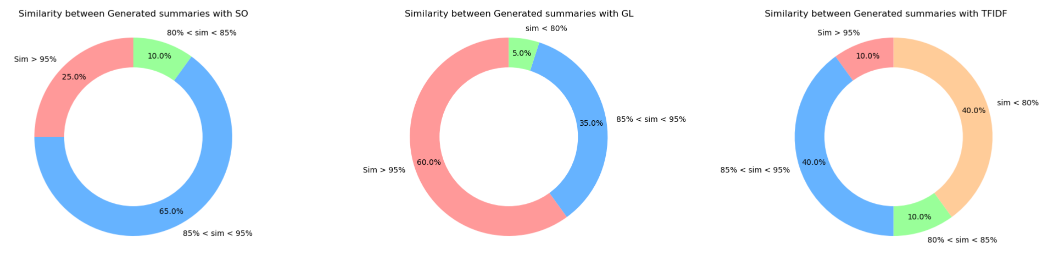
- **Word2Vec (SO):** Cosine similarity with SO word embedding technique;
- **Pre-trained GloVe Embedding:** Cosine similarity with pre-trained GloVe Embedding technique;
- **TF-IDF:** Cosine similarity with TF-IDF vectorization technique.

As it can be noticed from Table 5.2, if we calculate the average cosine similarity measurement among all the 20 methods tested, in both the SO and pre-trained GloVe Embedding test models, this average is higher than 90%. However, when we look at the average test we conducted using the TF-IDF technique when measuring cosine similarity, we notice that it is 12% lower than the test performed through GloVe and 10% lower than the average test.

Table 5.2 The similarity between the automatically generated summaries, with three different tests

Method		**Word2Vec (SO)**	**GloVe**	**TF-IDF**
Popular Methods	onCreate	0.930836572	0.959548616	0.538467551
	onCreateView	0.880537633	0.781056605	0.811248577
	onPostExecute	0.9194409	0.923173489	0.900270106
	onActivityResult	0.958559245	0.972170976	0.865395488
	doInBackground	0.933855155	0.957139724	0.884310241
	onPause	0.82094175	0.920730894	0.568897949
	findViewById	0.904855574	0.921489336	0.797378169
	onDestroy	0.957354487	0.981751892	0.942125115
	finish	0.838817046	0.875283293	0.56079035
	setContentView	0.962592707	0.973058436	0.894478601
	onStop	0.903107079	0.96978717	0.797045967
	startActivityForResult	0.917139257	0.964292087	0.768199778
	onBindViewHolder	0.937935807	0.954124698	0.721685393
	startActivity	0.951441344	0.977525834	0.920573654
	onBackPressed	0.917280543	0.877196461	0.862639168
Unpopular Methods	onItemClick	0.94845298	0.960231372	0.837631771
	onNewIntent	0.944872395	0.96979738	0.860356833
	onMeasure	1	1	1
	onViewCreated	0.892783716	0.901188169	0.740945715
	onRestoreInstance	0.912871451	0.946635153	0.97530483
Average		0.921683782	0.939309079	0.812387263

As depicted in Figure 5.1, 95.0% and 90% of the generated summaries from two different methods, are similar more than 85% when the pre-trained GloVe Embedding and SO are used as a vectorization model in cosine similarity. The GloVe technique resulted in an accuracy of over 95% for 60% of tests compared to 25% in SO technique. Whereas when TF-IDF is applied in cosine similarity to find the similarity between two models of generated summaries, only 50% of them are similar more then 85%. The outputs from the TF-IDF technique is far from the other two techniques and resulted in an accuracy of over 95% just for 10% of the tests.



a) Word2Vec (SO) b) Pre-trained GloVe Embedding c) TF-IDF

Figure 5.1 Cosine Similarity between the automatically generated summaries, when using different vectorization in Cosine similarity

Additionally, if we use the TextRank model to summarize unpopular methods discussed in the posts in Stack Overflow, we find that the similarities between the two methods of summarizing are much closer and more alike. This is because of the number of posts for unpopular methods is limited. Accordingly, the most important sentences in these two examinations for unique methods are more likely to be identical and unique. So for instance, for the "onMeasure" method this similarity is 1. This means that our produced corpus has limited sentences for this method, which leads to the extraction of exactly the same sentences.

Overall, we can conclude that the results of this approach are very similar with different vectorization techniques. In the following, we provide a benchmark to compare the two produced summaries with word embedding and TF-IDF vectorization in our approach..

Figure 5.2 show these three techniques trend in finding the similarity between the generate summaries. As it can be noticed, GloVe and SO techniques have the same trend in both popular and unpopular methods. However, in TF-IDF vectorization, we can observe that, in the popular methods which have a lot of words in it, it couldn't yield good results, which means that more likely TF-IDF is not convenient in this context.

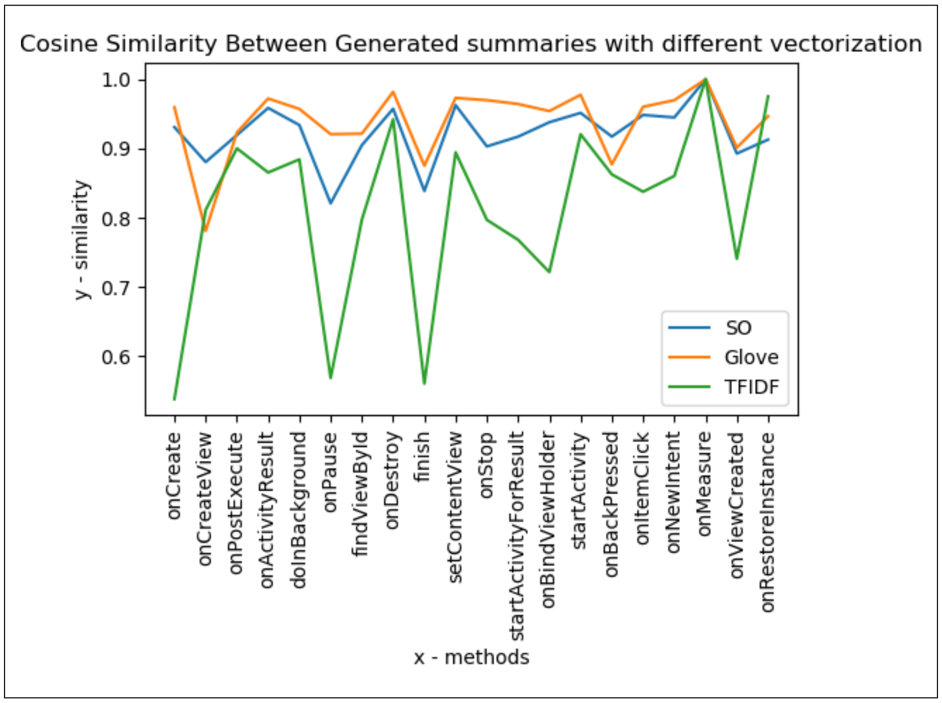


Figure 5.2 Cosine Similarity between the automatically generated summaries, with different Vectorization

5.2 Results of RQ2 - How do our automatically produced summaries compare to official documentation?

The main objective of this analysis is to examine the advantage of exploiting Stack Overflow as an informal documentation when providing summaries of the Android methods. For such a purpose, We have first chosen the most appropriate technique that represents the vectors associated with a particular corpus, which is compatible with the TextRank approach. Then, we have applied two techniques to represent the vectors of each document: TF-IDF and word embedding. These vectors are then employed as input to the TextRank approach to generate the summary.

To determine whether the generated summaries for the methods discussed in Stack Overflow are reasonable, we have applied cosine similarity to compare the two vector methods in the TextRank model. Additionally, this cosine similarity analysis is conducted based on different vectorization (TF-IDF, Word2Vec which is trained with Stack Overflow post, and GloVe word Embedding).

