Improving Code Search Using Learning-to-Rank and Query Reformulation Techniques

by

Haoran Niu

A thesis submitted to the
Department of Electrical and Computer Engineering
in conformity with the requirements for
the degree of Master of Applied Science

Queen's University
Kingston, Ontario, Canada
July 2015

Abstract

During the process of software development, developers often encounter unfamiliar programming tasks. Online Q&A forums, such as StackOverflow, are one of the resources that developers can ask for answers to their programming questions. Automatic recommendation of a working code example can be helpful to solve developers' programming questions. However, existing code search engines support mainly keyword-based queries, and do not accomodate well natural-language code search queries. Specifically, natural-language queries contain less technical keywords, *i.e.*, class or method names, which negatively affects the success of the code search process of existing code search engines. On the other hand, a code search engine requires a ranking schema to place relevant code examples at the top of the result list. However, existing ranking schemas are hand-crafted heuristics where the configurations are hard to determine, which leads to the difficulty in using them for new languages or frameworks.

In this paper, we propose the approach which uses query reformulation techniques to improve the search effectiveness of existing code search engines for natural-language queries. The approach automatically reformulate natural-language queries using class-names with semantic relations. We also propose an approach to automatically train a ranking schema for the code example search using the learning-to-rank

technique. We evaluate the proposed approaches using a large-scale corpus of code examples. The evaluation results show that our approaches can effectively recommend semantically related class-names to reformulate natural-language queries, and the improvement on the search effectiveness over existing query reformulation approaches is statistically significant. The automatically trained ranking schema can effectively rank code examples, and outperform the existing ranking schemas by 35.65% and 48.42% in terms of normalized discounted cumulative gain (NDCG) and expected reciprocal rank (ERR), respectively.

Co-Authorship

I am the primary author of the studies presented in the thesis. Dr. Ying Zou supervised all my work and provided constructive feedbacks to the studies. Dr. Iman Keivanloo provided insightful guidelines on the studies described in Chapter 3 & 4. Feng Zhang gave me some constructive suggestions on the study described in Chapter 3. A poster based on the study of Chapter 4, titled "Learning to rank Code Examples for Code Search Engines", by Haoran Niu, Iman Keivanloo, and Ying Zou, has been awarded the best poster in the Consortium on Software Engineering Research (CSER) Fall meeting. The detail of the poster is listed as follows:

Haoran Niu, Iman Keivanloo, and Ying Zou. Learning to rank Code Examples for Code Search Engines. In the Consortium on Software Engineering Research (CSER) Fall meeting, Toronto, Nov. 2014. The link to the poster: https://www.dropbox.com/s/l70mwf9llp3qx1z/best%20poster.pdf?dl=0.

Acknowledgments

I would like to express the deepest appreciation to my supervisor, Dr. Ying Zou, for her persistent support and encouragement of my M.Sc study. She provided valuable guideline and help for my thesis research. Without her support and guideline, the journey could not have happened.

Besides my supervisor, I would like to thank Dr. Iman Keivanloo who gave me insightful guidelines, and assisted me patiently in solving the problems throughout my studies.

Thanks to my co-workers in the Software Reengineering Research lab for the friendly and cheerful working environment. I would like to thank Feng Zhang who provided useful suggestions for the tough problems I was faced with. I also extend my thanks to Dr. Bipin Upadhyaya, Hanfeng Chen for their help in the manual labeling work of the study.

I also thank Dr. Mei Nagappan and Ehsan Salamati who kindly shared the dataset used for the study presented in the thesis.

I appreciate the hard work from my committee members: Dr. Afsahi, Dr. Zulkernine, Dr. Kim and Dr. Ying Zou.

Last but not least, I would like to thank my parents for their great love, continuous support and encouragement during my life.

Glossary

API Application Programming Interface is a set of protocols, routines, and tools for building software applications.

CBOW Continuous Bag-of-Words Model is a neural network language model that predicts the word given its context.

ERF Explicit Relevance Feedback is a method where feedback is obtained from assessors expressing the relevance of a document retrieved for a query.

ERR Expected Reciprocal Rank is the expectation of the reciprocal of the position of a result at which a user stops. .

Learning-to-rank The application of machine learning in the construction of ranking models for information retrieval systems.

LOC The lines of code is a feature or metric of source code.

LSI Latent Semantic Indexing is an indexing and retrieval method in natural language processing.

NDCG Normalized discounted cumulative gain measures the performance of a recommendation system based on the graded relevance of the recommended entities.

Neural network language model A language model based on Neural Networks.

OR Odds ratio is used to quantify how strongly the presence or absence of a property is associated with the presence or absence of the other property in a given population.

PRF Pseudo Relevance Feedback is the method where the top retrieved documents are approximated as relevant documents to the queries.

Query reformulation A technique to improve the retrieval performance for text retrieval systems.

StackOverflow A language-independent collaboratively edited question and answer site for programmers.

Tf-idf Term frequency-inverse document frequency is used to reflect the importance of a word to a document in a corpus or collection.

VSM Vector Space Model is a model for representing text documents as vectors of terms, and is used in information retrieval area..

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

A code example is a source code snippet, *i.e.*, a few lines of source code, used to show how a specific programming task can be implemented [42]. In general, code examples play an important role in programming since they are existing solutions that can be used for learning [69] and reuse [39]. Developers rely on code examples to learn the correct way to use an unfamiliar library, framework, or Application Programming Interface (API) [69]. Moreover, code examples are commonly used by developers as a form of pragmatic code reuse [39], which involves copying and pasting code examples from external resources (*e.g.*, Web) into a product under development. Availability of code examples for learning and reuse can accelerate developers' productivity [5] and improve product quality [49].

Since it is not a common practice in software development to collect and document code examples [65], previously written projects and publicly available code repositories (*e.g.*, sourceforge.net) have become useful resources for code examples [42]. The process of locating code examples over available source code on the Web

to find relevant code snippets for software development is defined as code search [28]. More specifically, a code search query usually consists of API tokens, *i.e.*, class or method names. For example, For example, as shown in Fig. 1.1, if a developer wants to play sound using the API, *Clip*, he or she might use a code search engine, such as Open Hub[1], to search code examples that match with the query: "{Clip, start()}". Then the search process of code search engines exploits a corpus of code snippets to automatically extract the candidate code examples that match with the API tokens specified in the query. The candidate code examples are ranked based on the relevancy to the query. The study by Brandt *et al.* [8] reports that developers spend up to 20% of their time searching for code examples in the software development process. Searching for code examples can be realized by code search engines. To provide a better support for developers to find code examples, a plethora of code search engines (*e.g.*, [43][52][72]) are proposed to recommend code examples relevant to users' queries.

However, for a code search query, the code examples recommended by a code search engine do not have equal quality [80]. Effective code examples are expected to be concise, complete, and easy to understand [42]. Fig. 1.3 and Fig. 1.4 show two candidate code examples relevant to the query "$Clip, start()$". The first candidate answer (Fig. 1.3) provides a complete solution as a code example. At first, the developer should create an AudioInputStream object to obtain an audio input stream from the provided file. The format information of the audio resource can be extracted and stored into a DataLine.Info object. Then, the data line, *i.e.*, Clip, that matches with the description in the DataLine.Info object would be created, opened, and start to engage in audio data output. The queued data from the line is drained

[1]https://www.openhub.net/

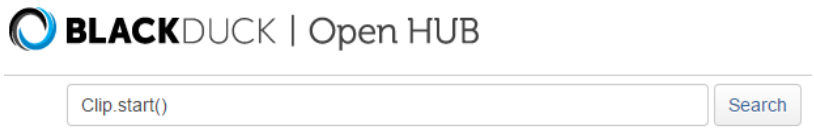

Figure 1.1: The query entered to the code search engine Open Hub for the task of playing sound

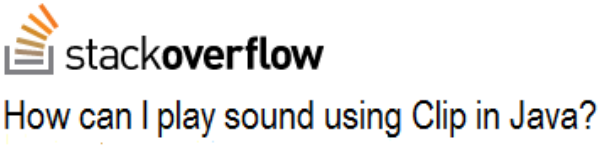

Figure 1.2: The question posted in StackOverflow for the task of playing sound

by continuing data output until the internal buffer has been emptied. Finally, the line is closed, and any system resources in use by the line would be released. However, the second candidate answer (Fig. 1.4) is not an effective code example since it is incomplete and contains some irrelevant code. For example, the second candidate answer does not show how to create the data line for a specified audio resource, and contains unnecessary code lines, such as setting the media position in sample frames.

Similar to Web search, developers prefer to receive effective code examples appearing toward the top of the ranked result list [12]. Therefore, the ranking capability plays an important role in the success of code search engines to rank the effective code examples at the top of the result list. Several ranking features, such as the textual similarity between code examples and a query [52][72], and the popularity of code examples [42][72], are proposed for ranking code examples. Earlier study [52] shows that it is not sufficient to use a single feature, *e.g.*, just the textual similarity between code examples and a query, for ranking since it would miss some other important information of the code examples, such as size, popularity, etc. When two or more features, *e.g.*, the textual similarity between code examples and a query and

```
Clip audioClip=null;
AudioInputStream audioInputStream=null;
try {
  audioInputStream=AudioSystem.getAudioInputStream(file);  }
catch (  IOException ioe) {
  ioe.printStackTrace();  }
if (audioInputStream != null) {
  AudioFormat format=audioInputStream.getFormat();
  DataLine.Info info=new DataLine.Info(Clip.class,format);
  if(AudioSystem.isLineSupported(info)){
    audioClip=(Clip)AudioSystem.getLine(info);
    audioClip.open(audioInputStream);
    audioClip.start();
  }
  audioClip.drain();
  audioClip.close();
}
else {
  project.log("Can't get data from file " + file.getName());
}
```

Figure 1.3: Effective code example for the query {Clip, start()}

```
if (clip == null) {    return false;  }
if (looping) {
  clip.loop(Clip.LOOP_CONTINUOUSLY);  }
else {
  clip.setFramePosition(0);
  clip.start();  }
return true;
```

Figure 1.4: Low-quality code example for the query {Clip, start()}

the popularity of code examples are combined, the ranking performance of the code search engines has improved. A ranking schema specifies how to combine the ranking features at run-time to produce the final ranked result set [50]. Essentially, a code search engine provides a ranking schema, which combines a set of ranking features to calculate the relevancy between a query and candidate code examples.

In addition to searching code examples using code search engines, developers also often turn to online Q&A forums to obtain the potential solutions for their programming tasks. An online Q&A forum is a place where developers can post programming questions, and rely on other fellow developers to provide potential answers [30]. Code

search questions in the online Q&A forums are mainly full-sentence programming questions that used to specify the task that the questioner attempts to accomplish. For example, as shown in Fig. 1.2, if a developer wants to accomplish the programming task described above, *i.e.*, playing sound using Clip, the question posted on a representative programming forum, *e.g.*, StackOverflow, by the developer might be, "How can I play sound using Clip in Java?", while the query entered to a search engine is "Clip, start()". We can see from Fig. 1.1 and Fig. 1.2 that the full-sentence code search questions posted in the online Q&A forums are different from the code search queries accepted by code search engines. Specifically, compared with the code search queries entered to the code search engines, full-sentence questions contain less technical keywords, *i.e.*, class or method names. To distinguish the full-sentence code search questions from the queries consisting of keywords, we denote them as natural-language and keyword-based code search queries, respectively.

1.2 Problem Statement

Online Q&A forums, such as StackOverflow, are one of the resources that developers can ask for answers to their programming questions. Our study on StackOverflow shows that code search questions account for the largest proportion, *i.e.*, 31%, of all the questions in StackOverflow. It is well-known in educational psychology that a working code example is an efficient mean to solve developers' programming questions since they can reuse the code examples directly instead of implementing them from the scratch [57]. Therefore, automatic recommendation of code examples for developers' code search questions would reduce the programming burden of developers to a great extent. However, the existing code search engines support mainly keyword-based code

search queries which are composed of class or method names, and do not accomodate well natural-language code search queries. Specifically, natural-language code search queries contain less API tokens (*i.e.*, class or method names), which negatively affects the success of the code search process of the existing code search engines. Therefore, to improve the effectiveness of the existing code search engines for natural-language code search queries, it is crucial to expand natural-language queries using related keywords [62].

The ranking schema is another important factor that affects the search effectiveness of code search engines. The ranking schemas used in the existing code search engines are mainly hand-crafted heuristics. The configurations of the existing ranking schemas (*i.e.*, the participating features or the weights of the participating features) are determined based on observations on a specific dataset, which demands the domain experts repeatedly to tune and evaluate the ranking schema. Therefore, we conjecture that it is difficult to apply the existing hand-crafted ranking schemas on different datasets. To improve the ranking capability of the existing code search engines, it is desirable to automatically build ranking schemas for the code search engines.

To summarize, the existing code search engines have two major limitations, which are listed as follows:

- Existing code search engines can not accommodate natural-language code search queries well due to the low proportion of API tokens in the queries.
- The ranking schemas used by the existing code search engines are mainly human-crafted heuristics where the configurations are hard to determine.

1.3 Thesis Objectives

In this thesis, we aim to address the two limitations of the existing code search engines listed above by applying query reformulation and learning-to-rank techniques, respectively.

The process of forming a new query, starting from the initial one, is defined as query reformulation. Query reformulation has been an effective way to improve the retrieval performance of text retrieval engines. In recent years, various query reformulation approaches have been proposed in the software engineering area. Those approaches can fall into two categories: interactive approaches and automatic approaches. Interactive approaches would place additional burden on the developers since it requires developers to provide the relevance feedback on the resulting documents. Automatic approaches reformulate a query by identifying the words that are semantically related to the query. To improve the performance of the existing code search engines for natural-language code search queries, we attempt to apply query reformulation technique to reformulate natural-language queries using semantically related keywords.

Learning-to-rank [45] is the application of machine learning algorithms in building ranking schemas for information retrieval systems. Binkley and Lawrie [7] find that learning-to-rank can provide universal improvement in the retrieval performance by comparing learning-to-rank with three baseline configurations of a search engine. Learning-to-rank has also been demonstrated to perform well in different tasks in the software engineering context, such as fault localization [76], duplication detection [83], feature traceability [7], etc. The significance of the findings is that a developer does not need to struggle with the configuration problem but can use learning-to-rank

to automatically build a ranking schema that performs better than the best configuration. In addition, Binkley and Lawrie [7] show that learning-to-rank is robust. The robustness is important because it means that the efforts of learning a ranking schema can be done once, and then the learned ranking schema can be applied to a new dataset without a significant loss of performance. Therefore, to address the limitation of the existing ranking schemas, we propose to apply leaning-to-rank to automatically build the ranking schemas for code example search.

Therefore, the objectives of the thesis are summarized as follows:

1. Reformulating natural-language code search queries using semantically related keywords

We automatically reformulate natural-language code search queries by identifying the most relevant keywords for the queries. The identification process of the keywords is based on the semantic relevances of the keywords to the queries. Top semantically related keywords are selected to reformulate a natural-language query based on the semantic distances between the query and the keywords.

2. Automatically building a ranking schema to rank code examples

We extract candidate code examples for queries to create training data. Then we apply a learning-to-rank algorithm to automatically learn a ranking schema from the training data. The learned ranking schema can be used to rank the candidate code examples for a new query at run-time.

1.4 Contribution

The thesis makes the following contributions:

- Propose an approach that enables the existing keyword-based code search engines accommodate natural-language queries. The evaluation results show the proposed approach can significantly improve the effectiveness of the existing code search engines for natural-language queries.

- Propose an approach that improves the ranking capability of code search engines. Based on the evaluation results, the proposed approach outperforms the existing ranking schemas in ranking code examples in the context of code search.

1.5 Organization of Thesis

The rest of the thesis is organized as follows:

- **Chapter 2 Related Work.** We present the studies related to current code search approaches, the application of query reformulation and learning-to-rank techniques in different areas.

- **Chapter 3 Reformulating Semantic-based Code Search Queries.** We propose an approach that can automatically reformulate natural-language code search queries using most relevant class-names which are identified by computing the semantic distances between the query and class-names.

- **Chapter 4 Learning to Rank Code Examples for Code Search Engines.** We study the application of learning-to-rank in code example search and automatically build ranking schemas for code search engines.

- **Chapter 5 Conclusion.** This chapter concludes the thesis by highlighting the contribution and potential directions of future work.

Chapter 2

Related Work

In this chapter, we discuss existing code search and recommendation engines, previous studies related to query reformulation techniques, and related applications of learning-to-rank techniques.

2.1 Code Search

2.1.1 Code Search Queries

To help developers solve programming tasks, lots of code search and recommendation approaches have been proposed to find candidate code examples relevant to a given query. Most of the approaches search through the source code of code examples to match keywords from queries to the names of program variables and types. Code search engines, such as Google Code[1] [24], Ohloh[2], can return code examples that contain user-specified keywords. Thummalapenta and Xie [72] develop an approach called PARSEWeb which accepts the queries of the form "Source → Destination" as input, and recommends relevant method-invocation-sequences (MISs) that contain

[1]https://code.google.com/
[2]https://www.openhub.net/

the objects of given types to implement an object transformation task. Another tool, Strathcona [38], locates relevant code examples using structural matching. The tool extracts the structural context of the code under development, including the method name being written and its containing class, as the query. McMillan *et al.* [32] propose a search engine called Exemplar to find highly relevant software projects that implement high-level concepts contained in the query.

Recently, code search for different types of queries has also been studied. Reiss *et al.* [68] design a semantics-based code search system, which aims to generate specific functions that meet users' specifications. The system needs the users to specify what they are searching for as precisely as possible using a combination of keywords (*e.g.*, class or method signatures) and dynamic specifications (*e.g.*, test cases and security constraints). Mishne *et al.* [54] propose a code search approach called PRIME which is capable of searching over partial code snippets to answer partial-code queries. PRIME uses sequences of API calls in the partial code as queries, which are matched against the temporal specifications extracted from partial code snippets.

Existing code search engines support mainly keyword-based queries, or demand users to provide specific information in the queries. In this thesis, we propose the approach which can answer natural-language code search queries by reformulating the queries with semantically related keywords.

2.1.2 Ranking for Code Search

Existing code search engines and recommendation systems adopt different ranking strategies to rank candidate code examples. For example, Google Code[3] [24] and

[3]https://code.google.com/

Ohloh[4] rank the returned code examples based on their textual similarities to the query. ParseWeb [72] uses the frequency and length of the recommended method-invocation-sequences(MISs) to obtain ranked result list. The code search engine Exemplar [32] adopts three different ranking schemes called word occurrences schema (WOS), dataflow connection schema (DCS) and relevant API calls schema (RAS) to sort the list of the retrieved applications. Mishne *et al.* [54] present a code search approach to answer queries focused on API-usage. The ranking of returned code snippets is achieved by counting the number of similar code snippets. Keivanloo *et al.* [42] propose a pattern-based approach for spotting working code examples. The approach considers the code snippet's similarity to the query, the popularity and line length of the code snippet to rank working code examples.

The above approaches use a single feature or propose some heuristics to combine the adopted features to rank software-related entities (*e.g.*, code example or software packages) relevant to the queries. As opposed to the existing work on ranking for code search, we automatically build ranking schemas for the code search engines using learning-to-rank techniques.

