Towards Recommending Code Examples Using Informal Documentation

by

Sajjad RAHMANI

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

MONTREAL, FEBRUARY 07, 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

TABLE OF CONTENTS

Page

INTRODUCTION 1

CHAPTER 1 TECHNICAL BACKGROUND 7
1.1 Neural Networks, Transformers and BERT 7
1.2 Locality Sensitive Hashing 10
1.2.1 Query-aware vs Query-oblivious LSH 11
1.2.2 Cosine Similarity 14

CHAPTER 2 RELATED WORK 17
2.1 Code Element Extraction from Informal Documentation 17
2.2 Recommendation Systems based on Usage Patterns 19
2.3 Recommendation Systems based on Related Code Search 22
2.4 Locality Sensitive Hashing for Recommendation Systems 25

CHAPTER 3 METHODOLOGY 29
3.1 Overview of the Followed Methodology 29
3.1.1 Collecting Data 30
3.1.2 Pre-processing Data 30
3.1.3 Extracting Code Examples from Informal Documentation 31
3.1.4 Applying the BERT model 32
3.1.5 Locality Sensitive Hashing (LSH) vs Clustering 34
3.1.6 Applying *LSH* Algorithms 35
3.1.6.1 Random Hyperplane-based LSH 36
3.1.6.2 Query-aware approach 38
3.1.7 Model Assessment 40

CHAPTER 4 EMPIRICAL EVALUATION 43
4.1 Definition and planning of the study 43
4.2 Research questions 43
4.3 Variable section 44
4.4 Analysis Method 46

CHAPTER 5 RESULTS AND DISCUSSION 49
5.1 Code Examples Recommendations based on Informal documentation 49
5.2 *Results of RQ11: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of HitRate*? 53
5.3 *Results of RQ12: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Mean Reciprocal Rank?* 57
5.4 *Results of RQ13: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Average execution time*? 61

5.5 *Results of RQ14: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Relevance*? 66
5.6 *Results of RQ2: Is there any difference in terms of Relevance values between the Random Hyperplane-based LSH and the Query-Aware LSH algorithms?* 69
5.7 Threats to validity 71

CHAPTER 6 CONCLUSION AND FUTURE WORK 73

APPENDIX I TABLES OF RECOMMENDED CODE EXAMPLES 77

REFERENCES 87

LIST OF FIGURES

Page

Figure 1.1 BERT Embedding Taken from Devlin, Chang, Lee & Toutanova (2018) 9

Figure 1.2 An example of two hyperplanes in LSH 10

Figure 1.3 Query-Oblivious bucket allocation Taken from Huang, Feng, Zhang, Fang & Ng (2015) 12

Figure 1.4 Query-Aware bucket allocation Taken from Huang *et al.* (2015) 13

Figure 3.1 The main steps of the followed Methodology 29

Figure 3.2 The overall structure of *BERT* 33

Figure 5.1 The rank of the first code examples generated for natural language-based queries 60

Figure 5.2 The rank of the first code examples generated for API names-based queries 61

Figure 5.3 LSH creation time based on the different umber of Hash tables 63

Figure 5.4 The Boxplot of the LSH creation time (sec) based on the different number of Hash tables for the two applied algorithms 64

Figure 5.5 LSH recommendation time based on the number of Hash tables 65

Figure 5.6 The Boxplot of the LSH recommendation time (sec) based on the different number of Hash tables for the two applied algorithms 65

Figure 5.7 Relevance score of the two leveraged algorithms in case of natural language-based queries 68

Figure 5.8 Relevance score of the two studied algorithms in case of API methods' names-based queries 69

LIST OF ABBREVIATIONS

ETS	École de Technologie Supérieure
BERT	Bidirectional Encoder Representations from Transformer
ANN	Approximate Nearest Neighbors
LSH	Locality-Sensitive Hashing
MRR	Mean Reciprocal Rank
LSTM	Long Short-Term Memory
RNN	Recurrent Neural Networks
FQN	Fully Qualified Names
AST	Abstract Syntax Tree
FOCUS	API FunctiOn Calls and USage patterns
QA	Query-Aware
RH	Random Hyperplane-based

Vers la Recommandation d'Exemples de Code à l'Aide de la Documentation Informelle

Sajjad RAHMANI

RÉSUMÉ

Au cours des dernières années, un ensemble de travaux de recherche dans le domaine de la recommandation des exemples de code a été accompli afin de faciliter les tâches de développement de logiciels pour les développeurs. En effet, ces derniers passent beaucoup de temps sur Internet pour trouver les exemples de code pertinents pour leurs tâches, y compris les projets open-source et la documentation informelle. La documentation informelle est une ressource inestimable pour trouver des exemples de code utiles incorporant des discussions, des forums, des e-mails, le Stack Overflow, etc. Dans nos recherches, nous nous sommes concentrés sur le Stack Overflow qui est l'un des sites Web les plus populaires englobant les discussions des développeurs logiciels autour de différents langages de programmation. Nous avons collecté et recommandé les meilleurs exemples de code dans le langage de programmation Java, qui ont été étiquetés comme des exemples de code dans les discussions du type réponses pour augmenter la qualité des exemples de code recommandés. Dans notre approche, nous avons appliqué les représentations d'encodeurs bidirectionnels à partir de transformateurs (BERT), un modèle de représentation de texte capable d'extraire correctement les informations sémantiques du texte ; Nous avons utilisé BERT pour coder des exemples de code avec des vecteurs numériques. Dans l'étape suivante, nous avons appliqué le hachage sensible à la localité (LSH) qui est appliqué pour trouver les voisins les plus proches approximatifs (ANN). Nous avons appliqué deux versions de cette approche, *i.e.*, *Random Hyperplane-based* et *Query-Aware*. Ces algorithmes sont comparés en fonction de plusieurs paramètres. Nous avons comparé ces deux algorithmes en termes de quatre paramètres, notamment *HitRate*, *Mean Reciprocal Ranking (MRR)*, *Average Execution Time* et *Relevance*. Les résultats de la comparaison montrent que l'approche *Query-Aware* est performante par rapport au *Random Hyperplane-based*. L'algorithme *Query-Aware* agit en tenant compte de la requête, *i.e.*, la méthode pour laquelle nous faisons des recommendations des exemples de code. Il est capable, dans un premier lieu, de filtrer les exemples de code non pertinents aux premières étapes. Ensuite, il recommande des exemples de code supposés être pertinents en se basant sur les valeurs de la requête, par rapport à l'approche basée sur l'hyperplan aléatoire.
Nous avons comparé ces deux algorithmes sur la base des paramètres mesurés. Les conclusions de notre étude ont montré que l'approche *Query-Aware (QA)* est plus performante que l'approche *Random Hyperplane-based (RH)* (de 20% à 35% respectivement) quand elle est basée sur des paires de requêtes dans *HitRate*.

Mots-clés: Stack Overflow, BERT, LSH, ANN, LSH sensible aux requêtes, basé sur un hyperplan aléatoire, HitRate, MRR, pertinence

Towards Recommending Code Examples Using Informal Documentation

Sajjad RAHMANI

ABSTRACT

Past and recent research in the code example recommendation area has been accomplished to guide developers during their software development tasks. In effect, programmers spend a considerable amount of time on the Internet to find the relevant code examples to their tasks using source code of open-source projects and–or informal documentation. Informal documentation is an invaluable resource for finding helpful code examples trapped in Stack Overflow discussions, forums, emails exchanged between developers, etc.

In our research, we have focused on Stack Overflow, which is one of the most popular resources for software developers' discussions on different topics. We have collected and recommended the best code examples in the Java programming language that have been labeled as the answer code examples in order to increase the quality of the recommended code examples.

Our approach has applied the *Bidirectional Encoder Representations from Transformers (BERT)*, a model for text representation that is able to extract the semantic information of textual data properly. We have used *BERT*, in a first stage, to encode code examples with numerical vectors. In the next step, we have applied *Locality-sensitive hashing (LSH)* that has been leveraged to find the *Approximate Nearest Neighbors (ANN)*. In our research, we have applied two versions of this approach, *i.e.*, the *Random Hyperplane-based* and the *Query-Aware* LSH.

We have compared these two algorithms based on four parameters including *HitRate*, *Mean Reciprocal Rank (MRR)*, *Average Execution Time*, and *Relevance*. The findings of our study have shown that the *Query-Aware (QA) approach* performs better, in terms of *HitRate*, compared to the *Random Hyperplane-based (RH)* (from 20% to 35%) for query pairs respectively. *QA* is at least 4 times faster than *RH* for creating Hashing tables and assigning data samples to buckets. it returns code examples in milliseconds (ms), however, it takes seconds (sec) for *RH* to recommend the code examples.

The *QA algorithm* takes into account the query, *i.e.*, a Natural Language query or an API method and it is able to filter out the irrelevant code examples at the initial steps of the algorithm. Additionally, based on the evaluation of the obtained results, *i.e.*, recommended code examples by two experienced software developers and the metrics' values, *QA approach* recommends more similar examples based on the query values compared to the *RH approach*.

Keywords: Stack Overflow, BERT, LSH, ANN, Query-Aware LSH, Random Hyperplane-based, HitRate, MRR, Relevance

INTRODUCTION

Recommendation systems are widely used in various fields to improve the efficiency and quality of various tasks. According to Zhou, Shen & Zhong (2019), software developers often write similar code examples multiple times because they need to implement similar functionalities in different projects. A recommendation system can therefore help programmers complete their tasks quickly and effectively, during the software development process, by returning the most relevant and high-quality examples that have been written by other programmers (Di Rocco, Di Ruscio, Di Sipio, Nguyen & Rubei (2021)). To perform their programming tasks, developers rely on two main sources of information: open-source projects and informal documentation. Open-source projects, such as the ones on GitHub, provide examples for different tasks and offer their code resources for use.

On the other side, informal documentation refers to data sources that are primarily used for exchanging information and ideas among developers about various tasks. These sources often include incomplete code examples (Kim *et al.* (2018)). Examples of informal documentation include bug reports, emails, and Stack Overflow. However, code examples in these types of resources are often embedded in natural language comments, making it difficult to extract code entities and elements. The goal of this thesis is to propose a recommendation system that suggests relevant code examples from Stack Overflow based on the developers' tasks and what they need. Stack Overflow was selected as the data source for our recommendation system because it is one of the most popular resources among developers for solving programming issues (Rubei, Di Sipio, Nguyen, Di Rocco & Di Ruscio (2020)).

It includes a large number of high-quality code examples that have been used and validated by programmers as helpful. This study focuses on Java code examples that were posted on Stack Overflow and labeled as answers to questions. The posts were made between 2008 and May 2022 and were saved in dump files. After pre-processing, we have converted the code examples from

Stack Overflow into numerical vectors using the *BERT* model and implemented two *LSH-based* algorithms *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH* to reduce the search space and find similar code examples based on a user's query.

Motivation

Software developers constantly seek ways to complete their tasks more quickly and efficiently. A common challenge they face is finding examples of how to use API methods that are applicable to their tasks. Searching various resources, including open-source projects and informal documentation, can be time-consuming and difficult. A recommendation system that suggests high-quality code examples for desired usage of API methods can be beneficial, time and effort-saving.

There are already some research works that have focused on using open-source projects in the context of code examples recommendation system, with the purpose of guiding developers during their development tasks. For instance, *FOCUS* (Nguyen *et al.* (2019)), *MAPO* (Zhong, Xie, Zhang, Pei & Mei (2009)) and *UP-Miner* (Wang *et al.* (2013)) are examples of code example recommendation systems that have used open-source projects to find and recommend the best code examples to developers. In these and other studies, using open-source projects as the source for code example recommendation systems is relatively straightforward because the entire code source is available and there are tools such as Eclipse JDT that have been designed to extract code entities and elements from these resources.
In contrast, there are fewer studies that have used informal documentation as a source for recommendation systems. Extracting code entities, such as method declarations and method invocations, from informal documentation is more challenging because code examples are incomplete and often trapped in natural language comments (Fischer *et al.* (2017)). For instance, PostFinder (Rubei *et al.* (2020)) is a plugin in Eclipse that attempts to recommend Stack Overflow posts by finding similarity between the developing project and Stack Overflow posts. FaCoy

(Kim *et al.* (2018)) is another code-to-code recommendation system that suggests code examples from Stack Overflow posts based on a query code example.

In our research, we have also focused on Stack Overflow as a type of informal documentation, since it is the most popular question-and-answer source (Rubei *et al.* (2020)) that embraces useful code examples and it is a useful resource for exchanging programmers' knowledge (Guerrouj, Bourque & Rigby (2015)). Additionally, we have dealt with Java programming language because it is popular and a large number of uploaded projects on GitHub are written with this language (Kim *et al.* (2018)). Hence, a vast number of developers spend their effort to find Java code examples that are related to their task on Stack Overflow. In this research, we aim to meet developers' requirements for finding and recommending relevant code examples among a large number of Stack Overflow code snippets.

Problem statement

As previously mentioned, the majority of research on code example recommendation systems has focused on using open-source projects as a data source to provide users with useful examples of API methods. Open-source projects have the advantage of having the entire source code available, making it easier to extract information about code entities *i.e.*, classes, methods, and variables. However, there are some problems with the recommendation systems that leverage open-source projects. First, they are intended only for object-oriented programming languages (Zhong *et al.* (2009)). Second, the clustering-based recommendation systems have a high degree of redundancy (Nguyen *et al.* (2019)). Furthermore, no recommendation candidate exists for less frequent API usage (Grahne & Zhu (2003)). Finally, no measurement metric is introduced to evaluate the quality of the mined patterns (Zhong *et al.* (2009)).

In contrast, code examples in informal documentation are often embedded in natural language sentences, which makes it difficult to extract the code. These examples are also often incomplete because the goal of the writer is usually to share their knowledge about how to use different classes and methods, rather than how to include the entire source code in their posts or comments. Some research, such as Diamantopoulos & Symeonidis (2015), has attempted to extract code entities from Stack Overflow posts, but with limited success, only achieving a success rate of 10%. Additionally, recent research works that have focused on code example recommendations from Stack Overflow, require both code examples and natural language comments to return relevant code examples based on the programming context (Rubei *et al.* (2020) and Kim *et al.* (2018)).

In this research, we extracted code examples from Stack Overflow posts and used the *BERT* model to convert them into numerical vectors that incorporate the tokenized and semantic information of the examples. Then, we implemented two *LSH*-based algorithms *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH* to find the most similar code example based on a query value. Finally, we ranked the retrieved code examples based on their similarity score to the query.

We applied *BERT* and *LSH* for two main reasons. First, since *BERT* is able to extract semantic information from natural language documents in different tasks, such as translating between different languages (Devlin *et al.* (2018)), we believe it could potentially extract semantic information from code examples and identify similar examples. Second, given that the number of retrieved data samples exceeds 60K, finding relevant code examples can be time-consuming. LSH-based approaches, *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH*, can reduce the search space and aid in finding pertinent code examples.

Finally, we investigate two main research questions to evaluate our *LSH*-based algorithms:
RQ1 How do the *Random Hyperplane-based LSH* and the *Query-Aware LSH* perform when recommending code examples, in terms of *HitRate, Mean Reciprocal Rank, Average execution time*, and *Relevance*?

RQ2 Is there any difference in terms of *Relevance* values between the *Random Hyperplane-based LSH* and the *Query-Aware LSH* algorithms?

Outline: This thesis has 6 chapters. In the first chapter, we give some background about *BERT* and *LSH*. In Chapter 2, literature review is done in the area of code example recommendation and *LSH*. In Chapter 3, the methodology is explained in detail. Chapter 4 defines the parameters for the assessment of our model. In Chapter 5, the results are added, and the research questions are answered. Finally, we discuss the conclusions and the future research directions in Chapter 6.

CHAPTER 1

TECHNICAL BACKGROUND

In this chapter, we present the technical background related to our work. First, we discuss Neural Networks and Transformers that help in understanding *BERT* model. Transformers are a type of neural network architecture, and *BERT* is a Transformer-based model commonly used in natural language processing. We use *BERT* to convert each code example to a numerical vector. Next, we discuss the Locality Sensitive Hashing (*LSH*) which is a technique applied to accomplish efficient approximate nearest neighbor searches in high-dimensional spaces. In our work, *LSH* is used to find the nearest code examples to a query by reducing the search space and then recommending the most similar samples.

1.1 Neural Networks, Transformers and BERT

In this section, we introduce neural networks and transformers to provide context for using the *BERT* model in our research. We begin by discussing neural networks and then introduce the concept of transformers, which forms the basis of the *BERT* model. We then describe the *BERT* model, which uses the *attention mechanism* to extract the semantics of words based on their surrounding words.

Neural network is an artificial intelligence model inspired by the structure and function of brain neurons. It is used to solve machine learning problems, particularly when the relationship between input and output is complex and nonlinear (Zou, Han & So (2008)). Neural networks are often used to learn long text sequences with a large number of tokens, in order to extract semantic information from the tokens and sentences. By analyzing the semantics of the text, a neural network can predict the next word in a sequence and understand the context of a discussion.

However, neural networks have two main problems. The first problem is that they can be slow to train, as they process tokens sequentially and may take a long time to train on a whole text. The second problem is that they can struggle to handle long sequences due to the vanishing effect. This occurs during training when gradient-based methods such as backpropagation are applied, and it results in the initial layers of a neural network being unable to be trained properly, leading to higher error rates and low accuracy in various tasks (De Mulder, Bethard & Moens (2015)).

Transformer is a new architecture that is designed to solve the mentioned problems of Neural Networks. Transformer has encoder and decoder parts. The encoder part extracts information such as features and the decoder part utilizes the extracted features to generate the results such as language translation, question-answering, etc. It acts based on three concepts. These concepts are positional encoding, attention, and self-attention. Positional encoding preserves information about the words' orders in a sentence. Attention means giving weight to more important features in text, images, etc. For instance, the subject may have more importance inside a sentence than other elements. Self-Attention facilitates the understanding of a word based on the surrounding words. For example, the word server may have different meanings in two separate sentences. In one sentence, it may convey the person who serves a service, while in another, it may mean a computer system (Vaswani *et al.* (2017)).

BERT is a transformer-based model that is used for language representation and has the only encoder part of the transformer. This model applies statistics to find the relationship between words and applies positional encoding, attention and self-attention to incorporate the semantic information in a better way. *BERT* is used for tasks such as question answering, next sentence prediction, semantic inference, etc. This approach is beneficial because in contrast to the Neural Network-based approaches such as *Recurrent Neural Networks (RNN)* and *Long Short-Term Memory (LSTM)* that read the words sequentially (*i.e.*, from left to right), it reads and processes the text based on the sentences instead of words. Reading a whole sentence assists to extract the meaning of the words based on the context. *BERT* assists in extracting semantic information from the context that leads to higher accuracy in different tasks (*e.g.*, question-answering) compared to Neural Network approaches (*i.e.*, *RNN* and *LSTM*). The *Attention* in *BERT*, gives more weight to the tokens (in a sentence) that play an integral role based on their importance. Moreover, self-attention is applied in a bidirectional way to extract the information from the left and right sides of the tokens.