In this research, we have collected all the public and protected methods in Android for all libraries in a document that has been extracted from the official Android site. On this site, in addition to the descriptions provided for each method, a summary is provided that is an overview of that method. Accordingly, we call this document the official documentation for Android. We have considered the official documentation for Android methods as a baseline for comparison with summaries generated by two different vector methods (Word2Vec and TF-IDF).

In the first comparison, we have considered the following configuration of the baseline: all the sentences in the methods extracted from the official documentation as our baseline. And in the second configuration, we have considered just the summary available in the official documentation for each method, which is usually indicating the use of that method in one or at most two sentences. Hence, automatically produced summaries by our approach for Android methods extracted from Stack Overflow are compared with their corresponding summaries provided by official documentation Android API[2].

As we know there exists official documentation for Android. However, researchers have shown in previous works that the official documentation lacks insights, completeness, and context Robillard (2009). Hence, we attempt to summarize code entities using informal documentation that may provide developers with more insights on code entities.

We are aware that while our purpose is to summarize the usage of methods discussed in Stack Overflow, summaries provided by official documentation lacks insights, completeness, and context (Robillard (2009)) and hence one can argue that we are comparison summaries of two different natures. Also, we are aware that official documentation lacks context, completeness (Robillard (2009), Uddin & Robillard (2015)). Our plan was to evaluate this summaries via controlled experiments to assess their accuracy and usefulness. However, due to COVID-19 exceptional circumstances, we could not make this evaluation and we opted for this basic

[2] https://developer.android.com/reference

comparison in this preliminary investigation to at least have some clues on whether our summaries have something in common with official documentation or not. It is also worth-mentioning that we do not put any restrictions on the size of the produced summaries. The reason is that the size of the produced summary is related to how difficult or easy is a method. Some methods may be easier than others and their summaries can be, therefore, shorter than others and vice-versa.

We investigated the results of the summaries produced in this analysis under two separate tests with 20 samples of summarized methods. The aim is to determine to what extend the produced summaries make sense in relation to the official documentation. We are aware that our summaries are context-aware since based on discussions in Stack Overflow and usage-oriented and hence may contain more insights and information that does not exist in official documentation. In a future investigation, we are going to compare them in terms of their quality. For instance, we just want to validate if our automatically generated summaries are reasonable. Further investigations are planned as part of our future work.

For this purpose, we have performed several experiments with various configurations to determine how utilizing different vector techniques can help produce a more useful summary. At a first stage, we have considered only useful sentences that are parts of the descriptions of methods in official documentation and compared them to the summaries we automatically generated using unsupervised learning. In these two evaluations, out of 20 methods, we have selected 15 of the most repetitive methods in Stack Overflow with a number of questions and answers of not less than 2,000 (Table 4.1), and the other 5 methods are subdivided into two levels, two of which are among the last two unpopular methods in the first thirty methods and the last three are related to the last three unpopular methods in the first fifty methods.

Table 5.3 reports the results of the comparison between the automatically generated summaries and summary descriptions (short description) in official documentation for Android methods, considering two techniques: Word embedding vectorization and TF-IDF vectorization applied in TextRank model. For example, for the *onCreate* method, the similarity between our automatically generated summary with W2V vectorization and the short description of the code entity extracted

Table 5.3 Results of the comparison using different vectorization techniques, with counterpart methods in official documentation (important sentences only)

Method		**WordEmbedding_SO**	**TF-IDF_SO**	**Best Result**
Popular Methods	onCreate	0.565661851	0.534035781	Word Embedding
	onCreateView	0.596149943	0.622425014	TF-IDF
	onPostExecute	0.650774704	0.756475185	TF-IDF
	onActivityResult	0.780983937	0.805050723	TF-IDF
	doInBackground	0.822704552	0.804805112	Word Embedding
	onPause	0.839828775	0.610247866	Word Embedding
	findViewById	0.660941726	0.653051119	Word Embedding
	onDestroy	0.738225762	0.677758291	Word Embedding
	finish	0.751145686	0.748650506	Word Embedding
	setContentView	0.760736789	0.750972587	Word Embedding
	onStop	0.700300935	0.686982384	Word Embedding
	startActivityForResult	0.823765841	0.862835667	TF-IDF
	onBindViewHolder	0.719573233	0.692317149	Word Embedding
	startActivity	0.679359509	0.746863252	TF-IDF
	onBackPressed	0.806992088	0.713468276	Word Embedding
Unpopular Methods	onItemClick	0.806934813	0.798380441	Word Embedding
	onNewIntent	0.829393074	0.84397362	TF-IDF
	onMeasure	0.769783452	0.769783452	SAME
	onViewCreated	0.725313427	0.67338287	Word Embedding
	onRestoreInstance	0.65471715	0.472836122	Word Embedding

from official documentation is 0.56. When using TF-IDF as a vectorization technique in TextRank, the similarity is 0.53. In Table 5.4, we show the results of the comparison when considering the long description in Official documentation. In this case, for example, the similarity between the generated summary with word2Vec vectorization and TF-IDF is 0.98 and 0.92, respectively.

As it can be noticed from the results reported in Table 5.4, when the Word embedding technique is applied in the TextRank model to obtain a summary, in 70% of cases, the results are more similar to the official documentation. However, when the TF-IDF technique is applied in the TextRank model for these 20 methods, only 20% of the results are more similar to official documentation than the Word embedding technique. Moreover, solely in the "onMeasure" method, the result of these two tests is similar, which is due to the fact that the number of available posts for the method is very limited.

In the second test, as we explained earlier, we have compared the generated summaries for Android methods with the summaries provided by the official Android documentation. As

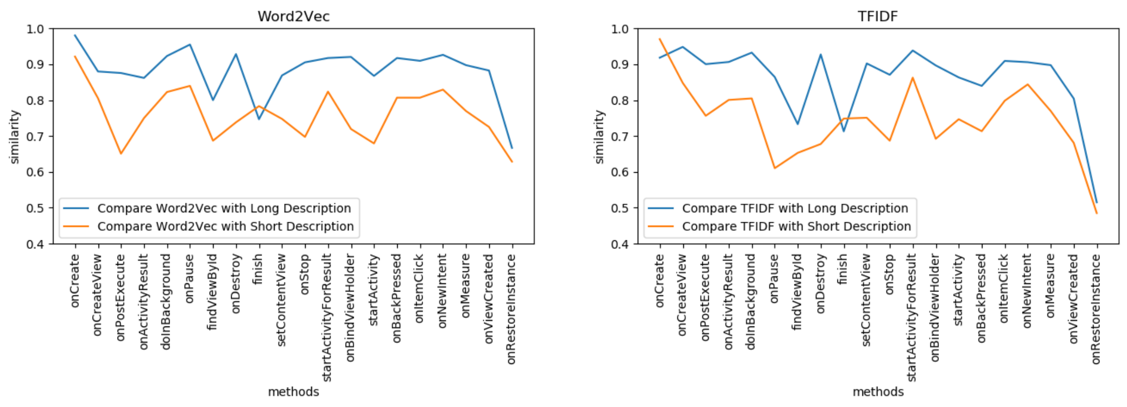
Table 5.4 Comparison of the summarization results with different vectorization with counterpart methods in official documentation (all sentences)

Method		**WordEmbedding_SO**	**TF-IDF_SO**	**Best Result**
Popular Methods	onCreate	0.980570979	0.927563723	Word Embedding
	onCreateView	0.876903606	0.873812941	Word Embedding
	onPostExecute	0.853018139	0.855285499	TF-IDF
	onActivityResult	0.86201519	0.906504674	TF-IDF
	doInBackground	0.923146588	0.932550332	TF-IDF
	onPause	0.955104505	0.86455119	Word Embedding
	findViewById	0.816240886	0.733085415	Word Embedding
	onDestroy	0.928478143	0.927509973	Word Embedding
	finish	0.73386591	0.713104284	Word Embedding
	setContentView	0.869617164	0.906807933	TF-IDF
	onStop	0.89392534	0.870769745	Word Embedding
	startActivityForResult	0.917507208	0.938503803	TF-IDF
	onBindViewHolder	0.920607946	0.896916332	Word Embedding
	startActivity	0.867660855	0.863415078	Word Embedding
	onBackPressed	0.917503777	0.83983228	Word Embedding
Unpopular Methods	onItemClick	0.909683165	0.90944538	Word Embedding
	onNewIntent	0.926397273	0.905906183	Word Embedding
	onMeasure	0.897646704	0.897646704	SAME
	onViewCreated	0.882631761	0.790317452	Word Embedding
	onRestoreInstance	0.683879921	0.492977784	Word Embedding

the results of this test show, when the word embedding method is applied with the TextRank model, the result of this comparison is 65%. When the TF-IDF method is used to compare the automatically generated summaries with the summaries provided in the official Android documentation, the results of the test is 30% (Cf. Table 5.3).