2.2 Query Reformulation

Query reformulation has long been an effective way to improve the results returned by text retrieval (TR) engines [59]. Many query reformulations approaches have been proposed in recent years. Those approaches can fall into two categories: interactive approaches and automatic approaches [34].

Interactive approaches mark the relevances of the documents returned to the initial

[4]https://www.openhub.net/

query and use the "best" terms in the relevant documents to augment the query for further retrieval. In other words, the interactive approaches are based on relevance feedback. In explicit relevance feedback (ERF), users are required to provide his/her judgments explicitly on the relevances of the returned documents to the query. Hayes *et al.* [36] build a requirement tracing tool RETRO which incorporates relevance feedback in the tracing process to improve the effectiveness of TR-based traceability link recovery. Gay *et al.* [29] use the relevance feedback mechanism to improve the IR-based approach for the concept location. Although ERF can provide the most accurate type of feedback, its use in source code retrieval is highly limited as it demands users' numerous efforts in marking the relevant and irrelevant documents. In comparison to ERF, Pseudo (Blind) Relevance Feedback (PRF) does not require user involvement. With PRF, the top set of the retrieved documents is considered as an approximation to the set of relevant documents. For example, Haiduc *et al.* [34] propose an approach which applies machine learning algorithms to recommend a better strategy among four query reformulation strategies. Three of them are based on some form of PRF. Obviously, PRF can only work if the initial query is reasonably strong so that at least some of the relevant documents can be retrieved.

Recently, some automatic query reformulation approaches [51][66][77] have been studied. The automatic approaches expand a query using the words that are automatically identified to be semantically related to the query. Sridhara *et al.* [67] perform an investigation on a few publicly available, English-based semantic-similarity techniques. The study shows that the techniques do not perform well in discovering semantically related words in software products since many words that are semantically related in English are not semantically related in software products. Therefore,

some approaches attempt to identify semantically related words directly from source code for queries. The identifying process of the existing automatic approaches are mainly based on the general statistical properties of a software library [66]. For example, Marcus *et al.* [51] use Latent Semantic Indexing (LSI) to determine the words that are more likely to co-occur in source code. obtained co-occurence information is later used to automatically recommend the semantically related words for the queries. Along the same lines, Yang *et al.* [77] automatically identify semantically related pairs of words using a pairwise comparison between the code segments and the comments in the source code. Sisman and Kak [66] identify the terms that are "close" to the initial query terms in the source code on the basis of positional proximity.

In this thesis, we propose an automatic approach to reformulate natural-language code search queries by identifying the semantically related words based on the semantic relevances of the words to the queries. The approach adopts a technique which can capture the semantic representations of words. Then the semantically related words can be identified by computing the semantic distances between the query words and the words in the source code.

2.3 Learning-to-Rank

As a relatively new application of machine learning techniques, learning-to-rank has received much attention in recent years. In the field of information retrieval, learning-to-rank is a class of machine learning algorithms which learn a ranking model from ordered documents in the training data for the information retrieval systems.

Recently, learning-to-rank has also been applied to specific problems related to software maintenance and evolution. Zhou and Zhang [83] have proposed a method,

BugSim, which uses learning-to-rank approach to automatically retrieve duplicate bug reports. The evaluation results show that their proposed method outperforms previous methods that are implemented using Support Vector Machine (SVM) model and extended Best Matching (BM25Fext) model. Xuan and Monperrus [76] propose Multric which locates fault position in source code by applying learning-to-rank. It is observed that Multric performs more effectively than existing approaches in fault localization. Ye *et al.* [79] introduce a learning-to-rank approach to rank source code files that might cause a specific bug. It shows that their proposed approach significantly performs better than two state-of-the-art methods, *i.e.*, standard Vector Space Model (VSM) method and Usual Suspects method, in recommending relevant files for bug reports. Binkley and Lawrie [7] explore the application of learning-to-rank in feature location and traceability. It demonstrates that learning-to-rank works well in the context of software engineering. Based on the results of applying learning-to-rank in different areas, we can observe that learning-to-rank is robust and is widely applicable.

In this thesis, we consider the code snippets in a source code corpus as documents, and a code search query as a query. Then we apply the learning-to-rank technique to automatically build ranking schemas for code example search.

2.4 Summary

In this chapter, we introduce the related studies about existing code search and recommendation engines, query reformulation and learning-to-rank techniques.

Chapter 3

Reformulating Natural-language Code Search Queries

By studying the questions posted in StackOverflow, we observed that code search questions account for the largest proportion of all the questions in StackOverflow. To answer the code search questions, it is an efficient way to automatically recommend working code examples for the questions. However, the existing code search engines accept mainly keyword-based queries, and do not accommodate code search questions (*i.e.*, natural-language code search queries) well. Specifically, natural-language code search queries contain less API tokens (*i.e.*, class or method names), which would negatively affect the success of the code search process of the existing code search engines. To improve the effectiveness of the existing code search engines to handle natural-language code search queries, it is crucial to reformulate the natural-language queries using relevant keywords.

In this chapter, we propose an approach which applies Query reformulation techniques to improve the search effectiveness of the existing code search engines for natural-language queries. The proposed approach reformulates natural-language code

search queries using the class-names that are semantically related to the queries. First, we discuss a preliminary study using StackOverflow questions. The observations of the preliminary study motivate us to propose the approach to automatically recommend semantically related class-names for natural-language code search queries. Then, we use a large-scale corpus of code snippets to evaluate the performance of the proposed approach in terms of recommending relevant class-names for natural-language queries. Finally, we discuss the threats to validity of our study.

3.1 Motivation Study

This section describes the preliminary studies to motivate both our research context (answering natural-language code search queries) and approach (reformulating natural-language queries using class-names).

3.1.1 RQ3.1: How prevalent are code search questions in StackOverflow?

Motivation. Developers always encounter different programming problems during software development. They rely on different sources, such as online programming Q&A websites (*e.g.*, StackOverflow) to seek answers to their programming questions. Recommending a working code example is an efficient way to answer developers' programming questions since developers can re-use the code examples directly [57]. In this research question, we aim to investigate the prevalence of code search questions in StackOverflow. If code search questions are prevalent, it is desirable to have an automatic approach (*e.g.*, a code search engine) that automatically recommends working code examples to answer the code search questions (*i.e.*, natural-language code search queries). Such code search engines can relieve developers' programming burden to a

great extent.

Approach. We extracted 725,492 questions tagged with "Java" posted in StackOverflow from August 2008 to October 2014, and then selected a sample of 384 questions with the confidence level of 95% and the confidence interval of 5 to reflect the question population precisely. The first author manually reviewed the 384 questions and attempted to classify them into different categories, similiar to Stolee *et al.* [68] who categorized the questions in SO into seven types. Since we are intended to investigate the questions related to "Java" (*i.e.*, with "Java" tags), one of the seven types of the SOquestions, *i.e.*, "Can I do (a specific task) with (other languages)", is not applicable to our study. Based on the definition of the remaining question types, we classify our studied StackOverflow questions into six categories:

- *Code Search*: Asking how to implement a programming task.
- *Concept Explanation*: Questions regarding some programming concepts or techniques.
- *Debugging*: Dealing with the run-time errors or unexpected exceptions.
- *Procedure*: Asking for providing a scenario to accomplish certain tasks that are not about programming.
- *Suggestion*: Asking the availability of certain tools.
- *Discussion*: Questions discussing performance or programming languages, etc.

Finally, we computed the proportion of the questions belonging to code search category to reflect the popularity of code search questions in StackOverflow.

Table 3.1: Summary of the distribution of different types of questions in our sample

Question Category	**Percentage(%)**
Code Search	31
Concept Explanation	7
Debugging	16
Procedure	27
Suggestion	10
Discussion	9

Result. Table 3.1 shows the distribution of different types of questions in the sample data. We can see from Table 3.1 that code search questions account for the largest proportion (*i.e.*, 31%) of the sample questions. The observation is in line with the result shown in Stolee *et al.* 's study [68]. Based on the principle of statistics [14], the distribution of the sample questions can approximately reflect that of StackOverflow question population. Therefore, we can conclude that code search questions are prevalent in StackOverflow, which becomes the first motivation for our study. In addition, we observe that the accepted answers of the code search questions in StackOverflow are mainly code fragments. Therefore, it is reasonable to recommend a single code snippet as the code example for a code search query.

Code search questions are prevalent in StackOverflow, and account for the largest proportion of SO questions, i.e., 31%.

3.1.2 RQ3.2: Does the success of Koders depend on the existence of API tokens?

Motivation. The existing code search engines, such as Koders[1], mainly accept keyword-based code search queries. We wonder whether the existing code search

[1]Koders: http://code.openhub.net/

engines can accommodate natural-language code search queries well. To this end, it is necessary to figure out the factor that contributes significantly to the search effectiveness of the existing code search engines. As defined in the introduction section, keyword-based code search queries are usually composed of class or method names. Therefore, we conjecture that the existence of API tokens (*i.e.*, class or method names) in the queries would affect the success of the searching process of the existing keyword-based code search engines. In this research question, we aim to study whether the existence of API tokens (*i.e.*, class or method names) contributes significantly to the success of keyword-based code search engines. If the existence of API tokens is a key factor to the search performance, then the keyword-based code search engines might not support natural-language code search queries very well.

Approach. Koders is the typical code search engine that mainly accepts keyword-based code search queries. Therefore, we use Koders to represent the existing code search engines. We download a year-long usage log data [2] of Koders.com, which contains 628,862 code search queries. Then, we split the queries into two groups: the first group contains at least one API token in the query, the queries without any API tokens belong to the second group. For each query in the two groups, we can obtain the information whether the result provided by Koders satisfies the query's owner or not using clicking analysis [2]. Clicking analysis is based on the fact that if a correct answer was found in the result list provided by Koders, it should have resulted in clicking on the correct answer to view it.

A Chi-Squared test is used to check if there is a relationship between two categorical variables [33]. We conduct Chi-Squared test to determine if there is a significant difference between the two groups in terms of being answered or not by Koders. We

Table 3.2: Summary of the number of queries that have been (not) answered in two groups

	Answered	**Not answered**
At least one API token exists in query	5,089 (0.8%)	15,864 (2.5%)
No API token exists in query	76,839 (12.2%)	531,026 (84.4%)

use the confidence level of 95% (*i.e.*, p-value < 0.05) to determine the significance of the comparison result.

We compute odds ratio (OR) [63] to quantify whether the existence of API tokens is a key factor to the success of Koders. Odds ratio denotes the ratio of the odds of an event occurring with the presence of a factor to the odds of an event occurring with the absence of the factor. $OR = 1$ indicates that the existence of API tokens does not matter to the success of Koders; $OR > 1$ shows that the existence of API tokens is a key factor that contributes to the success of Koders and $OR < 1$ means that the existence of API tokens is a key factor that prevents the success of Koders.

Result. Table 3.2 shows the number of queries that have been answered and have not been answered in the two query groups. The p-value of the Chi-Squared test in terms of being answered or not by Koders is less than 2.2e-16, which indicates that the difference between the two groups in terms of search performance is significant. Based on Table 3.2, we can obtain the odds ratio of 2.22, which is larger than 1 and indicates that the existence of API tokens is a key factor to the success of Koders. Therefore, based on the analysis of 628,862 real-word keyword-based code search queries, we observe that missing relevant API tokens in code search queries significantly reduces the success of code search using keyword-based code search engines.

The existence of API tokens (i.e., class or method names) in the code search queries is a key factor to the success of existing code search engines.

3.1.3 RQ3.3: Are natural-language and keyword-based code search queries different in terms of API tokens?

Motivation. We can observe from RQ3.2 that the existence of API tokens in the code search queries is an important factor that affects the success of the existing code search engines. To study whether the existing code search engines can accommodate natural-language queries well, in this research question, we aim to identify whether natural-language and keyword-based code search queries are significantly different in terms of the proportion of API tokens.

Approach. Similiar to Stolee *et al.* 's study [68], we use the StackOverflow questions belonging to the code search group, as discussed in RQ3.1, to represent natural-language code search queries, totaling 118 queries. To obtain keyword-based code search queries, we randomly select queries from Koders data [2] used in RQ2 with the confidence level of 95% and the confidence interval of 5. Then, we compare the natural-language and keyword-based code search queries in terms of API density. API density measures the proportion of API tokens (*i.e.*, class or method names) in the natural-language or keyword-based code search queries. We first remove stop words (*e.g.*, a, the, is, etc.) in natural-language or keyword-based code search queries and tokenize them into tokens. Then we manually identify the API tokens in natural-language code search queries, and identify the API tokens in keyword-based code search queries by checking whether the query contains "mdef" or "cdef" which are search commands of Koders engine to specify method and class tokens in a query.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data [63]. We conduct Mann-Whitney U test with 95% confidence level (*i.e.*, p-value < 0.05) to determine if the observed differences in the API density

of natural-language and keyword-based code search queries are significant.

Result. Fig. 3.1 shows the comparison result of API density. As shown in Fig. 3.1, the API density of natural-language queries is less than that of keyword-based queries noticeably, and the p-value of API density comparison is 2.65e-03. Therefore, we can conclude that the natural-language and keyword-based code search queries are significantly different in terms of API density. Existing code search engines might not accommodate natural-language queries well. The observation specifically supports our interest in enriching natural-language code search queries with the related keywords as a form of query reformulation.

Natural-language queries are significantly different from keyword-based queries in terms of the proportion of API tokens (i.e., class or method names).

In summary, the existence of API tokens in queries contributes significantly to the success of the existing keyword-based code search engines. However, the proportion of API tokens in the natural-language code search queries is low, which negatively affects the search effectiveness of the existing code search engines for natural-language queries. Therefore, the existing code search engines can not accommodate natural-language code search queries well. To solve the problem, we propose an approach to automatically reformulate natural-language queries with the semantically related class-names.

3.2 Our Proposed Approach

In this section, we first briefly describe neural network language model used in our approach. Then we present the details of our approach.

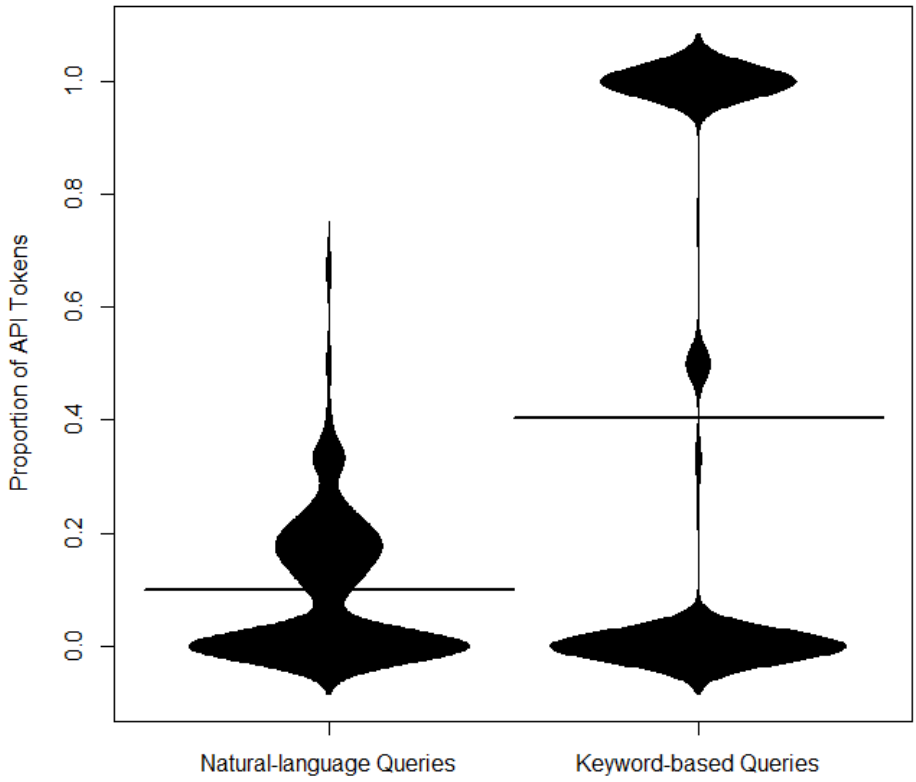

Figure 3.1: API density comparison between natural-language and keyword-based code search queries

3.2.1 Neural Network Language Model

A language model assigns a probability to a sequences of t words $p(w_1, ..., w_t)$ [50]. For example, the probability of the sentence, "The girl is running", can be denoted as p(The, girl, is, running). An n-gram model [11] is a popular language model which predicts the next word w_t (*e.g.*, running) based on its prior context $w_{t-(n-1)}, ..., w_{t-1}$ (*e.g.*, The, girl, is), which is a sequence of n consecutive words called n-gram. In an n-gram model, the probability of a word sequence $p(w_1, ..., w_t)$ is approximated as $p(w_1, ..., w_t) = \prod_{i=1}^{T} p(w_i|w_{i-(n-1)}, ..., w_{i-1})$. For example, in a trigram ($n = 3$) language model, the approximation for the sentence *The girl is running* would be p(The, girl, is, running)=p(The| <s>, <s>)p(girl| <s>, The)p(is|

The, girl)p(running| girl, is)p(<\s> | is, running). The conditional probabilities can be estimated from the raw text based on the relative frequency of word sequences, *i.e.*, $p(w_i|w_{i-(n-1),\ldots,w_{i-1}}) = \frac{count(w_{i-(n-1)},\ldots,w_{i-1},w_i)}{count(w_{i-(n-1)},\ldots,w_{i-1},\cdot)}$ where the counts can be obtained from a training corpus. To train the model to be more realistic, a larger n is desired. However, for a large value of n, it is likely that a given n-gram will not have been observed in the training corpus.

Neural network language model [6] is proposed to model the conditional probability $p(w_i|w_{i-(n-1),\ldots,w_{i-1}})$ with a neural network. The neural network language model learns vector representations of words to generalize n-grams that are not seen in the training corpus. For example, for the conditional p(running|the, girl, is), assuming that the 4-gram (the, girl, is, running) is not in the training corpus, but (the, boy, is, running) is in the training corpus, then "girl" can be generalized to "boy" if the word representations of "girl" and "boy" are similar. Based on other 4-grams in the corpus, *e.g.*, (the, girl, was, walking) and (the, boy, was, walking), the neural network could learn similar word representations for "girl" and "boy". Each dimension of the vector representations corresponds to a semantic or grammatical characteristic of words. Therefore, the basic idea of the neural network language model is to map the words with vector representations so that the two words that are functionally similar have similar vector representations [6].