BERT algorithm has two main steps, *i.e.*, pre-training and fine-tuning (Devlin *et al.* (2018)). Pre-training means training the algorithm on a large dataset. *BERT* uses BooksCorpus, which contains 800 million words as well as English Wikipedia, which incorporates 2500 Million words. In this step, the texts are selected from documents to extract the semantics in a better way. While fine-tuning occurs when the algorithm is applied to a specific dataset or task, the parameters are initialized with pre-trained parameters. Thus, in this case, the already pre-trained model exists and is applied to a new dataset.

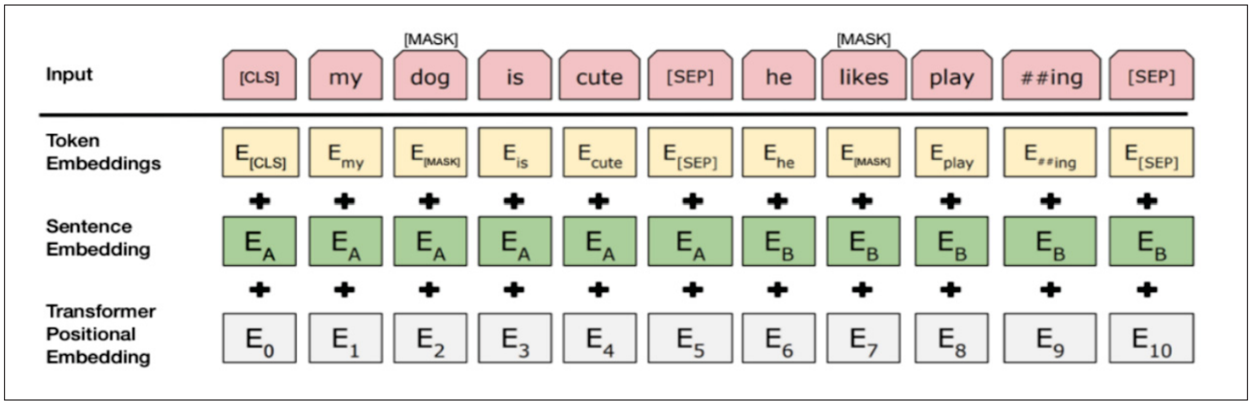

Input
[CLS]
my
[MASK]
dog
is
cute
[SEP]
he
[MASK]
likes
play
##ing
[SEP]
Token Embeddings
$E_{[CLS]}$ E_{my} $E_{[MASK]}$ E_{is} E_{cute} $E_{[SEP]}$ E_{he} $E_{[MASK]}$ E_{play} $E_{\#\#ing}$ $E_{[SEP]}$
Sentence Embedding
E_A E_A E_A E_A E_A E_A E_B E_B E_B E_B E_B
Transformer Positional Embedding
E_0 E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 E_9 E_{10}

Figure 1.1 BERT Embedding
Taken from Devlin *et al.* (2018)

This model applies three-level embedding from the sequences to extract information. The Figure 1.1 demonstrates an instance of embedding. As it can be noticed, *BERT* makes use of these three level embedding to represent the contexts. Token embedding encodes each token and word into numerical values. [CLS] token is inserted at the start of the text, and [SEP] token is used to separate the sentences. Next, sentence embedding is applied to label the tokens based on the sentences they belong. In the last step, positional embedding is added to denote the position of tokens. The main advantage of *BERT* over other models, such as *Word2Vec* is that besides the fixed representation of the words, they incorporate the representation of the surrounding words. Thus, a word might have two different representations in two separate sentences because their surrounding words are different (Devlin *et al.* (2018), Shen & Liu (2021)).

1.2 Locality Sensitive Hashing

Locality Sensitive Hashing (*LSH*) is a dimensionality reduction algorithm that attempts to map similar data points to the same buckets by applying some hash functions. This approach is efficient, especially for problems that have the curse of dimensionality issues. Curse of dimensionality refers to the problems when the dimension of a problem increases, leading to an exponential increase in the time and cost of the solution (Kuo & Sloan (2005)). *LSH* decreases the dimension of a high-dimensionality problem into smaller ones and then puts the similar data samples into the same buckets. Consequently, it ranks the most similar samples by ranking them using some criteria such as Euclidean, Jaccard, and Cosine (Dasgupta, Kumar & Sarlós (2011)). Figure 1.2 demonstrates a simple example of Locality Sensitive Hashing to represent the points in the space by two bits. For this aim, we introduce two hyperplanes (blue and red) for representation of 4 points (A, B, C and D) into 2-bit values:

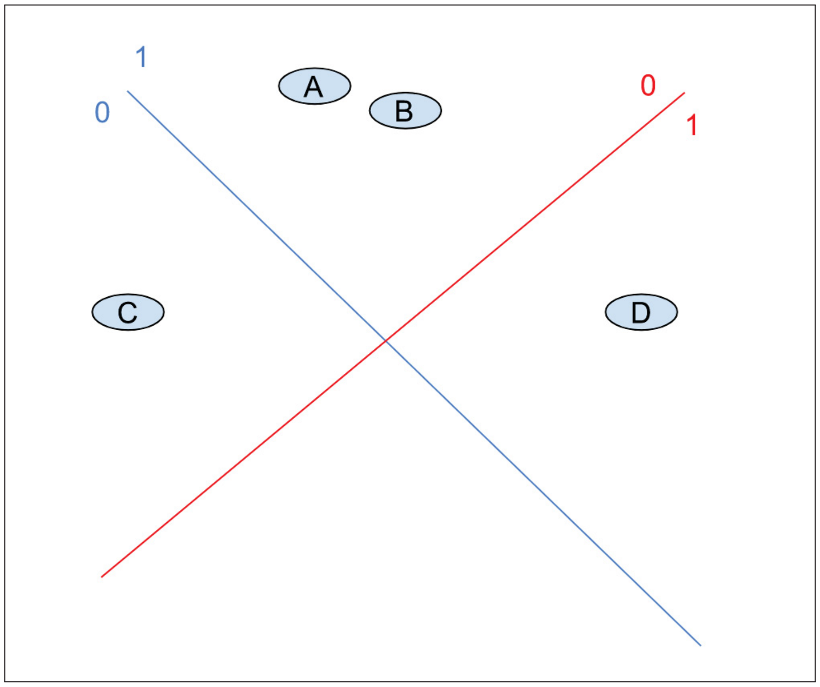


Figure 1.2 An example of two hyperplanes in LSH

Each of these hyperplanes assigns 0 or 1 values for each point based on their positions. Table 1.1 demonstrates the 2 bit values for 4 points. This method of showing points by 2 bits is an example of applying hashing that is mostly used for dimensionality reduction. LSH applies different approaches (like this one) to decrease the dimension of the points to reduce the space and processing time especially when the data size is large.

Table 1.1 Points Hash values

Point	**Red Hyperplane**	**Blue Hyperplane**
A	0	1
B	0	1
C	0	0
D	1	1

In the following, we will discuss a subset of different versions of the LSH algorithm and its variants. Specifically, we will focus on the query-aware and query-oblivious approaches that find the nearest candidates independently from or based on the query values.

1.2.1 Query-aware vs Query-oblivious LSH

In this part, we give some background about two *LSH-based* approaches: *Query-Oblivious* and *Query-Aware*. In this research, we have implemented two algorithms and our first algorithm is categorized as a *Query-Oblivious* approach because data samples are assigned to buckets without considering the query sample. On the contrary, *Query-Aware* approach considers query data when applying Hash mapping. We give a summary of these approaches.

Finding the most similar items is one of the most popular problems that are widely used in different types of data such as text, images, videos, etc. One of the approaches that are extensively used in different applications is the *c-Approximate Nearest Neighbors (c-ANN)*. This approach returns the nearest samples whose distances to the query are smaller than or equal to a threshold value (w) and is a helpful approach for reducing the search space especially when there is a large number of data samples. We explain Query-Oblivious and Query-Aware approaches respectively.

Datar, Immorlica, Indyk & Mirrokni (2004) introduced the Euclidean-based *LSH* (*E2LSH*) approach, which is based on *p-stable* distributions. In this approach, the probability of collision in buckets when data samples are assigned depends on the Euclidean distance between the samples. If the Euclidean distance between two data points is high, the probability of collision is lower. Collision occurs when data samples are placed in the same bucket as the query sample. The basic formula of *LSH* is as follows (Datar *et al.* (2004)):

$$h_{(a,b)} = \lfloor \frac{a.o + b}{w} \rfloor \tag{1.1}$$

In this formula, o is an object and a is a random vector, b denotes a random shift and w is the projection of the result inside the w interval. This interval represents a *LSH* bucket that is used to assign similar data samples into the same category and if a data sample would fit into a bucket interval, it will be assigned to the corresponding bucket. In this approach, query is a data sample like others and its assignment to buckets is the same as other samples. Therefore, as data samples are assigned to different buckets without the consideration of query, this approach is called Query-Oblivious LSH. For instance, if the number of the buckets would be *2n*, then the data samples will be placed into $[-nw, -(n-1)w)$, $[-(n-1)w, -(n-2)w)$, ..., $[(n-2)w, (n-1)w)$, $[(n-1)w, nw)$ intervals based on their Hash values. Although the *Query-Oblivious* approach may have less

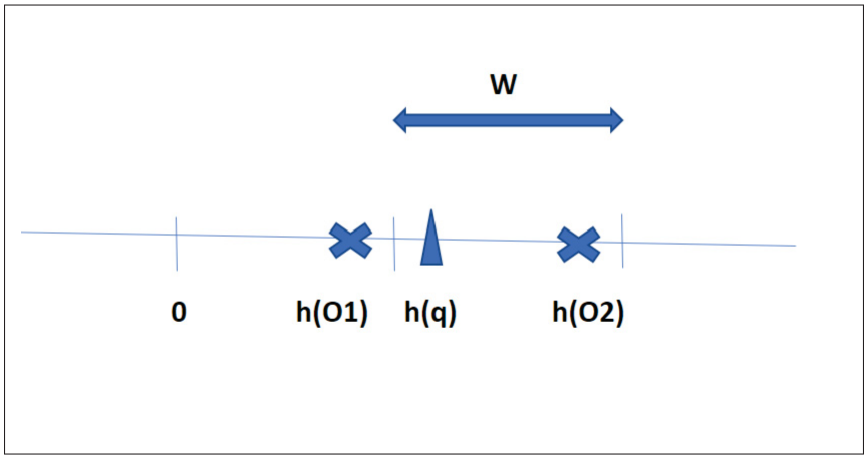


Figure 1.3 Query-Oblivious bucket allocation
Taken from Huang *et al.* (2015)

preprocessing overhead compared with *Query-Aware* approach, similar data samples may be allocated to different buckets which affects negatively its accuracy (Huang *et al.* (2015)). As an illustration, according to Figure 1.3, object O_1 is closer to the query than O_2. However, the hash values of the query and objects puts the O_2 and q at the same bucket. Although O_2 is farther than O_1 to the query (q). O_1 is assigned to a different bucket and it is not the same as the query bucket.

In contrast, the *Query-Aware* approach aims to put the objects based on the query mapping. In this approach, the mapped query vector is assumed as the center of the bucket and other objects are placed into the bucket based on their similarity to the query.

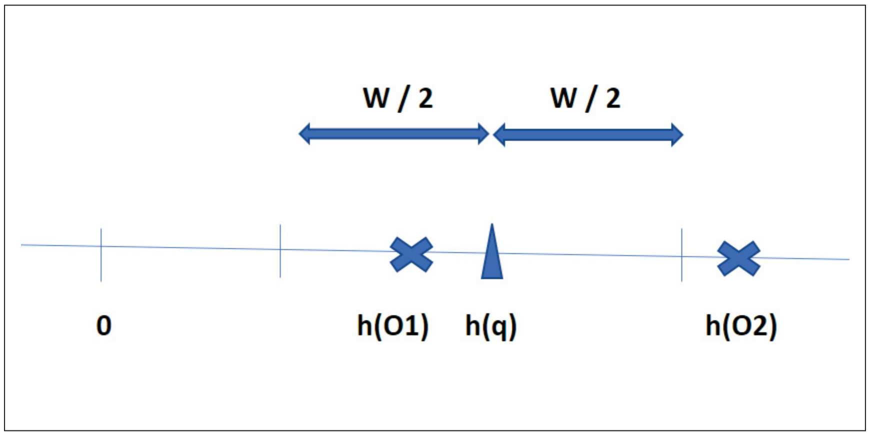


Figure 1.4 Query-Aware bucket allocation
Taken from Huang *et al.* (2015)

In the *Query-Aware* approach, an object o (with dimension N) is mapped to a lower-dimensional space (dimension d where $d < N$) by multiplying a randomized vector (a) and o as follows :

$$h_a(o) = a.o \tag{1.2}$$

Figure 1.4 demonstrates an example of the Query-Aware approach. The difference between this approach and the Query-Oblivious approach is that the center of the comparison is the hash value of the query. This means that objects will be placed at the query buckets if their distance to the hash vector would be less or equal to *w/2*. If this object would be q, then the $h_a(q)$ will be considered as the center of the query bucket to specify the query bucket with length w. So, if the projection of an object (o) is $h_a(o)$, then this object will be at the same bucket with the query if the $|h_a(o) - h_a(q)| <= w/2$.

1.2.2 Cosine Similarity

Cosine similarity (Li & Han (2013)) is used to calculate the angle between two numerical vectors. Text documents are converted into numerical vectors, and their similarity is calculated by measuring the Cosine angle between their vectors. In our work, we have leveraged *BERT* model to convert the text data to numerical vectors and then we have applied the Hashing function to reduce the dimension of the vectors. After finding the similar data samples to our query vectors, we use the Cosine Similarity measure to rank them based on their level of similarity. The highest-ranked samples are considered to be the most similar to the query vectors.

As the query and data samples are numerical vectors, the less the angles are between vectors, the more similarity is there for the vectors. Cosine similarity is measured using the following formula for V and W vectors and we have applied this measure for ranking code examples in both algorithms.

$$Cosine(V, W) = \frac{V.W}{|V| * |W|} = \frac{\sum_{i=1}^{k} V_i * W_i}{\sqrt{\sum_{n=1}^{k} {V_i}^2} * \sqrt{\sum_{n=1}^{k} {W_i}^2}} \tag{1.3}$$

One of the advantages of the Cosine similarity is its measurement based on the angle between vectors not only the size. If two documents are of different sizes and their Euclidean distance is a large value, it is not reasonable to decide their similarity based on their sizes. However, if we apply the Cosine measure to decide to what extent they are similar, the angle between the vectors will be a deciding factor independent of their lengths. For instance, if there are two similar documents with different sizes and one has the word "apple" 10 times and the other 50 times, their differences will be measured based on their angles and not their size. Another advantage of Cosine similarity is its similarity measurement based on the value between [-1,1] since it is based on Cosine value. However, in some other metrics such as Euclidean, there is no maximum and minimum value to measure the similarity.

CHAPTER 2

RELATED WORK

In this chapter, we present a literature review to discuss the research works that are related to the area of the code example recommendation systems. We have divided the related work into different categories which are specified as subsections of this chapter. At first, we discuss the works that have been proposed to extract the code elements from informal documentation and mine patterns of API methods that are used together. Then we present the research works that have focused on the recommendation of code examples based on the usage of API methods. Finally, we discuss the most relevant research works that have leveraged *LSH* when suggesting recommendation systems.

2.1 Code Element Extraction from Informal Documentation

Recent research has leveraged informal documentation to extract code elements. Informal documentation contains different elements, including source code, links, and natural language comments. However, the main problem is the incompleteness of code examples in the forums and discussions, which makes the code element extraction a difficult task (Diamantopoulos & Symeonidis (2015)). Additionally, preprocessing is essential to separate natural language comments from code elements. Stack Overflow is an example of informal documentation that is used widely by developers. This resource is founded by Jeff Atwood and Joel Spolsky in 2008 and is used for exchanging ideas about the technical and conceptual issues that programmers and developers face daily in the computer programming area. In the following, we will discuss several research work that attempted to extract code entities from informal documentation such as Stack Overflow and forums.

InfoZilla (Bettenburg, Premraj, Zimmermann & Kim (2008)) is a tool proposed to extract different elements such as patches, Stack traces, source code, and enumerations from bug reports. This approach uses an island parser (Moonen (2001) and Bacchelli, Cleve, Lanza & Mocci (2011)) to extract code elements. The island name stemmed from the island concept inside the sea, which finds islands based on identifiers. InfoZilla attempts to find classes, conditional statements, functions, and assignments as islands to find code elements from discussions. It starts to traverse the adjacent text of the islands to increase the region of the code part until reaching the area that does not have code elements anymore. Unlike this research, we do not attempt to find the code entities embedded in discussions using an island parser. However, we have extracted code examples form Stack Overflow posts that are surrounded by <the code> tags. This work is done to collect and recommend best code examples based on the developers' requirements.

ACE (Rigby & Robillard (2013)) also uses the notion of the island parser along with naming conventions (*e.g.*, Camel Case) to extract code elements trapped in informal documentation. This approach uses Island parser to find Java code elements inside Stack Overflow posts. ACE attempts to find qualified terms in posts (*i.e.*, package names, variable declarations, qualified variables) and class concepts such as inheritance, constructors, exceptions, etc. It has also applied some heuristics to local and global variables and used the definition of objects to resolve the variables and classes to their types. This approach achieved 0.92 of precision and 0.9 of recall when identifying the code elements from Stack Overflow posts. While ACE focuses on extracting code elements such as API classes and API methods discussed in Stack Overflow posts, our research extracts code examples from Stack Overflow posts to make relevant code examples based on queries, *i.e.*, API methods in our case.

Diamantopoulos & Symeonidis (2015) have proposed an approach to extract code entities from Stack Overflow posts. This approach extracts three types of entities from Stack Overflow, *i.e.*, Java code examples, including Assignments (AM), Function calls (FC), and Class instantiations (CI). It consists of the following steps. First, all the declarations are extracted, such as classes, methods, and variables. Next, a lookup table is created based on the variables and their types. After identifying the entity types, the sequence of these elements is created, and their entity types are added as prefixes to their names. Building these sequences helps find similar code examples, especially when the code examples are incomplete. This approach is able to extract code entities from Stack Overflow code examples and return them only ten percent of the time. This approach cannot, therefore, identify code entities such as classes, variables, and functions almost 90 percent of the time. However, our approach finds similar code examples based on the semantic similarity that was implemented using the *attention mechanism* in *BERT* model instead of code entities (*i.e.*, function calls, class instantiation or assignment). The *attention mechanism* allows us to identify code examples that are related in terms of their underlying meaning and purpose, rather than just the specific code entities they use.

Abdalkareem, Shihab & Rilling (2017) have extracted code examples from Stack Overflow posts by filtering special tags. To extract these code examples, the bodies of Stack Overflow posts are encoded in HTML, and the code examples are embedded in <code> tags, which can be obtained by filtering these tags. Their approach then applied some heuristics to filter out non-code elements. We have used this work's code extraction approach in our own work, as in both cases the code elements were embedded within Stack Overflow posts.

According to this work, small Android projects rely on Stack Overflow posts for development 84.79 percent of the time, while large Android projects rely on posts 12.83 percent of the time. This Approach attempts to discover to what extent the Stack Overflow posts are used by developers for developing Android projects based on the size of their projects. Unlike this approach, our research work tries to extract the code examples from the natural language posts and recommend the relevant candidates of code examples based on the developers' queries.