As you can observe in Figure 5.3, the similarities between the generated summaries and official documentation (all sentences in official documentation) varies when tests aiming at finding the similarity have been divided into two tests. In effect, when comparing the generated summaries, using word embedding techniques in the TextRank model, and the description of counterpart methods in official documentation, the similarity reaches around 98%. However, when comparing the generated summaries, using word embedding techniques in the TextRank model, and the summary of counterpart methods in official documentation, the peak occurs around 83%. If we look at the results: in general, we find that when we have applied word embedding vectorization to generate a summary in the TextRank approach, on average in these

20 selected methods, the similarity between these summaries and the descriptions in official documentation and the other test, only with a short description in the official documentation, it is around 88% and 73%, respectively. However, if we used TF-IDF as vectorization techniques in generating summaries, on average these similarities are approximately 85% and 71%.



a) Compare Generated summary with Word2Vec technique with Official Documentation

b) Compare Generated summary with TF-IDF technique with Official Documentation

Figure 5.3 Similarity of the generated documents from word embedding and TF-IDF vectorization with long description and short description in official description

As shown in Figure 5.4, the similarity for the generated summaries in these two tests is not the same. It ranges between 88% for word embedding vectorization technique in the TextRank approach and 85% for the TF-IDF vectorization technique in the TextRank approach in test1, and between 73% for word embedding vectorization and 71% for the TF-IDF vectorization technique in test2. In Test 1 and Test2, the boxplots (Cf. Figure 5.4), and the tables (Cf. Tables 5.3 and 5.4) show that the word embedding vectorization technique in the TextRank approach achieves the best similarities when compared with the description in the official documentation. Using the considered description in the official documentation for comparison results is better than just the summary of official documentation. Comparing between the generated summaries, when using TF-IDF in the TextRank approach and the summary in official documentation, yields lower similarity than the other tests.

The results reveal that by summarizing Android methods with the Word embedding technique, which is trained by a Stack Overflow database, the results are 74% closer to the results extracted from official documentation corresponding to that method. However, in 74% of samples, summaries generated, using the TF-IDF method, do not resemble official documentation.

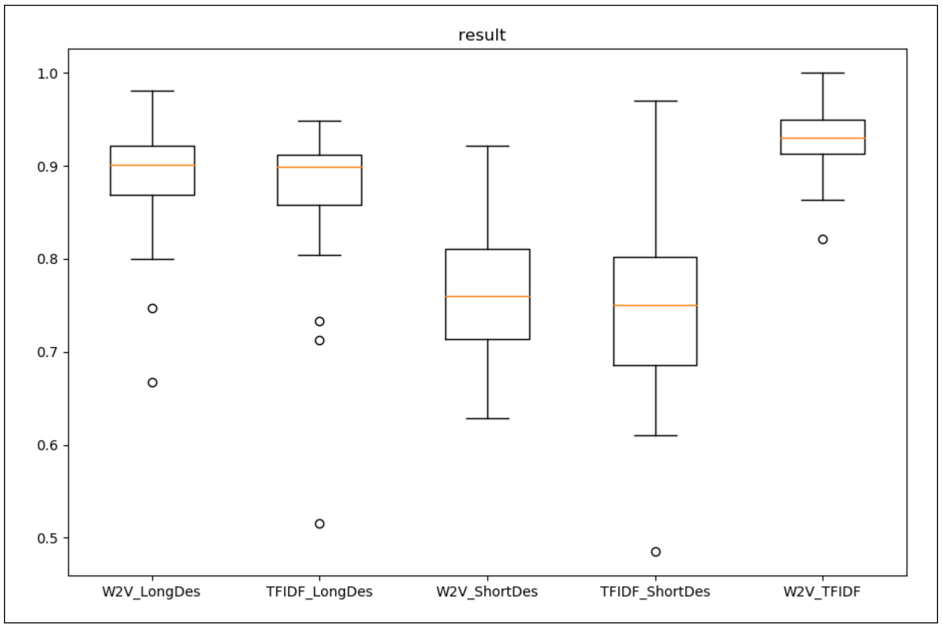


Figure 5.4 Boxplot of similarities between generated summaries and official documentation, for different tests

Another result of these experiments is that the similarity of the summaries created by the TextRank model using two different vectorization techniques is in most cases the same. As shown in Figure 5.4, the similarities between the two generated summaries are between 82 and 100%. In addition, on average, if we compare this result with the result of Test1, the similarity between the created summaries is 4% more than the summary produced by the method of word embedding with the description by the official documentation.

Overall, the findings reveal that when the word embedding technique is employed as a vector technique to automatically generate summaries, the result is more similar to official documentation. Whereas when applying TF-IDF as a vector technique in the TextRank model for summarization, it is less similar to official documentation than the word embedding technique. In addition, based on Figure 5.4, the results show that the comparison of these two automated summaries generated by our approach using the cosine similarity is similar.

5.3 Results of RQ3 - Do the obtained results vary when using different vectorization techniques?

As discussed in the earlier sections, we have utilized the cosine similarity metric to evaluate and assess the summaries generated using the TF-IDF and Word2Vec vectorization in the TextRank model. We have used official documentation as a baseline for comparison with the automatically produced summaries. In this section, we want to show how different vectorization configurations affect the cosine similarity measure.

Under this measurement, we have applied three vectorization techniques to explain the similarity between the automatically generated summaries and the official documentation. Figures 5.5 and 5.6 report the results of the these three measurements. By implementing these three different vectorization techniques, we can determine which type of vectorization is appropriate for the cosine similarity in this context.

In this evaluation, we have conducted six separate experiments for each Android method independently, which are classified into three categories. Each of these categories is related to identifying similarities between the automatically generated summaries and the official documentation, using different vectorizations. Moreover, for every category of vectorization, we have two configurations: sentences considered without pre-processing and pre-processed sentences. Our main focus is on whether the results differ depending on the different vectorizations in cosine similarity or not.

The findings show that, when in this experiment we use TF-IDF as a vectorization technique in cosine similarity, the range of the results of a comparison between the automatically generated summaries and the official documentation as well as between the generated summaries with each other, are more different than the experiment using pre-trained Word2Vec in Stack Overflow or pre-trained GloVe Embedding. According to Figure 5.7, which illustrates the distribution of data, we can observe a difference among TF-IDF, Word2Vec, and GloVe vectorization in terms of cosine similarity. The following plot shows that the distribution between the data when we use Word2Vec and GloVe as a configuration of cosine similarity hold quite similar. For example, if we interpret the Figure a-5.7 based on a graph plot and boxplot, we can notice that, in the terms of the median, the TF-IDF is outside of the box of Word2Vec (SO) and GloVe, and also the data are more dispersed in TF-IDF than the other two vectorization techniques in cosine similarity. As we can see, a similar trend was observed for the Word2Vec (SO) and the GloVe vectorization in cosine similarity, but slightly lower in Word2Vec (SO). The similarity when using GloVe vectorization in cosine similarity becomes even higher than the Word2Vec (SO) vectorization in cosine similarity. Additionally, Figure 5.7 shows that when the GloVe technique is applied in cosine similarity, the results are closer to formal documentation than when the Word2Vec (SO) technique was used.