Mikolov *et al.* [53] propose a new neural network model called continuous Bag-of-Words Model (CBOW). CBOW can train high-quality word vectors using simple model architectures with lower computational complexity. In this study, we use CBOW to learn the vector representation of each word in the training corpus. Therefore, a set of words is translated into a set of vector representations which capture

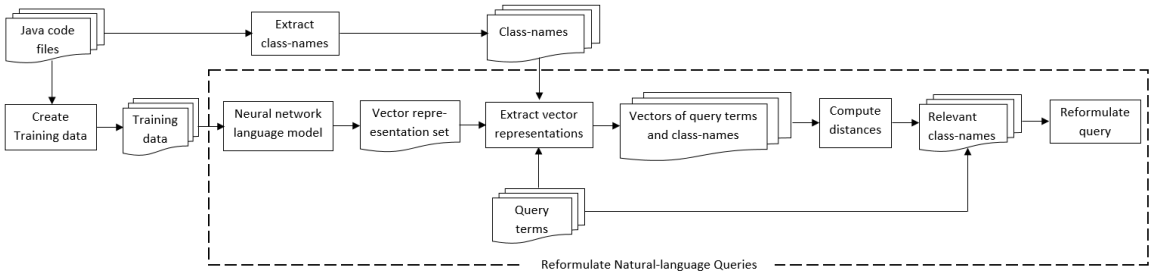

Figure 3.2: The process of reformulating natural-language queries

the semantic information. The semantic similarity between words can be quantified using the similarity measures (*e.g.*, Cosine Similarity) between word vectors.

3.2.2 Approach Details

The proposed approach contains three major phases. We first process code snippet corpus to obtain the training data, *i.e.*, a corpus of terms extracted from the source code. Then we learn the semantic representations of the terms in the training data and reformulate natural-language queries using semantically related class-names. The first two phrases are illustrated in Fig. 3.2. Finally, we execute the reformulated queries to retrieve code snippet lists from the code snippet corpus. We discuss the details of each phase in the following paragraphs.

Before creating the training data, we have to obtain the names of the classes used in the Java code files since our reformulation approach only uses the class-names to reformulate natural-language queries. Import statements specify classes used in a specific Java file. A class specified in an import statement has a full name which comprises of the package-name and the class-name. In our study, we parse the import statement of all Java files to extract the class-names. For each class-name, we can obtain its popularity, *i.e.*, the number of Java files using the class.

1) Creating Training Data

For each code snippet in the corpus, we tokenize the source code of the code snippet using camel case splitting that has been used in relevant studies [44][66]. We keep the class-names used in the code snippet intact. Then, we normalize the tokenized terms by removing stop words and stemming. We remove common English stop-words (*e.g.*, a, the, is, etc.) and programming keywords (*e.g.*, if, else, for, etc.) in the process of stop words removal. For the stemming process, we use the Porter stemmer [50]. All the normalized terms are included into the training data.

2) Reformulating Semantic-based Queries In our study, we use Word2vec[2] [53] to generate vector representations for the terms in the training data. Word2vec provides an efficient implementation of the continuous bag-of-words model. It takes training data (a term corpus) as the input and produces a set of term vectors as the output. The resulting term vectors include all the terms contained in the training data. Each element of the term vectors is a float value, which reflects one semantic feature of the terms.

For each natural-language query, we perform the same text normalization process adopted for obtaining the terms in the source code (*i.e.*, camel case splitting, stop words removal, and stemming). We use a vector $V_q = (t_1, t_2, ..., t_i, ..., t_m)$ to represent a normalized natural-language query, where m is the total number of terms in the normalized query, and t_i represents the i_{th} term. For example, the natural-language query, "How can I play sound using Clip in Java?", can be represented by the vector V_q =(plai, sound, us, clip). Since our training data is created using a large-scale code snippet corpus, the terms contained in the normalized natural-language queries can also be found in the training data. Therefore, for each term t_i in the normalized query, we extract its term vector representation (*i.e.*, a float vector) from the generated term

[2]word2vec: http://code.google.com/p/word2vec/

vector set and denote it as $t_i = (f_{i1}, f_{i2}, ..., f_{ii}, ..., f_{in})$, where n is the total number of the vector elements, and f_{ii} is the i_{th} vector element, which is a float value. Then, the vector representation of the normalized natural-language query can be obtained by adding the corresponding elements in the vector of each term contained in the query, *i.e.*, $T_q = \sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (f_{i1}, f_{i2}, ..., f_{ii}, ..., f_{in})$, which is also a float vector. For example, the vector representation of the query, "How can I play sound using Clip in Java?", can be obtained by adding the vectors of corresponding normalized query terms, *i.e.*, "plai", "sound", "us", and "clip".

Similarly, the vector representation of each class-name extracted in the first phase can also be obtained from the term vector set. We denote the vector representation of a class-name (*i.e.*, a float vector) as $T_c = (f_1, f_2, ..., f_i, ..., f_n)$. The semantic distance between a natural-language query q and a class-name c can then be computed using cosine similarity [61] between their vector representations as follows:

$$D_{q,c} = \frac{T_q \cdot T_c}{|T_q||T_c|} \tag{3.1}$$

Where T_q is the vector representation of the natural-language query q, T_c is the vector of the class-name c.

For a natural-language code search query, we can obtain the semantic distances between the query and all the extracted class-names using Eq. 3.1. In addition to the semantic distances, the popularity of the class-names, *i.e.*, the number of Java files using the class in the corpus, is another important feature needed to be considered when recommending class-names since the higher popularity of a class has, the more widely-used the class is to accomplish a programming task. We filter the less-popular class-names by discarding user-defined ones, *i.e.*, the classes that are not contained

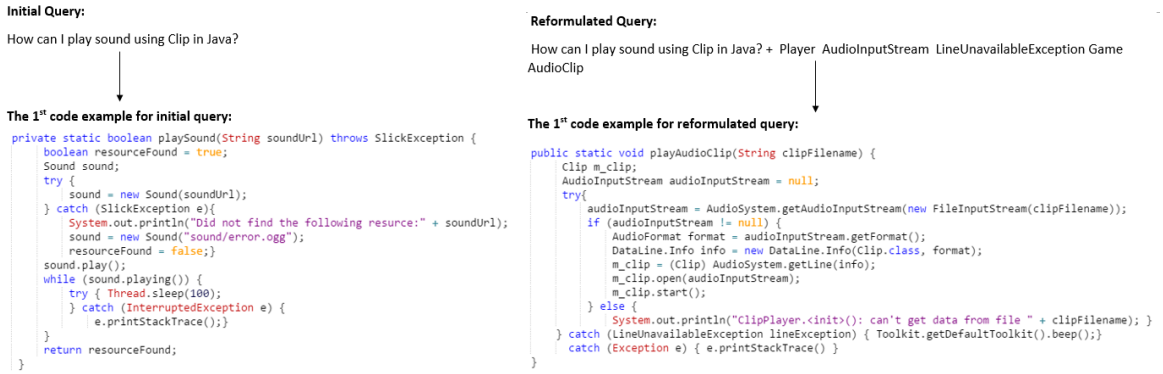

Figure 3.3: The first code examples returned for the initial and reformulated queries in Java API documentation. Then, we rank the remaining class-names based on the computed semantic distances and then re-rank the top 10 class-names based on their popularities [42]. Finally, we use the top k class-names in the ranked class-name list to reformulate the natural-language code search queries. In our implementation, we set k to 5 after attempting different numbers, such as 1, 3, 5, 7, 10, in order to balance the completeness and redundancy of useful class-names. Specifically, recommending few class-names (*e.g.*, top 1 class-name) to a query might not locate the most relevant class-name, which would lead to the incompleteness of recommended class-names and make the query reformulation hard to improve the searching performance. On the other hand, recommending many class-names (*e.g.*, top 10 class-names) to a query would include the irrelevant class-names, which lead to the redundancy of recommended class-names and might make the relevant code examples placed at the bottom. As shown in Fig. 3.3, the top 5 class-names that are used to reformulate the query, "How can I play sound using Clip in Java?" are "Player AudioInputStream LineUnavailableException Game AudioClip".

3) Executing Reformulated Queries We use the Vector Space Model (VSM) technique [50] to retrieve relevant code examples for reformulated code search queries.

The VSM technique also reports the similarity scores of the retrieved code examples to the query. To make the desired code examples as close as possible to the top of the result list, it is not sufficient to rank based on the similarity between the code example and the query [52]. In our study, we rank the returned code examples by applying two common ranking schemas: weighted-sum ranking schema [32] and re-ranking schema [42].

The weighted-sum ranking schema computes the ranking score of a candidate code example using a weighted sum of different features of the code example with equal coefficients [32]. In our study, we consider 5 code example features denoted as f_r, f_p, f_s, f_u and f_a respectively. f_r is the similarity score between the query and the code example reported by a VSM technique, *i.e.*, Lucene[3], along with the code example. f_p denotes the sum of the frequencies of each line of code of the code example in the corpus [22]. f_s is the cosine similarity between the query and the signature of the code example. f_u denotes whether the recommended class-names used for reformulation appear in the parameters of the code example. If the answer is yes, f_u is set to -1, otherwise, it is 0. f_a is the number of the parameters of the code example. As shown in Eq. 3.2, the ranking score of a code example is computed as the weighted sum of its five features. In our implementation, we assign equal weights to each feature (except for f_p). We set the weight of f_p twice as large as other weights since ranking code examples based on the feature f_p performs better than other features [42].

$$S = wf_r + 2wf_p + wf_s + wf_u + wf_a \tag{3.2}$$

Where f_r, f_p, f_s, f_u, f_a are the five features of a code example, and w is the weight

[3]Lucene: https://lucene.apache.org/

for the features.

Re-ranking schema re-orders the top-k candidate code examples determined by the primary feature using a secondary feature [42]. Since developers are more interested in top 10 answers, we set k to 10. We apply re-ranking schema by re-ranking top 10 code examples in the result list determined by weighted-sum ranking schema. We select the top 10 code examples from the result lists returned for the reformulated queries and the initial queries, and represent them as L_r (reformulated queries) and L_i (initial queries) respectively. For each code example c in the code example list L_r, we check whether it occurs in code example list L_i. The code examples in the list L_r would be re-ranked using the following rule:

$$p_c = \begin{cases} p_r + p_i, & if\ code\ example\ c\ occurs\ in\ L_i \\ p_r + (k+1), & otherwise. \end{cases} \tag{3.3}$$

Where L_i is the list containing the top 10 code examples for the initial query after applying weighted-sum ranking; L_r is the list containing the top 10 code examples for the reformulated query after applying weighted-sum ranking; c is a code example in the list L_r; p_c is the position of code example c in the final result list, p_i is the position of code example c in the list L_i if it occurs in list L_i; p_r is the rank of code example c in list L_r.

After applying the weight-sum ranking schema and re-ranking schema, we can obtain the final ranked list of relevant code examples for the reformulated queries. For example, we denote the code example list for an initial query and the corresponding reformulated query after applying weighted-sum ranking schema as $s_{i1}, s_{i2}, ..., s_{in}$ (for initial query) and $s_{r1}, s_{r2}, ..., s_{rm}$ (for reformulated query), respectively. Then, we

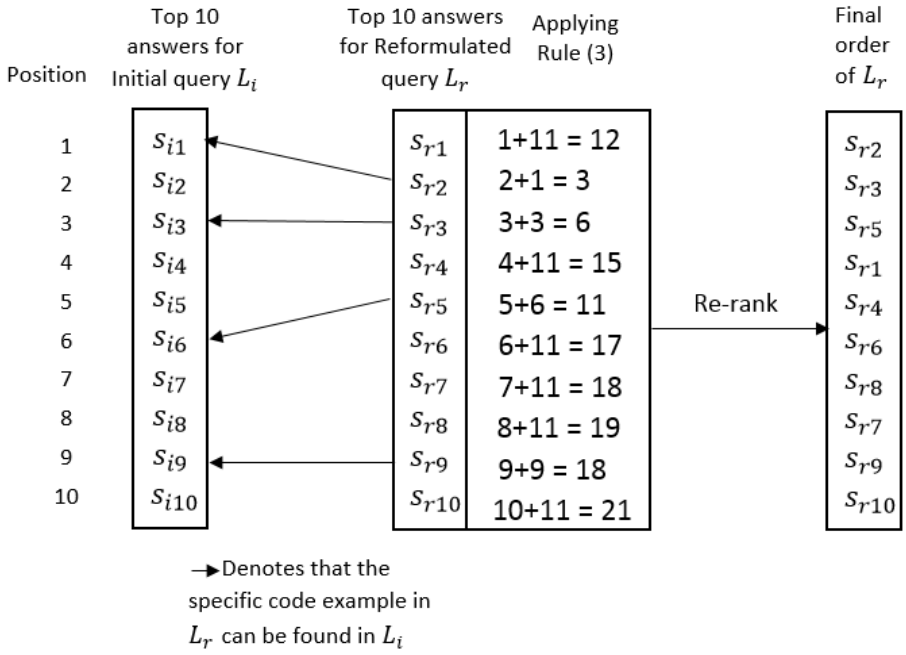

Figure 3.4: Illustration of applying re-ranking schema

re-rank the top 10 code examples in the result list for reformulated query, *i.e.*, L_r : $s_{r1}, s_{r2}, ..., s_{ri}, ..., s_{r10}$, by checking whether s_{ri} can be found in the top 10 result list for the initial query, $Li : s_{i1}, s_{i2}, ..., s_{i10}$. As illustrated in Fig. 3.4, assuming that the positions of $(s_{r1}, s_{r2}, ..., s_{ri},$

$..., s_{r10})$ in the list $s_{i1}, s_{i2}, ..., s_{i10}$ are (NA, 1, 3, NA, 6, NA, NA, NA, 9, NA), where NA means the code example can not be found in the list. Then the position of $s_{r1}, s_{r2}, ..., s_{r10}$ would be computed as (1+11, 2+1, 3+3, 4+11, 5+6, 6+11, 7+11, 8+11, 9+9, 10+11) = (12, 3, 6, 15, 11, 17, 18, 19, 18, 21) using the rule defined in Eq. 3.3. Therefore, the final order of the list $s_{r1}, s_{r2}, ..., s_{rm}$ would be re-ranked as$(s_{r2}, s_{r3}, s_{r5}, s_{r1}, s_{r4}, s_{r6}, s_{r8}, s_{r7}, s_{r9}, s_{r10}, s_{r11}, s_{r12}, ..., s_{rm})$.

3.3 Case Study Setup

The goal of the case study is to evaluate the performance of our proposed approach in recommending the relevant code examples for the natural-language code search queries. For the case study, we need a corpus of code snippets, a benchmark and the comparison baselines. In this section, we describe the process of creating the corpus of code snippets, the detail of the benchmark, and the brief summary of baseline approaches.

3.3.1 Corpus

To retrieve code examples for code search queries, we have to prepare a large-scale corpus of code snippets. To build the corpus, we download a big inter-project repository IJaDataset[4] created by Keivanloo *et al.* [42] and later used by Svajlenko *et al.* [70] to build the benchmark for clone detection. The dataset covers 24,666 open-source projects from SourceForge and Google Code, and 2,882,451 unique Java files exist in the open-source projects. To extract code snippets from the Java files, the dataset uses the same approach as the earlier work on code search and recommendation for Java source code [54]. The approach uses a Java syntax parser[5] available in Eclipse JDT to extract one code snippet from each single method in the Java files. Finally, 382,073 code snippets are extracted and added into the code snippet corpus. Table 3.3 and Fig. 3.5 summarizes the key descriptive statistics of the corpus.

[4]IJaDataset: https://github.com/clonebench/BigCloneBench
[5]The parser from Eclipse JDT: http://www.eclipse.org/jdt/

Table 3.3: Summary of our corpus

Feature	Number
Num. of Projects	24,666
Num. of Java Files	2,882,451
Num. of Code Snippets (LOC. $\geq$ 5)	382,073
Total LOC of Code Snippets (LOC. $\geq$ 5)	6,670,395

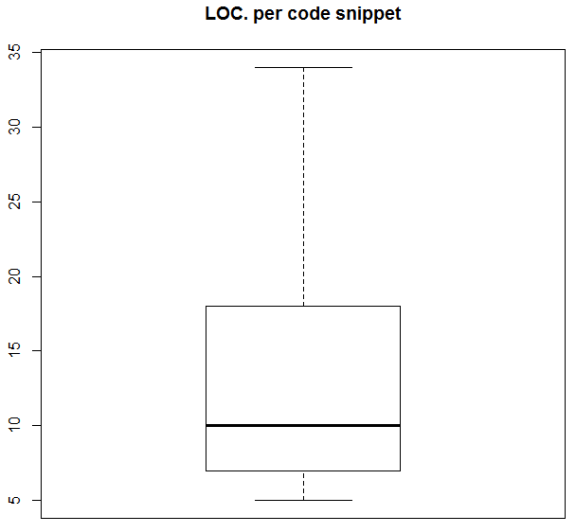

Figure 3.5: Detailed summary of the LOC. of code snippets

3.3.2 Benchmark

To evaluate the performance of our approach in recommending the relevant code examples for the natural-language code search queries, we need to identify a set of correct code examples for the natural-language queries. The set of correct code examples constitutes the "gold set" used for evaluation. In the following paragraphs, we describe the process of creating the "gold set".

To create the "gold set", we use the benchmark data created for clone detection by Svajlenko *et al.* [70]. In Svajlenko *et al.* 's study, thousands of code snippets have been tagged as true or false positives of 10 distinct functionalities (hereby, target functionalities) that are frequently used in open-source Java projects. The tagging

process has been done by three different judges in the study. In our study, we consider the set of true positives, *i.e.*, the code snippets that correctly implement the target functionality, as the "gold set" for the target functionalities. Now, the number of target functionalities is expanded to 40 in the new version of the benchmark dataset[6].

For each target functionality, we locate its corresponding natural-language code search query from StackOverflow. The benchmark data contains the specification of the implementation of each target functionality, therefore, we select the natural-language query from StackOverflow for a target functionality by checking whether the accepted answer of the StackOverflow question exactly matches the implementation specification of the target functionality. In our study, query reformulation aims to improve search results by expanding the initial queries with class-names, the query reformulation for the queries that do not require APIs in the implementation would make no differences on the search performance. Therefore, the trivial queries that do not use APIs in the implementation are not suitable to be used to evaluate our approach. The authors manually review the 40 target functionalities in the benchmark [70] and discard the trivial queries. Finally, 24 out of 40 natural-language code search queries are selected to evaluate the performance of our approach. We measure the search performance of our approach by identifying the position of the first correct answer in the ranked result list according to the "gold set".

We created a second "gold set" for additional 26 natural-language queries selected from StackOverflow. The correct answers in the "gold set" are identified using the pooling approach [50], which was previously used by Bajracharya *et al.* [3] to evaluate the code example retrieval schemas. In the pooling approach, a set of candidates is collected in a pool by retrieving top k results from each retrieval approach to be

[6]Benchmark dataset: https://github.com/XBWer/BigCloneBench

evaluated. In our evaluation, we set k to be 10 considering the fact that most search engines present top 10 results in the first result page [3]. Therefore, the top 10 code examples returned by Dice, Rocchio, RSV, and our approach, are collected into a pool to be checked for correctness. For a query, if no relevant code example is found in the top 10 results, we explore further to check the remaining code examples. The number of the code examples returned by each retrieval approach is numerous, so it is impossible to check the correctness for all the remaining code examples. Therefore, we first figure out the specific terms that are necessary in the query implementation. Then only the remaining code examples that contain the specific terms are checked for correctness. The correctness checking process of the candidate answers in the second "gold set" has been done by three judges, who are graduate students with more than five years' Java development experience. The majority rule is applied when there is disagreement on the correctness of the code examples.