2.2 Recommendation Systems based on Usage Patterns

Nguyen *et al.* (2019) have suggested FOCUS (API FunctiOn Calls and USage patterns) for the recommendation of a set of methods that are commonly used together by developers when they are using specific APIs in their development projects. These methods are recommended with usage patterns that are used as a reference for completing the developing code. This approach builds upon concepts which are originated from Collaborative-Filtering concept. The fundamental idea of collaborative filtering is to select and recommend items to a user that are chosen by similar users in similar contexts. In this research work, methods and projects are considered as products and customers, and FOCUS attempts to find the patterns of API method usages in regard to a collaborative-filtering recommendation system. This research tries to answer the question of what methods will be called by a part of a client code based on the already called API methods. FOCUS mines open-source software (OSS) repositories to prepare developers with API function calls and usage patterns, it makes a mutual relationship between projects by using a 3D matrix and mining API usage from similar projects. FOCUS has five main steps for making recommendations. First, it gathers different Android projects from OSS repositories such as GitHub. Second, the code parser extracts method declarations and invocations from the source code and encodes the relations in a 3D matrix with projects, declarations, and invocations dimensions. Third, it computes the similarity between projects in repositories and the project under development. In this step, FOCUS uses the Jaccard similarity to compute similarities among method declarations. Fourth, FOCUS generates a ranked list of API function calls. Finally, the recommendation engine in FOCUS generates recommendations either as a ranked list of methods or a usage pattern. In a nutshell, FOCUS uses CF (Collaborative Filtering) technique to recommend and rank API methods and usage patterns from similar projects. Thus, by applying FOCUS, relevant API invocations are recommended, and also code examples are returned to the developers as usage examples. Unlike our work, this approach uses the similarity of API method invocations to find the similar projects and it does not recommend code examples using informal documentation.

Zhong *et al.* (2009) introduced MAPO (Mining API usage Pattern from Open-Source Repositories) to mine patterns and recommend the code examples based on the users' requirements. This work collects the API methods and identifies the conditional statements by extracting different sequences of API methods inside the body of a specific method (m), which helps to find methods that are used together. After acquiring all sequences of API methods, a subset of common sequences is selected for each method (m) and then the algorithm collects all sequences of methods that are called by the method m and creates a graph. Consequently, the algorithm mines the API usage by finding similarities between code sequences and a number of API methods that contain similar patterns. Finally, it recommends similar code examples from open-source projects based on the mined patterns. This approach is applied on a limited number of open-source projects, *i.e.*, 20 projects and also focuses on extracting the frequent patterns that may be useful for developers (Allamanis, Barr, Devanbu & Sutton (2018)). Since MAPO selects open-source projects to mine the API methods, it is easy to extract API methods from these resources. However, eliciting API methods is not easy in informal documentation since the code examples are incomplete. In MAPO, multiple sequences of frequent API methods that are used together, are mined. However, this approach does not recommend code examples of the API methods. Unlike MAPO, our approach recommends code examples of API methods from Stack Overflow, which is in the category of informal documentation.

Wang *et al.* (2013) have proposed Usage-pattern Miner (UP-Miner) that mines frequently used API methods from source code. First, it clusters API methods based on the similarity of sequences. Second, the algorithm uses the BIDE (BI-Directional Extension) algorithm (Wang *et al.* (2004)) to mine frequent closed sequences from the first step. In the next step, it clusters the frequently closed clusters. Finally, the algorithm determines the optimal number of patterns in order to maximize the coverage, *i.e.*, the ability to cover all usage patterns of an API method and succinctness, *i.e.*, the ability to return concise patterns and remove useless parts of the patterns criteria. Like MAPO, UP-Miner is not able to extract API methods in informal documentation to extract the frequent API methods and it is not recommending the usage example of API methods. However, our approach focuses on returning the best code examples for API methods that are extracted from Stack Overflow.

Gu, Zhang, Zhang & Kim (2016) have introduced DeepAPI, which is a deep learning-based approach for generating API usage sequences. In this work, a user gives a natural language query as input and receives an example of the related APIs. It applies a natural language model called RNN Encoder-Decoder, which encodes user queries to a vector and generates the API sequences based on the context vector. This work has used a corpus of seven million annotated code examples from GitHub, meaning a large amount of data is used for training this model. It has also used JavaDoc to annotate between Java APIs and natural language in neural networks. In some cases, JavaDoc has not described the APIs precisely and the extracted sentences as a metadata for neural network, may not be able to describe the exact application of an API. Moreover, DeepAPI has used GitHub and JavaDoc to recommend code examples and has neglected other useful resources from informal documentation such as Stack Overflow. Our approach leverages StackOverflow and uses code examples that were tagged as the answers to questions.

Niu, Keivanloo & Zou (2017) have introduced a network of object usage in which every object usage is a set of methods that have been called inside an API class. This approach uses two clustering approaches. The first one creates different communities based on called objects within different methods and merges those communities if they are highly connected. The second approach clusters object based on the method-call similarity. In this approach, if a subset of methods in object usage O_1 is included in the second object usage O_2, then an arrow will be drawn from O_1 to O_2 and the connected nodes of objects will create a graph for clustering the nodes of objects.

This algorithm employs a hierarchical structure to cluster object usages in order to reduce redundancy. The results of the two above-mentioned approaches, which analyze the objects used within methods and the objects called from other objects, are combined to select representative object usages. This research works extracts the frequent API patterns with less redundancy compared with old frequent mining algorithms, it also extracts the less frequent patterns of API usages that could be useful in some cases. This research is limited to open-source projects. Additionally, the invoked API methods cannot be extracted for incomplete code examples inside informal documentation. Unlike this research work, our approach leverages informal documentation to make code examples recommendations to developers.

2.3 Recommendation Systems based on Related Code Search

In this section, we will review recommendation systems that are based on finding related code examples. Unlike our research work, these works leverage code from open-source projects to extract samples of code.

Raychev, Vechev & Yahav (2014) have suggested an approach for code completion based on the Neural Networks. The main idea of their approach is to apply a natural-language processing model for predicting probabilities of sentences in the context of finding sequences of method invocations for code completion. This work uses static analysis to extract histories of API methods from code examples in GitHub and other repositories. These histories are used as training sequences for a statistical language model. It extracts sequences of methods and indexes into the statistical language model by using N-gram and Recurrent Neural Network (*RNN*) approaches. Unlike this approach, our proposed algorithms are based on the Transformer (*BERT*) instead of Neural Network (*RNN*) and *BERT* is faster than *RNN* in processing sequential data like text (De Mulder *et al.* (2015)) and extracts semantic information in a better way by incorporating Attention and Self-Attention mechanisms (Devlin *et al.* (2018)). While this work has focused on recommending the sequence of invoked API methods from open-source projects, our work focuses on recommending relevant code examples from Stack Overflow based on queries, *i.e.*, API methods.

Thummalapenta & Xie (2008) have introduced an approach to detect hotspots and coldspots in a given framework by mining code examples from open-source repositories. Hotspots are API classes and methods which are frequently used. These parts allow users to reuse and get beneficial information about the given framework. In contrast, coldspots are API classes and methods which are rarely used. As these parts are less used, there are fewer examples of these APIs usages. The proposed approach, called SpotWeb, applies a code search engine (CSE) to collect relevant code examples of classes related to the input framework. CSE extracts code examples from open-source repositories for the input query. This approach could help in the prioritization of bug reports based on hotspots. However, our approach acts based on Stack Overflow code examples tagged in answers of questions by developers, which means the useless code examples have been already filtered out.

Kim *et al.* (2018) have proposed FaCoY(Find a Code Other than Yours) that is a code-to-code recommendation system that finds the code examples that are semantically similar to the query code. This approach extracts syntactical information from the query input and modifies the query code fragments to find the code examples that have the similar functionality. It extracts the contextual information from collected Java files in GitHub and Stack Overflow posts to find more semantic information from the code examples. Specifically, this approach extracts the top 10 questions of the Stack Overflow and evaluates its performance with *Precision@k* parameter. This parameter represents the quality of the returned code examples as the result of the query questions. The acquired values for *Precision@10* and *Precision@20* are 57.69% and 48.82% respectively. One of the drawbacks of this work is its high value of false positive, which means it returns a high number of irrelevant Stack Overflow posts as a result (Kim *et al.* (2018)). In our research, we have used code examples from Stack Overflow posts that were labeled as answer posts to increase the quality of our dataset for code example recommendation. In addition, we applied the *BERT* model to find high-quality code examples based on a query value.

Rubei *et al.* (2020) have introduced PostFinder as a recommendation system for software developers; PostFinder has been designed as a plugin in Eclipse IDE that extracts the contextual information from the developing project such as Method declaration, Method invocation, variable type, etc. On the other side, it extracts the contextual information from the Stack Overflow posts such as the title, question and answer, as well as code examples discussed in those posts. It applies code wrapping on Stack Overflow code examples to find the Fully Qualified Name (*FQN*) of libraries and API classes[1]. Then, it attempts to find the similarity between extracted entities from the developing project and processed Stack Overflow posts and recommends the top N Stack Overflow posts based on their similarities. PostFinder is evaluated using three parameters that are *Success Rate, Precision* and *Relevance* and their values are 0.95, 0.66 and 2.78 respectively for a set of 50 queries. Unlike this work, our research work focuses on code examples recommendations instead of Stack Overflow posts recommendation. Additionally, we apply a hybrid approach (*BERT* + *LSH*) instead of focusing on code element extraction, *i.e.*, extraction of Methods and Variables for recommending relevant code examples.

Lancer (Zhou *et al.* (2019)) is a code-to-code recommendation system that uses the *BERT* model to solve *OOV* (Out Of Vocabulary) problem. *OOV* occurs when terms are not a meaningful

[1] https://zenodo.org/record/1489120.Y33fMEnMLEY

part of Natural Language and it usually happens when they are misspelled or a combination of multiple terms. Lancer is a Library-sensitive approach that attempts to find the code examples based on the Java library tokens. It applies the *PageRank* algorithm to find the relevant libraries which are used together. According to Lancer, the names of variables in code can demonstrate the functionality of the variables. For such a purpose, it has used *BERT* to rank similar code examples based on the semantics of the tokens. In *BERT* algorithm, the word Tokens are separated by camel names, for example, "*OutputStream*" is divided into "Output" and "Streams", which could assist to extract the semantics of the code. Lancer has been applied for Java open-source projects and not informal documentation. In addition, it needs to extract the library names and Fully Qualified Names (FQN) of API methods. Unfortunately, the code examples that exist on Stack Overflow are mostly incomplete and the extraction of FQNs and the Abstract Syntax Tree is difficult. Moreover, this approach has been evaluated based on two parameters: *HitRate* and *MRR* (Mean Reciprocal Rank). In this work, a query is a sequence of tokens that are given as input to Lancer and HitRate@k denotes if the first *k* returned results, *i.e.*, the top *k* recommendation candidates include the relevant code examples to a query or not. If the top *k* returned results contain at least one relevant item, then the *HitRate@K* value is 1, otherwise it is 0. Additionally, *MRR* is the mean value of the inverse of the first rank of the recommended items when *HitRate* occurs. The obtained *HitRate* values for this approach are 0.454, 0.595, 0.629 for *HitRate@1*, *HitRate@5*, *HitRate@10*; The obtained value for *MRR* is 0.515. Unlike Lancer, which uses open-source projects, our focus is on code example recommendations from Stack Overflow. We also apply a hybrid approach of *BERT* and *LSH* algorithms instead *PageRank* and *BERT*. While our approach is different from this work, our evaluation leverages the same metrics, namely the *HitRate* parameter to evaluate our recommendations of code examples.

2.4 Locality Sensitive Hashing for Recommendation Systems

Ding, Fung & Charland (2016) have combined LSH and graph search to find similar assembly code examples. Their approach involves using a hybrid approach based on the assembly code sources, particularly when the main source code is unavailable. This approach, called Adaptive

LSH (*ALSH*), involves searching through a tree structure. When a query (q) is given, the leaf node is found in different prefix trees. Once enough points have been discovered, the subtrees are split and the search moves up a level, reducing the search space by filtering out the least similar samples. This approach is classified as an *LSH-Forest* (Bawa, Condie & Ganesan (2005)), which uses a tree structure to reduce the search space based on the query value.

However, in our second algorithm (*Query-Aware LSH*), we use a type of LSH that is under the category of *C2LSH* (Collision Counting LSH) meaning if a data sample (d) collides with query (q) more than a pre-defined threshold time (L) in different Hash tables, then that data sample is considered as a similar data sample to query (Gan, Feng, Fang & Ng (2012)). Hence, the difference between ALSH-based work and ours is that this research work applies a type of *LSH-Forest* to decrease the search space while ours is a version of *C2LSH*.

Silavong, Moran, Georgiadis, Saphal & Otter (2022) have proposed a code-to-code recommendation system based on the Locality Sensitive Hashing. This research extracts *AST* of code examples by *ANTLR* (ANother Tool for Language Recognition) and then extracts *SPT* (Simplified Parse Tree) of the generated structure. This is a query-based approach that applies *Minwise Hashing* based on the four extracted feature sets, which are *Token, Parent, Sibling,* and *Variable usages*. Then, it converts the sets into signatures to measure the sets based on the Jaccard similarity. Different k independent hash functions are applied on each set and make a vector to create a signature $[\mathrm{h}^1_{min}(s), h^2_{min}(s), ..., h^k_{min}(s)]$. Next, the algorithm categorized the signatures into B bands, each having the size R. Then, the samples at the same band of the query vector are selected as similar candidates and ranked based on their dot product values with the query *(q)* signature. In our research work, *ANTLR* and *SPT* could not be applied to extract code elements because the code examples are extracted from informal documentation and they are mostly incomplete. Additionally, our *LSH-based* algorithms are based on *SimHash*, while this research work is based on *MinWise* Hashing.

Zhang, Fan & Wang (2018) have proposed a recommendation system by applying *LSH* and *CF* (Collaborative Filtering) concurrently. They have used a hybrid approach to tackle time, space, and accuracy. This approach applied *minHash* and *SimHash* to create a signature matrix and then assigned signatures to buckets to put the most similar items in the same buckets.

Aytekin & Aytekin (2019) have introduced a new version of LSH for making a recommendation system from a large amount of data, while maintaining high accuracy. Their approach have performed faster than the standard LSH and recommended more diverse candidates.

CHAPTER 3

METHODOLOGY

In this chapter, we describe the steps of our methodology in more detail. Our methodology consists of five main steps: Data collection, Data pre-processing, Extraction of code snippets, applying *BERT* and reducing the search space using *LSH*. For the *LSH* part, we have proposed two algorithms called *Random Hyperplane-based (RH)* and *Query-Aware (QA)*. Finally, we have evaluated our algorithms by two experienced software developers to provide feedback on the recommendation items based on the tasks they were working on.

3.1 Overview of the Followed Methodology

We have six main steps for our proposed methodology. First, we have collected data by downloading the Stack Overflow dataset. Then, we have pre-processed it by filtering Java posts. In the next step, we have extracted the embedded code examples from posts. After cleaning the data, the obtained outcome is given to the BERT model. Next, the LSH algorithm is applied to find and recommend code examples that correspond to a query. Finally, the model will be assessed using proper parameters. The main steps of our implementation are as follows (Figure 3.1):

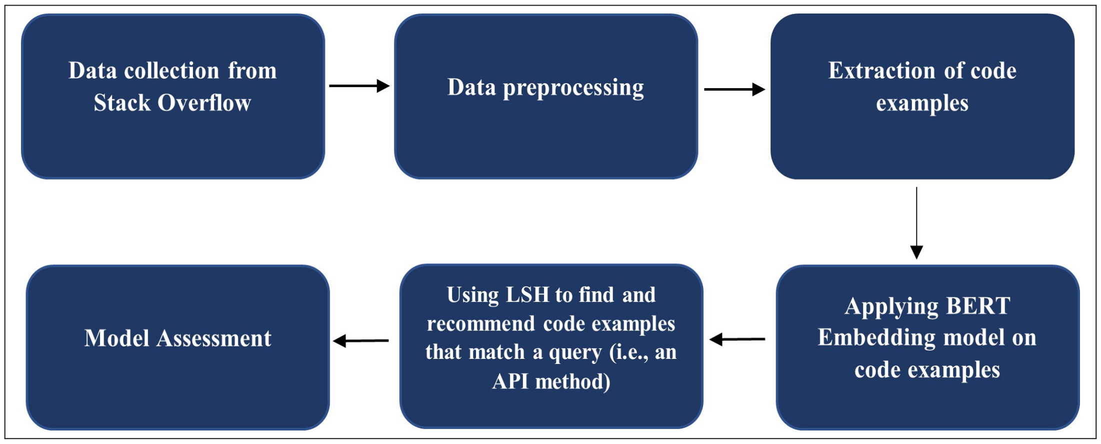

Data collection from Stack Overflow
Data preprocessing
Extraction of code examples
Applying BERT Embedding model on code examples
Using LSH to find and recommend code examples that match a query (i.e., an API method)
Model Assessment

Figure 3.1 The main steps of the followed Methodology

3.1.1 Collecting Data

We have selected Stack Overflow posts as our input dataset, one of the most popular sources for finding discussions about different API methods from different programming languages. In our work, we have downloaded Stack Overflow dump files[1], which are available on the internet. These dump files include all the discussions that have been submitted between 2008 to 2022. Like Kim *et al.* (2018), we have downloaded the dump files that are archived from Stack Overflow posts. To retrieve the code examples from the posts, we needed to import these dump files into a database and create an SQL Server database. Each row of the imported database incorporates different elements of a post, such as a Title, Post, Scores, etc. The Post field consists of the natural language texts and the code examples are embedded into the Post field. We have extracted these elements by applying Abdalkareem *et al.* (2017) approach for code example extraction from Stack Overflow posts that we will discuss in this chapter.

3.1.2 Pre-processing Data

After the importation of the dataset into an SQL database for organization purposes, we have cleaned the data. For such a purpose, we did a preprocessing on the data to make it ready for the next step. We have followed four steps to accomplish data preprocessing.

Step 1: We executed an SQL query to select posts that were labeled as Java posts but did not include other tags such as Javascript and C. During our experiment, we noticed that many posts labeled as Java posts contained code segments in Javascript and C. Therefore, we filtered out these samples from our dataset because we believed they would result in unrelated recommendation items.

Step 2: In our database, each row shows a post that encompasses 20 columns demonstrating different elements of a post. We have worked with *Id, AcceptedAnswerId, and Score* columns to filter the posts that are needed for our study. *Id* is the identifier of a post and *AcceptedAnswerId*

[1] https://archive.org/details/stackexchange

denotes the *Id* of a post that is labeled as the answer to the question post. Most of the Question posts have Answer posts, and if they have answer posts, then the answer posts' *Id* is identified as *AcceptedAnswerId*. Therefore, the valid *AcceptedAnswerId* values are saved in a table of the database for the next step.

Step 3: In this step, the answer posts are extracted from the SQL database by filtering the posts whose *Id* is equal to *AcceptedAnswerId*. Some Stack Overflow questions do not have answers, so the *AcceptedAnswerId* field is empty on these rows.