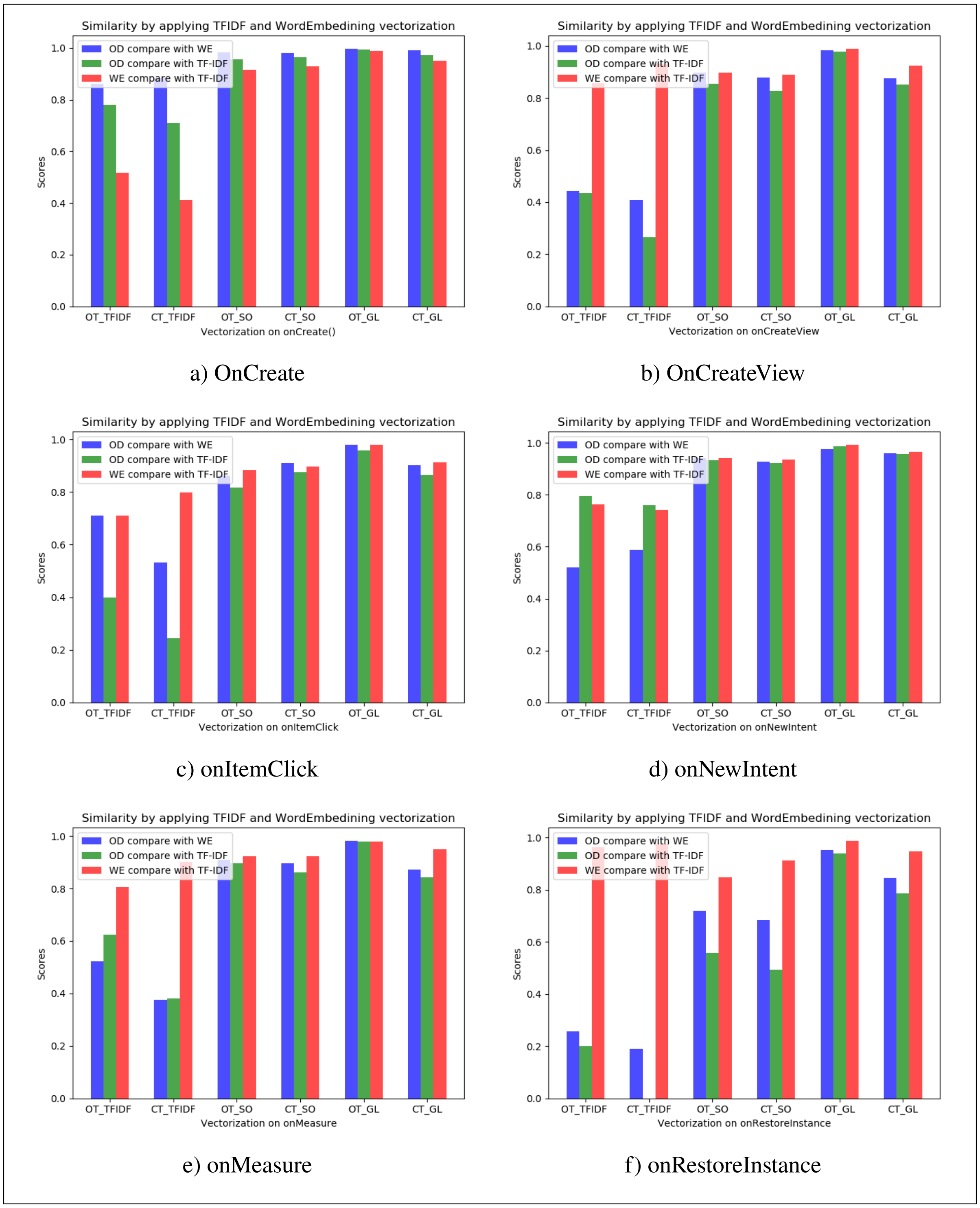
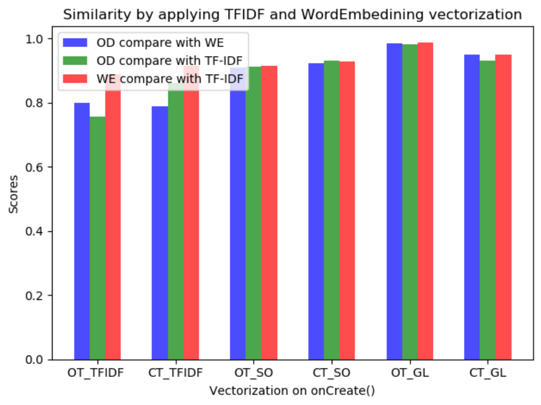
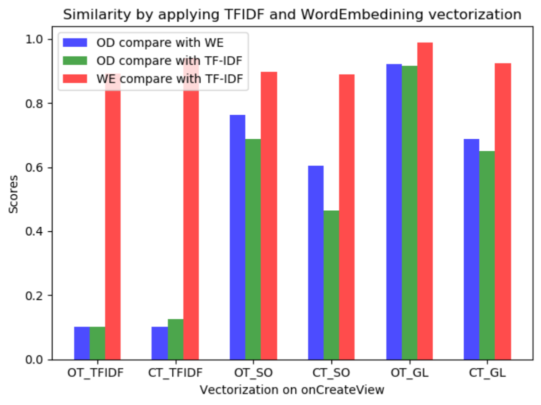


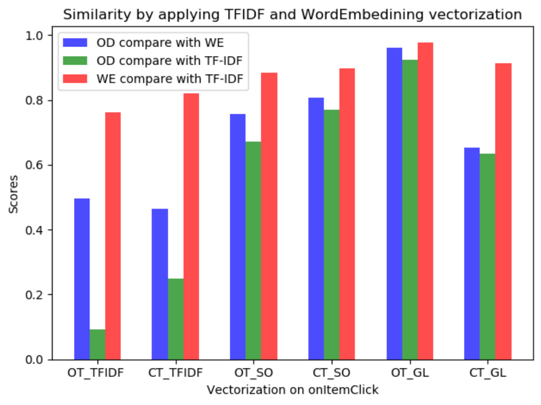
Figure 5.5 Three vectorizations to explain the similarity between the automatically generated summaries and official documentation (All sentences)