Using the two "gold sets", the number of queries meets the minimum requirement (*i.e.*, at least 50 queries) for the evaluation of search engines [50]. The 50 natural-language code search queries in the two "gold sets" can be found in the replication package.

3.3.3 Comparison Baselines

A variety of query reformulation approaches have been proposed in the field of text retrieval area [20]. Three approaches [26][59][81] are demonstrated to perform the best using SE data [34]. In the three approaches, the top k code examples in the result list are considered to be relevant results to a query. Then, the terms appearing in the top k code examples are ranked by a ranking strategy. Finally, the top n

terms are selected for query reformulation. Similar to Haiduc *et al.* 's study, we set k to 5, and n to 10 in our study. We compare our approach with the three query reformulation approaches. The strategies used in the three approaches are described as follows.

Dice

Dice ranks the terms in the top k code examples based on their *Dice* similarity with query terms. *Dice* similarity [26] can be computed as:

$$Dice = \frac{2df_{q \cap s}}{df_q + df_s} \tag{3.4}$$

Where q denotes a term contained in a query, s is a term extracted from the top k code examples, and df denotes the number of code examples in the corpus that contain q, s, or both q and s, respectively [26].

Rocchio

Rocchio orders the terms in the top k code examples using Rocchio's method [59]. As shown in Eq. 3.5, it computes a score for each term based on the sum of the $tfidf$ scores of the term in the top k code examples. Term frequency (tf) is the number of times that the term appears in a code example, and it indicates the importance of the term over other terms in the code example [50]. The inverse document frequency (idf) indicates the specificity of the term to a code example containing it, and can be considered as the inverse of the number of the code examples in the corpus thats contain the term [50]. The $tfidf$ score of the term can be computed by multiplying

the tf value by the idf value of the term.

$$Rocchio = \sum_{c \in R} tfidf(t, c) \tag{3.5}$$

Where R is the set containing the top k code examples in the result list; c is a code example in R; t is a term in the code example c [59].

RSV

RSV uses the Robertson Selection Value (RSV) [81] as a scoring function for the terms in the top k code examples.

$$\begin{aligned} RSV &= \sum_{c \in R} tfidf(t, c)\,[p(t|R) - p(t|C)] \\ p(t|R) &= freq(t\,in\,R)/number\,of\,terms\,in\,R \\ p(t|C) &= freq(t\,in\,C)/\,number\,of\,terms\,in\,C \end{aligned} \tag{3.6}$$

Where C denotes code snippet corpus, R is the set of top k code examples, c is a code example in R, and t is a term in code example c. $freq(t\,in\,R)$ means the number of times that term t appears in the top k code examples, and $freq(t\,in\,C)$ is the number of times that t appears in the corpus [81].

3.4 Research Questions and Results

In this section, we discuss the results of our case study. We describe the motivation, analysis approach, and findings for each research question.

3.4.1 RQ3.4: Is our approach effective in answering natural-language code search queries?

Motivation. The findings of the motivation study show that the low proportion of API tokens (*i.e.*, class or method names) in the natural-language code search queries would negatively affect the success of the existing code search engines. To improve the search performance of the existing code search engines for natural-language queries, we propose an approach to reformulate natural-language queries using semantically related class-names. In this research question, we evaluate whether our approach is effective in answering natural-language queries, *i.e.*, placing relevant code examples at the top of the ranked result list.

Approach. We use our approach to reformulate the initial natural-language queries using the semantically related class-names, and then execute the initial queries and the reformulated queries to retrieve the corresponding code examples from the code snippet corpus described in Section 3.3.1. After getting the code examples lists, we compare the search performance of our approach for the initial queries and the reformulated queries by identifying the position of the first correct answer in the result lists. The higher the position is, the better the performance is [42].

We apply Mann-Whitney U test with 95% confidence level (*i.e.*, p-value < 0.05) to determine if the observed difference in the search performance for initial and reformulated queries is significant.

Result. Fig. 3.6 summarizes the performance comparison result between 50 initial natural-language queries and the corresponding reformulated queries. The grey side represents the performance for the initial queries while the black side denotes the performance for the reformulated queries. As shown in Fig. 3.6, for the queries with

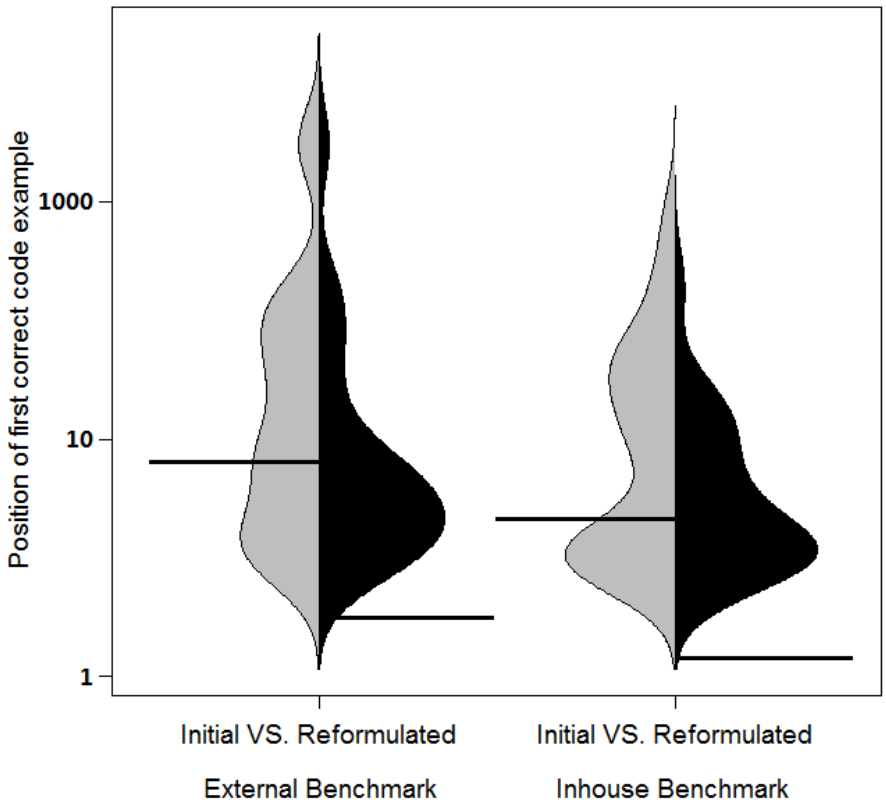

Figure 3.6: Performance comparison between initial natural-language queries and reformulated queries

the external benchmark evaluation, we can observe that the search performance for the reformulated queries is better than the initial queries, and the average position of the first correct code example in the result list has been lifted up from 9 to 3. For the queries with the inhouse benchmark evaluation, we can observe that the average position of the first correct code example in the result list is improved from 6.5 to 1.5. The p-values of the performance comparison with the external and inhouse benchmark evaluations are 0.02 and 5.82e-03, respectively. Therefore, we can conclude that our approach is effective in answering natural-language code search queries by reformulating natural-language queries using relevant class-names.

The proposed approach can answer natural-language code search queries well by effectively reformulating natural-language queries using the related class-names.

3.4.2 RQ3.5: Does our approach outperform the off-the-shelf query reformulation approaches in answering natural-language code search queries?

Motivation. In recent years, a variety of approaches [26][59][81] to generate candidate reformulations for an initial query have been proposed in the area of text retrieval. In this research question, we aim to study whether our approach performs better than these reformulation approaches in terms of answering natural-language queries in the field of code search.

Approach. We implemented three query reformulation approaches described in Section 3.3.3. After obtaining the code example lists for the queries reformulated by three off-the-shelf approaches and our approach, we compare the search performance between the off-the-shelf approaches and our approach by identifying the position of the first correct code example in the corresponding code example lists.

We apply Mann-Whitney U test with 95% confidence level (*i.e.*, p-value < 0.05) to determine if the observed difference in the search performance of the off-the-shelf approaches and our approach is significant.

Result. Fig. 3.7 shows the performance comparison result between three off-the-shelf approaches and our approach using both external and inhouse benchmarks. The grey side represents the performance for the queries reformulated by different off-the-shelf approaches while the black side denotes the performance for the queries reformulated by our approach. We can see from Fig. 3.7 that our approach performs

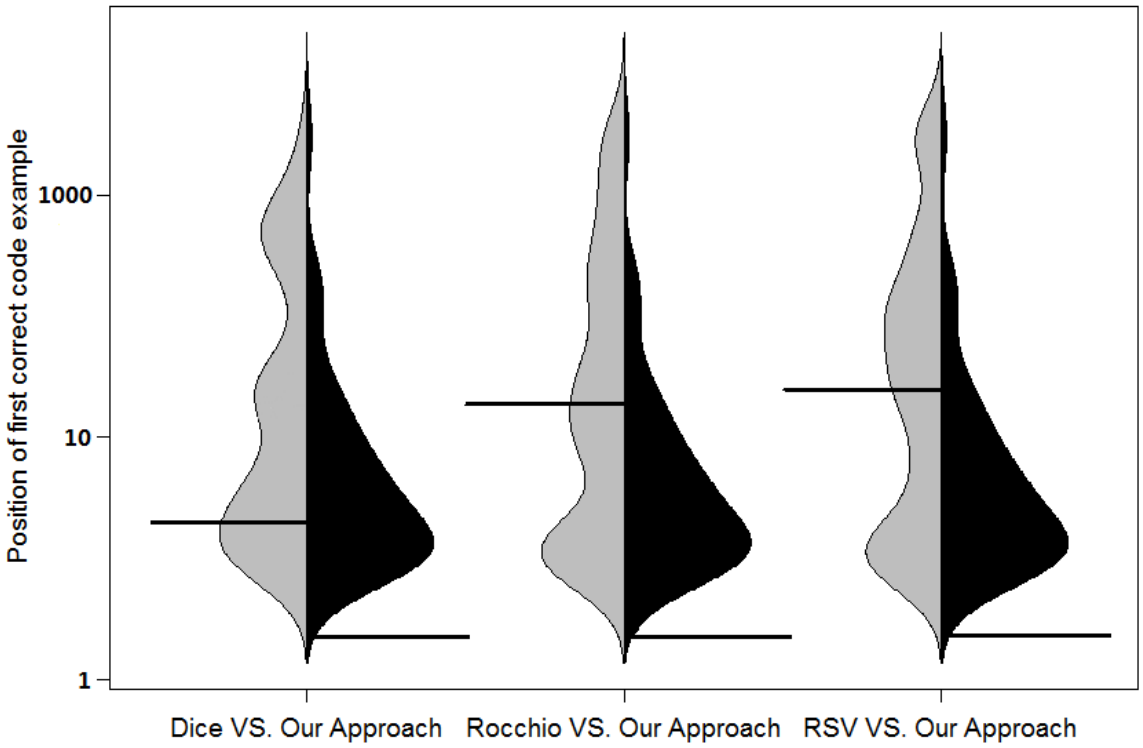

Figure 3.7: Performance comparison between off-the-shelf approaches and our approach

better than the three off-the-shelf approaches. The position of the first correct answer for the queries reformulated by our approach averages 2.5 while that for the queries reformulated by the three off-the-shelf approaches are 6.5 (Dice), 12.5 (Rocchio) and 24.5 (RSV) respectively. The p-values for the comparison between the three existing approaches and our approach are 7.29e-03 (Dice), 5.40e-03 (Rocchio) and 9.79e-04 (RSV), respectively. Therefore, we can conclude that the off-the-shelf query reformulation approaches are not effective in answering natural-language code search queries, and the improvement by our approach over the off-the-shelf approaches in answering natural-language code search queries is significant.

Our proposed approach outperforms the off-the-shelf query reformulation approaches in answering natural-language code search queries.

3.5 Threats to Validity

In this section, we discuss the threats to validity that might affect the results of our study following the common guidelines provided by Robert [58].

Threats to construct validity concern whether the setup and measurement in the study reflect real-world situations. The threats to construct validity of our study mainly come from the categorization of StackOverflow questions, and the creation of the "gold sets". For the classification of the SO questions, we refer to the question types defined in Stolee *et al.* 's study [68]. Although we do not conduct cross validation in the classification process, the result is in line with Stolee *et al.* 's observation. We create two "gold sets" for the evaluation study. One "gold set" is created based on the benchmark data used for clone detection study [70]. The other one is created using the pooling approach, which was previously used by Bajracharya *et al.* [3] to evaluate code example retrieval schemas. The creation process of the two "gold sets" has been done by different judges, and the majority rule is applied when there is disagreement on the correctness of code examples.

Threats to internal validity concern the co-factors that may affect the experimental results. For our study, the main threats to internal validity are the number of recommended class-names and the clicking analysis on the Koders data. We have tried different number of class-names recommended to reformulate the natural-language code search queries, and find that expanding the queries with the top 5 class-names performs better than others. Therefore, we set the number of recommended class-names to 5 in our experiments. In RQ3.2, we decide whether the results returned by Koders answer users' queries using clicking analysis. It could be argued that users could obtain their needed information by just previewing the search results without

actually clicking into the results. The result page of Koders shows only the top few lines of each returned result. it indicates that users are likely to click into the result if they find the relevant one to see more details.

Threats to conclusion validity concern the relationship between treatment and outcome. We use a non-parametric statistic test (Mann-Whitney U test) to show statistical significance of the obtained experiment results. Mann-Whitney U test does not hold assumption on the distribution of data [63].

Threats to external validity concern the generalization of the experiment results. The main threats to external validity of our study are the representativeness of our corpus and code search queries. Our corpus contains 382,073 code snippets extracted from 24,666 open-source projects. We select 50 descriptive questions from SO as natural-language code search queries to study the search effectiveness of our approach. The size of our query set is comparable to the similar studies on code search and recommendation [72][74], and meets the minimum number of queries recommended for search engine evaluation [50].

Threats to reliability validity concern the possibility of replicating the study. We provide the necessary details needed to replicate our work. Replication package is publicly available: http://bit.ly/1Gi5pI2.

3.6 Summary

The preliminary study over StackOverflow questions shows that code search questions are prevalent in StackOverflow, and they are significantly different from keyword-based code search queries in terms of the proportion of API tokens (*i.e.*, class or method names). However, the low proportion of API tokens in the natural-language

code search queries (*i.e.*, code search questions) would negatively affect the success of the code search process of the existing code search engines. Therefore, we propose an approach to automatically reformulate the natural-language code search queries using the most semantically-related keywords. The proposed approach takes a natural-language code search query as input and returns the most semantically related class-names to the query. The most semantically related class-names are then used to reformulate natural-language queries. The evaluation result shows that our approach can effectively recommend the semantically related class-names to reformulate natural-language queries. The improvement on search effectiveness over the existing query reformulation approaches is statistically significant.

Chapter 4

Learning to Rank Code Examples for Code Search Engines

In recent years, the market of mobile applications has experienced a tremendous success [73]. The existence of Application Programming Interfaces (APIs) is one of the important factors contributing to the success of application development economy [47]. During the development process, developers often encounter unfamiliar APIs. Code search engines are designed to help developers learn unfamiliar APIs by recommending relevant code examples.

A code search engine requires a ranking schema to place the most relevant code examples at the top of the result list. However, the ranking schemas used by the existing code search engines are mainly hand-crafted heuristics. The configurations of the existing ranking schemas are determined based on observations on a specific dataset, which demands the domain experts repeatedly to tune and evaluate the ranking schema. Therefore, it is hard to apply existing ranking schemas on different datasets.

To address the limitation of the existing ranking schemas, in this chapter, we

propose an approach which applies learning-to-rank techniques to automatically learn a ranking schema from the training data for a code search engine. Then we evaluate the performance of our learning-to-rank approach in terms of ranking relevant code examples in the context of Android application development. Finally we discuss the threats to validity that might affect the study results.

4.1 Background

4.1.1 Ranking Schema

A ranking schema is used to compute a relevance score between a query q and a candidate code example c. As shown in Eq. 4.1, the scoring function is defined as a weighted sum of k (k= 12) features (Table 4.2), where each feature $f_i(q, c)$ measures the relevancy between a query q and a candidate code example c from the i_{th} feature's point of view:

$$rel(q, c) = \sum_{i=1}^{k} w_i \times f_i(q, c) \tag{4.1}$$

Where $f_i(q, c)$ represents the i_{th} ranking feature; w_i represents the weight of the i_{th} ranking feature [79].

Given a query q as input at run-time, the ranking schema computes the score $rel(q, c)$ for each candidate code example c relevant to the query, and uses the relevance scores to rank all the candidate code examples in a descending order. The developer would be presented with a ranked list of code examples, with the expectation that the code examples appearing higher in the list are more relevant to the query.

4.1.2 Learning-to-rank Technique

In the ranking schema, the weights w_i for different ranking features are hard to determine. In this study, we attempt to train the weights automatically using learning-to-rank techniques, which are the application of machine learning algorithms in building ranking models.

Learning-to-rank techniques can learn a ranking model from the labeled objects in the training dataset; the trained model can then be used to rank new objects. As stated by Li [45], learning to rank techniques can fall into three categories: pointwise [25][46], pairwise [15][27][37], and listwise [19][75] approaches. Pointwise approaches compute the absolute relevance score for each code example. In pairwise approaches, ranking is transformed to the classification on code example pairs to reflect the preference between two code examples. In listwise approaches, code example lists are generated through the comparison between code example pairs. In our study, it is not necessary to obtain the complete order of candidate code examples for a query. We are more interested in placing relevant code examples above less relevant ones. Thus, we apply the pairwise approach in our work. Specifically, we apply a pairwise algorithm, called RankBoost [27], to learn the ranking schema. The learning process identifies how the features should be combined in Eq. 4.1 to create a ranking schema for code example search.

4.2 Approach Overview

In this section, we first introduce the overall process of our approach. Then, we present the steps for extracting the features of code examples. Lastly, we show the details for learning a ranking schema from the training data.

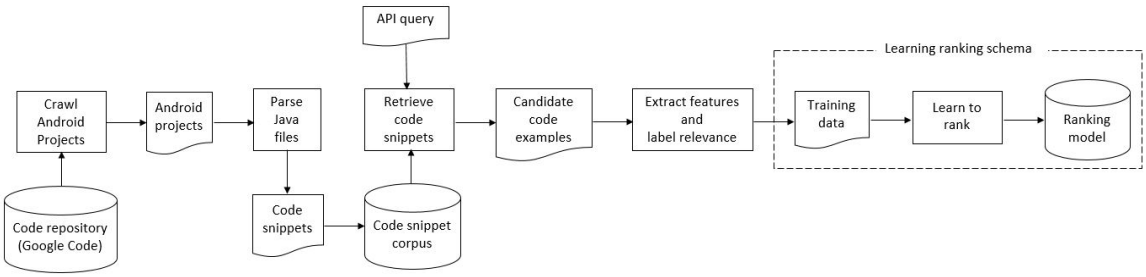

Figure 4.1: The process of the proposed approach for building a ranking schema

4.2.1 Overall Process

The overall process of our code search approach consists of four major phases: 1) crawling Android projects; 2) extracting features; 3) learning a ranking schema; and 4) ranking candidate code examples for new queries. The first three phases are off-line processes, and are illustrated in Fig. 4.1. The ranking phase occurs at run-time when a query is issued by a developer. The input of our approach is the queries containing a class and one of its methods, and a corpus of code snippets. Given a query, the approach retrieves all the code snippets that contain the class or method names specified in the query from the corpus, and computes the cosine similarities between the query and the code snippets. Finally, we select the code snippets that are the most similar to the query as the candidate code snippets. The candidates are ranked using the trained ranking schema.