Step 4: Finally, our dataset includes the Answer posts which are labeled by developers, and the posts' scores which show the degree to which the posts are liked by other users. There are also some posts whose scores are 0 or a negative value and users did not have a positive opinion about them. We will explain more about them in the next section.

3.1.3 Extracting Code Examples from Informal Documentation

After the extraction of the answer posts, we needed to extract code examples because they are mixed with natural language comments. To this aim, we have extracted the code examples from the posts by filtering the code between <pre><code> and </code></pre> tags. We have mirrored, in this step, the work by Abdalkareem *et al.* (2017) that have suggested this method for code example extraction from Stack Overflow posts. We have extracted posts with Java tags since our focus in this research is on recommending code examples for Java API methods. After the extraction of code examples from posts, we have noticed that there are a lot of examples that were tagged as Java. However, they incorporate Javascript and C# code examples. We have therefore filtered posts that include Java tags and do not contain the mentioned tags. By applying this filter, the number of returned results was 377 517 code examples. In the next step, we have observed that there were some results that were not code examples and mostly comments and sentences. After removing these posts that are incorporating the <code> tags, the number of the results has been reduced to 198 911. Next, we have extracted code examples from posts and put the limitation of at least 100 characters on them because there were around 70K results that were labelled as code examples. Yet, they were short comments, descriptions and commands and they were not part of the code elements. By applying this limitation, the number of retrieved results (*i.e.*, code examples) decreased to 128 688. Moreover, we have observed that all of the extracted elements between the mentioned tags are not code examples and they are elements such as StackTraces, commands, SQL queries, and links. We have also filtered out the code examples which were starting with special characters such as {#, /, , @}. By filtering out these elements, the number of the filtered code examples was reduced to 97 084. Finally, some of the retrieved examples had low scores and they were not considered useful code examples from the developers' perspectives because their scores were equal to zero. In other cases, their scores were negative and the users did not consider them as useful examples. We set the threshold value to two and we achieved it empirically. When we set the threshold value to three, more than 6K useful code examples were filtered out. Additionally, by setting the threshold value to one, around 8K unsuitable code examples are added to our dataset that deteriorated the quality of the code examples' recommendation. By selecting this threshold value, the number of retrieved code examples was reduced to 61 361. Table 3.1 demonstrates the filtering process performed on the Stack Overflow dataset with the number of results in each step.

Table 3.1 The filtering steps of Stack Overflow posts

Followed Steps	**#Posts**
Filtered posts based on Java tags and not other programming languages tags	377 517
Posts incorporating <code> tag	198 911
Code examples with more than 100 characters	128 688
Filtering non-code examples	97 084
Code examples with score greater than or equal to 2	61 361

3.1.4 Applying the BERT model

To apply the *BERT* model on our cleaned code examples, we have used Python's sentence_-transformers. This framework is used to represent text and images using numerical dense vectors. In our work, we have used the pre-trained BERT model, which is available in over 100 languages and has been made publicly available. We have used this model to fine-tune based on our dataset.

Each sample of the code example is given as input to BERT, which incorporates token-based and semantic-based information into a constant-size (1*768) vector.

Figure 3.2 demonstrates the overall architecture of *BERT* model[2]. According to this figure, *BERT* has 12 encoder layers and each hidden vector's dimension is (1*768). Each layer imposes self-attention and forwards the result to next layer. Self-attention attempts to connect different tokens in different positions inside a sequence (*i.e.*, code example in our case) to create a numerical representation of that code example.

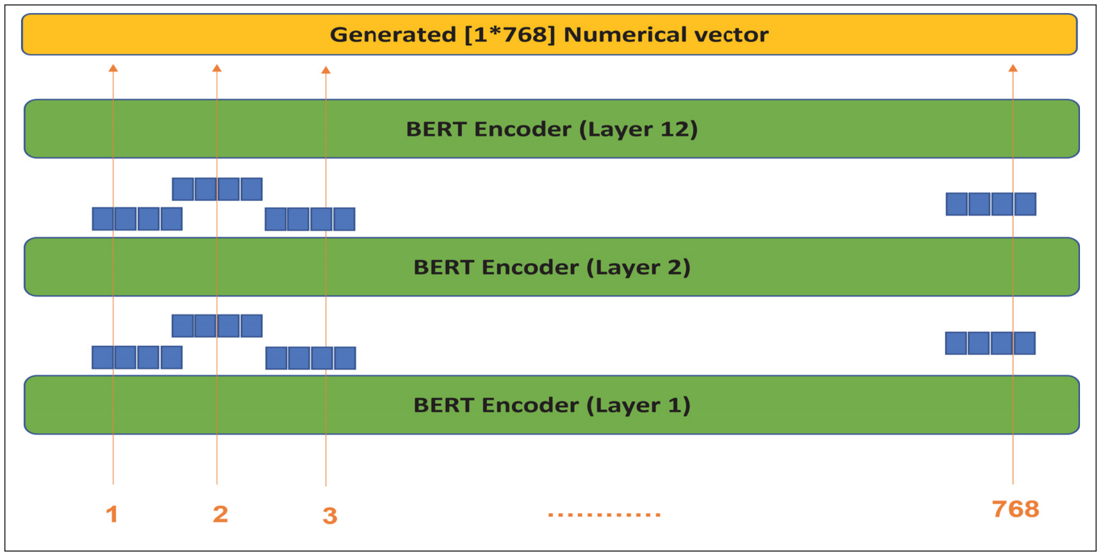


Figure 3.2 The overall structure of *BERT*

At this stage, we have provided our cleaned dataset of code examples as input to the *BERT* model. Each code example is a data sample, and the special characters are filtered out from each of them. This model embeds each sample into a (1*768) numerical vector. The *BERT* model incorporates the contextual information of these code examples in three levels, *i.e.*, token, sentence, and positional-based levels. By having this type of embedding, both the textual and contextual information are included in numerical vectors. In the following, we will explain the steps of the *BERT* model in more detail.

First, the words inside each code example are tokenized and mapped to their indices in the

[2] https://www.geeksforgeeks.org/explanation-of-bert-model-nlp/

BERT vocabulary. *BERT* creates this vocabulary in a pre-training step on input corpora and text, breaking down the words into subwords and characters and adding them to the vocabulary. For example, the *BERT* model breaks down the word "Embedding" into "Em", "##bed" and "##ding" and adds them to the vocabulary, with "##" indicating that the subword is part of a larger word. We use a pre-trained *BERT* model whose vocabulary has already been created, so the tokens of the code examples are mapped to this existing vocabulary.

Second, the segment ID is assigned to the tokens. In our problem, we consider that all of the tokens inside each sample, *i.e.*, code example belong to and comprise a data sample. Third, the tokens are converted to tensors and then provided to the BERT model. Tensors are multi-dimensional numerical arrays that have the same data types and where each token is converted to a (1*768) tensor (*i.e.*, Numerical vectors). In our study, We have used the Bert$_{Base}$, which is a standard version of BERT model that has 12 layers. For each token, there are 12 different vectors (*i.e.*, the number of layers), and the dimension of each token is (1*768). The generated vector of each data sample (*i.e.*, code example) is (1*768) as well. Hence, the parameters' values of the applied *BERT* are the same as the standard Bert$_{Base}$. One of the main advantages of BERT over the other token-based approaches is its ability to extract semantic information from token sequences Devlin *et al.* (2018).

3.1.5 Locality Sensitive Hashing (LSH) vs Clustering

After embedding the samples and a query in the *BERT* model, we have implemented two *LSH*-based algorithms to find and recommend the most similar data samples to the query. The algorithms have to include the most similar samples in the same categories based on the query vector. In our problem, a query is a natural language question like *'How to add an image to a JPanel?'* or its related API method like *'Jpanel.add()'* that is given as input to *BERT* model and it creates a numerical vector as output. Therefore, *BERT* creates a vector for each code example and a query.

Clustering algorithms are widely used to solve various problems such as finding the most similar data samples and creating recommendation systems. There are various types of clustering that can be applied depending on the nature of the problem. However, clustering algorithms have some drawbacks, such as high time complexity and poor performance when dealing with a large number of data samples (Xu & Tian (2015)). In our problem, we have a query vector and over 60,000 code examples that our approach must filter and process to make relevant recommendations of code examples. As mentioned, applying clustering algorithms is not an appropriate solution for our problem because our dataset is large and we need to use approaches that can reduce the search space at the initial stages and find similar data samples in a much shorter time compared to clustering approaches. To address this challenge, we have leveraged appropriate solutions, *i.e.*, Locality Sensitive Hashing (*LSH*). *LSH*-based algorithms require less time and memory to find similar data samples as recommendations because they reduce the search space and find approximate results instead of exact results (Jafari, Maurya, Nagarkar, Islam & Crushev (2021)).

3.1.6 Applying *LSH* Algorithms

We have applied two algorithms of *LSH* to our problem: The *Random Hyperplane-based approach* and the *Query-aware* one. While *Query-aware* considers the query when it assigns data samples into different buckets, the *Random Hyperplane-based* approach does not consider the query, while assigning data samples into different buckets. We have applied both approaches in our methodology and compared their results since based on Huang *et al.* (2015), Query-Aware (*e.g.*, the second algorithm) approach outperforms Query-Oblivious approaches (*e.g.*, the first algorithm) in returning more similar data samples to a query. We will elaborate on the two above-mentioned LSH approaches in the following sections.

3.1.6.1 Random Hyperplane-based LSH

This approach makes use of a random hyperplane with dimension *d* that obeys Gaussian distribution. It computes the Hash value based on the following formula (Charikar (2002)).

$$h_r(u) = \begin{cases} 1, & \text{if } r.u \geq 0 \\ 0, & \text{if } r.u < 0 \end{cases} \tag{3.1}$$

And for vector *u* and *v*, the probability of putting u and v at the same bucket is as follows ((Charikar (2002)), Shrivastava & Li (2014)).

$$Pr(h_r(u) = h_r(v)) = 1 - \frac{\theta(u,v)}{\pi} \tag{3.2}$$

$$theta(u,v) = cos^{-1}(\frac{|u \cap v|}{\sqrt{|u|.|v|}}) \tag{3.3}$$

By applying random hyperplanes, each dimension is chosen from normally distributed variables. In this case, if the number of vectors would be *n*, then the number of the random bits will be O(($\log_2 n$)) and if two data samples have a high degree of similarity in a binary vectorization, then the probability of assigning two vectors to the same hash values increases.

LSH is known to be applied for the reason of curse of dimensionality and for problems whose dimension is not low (>= 10), it is also used for finding approximate results instead of exact results (Jafari *et al.* (2021)). This approach attempts to apply a tradeoff between *accuracy* and *performance*. *Accuracy* refers to how one algorithm returns the most relevant and similar items to a query, while *performance* means how much time it takes for an algorithm to return the results based on the query. In effect, instead of getting a high accuracy for finding duplicate and exact results, similar results are achieved faster and in a shorter time. *LSH* attempts to find the nearest points in the neighborhood of the query by searching for the points in the area of *c * R (c >1)*, with *R* demonstrating the nearest point to the query point (Jafari *et al.* (2021)).

In our problem, the search space is large (60K data samples), and applying clustering algorithms may take a considerable amount of time. In addition, there is no guarantee that there would be exact results for the query, and finding similar results could satisfy the developers' requirements. Moreover, by applying *LSH*, the search space gets smaller, and the recommendation system could return the results faster without further pre-processing, such as making different clusters. We have applied this approach to assign similar data samples, *i.e.*, code examples into the same buckets. Charikar (2002) have applied the Rounding techniques to map similar data samples into the same buckets. Based on our problem, we have followed the following steps. From the previous step, each code snippet (*i.e.*, data sample) and the query text are converted into (1*768) numerical vectors, and for N data samples, the dimension of the vector is (N*768). We have

Algorithm 3.1 Random Hyperplane based LSH
Adapted from Charikar (2002)

```
For $D$ as Data samples, $R$ as a Randomized vector and $M$ as the number of hash tables;
for $i \leftarrow 1$ to $M$ do
    For Data Samples:
    $Result_{N*K} \leftarrow D_{N*768} * R_{768*K}$;
    $SgnResult \leftarrow Sign(Result_{N*K})$;
    if the items of the SgnResult are negative assign 0, otherwise they are already 1;
    $BucketId \leftarrow \sum_{i=0}^{k-1} 2^i * SgnResult[i]$;
    For Query Vector:
    For Q as Query vector and R as a Randmized vector;
    $QResult_{N*K} \leftarrow Q_{1*768} * R_{768*K}$;
    $SgnQRes \leftarrow Sign(QResult_{N*K})$;
    if the items of the SgnQRes are negative assign 0, otherwise they are already 1;
    $QBucketId \leftarrow \sum_{i=0}^{k-1} 2^i * SgnQRes[i]$;
end for
Retrieve the samples (u) that are at the same bucket as query vector (q) from all of the
  hash tables and rank them based on Cosine similarity; Retrieve top N similar samples
  as recommendation items;
```

applied Algorithm 3.1 (Charikar (2002)) to recommend code examples for API methods. This approach is applied for dimensionality reduction of the numerical vectors from a high to lower dimensions. We have converted the obtained numbers from decimal values to binary values and created buckets based on their binary values. We have used this approach to find similar items to a given query (either a natural language question or an API method) in a lower-dimensional space, using both data samples (code examples) and query vectors.

In the following, we explain the *Random Hyperplane-based LSH* algorithm step by step based on the Algorithm 3.1. In this algorithm, if the data samples' vector would be *D*, the randomized vector be *R*, and the number of hash tables would be *M*. Then, based on the value of *M*, each time the data samples are multiplied to generated random vectors. Hence, the dimension of data samples is reduced by mapping them to a lower dimension (*i.e.*, from 768 to k) (step 4). Then the items of obtained results are replaced with binary values (1 and 0) if the items' values are greater or less than 0 (steps 5 and 6). In the next step, each data sample is assigned to a bucket based on its binary values. In each hash table, the number of the buckets is 2^k buckets and data samples are assigned to the buckets based on their binary values (*i.e.*, *BucketId*) (step 7). These steps are applied to the query vector and assigned it to a bucket (steps 9 to 13). Finally, all of the data samples whose *BucketId* is the same as the query vector are collected from all of the hash tables. Then for each data sample, the cosine similarity between them and the query is calculated and ranked based on their similarity values (From the top to down) (step 15). Finally, based on the number of recommendations, the top N of them are returned to the user (step 16). In Algorithm 3.1, we have used random vectors to map the data samples from a higher dimension to a lower one (*d*), and for this purpose, we have created random vectors with dimension of (768 * *d*) that obey Gaussian distribution as in Charikar (2002), who have applied Gaussian distribution to create random vectors to map the data samples into *d*-dimensional vectors.

3.1.6.2 Query-aware approach

Query-oblivious approaches assign data samples without considering the query, *i.e.*, similar data samples may be filtered out from the candidate samples for the query. To tackle this problem, Huang *et al.* (2015) have introduced the Query-Aware LSH, in which a query is the deciding factor for assigning the objects (data samples) to the query bucket partition. Similar to the *Random Hyperplane-based* algorithm, in *Query-Aware* algorithm, data samples are code examples and a query is a natural language question or an API method. Query-aware approach

Algorithm 3.2 Query-Aware LSH
Adapted from Huang *et al.* (2015)

```
1  For $D$ as Data samples and $R$ as a Randomized vector, $M$ as the number of hash tables, $l$
     as the threshold of the occurrence of a data sample ;
2  for $i \leftarrow 1$ to $M$ do
3      Impose Hash functions by multiplying objects ($O_i$) and query vector to Randomized
         vectors ;
4      Increase #Col($O_i$) if the $|H_i(O_i) - H_i(q)| <= w/2$;
5      if $\#Col(O_i) \geq l$ then
6          $C = C \cup O_i$;
7      end if
8  end for
9  Calculate the Euclidean distance between $O_i$ in C and q;
10 Sort the Euclidean distances incrementally ;
11 Retrieve top N candidates as recommendation items from the sorted list
```

solves the problem of random shift that is applied during mapping data samples to a lower dimension for assigning the samples to buckets in traditional *Query-Oblivious LSH* algorithms. For instance, *E2LSH* (Andoni (2004)) applies ($\lfloor \frac{a.o+b}{w} \rfloor$) random shift ($b$) after mapping the data sample into a lower dimension space. Hence, this random shift may affect negatively the process of finding the most similar data samples and putting them in different buckets other than the query. Query-Aware also simplifies the computation by removing the random shifting (Huang *et al.* (2015)).

According to Algorithm 3.2, data samples are represented by *D*, while *R* is the randomized vector, *M* is the number of hash tables and *l* is the threshold value that specifies the number of the times that the Euclidean distance between a hash vector of a data sample and the query's hash vector is less or equal to *w/2*. According to this algorithm, based on the number of hash tables, Random vectors are created. Inside each hash table, data samples are multiplied by their random vector, and their hash values are generated (step 3). Then the Euclidean distance of each data sample's hash value is measured with the hash value of the query sample and if the distance value would be less than equal to *w/2*, then the occurrence number of these samples is increased by 1 (step 4). If the occurrence number of a data sample (*i.e.*, code example) is greater than or equal to l (threshold), then it will be added to the candidate set (*C*) (steps 5 to 7). After collecting the candidate samples, their Euclidean distance with the query vector is calculated and ranked incrementally (steps 9 to 10). Finally, the top N samples will be recommended as the most similar code examples to the query (step 11). This approach filters the data samples based on a given query in two steps. Firstly, it considers samples based on their hash value similarity to the query's hash value. After that, the filtered samples, from the first step, are selected based on their similarity to the query.

In our implementation, we set the occurrence threshold value to two, meaning that if a data sample falls into the same bucket as the query vector at least two times, it will be added to the candidate set. The candidate set consists of data samples that have fallen into the same bucket as the query at least *l* times (the threshold value). We set the threshold value to two empirically, by observing that if the threshold value was higher than two, some useful code examples were filtered out, even though their occurrence frequency was less than the threshold value. On the other hand, when we set the threshold value to one, the search space was not reduced efficiently enough to filter out irrelevant data samples, and the algorithm was unable to retrieve pertinent code examples in the top 30 candidates.

Moreover, since the number of data samples (*i.e.*, code examples) is high (60K) and the number of buckets is rather small (Maximum 50), there is no limitation on the capacity of the buckets. We have also attempted to increase the collision chance of the most similar data samples by increasing the number of hash tables. Based on our experiments, when the number of hash tables increases, the number of data samples at the same buckets with the query sample rises. In addition, the probability of finding similar data samples rises because the number of collided samples surges. As the collided data samples increases, finding more similar data samples is more probable.

3.1.7 Model Assessment

After implementing our *LSH-based* approaches, namely the *Random Hyperplane-based* and the *Query-Aware* approaches, we needed to assess how well our algorithms work. For this purpose, and for comparison purposes as well, we evaluated our algorithms using four parameters: *HitRate, Mean Reciprocal Rank (MRR), Average Execution time,* and *Relevance*. The *HitRate* and *MRR* parameters were adopted from Lancer's research (Zhou *et al.* (2019)), while the *Relevance* parameter was used from PostFinder (Rubei *et al.* (2020)). We measured these parameters for both examined algorithms based on the feedback of two experienced software developers, who were asked to give feedback about the recommended code examples.