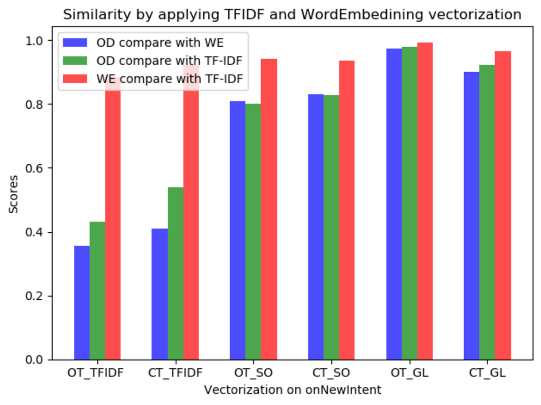
a) OnCreate



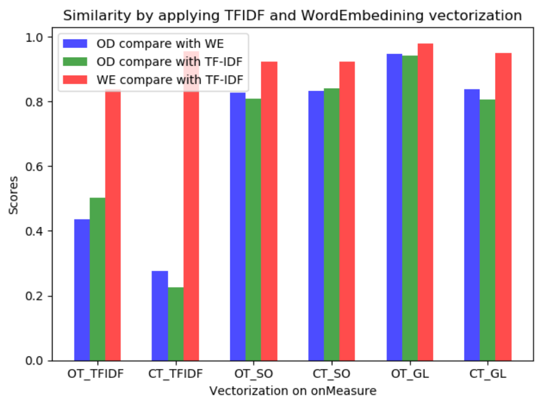
b) OnCreateView



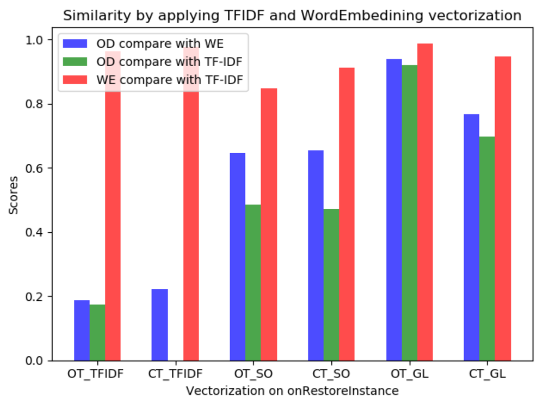
c) onItemClick



d) onNewIntent

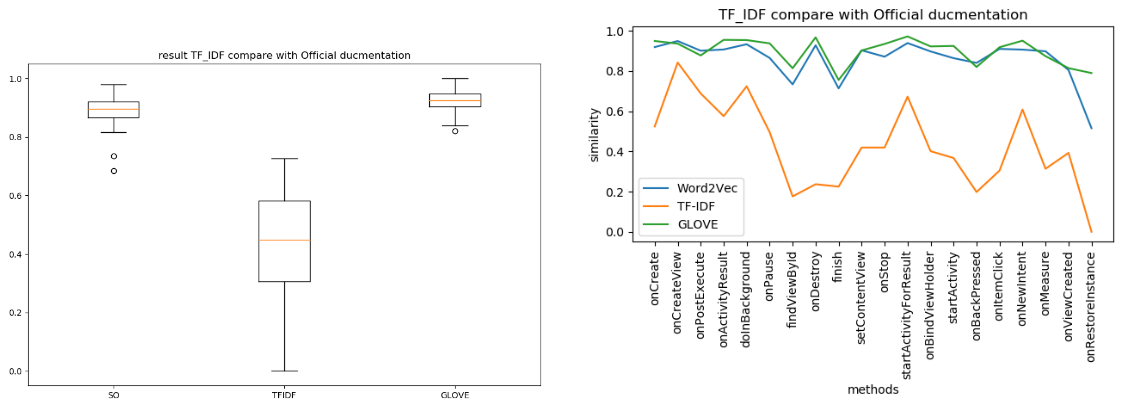


e) onMeasure

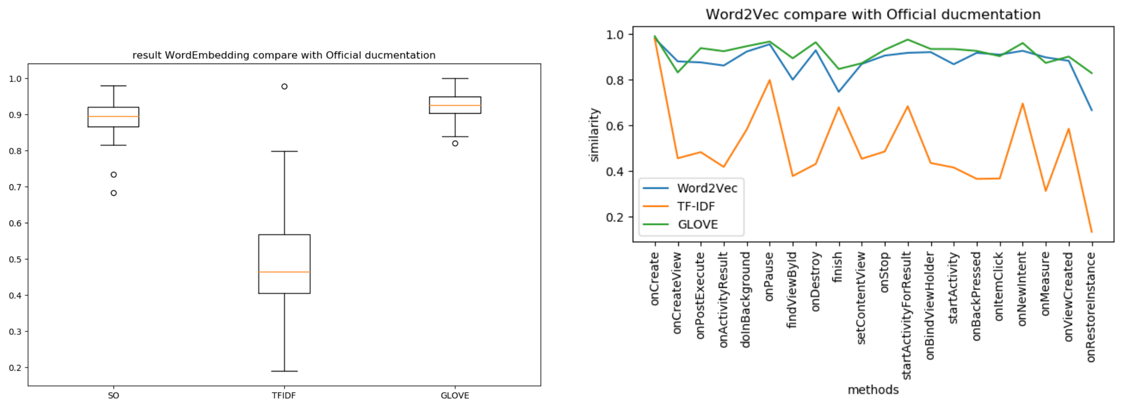


f) onRestoreInstance

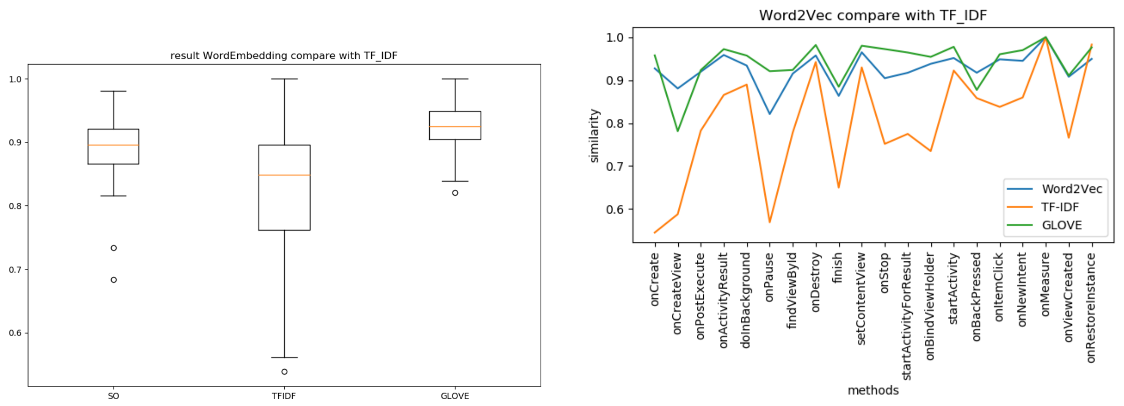
Figure 5.6 Three vectorizations to explain the similarity between the automatically generated summaries and the official documentation (Important Sentences only)



a) TF-IDF comparison with Official documentation



b) Word2Vec comparison with Official documentation



c) Word2Vec comparison with TF-IDF

Figure 5.7 Results of the different configurations examined with the cosine similarity metric

5.4 Threats to Validity

Despite the results obtained in the above section, there are threats to validity related to our empirical study. In the following, we discuss the most prevalent ones to our empirical study:

- **Construct Validity:** One of the main threats to construct validity of our study is the lack of surveys and–or controlled experiments that can help better evaluate and measure the accuracy and usefulness of the generated summaries. Due to COVID-19, we could not run a controlled experiments to fully evaluate our approach. However, as a follow-up of this research work, we have already started conducting surveys with professional developers to evaluate the accuracy of the produced summaries. This investigation will be followed by controlled experiments to study the impact of this summaries on the tasks of software developers.

- **Internal Validity:** One of the main threats to internal validity is the various number of code entities that might be discussed in the same Stack Overflow post, and which may affect the results of our approach since different code entities may result in having similar summaries. We have mitigated this threat to validity by considering *proximity* when summarizing a particular code entity. Specifically, we have limited the number of sentences that are part of the corpus of each code entity to only four sentences that surround the code entity. We plan to tackle this challenge as part of our future work.

  Another threat to validity is related to the versions of Android API methods used. In effect, Android APIs evolve and summaries may have to evolve as well. In this preliminary study, we have not considered the evolution of the code entities and their summaries. In fact, among the 20 code entities that we have examined, only one of them has been deprecated. Android versioning could be a threat to our work, but we have mitigated such a threat since the code entities that we have investigated were among the APIs that have been added mainly in early versions and have been stable over the years, according to the official documentation.