Crawling Android projects. To retrieve code examples for queries, we have to prepare a corpus of code snippets. To build the corpus of code snippets for Android application development, we crawl a code hosting repository, GoogleCode[1], to find the projects labeled as "Android". The crawler downloads the source code of the projects.

[1]Google Code project hosting: https://code.google.com/.

Table 4.1: Summary of our corpus

Item	Number
Num. of Projects	586
Num. of Java Files	65,592
Num. of Code Snippets	360,068
Loc. of Code Snippets	3,876,295

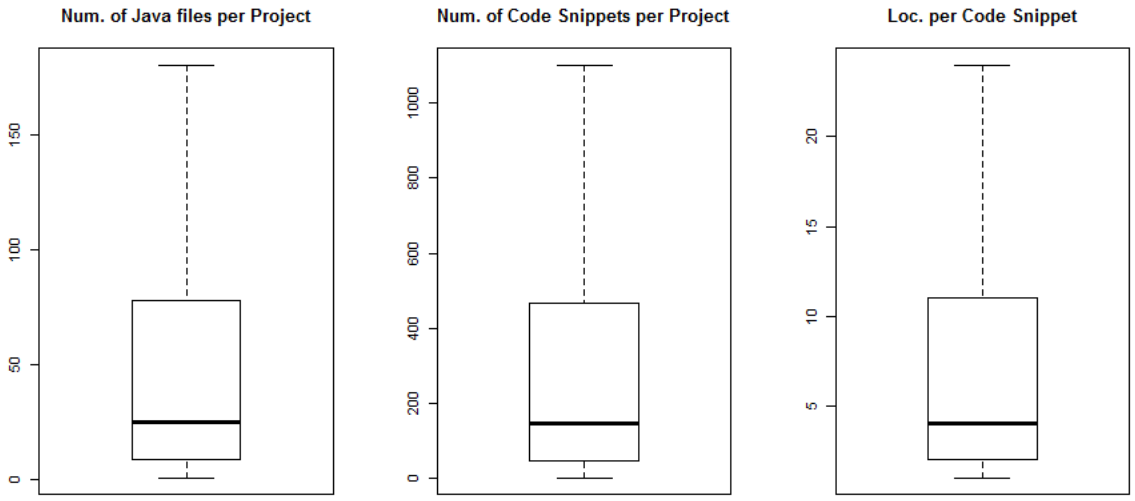

Figure 4.2: Detailed summary of our corpus

To extract code snippets from the open-source projects, we use a Java syntax parser[2] available in Eclipse JDT to extract one code snippet from each method defined in the source code files with "java" extension. The approach of extracting code snippets has been used by the earlier work on code example search and recommendation for Java source code [42][54]. The extracted code snippets are added to the corpus. Table 4.1 and Fig. 4.2 summarize the description of the corpus used in this study.

Extracting features. We represent each candidate code example as a vector, V_s, containing a set of feature values (Table 4.2) extracted from the code example.

[2]We use the parser from Eclipse JDT: http://www.eclipse.org/jdt/

The feature vector can be represented as $V_s = (f_1, ..., f_i, ..., f_n)$ where f_i denotes the value of the i_{th} feature, and n denotes the total number of features, *i.e.*, 12, in our approach.

Learning a ranking schema. In this phase, our approach automatically learns a ranking schema from the training data. The training data contains a set of queries and their relevant candidate code examples. We represent the training data as a set of triples (q, r, V_c). More specifically, q represents a query. r denotes the relevance between a query q and a candidate code example s. The relevance is manually labeled by curators. V_c represents a vector that contains the feature values of the candidate code example c. Then we learn a ranking schema from the training data using RankBoost [27], one of the learning-to-rank algorithm.

Ranking candidate code examples for new queries. For a new query, the trained ranking schema would compute a score for each candidate code example. The score denotes the relevancy between the query and the candidate code example. All the candidate code examples are then ranked based on the computed scores in a descending order. The code examples appearing higher in the ranked result list are more relevant to the query, *i.e.*, more likely to be effective code examples for the query.

4.2.2 Feature Extraction

In this section, we describe the features of code examples used in our approach to train the ranking schema. In total, we adopt 12 features used in earlier studies [32][42][72]. As listed in Table 4.2, we classify the features into four categories: similarity, popularity, code metrics and context. Furthermore, we can divide the features into two

Table 4.2: Selected features of code examples

Group	**Feature name**	**Feature description**	**Query-dependent**
Similarity	Textual similarity	Cosine similarity between a query and a candidate code example	yes
Popularity	Frequency	The number of times that the frequent method call sequence of a candidate code example occurs in the corpus.	no
	Probability	The probability of following the method call sequence in a candidate code example	
Code Metrics	Line length	The number of lines of code in a candidate code example	no
	Number of identifier	The average number of identifiers per line in a candidate code example	
	Call sequence length	The number of method calls in a candidate code example	
	Comment to code ratio	The ratio of the number of comment lines to the LOC of a candidate code example [16]	
	Fan-in	The number of unique code snippets that call a specific candidate code example	
	Fan-out	The number of unique code snippets called by a candidate code example	
	Page-rank	The measurement of the importance of a candidate code example	
	Cyclomatic complexity	The number of decision points (for, while, etc.) in a code example	
Context	Context similarity	Jaccard similarity between the method signatures of a candidate code example and the method body where a query is issued	yes

groups: query-dependent and query-independent features [48]. Query-dependent features are calculated with regard to a query. Query-independent features only reflect characteristics of a code example regardless of the query. The 12 features are described in the following paragraphs.

Textual Similarity

Textual similarity between a query and candidate answers is the basic feature used to judge relevancy in code search [43][52][72]. We use Vector Space Model (VSM) [61] to compute the textual similarity between a query and candidate answers. In this model, the query and the candidate code examples would be represented as vectors of term weights. We compute the term weight $w_{t,d}$ for each term t in the query or code examples using the classical term frequency-inverse document frequency (tf-idf) weighting schema [50]. As defined in Eq. 4.2, Tf-idf weighting schema can reflect the importance of a term to a document in a document collection.

$$\begin{aligned} w_{t,d} &= nf_{t,d} \times idf_t \\ nf_{t,d} &= 0.5 + \frac{0.5 \times tf_{t,d}}{max_{t \in d} tf_{t,d}} \quad idf_t = log\frac{N}{df_t} \end{aligned} \tag{4.2}$$

Manning et al. [50] defines tf-idf where term frequency $tf_{t,d}$ means the number of times that term t appears in document d (query or code example); document frequency df_t denotes the number of documents in which term t appears; idf_t means the inverse document frequency which represents the specificity of term t for the document that contains it. idf_t is defined as the inverse of the number of documents in which term t appears, and N is the total number of documents.

Then we compute the textual similarity between a query and a candidate answer

using cosine similarity [61] as defined in Eq. 4.3.

$$textualSim(V_q, V_c) = \frac{V_q^T V_c}{|V_q||V_c|} \tag{4.3}$$

Where V_q is a query vector and V_c is a candidate code example vector.

Popularity

Recent studies on code search [72], recommendation [17], and completion [78] use popularity to identify candidate answers with a higher acceptance rate. Popularity represents the closeness of a code snippet to the common implementation pattern frequently observed in a corpus of source code. The underlying rationale is that the closer to the common patten in the corpus, the higher chance for the recommended code snippet to be accepted by a developer. The popularity of the underlying pattern in a code example is query-independent, and it can be measured using frequency [72] or probability [74] feature.

To measure the popularity using frequency, we use the same approach suggested by Keivanloo *et al.* 's study [42]. An usage pattern is an ordered sequence of method calls that are always used together to implement a reasonable functionality. The frequency of a usage pattern is the number of times that a set of method calls in the usage pattern are used together in the corpus. The underlying usage pattern of a code example is the usage pattern most similar to the call sequences of the code example. We consider the frequency of the underlying usage pattern of a code example as the popularity of the code example. To identify the underlying usage pattern of a code example, we extract the method call sequence for each code snippet in the corpus and use the frequent itemset mining technique [31] to identify the usage patterns in

the corpus by analyzing the method calls that are frequently used together. Then, we identify the most similar usage pattern to the call sequence of a code example by computing the cosine similarity between them, and consider the most similar one as the underlying usage pattern of the code example.

In addition to measuring the popularity using frequency, popularity can be calculated using a probability-based approach proposed by Wang *et al.* [74]. We split the call sequences extracted from the corpus into pairs of two consecutive method calls. A method call sequence in a code snippet can be denoted as $S_m = m_1, m_2, ..., m_n$, where n represents the total number of method calls in the code snippet. Then we split the call sequence into method call pairs $P = p_1, p_2, p_i, ..., p_{n-1}$, where p_i denotes method call pair (m_i, m_{i+1}); (m_i, m_{i+1}) means method m_i is called before m_{i+1}. For example, the method sequence in the code example shown in Figure 1.3 can be split into 7 method call pairs, *i.e.*, {*getAudioInputStream(), getFormat()*}, {*getFormat(), DataLine.Info()*}, {*DataLine.Info(), isLineSupported()*}, {*isLineSupported(), getLine()*}, {*getLine(), open()*}, {*open(), start()*}, {*start(), drain()*}, {*drain(), close()*}. After splitting all the method call sequences in the corpus into method call pairs, we can compute the probability of method call pair p_i as: $P(p_i) = \frac{1}{N}$, where N is the number of method call pairs where method m_i is called before the other method in the corpus. The probability feature of a specific code example can be computed as follows [74]:

$$probability = \prod_{i=1}^{n-1} P(p_i) \tag{4.4}$$

Where n represents the total number of method calls in the code example, $P(p_i)$ is the probability of method call pair p_i.

Code Metrics

Code metric group contains four code metrics that are used in earlier studies [16] on code search and code quality prediction. Code metrics are a set of query-independent features. Table 4.2 summarizes the code metric features used in our approach. As indicated by Buse and Weimer [16], lines of code and the average number of identifiers per line can be used to predict code readability as one of the quality aspects of code examples. Call sequence length denotes the number of method calls in the call sequence of a code example. Comment to code ratio represents the proportion of comments in the code example. Fan-in, fan-out and page-rank measures the complexity of inter-code-snippets. Fan-in is defined as the number of code snippets that call a specific code snippet. Fan-out describes the number of code snippets called by a specific code snippet. Page-rank works by counting the number links to a particular code example to determine the importance of the code snippet. The underlying assumption is that more important code snippets are likely to receive more links from other code snippets. To measure page-rank for code search [52], we build a graph for code snippets in the corpus based on the their call relations. In the graph, if code snippet A calls code snippet B, then, a link would exist between code snippet A and code snippet B. Then we compute the page-rank value for each code snippet in the call graph using a R package called "igraph"[3]. Assuming that code snippets, $C_1, ...C_n$, call code snippet C, and $N(E)$ represents the number of code snippets called by code snippet E. Then the package computes the page-rank of each code snippet E using

[3]Igraph package: http://cran.r-project.org/web/packages/igraph/igraph.pdf

Eq.4.5 [10]:

$$PR(E) = (1 - d) + d(PR(C_1)/N(C_1) + ... + (PR(C_n)/N(C_n)) \tag{4.5}$$

Where the parameter d is a damping factor, and is usually set to 0.85 [10].

Context Similarity

The context similarity refers to the similarity between the context of the query and the candidate code snippets [82]. An earlier study [38] observed that the context feature improves the success rate of code search. We use the method signature of a code example and the method signature of the method body where a query is issued to represent the context of the code example and the query, respectively. Hence, context similarity measures the method signature similarity between candidate code examples and the method where the query is issued (the query formulation process is described in Section 4.3.1).

We tokenize the method signatures of the method where query q is issued using camel case splitting, and represent the set containing the tokenized terms as S_q. Similarly, we denote the set containing the tokenized terms from the method signature of a candidate code example as S_c. Then, the context similarity between a query and a code examples can be computed using Jaccard index [40] between the two term sets as follows:

$$contextSim(S_q, S_c) = \frac{S_q \cap S_c}{S_q \cup S_c} \tag{4.6}$$

Where S_q is the set containing the tokenized terms from the query context; S_c is the

set containing the tokenized terms from the code example context.

For example, for the query, "$Clip, start()''$", assuming that the method signature of the method which issues the query is *"private void playAudioClip(String clipFileName)"*, and the method signature of the candidate code example (shown in Figure 1.3) is *"public void play(File file)"*, then we can tokenize the two method signatures into {private, void, play, audio, clip, string, file, name} and {public, void, play, file}, respectively. Therefore, the context similarity between the query and the candidate code example can be computed as $3/9 = 0.33$ using Eq. 4.6.

4.2.3 Training a Ranking Schema

In our approach, we train the ranking schema using a learning-to-rank algorithm that is known as RankBoost proposed by Freund *et al.* [27]. This section provides a summary of the algorithm as defined in [27].

The input of the learning-to-rank algorithm is the training data which contains the candidate code examples relevant to a set of queries. Each code example is represented as a record with the form (q, r, V_c), where q means a query id; r denotes the relevance between a query q and a candidate code example c, which is tagged by curators; and V_c is the vector containing different feature values of a code example c.

The known relevance information of code examples in the training data can be encoded as a feedback function ϕ. For any pair of code example $(c_0,\ c_1)$ in the training data, $\phi(c_0, c_1)$ denotes the difference between the tagged relevance of c_0 and c_1. $\phi(c_0, c_1) > 0$ means that the code example c_1 is tagged with higher relevance than c_0. $\phi(c_0, c_1) < 0$ means the opposite. A value of zero indicates no preference between c_0 and c_1.

The learning algorithm aims to find a final ranking H that is similar to the given feedback function ϕ. To maximize such similarity, we focus on minimizing the number of pairs of the code examples which are misordered by the final ranking relative to the feedback function. To formalize the goal, let $D(c_0, c_1) = x * max\{0, \phi(c_0, c_1)\}$ so that all negative values of ϕ are set to zero. Here, x is a positive constant chosen so that $\sum_{c_0,c_1} D(c_0, c_1) = 1$. Let us define a pair (c_0, c_1) to be crucial if $\phi(c_0, c_1) > 0$. Now the goal of the learning algorithm is transformed to find a final ranking H which can minimize the (weighted) number of the crucial-pair misorderings, which is defined as *ranking loss* [27] and shown in Eq. 4.7.

$$loss = \sum_{c_0,c_1} D(c_0, c_1)|H(c_1) \leq H(c_0)| \tag{4.7}$$

where $|H(c_1) \leq H(c_0)|$ *is 1 if* $H(c_1) \leq H(c_0)$ *holds and 0 otherwise.*

Algorithm 1 shows the details of the learning process of RankBoost [27] for the final ranking H. RankBoost operates in rounds. In each round t, it produces a ranking function f_t based on the ranking feature listed in Table 4.2. Meanwhile, it maintains a value $D_t(c_0, c_1)$ over each pair of code examples to emphasize the different parts of the training data. As shown in Eq. 4.8, the algorithm uses the ranking function f_t to update the value D_t in round t.

$$D_{t+1}(c_0, c_1) = D_t(c_0, c_1)exp(\alpha_t(f_t(c_0) - f_t(c_1))) \tag{4.8}$$

Where $D_t(c_0, c_1)$ *is the maintained value for each pair of code examples in round* t*;* f_t *is the produced ranking function at round* t*;* α_t *is a parameter for the ranking function* f_t*, and* $\alpha_t > 0$ *[27].*

Suppose we expect the code example c_0 to be ranked higher than the code example c_1, based on the definition $D_{t+1}(c_0, c_1)$ in Eq. 4.8, $D_{t+1}(c_0, c_1)$ will decrease if the ranking function f_t gives a correct ranking ($f_t(c_1) > f_t(c_0)$) and increase otherwise. Therefore, D_{t+1} will tend to concentrate on the misordered code example pairs. With the indication, the ranking loss of the final ranking H is proven to hold the equation [27]: $loss(H) \leq \prod_{t=1}^{T} D_{t+1}$. To minimize the loss function of the ranking schema H, we have to minimize D_{t+1}. Freund *et al.* study [27] shows that D_{t+1} is minimized when

$$\alpha_t = \frac{1}{2}\ln(\frac{1+l_t}{1-l_t}), \quad l_t = \sum_{c_0,c_1} D_t(c_0, c_1)(f_t(c_1) - f_t(c_0)) \tag{4.9}$$

Where $D_t(c_0, c_1)$ is the maintained value for each code example pair at round t; f_t is the accessed ranking function at round t.

In the process of minimizing the ranking loss of the final ranking H, the parameter α_t for the ranking function f_t is obtained using Eq.4.9. Therefore, the final ranking can be represented as a weighted sum of different ranking features, *i.e.*, $H(c) = \sum_{t=1}^{T} \alpha_t f_t(c)$, which can achieve the best performance with regard to the loss function.

Algorithm 1 : RankBoost [27]

Require: Initial values of $D(c_0, c_1)$ over each code example pair in the training data.

1: Initialize:$D_1(c_0, c_1) = D(c_0, c_1)$
2: **for** $t = 1, ..., T(T = 12)$ **do**
3: Build ranking function $f_t(c)$ based on the ranking feature.
4: Choose α_t using Eq. 4.9.
5: Update: $D_{t+1}(c_0, c_1) = D_t(c_0, c_1)exp(\alpha_t(f_t(c_0) - f_t(c_1)))$
6: **end for**
7: Output the final ranking $H(c) = \sum_{t=1}^{T} \alpha_t f_t(c)$

4.3 Case Study Setup

To evaluate the performance of our approach, we conduct a case study. The goal of this case study is two-fold: (1) evaluate the effectiveness of our proposed approach; and (2) compare the impact of the studied features on the performance of our approach. For the case study, we need a corpus of Android code snippets, training and testing data, and performance measures. We have described the steps for building the corpus in Section 4.2.1. In this section, we discuss the methods for creating the training and testing data, the definition of the performance measures and the comparison baselines.

4.3.1 Training and Testing Data Collection

Selecting Queries and Code Examples

To create the training and testing datasets, we need a set of queries and candidate code examples. We use the automatic framework proposed by Bruch *et al.* [13] to randomly select a set of queries from our corpus. In this framework, at first, a code snippet is randomly selected from the corpus. The code snippet acts as an expected answer. Then, a query is automatically generated for the expected answer by randomly selecting an API and the invoked method from the content of the expected answer. Our final query set used for both training and testing steps includes 50 queries along with the corresponding expected answers. The size of query set meets the acceptable number of queries required for performance evaluation of ranking schemas [56].