CHAPTER 4

EMPIRICAL EVALUATION

In this section, we evaluate our methodology using the Basili framework (Basili, Selby & Hutchens (1986)), which consists of four steps: definition, planning, operation, and interpretation. In addition, we have used the guidelines provided by Wohlin (Wohlin *et al.* (2012)) for conducting experiments based on the Basili framework to evaluate our recommendation system. We have also defined the evaluation parameters and research questions that will be answered based on the defined parameters and proposed algorithms.

4.1 Definition and planning of the study

The goal of this study is to evaluate our proposed algorithms and find if the recommendation approaches return the relevant code examples from Stack Overflow or not.
The quality focus measures are the different metrics used to measure the peformance of the considered algorithms, *i.e.*, *HitRate*, *Mean Reciprocal Rank (MRR)*, *Average Execution Time* and *Relevance* for the two implemented LSH-based algorithms.
The perspective is of developers and researchers interested in recommendation systems that suggest code examples using informal documentation to help guide developers during their engineering tasks.
The context consists of the Stack Overflow code examples mentioned by programmers and software developers in Stack Overflow posts. The used dataset incorporates the Stack Overflow posts submitted between 2008 to 2022.

4.2 Research questions

To evaluate our proposed approaches, we investigate two main research questions that could compare how the proposed algorithms work in recommending relevant code examples. Hence, we will address the following research questions.

RQ1: How do the *Random Hyperplane-based LSH* and the *Query-Aware LSH* approaches perform when recommending code examples, in terms of *HitRate, Mean Reciprocal Rank, Average execution time*, and *Relevance* ?

The first research question has been divided into four sub-questions:

- ***RQ11***: *How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of HitRate?*
- ***RQ12***: *How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Mean Reciprocal Rank?*
- ***RQ13***: *How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Average execution time?*
- ***RQ14***: *How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Relevance?*

RQ2 : *Is there any difference between the Random Hyperplane-based LSH approach and the Query-Aware LSH one in terms of Relevance?*

We have elaborated on the second research question by formulating a null and alternative hypothesis, as follows:

- H_{02} : There is no statistically significant difference between the *Query-Aware LSH* and the*Random Hyperplane-based LSH* in terms of *Relevance*.
- H_{a2} : There is a statistically significant difference between the *Query-Aware LSH* and the *Random Hyperplane-based LSH* in terms of *Relevance*.

4.3 Variable section

In this section, we present and explain the metrics that we have leveraged for our evaluation.

The main independent variable of our study is the kind of approach being applied. There are two different values for this factor:

- *Random Hyperplane-based LSH approach*

- *Query-Aware LSH approach*

The dependent variables considered in our study are *HitRate*, *MRR*, *Average execution time* and *Relevance*. In the following, these metrics are as follows.

HitRate: If we have a set of queries (Q), then *HitRate@k* measures the percentage of the queries that have returned at least one relevant result among the top k recommended items. The following formula defines this parameter (Zhou *et al.* (2019)) as follows:

$$HitRate@K = \frac{1}{|Q|} \sum_{q \in Q} H(R(Q), k) \tag{4.1}$$

According to this equation, Q shows a set of queries and function *H(R(Q), k)* returns one if there is at least one relevant result among the top k recommended items. Otherwise, it returns 0. *HitRate* is the average of the 0 and 1 values, and if the value of this parameter is close to 1, then it shows that a recommendation system is successful.

Mean Reciprocal Rank (MRR): This variable is calculated as the mean of the inverse of the first rank of the recommended items when the *HitRate* occurs. For instance, if two queries have been executed and the *HitRate* of returned results for each of them are at the second and third ranks, then *MRR* is the average of the (1/2 + 1/3). The following formula defines the *MRR* metric (Zhou *et al.* (2019)):

$$MRR = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{\textit{First rank of the relevant result}} \tag{4.2}$$

Average Execution Time: We measure the execution time of our approach in two stages. In the first stage, we measure the time it takes to apply hashing algorithms to data samples. In the second stage, we measure the time it takes to return the results when the hashing has already been applied. We evaluate these measures based on the number of hash tables used.

Relevance: This metric shows the score that is given by developers for the recommended items, *i.e.*, code examples in our research. The scores are defined based on the following table (Rubei *et al.* (2020)):

Table 4.1 The scoring scale of the Relevance metric

Score	Description
0	There is no returned result
1	The returned result is irrelevant
2	There are some hints but still out of the context
3	The results contain some relevant results but not key features
4	The returned results are in the context of the query and they are useful

The top 10, 20 and 30 recommended code examples will be evaluated based on the scores for a collection of query questions, and will be compared for two algorithms, *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH*.

4.4 Analysis Method

For the sake of evaluating the proposed approaches for code examples recommendation, we have applied descriptive statistics and statistical analyses to decide how these algorithms perform. Boxplot (Frigge, Hoaglin & Iglewicz (1989)) is a kind of chart used for showing the distribution of numerical data and it is used for descriptive analysis. There are five identifiers for this chart: minimum, lower quartile, upper quartile, median and maximum.
To compare metrics with numerical values such as the ones of relevance, we have applied the Wilcoxon Rank Sum Test (Marascuilo & Serlin (1988), Siegel (1956)) to understand if there are differences between two paired numerical groups. This test is a non-parametric test since our data groups do not obey a normal distribution. It works based on the medians and their locations in two lists. First, it mixes the samples and ranks them from lowest to highest. Then the rank of the data samples from the two groups is summed up. This test evaluates if there is a significant or not difference between the median of two data samples. Next, the hypothesis is investigated by comparing the achieved values with some threshold values (Wilcoxon (1992)). There are two hypotheses for comparison. The first one assumes that there is no difference between the

two dependent groups and the alternative hypothesis assumes that there is a difference between the two dependent groups. We have applied the Wilcoxon Rank Sum test in the case of the Relevance metric for both examined algorithms for our list of examined queries, which are either Natural Language queries or Java API methods. In the next chapter, we will discuss this in detail.

CHAPTER 5

RESULTS AND DISCUSSION

In this chapter, we present the results of our empirical evaluation that we have conducted to evaluate the recommendations of code examples made by our two proposed approaches. At first, we present a set of examples of the returned code examples for the two proposed approaches based on a set of examined queries, *i.e.*, Natural Language and API-based queries. Then we answer the research questions one by one in each section based on the obtained results.

5.1 Code Examples Recommendations based on Informal documentation

Tables 5.1 and 5.2 show examples of natural language-based and API names-based queries. These tables have three columns, *i.e.*, Query, Rank and Recommended code examples. In Table 5.1, the queries are natural language-based and in Table 5.2, they are in Java API method name formats. In both tables, the Rank column denotes the algorithm type (*RH for Random Hyperplane-based LSH* and *QA for Query-Aware LSH*) and the rank of the first relevant example that was specified by two experienced software developers among which one is the author of this thesis and the second developer is not part of this research work. Finally, the third column shows the Recommended code example, from Stack Overflow, for the query in question. We provide more examples of code examples' recommendations in APPENDIX I (Cf. Table-A I-1 and Table-A I-2).

Table 5.1 First relevant recommended code examples (with their ranks) returned by the two suggested algorithms for natural language-based queries

Query	Rank	Recommended code examples

How do I create a file and write to it?	RH (Rank = 3)	[DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
		//build DOMs
		Document doc1 = builder.parse(new File("file1.xml"));
		Document doc2 = builder.parse(new File("file2.xml"));
		//get all node ids from doc2 and iterate
		NodeList list = doc2.getElementsByTagName("node_(id)");
		for(int i = 0 ; i < list.getLength() ; i++){
		Node n = list.item(i);
		//extract the id
		String id = n.getTextContent();
		//now get all node id elements from doc1
		NodeList list2 = doc1.getElementsByTagName("node_"+id);
		for(int j = 0 ; j < list2.getLength() ; j++){
		Node m = list2.item(j);
		//import them into doc2
		Node imp = doc2.importNode(m,true);
		n.getParent().appendChild(imp);
		} }
		//write out the modified document to a new file
		TransformerFactory tFactory = TransformerFactory.newInstance();
		Transformer transformer = tFactory.newTransformer();
		Source source = new DOMSource(doc2);
		Result output = new StreamResult(new File("merged.xml"));
		transformer.transform(source, output);]

Continued on next page

Table 5.1 First relevant recommended code examples (with their ranks) returned by the two suggested algorithms for natural language-based querie (continued)

Query	Rank	Recommended code examples

	QA (Rank = 4)	[public DDHLogger(String filePath, String yourfilename) {
		System.out.println("filePath: " + filePath);
		System.out.println("yourfilename: " + yourfilename);
		//String fileName = "C:\\Users\\ home-1 \\Desktop \\" + yourfilename;
		String fileName = filePath + yourfilename;
		System.out.println("fileName::–> " + fileName);
		// file = new File(fileName);//Creates the file
		file = new File(fileName);//Creates the file
		System.out.println("file: " + file);
		try {
		fw = new FileWriter(file, true);
		//INSTANTIATE PRINTWRITER HERE
		pw = new PrintWriter(fw);
		} catch (IOException e) {
		e.printStackTrace();
		}
		//allows append to the file without over writing. The TRUE keyword is used for append }]

Table 5.2 First relevant recommended code examples (with their ranks) returned by the two suggested algorithms for API-based queries

Query	**Rank**	**Recommended code examples**[1]
Writer.write()	RH (Rank = 0)	[No Recommended code Example]
	QA (Rank = 1)	[CsvListReader reader = new CsvListReader(new FileReader(inputCsv),
		CsvPreference.STANDARD_PREFERENCE);
		CsvListWriter writer = new CsvListWriter(new FileWriter(outputCsv),
		CsvPreference.STANDARD_PREFERENCE);
		List<String> columns;
		while ((columns = reader.read())= null) {
		System.out.println("Input: " + columns);
		// Add new columns
		columns.add(1, "Column_2");
		columns.add("Last_column");
		System.out.println("Output: " + columns);
		writer.write(columns);
		}
		reader.close();
		writer.close();]

To address our research questions, we have selected 20 natural language-based programming questions from other publications (Kim *et al.* (2018) and Diamantopoulos, Karagiannopoulos & Symeonidis (2018)). We have also selected Java API methods that were used in the answers to these questions. Therefore, our experiment will involve two sets of queries: natural language questions as well as Java API methods. We have chosen two sets of 20 questions to ensure that we have a sufficient number of samples to test and to generate the metrics according to query type.

5.2 *Results of RQ11: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of HitRate?*

In this section, we have focused on the performance of the two examined algorithms, *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH* that we have measured in terms of *HitRate@k*, which measures the fraction of the queries that have returned at least one relevant result in the top *k* recommended items. We have run the two algorithms using queries in the Natural Language as well as API methods format.

Table 5.3 HitRate measure results obtained for the Random Hyperplane-based LSH

Number	Query	Top 10	Top 20	Top 30
1	How to add an image to a JPanel?	1	1	1
2	How to generate a random alpha-numeric string?	0	0	0
3	How do I create a file and write to it?	1	1	1
4	How do I invoke a Java method when given the method name as a string?	0	0	0
5	Remove HTML tags from a String?	0	1	1
6	How to get the path of a running JAR file?	0	0	0
7	Getting a File's MD5 Checksum in Java	0	0	0
8	Loading a properties file from Java package	0	1	1
9	How can I play sound in Java?	0	0	0
10	What is the best way to SFTP a file from a server?	0	1	1
11	How to read a CSV file?	1	1	1
12	How to generate MD5 hash code?	0	0	0
13	How to send a packet via UDP?	0	1	1
14	How to split a string?	0	0	0
15	How to play an audio file?	0	0	0
16	How to upload a file to FTP?	0	1	1
17	How to initialize a thread?	0	0	1
18	How to connect to a JDBC database?	1	1	1
19	How to read a ZIP archive?	0	1	1
20	How to send an email?	0	0	0
	HitRate	0.2	0.5	0.55

Tables 5.3 and 5.4 show the HitRate values for the first and second considered algorithms respectively, *i.e.*, *Random Hyperplane-based LSH* and *Query-Aware LSH*. As it can be noticed from Tables 5.3 and 5.4, the *HitRate* values for the Natural language queries are (0.2, 0.5, 0.55) and (0.5, 0.8, 0.9) for *Random Hyperplane-based LSH* and *Query-Aware LSH* respectively, for the top 10, 20 and 30 generated recommendations. As it can be noticed from the obtained results, the *Query-Aware LSH* yields better results in terms of *HitRate* than the *Random*

Table 5.4 HitRate measure results obtained for the Query-Aware LSH approach

Number	Query	Top 10	Top 20	Top 30
1	How to add an image to a JPanel?	0	1	1
2	How to generate a random alpha-numeric string?	1	1	1
3	How do I create a file and write to it?	1	1	1
4	How do I invoke a Java method when given the method name as a string?	0	1	1
5	Remove HTML tags from a String?	1	1	1
6	How to get the path of a running JAR file?	0	0	0
7	Getting a File's MD5 Checksum in Java	0	1	1
8	Loading a properties file from Java package	0	0	1
9	How can I play sound in Java?	1	1	1
10	What is the best way to SFTP a file from a server?	0	1	1
11	How to read a CSV file?	1	1	1
12	How to generate MD5 hash code?	1	1	1
13	How to send a packet via UDP?	0	1	1
14	How to split a string?	1	1	1
15	How to play an audio file?	1	1	1
16	How to upload a file to FTP?	0	0	1
17	How to initialize a thread?	0	1	1
18	How to connect to a JDBC database?	1	1	1
19	How to read a ZIP archive?	0	0	0
20	How to send an email?	1	1	1
	HitRate	0.5	0.8	0.9

Hyperplane-based LSH. This is due to *Query-Aware LSH*'s ability to identify more relevant code examples based on the given query. As the number of recommended examples increases (from 10 to 30), the probability of finding relevant code also increases, resulting in a higher *HitRate* value.

Additionally, we can notice that the *HitRate* values obtained for the *Query-Aware LSH* algorithm are higher than the *Random Hyperplane-based* one. In effect, the *Query-Aware LSH* approach uses the distance of Hash vectors to determine which data points to collect in the query vicinity. It is likely that points with hash values that are close to the query's hash value will also be close to the query points in the given space. However, the *Random Hyperplane-based LSH* works differently. It maps numerical elements of a vector from decimal to binary space (either 0 or 1) based on their sign (negative values are mapped to 0 and positive values are mapped to 1) after applying the Hash mapping. Vectors that have the same binary values are then placed in the same buckets. This means that the bucket assignment for data samples in *Random Hyperplane-based LSH* is based on the positive and negative elements in their Hash vectors, rather than their Euclidean distances. This suggests that assigning data samples to buckets based on the sign of the numbers is not as precise as assigning based on the distance between the vectors.

Table 5.5 HitRate values obtained for the Random Hyperplane-based LSH using the API method's names

Number	Query	Top 10	Top 20	Top 30
1	Jpanel.add()	1	1	1
2	StringBuilder.append()	1	1	1
3	Writer.write()	0	0	0
4	Method.invoke()	0	1	1
5	Jsoup.parse()	0	0	0
6	URLDecoder.decode()	0	0	0
7	MessageDigest.digest()	0	0	0
8	Properties.load()	0	0	0
9	AudioSystem.getAudioInputStream()	1	1	1
10	JSch.getSession()	0	0	0
11	BufferedReader.readLine()	0	0	0
12	MessageDigest.getInstance()	0	0	0
13	DatagramSocket.send()	0	0	0
14	String.split()	0	0	0
15	MediaPlayer.play()	0	1	1
16	FTPClient.storeFile()	0	0	0
17	thread.start()	0	0	0
18	DriverManager .getConnection()	0	0	1
19	zipFile.getInputStream()	0	0	0
20	MimeMessage.setFrom()	0	0	0
	HitRate	0.15	0.25	0.3

Table 5.6 HitRate values obtained for the Query-Aware LSH using API names' methods

Number	Query	Top 10	Top 20	Top 30
1	Jpanel.add()	1	1	1
2	StringBuilder.append()	1	1	1
3	Writer.write()	1	1	1
4	Method.invoke()	1	1	1
5	Jsoup.parse()	0	0	0
6	URLDecoder.decode()	0	0	0
7	MessageDigest.digest()	0	0	0
8	Properties.load()	0	0	0
9	AudioSystem.getAudioInputStream()	1	1	1
10	JSch.getSession()	0	0	0
11	BufferedReader.readLine()	1	1	1
12	MessageDigest.getInstance()	0	0	0
13	DatagramSocket.send()	0	0	0
14	String.split()	1	1	1
15	MediaPlayer.play()	1	1	1
16	FTPClient.storeFile()	0	0	0
17	thread.start()	0	0	0
18	DriverManager .getConnection()	1	1	1
19	zipFile.getInputStream()	0	0	0
20	MimeMessage.setFrom()	1	1	1
	HitRate	0.5	0.5	0.5

Tables 5.5 and 5.6 show the results of the *Random Hyperplane-based LSH* and *Query-Aware LSH* algorithms in terms of *HitRate*, when dealing with queries that are exact names of API methods (*i.e.*, as extracted from Stack Overflow). As it can be noticed from both tables, the *HitRate* values for the examined API methods queries are (0.15, 0.25, 0.30) and (0.5, 0.5, 0.5) respectively for the top 10, 20 and 30 recommendations generated based on the two considered algorithms.

Table 5.7 Summary of the *HitRate* values obtained from the *Random Hyperplane-based LSH* and the *Query-Aware LSH* using the two types of examined queries

Query Type	Approaches	Top 10	Top 20	Top 30
Natural Language-based Queries	*Random Hyperplane-based LSH*	0.2	0.5	0.55
	Query-Aware LSH	0.5	0.8	0.9
API Names-based Queries	*Random Hyperplane-based LSH*	0.15	0.25	0.3
	Query-Aware LSH	0.5	0.5	0.5

Not surprisingly, when converting the names of API methods into natural language words, the results in terms of *HitRate* values are better as indicated in Table 5.7. This could be justified by the fact that the BERT model is semantic-based and not token-based, and since our algorithms are based on BERT, using natural language-based queries instead of names of API methods can be useful when making recommendations of code examples since the algorithms could extract more information, particularly in terms of semantic, from the natural language sentences than API methods.

Regardless of the type of the query given as input to the suggested recommender, the findings summarized in Table 5.7, show that overall that the *Query-Aware* algorithm returns higher values of HitRate than the *Random Hyperplane-based*, indicating therefore that the Query-based *LSH* approach is in general doing better than the *Random Hyperplane-based* in returning the relevant code examples. Higher HitRate values indicate that the algorithm is more effective at providing relevant code examples based on the given queries. In effect, the more successful the algorithm is at providing relevant code examples, the higher the values of HitRate will be.

5.3 *Results of RQ12: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Mean Reciprocal Rank?*

Table 5.8 and 5.9 indicate the results of the *MRR* measure for the two considered algorithms, *i.e.*, the *Random Hyperplane-based LSH* and the *Query-Aware LSH* based on the two types of examined queries, *i.e.*, Natural language and API methods' names respectively. We recall that *MRR* is calculated as the mean of the inverse of the first rank of the recommended items when the *HitRate* occurs.