- **External Validity:** Other yet unexploited types of informal documentation can be exploited to explore the ones who is/are useful and practical to generate automatic summaries on the usage of code entities. One can think of considering emails exchanged between developers, tutorials discussing code entities, and–or their combinations for example.

Additionally, our approach is limited to the Java programming language in the Android domain and code entities discussed from Stack Overflow Android posts. Extending our approach to other programming languages such as C#, python and– or other applications' domains would be desirable to generalize our results.

Another threat in this category is related to the restricted number of code entities. In our study, we have selected 20 code entities from 4 android libraries: *android.app*, *android.widget*, *android.os*, and *androidx.recyclerview*. More methods need to be examined in order to generalize the obtained results.

CONCLUSION

In this research, we propose a novel automatic code summarization that summarizes code entities discussed in informal documentation for Android projects.

As informal documentation, we have used methods discussed in Stack Overflow Forum, a popular FAQ forum among software developers. As a platform, we have used the Android mobile platform. We have selected the Android Stack Overflow posts with a score higher than the average score and classify them based on the existence of the code entities. We have examined 20 popular and unpopular methods from 4 different Android libraries. Out of 20 methods, 15 were selected based on the most repetitive methods in Stack Overflow, and 5 of them are not repeated much.

We have also leveraged the NLP technique to convert our data into a comprehensible input for our machine learning algorithm. To see the effect of NLP techniques on the output, we used two techniques, TF-IDF and Word2Vec. The Word2Vec technique used in this study was a pre-trained model that used more than 15GB of Stack Overflow text, which is about 6 billion words. And one of the advantages of this technique is that it carries relevant meanings in the field of software engineering. We applied unsupervised learning, in particular, the TextRank machine learning algorithm. This algorithm has been applied with different configurations: we have used TextRank with both Word2Vec and TF-IDF to generate our automatic summaries. The output of our approach is a summary in the form of NLP sentences that describe the usage of popular Android methods. To evaluate the generated summaries, we have compared the summaries obtained for each method to see the impact of the different NLP techniques. Word2Vec vectorizations techniques capture the semantic between words which yields better results, when comparing our approach to the official documentation. In addition, we have also compared each of the produced summaries with the descriptions provided by the official documentation to investigate if our summaries make sense in this preliminary study. The findings reveal that our summaries share similar content with long descriptions of official documentation. However, we are currently investigating, using a survey, if they contain more insights and relevant information than official documentation.

As future work, we aim to extend our work to other types of informal documentation such as emails exchanged between developers, etc. In addition, we plan to conduct large-scale controlled experiments with Android professional developers to assess the quality and usefulness of the automatically generated summaries. Moreover, We can combine multiple informal documentation such as source code plus stack overflow and possibly link summaries to proper code examples.

APPENDIX I

TABLE OF ANDROID CODE ENTITY SUMMARY

In Table-A I-1, you will observe that our summaries contain sentences from Stack Overflow. The reason is that our approach uses as a dataset the Android Stack Overflow.

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques

Method	Vectorization	Usage summary[1]
onCreate	Word2Vec	[’"As you can see, 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 inside onActivityResult() 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 OnClickListener for your button and implement the onClick() method so that it starts your new Activity."]
	TF-IDF	[’In your MainActivity, call the Injector in the onCreate() method.’, "The problem is that you are ’planting’ trees in the Activities onCreate method.", "As you can see, 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()."]

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onPostExecute	Word2Vec	['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.']
	TF-IDF	['When you use a Thread, you have to update the result on the main thread using the runOnUiThread() method, while an AsyncTask has the onPostExecute() method which automatically executes on the main thread after doInBackground() returns.', 'Note that the AsyncTask run the doInBackground() method in a different thread than the UI thread and onProgressUpdate() and onPostExecute() on the UI thread because they should update UI.', 'When you use AsyncTask you can use the onPostExecute method to return the value to the main thread.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onCreateView	Word2Vec	['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.']
	TF-IDF	["The onCreate() method in a Fragment is called after the Activity's onAttachFragment() but before that Fragment's onCreateView().", "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.", 'In your Fragment onCreateView you inflate the fragment layout and use the view object to initialize views of that layout.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onActivityResult	Word2Vec	['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..).']
	TF-IDF	['As of today, the receipt of a result is bound to the Activity.', "When you launch an activity for result with requestCode &gt;= 0, this code will be returned to the First Activity's onActivityResult() when second activity is finished.You can start multiple Activity for result from your Activity.", '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.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
doInBackground	Word2Vec	['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."]
	TF-IDF	['When you use a Thread, you have to update the result on the main thread using the runOnUiThread() method, while an AsyncTask has the onPostExecute() method which automatically executes on the main thread after doInBackground() returns.', 'Basically, from inside your doInBackground method you call publishProgress() and the AsyncTask class handles all thread-related headaches and calls your onProgressUpdate as soon as possible, on the UI thread, ensuring that you can modify the UI (for example call setText) without any problems.', 'An AsyncTask has several parts that you can override: a doInBackground method that does, in fact, run on a separate thread, and three methods&mdash;onPreExecute, onProgressUpdate, and onPostExecute&mdash;that run on the UI thread.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	**Vectorization**	**Usage summary**
onPause	Word2Vec	['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.']
	TF-IDF	["Activity remains in foreground processes from the time onResume has been called but onPause hasn't been called.", 'In most Activities, you will find that you will need to put code in onResume() and onPause().', 'You should never use the new keyword to instantiate an Activity directly, nor should you call onResume or onPause.']
findViewById	Word2Vec	["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...", '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().']
	TF-IDF	['For this to work, you need to go into your activity_detailed_-view.xml layout and give the first container an id of "@+id/activity_-detailed_view".', 'You have to call setContentView() before you can use findViewById() in your Activity.', 'Your onCreate() must call setContentView() to set the layout before it calls findViewById() to get a view from within that layout.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onDestroy	Word2Vec	['You will see that 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."]
	TF-IDF	['It seems to me that onDestroy is called when GC reclaims the service.', 'You will see that 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.', 'The above code needs to be present also in onResume() method because each time orientation is changed onDestroy() is called and then onCreate() is called so onResume() needs to have this piece of code as well as onCreate().']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
finish	Word2Vec	['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.']
	ŢF-IDF	['You call finish() when you go from 2) to 3).', 'After you call startActivity(...) in the LoginScreen activity, call finish().', 'When you call finish(), the resulting activity which is resuming will be your previous activity.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
setContentView	Word2Vec	["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...", 'In your case 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.']
	TF-IDF	['In your case 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.', 'You are calling setContentView(R.layout.activity_puzzle); in onResume(), but the tvPuzContent variable is instantiated in onCreate(), so it refers to the text view from the original call to setContentView in onCreate.', 'Your onCreate() must call setContentView() to set the layout before it calls findViewById() to get a view from within that layout.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onStop	Word2Vec	["Usually this would look like you are trying to do some transaction for an up coming fragment, meanwhile the host activity already call savedInstanceState method(user may happen to touch the home button so the activity calls onStop(), in my case it's the reason)", "You may be able to track this yourself (e.g., 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."]
	TF-IDF	['When you start Activity from Activty, onPause() and onStop() method called instead of onDestroy().', 'If I call finish() after on-Pause(), I see onStop(), and onDestroy() immediately called.', "Well, if you study the structure of how the application life-cycle works,here , then you'll come to know that onPause() is called when another activity gains focus, and onStop() is called when the activity is no longer visible."]

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
startActivityForResult	Word2Vec	['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.']
	TF-IDF	['You can do this by introducing proxy activity: A ->PendingIntent ->ProxyActivity –>startActivityForResult –>B.', "When you launch an activity for result with requestCode &gt;= 0, this code will be returned to the First Activity's onActivityResult() when second activity is finished.You can start multiple Activity for result from your Activity.", "In this case you should implement onActivityResult in Activity2, handle the results coming back from Activity3 and set them as Activity2's results to pass back to Activity1 and then finish; An activity will only receive results from activities it directly starts via startActivityForResult."]