For a given query, the number of code snippets that contains the class or method specified in the query is very large. Building a training dataset covering all candidates

requires a considerable amount of time and resources for the learning process, and it is not feasible in practice [79]. Therefore, as suggested by Ye *et al.* study [79], as a practical solution for building the training dataset in the field of learning-to-rank, we consider only the code snippets that are the most similar to the query as the candidate code examples. This approach is also used by the earlier studies in code search engines, *e.g.*, Bajracharya *et al.* [4]. For each query q in the query set, we first extract all the code snippets that contain certain items of the query, and then use the cosine similarity feature $textSim(q, c)$ to rank all the extracted code snippets. Similiar to Niu *et al.* 's study [56], we select the top 50 code snippets as the candidate code examples for a query. In total, we select 2,500 candidate code examples for the 50 queries.

Relevance Judgement

To create a training dataset for our approach to learn how to rank code examples, we need to provide the relevances between queries and their candidate code examples. The relevance between a query and a candidate answer is described by a label which represents the relevance grade. We use the widely accepted multi-graded absolute relevance judgment method [45] for labeling. Specifically, we use four relevance levels, *i.e.*, bad, fair, good, and excellent. The definition of the relevance levels is listed in Table 4.3. Similar to earlier studies on learning-to-rank or code search [17][45], we ask assessors to assign a relevance label to each pair of query and candidate answer. Assessors need to judge the relevance between each query and a candidate answer by comparing the candidate answer to the expected answer of the query. We describe the setup of relevance judgement process as follows:

Table 4.3: Relevance level instruction

Relevance level	Definition
Excellent	A candidate code example is exactly matched with or highly similar to the expected answer
Good	A candidate code example contains major operations of the expected answer
Fair	A candidate code example contains a part of operations of the expected answer and contains many irrelevant lines of code
Bad	A candidate code example contains few operations of the expected answer, or is totally irrelevant to the expected answer

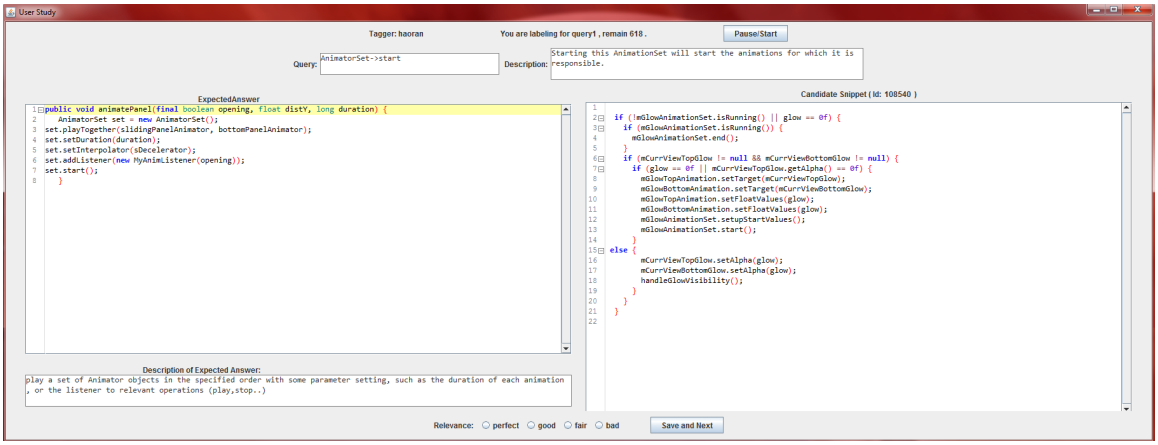

Figure 4.3: Interface of relevance labeling tool

Assessors. To judge the relevances between queries and code examples, we recruited 5 assessors to participate in our study. The 5 assessors are 3 graduates and 2 undergraduates. The 3 graduates have more than five-years Java programming experience; and the 2 undergraduates have one-year Android application development experience. Before the assessment process, they all received a 30-minutes training on the usage of our labeling tool and the definitions of the four relevance levels.

Assignment. To reduce the subjectivity issue in human judgement, different judgements for each code example are needed for cross-validation. As suggested by

Niu *et al.* [56], we divide the 50 queries into five groups and label them as G_1, G_2, G_3, G_4, and G_5. Each group contains 10 queries. Then three consecutive query groups, *i.e.*, $\{G_1, G_2, G_3\}$, $\{G_2, G_3, G_4\}$, $\{G_3, G_4, G_5\}$, $\{G_4, G_5, G_1\}$ and $\{G_5, G_1, G_2\}$, are randomly assigned to one assessor. Therefore, the candidate answers in each group are reviewed by three independent assessors. In total, we gathered 7,500 relevance labels from the five assessors for the 2,500 candidate answers of the 50 queries in our dataset. The majority rule [74] is used when there is a disagreement on the relevance labels for a specific candidate answer.

Labeling Environment. We develop a labeling tool for the four-graded relevance judgment on candidate code examples. The interface of the tool is illustrated in Fig. 4.3. In this tool, a query and the query description are shown on top of the window. A candidate answer (*i.e.*, code snippet) and the expected answer of the query are shown in the main area. Four-graded buttons are displayed at the bottom of the interface. The assessors label the code examples query by query. But they are allowed to jump back and forth among different queries in case certain code examples are wrongly labeled. To ensure fatigue does not affect the labeling process, the labeling sessions are limited to 30 minutes to have a 10-minute break [56]. On average, an assessor needs 5 sessions to label the code examples for a specific group (*i.e.*, 10 queries).

4.3.2 Performance Measures

We evaluate the performance of our approach by ascertaining the goodness of the ranked results lists produced by our approach. We need the graded relevance metrics to evaluate the result lists since the multi-graded relevance scale is used in the result lists. Discounted Cumulative Gain (DCG) is the only commonly used metric for

graded relevance [21]. Chapelle *et al.* [21] propose another evaluation metric called Expected Reciprocal Rank (ERR) and find that ERR quantifies user satisfaction more accurately than DCG. Therefore, we use DCG and ERR to evaluate our approach. Since developers are always interested in the top k answers, we use the extended evaluation measures, k-DCG and k-ERR, to emphasize the importance of the ranking of the top k answers.

DCG [41] measures whether highly relevant answers appear towards the top of ranked list. The premise of DCG is that highly relevant code examples appearing lower in the result list should be penalized. To achieve better accuracy, the DCG at a particular position k (k-DCG) should be normalized across queries. This is done by ranking the code examples in a result list based on their relevances, producing the maximum DCG till position k, called ideal DCG (IDCG). Then, the normalized k-DCG (k-NDCG) is defined as follows. The values of k-NDCG range from 0.0 to 1.0. Higher k-NDCG values are desired.

$$k\text{-}NDCG = \frac{k\text{-}DCG}{k\text{-}IDCG}, \;\; k\text{-}DCG = \sum_{j=1}^{k} \frac{2^{r_j} - 1}{\log\,(1+j)} \tag{4.10}$$

Where r_j is the relevance label of the code example in the j_{th} position in the ranked result list; k-IDCG is the k-DCG of an ideal ordering of top k code examples, which means that top k code examples are ranked decreasingly based on their relevances [56].

k-ERR [21] is defined as the expectation of the reciprocal of the position k at which a developer stops when looking through a result list, which can be denoted as $\sum_{i=1}^{k} \frac{1}{k} P(user\ stops\ at\ position\ k)$. Suppose for a query, a ranked result list of k code examples, $c_1, ..., c_k$, is returned. Stopping at position k involves being satisfied with code example k, and not having been satisfied with any of the previous code examples

at the positions $1, ..., k-1$, the probabilities of which can be denoted as $P(c_k)$ and $\prod_{j=1}^{k-1}(1-P(c_j))$, respectively. The two probabilities are multiplied by $1/k$, because it is the inverse stopping rank whose expectation is computed. Therefore, the definition of k-ERR is given as Eq. 4.11. The values of k-ERR range from 0.0 to 1.0. Higher k-ERR values are desired.

$$k\text{-}ERR = \sum_{i=1}^{k} \frac{1}{k} P(r_i) \prod_{j=1}^{i-1}(1 - P(r_j)), \;\; P(r) = \frac{2^r - 1}{2^{r_{max}}} \tag{4.11}$$

Where r denotes the relevance label of a code example; r_i and r_j are the relevance labels of the code examples in the i_{th} and j_{th} position respectively in the ranked result list; r_{max} is the highest relevance label in the candidate code example list. [21]

To make sure the stability of the training process, we conducted 10-fold cross validation in the performance evaluation study. The queries are split into 10 equally sized folds $fold_1, fold_2, ..., fold_{10}$. In each round, 9 folds are the training data, and the remaining fold is the testing data. The training and testing data for each round are independent. For a given k value, we can obtain 10 k-NDCG and k-ERR values for the evaluation of a ranking schema. We compute the average of the 10 k-NDCG or k-ERR values to represent the performance of the ranking schema.

4.3.3 Comparison Baselines

We summarize the ranking schemas used in the existing ranking approaches into five schemas, including Random Ranking, Similarity Ranking, WeightedSum Ranking, Priority Ranking, and ReRanking. We list the five existing ranking schemas as follows:

- **Random Ranking** randomly ranks the candidate answers within a result set. It is a common practice in machine learning studies to measure the improvement

over random guesses [35].

- **Similarity Ranking** ranks candidate code examples based on their textual similarity with the corresponding query.
- **WeightedSum Ranking** computes the relevance value between a query and a candidate code example using a weighted sum of different features of the code example with equal coefficients [32].
- **Priority Ranking** ranks code examples based on the primary feature. If two code examples have the same value for the primary feature, then the secondary feature is used to prioritize the two code examples [54][72].
- **ReRanking** would re-rank the top-k candidate answers determined by the primary feature using a secondary feature [42].

We also compare our approach against a commercial code recommendation system, Codota. More specifically, Codota is the only publicly available code example search engine that is capable of ranking Android code examples. Codota extracts the code snippets from GitHub, Google Code, and StackOverflow. The ranking schema used in Codota identifies common, credible and clear code snippets. However, the details of Codota's ranking schema is not known since it is a closed source commercial product.

4.4 Case Study Result

This section discusses the results of our three research questions. We describe the motivation, analysis approach and findings for each research question.

4.4.1 RQ4.1: Does the learning-to-rank approach outperform the existing ranking schemas for code example search?

Motivation. A ranking schema can sort the candidate answers as a ranked list based on their relevances to the query. Existing ranking schemas are mainly hand-crafted heuristics with predefined configurations. We propose a ranking schema for code search using a learning-to-rank technique, which can automatically learn a ranking schema from a training dataset. In this research question, we evaluate whether the ranking performance of our learning-to-rank approach is better than the existing ranking schemas for code example search in the context of Android application development.

Approach. At first, we conduct correlation analysis for the 12 identified features of code examples. Correlated features have similar contribution to the training process of ranking schema. Minimizing the correlation between features can reduce the time and resource of training process and increase the stability of the trained ranking schema [64]. Spearman's rank correlation coefficient (Spearman's rho) and Pearson product-moment correlation coefficient (Pearson's r) are two commonly used measures for the correlation analysis. Pearson's r evaluates whether two variables tend to change together at a constant rate. When the relationship between the variables is not linear, Spearman's rho is more appropriate to use. Since the different feature values of code examples are not always change with a consistent rate, we use Spearman's rho to analyze the correlation between different features of code examples. The values of Spearman's rho range from -1 and +1. A value close to +1 or -1 means that one variable is a monotone function of the other. A value close to 0 indicates that the two variables have no correlation. Table 4.4 shows the mapping between the values of

Table 4.4: Mapping Spearman's rho with coefficient level [18]

Spearman's rho	**Coefficient Level**
over 0.8	very high
0.6 - 0.8	high
0.4 - 0.6	normal
0.2 - 0.4	low
less than 0.2	very low

Spearman's rho and the level of correlation [18]. After finding the highly correlated features, we select one representative feature from the correlated features.

Once we narrow down to minimally collinear features, we use these features to train a ranking schema using our approach, and then compare the ranking performance of our approach with the existing ranking schemas as described in Section 4.3.3. We use the corpus and dataset described in Section 4.3.1 for this study. We measure the performance of the trained ranking schema using k-NDCG and k-ERR defined in Eq. 4.10 and Eq. 4.11 respectively. Higher k-NDCG and k-ERR values are desired. We compute the performance improvement of our approach by comparing the performance of our approach with baseline ranking schemas using the formula: $Improvement = \frac{O-B}{O}$, where O denotes the ranking performance of our approach, B means the ranking performance of a baseline ranking schema. To ensure the reliability of our results, we perform a sensitivity study, with k ranging from 1 to 20.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data [63]. We use Mann-Whitney U test to determine if the observed difference in the performance of our approach and the existing ranking schemas is significant. Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data [63]. We conduct Mann-Whitney U test

Table 4.5: Mapping Cliff's delta with Cohen's standards [60]

Cliff's Delta	**Cohen's d**	**Cohen's standards**
0.147	0.2	small
0.330	0.5	medium
0.474	0.8	large

with 95% confidence level (*i.e.*, p-value < 0.05). We also calculate effect size using Cliff's delta [60] to quantify the difference between our approach and the existing schemas. Cliff's delta is -1 or +1 if all the values in one distribution are larger than the other one, and it is 0 when two distributions are identical [23]. Cohen's standards are commonly used to interpret the effect size. Therefore, we map the Cliff's delta to Cohen's standards as summarized in Table 4.5.

Result. Fig. 4.4 shows the correlation structure of the 12 identified features. As indicated by Table 4.4, five features, including line length, call sequence length, fan-in, fan-out and cyclomatic complexity, are correlated. Since line length is commonly used to judge about code quality, *e.g.*, [16][55], we select line length to represent the five correlated ones. Therefore, we have 8 code example features to build the ranking schemas, *i.e.*, the features in Table 4.2 except for call sequence length, fan-in, fan-out and cyclomatic complexity.

Fig. 4.5 and Fig. 4.6 present the k-NDCG and k-ERR results for our approach and five baselines, with k ranging from 1 to 20. We can observe from Fig. 4.5 and Fig. 4.6 that our approach achieves better performance than the other ranking schemas on both k-NDCG and k-ERR. The k-NDCG improvement achieved by our approach ranges from 32.37% to 54.61%. The k-ERR improvement achieved by our approach ranges from 43.95% to 51.66%. Considering the fact that most search engines always

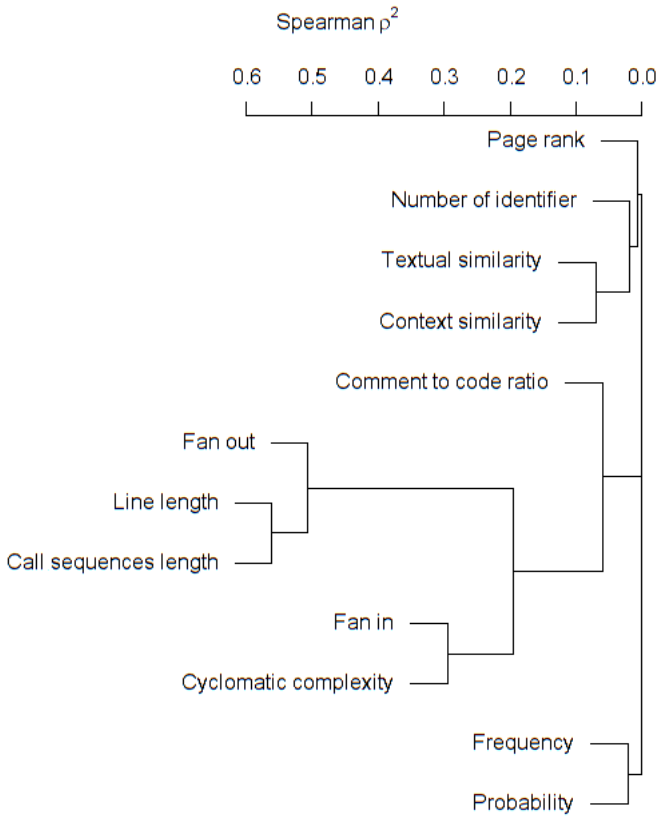

Figure 4.4: The correlation structure of the 12 identified features

show top 10 results in their first result page [52], we care more about the top 10 results (*i.e.*, $k = 10$). As shown in Fig. 4.7 and Fig. 4.8, 10-NDCG and 10-ERR values for our approach are 0.46 and 0.38, which are better than all of the studied baselines, *i.e.*, Random Ranking (0.26 and 0.22), Similarity Ranking (0.31 and 0.19), WeightedSum Ranking (0.31 and 0.18), Priority Ranking (0.29 and 0.19), ReRanking (0.31 and 0.2). Tables 4.6 and 4.7 list the performance improvement of our approach, p-values and effect size of performance comparison between our approach and baselines using k-NDCG and k-ERR measures with k=10. On average, our approach achieved 35.65%

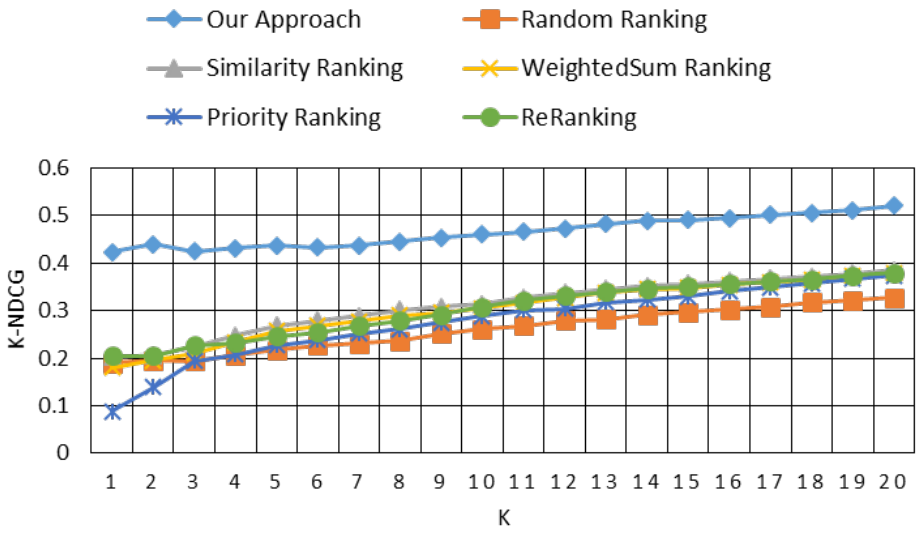

Figure 4.5: The k-NDCG comparison result between our approach and the existing ranking schemas