As it can be noticed from Table 5.8, the obtained values of the *MRR* measure for the *Random Hyperplane-based LSH* and the *Query-Aware LSH* are 0.0774 and 0.2582 respectively. High *MRR* values means that the relevant code examples, for a particular API method, are recommended in lower ranks of the list of code examples returned by the algorithm in question.

Table 5.8 MRR results for both suggested approaches using natural language-based queries

Queries	First rank of the HitRate (1)	Inverse of the first rank(1)	First rank of the HitRate (2)	Inverse of the first rank(2)
Q1	9	0.11	7	0.142
Q2	–	0	1	1
Q3	3	0.33	4	0.25
Q4	–	0	14	0.072
Q5	15	0.066	6	0.16
Q6	–	0	–	0
Q7	–	0	16	0.062
Q8	12	0.083	23	0.043
Q9	–	0	2	0.5
Q10	11	0.09	13	0.076
Q11	2	0.5	3	0.33
Q12	–	0	6	0.166
Q13	18	0.055	17	0.058
Q14	–	0	1	1
Q15	–	0	2	0.5
Q16	12	0.083	–	0
Q17	22	0.045	18	0.055
Q18	9	0.11	4	0.25
Q19	13	0.076	–	0
Q20	–	0	2	0.5
MRR	0	0.0774	0	0.2582

According to Table 5.8, when comparing the rank of the returned results for both algorithms, *i.e.*, the *Random Hyperplane-based LSH* and the *Query-Aware LSH*, we can notice that the rank of the code examples returned by the *Query-Aware LSH* is mostly lower than *Random Hyperplane-based LSH* if both return at least one relevant result. For instance, Let us take the first query (Q1), the *Query-Aware LSH* returns the first relevant code example at rank 7, while the *Random Hyperplane-based LSH* recommends the first relevant code example at rank 9.

It is important to mention also that there are some queries for which the second algorithm, *i.e.*, the *Query-Aware LSH* was not able to return relevant code examples, while the first one, *i.e.*, the *Random Hyperplane-based LSH* was able to do so. For example, Let us take the 16th question (Q16), the *Random Hyperplane-based LSH* returned the first relevant code example at rank 12, while the *Query-Aware LSH* was unable to return at least one pertinent code example. For

Table 5.9 MRR results for both suggested approaches using the API names methods

Queries	First rank of the HitRate (1)	Inverse of the first rank(1)	First rank of the HitRate (2)	Inverse of the first rank(2)
Jpanel.add()	6	0.166	4	0.25
StringBuilder.append()	10	0.1	1	1
Writer.write()	–	0	1	1
Method.invoke()	14	0.0714	1	1
Jsoup.parse()	–	0	–	0
URLDecoder.decode()	–	0	–	0
MessageDigest.digest()	–	0	–	0
Properties.load()	–	0	–	0
AudioSystem.getAudioInputStream ()	1	1	4	0.25
JSch.getSession()	–	0	–	0
BufferedReader.readLine()	–	0	3	0.33
MessageDigest.getInstance()	–	0	–	0
DatagramSocket.send()	–	0	–	0
String.split()	–	0	3	0.33
MediaPlayer.play()	17	0.058	3	0.33
FTPClient.storeFile()	–	0	–	0
thread.start()	–	0	–	0
DriverManager .getConnection()	30	0.033	3	0.33
zipFile.getInputStream()	–	0	–	0
MimeMessage.setFrom()	–	0	10	0.1
MRR	–	0.07142	–	0.246

queries that did not have any related results, the MRR assigns a value of 0 during its calculation of the *MRR* value.

Another interesting finding indicated in Table 5.9 relates to the difference in the *MRR* measure when the type of query varies. In effect, the *MRR* value, when dealing with API method's names, for the *Query-Aware LSH* is 0.246, which is higher than the average of the *MRR* value for the *Random Hyperplane-based LSH*, which is equal to 0.071. As previously explained, a high *MRR* value indicates that an algorithm returns the relevant code examples in lower ranks. However, when the *MRR* values is low, the algorithm returns the relevant code examples in higher ranks or do not return any relevant result.

Table 5.10 shows similar trends, regarding the types of examined queries. In effect, the natural language-based queries yield better than the API methods' names-based queries, which, as

Table 5.10 Values of the *MRR* measure obtained for the *Random Hyperplane-based LSH* and the *Query-Aware LSH* based on the query type

Query Type	Algorithm	*MRR*
Natural Language-based	*Random Hyperplane-based LSH*	0.0774
	Query-Aware LSH	0.2582
API- Method's names based	*Random Hyperplane-based LSH*	0.07142
	Query-Aware LSH	0.246

explained earlier, could be justified by the capability of the considered algorithms to extract more contextual and semantic information from the natural language Stack Overflow code examples than queries in the form of identifiers, *i.e.*, names of API methods.

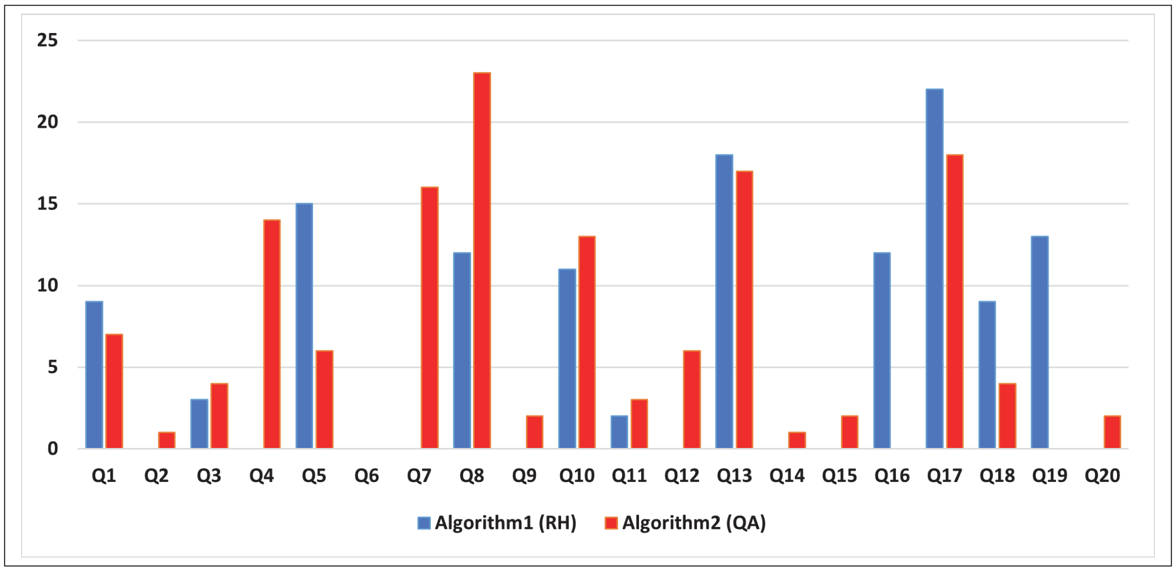


Figure 5.1 The rank of the first code examples generated for natural language-based queries

Additionally, when considering both Figures 5.1 and 5.2, we can notice that in 9 of 20 natural language-based questions, both algorithms returned at least one relevant code example. In this case, the average of the first rank of the returned relevant code example generated by the *Query-Aware LSH* algorithm is 10.55, which is lower than 11.22, which is the one for the *Random Hyperplane-based LSH*. For API-based queries, in 6 of 20 natural language-based questions, both algorithms returned at least one relevant code example. In this case, the average

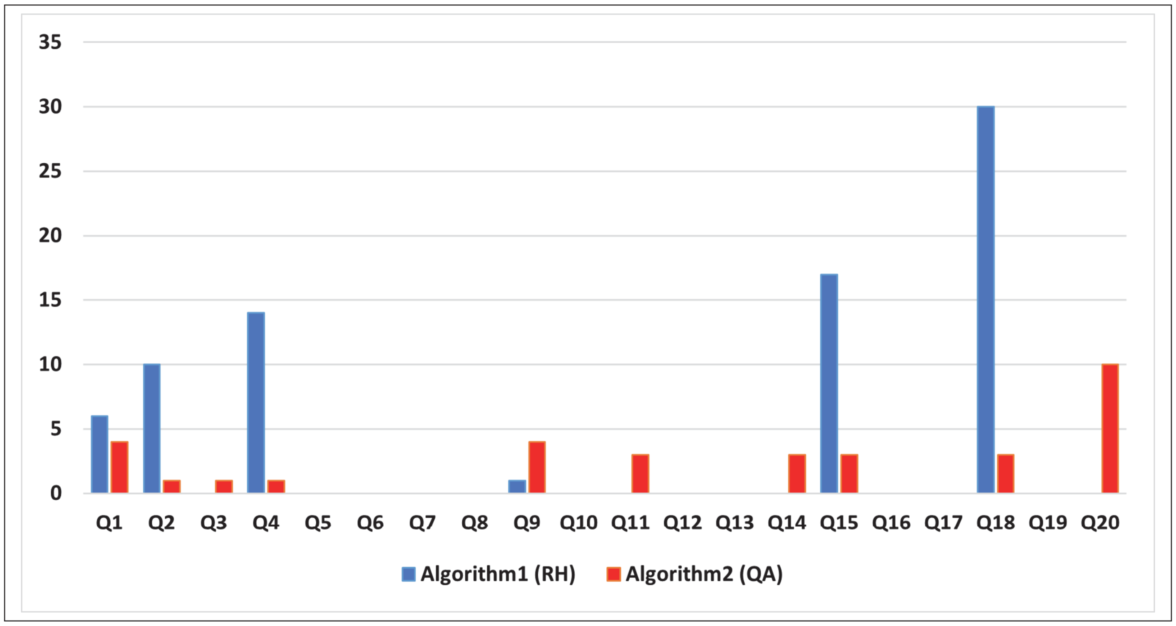


Figure 5.2 The rank of the first code examples generated for API names-based queries

of the first rank of the returned relevant code example for the *Query-Aware LSH* algorithm is 2.6, while this value is equal to 13 for the *Random Hyperplane-based LSH*.

The rank of the first relevant code example is lower in *Query-Aware LSH* for Natural language queries (5 out of 9 queries) and (5 out of 6) for API-based queries when both algorithms returned at least one relevant result. Additionally, in some cases, the *Random Hyperplane-based LSH* was not able to find at least one relevant code example (8 questions for Natural language queries, and 4 queries for API Method's names- based queries), while the *Query-Aware LSH* was able to return at least one relevant code example.

5.4 *Results of RQ13: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Average execution time*?

As indicated in Table 5.11, the LSH creation and recommendation times for both the *Random Hyperplane-based LSH* and the *Query-Aware LSH* algorithms are measured in terms of the *Average execution time*.

Table 5.11 The Average Execution Time of the proposed approaches, *i.e.*, *Random Hyperplane-based LSH* (1) and *Query-Aware LSH* (2) (in seconds)

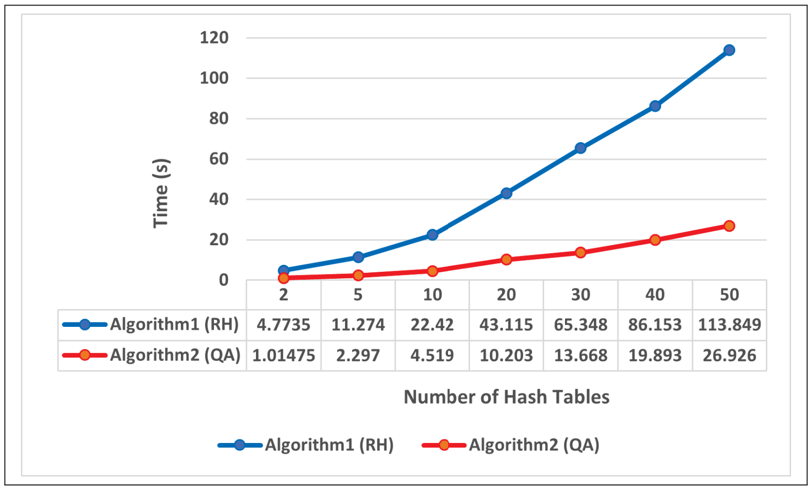
Number of hash tables	**LSH creation time(1)**	**Recommendation time(1)**	**LSH creation time(2)**	**Recommendation time(2)**
2	4.7735	0.3241	1.01475	0.000779
5	11.274	0.373	2.297	0.0007476
10	22.420	1.247	4.519	0.0008069
20	43.115	1.236	10.203	0.0008623
30	65.348	1.225	13.668	0.0007353
40	86.153	1.2814	19.893	0.0007533
50	113.849	1.241	26.926	0.000861

We have measured these elements based on the different number of hash tables and changed this parameter from 2 to 50. The number of hash tables defines the number of times that the hashing algorithm is applied to the query and data samples. This means that as the number of hash tables increases, more data points are mapped to the same bucket as the query. Hence, more hash tables can improve the accuracy of the nearest neighbor search by finding more similar data samples. Yet, an increase in hash tables may increase the computational complexity. We applied different numbers of hash tables experimentally based on our available processing resources, *i.e.*, available memory on Google Colab[2].

Table 5.11 reports the *Average Execution Time* that we have measured for all of the considered queries. As it can be noticed, the *Query-Aware LSH* algorithm takes less time than the *Random Hyperplane-based* at both the LSH creation phase as well as the recommendations generation phase. In effect, the *Query-Aware LSH* algorithm took on average 11.21 seconds when creating Hash tables, while it took less than 1 milliseconds on average when making recommendations of code examples. However, the *Random Hyperplane-based* algorithm took on average 49.56 seconds when creating Hash tables, while it took 0.98 seconds on average when making recommendations of code examples.

[2] https://colab.research.google.com/

The above-mentioned findings can be explained by the fact that the *Query-Aware LSH* does not require all data samples to be placed into buckets. Instead, only samples whose hash values that are close to the query's hash value are considered. Furthermore, the Query-Aware LSH algorithm reduces the search space by filtering data samples whose hash vectors are in the vicinity of the query when LSH is created. This occurs in the initial stages of the search process, enabling the algorithm to find relevant samples in a more efficient manner (Huang *et al.* (2015).



	2	5	10	20	30	40	50
Algorithm1 (RH)	4.7735	11.274	22.42	43.115	65.348	86.153	113.849
Algorithm2 (QA)	1.01475	2.297	4.519	10.203	13.668	19.893	26.926

Figure 5.3 LSH creation time based on the different umber of Hash tables

Figure 5.3 demonstrates the time of the Hash tables' creation as well as the assignment of data samples to their buckets for the two proposed algorithms *i.e.*, the *Random Hyperplane-based* as well as the *Query-Aware LSH* algorithms. As we can notice from Figure 5.3, the minimum and maximum values of this measure for these algorithms are (4.77,113.85) and (1.014, 26.926) respectively. The justification for these values could be that the *Random Hyperplane-based* attempts to allocate all of the data samples to their corresponding buckets and by increasing the number of Hash tables, more computation is necessary to calculate Hash values and assign the samples into buckets. On the other hand, the *Query-Aware LSH* filters data samples whose Hash values are in close proximity to a query's vector. The use of Hash values reduces the dimension of the vectors, therefore significantly decreasing computation time.

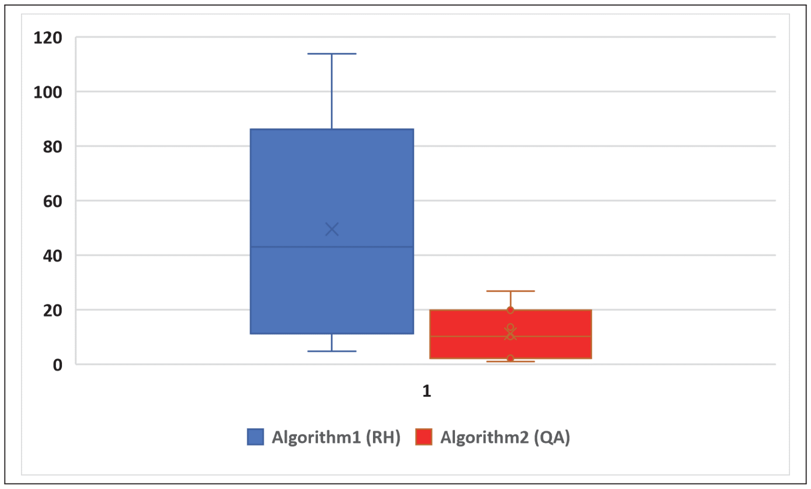


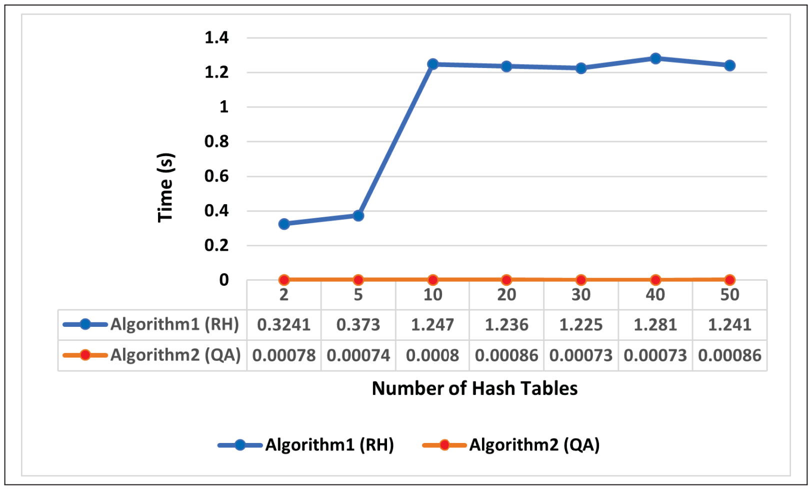
Figure 5.4 The Boxplot of the LSH creation time (sec) based on the different number of Hash tables for the two applied algorithms

Figure 5.4 denotes the Boxplot of the LSH creation time for the two considered algorithms, *i.e.*, the *Random Hyperplane-based* and the *Query-Aware LSH* algorithms. According to this Figure, the values of the first Quartile, Median and third Quartile are (11.274, 43.115, 86.153) and (2.297, 10.203, 19.893) for the first and second algorithms respectively. The differences between the Maximum and Minimum values for the two algorithms are 109.0755 and 25.912 respectively.

Figure 5.5 shows the average values of the recommendation time for both the *Random Hyperplane-based* and the *Query-Aware* algorithms.

According to Figure 5.5, the recommendation time of the *Random Hyperplane-based* algorithm varies from 0.3241 seconds to 1.241 seconds, while increasing the Hash table from 2 to 50. However, this time ranges from 0.00077 seconds to 0.000861 seconds for *Query-Aware LSH* algorithm.