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onBindViewHolder	Word2Vec	['Good idea is to make a class object with all data you need for one item in recycler view, also add there one boolean isItemWasClicked and inside onBindViewHolder() check this boolean and make buttons visible or not.', 'Once the view holder is attained (whether by creating a new one, or grabbing a scrapped / recycled one), it then calls onBindViewHolder from the adapter to set the correct data for the view holder.', 'In short, the "item view cache" holds elements in such a way that the RecyclerView can avoid calling both onCreate-ViewHolder() and onBindViewHolder(), whereas the recycled view pool holds elements such that the RecyclerView can avoid calling on-CreateViewHolder() but will still have to call onBindViewHolder().']
	TF-IDF	['You have not posted your ViewHolder code, but you should use holder.view.setBackgroundColor(...) instead of view.setBackgroundColor(...) in your onBindViewHolder method.', 'They use the same view holder created using the onCreate-ViewHolder() method.And each time any one of them comes into view the onBindViewHolder() method is called.', "If the RecyclerView view doesn't have an available view holder for this item type, it calls onCreateViewHolder in order to create a new view holder for this item type."]

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
startActivity	Word2Vec	['There is one thing to be aware of, though - as pointed out in the Service.startActivity() doc, "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.']
	TF-IDF	['Now when you call startActivity(), it looks at intent and knows that it needs start the SomeOtherActivity class.', '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.', 'When you call startActivity(), Android will put your current Activity to the background, and start (or bring to foreground) the new Activity specified in your Intent.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onBackPressed	Word2Vec	["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.']
	TF-IDF	['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.', 'You should override the onBackPressed() method in the activity scope and call it from your method.', 'There is no onBackPressed() method in fragment, and this method is just for the activity.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	**Vectorization**	**Usage summary**
onItemClick	Word2Vec	['It also looks like you meant to have the Activity implement AdapterView.OnItemClickListener - then the overridden method named onItemClick would get called when a list item is clicked.', 'The onItemClickListener method has an onClick interface already with the view and position passed into it: You can get check which view has been clicked by doing a switch statement and performing a dedicated action when the specific view has been clicked.', 'For example, if only one item can be selected, you might just set a member variable, say mPosition on the Adapter itself to the position passed in to onItemClick, and then check in getView if position == mSelectedPosition to decide how to set the View.']
	TF-IDF	['It also looks like you meant to have the Activity implement AdapterView.OnItemClickListener - then the overridden method named onItemClick would get called when a list item is clicked.', 'EDIT: Be aware that Android reuses list item View objects (also called view recycling), so the toggle state of each item needs to be stored for later use.', "Your ListView's Adapter contains a method called getView, which is called when a list view item needs to be displayed in an actual View."]

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onTouchListener	Word2Vec	['Try setting a onTouchListener and in the onTouch method display your popup and return true to consume the event and stop it from propagating to the view (Spinner in this case).', "Hence to pass on the touch event to the parent's (View Group) onTouchListener make sure to return true in the onInterceptTouchEvent method of the Parent (ViewGroup).", "Given that you will be using your own view you won't use an onTouchListener() but instead override the view's onTouchEvent(MotionEvent event)."]
	TF-IDF	['Implement onTouchListener on the view to get the touch events and process them according to their position.', 'The problem is that when you click an item in your list dialog, the touch event of the dialog is also fired causing it to be shown again.', 'You would also have to set BOTH onTouchListener and onClickListener on your views.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onNewIntent	Word2Vec	['However, remember that you will need to use onNewIntent if the Activity is already running an a new Intent is sent to the Activity to "start" it.', "When you get a new intent and either your activity is using singleTop or Intent is using FLAG_ACTIVITY_SINGLE_TOP, your new intents actually get delivered to the Activity's onNewIntent.", 'When your activity is singleTop or you call it with FLAG_ACTIVITY_SINGLE_TOP flag in the Intent, the new Intent object will be passed to the onNewIntent method of your activity if it is already running.']
	TF-IDF	['When your activity is singleTop or you call it with FLAG_ACTIVITY_SINGLE_TOP flag in the Intent, the new Intent object will be passed to the onNewIntent method of your activity if it is already running.', 'The closest you will be able to come is to use a distinctive Intent in the getActivity() PendingIntent (e.g., a custom action string), use setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP|Intent.FLAG_ACTIVITY_SINGLE_TOP) on the Intent to try to bring an existing instance of your activity forward, then check for your special Intent in both onCreate() and onNewIntent() of your activity and do whatever special thing you want done.', 'According to android developer site, In FLAG_ACTIVITY_CLEAR_TOP,FLAG_ACTIVITY_SINGLE_TOP and in launch mode : SingleTask , If the instance of activity you are calling is already running then instead of creating new instance, it call onNewIntent() method.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onMeasure	Word2Vec	['Your custom view needs to override the onMeasure method and properly set the measured width and height so that the parent views (in this case the ScrollView) can know how much space to allocate for the child.', 'However, if you want a more flexible approach, you can override the onMeasure() method to measure the view more precisely depending on the space available and layout constraints (wrap_content, match_parent, or a fixed size).', 'If the size of this view also changes you may need to override the onMeasure method too.']
	TF-IDF	['Your custom view needs to override the onMeasure method and properly set the measured width and height so that the parent views (in this case the ScrollView) can know how much space to allocate for the child.', 'However, if you want a more flexible approach, you can override the onMeasure() method to measure the view more precisely depending on the space available and layout constraints (wrap_content, match_parent, or a fixed size).', 'If the size of this view also changes you may need to override the onMeasure method too.']

Continued on next page

Table-A I-1 Automatically generated Summaries from Android Methods using our new approach, when applying two separate vectorization techniques (continued)

Method	Vectorization	Usage summary
onViewCreated	Word2Vec	['Now, I wouldn\'t use myFragment.showName("name"); because you don\'t know if the lifecycle of the fragment has already finished (attached to the activity and inflated the views), so instead, I would call the showName("name") in the onActivityCreated or onViewCreated callbacks.', 'Try to use postponeEnterTransition() in your second activity inside onCreate() and yourActivity.startPostponedEnterTransition() in your fragment after you created your view in onViewCreated().', 'Inside Fragment class we get onViewCreated() override method where we should always initialize our views because in this method we get view object.']
	TF-IDF	['Use the view returned in onViewCreated() to restrict your view search to just the layout you inflated in onCreateView().', 'In the onCreate method of your Activity or in the onViewCreated method of your fragment.', 'Always remember in case of Fragment that onViewCreated() method will not called automatically if you are returning null or super.onCreateView() from onCreateView() method.']
onRestoreInstance	Word2Vec	['You can achieve this by saving the lastItemSelected position in a int field of your Activity called lastItemSelected (remember to update this variabe every time you click on a element of the list)', 'Of course, you can use listview.smoothScrollToPosition(position) but that would be weird.', 'When you retrieve this position in your onRestoreInstance, you can scroll back to this position using listview.scrollTo(0, position).']
	TF-IDF	['When you retrieve this position in your onRestoreInstance, you can scroll back to this position using listview.scrollTo(0, position).', 'Of course, you can use listview.smoothScrollToPosition(position) but that would be weird.', 'It is pretty easy to save the items of your Adapter in onSaveInstance.']

[1] The summaries below are generated based on data from the following sources: https://stackoverflow.com, https://developer.android.com

BIBLIOGRAPHY

Abdalkareem, R., Shihab, E. & Rilling, J. (2017). On code reuse from stackoverflow: An exploratory study on android apps. *Information and Software Technology*, 88, 148–158.