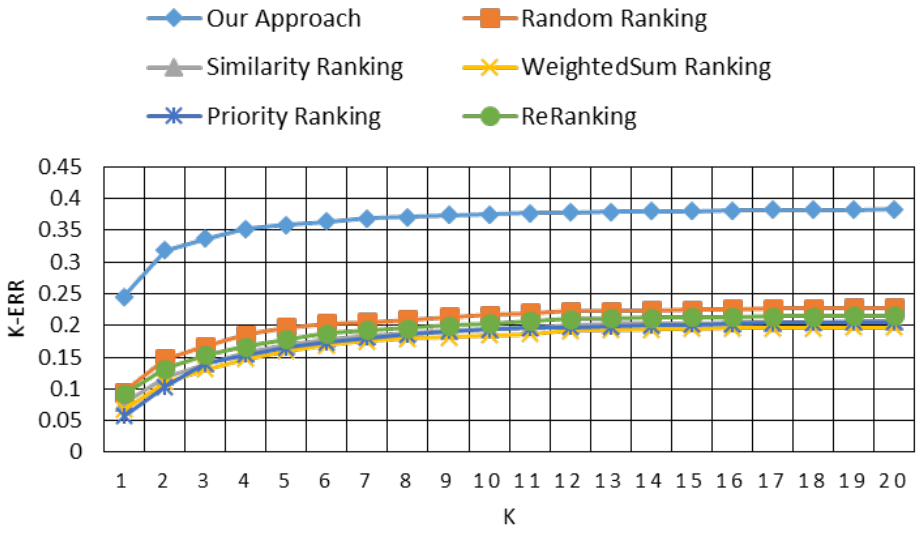

Figure 4.6: The k-ERR comparison result between our approach and the existing ranking schemas

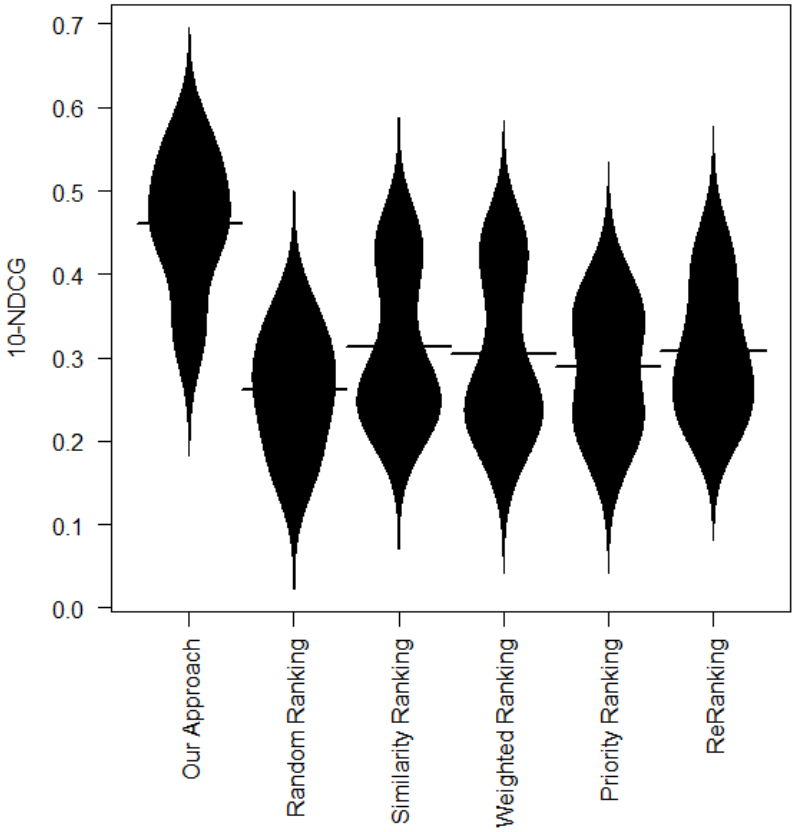

Figure 4.7: The result of performance evaluation study between our approach and the existing ranking schemas in terms of 10-NDCG

and 48.42% improvement than the existing ranking schemas for 10-NDCG and 10-ERR respectively. Applying Mann-Whitney U test and Cliff's delta shows that the improvement achieved by our approach is significant with large effect size.

Our learning-to-rank approach can outperform the existing ranking schemas in ranking code examples, with an average improvement of 35.65% and 48.42% in terms of 10-NDCG and 10-ERR respectively.

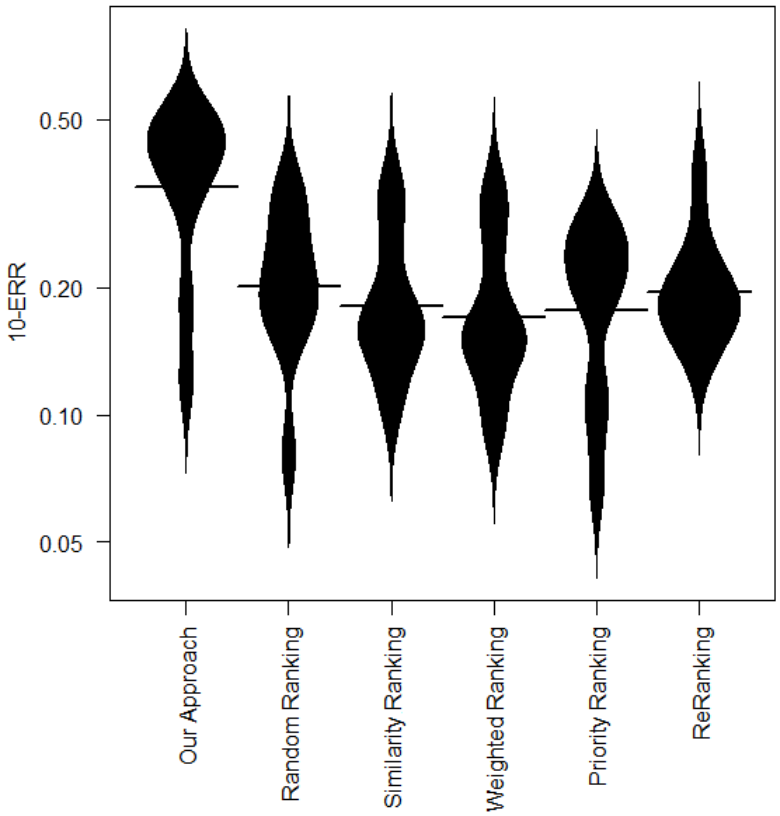

Figure 4.8: The result of performance evaluation study between our approach and the existing ranking schemas in terms of 10-ERR

Table 4.6: Summary of the improvement achieved by our approach comparing with the existing ranking schemas for code example search using 10-NDCG.

Ranking Schema	**10-NDCG**	**Improvement**	***p*-values**	**Cliff's Delta**	**Effect Size**
Our Approach	0.46	-	-	-	-
Random Ranking	0.26	43.48%	1.30e-04	0.92	large
Similarity Ranking	0.31	32.61%	2.09e-03	0.78	large
Weighted Ranking	0.31	32.61%	2.09e-03	0.78	large
Priority Ranking	0.29	36.96%	1.05e-03	0.82	large
ReRanking	0.31	32.61%	4.87e-04	0.86	large

Table 4.7: Summary of the improvement achieved by our approach comparing with the existing ranking schemas for code example search using 10-ERR.

Ranking Schema	**10-ERR**	**Improvement**	***p*-values**	**Cliff's Delta**	**Effect Size**
Our Approach	0.38	-	-	-	-
Random Ranking	0.22	42.11%	1.47e-02	0.64	large
Similarity Ranking	0.19	50%	3.89e-03	0.74	large
Weighted Ranking	0.18	52.63%	2.88e-03	0.76	large
Priority Ranking	0.19	50%	5.20e-03	0.72	large
ReRanking	0.2	47.37%	8.93e-03	0.68	large

4.4.2 RQ4.2: Are the studied features of code examples equally important to our approach?

Motivation. Our approach adopts 8 uncorrelated features used in earlier studies to learn how to rank candidate answers for code example search. We include the features since earlier studies reported an improvement on the ranking capability when the features are combined with the similarity features (*e.g.*, [32][72]). In this research question, we evaluate the impact of each feature on the performance of ranking code examples using our approach.

Approach. We consider the ranking schema built using our approach with 8 uncorrelated features, as the baseline ranking schema s^{base}. Then, we build alternative ranking schemas to analyze the impact of each feature on the ranking performance. We use the same approach as Breiman's study [9] to build alternative ranking schemas. We randomly order the values of one feature for the candidate code examples each time when building an alternative ranking schemas using our approach. The rationale is that the impact of one feature on the performance of the baseline approach can be

observed when the feature values are randomized. In total, we generate eight alternative ranking schemas. The first alternative ranking schema s^{alt}_{-sim} considers all of the features used in the baseline ranking schema s^{base} except for the textual similarity feature. Following the same naming convention, the other alternative ranking schemas are denoted by s^{alt}_{-ctx}, s^{alt}_{-fre}, s^{alt}_{-len}, s^{alt}_{-cmt}, s^{alt}_{-ran}, s^{alt}_{-pro}, and s^{alt}_{-ide} where they consider all the features except for context similarity, frequency, line length, comment to code ratio, page-rank, probability, and the number of identifiers respectively. Finally, we compare the performance of the baseline with the alternative ranking schemas to analyze the impact of different features on the performance of the baseline approach in terms of k-NDCG and k-ERR, with k ranging from 1 to 20.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data [63]. We conduct Mann-Whitney U test with 95% confidence level (*i.e.*, p-value < 0.05) to study whether there is a statistical difference between the performance of an alternative schema and the baseline. If the difference of the ranking performance between an alternative schema and the baseline is significant, we can conclude that the feature randomized when building the alternative schema is actually important to the success of ranking code examples using learning-to-rank approach.

Result. Fig. 4.9 and Fig. 4.10 summarize the result of the performance comparison study between the baseline ranking schema and the alternatives in terms of k-NDCG and k-ERR, with k ranging from 1 to 20. We can see from Fig. 4.9 and Fig. 4.10 that randomizing textual similarity, context similarity or frequency obviously reduces the performance of our approach from k-NDCG and k-ERR point of view. Similar to the earlier studies on code example search [52], we look into the comparison

result when k is set to 10. Fig. 4.11 and Fig. 4.12 show the 10-NDCG and 10-ERR comparison result between the baseline and the alternative ranking schemas. Table 4.8 lists the decrease in the performance and p-values when we compare the baseline schema with the alternative schemas using 10-NDCG and 10-ERR. We observe that randomizing textual similarity (s^{alt}_{-sim}) or frequency (s^{alt}_{-fre}) features decreases the performance in terms of 10-NDCG significantly. Randomizing textual similarity (s^{alt}_{-sim}) or context similarity (s^{alt}_{-ctx}) features decreases the performance in terms of 10-ERR significantly. However, randomizing the other features does not affect the performance significantly. Therefore, we can conclude that textual similarity, context similarity, and frequency are the three influential features for the ranking performance of our approach.

Discussion. As illustrated in Fig. 4.11 and Fig. 4.12, the alternative ranking schema s^{alt}_{-ide} performs better than the baseline ranking schema s^{base}. It indicates that the feature, the number of identifiers per line, constitutes no predictive influence on the code example ranking. To elaborate, assuming two code examples that are relevant to the same query have the same source code, and the statements in one of the code examples are broken down into multiple lines while the other does not.The total number of identifiers of the two code examples are the same since the content of the two code examples are the same. However, the lines of code of the two code examples are different. In other words, the two code examples are different in terms of the number of identifiers per line. However, the code examples should be ranked with the same relevance labels. Therefore, the number of identifiers per line, is not a positive feature to predict the code example ranking.

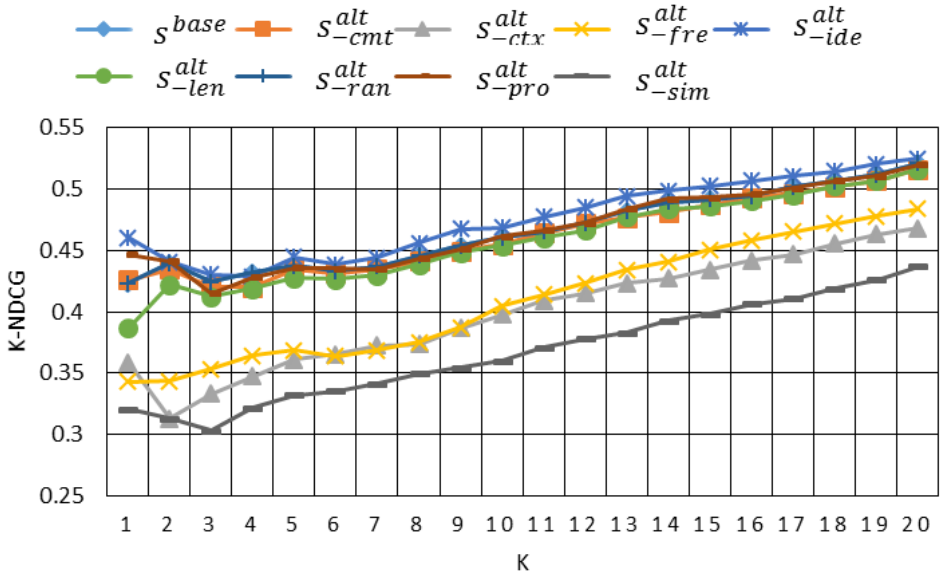

Figure 4.9: Studying the impact of features on the performance of our approach based on k-NDCG measure

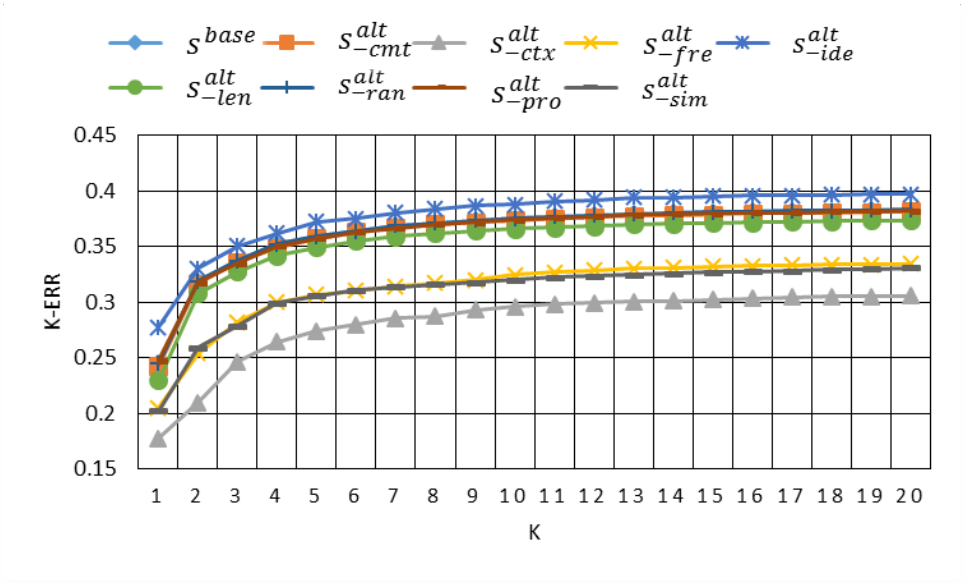

Figure 4.10: Studying the impact of features on the performance of our approach based on k-ERR measure

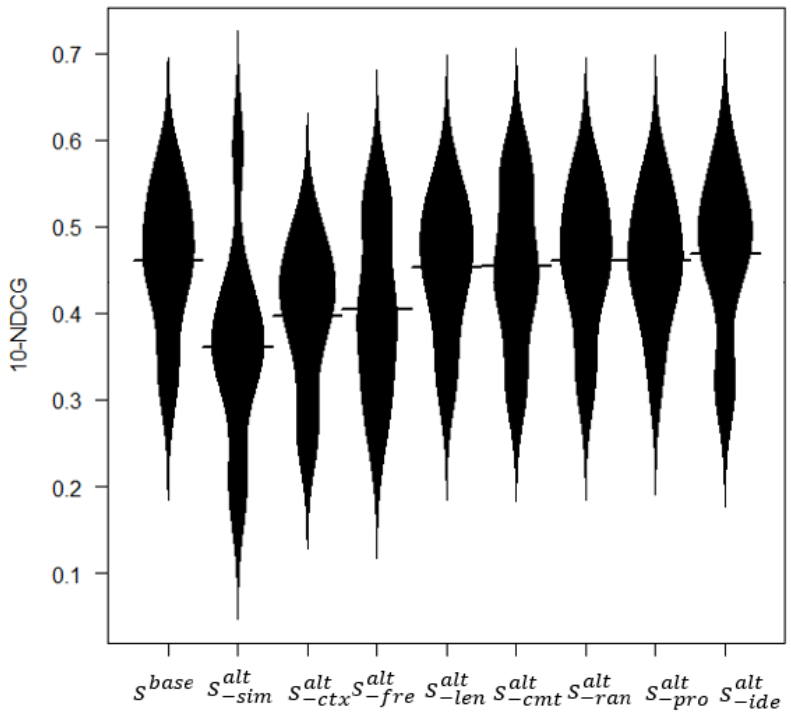

Figure 4.11: Performance comparison between baseline and alternative ranking schemas in terms of 10-NDCG measure

Textual similarity, frequency and context similarity are the three influential features for the ranking performance of the learning-to-rank approach in the context of code example search.

4.4.3 RQ4.3: Does our approach perform well in recommending effective code examples?

Motivation. Code examples can help developers learn how to use a specific API or class [42]. Successful code example search engine should place effective code examples at the top of ranked list to improve the user experience in searching for the desired code examples [56]. In this research question, we compare our approach with Codota in terms of recommending effective code examples to evaluate the usefulness of our

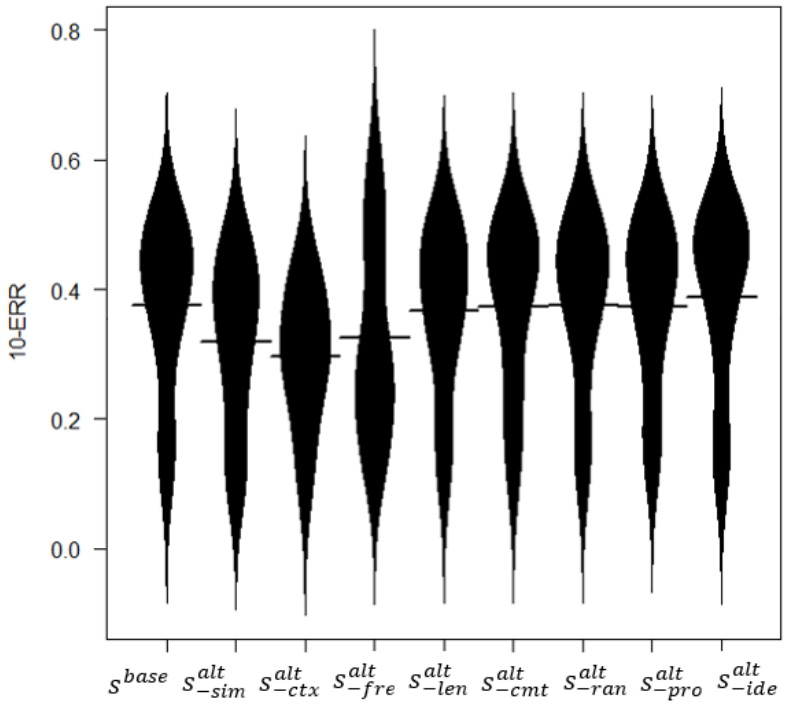

Figure 4.12: Performance comparison between baseline and alternative ranking schemas in terms of 10-ERR measure

approach.