Figure 5.6 denotes the Boxplot of the LSH recommendation time for both considered algorithms, *i.e.*, the *Random Hyperplane-based* and the *Query-Aware LSH* algorithms. According to this Figure, the values of the first Quartile, Median and third Quartile are (0.373, 1.236, 1.2814) and (0.0007476, 0.0008623, 0.0007533) for the first and second algorithms respectively. The



	2	5	10	20	30	40	50
Algorithm1 (RH)	0.3241	0.373	1.247	1.236	1.225	1.281	1.241
Algorithm2 (QA)	0.00078	0.00074	0.0008	0.00086	0.00073	0.00073	0.00086

Figure 5.5 LSH recommendation time based on the number of Hash tables

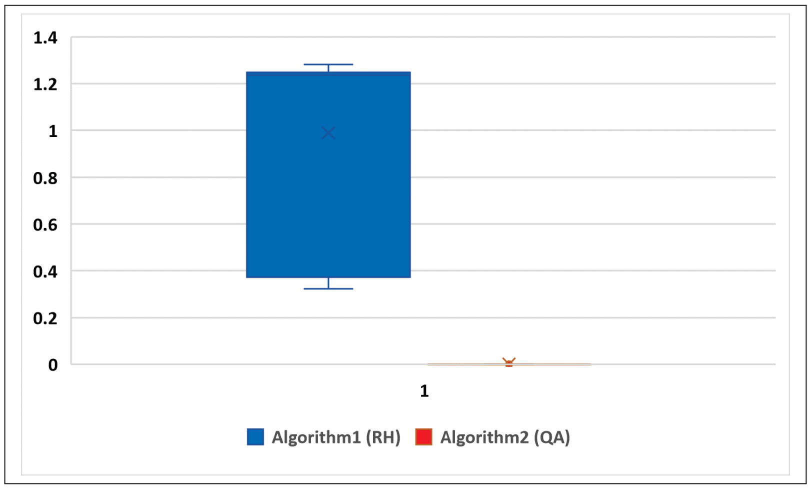


Figure 5.6 The Boxplot of the LSH recommendation time (sec) based on the different number of Hash tables for the two applied algorithms

difference between the Maximum and Minimum values for the two algorithms are 0.9573 and 0.0001257 respectively.

5.5 *Results of RQ14: How do the Random Hyperplane-based LSH and the Query-Aware LSH perform in terms of Relevance?*

Tables 5.12 and 5.13 report the Relevance values obtained for the two examined algorithms, *i.e.*, the *Random Hyperplane-based LSH* and the *Query-Aware LSH*, while considering the two types of considered queries, *i.e.*, Natural language-based and API-based queries for the top 10 returned recommendations.

Table 5.12 The results of the Relevance for both algorithms for the top 10 generated recommendations

Number	Query	The score of the Relevance (1)	The score of the Relevance (2)
1	How to add an image to a JPanel?	3	2
2	How to generate a random alpha-numeric string?	1	4
3	How do I create a file and write to it?	3	3
4	How do I invoke a Java method when given the method name as a string?	1	2
5	Remove HTML tags from a String?	1	4
6	How to get the path of a running JAR file?	1	1
7	Getting a File's MD5 Checksum in Java	1	1
8	Loading a properties file from Java package	2	2
9	How can I play sound in Java?	2	4
10	What is the best way to SFTP a file from a server?	2	2
11	How to read a CSV file?	3	3
12	How to generate MD5 hash code?	1	3
13	How to send a packet via UDP?	2	2
14	How to split a string?	0	4
15	How to play an audio file?	1	4
16	How to upload a file to FTP?	2	2
17	How to initialize a thread?	2	2
18	How to connect to a JDBC database?	4	4
19	How to read a ZIP archive?	2	1
20	How to send an email?	0	4
	The Average of Relevance	1.7	2.7

Table 5.12 reports the scores of the code examples recommendations for natural language-based queries for both studied algorithms. In this table, code examples with the *Relevance* score of 3 or 4 are considered useful for developers. There are more queries in the *Query-Aware LSH* that have received a score of 3 or 4 on the *Relevance* metric, indicating that *Query-Aware LSH* returns more relevant results (10 recommendations) than the *Random Hyperplane-based LSH* (4

Table 5.13 The results of the Relevance for both algorithms in case of API methods' names for top 10 recommendations

Number	Query	The score of the Relevance (1)	The score of the Relevance (2)
1	Jpanel.add()	4	4
2	StringBuilder.append()	4	4
3	Writer.write()	1	4
4	Method.invoke()	4	4
5	Jsoup.parse()	2	2
6	URLDecoder.decode()	2	1
7	MessageDigest.digest()	1	1
8	Properties.load()	1	1
9	AudioSystem.getAudioInputStream ()	4	4
10	JSch.getSession()	1	1
11	BufferedReader.readLine()	2	4
12	MessageDigest.getInstance()	1	1
13	DatagramSocket.send()	1	3
14	String.split()	1	4
15	MediaPlayer.play()	4	4
16	FTPClient.storeFile()	1	1
17	thread.start()	1	1
18	DriverManager .getConnection()	4	4
19	zipFile.getInputStream()	1	1
20	MimeMessage.setFrom()	1	4
	The Average of the Relevance	2.05	2.65

Table 5.14 Values of the *Relevance* for the *Random Hyperplane-based LSH* and the *Query-Aware LSH* based on the query type

Query Type	Algorithm	*Relevance*
Natural Language-based	*Random Hyperplane-based LSH*	1.7
	Query-Aware LSH	2.7
API-based	*Random Hyperplane-based LSH*	2.05
	Query-Aware LSH	2.65

recommendations). The average value of the *Relevance* for the *Query-Aware LSH* is equal to 2.7, higher than the average value of Relevance, *i.e.*, 1.7 for the *Random Hyperplane-based LSH*.

Table 5.13 shows that the number of the relevant recommendations generated by the *Query-Aware LSH* (11 results) is higher than the *Random Hyperplane-based LSH* (6 results). Additionally, the average value of the *Relevance* score for the *Query-Aware LSH* algorithm is equal to 2.65, *i.e.*, higher than the score for the *Random Hyperplane-based* algorithm, which is equal to 2.05. According to Table 5.13, the number of the irrelevant results (with score 1 or 2) for the *Random Hyperplane-based LSH* is 14, while this number is equal to 9 for the *Query-Aware LSH*.

Table 5.14 summarizes the results obtained for the *Relevance* metric based on the considered types of queries and algorithms. In effect, when it comes to the type of the query leveraged, our findings show similar trends to previous research questions. In fact, the natural language-based queries yield high scores in terms of *Relevance* when recommending code examples. This can be explained by the use of the *BERT* model which takes into account both the semantic and contextual information of the queries and code examples (Devlin *et al.* (2018)) and therefore performs well when dealing with natural language text.

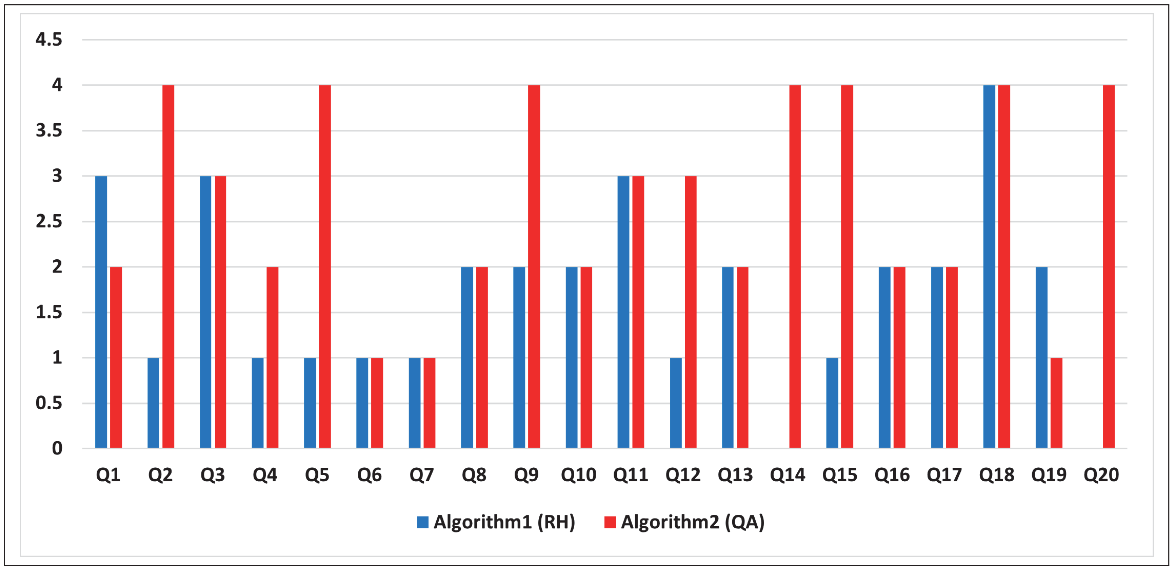


Figure 5.7 Relevance score of the two leveraged algorithms in case of natural language-based queries

Figures 5.7 and 5.8 show the obtained results in terms of *Relevance* when dealing with the two types of examined queries. According to these figures, the higher scores of relevance (*i.e.*, 3 or 4) indicate a more relevant recommendation of code examples. For most of the queries, the

Relevance score of the *Query-Aware LSH* approach is higher than the *Random Hyperplane-based* for both natural language-based and API methods' names-based queries as shown in both figures.

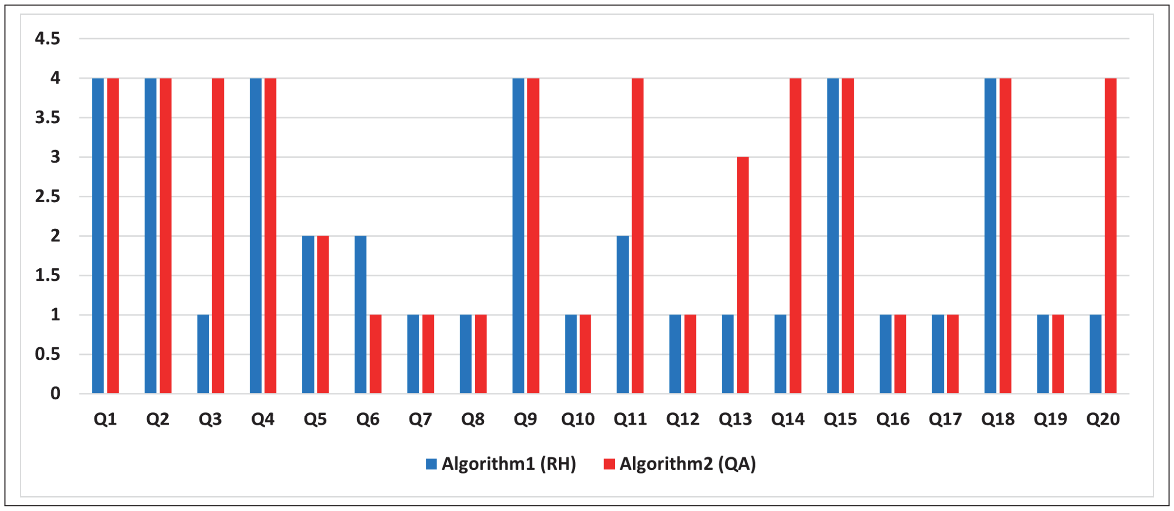


Figure 5.8 Relevance score of the two studied algorithms in case of API methods' names-based queries

5.6 *Results of RQ2: Is there any difference in terms of Relevance values between the Random Hyperplane-based LSH and the Query-Aware LSH algorithms?*

In this section, we report the results of the comparison that we have performed between the two examined algorithms, *i.e.*, the *Random Hyperplane-based LSH* and the *Query-Aware LSH* using the Wilcoxon Rank Sum Test.

We could investigate if there is a significant difference between the *Random Hyperplane-based LSH* and the *Query-Aware LSH* in terms of Relevance, while considering both types of examined queries. The *Relevance* metric is a numerical value from 0 to 4, and two software developers have provided their opinion for all of the queries in terms of this metric for both Natural language-based and API method names-based queries. We therefore could compare the results of both algorithms, *i.e.*, *Query-Aware LSH* and *Random Hyperplane-based LSH* in terms of the *Relevance* of the recommendations of code examples made by both algorithms using different types of queries by applying a proper test, *i.e.*, the Wilcoxon Rank Sum Test (Mann-Whitney U Test).

Table 5.15 Results of the comparison in terms of *Relevance* for the *Random Hyperplane-based LSH* and the *Query-Aware LSH* for both types of queries

Query Type	Critical U-value	U-value	Z-Score	Critical P-value	P-value	Result
Natural Language-based	127	107	-2.5	.05	.012	Significant
API-based	37	47	-1.41	.05	.16	Not Significant

As it can be noticed from Table 5.15, which summarizes the *Relevance* results for natural language-based and API methods' names-based queries. For natural language-based queries, the U-value is 107, and the critical value of U at p < .05 is 127. When the U-value is less than the critical value, the results are not achieved by chance and since the same applies in our case (since the U-value < 127), the results are not achieved by coincidence. Additionally, the Z-Score shows whether two groups are equally distributed or not, and if this is the case, this metric is close to zero. In our case, the Z-Score is -2.5 and the p-value is equal to 0.012. Since the p-value is less than the critical value (*i.e.*, 0.05). This means that in the case of the Natural language-based queries, the results of the two considered algorithms, *i.e.*, the *Query-Aware LSH* and the *Random Hyperplane-based LSH* are not achieved by chance, *i.e.*, there is a significant difference in terms of the *Relevance* metric between the two examined algorithms.

Table 5.15 reports the results obtained when considering the API methods' names- based queries. As it can be noticed from this table, the U-value is equal to 47 and the critical value of U at p < .05 is equal to 37. Since the U-value is greater than the critical value, the differences between the two groups are more likely to be due to chance, so the null hypothesis can not be rejected in this case. Additionally, the Z-Score is equal to -1.41 and the p-value is equal to 0.16. Moreover, the Z-Score is not close to zero. We therefore conclude that there are not significant differences, in terms of relevance of recommended code examples, between the *Query-Aware LSH* and the *Random Hyperplane-based LSH* when dealing with API methods' names-based queries.

Overall, our findings show that there are statistically significant differences in terms of *Relevance* of recommendations of code examples made by the *Query-Aware LSH* and *Random Hyperplane-*

based LSH, when dealing with natural language queries. However, this is not the case when examining the API methods' names-based queries.

5.7 Threats to validity

Although we have carefully conducted the evaluation step, there are some factors that may threaten the validity of our empirical evaluation. These threats to validity fall into three categories, which we will discuss in more detail in the following sections.

- **Internal Validity:** Internal validity pertains to the factors that have influenced the obtained results. It examines the degree of success in determining the relationship between the independent and dependent factors and the factors affecting accuracy (Wright, Kim & Perry (2010)).

  There are some potential threats to the internal validity of our study that could impact the accuracy of the results. One such threat is the pre-processing of the data samples, which involved filtering based on regular expressions, code example length, and score of the posts. While this process may have excluded some posts that contained correct API method usage, it is possible that the trapped code examples were too short (*e.g.*, one or two lines) and were therefore filtered out. This could threaten the validity of the results and should be carefully considered. In addition, some code examples that contained the correct API method usage may have been filtered out due to their low scores, which did not meet the specified threshold values.
- **External validity:** External validity denotes if a study's results could be generalized to other settings, contexts, and data samples.

  We applied our algorithms to Java code examples that were obtained from Stack Overflow posts. However, we may need to evaluate the performance of our algorithms on posts containing code examples in other programming languages, such as C/C++, Python, etc. This will allow us to determine the extent to which our *LSH*-based algorithms can return relevant code examples in other programming languages and measure their performance in terms of defined metrics.

Another threat to the study is the limited number of queries evaluated, including both natural language and API-based queries. We selected the Natural language queries that were previously investigated from software engineering literature and selected the API-based queries based on those queries. However, we did not evaluate a large number of questions on Stack Overflow, so the observed metrics may change with a larger number of queries.

- **Construct validity:** Construct validity attempts to ensure that the experiment is really able to measure the intended metrics. In other words, it assures that the link between the theory and the observation is well-established. One of the possible threats to construct validity is the lack of controlled experiments and surveys that may help gain more insights into the obtained results. Our recommendation approach is evaluated by two software developers that have completed multiple projects in Java and Android languages. However, to gain a more comprehensive understanding of how our algorithms perform, we need to evaluate them with a wider range of developers with varying skill levels, from junior to senior.

  Moreover, we have selected the Natural language queries (20 queries) that have been investigated in prior research works, *i.e.*, (Diamantopoulos *et al.* (2018) and Kim *et al.* (2018)). Regarding the APIs, we have extracted them from the Stack Overflow answer posts. Since there are a large number of Java libraries, classes, and API methods, these numbers of investigated queries may not be representative of all queries. However, we have mirrored previous research works published in venues in the area of empirical software engineering(Rubei *et al.* (2020) and Kim *et al.* (2018)).

CHAPTER 6

CONCLUSION AND FUTURE WORK

Software developers use open-source projects and informal documentation to find the existing implementation of code elements (*e.g.*, classes and methods) and use them to accomplish their programming tasks. Informal documentation is an integral resource for programmers that incorporates code examples of API methods in different programming languages (Rigby & Robillard (2013), Bacchelli, Lanza & Robbes (2010)).

Stack Overflow is one of the most invaluable resources as informal documentation that developers use daily to exchange their ideas (Rubei *et al.* (2020)). In our work, we have downloaded the Stack Overflow dataset that included all of the posts from 2008 to 2022 and imported them to a SQL database, then filtered the posts that were discussed about Java. We selected the posts that were verified as answer posts by developers and extracted code examples from these posts.

In this research work, we present a novel approach to recommend code examples based on two algorithms, *i.e.*, *Query-Aware LSH* and *Random Hyperplane-based LSH*. The considered algorithms use *BERT* and *LSH* to recommend examples from Stack Overflow posts. *BERT* is one of the transformer-based approaches that is widely-used for language translation. This algorithm is used for language representation and it has implemented the semantic concept in language sentences. We have applied BERT on Java code examples instead of language sentences since it has already been implemented on code examples in some publications. Then we have applied LSH to recommend code examples that correspond to the queries in question.

LSH is useful especially when the size of data samples is large because it reduces the search space by removing the irrelevant data samples at the initial steps of an algorithm. The implemented *LSH*-based algorithms, *i.e.*, *Random Hyperplane-based* and *Query-Aware LSH*, are summarized in the following paragraphs.

Random Hyperplane-based LSH operates by converting the elements of the Hash vectors into binary values (mapping positive numerical values to 1 and negative numerical values to 0).

These binary vectors are then placed into buckets based on their binary values. If the data samples are in the same bucket as the query vector, they are selected and ranked according to their Euclidean distance to the query. However, the *Query-Aware LSH* is a method for finding similar data samples to a given query sample based on the similarity of their Hash vectors. It works by calculating the Euclidean distance between the Hash vector of the query and the Hash vectors of other data samples. If the distance between a data sample's Hash vector and the query's Hash vector is below a certain threshold (*W/2*) and happens for at least L time (threshold time), then that data sample is considered as similar to the query and is added to the collision set. Finally, the data samples in the collision set are sorted and ranked based on their Euclidean distance to the query vector.

We have evaluated the two considered algorithms, *i.e.*, *Random Hyperplane-based* and *Query-Aware LSH* on two types of queries. The first type of queries consisted of 20 natural language questions that have been previously investigated in the software engineering literature. The second type of queries included 20 Java API methods that were chosen from code examples labeled as answers to the natural language questions.