Abid, N. J., Sharif, B., Dragan, N., Alrasheed, H. & Maletic, J. I. (2019). Developer reading behavior while summarizing java methods: Size and context matters. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 384–395.

Acar, Y., Backes, M., Fahl, S., Kim, D., Mazurek, M. L. & Stransky, C. (2016). You get where you're looking for: The impact of information sources on code security. *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 289–305.

Amin, S., Uddin, M. I., Hassan, S., Khan, A., Nasser, N., Alharbi, A. & Alyami, H. (2020). Recurrent neural networks with TF-IDF embedding technique for detection and classification in tweets of dengue disease. *IEEE Access*, 8, 131522–131533.

Bahdanau, D., Cho, K. & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bakhshandeh, R. & Azimifar, Z. (2012). Personalized search based on micro-blogging social networks. *The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012)*, pp. 283–286.

Basili, V. R., Caldiera, G. & Rombach, H. D. (1994). The Experience Factory, Encyclopedia of Software Engineering. *Wiley*, 469–476.

Baumel, T., Eyal, M. & Elhadad, M. (2018). Query focused abstractive summarization: Incorporating query relevance, multi-document coverage, and summary length constraints into seq2seq models. *arXiv preprint arXiv:1801.07704*.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine.

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.

Das, B. & Chakraborty, S. (2018). An improved text sentiment classification model using TF-IDF and next word negation. *arXiv preprint arXiv:1806.06407*.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K. & Harshman, R. (1990). Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6), 391–407.

Deshpande, A. R. & Lobo, L. (2013). Text summarization using Clustering technique. *International Journal of Engineering Trends and Technology*, 4(8), 3348–3351.

Efstathiou, V., Chatzilenas, C. & Diomidis, S. (2018). “Word Embeddings for the Software Engineering Domain”. *In Proceedings of the 15th International Conference on Mining Software Repositories,*, 38–41.

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.

Haiduc, S., Aponte, J., Moreno, L. & Marcus, A. (2010). On the use of automated text summarization techniques for summarizing source code. *Reverse Engineering (WCRE), 2010 17th Working Conference on*, 35–44.

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.

Hochreiter, S. & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780.

Hu, X., Li, G., Xia, X., Lo, D. & Jin, Z. (2018). Deep code comment generation. *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pp. 200–20010.

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.

Jiang, H., Nazar, N., Zhang, J., Zhang, T. & Ren, Z. (2017a). PRST: A PageRank-Based Summarization Technique for Summarizing Bug Reports with Duplicates. 27, 869-896.

Jiang, S., Armaly, A. & McMillan, C. (2017b). Automatically generating commit messages from diffs using neural machine translation. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 135–146.

Khan, A., Salim, N., Farman, H., Khan, M., Jan, B., Ahmad, A., Ahmed, I. & Paul, A. (2018). Abstractive text summarization based on improved semantic graph approach. *International Journal of Parallel Programming*, 46(5), 992–1016.

Kobayashi, H., Noguchi, M. & Yatsuka, T. (2015). Summarization based on embedding distributions. *Proceedings of the 2015 conference on empirical methods in natural language processing*, pp. 1984–1989.

Korayem, M. (2019). *Towards automatic context-aware summarization of code entities*. (Master's thesis, École de technolgie supérieure). Consulted at https://espace.etsmtl.ca/id/eprint/2433/.

LeClair, A., Jiang, S. & McMillan, C. (2019). A neural model for generating natural language summaries of program subroutines. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 795–806.

Li, B. & Han, L. (2013). Distance weighted cosine similarity measure for text classification. *International Conference on Intelligent Data Engineering and Automated Learning*, pp. 611–618.

Mamykina, L., Manoim, B., Mittal, M., Hripcsak, G. & Hartmann, B. (2011). Design lessons from the fastest q&a site in the west. *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 2857–2866.

Marcus, A., Maletic, J. I. & Sergeyev, A. (2002). Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 811–836.

McBurney, P. W. & McMillan, C. (2014). Automatic Documentation Generation via Source Code Summarization of Method Context. 279–290.

McBurney, P. W. & McMillan, C. (2015). Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering*, 42(2), 103–119.

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

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.

Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013a). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 8.

Mikolov, T., Chen, K., Corrado, G. & Dean, J. (2013b). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

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

Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L. & Vijay-Shanker, K. (2013). Automatic generation of natural language summaries for java classes. *2013 21st International*

Conference on Program Comprehension (ICPC), pp. 23–32.

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

Nallapati, R., Zhou, B., Gulcehre, C., Xiang, B. et al. (2016). Abstractive text summarization using sequence-to-sequence rnns and beyond. *arXiv preprint arXiv:1602.06023*.

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

Papineni, K., Roukos, S., Ward, T. & Zhu, W.-J. (2002). BLEU: a method for automatic evaluation of machine translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318.

Pennington, J., Socher, R. & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Pham, D.-H. & Le, A.-C. (2018). Exploiting multiple word embeddings and one-hot character vectors for aspect-based sentiment analysis. *International Journal of Approximate Reasoning*, 103, 1–10.

Phan, A. V., Chau, P. N., Le Nguyen, M. & Bui, L. T. (2018). Automatically classifying source code using tree-based approaches. *Data & Knowledge Engineering*, 114, 12–25.

Piskorski, J. & Jacquet, G. (2020). TF-IDF Character N-grams versus Word Embedding-based Models for Fine-grained Event Classification: A Preliminary study. *Proceedings of the Workshop on Automated Extraction of Socio-political Events from News 2020*, pp. 26–34.

Prajapati, R. & Kumar, S. (2016). Enhanced weighted pagerank algorithm based on contents and link visits. *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 1850–1855.

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

Rastkar, S., Murphy, G. C. & Murray, G. (2014). Automatic Summarization of Bug Reports. *Trans. on Soft Eng.*, 40(4), 366–380.

Reimers, N. & Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *IEEE software*, 26(6), 27–34.

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.

Rosen, C. & Shihab, E. (2016). What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3), 1192–1223.

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. 195–205.

Shen, Y. (2015). "Word embedding based correlation model for question/answer matching". *arXiv preprint arXiv*, 1511.04646.

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L. & Vijay-Shanker, K. (2010). Towards automatically generating summary comments for java methods. *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pp. 43–52.

Subramanian, N. B. (2019, September, 23). Introduction to Bag of Words, N-Gram and TF-IDF | Python [Blog Post]. Consulted at https://aiaspirant.com/bag-of-words/.

Treude, C., Barzilay, O. & Storey, M.-A. (2011a). How do programmers ask and answer questions on the web?(NIER track). *Proceedings of the 33rd international conference on software engineering*, pp. 804–807.

Treude, C., Barzilay, O. & Storey, M.-A. (2011b). How do programmers ask and answer questions on the web?(nier track). *Proceedings of the 33rd international conference on software engineering*, pp. 804–807.

Uddin, G. & Robillard, M. P. (2015). How API documentation fails. *IEEE Software*, 32(4), 68–75.

Wang, P., Xu, B., Xu, J., Tian, G., Liu, C.-L. & Hao, H. (2016). Semantic expansion using word embedding clustering and convolutional neural network for improving short text classification. *Neurocomputing*, 174, 806–814.

Ying, A. T. T., Murphy, G. C., Ng, R. T. & Chu-Carroll, M. (2004). Predicting Source Code Changes by Mining Change History. *IEEE Trans. Software Eng.*, 30(9), 574-586.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K. & Liu, X. (2019). A novel neural source code representation based on abstract syntax tree. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794.

Zhou, Y., Yan, X., Yang, W., Chen, T. & Huang, Z. (2019). Augmenting Java method comments generation with context information based on neural networks. *Journal of Systems and Software*, 156, 328–340.

Zou, W. Y., Socher, R., Cer, D. & Manning, C. D. (2013). Bilingual word embeddings for phrase-based machine translation. *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1393–1398.