Approach. Similar to the earlier studies on the comparison of Web search engines [1], we design a user study to identify which code search engine is more successful in providing effective code examples at the top of the ranked result set. There are five assessors participating in our user study by expressing the preference between the code examples recommended by our approach and Codota for 50 randomly selected queries. The five assessors are the same people who judge the relevances between the queries and the corresponding code examples in the labeling process. To avoid the issue of knowledge transfer, we assign an assessor with the queries he or she has never judged before in the labeling process. For each query, an assessor is provided with two lists of top 5 answers from our approach and Codota respectively. The two

Table 4.8: Summary of ranking performance for identifying the most influential features

Ranking schemas	**10-NDCG**			**10-ERR**		
	Avg.	Impact	p-value	Avg.	Impact	p-value
s^{base} (all features)	0.46	-	-	0.38	-	-
s^{alt}_{-sim}(all features except for textual similarity)	0.36	-21.74%	**9.77e-03**	0.32	-15.79%	**1.37e-02**
s^{alt}_{-ctx}(all features except for context similarity)	0.4	-13.04%	6.45e-02	0.3	-21.05%	**5.86e-03**
s^{alt}_{-fre}(all features except for frequency)	0.4	-13.04%	**2.73e-02**	0.32	-15.79%	0.1055
s^{alt}_{-len}(all features except for line length)	0.45	-2.17%	0.2807	0.37	-2.63%	0.2807
s^{alt}_{-cmt}(all features except for comment to ratio)	0.45	-2.17%	0.6356	0.37	-2.63%	0.9397
s^{alt}_{ran}(all features except for page-rank)	0.46	-0.00%	1.00	0.38	-0.00%	1.00
s^{alt}_{-pro}(all features except for probability)	0.46	-0.00%	0.6356	0.37	-0.00%	0.6356
s^{alt}_{-ide}(all features except for identifier)	0.47	+2.17%	0.5566	0.39	+2.63%	0.6953

result sets are presented side by side anonymously [71]. Therefore, the assessors do not know which list belong to which engine. Also for each query, the location (*i.e.*, left or right side of the window) of the result for each engine is selected randomly as suggested by Bailey *et al.* [1]. Assessors have to express the preference between the two code examples at rank k where $1 \leq k \leq 5$, *i.e.*, 5 code example pairs for one query. Each assessor evaluates 50 code example pairs for 10 queries. In total 250 pairs of code examples are evaluated by five assessors for 50 randomly selected queries as described in Section 4.1.1. If one of the result sets has more preferred code examples than the other one, we refer to it as the winner result set. Finally, we report (1) the

average number of preferred code examples in the result set for each query by each engine, and (2) the number of winning result sets for each engine.

Mann-Whitney U test is a non-parametric test that does not hold assumption on the distribution of data [63]. We conduct Mann-Whitney U test to check if the difference in the numbers of the preferred code examples in the result sets recommended by our approach and Codota is significant. However, Mann-Whitney U test is not suitable to be used to compare the difference in the number of winning results since the recorded observations are boolean. We use Chi-Square test [33] to determine if there is a significant difference between the studied code search engines in terms of the number of winning result sets. We use the 95% confidence level (*i.e.*, p-value $<$ 0.05) to identify the significance of the comparison results.

Results. Fig. 4.13 shows the average number of the preferred answers in the result set recommended by our approach and Codota. We observe that out of five answers per query, our approach achieves 3 preferred code examples while 2 answers of Codota are preferred in the top 5 answers. The p-values for the comparison of preferred code examples between our approach and Codota is $1.72e-04$. In addition, our approach can recommend winning result sets for 36 out of 50 queries while that for Codota is 14. The comparison between Codota and our approach in terms of the number of winning result sets is significant, with the p-value of $1.813e-11$. Therefore, we can conclude that our approach performs well in recommending effective code examples for queries related to Android application development.

Our learning-to-rank approach performs well in recommending effective code examples for code search queries.

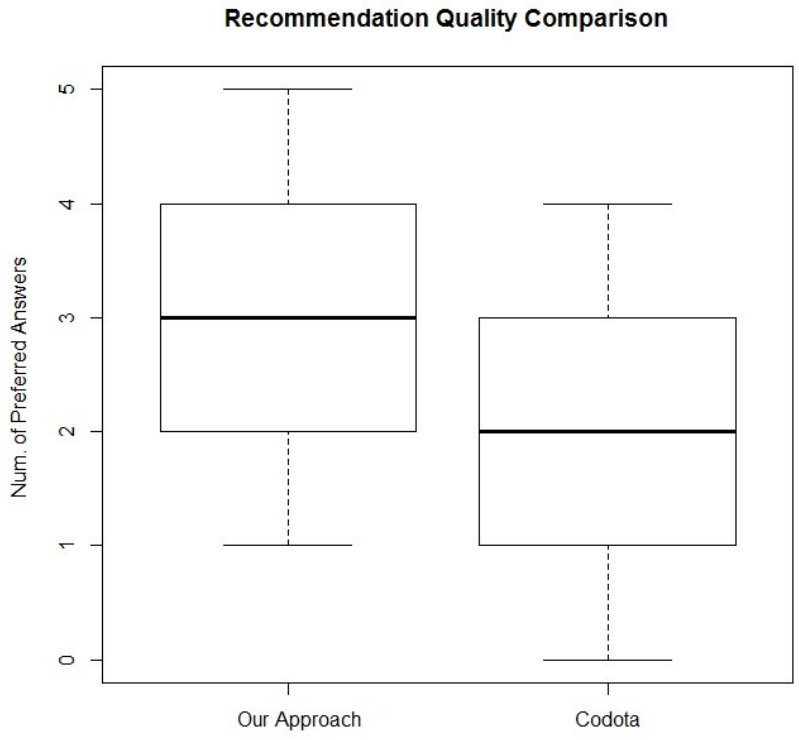

Figure 4.13: Comparison results between our learning-to-rank approach and Codota in terms of the number of preferred candidate code examples for one query

4.5 Threats to Validity

In this section, we discuss the threats to validity that might affect the results of our study following the common guidelines provided by Robert [58].

Threats to construct validity concern whether the setup and measurement in the study reflect real-world situations. In our study, the construct validity threats mainly come from the manual labeling of the relevances between queries and candidate code examples. The subjective knowledge about programming may affect the accuracy in judging on relevance. To reduce the subjectivity in relevance labeling,

the relevance between a query and each candidate code example are judged by three independent assessors, and we apply the majority rule if there is a disagreement. In addition, the candidate code examples are shown to the assessors query by query. It might be concerned that an assessor may just select one code example as relevant one and label the other code examples irrelevant. To alleviate the concern, we ask the the assessors to judge each code example based on the description and the relevance label specification for a specific query. The concern can be further reduced by comparing the relevance labels from three independent assessors.

Threats to internal validity concern the uncontrolled factors that may affect the experiment result. In our experiment, the main threats to internal validity come from feature extraction from code snippets. We use Eclipse JDT AST parser to parse the source code of Android projects and extract code snippets. For each code snippet, we compute the values of its features using the definition described in Section 4.2.2.

Threats to conclusion validity concern the relation between the treatment and the outcome. We conducted 10-fold cross validation as sensitivity study. When computing the significance of comparison results, we have used non-parametric test that does not assume the distribution of the data. The difference in the code corpus used by our approach and Codota may affect the results of RQ4.3. Our code corpus is created by extracting code snippets from open-source Android projects in GoogleCode. Codota also extracts code snippets from GoogleCode. And our corpus is large-scale, which can reduces the threats to the results of RQ4.3.

Threats to external validity concern whether our experimental results can be generalized for other situations. The main threats to external validity in our study is the representativeness of our corpus and queries. Our code base contains more than

360,000 code snippets extracted from open-source Android projects in Google Code. And we extract 2,500 code examples relevant to 50 queries to create the training data. The size of the training data is sufficient to learn a ranking schema that performs better than other baseline ranking schemas. In practice, the training data can be obtained from the usage logs of code search engines which contains the searching queries and the selected(relevant) candidate code examples. Therefore, the trained ranking schema can perform better using more training data available in practice. As for the queries, the size of our query set (*i.e.*, 50) is comparable to the similar studies on learning-to-rank [56], and meets the minimum number of queries recommended for search engine evaluation [50].

Threats to reliability validity concern the possibility to replicate this study. All the necessary details that are needed to replicate our work are publicly available: https://www.dropbox.com/s/4gxs85dii1nms9t/Replication.rar?dl=0.

4.6 Summary

We have proposed a code search approach which applies the learning-to-rank technique to automatically learn ranking schemas for code example search. To build a set of training data, we identify 12 features of code examples and collect the relevances between queries and the corresponding candidate code examples. Then we train the ranking schema using a learning-to-rank algorithm. The learned ranking schema can be used to rank candidate code examples for new queries. Finally, we evaluate our approach using k-NDCG and k-ERR measures. The evaluation results show that our approach can effectively rank and recommend relevant code examples for developers.

Chapter 5

Conclusion

Code search engines are designed to support developers' programming tasks. However, we find existing code search engines accept mainly keyword-based code search queries, and ranks code examples using hand-crafted heuristics. In this thesis, we propose two approaches to improve the effectiveness of existing code search engines. One approach enables existing code search engines accommodate natural-language queries. The other approach can automatically learn a ranking schema for an existing code search engine. In this chapter, we first describe the contribution of the thesis. Then, we explore the directions of future work.

5.1 Contributions

The contributions of the thesis has been listed as follows:

- We propose an approach to enable existing code search engines which normally support keyword-based queries accommodate natural-language queries. The approach reformulate natural-language code search queries using the most semantically related keywords. We evaluate the approach using a large-scale

corpus of code examples. The evaluation results show that the proposed approach can significantly improve the effectiveness of existing code search engines for natural-language code search queries.

- We propose an approach that can improve the ranking capability of existing code search engines. The approach uses the learning-to-rank technique to automatically learn a ranking schema for a code search engine. We evaluate our approach using the training and testing datasets consisting of 25,00 code examples related to 50 queries. The evaluation results show that the proposed approach can outperform existing ranking schemas in ranking candidate code examples.

5.2 Future Work

In the future, we want to evaluate the performance of proposed approaches using more queries selected from the usage logs of existing code search engines, and study the application of the approaches for recommending other types of software artifact, *e.g.*, components or libraries, for pragmatic reuse. In addition, we plan to combine our approaches with a code search engine so that a ranking schema can be automatically built for the search engine which can effectively recommend code examples for code search queries, including natural-language and keyword-based queries.

Bibliography

[1] P. Bailey, P. Thomas, and D. Hawking. Does brandname influence percerived search result quality? yahoo!, google, and webkumara. In *Proceedings of ADCS*, 2007.

[2] S. K. Bajracharya and C. V. Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, 17(4-5):424–466, 2012.

[3] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[4] S. K. Bajracharya, J. Ossher, and C. V. Lopes. Leveraging usages similarity for effective retrieval of examples in code repositories. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[5] V. R. Basili, L. C. Briand, and W. L. Melo. How reuse influences productivity in object-oriented systems. In *Communications of the ACM*, pages 104–116, 1996.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[7] D. Binkley and D. Lawrie. Learning to rank improves ir in se. In *Proceedings of 2014 IEEE International Conference on Software Maintenance and Evolution*, 2014.

[8] J. Brandt, P. Guo, J. Lewenstein, M. Dontcheva, and S. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598, 2009.

[9] L. Breiman. Random forests. pages 5–32, 2001.

[10] S. Brin and L.Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of 7th International World-Wide Web Conference*, 1998.

[11] P. F. Brown, V. J. D. Pietra, P. V. deSouza, J. C. Lai, and R. L Mercer. Class-based n-gram models of natural language. *Computational Linguistics*, 18:467–479, 1992.

[12] M. Bruch and T. Schfer. On evaluating recommender systems for api usages. In *2008 international workshop on Recommendation systems for software engineering*, 2008.

[13] M. Bruch and T. Schfer. On evaluating recommender systems for api usages. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 16–20, 2008.

[14] M. G. Bulmer. Principles of statistics. 1967.

[15] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

[16] R. P. L. Buse and W. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36:546–558, 2010.

[17] R. P. L. Buse and W. Weimer. Synthesizing api usage examples. In *34th International Conference on Software Engineering*, 2012.

[18] M.J. Campbell and T. D. V. Swinscow. Statistics at square one. 2009.

[19] Z. Cao, T. Qin, T. Y. Liu, M. F. Tsai, and H. Li. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*, pages 129–136, 2007.

[20] C. Carpineto and G. Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys*, 44:1–56, 2012.

[21] O. Chapelle, D. Metzler, Y. Zhang, and P. Grinspan. Expected reciprocal rank for graded relevance. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 621–630, 2009.

[22] S. Chatterjee, S. Juvekar, and K. Sen. Sniff: A search engine for java using free-form queries. In *12th International Conference on Fundamental Approaches to Software Engineering*, pages 385–400, 2009.

[23] N. Cliff. Dominance statistics: Ordinal analysis to answer ordinal questions. 1993.

[24] R. Cox. Regular expression matching with a trigram index or how google code search worked. In *Online resource: https://swtch.com/ rsc/regexp/regexp4.html*, 2012.

[25] K. Crammer and Y. Singer. Pranking with ranking. In *Advances in Neural Information Processing Systems 14*, page 641647, 2001.

[26] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26:297–302, 2012.

[27] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. In *The Journal of Machine Learning Research*, pages 933–969, 2003.

[28] R. E. Gallardo-Valencia and S. Elliott Sim. Internet-scale code search. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE '09)*, pages 49–52, 2009.

[29] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. On the use of relevance feedback in ir-based concept location. In *International Conference on Software Maintenance*, pages 997–1016, 2009.

[30] S. Gottipati, D. Lo, and J. Jiang. Finding relevant answers in software forums. In *26th International Conference on Automated Software Engineering*, 2011.

[31] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *IEEE ICDM Worshop on Frequent Itemset Mining Implementations*, 2003.

[32] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38:1069–1087, 2012.

[33] P. E. Greenwood and M. S. Nikulin. A guide to chi-squared testing. 1996.

[34] S. haiduc, G. bavota, A. Marcus, R. Oliveto, A. D. Lucia, and T. menzies. Automatic query reformulations for textual retireval in software engineeing. In *International Conference on Software Engineering*, pages 842–851, 2013.

[35] P. Harrington. Machine learning in action. 2012.

[36] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE Transactions on Software Engineering*, 32:4–19, 2006.

[37] R. Herbrich, T. Graepel, and K. Obermayer. Large margin rank boundaries for ordinal regression. In *Advances in Large Margin Classifiers*, pages 115–132. MIT Press, 2000.

[38] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *27th International Conference on Software Engineering*, pages 117–125, 2005.

[39] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. 21, 2012.

[40] P. Jaccard. tude comparative de la distribution florale dans une portion des alpes et des jura. In *Bulletin de la Socit Vaudoise des Sciences Naturelles 37*, page 547579, 1901.

[41] K. Jarvelin and J. Kekalainen. Cumulated gain-based evaluation of ir techniques. In *ACM Transactions on Information Systems*, pages 422–446, 2002.

[42] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675, 2014.

[43] J. Kim, S. Lee, S. Hwang, and S. Kim. Towards an intelligent code search engine. In *24th AAAI Conference on Artificial Intelligence*, 2010.

[44] A. Kuhn, S. Ducasse, and T. Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49:230–243, 2007.

[45] H. Li. A short introduction to learning to rank. 94:18541862, 2011.

[46] P. Li, C. Burges, and Q. Wu. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Proceedings of the 21st Annual Conference on Neural Information Processing Systems*, pages 897–904, 2008.

[47] M. Linares-Vsquez, G. Bavota, C. Bernal-Crdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A thread to the success of android apps. In *9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 477–487, 2013.

[48] T. Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 38:225–331, 2009.

[49] S. Thummalapenta M. R. Marri and T. Xie. Improving software quality via code searching and mining. In *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE '09)*, pages 33–36, 2009.

[50] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to information retrieval*. 2008.

[51] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Working Conference on Reverse Engineering*, pages 214–223, 2004.

[52] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu. Portfolio: Searching for relevant functions and their usages in millions of lines of code. 22, 2013.

[53] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Workshop at ICLR*, 2013.

[54] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 997–1016, 2012.

[55] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of 27th international conference on Software engineering*, pages 284–292, 2005.

[56] S. Niu, J. Guo, Y. Lan, and X. Cheng. Top-k learning to rank: labeling, ranking and evaluation. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 751–760, 2012.

[57] J. L. Plass, R. Moreno, and R. Brnken. *Cognitive Load Theory*. Cambridge University Press, 2010.

[58] K. Y. Robert. Design and methods. 2002.

[59] J. J. Rocchio. The smart retrieval system - experiments in automatic document processing. *Relevance feedback in information retrieval*, pages 313–323, 1971.

[60] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In *AIR Forum*, pages 1–33, 2006.

[61] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. In *Communications of the ACM*, pages 613–620, 1975.

[62] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. V. Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *6th international conference on Aspect-oriented software development*, page 221224, 2007.

[63] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman and Hall/CRC, 2007.

[64] E. Shihab, M. Zhen, W. M. Ibrahim, B. Adams, and A. E. Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010.

[65] J. Singer. Practices of software maintenance. In *International Conference on Software Maintenance*, pages 139–145, 1998.

[66] B. Sisman and A. C. Kak. Assisting code search with automatic query reformulation for bug localization. In *10th IEEE Working Conference on Mining Software Repositories*, pages 309–318, 2013.

[67] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying word relations in software: A comparative study of semantic similarity tools. In *International Conference on Program Comprehension*, 2008.

[68] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology*, 23, 2014.

[69] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers. Improving api documentation using api usage information. In *Visual Languages and Human-Centric Computing*, pages 119–126, 2009.

[70] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. Towards a big data curated benchmark of inter-project code clones. In *30th International Conference on Software Maintenance and Evolution*, 2014.

[71] P. Thomas and D. Hawking. Evaluation by comparing result sets in context. In *Proceedings of ACM International Conference on Information and Knowledge Management*, 2006.

[72] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, page 204213, 2007.

[73] VisionMobile. Developer tools: The foundations of the app economy. In *Developer Economics*, 2013.

[74] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Proceedings of the 10th Working Conference on Mining Sotware Repositories*, pages 319–328, 2013.

[75] J. Xu and H. Li. Adarank: a boosting algorithm for information retrieval. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, page 391398, 2007.

[76] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of 30th International Conference on Software Maintenance and Evolution*, 2014.

[77] J. Yang and L. Tan. Inferring semantically related words from software context. In *IEEE Working Conference on Mining Software Repositories (MSR)*, pages 161–170, 2012.

[78] X. Ye, R. Bunescu, and C. Liu. On the naturalness of software. In *Proceedings of IEEE International Conference on Software Engineering*, 2012.

[79] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

[80] A. T. T. Ying and M. P. Robillard. Selection and presentation practices for code example summarization. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2014.

[81] H. Zaragoza, N. Craswell, M. Taylor, S. Saria, and S. Robertson. Microsoft cambridge at trec-13: Web and hard tracks. 2004.

[82] H. Zhong, T. Xie, P. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of Euuropean Conference on Object-Oriented Programming*, pages 318–343, 2009.

[83] J. Zhou and H. Zhang. Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 852–861, 2012.