Our findings indicate that using natural language in queries results in more accurate results compared to using API-based queries. Furthermore, the recommendation system is more successful in retrieving useful information when using natural language queries. When evaluating both types of queries, the *Query-Aware LSH* consistently returns more relevant code examples and outperforms the *Random Hyperplane-based LSH* in terms of *HitRate, MRR (Mean Reciprocal Rank), Average Execution time* and *Relevance*. According to the Wilcoxon Rank Sum test applied for the *Relevance* metric, the *Query-Aware LSH* performs better than the *Random Hyperplane-based LSH* when the query is in a natural language format. However, this is not the case for the API Names-based queries.

As a part of future work, we intend to develop our approach into a recommendation system that can be implemented as a plugin for integration into the Eclipse IDE. This will enable the system to assist developers with tasks they are working on within the IDE. We also plan to conduct large-scale controlled experiments that involve software developers who will use our recommender system, when performing their software engineering and evolution tasks. In our future work, we will also investigate other informal documentation such as email discussions, forums, and bug reports, to gather more code examples for recommendations to developers.

APPENDIX I

TABLES OF RECOMMENDED CODE EXAMPLES

Table-A I-1 and Table-A I-2 show examples of natural language-based and API names-based queries. These tables have three columns, *i.e.*, Query, Rank and Recommended code examples. In Table-A I-1, the queries are natural language-based and in Table-A I-2, they are in Java API method name formats. In both tables, the Rank column denotes the algorithm type (*RH for Random Hyperplane-based LSH* and *QA for Query-Aware LSH*) and the rank of the first relevant example that was specified by two experienced software developers among which one is the author of this thesis and the second developer is not part of this research work.

Table-A I-1 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for Natural language-based queries

Query	Rank	Recommended code examples

How to add an image to a JPanel?	RH (Rank = 9)	[import java.awt.*;
		import java.awt.event.*;
		import javax.swing.*;
		public class EventTest {
		public static void main(String... args) {
		SwingUtilities.invokeLater(new Runnable() {
		@Override public void run() {
		final JComponent outside = new JPanel();
		JComponent inside = new JPanel();
		inside.setBackground(Color.red);
		inside.setPreferredSize(new Dimension(200, 200));
		inside.addMouseMotionListener(new MouseAdapter() {
		@Override public void mouseDragged(MouseEvent e) {
		System.err.println("dragged");
		}
		@Override public void mouseMoved(MouseEvent e) {
		System.err.println("moved inside");
		outside.dispatchEvent(e);
		} });
		outside.add(inside);
		outside.setPreferredSize(new Dimension(300, 300));
		outside.addMouseMotionListener(new MouseAdapter() { @Override
		public void mouseMoved(MouseEvent arg0) {
		System.err.println("moved outside"); }}); JFrame frame = new JFrame();
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.add(outside);
		frame.pack();
		frame.setVisible(true);
		}});}]

Continued on next page

Table-A I-1 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for Natural language-based queries (continued)

Query	Rank	Recommended code examples

	QA (Rank = 7)	[class MyPanel extends JPanel {
		ImageIcon icon;
		JTextArea textArea;
		MyPanel(ImageIcon icon, String text) {
		this.icon = icon;
		this.setPreferredSize(/*max size of your panel */);
		textArea = new JTextArea(10, 50);
		textArea.append(text);
		//the default manager will be a flow layout for single jpanels
		this.add(icon);
		JScrollPane sp = new JScrollPane(textArea);
		sp.setPreferredSize(new Dimension(/* size of your text label */));
		this.add(new JScrollPtextArea);
		} }
		class MyContainer extends JFrame {
		JPanel container;
		JScrollPane spContainer;
		MyContainer() {
		container = new JPanel()
		container.setGridLayout(100,1); //100 elements max
		spContainer = new JScrollPane(container);
		spContainer.setPreferredSize(/* max size of whole thing */);
		this.add(spContainer);
		pack(); }
		void addElement(MyPanel panel) {
		container.add(panel);
		this.pack();
		} }]

Continued on next page

Table-A I-1 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for Natural language-based queries (continued)

Query	Rank	Recommended code examples
How to generate a random alpha-numeric string?	RH (Rank = 0)	[No Recommended Code Example]

	QA (Rank = 1)	[import java.security.SecureRandom;
		public class StringUtils {
		private static final SecureRandom RANDOMSOURCE;
		private static String CANDIDATES = "ABCDEFGHIJKLMNOPQRSTU-VWXYZabcdefghijklmnopqrstuvwxyz0123456789";
		static {
		RANDOMSOURCE = new SecureRandom();
		}
		/** * Generate a random string of alphanumeric characters.
		* <;p>;
		* The string returned will contain characters randomly
		* selected from upper- and lower-case a through z as
		* well as the digits 0 through 9.
		* @param length the length of the string to generate
		* @return a string of random alphanumeric characters of the requested length */
		public static String generateRandomString(int length)
		{
		final StringBuffer sb = new StringBuffer(length);
		for (int i = 0; i &lt; length; i++) {
		sb.append(CANDIDATES.charAt(RANDOMSOURCE.nextInt(62)));
		}
		return sb.toString();
		} }]

Continued on next page

Table-A I-1 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for Natural language-based queries (continued)

Query	Rank	Recommended code examples

How do I create a file and write to it?	RH (Rank = 3)	[DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
		//build DOMs
		Document doc1 = builder.parse(new File("file1.xml"));
		Document doc2 = builder.parse(new File("file2.xml"));
		//get all node ids from doc2 and iterate
		NodeList list = doc2.getElementsByTagName("node_(id)");
		for(int i = 0 ; i < list.getLength() ; i++){
		Node n = list.item(i);
		//extract the id
		String id = n.getTextContent();
		//now get all node id elements from doc1
		NodeList list2 = doc1.getElementsByTagName("node_"+id);
		for(int j = 0 ; j < list2.getLength() ; j++){
		Node m = list2.item(j);
		//import them into doc2
		Node imp = doc2.importNode(m,true);
		n.getParent().appendChild(imp);
		} }
		//write out the modified document to a new file
		TransformerFactory tFactory = TransformerFactory.newInstance();
		Transformer transformer = tFactory.newTransformer();
		Source source = new DOMSource(doc2);
		Result output = new StreamResult(new File("merged.xml"));
		transformer.transform(source, output);]

Continued on next page

Table-A I-1 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for Natural language-based queries (continued)

Query	Rank	Recommended code examples

	QA (Rank = 4)	[public DDHLogger(String filePath, String yourfilename) {
		System.out.println("filePath: " + filePath);
		System.out.println("yourfilename: " + yourfilename);
		//String fileName = "C:\\Users\\ home-1 \\Desktop \\" + yourfilename;
		String fileName = filePath + yourfilename;
		System.out.println("fileName::–> " + fileName);
		// file = new File(fileName);//Creates the file
		file = new File(fileName);//Creates the file
		System.out.println("file: " + file);
		try {
		fw = new FileWriter(file, true);
		//INSTANTIATE PRINTWRITER HERE
		pw = new PrintWriter(fw);
		} catch (IOException e) {
		e.printStackTrace();
		}
		//allows append to the file without over writing. The TRUE keyword is used for append }]

Table-A I-2 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for API methods'names-based queries

Query	Rank	Recommended code examples[1]

Jpanel.add()	RH (Rank = 6)	[public JPanel createOscPanel(final Osc o) {
		JPanel cbPanel = new JPanel();
		final String[] comboItems = {"Sine", "Custom", "FSaw", "FSquare"};
		JComboBox cb = new JComboBox(comboItems);
		cbPanel.add(cb);
		cb.addActionListener(new ActionListener(){
		public void actionPerformed(ActionEvent e) {
		JComboBox cb = (JComboBox)e.getSource();
		int value = cb.getSelectedIndex();
		o.setWaveType();
		System.out.println(value);
		} });
		return cbPanel; }]

	QA (Rank = 4)	[frame.setLayout(new BoxLayout(frame.getContentPane(), BoxLay-out.Y_AXIS));
		JPanel panelMain = new JPanel(new GridLayout(0,2));
		ChartPanel chartPanel = createChart();
		JPanel subPanel = new JPanel(new BorderLayout());
		subPanel.setBorder(BorderFactory.createTitledBorder("Consommation"));
		subPanel.setPreferredSize(new Dimension(400, 200));
		subPanel.add(chartPanel);
		panelMain.add(subPanel);
		frame.add(panelMain);
		frame.pack(); .setLocationRelativeTo(null);
		frame.setVisible(true);]

Continued on next page

Table-A I-2 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for API methods'names-based queries (continued)

Query	Rank	Recommended code examples

StringBuilder.append()	RH (Rank = 10)	[private static final int POW10[] = 1, 10, 100, 1000, 10000, 100000, 1000000;
		public static String format(double val, int precision) {
		StringBuilder sb = new StringBuilder();
		if (val < 0) {
		sb.append('-');
		val = -val; }
		int exp = POW10[precision];
		long lval = (long)(val * exp + 0.5);
		sb.append(lval / exp).append('.');
		long fval = lval % exp;
		for (int p = precision-1; p > 0 && fval < POW10[p]; p–) {
		sb.append('0'); }
		sb.append(fval);
		return sb.toString(); }]

Continued on next page

Table-A I-2 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for API methods'names-based queries (continued)

Query	Rank	Recommended code examples

	QA (Rank = 1)	[public String center (String s, int length) {
		if (s.length() > length) {
		return s.substring(0, length); }
		else if (s.length() == length) {
		return s;
		}
		else {
		int leftPadding = (length - s.length()) / 2;
		StringBuilder leftBuilder = new StringBuilder();
		for (int i = 0; i &lt; leftPadding; i++) {
		leftBuilder.append(" "); }
		int rightPadding = length - s.length() - leftPadding;
		StringBuilder rightBuilder = new StringBuilder();
		for (int i = 0; i &lt; rightPadding; i++)
		rightBuilder.append(" ");
		return leftBuilder.toString() + s + rightBuilder.toString();
		} }]

Writer.write()	RH (Rank = 0)	[No Recommended code Example]

Continued on next page

Table-A I-2 First relevant recommended code examples (with their ranks) returned by the two considered algorithms for API methods' names-based queries (continued)

Query	Rank	Recommended code examples

	QA (Rank = 1)	[CsvListReader reader = new CsvListReader(new FileReader(inputCsv),
		CsvPreference.STANDARD_PREFERENCE);
		CsvListWriter writer = new CsvListWriter(new FileWriter(outputCsv),
		CsvPreference.STANDARD_PREFERENCE);
		List<String> columns;
		while ((columns = reader.read())= null) {
		System.out.println("Input: " + columns);
		// Add new columns
		columns.add(1, "Column_2");
		columns.add("Last_column");
		System.out.println("Output: " + columns);
		writer.write(columns);
		}
		reader.close();
		writer.close();]

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.

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

Andoni, A. (2004). E2lsh: Exact euclidean locality-sensitive hashing. *http://web. mit. edu/andoni/www/LSH/*.

Aytekin, A. M. & Aytekin, T. (2019). Real-time recommendation with locality sensitive hashing. *Journal of Intelligent Information Systems*, 53(1), 1–26.

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

Bacchelli, A., Cleve, A., Lanza, M. & Mocci, A. (2011). Extracting structured data from natural language documents with island parsing. *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 476–479.

Basili, V. R., Selby, R. W. & Hutchens, D. H. (1986). Experimentation in software engineering. *IEEE Transactions on software engineering*, (7), 733–743.

Basodi, S., Ji, C., Zhang, H. & Pan, Y. (2020). Gradient amplification: An efficient way to train deep neural networks. *Big Data Mining and Analytics*, 3(3), 196–207.

Basten, B., Hills, M., Klint, P., Landman, D., Shahi, A., Steindorfer, M. J. & Vinju, J. J. (2015). M3: A general model for code analytics in rascal. *2015 IEEE 1st international workshop on software analytics (SWAN)*, pp. 25–28.

Bawa, M., Condie, T. & Ganesan, P. (2005). LSH forest: self-tuning indexes for similarity search. *Proceedings of the 14th international conference on World Wide Web*, pp. 651–660.

Bettenburg, N., Premraj, R., Zimmermann, T. & Kim, S. (2008). Extracting structural information from bug reports. *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 27–30.

Broder, A. Z., Charikar, M., Frieze, A. M. & Mitzenmacher, M. (1998). Min-wise independent permutations. *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 327–336.

Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 380–388.

Collard, M. L., Decker, M. J. & Maletic, J. I. (2013). srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. *2013 IEEE International Conference on Software Maintenance*, pp. 516–519.

Dasgupta, A., Kumar, R. & Sarlós, T. (2011). Fast locality-sensitive hashing. *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1073–1081.

Datar, M., Immorlica, N., Indyk, P. & Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. *Proceedings of the twentieth annual symposium on Computational geometry*, pp. 253–262.

De Mulder, W., Bethard, S. & Moens, M.-F. (2015). A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, 30(1), 61–98.

Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Di Rocco, J., Di Ruscio, D., Di Sipio, C., Nguyen, P. T. & Rubei, R. (2021). Development of recommendation systems for software engineering: the CROSSMINER experience. *Empirical Software Engineering*, 26(4), 1–40.

Diamantopoulos, T. & Symeonidis, A. (2015). Employing source code information to improve question-answering in stack overflow. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 454–457.

Diamantopoulos, T., Karagiannopoulos, G. & Symeonidis, A. L. (2018). Codecatch: extracting source code snippets from online sources. *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pp. 21–27.

Diamantopoulos, T., Oikonomou, N. & Symeonidis, A. (2020). Extracting Semantics from Question-Answering Services for Snippet Reuse. *Fundamental Approaches to Software Engineering*, 12076, 119.

Ding, S. H., Fung, B. C. & Charland, P. (2016). Kam1n0: Mapreduce-based assembly clone search for reverse engineering. *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 461–470.

Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M. & Fahl, S. (2017). Stack overflow considered harmful? the impact of copy&paste on android application security. *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 121–136.

Fowkes, J. & Sutton, C. (2016). Parameter-free probabilistic API mining across GitHub. *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 254–265.

Frigge, M., Hoaglin, D. C. & Iglewicz, B. (1989). Some implementations of the boxplot. *The American Statistician*, 43(1), 50–54.

Gan, J., Feng, J., Fang, Q. & Ng, W. (2012). Locality-sensitive hashing scheme based on dynamic collision counting. *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pp. 541–552.

Grahne, G. & Zhu, J. (2003). Efficiently using prefix-trees in mining frequent itemsets. *FIMI*, 90, 65.

Gu, X., Zhang, H., Zhang, D. & Kim, S. (2016). Deep API learning. *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pp. 631–642.

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.

Holmes, R. & Murphy, G. C. (2005). Using structural context to recommend source code examples. *Proceedings of the 27th international conference on Software engineering*, pp. 117–125.

Huang, Q., Feng, J., Zhang, Y., Fang, Q. & Ng, W. (2015). Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1), 1–12.

Jafari, O., Maurya, P., Nagarkar, P., Islam, K. M. & Crushev, C. (2021). A survey on locality sensitive hashing algorithms and their applications. *arXiv preprint arXiv:2102.08942*.

Kim, K., Kim, D., Bissyandé, T. F., Choi, E., Li, L., Klein, J. & Traon, Y. L. (2018). FaCoY: a code-to-code search engine. *Proceedings of the 40th International Conference on Software Engineering*, pp. 946–957.

Kuo, F. Y. & Sloan, I. H. (2005). Lifting the curse of dimensionality. *Notices of the AMS*, 52(11), 1320–1328.

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.

Luan, S., Yang, D., Barnaby, C., Sen, K. & Chandra, S. (2019). Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 1–28.

Lv, Y. & Zhai, C. (2011). Adaptive term frequency normalization for BM25. *Proceedings of the 20th ACM international conference on Information and knowledge management*, pp. 1985–1988.

Marascuilo, L. A. & Serlin, R. C. (1988). *Statistical methods for the social and behavioral sciences.* WH Freeman/Times Books/Henry Holt & Co.

Moonen, L. (2001). Generating robust parsers using island grammars. *Proceedings eighth working conference on reverse engineering*, pp. 13–22.

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.

Nguyen, P. T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T. & Di Penta, M. (2019). Focus: A recommender system for mining api function calls and usage patterns. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 1050–1060.

Niu, H., Keivanloo, I. & Zou, Y. (2017). API usage pattern recommendation for software development. *Journal of Systems and Software*, 129, 127–139.

Raychev, V., Vechev, M. & Yahav, E. (2014). Code completion with statistical language models. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 419–428.

Rigby, P. C. & Robillard, M. P. (2013). Discovering essential code elements in informal documentation. *2013 35th International Conference on Software Engineering (ICSE)*, pp. 832–841.

Robertson, S., Zaragoza, H. et al. (2009). The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4), 333–389.

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

Rubei, R., Di Sipio, C., Nguyen, P. T., Di Rocco, J. & Di Ruscio, D. (2020). PostFinder: Mining Stack Overflow posts to support software developers. *Information and Software Technology*, 127, 106367.

Sanh, V., Debut, L., Chaumond, J. & Wolf, T. (2019). DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

Shen, Y. & Liu, J. (2021). Comparison of text sentiment analysis based on bert and word2vec. *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, pp. 144–147.

Shrivastava, A. & Li, P. (2014). In defense of minhash over simhash. *Artificial Intelligence and Statistics*, pp. 886–894.

Siegel, S. (1956). *Nonparametric statisticsfor the behavioral sciences.*

Silavong, F., Moran, S., Georgiadis, A., Saphal, R. & Otter, R. (2022). Senatus-A Fast and Accurate Code-to-Code Recommendation Engine. *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 511–523.

Subramanian, S. & Holmes, R. (2013). Making sense of online code snippets. *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 85–88.

Subramanian, S., Inozemtseva, L. & Holmes, R. (2014). Live API documentation. *Proceedings of the 36th international conference on software engineering*, pp. 643–652.

Thummalapenta, S. & Xie, T. (2008). Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 327–336.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

Wang, J., Dang, Y., Zhang, H., Chen, K., Xie, T. & Zhang, D. (2013). Mining succinct and high-coverage API usage patterns from source code. *2013 10th Working Conference on Mining Software Repositories (MSR)*, pp. 319–328.

Wilcoxon, F. (1992). Individual comparisons by ranking methods. In *Breakthroughs in statistics* (pp. 196–202). Springer.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B. & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Wright, H. K., Kim, M. & Perry, D. E. (2010). Validity concerns in software engineering research. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 411–414.

Xu, D. & Tian, Y. (2015). A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2), 165–193.

Zhang, K., Fan, S. & Wang, H. J. (2018). An efficient recommender system using locality sensitive hashing. *Proceedings of the 51st Hawaii International Conference on System Sciences*.

Zhong, H., Xie, T., Zhang, L., Pei, J. & Mei, H. (2009). MAPO: Mining and recommending API usage patterns. *European Conference on Object-Oriented Programming*, pp. 318–343.

Zhou, S., Shen, B. & Zhong, H. (2019). Lancer: Your code tell me what you need. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1202–1205.

Zou, J., Han, Y. & So, S.-S. (2008). Overview of artificial neural networks. *Artificial Neural Networks*, 14–22.